FPGA Implementation of a Compact Genetic Algorithm using Cellular Automata Pseudo-Random Generator

Marco A. Moreno-Armendáriz\textsuperscript{1}, Carlos A. Cruz-Villar\textsuperscript{2}, Victor H. Ponce-Ponce\textsuperscript{1}
\textsuperscript{1}Centro de Investigación en Computación, Instituto Politécnico Nacional, Av. Juan de Dios Bátiz s/n, México D.F., 07738, México.
email: mam-armendariz@cic.ipn.mx
\textsuperscript{2}Sección de Mecatrónica, CINVESTAV, Av. Instituto Politécnico Nacional 2508, Apdo. Postal 14 740, México D.F., México

Abstract—In this paper the design and implementation of the compact Genetic Algorithm (cGA) and a Cellular Automata-based pseudo-random number generator on a Field Programmable Gate Arrays (FPGA) is accomplished. The design is made using the Hardware Description Language, called VHDL. Accordingly, the obtained results show that it is viable to have this searching algorithm in Hardware to be used in real time applications.

Keywords: Compact Genetic Algorithm; Cellular Automata; FPGA design.

1. Introduction

Genetic Algorithms (GA) are very well known optimization techniques originally proposed in [1]. GA have demonstrated to be successful in the solution of complex numerical and combinatorial optimization problems, for single and multiple objectives [2] simulating natural evolution over populations of candidate solutions. GA handle a set of potential solutions instead of only one, but GA must evaluate the objective function several times, thus one of their main disadvantages is the high computing time required to solve complex problems. Some strategies have been proposed to deal with this drawback; for example, parallel implementations, efficient operators design, and hardware implementations, among others.

On the other hand, Compact Genetic Algorithms (cGA), are a kind of probabilistic model-building genetic algorithms or Estimation Distribution Algorithms (EDA) [3]. cGA operates on probability vectors by replacing the variation operators (crossover and mutation) that describes the distribution of a hypothetical population of solutions. It is known that the cGA obtains solutions of the same quality as the simple GA with binary representation and uniform crossover but with the advantage of an important reduction in the memory requirements, i.e. it only needs to store the probability vector (instead of the entire population). [4]. Therefore, the cGA may be useful in memory-constrained applications such as evolvable hardware [5].

In this paper, a novel and efficient design of a cGA in a hardware platform is shown. This design presents the following features: modularity, concurrency, minimal resource consumption, real time execution, and high scalability properties.

Nowadays there exists different computing machines to implement parallelism, nevertheless in this study we looking for a portable and autonomous evolvable hardware to be used in real-time tasks [22], therefore we select a FPGA to develop our design.

Also, it is important to mention that the population sizes in GA are problem-dependent, however in most of the cases they are from hundreds to thousands of individuals. Some exceptions are called “micro Evolutionary Algorithms” which have especially small population sizes, i.e. from 3 to 5, but in most of the cases they need additional mechanism to maintain the diversity.

The Genetic Algorithms are inherently parallel mainly due to their population-based nature. Thus, each individual of the population could be executed in a parallel manner. However, the compact Genetic Algorithm (cGA) has not a population, instead it only has a Probability Vector (PV), then, and its parallelization is slightly different.

2. Related Work

In this section we will present some of the most representative works including those related to Evolvable Hardware as well as those studies related to the compact Genetic Algorithms.

The term Evolvable Hardware is a research area that includes both, the design and implementation of Evolutionary Algorithms into hardware platforms to execute specific tasks; as well as the usage of Evolutionary Algorithms to generate hardware designs.

We will refer only to the former, that is, those works where the authors designed and implemented Evolutionary Algorithms into hardware. More than that, we will
refer only to the Compact Genetic Algorithms (cGA) (i.e. excluding other kind of Evolutionary Algorithms).

In 2001 Aporntewan and Chongstitvatana [10] proposed a cGA implementation in a FPGA using the language Verilog (Hardware Description Language). The authors showed that their design runs 1000 times faster than its software version executed in a workstation. The design is composed of five modules: random number generator, probability register, comparator, buffer, and fitness function evaluator. It is based on three basic operations: addition, subtraction, and comparisons. The probability vector updating is executed in parallel. It was tested on the Max-One problem with 32 bits and implemented in a Xilinx FPGA 23.57 MHz using 15,210 gates and a population size equal to 256. This cGA executes one generation per three clock cycles for the Max-One problem. For other more complicated problems, one generation would take $3+\varepsilon$ clocks, where $\varepsilon$ is the number of clocks used to evaluate the fitness function.

In [11], [9], the authors argued the cGA is very weak to optimize problems interesting for Evolvable Hardware’s researchers. So, to overcome this issue, they proposed some modifications to the cGA by incorporating elitism, mutation and resampling. They demonstrated it increases cGA exploration capabilities without increasing the hardware complexity. It was implemented with VHDL in a XC400 BORG and Virtex xc2v1000 whose architecture is composed by the following five modules: Random number generator, registers for the vector and mutation probabilities, buffer (2-ports RAM storing the individuals), modules INC/DEC that update each probability vector element and, a register indicating which is the winning individual. It was tested using a Max-One problem with 32-bits and 255 individuals. Their results were compared against the ones in [10], reporting similar results with a very small increment of the consumed resources. Furthermore, the algorithm was implemented on Software and tested on the De Jong functions [12]. Additionally, the authors conducted experiments on some dynamic optimization problems.

In [13] a Cellular Compact Genetic Algorithm implemented in a FPGA was proposed. It consists of a set of identical cGA. Each is called a cell and interacts only with its four neighbours. Each cGA cell exchanges probability vectors with its neighbours in an asynchronous schema. The probability vectors are combined by using an equation proposed by the authors. They argued the Cellular cGA parallelization is straightforward and suited to be implemented in a FPGA. They experimented with the Max-One problem and two numerical optimization problems demonstrating that their proposal is better than the simple cGA.

Other interesting works were published in [5], [14], [15] where the authors implemented different versions of the cGA. They developed a real-valued version of the cGA obtaining solutions with the same quality of those found by binary cGA with reduced computational costs. Furthermore, they implemented the non-persistent elitist cGA in the same micro-controller used to implement the cascade control of an induction motor drive to self-tune its velocity and position. The cGA is executed online and the motor’s sensors allow evaluating the fitness function. This work outperforms alternative schemata obtained with linear design techniques.

In [16] a novel architecture for a massive parallelization of the cGA is proposed. This schema presents three main advantages: low synchronization costs, fault tolerance, and high scalability. This architecture was simulated and tested in several processors (in software) obtaining an almost linear speed-up with a growing number of processors.

Only few publications about the analysis of the cGA can be found in specialized literature. Among them we can mention the following: The work published in [17] presents a rigorous runtime analysis of the cGA for pseudo-boolean functions on $n$ variables, and proves that not all linear functions have the same asymptotic runtime. In [18] the authors modelled the cGA as a Markov process and approximated it by an ordinary differential equation with a small learning step. The differential equation was studied to determine its convergence and stability properties.

As a applications of GA algorithms, in [6] a methodology based on a genetic algorithm (GA) to automate the design of combinational logic circuits in which the aim is to minimize the total number of gates used is presented. In [7], a GA is used for evolving and testing new rules for intrusion detection, and finally in [8] the major problem of the computational requirements for optimizing the place and route operations of a VLSI circuit is studied, the authors investigates the feasibility of using reconfigurable computing platforms to improve the performance of CAD optimization algorithms for this problem.

3. Preliminaries

The Genetic Algorithms (GA) presented in [1] make use of a population of 100 or more individuals to search for a solution, eventually one of them is selected as final solution. For the case of the cGA (see Algorithm 1) the individuals of the population are not used directly, in reality are represented through the probability vector $PV$, then two possible solutions ($ind_1$ and $ind_2$) are generated to find the final solution, which remains in $PV$. 

3.1 The Compact Genetic Algorithm

The compact Genetic Algorithm (cGA), originally proposed in [4], is the simplest algorithm from the Estimation of Distribution Algorithms family [3] whose main purpose is a simplification of the Genetic Algorithm. The cGA generates two possible solutions through an estimated probabilistic model of the original population instead of using traditional recombination and mutation operations and it is operationally equivalent to the order-one behaviour of the simple GA with uniform crossover. The length \( l \) of the PV must be able to represent the length that has the desired solution.

The values of PV can be represented as real values \( p_i \in [0,1], \forall i = 1, ..., l \) where \( l \) is the chromosome's length (binary string). It measures the proportion of '1' alleles in the \( i \)-th locus of the simulated population [19]. The cGA initializes the PV with 0.5 each of its elements. Two strings are generated using these probabilities. The fitness values for each generated individual are computed, then the PV is updated based on these strings. This process is repeated until the PV converges (to 0 or 1).

The overall of the algorithm is as follows: The cGA initializes the PV to 0.5, that is \( p_i = 0.5, \forall i = 1, ..., l \). Next, the strings \( ind_1 \) and \( ind_2 \) are generated according to the probabilities in PV. The fitness values of the strings \( ind_1 \) and \( ind_2 \) are compared and, the string with better fitness is named winner and the other is called loser. Then, if (winner[\( i \]) \neq loser[\( i \])], the PV[\( i \]) will be updated as follows: if (winner[\( i \]) = 1) then PV[\( i \)] will be increased by \( 1/n \), otherwise, PV[\( i \)] will be decreased by \( 1/n \). \( n \) is the population size that the cGA is emulating. Note that if (winner[\( i \]) = loser[\( i \])], the PV[\( i \)] will not be updated.

The loop is repeated until each PV[\( i \)] becomes zero or one. Finally, PV represents the final solution. The pseudo-code of the cGA is shown in Algorithm 1. It is assumed a maximization problem.

4. Hardware Design

The proposed design accomplishes the following features:

- Modularity and Scalability: The complete design of the cGA is divided into components that perform a specific task. So that, it can be reused for future developments.
- Concurrency: Due to the nature of the algorithm, there are some parts that can be implemented in parallel, for example the updating of the probabilities vector and the individuals generation.
- Minimum resource consumption: In the proposal we explore different techniques for the generation of pseudo-random numbers.

- Real-time performance: Thanks to the characteristics of the selected technology it is possible to use the design for real time applications.

The complete design is conformed of five components, and the manner to joint them by using a Finite State Machine.

4.1 Hardware Components for the cGA

Considering that these designs (cGA and EcGA) will be tested with 32-bits functions (see subsection 5.1), the following components are customized to accomplish this case. Nevertheless, these could be modified to work with any size of this kind of functions. Considering this, the new components are explained next:

1) Pseudo-Random Number Generator (PRNG): This component generates the random numbers which are essential for the creation of the individuals. Any known random generator algorithm can be used. However, others will strongly impact on the overall design. We considered implementing the Cellular Automata Based Generator, explained next.
• **Cellular Automata based Pseudo-Random Number Generator (CA-PRNG):** The idea of using Cellular Automata as generators of pseudo-random numbers was proposed in [20], where the authors used an one-dimensional cellular automata with \( n \) cells. The automata’s states are '0' and '1'. In [23] a set of CA-PRNG with lengths from 4-bits to 53-bits was proposed. It used the transition rules known as Rule 90 and Rule 150 (so named according to Wolfram [20]). Rule 90 is \( a_i(t+1) = a_{i-1}(t) \oplus a_{i+1}(t) \) and the Rule 150 is \( a_i(t+1) = a_{i-1}(t) \oplus a_i(t) \oplus a_{i+1}(t) \), where \( a_i(t+1) \) is the next state of the cell \( i \), \( a_{i-1}(t) \) is the present state of the cell \( i - 1 \) (left neighbour), \( a_i(t) \) is the present state of cell \( i \) (current cell), and \( a_{i+1}(t) \) is the current state of cell \( i + 1 \) (neighbour on the right). For example, Rule 0101 means cell '1' will use Rule 90, cell '2' will use Rule 150, cell '3' Rule 90 and cell '4' Rule 150. So, where there is a 'zero' Rule 90 is used, and where there is a 'one' Rule 150 applies. Cell memory (which stores the automata state) can be implemented with a synchronous flip-flop type D, and each state of cell \( i \) stores float numbers into \([0, 1]\) that must be increased/decreased with steps \( 1/n \) (\( n \) is the population size, see Lines 21 and 23 in Algorithm 1). On the other hand, the \( PV \) elements must be compared against the random numbers. Hence, the length of the binary chains representing each element of \( PV \) are the same length as the random numbers. In the case of CA-PRNG, it is necessary to utilize a substring of 6-bits to emulate a population with \( n = 50 \) individuals because it must store \( n + 1 \) possible numbers \( (0/n, 1/n, ..., n/n) \). For the MS-PRNG, the length of the \( PV \) was selected in 32-bits, however another size may be used.

3) **Fitness Evaluator (FEv):** This component is used to evaluate the fitness function. Its design depends on the objective function to be optimized. This block can be optimized according to the designer’s ability to implement complex functions of segmented or parallel mode. The input vectors are the two strings \( Ind1, Ind2 \), and \( RES \) as an output signal, it generates a '1' if the ability of the individual \( Ind1 \) is greater than or equal to that of the individual \( Ind2 \); or '0' otherwise.

4) **Probability Vector Updater (PVU):** The task performed by this component is adding or subtracting a certain amount \((1/n)\) to each element of the \( PV \) (see Lines 21 and 23 in Algorithm 1). If the 6-bit substring of CA-PRNG is utilized, then the \( PV \) elements are increased/decreased by one unit. For the case of 32-bit MS-PRNG the step is equal to 42949672.

To update the \( PV \) elements, three signals are received: (1) the winning individual (op), (2) the
bit from individual one of the \( i - \text{th} \) position (\( \text{ind}1[i] \)), and (3) the bit from individual two of the \( i - \text{th} \) position (\( \text{ind}2[i] \)). The necessary condition to update the \( i - \text{th} \) element of \( PV \) is that \( \text{ind}1[i] \) and \( \text{ind}2[i] \) are different. If this is the case, then an addition or subtraction is applied to that element. This updating process is computed in parallel\(^1\).

5) **Probability Vector Checker (PVC):** This component verifies if all the elements of the PV vector are '1' or '0'. When this condition is reached, the register \( RDY_{-PV} \) is equal to one, indicating that the cGA has finished.

### 4.2 Connecting the components

The complete design of the proposed architecture is illustrated in Figures 1, 2, 3.

![Fig. 1: One bit element of the cGA proposed design.](image1.png)

After that, the two bits are used as input to FEv. Then, FEv receives all the pair of bits, that make up two complete individuals, and to determine which of them is the winner. With this information, the component \( PVU[i] \) updates its corresponding \( PV[i] \) value. Then, the component PVC\([i]\) verifies the \( PV[i] \) convergence. In that case, the whole PV is given as the algorithm output.

In order to synchronize all the components, a Finite State Machine (FSM) is used [13]. The usage of this machine will guarantee the correct behaviour of the whole design. The FMS is defined by two functions: the first calculates the next state of the system, and the second, the output. A Mealy Machine is applied, which uses a clock as a synchronizing signal for transitions. The FMS is shown in Figure 4.

![Fig. 4: Finite State Machine (FSM) of the proposed design.](image4.png)

### 5. Experimental results

#### 5.1 Test functions

To evaluate the performance of the cGA in hardware, we use five types of test functions: Max-One, Min-One, Hamming-1431655765, Hamming-858993459 and Hamming-477218588. The so-called Hamming experiments consist of the minimization of the Hamming distance to the corresponding binary chain; for example,
the Hamming-1431655765 indicates minimizing the distance to the chain '01010101010101010101010101010101' represents the decimal number 1431655765.

Hardware

<table>
<thead>
<tr>
<th>Total logic elements</th>
<th>1,791 out of 68,416 ≈ 3%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total combinational functions</td>
<td>1,790 out of 68,416 ≈ 3%</td>
</tr>
<tr>
<td>Dedicated logic registers</td>
<td>754 out of 68,416 ≈ 1%</td>
</tr>
<tr>
<td>Total registers</td>
<td>754</td>
</tr>
<tr>
<td>Embedded Multiplier (9-bits)</td>
<td>0 out of 300 = 0%</td>
</tr>
</tbody>
</table>

Software

| cGA programmed in C++, executing in AMD Athlon 64 X2 Dual Core 4000+ at 2.09 GHz with 1 GB RAM and Ubuntu 8.04 |

Table 1: Experimental conditions

<table>
<thead>
<tr>
<th>HW</th>
<th>SW</th>
</tr>
</thead>
<tbody>
<tr>
<td>cGA</td>
<td>cGA</td>
</tr>
</tbody>
</table>

Max-One

| Iterations | 694 | 672 |
| Execution time | 82.41 µs | 4090 µs |

Min-One

| Iterations | 849 | 833 |
| Execution time | 101.4 µs | 4830 µs |

ReverseHamming-1431655765

| Iterations | 657 | 628 |
| Execution time | 74.22 µs | 3860 µs |

ReverseHamming-858993459

| Iterations | 712 | 695 |
| Execution time | 84.33 µs | 4270 µs |

ReverseHamming-477218588

| Iterations | 691 | 686 |
| Execution time | 78.6 µs | 4180 µs |

Table 2: Comparison between the hardware and software

5.2 Discussion

Here some important issues about the Hardware implementation are discussed.

- It is obvious that the execution time depends on the hardware clock frequency. However, a manner to measure the performance independent of the hardware velocity, is obtaining the number of clock cycles consumed by one algorithm iteration. Thus, in this implementation, one algorithm iteration requires 6 clock cycles, no matter the clock frequency utilized. For this specific hardware platform, with a frequency of 50 MHz, the number of algorithm iterations per second is computed with

  \[ \text{Gen}_{\text{second}} = \frac{\text{Frequency(}} Hz}{\text{clock}_{\text{generation}}} \]  

Then, it can perform 8.33 millions of generation per second.

- The goals established in Section 4 were successfully achieved. The Modularity, Scalability and Concurrency were implemented since each bit of the individuals are generated in separate components. This allows increasing the size of the binary chain very easily. By the use of the CA-PRNG we reduce the consumption of the FPGA resources and at the same time adequate pseudo-random numbers are used in the cGA algorithms.

- The minimal resource consumption stands for all proposed hardware components without consider the resources required for the FEv component (this block can be optimized according to the designert’s ability to implement complex functions), for the tested FEvt’s in this paper only 3% of the FPGA resources are used. However, we carefully analyze all aspects of the hardware design of the remains components such as: data representation (fixed or floating point) and arithmetic representation. These aspect are very important in order to avoid use all of the resources on the FPGA. So, It is important to maintain a compact cGA design, because it could be attached to other components, such as a Neural Network-based controllers for a real application, resulting necessary to reach a high performance design that consumes as less FPGA’s resources as possible.

- Each PB element stores a value into the range [0,1] that is utilized as the probability to generate an 1 (or complementary a zero). To maintain the FPGA’s computations simple, each PV element is compared directly against the random number just generated. Thus, it seems convenient that both, the PV values and the generated random numbers, are into the same range.

On the other hand, the cGA implemented in this work uses binary representation, which means that each PV element uses binary representation; therefore it becomes necessary to apply a discretization of its values. Considering that the cGA algorithm updates the PV values by increments/decrements of \(1/n\), (where \(n\) is the size of the emulated population) it would be enough if the discretization contains the values \(0/n, 1/n, 2/n, ..., n/n\). In most of the cases, the population sizes \(n\) go from 50 to 100, therefore, each element of PV could be represented with 6 bits (with \(n \in [50,...,64]\) or 7 bits (with \(n \in [65,...,100]\)). The cGA and the CA-PRNG seem to be very suitable to work together, especially for their implementation on an FPGA device, mainly because they both use binary representation, which avoids further expensive computations.

- Finally, since the execution time (see Table 2) is in the order of microseconds a real-time implementa-
tion is very feasible, some successful published works are in [14], [15] and [22].

6. Conclusion and Future Work

While it is true that any known pseudo-random numbers generator would work, here we found that the MS-PRNG occupies almost all the resources with a relatively small binary chain. Thus, it could be useful only with very small problems. However, if the Cellular Automata-based PRNG (CA-PRNG) is applied, the design utilizes only 3% out of the FPGA resources (see Tables 1-2).

For the kind of problems whose objective function can be represented with binary numbers, our design is suitable to be used in real applications. So, if the complexity of the fitness function is bigger, then the designer needs to explore different ways to maintain a good balance between the velocity and the amount of FPGA resources to be used.

The previously published cGA hardware implementations [10], [5], [14], [15], [11], [9], [13] do not present a detailed description of their designs, i.e., they do not show the components structure (I/O ports) or the manner to synchronize them. Furthermore, the presented design utilizes 6 clock cycles per one iteration.

As a future work we plan to use these algorithms for tuning intelligent controllers, like the fuzzy visual control developed by our research group in [21]. This is a successful case where the implementation of the whole system in hardware gives an excellent performance in real time. However, one of the biggest problems is the fuzzy tuning. Therefore, we want to use our hardware version of the cGA algorithms as a self-tuning tool for these kinds of controllers. Another successful application was presented in [22].

7. Acknowledgements

The authors wish to thank to Instituto Politécnico Nacional (SIP-IPN grants 20131514 and 20130086, COFAA-IPN y PIFI-IPN), CINVESTAV and the Government of Mexico (SNI y CONACYT) for providing necessary support to carry out this research work.

References


