Optimizing the Compact Genetic Algorithm Design with Cellular Automata

Alejandro León-Javier[1], Marco A. Moreno-Armendáriz[1] and Nareli Cruz-Cortés[1],

[1] Centro de Investigación en Computación - IPN
Av. Juan de Dios Batíz s/n Unidad Profesional Adolfo López Mateos
Col. Nueva Industrial Vallejo, Mexico,D.F., 07738, Mexico.
ljavierb07@sagitario.cic.ipn.mx, {marco_moreno, nareli}@cic.ipn.mx

Abstract. The Compact Genetic Algorithms (cGAs) are searching methods used in different engineering applications by their simplicity and low consumption of resources when compared with other Evolutionary Algorithms. These characteristics make them very attractive to be implemented in hardware. In this paper we propose some manners of optimizing a cGA design in VHDL by using Cellular Automata (CAs) with the aim of implementing them on a FPGAs.

Keywords: Compact Genetic Algorithm, Cellular Automata, FPGA, VHDL.

1 Introduction

The Compact Genetic Algorithms (cGAs) are classified as Estimation of Distribution Algorithms (EDAs) or Probabilistic Model Building Genetic Algorithms (PMBGAs). They are searching methods that mimic the behavior of conventional Genetic Algorithms (GAs) with uniform crossover [1]. The cGAs replace the population with a probability vector, where each component of the vector is updated by shifting its value by the contribution of a single individual to the total frequency assuming a particular population size [2].

Due to their particular characteristics, the cGAs can be implemented in a hardware platform easier than other Evolutionary Algorithms. The cGAs offer notable advantages in resources consumption due to their intrinsic simplicity.

There exist some publications showing Evolutionary Algorithms implementations based on hardware platforms. For example, in [3] the authors proposed the design of a hardware-based architecture to perform a Genetic Algorithm on a FPGA, called FPGA-based Genetic Algorithm Kernel. In [4] was proposed a Genetic Algorithm based on a chip LSI that includes reconfigurable logic hardware, a memory maintaining the individuals and a training data memory in a 16-bits CPU core. In [5] a cGA design and its

Special Issue in Advances in Pattern Recognition
Research in Computing Science 44, 2009, pp. 137-144

implementation was presented by using VERILOG language on a Xilinx FPGA. Gallagher et al, proposed in [6] to add elitism, mutation, and resampling to a cGA. These features improved the quality of found solutions by the cGA. In addition to that, they proposed to use a modular design to implement this algorithm on micro-controllers using VHDL. In [7] the authors proposed a parallel cellular cGA topology which is implemented on a FPGA.

Despite the existence of diverse architectures for different Evolutionary Algorithms, the majority of them have some disadvantages, maybe the most remarkable is that they consume many hardware resources or computing time (although the computing time is less when compared against software versions, it is not enough for real-time applications). Then in this work, we propose a cost-reducing architecture of hardware resources while preserving a good velocity of convergence by using Random Number Generators based on Cellular Automata.

```
1) Initialize probability vector
     for i:=1 to l do PV[i]:=0.5;
2) Generate two individuals from the vector
     a:=generate(PV);
     b:=generate(PV);
3) Let them compete
     winner,loser:=compete(a,b);
4) Update the probability vector towards the better one
     for i:=1 to l do
        if winner[i]≠loser[i] then
            if winner[i]=1 then
               PV[i]:= PV[i]+1/n;
            else
               PV[i]:= PV[i]-1/n;
5) Check if the vector has converged
     for i:=1 to l do
        if PV[i]>0 and PV[i]<1 then
            return to step 2;
6) PV represents the final solution
Compact GA parameters:
n: population size.
l: chromosome length.
```

Fig. 1. Pseudocode of cGA [8].

2 The Compact Genetic Algorithm

The cGA processes a Probability Vector (*PV*) with length *l*, which is initialized to *(0.5, 0.5,... , 0.5)*. Next, the individuals *a* and *b* are generated according to the *PV*. Then,

the fitness value of these individuals is compared and, the individual with better fitness is named the *winner* and the other one is called the *loser*.
Now, if *winner[i]* $\neq$ *loser[i]*, then *PV[i]* will be updated as follows: if *winner[i]=1* then *PV[i]* will be increased by *1/n*, otherwise, *VP[i]* will be decreased by *1/n*. Note that if *winner[i]=loser[i]*, the *VP[i]* will not be updated. These statements are repeated until each *PV[i]* becomes zero or one. Finally, *PV* represents the final solution [5]. The pseudocode of the cGA is shown in Figure 1.

3 Design of a Compact Genetic Algorithm Based on Hardware

The proposed design will be implemented on a FPGA, therefore, we will use VHDL to create the cGA's modules. They will be able to be connected to process the *PV* and at the same time, to parallelize the existing modules. These modules are called components and they have been designed as follows.

Compact Genetic Algorithm (cGA). This is the main component. It is based on the other components. Here, we have the registers *PV0,PV1,PV2,..., PVk* which are bit arrays representing a unsigned integer data in a range [*0, 2^k-1*], being *k* the length of bit array. The *PV* will be an integer array because VHDL is not synthesize floating point data (unless one designs them) and we would need a Random Number Generator in floating point for individual's generation, making more complicated the design. Also it includes registers (*ind1* and *ind2*) to store the individuals generated according to *PV*. The registers *RN1_0, RN1_1,..., RN2_0, RN2_1,...* store the random numbers generated by *RNG* components (all *RNG* components work in parallel). The *cGA* component is conformed by the following components: Random Number Generator (*RNG*), Individuals Generator (*IndGen*), Fitness Evaluator (*FEv*), Probability Vector Updater (*PVU*) and Probability Vector Checker (*PVC*). All this components are synchronize by a Finite State Machine descrited later.

Random Number Generator (RNG). Having a function or component as Random Number Generator is essential for any heuristic algorithm. Although exist many ways of generating pseudo-random numbers, only few of them are feasible to be designed into hardware due to consumption of resources that they demand. One interesting manner of generating random numbers is by using Cellular Automata. The Cellular Automata are composed by memory cells and rules. The Cellular Automaton-based Pseudo-Random Number Generators (CAPRNG's) can be of any length (no necessarily 32 bits), that is, we could have sequences of 2^n-1 elements (being *n* the number of bits of CAPRNG) for which we only need a memory cell (a D-flip-flop synchronous with *set* and *clear*) and its corresponding state rule (a combinatorial function) for each bit. These generators emulate populations of 2^n-2 individuals (being *n* the number of bits of CAPRNG), this way we allow achieving a reduction of bits in the generator and probability vector.

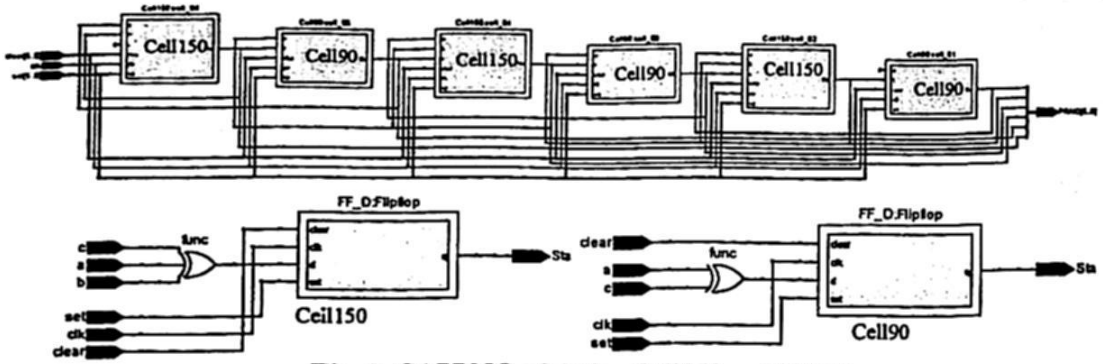


Fig. 2. CAPRNG of 6-bits, Cell150 and Cell90.

In Figure 2 we can notice that CAPRNG of 6 bits is composed of cells containing the rule 90 or 150 (Cell90 and Cell150 respectively). The rule 90 is: $a \oplus c$, and the rule 150 is: $a \oplus b \oplus c$. According to a selected cell, the pin *a* is connected with the pin *sta* of the left cell, the pin *b* is connected with the pin *sta* of the same cell, and pin *c* is connected with the pin *sta* of the right cell. In [9] it is shown how to design a CAPRNG's of 4 to 28 bits of length. For seeding the CAPRNG we should use the pins *set* and *clear* (for example, *set* array ='100101', *clear* array='011010') at the beginning and then, the pins *set* and *clear* will be cleared.

Individual Generator (IndGen). When this component is activated, it receives two random numbers and the *PV[i]* at a time. It generates one bit of each individual (*ind1[i]* and *ind2[i]*) in the i-th position at a clock cycle. The condition for generating one bit of a individual is: *if RN1≤PV[i] then ind1[i]='1', else ind1[i]='0'*, being *RN1* and *PV[i]* bit arrays of same length representing unsigned integers. All *IndGen* components contribute in generating two individuals in a clock cycle (working in parallel).

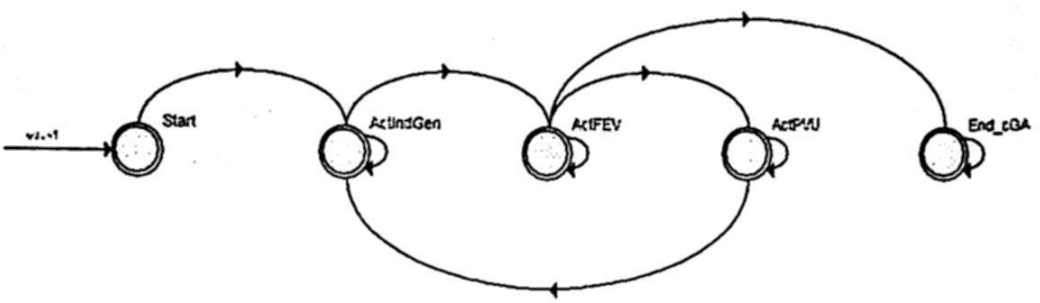
Fitness Evaluator (FEv). This component can be very variable because it contains the objective function, which can be very complex or very simple, depending on problem at hand. There exists the possibility that this component includes a Finite State Machines, where it accomplishes decoding and evaluation for floating point computing. The *FEv* component receives two bit arrays representing to individuals and it outputs a bit called *RES*. In general, first, we obtain both fitness values *fit1* and *fit2*, then we apply the next assignations: *if fit1≥fit2 then RES<='1', else RES<='0'*. where *RES* is a flag indicating which one of both individuals is the winner.

Probability Vector Updater (PVU). To update the *PV*, three data are received: the winner individual (*op*), the bit from the individual one on the i-th position (*ind1[i]*) and the bit from individual two on the i-th position (*ind2[i]*). The *oe* signal is the result of *ind[1] xor ind2[i]*. *PV[i]* is updated according to the next conditions:

```
if (oe='1' and ind1[i]='1') then
 if (op='1' and PV[i]<lim) then
   PV[i]<=PV[i]+1;
 elsif (op='0' and PV[1]>1)  then
   PV[i]<=PV[i]-1;
 end if;
elsif oe='1' and ind1[i]='0' then
 if (op='1' and PV[i]>1) then
   PV[i]<=PV[i]-1;
 elsif (op='0' and PV[i]<lim) then
   PV[i]<=PV[i]+1;
 end if;
end if;
```

lim is an integer value and it is equal to 2^n-*1* (being *n* the length of *PV[i]*). This limit is used to avoid overflow of *PV[i]*.

Probability Vector Checker (PVC). Due that each element of *PV* is an integer (or bit array), it is necessary to have other register (*SolPV*), where each element of *PV* be zero or one (final values). The *PVC* component updates each element of *SolPV* at each generation. This component activates a register called *RDY_PVC* when all elements of *PV* have converged. When *RDY_PVC=1*, the cGA has finished.



Transition conditions

	Current state	Next State	Rule
	Start	ActIndGen	
2	ActIndGen	ActIndGen	flag≠'0' and RDY_PVU≠"111"
	ActIndGen	ActFEV	flag='0' and RDY_PVU≠"111"
4	ActFEV	ActFEV	RDY_IndGen≠"111"
	ActFEV	ActPVU	RDY_IndGen="111" and RDY_PVC≠"111"
6	ActFEV	End_cGA	RDY_IndGen="111" and RDY_PVC="111"
	ActPVU	ActPVU	RDY_FEV≠'1'
8	ActPVU	ActIndGen	RDY_FEV='1'
	End_cGA	End_cGA	

Fig. 3. Finite State Machine (FSM) of the cGA component.

4 Synchronization by Finite State Machine

In order to synchronize the components previously proposed, we will use a Finite State Machine. A Finite State Machine is a sequential circuit with a finite number of states. The usage of this machine will guarantee the correct behavior of the whole design. The machine is defined by two functions, the first one calculates the next state of the system and the second one the output [10].

We use a Mealy Machine, which uses a clock as synchronizing signal for transitions. Our Finite State Machine is showed in Figure 3.

The FSM use some registers as *RDY_PVU*, indicating that *PVU* components have finished their execution if it is equal to *"1111..."*, otherwise, they have not finished. The same nomenclature is used for other components. Another register used is *ActIndGen*, which activate the *IndGen* component for generate the individuals. The same nomenclature is used for activate other components. All the transition conditions that synchronize the FSM are showed in Figure 3. The *RNG* is activated when *FEv* is activated. In *Start* state, the *RNG* component is seed.

5 Experiments and results

In our experiments, almost all registers of the cGA were sized with the same length (typically *PV* length). To evaluate our new design of cGA, we did some simulations on a Quartus II of Altera. For simulations, we use a Cyclone II EP2C70F896C6 with a clock of 50 MHz obtaining the results shown in Table 1. For the simulations we designed the cGA to solve the max-one problem, which consists on maximizing the number of ones of a bitstring, this implies a simple objective function, which is not costly. Table 1 shows a comparison between our design (cGA with CAPRNG) and an older version (cGA with RNG of 32 bits).

Table 1. Comparison of cGA designs for max-one problem.

Max-One		cGA with RNG of 32 bits	cGA with CAPRNG
8-bits	Runtime:	40.28μs	41.86μs
	Iterations:	335	348
	Combinational functions:	18796/68416	496/68416
	Dedicated logic registers:	826/68416	226/68416
	Embedded Multiplier 9-bits elements:	192/300	0/300
12-bits	Runtime:	41.48μs	43.06μs
	Iterations:	345	358
	Combinational functions:	28149/68416	703/68416
	Dedicated logic registers:	1214/68416	314/68416
	Embedded Multiplier 9-bits elements:	288/300	0/300

According to Table 1, although our design is ligthly slower for converge, the reduction of FPGA's resource consumption is impressive, while the cGA with RNG of 32 bits occupies approximately 10% of the FPGA's resources, our cGA with CAPRNG occupies about the 1% of FPGA's resources. This reduction of resource consumption is because we used a very simple *RNG* component. The Table 2 shows the differences of both components, demonstrating that CAPRNG is cheaper in the resource consumption of FPGA. Regarding to the number of iterations, our cGA executed more iterations because it emulates a population of 62 individuals (2^n-2), and the other design emulates a population of 50 individuals.

Table 2. Comparison of *RNG* components

Component		Data Synthesis
Random Number Generator based on Park & Miller Rules	Combinatorial functions:	1108/68416
	Dedicated logic registers:	826/68416
	Embedded Multiplier 9-bits elements:	192/300
Random Number Generator based on Cellular Automata	Combinatorial functions:	24/68416
	Dedicated logic registers:	6/68416
	Embedded Multiplier 9-bits elements:	0/300

6 Conclusions and Future Work

We obtained great benefits when substituted the RNG component based in Park and Miller Rules by the version based on Cellular Automata. We reduced about of 10 times the FPGA's resource consumption allowing a simpler and more parallel design (for RNG component case). This reduction of resource consumption will allow to implement another cGA variants in a better manner (for example, with elitism, mutation, etc.) or versions with more complex objective functions (because there will be more available resources for such purposes).

The convergence speed continues being attractive for applications that require to perform optimization in real time, although the objective function evaluation could consume more resources and computer time. According to the designer's abilities, it could be possible to parallelize, total or partially this section allowing a bigger convergence speed of the algorithm.

As a future work, we will conduct some experiments to design other cGA versions, as well as some simulations with objective functions capable of evaluating floating point values.

7 Acknowledgment

The authors thank the support of the Mexican Government (CONACYT, SNI, SIP-IPN, COFAA-IPN, and PIFI-IPN), also we appreciate the support of Altera Corporation and Víctor Maruri for the donation of DE2-70 kit and Quartus II academic licenses.

References

1. F. Cupertino, E. Mininno, E. Lino and D. Naso, "Optimization of Position Control of Induction Motors using Compact Genetic Algorithms", IECON 2006 - 32nd Annual Conference on IEEE Industrial Electronics, pp. 55-60, 2006.
2. M. Pelikan, D. Goldberg and F. G. Lobo, "A survey of optimization by building and using probabilistic models", Technical Report 99018, Illinois University, Illinois, USA, 1999.
3. X. Zhang, C. Shi and F. Hui. "FPGA-based Genetic Algorithm Kernel Design", ICES 2007, LNCS 4684, Springer-Verlag, 2007.
4. S. Scott and A. Seth, "HGA: A hardware-based genetic algorithm," in Proceedings ACM/SIGDA 3rd Int. Symposium. Field-Programmable Gate Arrays, pp. 53–59, 1995.
5. C. Apomtewan and P. Chongstitvatana, "A Hardware Implementation of the Compact Genetic Algorithm" in Proceedings 2001 IEEE Congress Evolutionary Computation, Seoul, Korea, Vol. 1, pp. 624–629, 2001.
6. J. C. Gallagher et al., "A Family of Compact Genetic Algorithms for Intrinsic Evolvable Hardware" in Proceedings 2004 IEEE Transactions on Evolutionary Computation, Vol.8, Issue 2,pp. 111–125, 2004
7. Y. Jewajinda and P. Chongstitvatana, "FPGA Implementation of a Cellular Compact Genetic Algorithm", NASA/ESA Conference on Adaptive Hardware and Systems, Vol. 22 June 2008, pp. 385 – 390, 2008.
8. G. Harik, F. G. Lobo, and D. E. Goldberg, "The compact genetic algorithm", IEEE Transactions on Evolutionary Computation, Vol. 3, Issue 4, pp. 287-297, Nov. 1999.
9. P. D. Hortensius, R. D. McLeod and H. C. Card, "Parallel Random Number Generation for VLSI Systems Using Cellular Automata", IEEE Transactions on Computers, Vol. 38, Issue 10, October 1989, pp. 1466-1473.
10. F. Pardo and J. A. Boluda, VHDL Lenguaje para síntesis y modelado de circuitos. Alfaomega, 2nd edition, 2004, México (In Spanish).
11. A. E. Eiben and J. E. Smith, Introduction to Evolutionary Computation, Springer, First Edition, 2003.
12. G. Harik, "Linkage Learning via Probabilistic Modeling in the ECGA", IlliGAL Technical Report 99010, Illinois University, Illinois, USA, 1999.