A Cellular Genetic Algorithm Architecture for FPGAs

Pedro Vieira dos Santos and José Carlos Alves
INESC Porto
Faculdade de Engenharia, Universidade do Porto
pedro.mvs@gmail.com, jca@fe.up.pt

Abstract

This paper proposes a new architecture of a cellular genetic algorithm (cGA) suitable for FPGA implementation. By spreading the algorithm solutions (population) into subpopulations accessed from different processing nodes, a scalable array of processing elements can be run in parallel. Each subpopulation is saved in a dual-port memory block (BRAM) so that two different processing elements can share the same information. Preliminary results of a simple GA implementation for the travelling salesman problem (TSP) have shown that the problem size allocated for the algorithm is mainly constrained by the available memory and not by the other logic resources. Simulations performed to evaluate the effectiveness of the cGA as an optimization procedure have shown that this cGA architecture does not degrade the quality of the final solution and the performance almost linearly increases with the number of processing nodes.

1. Introduction

The Genetic Algorithm (GA) is one of the most known techniques used to solve complex optimization problems. It is a population based metaheuristic inspired in the evolution of species, where principles of natural selection and genetic are applied to candidate solutions of a problem with the goal of generating better ones [1].

However, a GA is a very computational intensive algorithm, demanding a huge number of iterations to solve real world optimization problems with effectiveness. With the increase of the need for solving such optimization problems in embedded systems, it becomes more important new architectures for accelerating the execution time of a GA into dedicated hardware.

This paper presents a proposal for a scalable cellular genetic algorithm (cGA) architecture suitable for a field-programmable gate array (FPGA) implementation. The architecture makes use of the dual port memory blocks (BRAMs) available in such devices to distribute the population (solutions of the algorithm) and to share a subpopulation of the cGA between two different GA processors. The travelling salesman problem (TSP) was elected as a case study to evaluate the cGA. Given a list of cities and their relative distances, the objective of this optimization problem is to visit each city exactly once in the shortest possible tour. To represent a solution of the TSP, the path representation was used which is a list of unique numbers representing the order that the cities must be visited [2].

The remainder of this paper is organized as follows. Section 2 presents a review of GAs and parallel GAs as well as related work performed in the field of hardware implementations of these algorithms. The proposed cGA is explained in Section 3 together with software simulations to validate the effectiveness of the algorithm as an optimization procedure. In Section 4 it is discussed the maximum size of an instance of the TSP together with the size of the array of the cGA that can be implemented in a FPGA. Finally, the paper concludes with Section 5.

2. Related Work

2.1. The Genetic Algorithm

The algorithm keeps a pool of solutions, called "population", coded in a proper way so that they can be combined to generate new ones. A "selection" procedure chooses solutions (typically two) to interact with each other based on genetic inspired operators like "crossover" and "mutation" to produce new solutions that explore new areas of the search space. To measure the quality of a solution an objective function quantifies its "fitness", so that the algorithm evolves towards better solutions. A "replacement" strategy guarantees that less fit solutions should be removed from the population.

In the simple GA the population is considered as a whole, meaning that any two solutions can interact with each other for a given iteration of the algorithm. In contrast, with a parallel genetic algorithm (PGA) the population is somehow divided into subpopulations leading to several simple GAs that can run in parallel [3]. Two main architectures of PGAs are known: the distributed GA (dGA) and the cellular GA (cGA). With the dGA the population is partitioned in a set of islands that execute a GA and a migration of solutions is performed from time to time among the different islands. In the cGA, the solutions of the population are distributed into a regular structure and the operations of the GA are applied to solutions within a small neighbourhood. An overlap must exist of neighbourhoods to ensure that solutions propagate along the structure to promote the interaction among all the population. In contrast to the dGA, the cGA does not need a migration scheme.

2.2. Hardware Implementations of GAs

Since a GA deals with a considerable number of solutions (the population), one of the main difficulties to build a dedicated hardware architecture is the management of the memory needed to store that information. In [4], the population of the GA is completely replaced for a new one in each generation of the algorithm. Although with this it is possible to parallelize the crossover and mutation operations, a bottleneck may happen while reading and writing back all the solutions to the memory. This clearly can be a drawback for solving problems that need a large amount of memory which is limited in bandwidth.

To avoid such situation, a "steady-state" GA is adopted where a single solution replaces an existing one from the population on each generation of the algorithm [5]. With this technique, it is possible to build more efficient hardware implementations since it avoids exhaustive memory accesses. The goal here is to generate, within a certain limit, a new solution at each hardware clock cycle. In order to accomplish this, the genetic operations like the selection, crossover or the fitness evaluation, should avoid pipeline stalls [6]. Accelerations of a few hundreds have been reported while comparing a dedicated hardware implementation against a software counterpart [5, 6].

In order to speedup the algorithm some authors adopt the distributed GA model [4, 7]. The idea is to use available hardware resources to increase the number of simple GAs running in parallel. With this, the amount of new generated solutions increases in proportion to the number of islands of the dGA. However, it is necessary to implement a migration strategy of solutions among the different processor elements to guarantee the quality of the final solution of the problem.

It should be noted, however, that the operations of the GA can have different computational efforts, leading to different bottlenecks in the algorithm. The fitness evaluation is usually the most critical operation and is highly dependent on the problem that is being optimized. In some cases this operation is simple to be computed and it may not compromise the execution time of the algorithm, especially if the representation of a solution is simple leading to a straightforward combinatorial circuit to calculate the fitness. The set covering problem is one of this [6]. However, it may happen that the fitness function is the most time consuming operation of the GA. For example, in a chip partitioning problem after a software profiling of the algorithm, the fitness evaluation represents as much as 95% of the total execution time [8]. This clearly shows that this operation can have a tremendous impact in the algorithm execution time.

3. A Proposal for a Cellular GA

By spreading the population into subpopulations, a parallel GA allows the parallelization of several (simple) GAs. Moreover, with this algorithm the management of the memory becomes easier since several memories may exist to store different subpopulations, each one attached to a GA processor. With this, parallelism can be explored without leading to critical bottlenecks in the memory accesses as it happens with the simple GA.

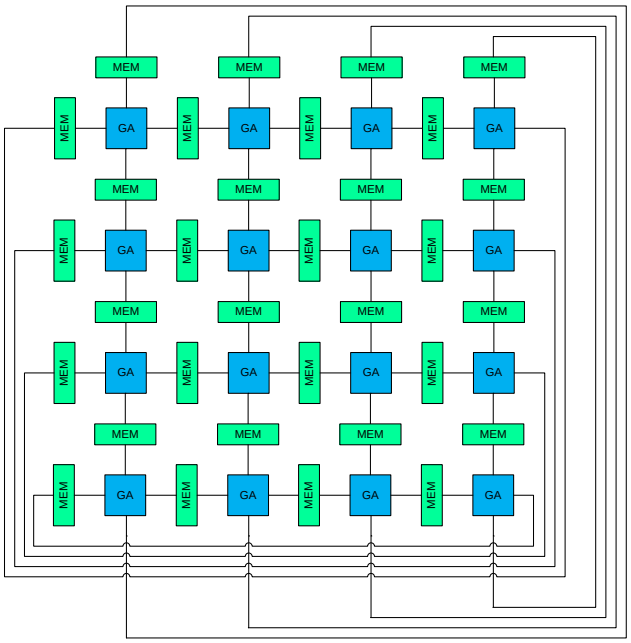


Figure 1. Hardware cellular GA.

This section presents a new architecture of a parallel GA suitable for hardware implementation, mainly for FPGAs devices. The algorithm is based in the cellular GA model which ideally can have a GA processor for each solution of the population, thus maximizing the parallelization of the algorithm.

3.1. The cGA Architecture

The overall architecture of the cGA is depicted in figure 1 exemplified as an array of 4×4 processors. The basic element present in the architecture consists of a GA processor connected to four memories. Each memory stores a subpopulation of the cGA and has two input/output ports that connect to different neighbour processors. By using a same memory block shared by the GA processors nodes at the opposite sides of the array (both for the top-bottom and left-right), the overall structure of the cGA is kept linear and it assumes a toroidal shape. In total, the array has two times more memory blocks than GA processors. In this example, 16 GA processors need a total of 32 memories or subpopulations of the cGA.

Since each processing element has access to four different memories, this architecture allows the reading of two solutions and the writing of a new one in parallel. This can be explored in order to pipeline the operations of a GA processor to speedup even further the algorithm.

3.2. Architecture Simulation

To validate and study the proposed cGA a simulation model of the architecture was developed in SystemC [9].

This language, which is a set of C++ classes capable of providing event-driven simulations, was adopted to reproduce accurately the changes of contents in the shared memories that store the subpopulations of the cGA. The Microsoft Visual Studio 10 was used for compiling the C++ code. For generating all the random numbers needed in the cGA, the function *rand_s*() of the Visual Studio library was adopted which is capable of generating pseudo-random numbers in multi-thread applications, which is needed for SystemC simulation. The travelling salesman problem was chosen as the optimization problem to evaluate the proposed algorithm.

For each GA processor, the algorithm starts by selecting two random solutions from two different memories. A crossover and mutation are performed and a new solution is generated followed by a fitness evaluation. Finally, in the replacement, the new solution is written into one of the local memories. It is assumed that the selection, fitness and replacement operations take 10 clock cycles each one to be processed. For the crossover and mutation operations together a random number between 5 and 15 determines the delay. The randomness of this time aims to desynchronize the operations of different GA processors, reflecting this way different execution times for different generations of a new solution by a given processor. This behaviour is what actually may happen for some techniques of crossovers. The average time of 10 clock cycles per operations is chosen only for simulation purposes and different values will affect the results by a factor of scale for the execution time (or the number of clock cycles).

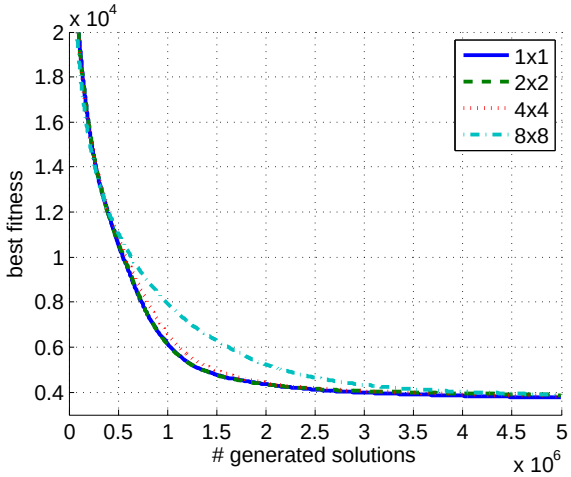
No pipeline of operations is considered inside a GA processor. The maximal preservative crossover (MPX) is responsible to implement the crossover and mutation operation as described in [2]. It should be noted that this operator creates connections between two cities of the TSP which are not present on both parents, acting this way as a mutation. For this reason no mutation operation is performed explicitly.

3.3. Results

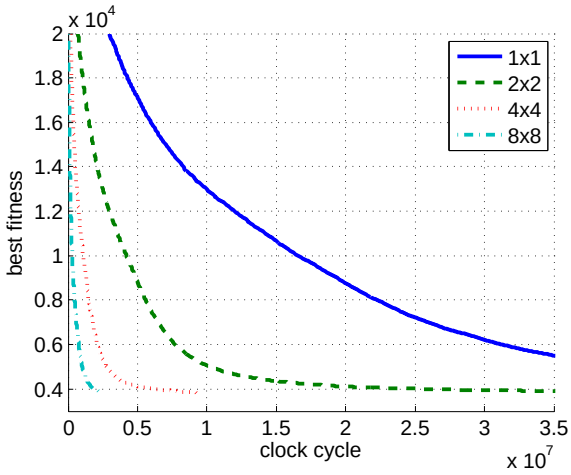
The cGA architecture was evaluated with an instance of 280 cities of the TSP [10]. The experiments start with randomly generated solutions and, in order to have good statistical results, 20 independent runs were performed. For the replacement strategy of a GA processor, at each iteration of the algorithm, the new generated solution substitutes the worst previously selected solution if this one is worse than the new solution. If the two selected solutions have a better fitness than the new one, no replacement occurs.

A total population of 256 individuals and architectures of 1×1, 2×2, 4×4 and 8×8 GA processors in the cGA were tested. This means that 128, 32, 8 and 2 solutions are available in each subpopulation, correspondingly. The case of 1×1 corresponds to a simple GA with two memories (to keep the same structure of the proposed architecture).

Figure 2(a) shows the fitness evolution over 5 million new generated solutions in all GA processor of the cGA array. From the simulation results obtained, it is clear that



(a) Fitness evolution with number of generated solutions.



(b) Fitness evolution with number clock cycles.

Figure 2. Fitness evolution for different array sizes of the cGA (population of 256 solutions).

an increase of the number of processors does not degrade the quality of the final solution found by the cGA.

However, in the architectures with a larger number of processors, especially for the 8×8 case, the convergence rate starts clearly to decrease after a certain number of generated solutions. This behaviour can be explained as the number of the GA processors increases, a solution's information takes more time to spread to the rest of the architecture and to combine with other solutions and so each GA processor tends to get stuck in local minimums.

As hardware execution time is concerned, figure 2(b) depicts the fitness evolution accordingly to the number of clock cycles needed to generate 5 million solutions for different array sizes of the cGA. The 8×8 architecture achieves a considerable speedup over the other architectures with less GA processors. The effects of a slow convergence rate per iteration previously seen for arrays with high number of GA processors are fully compensated by the speedup obtained by the parallelism.

It should be emphasized that for array configurations with a higher number of processors, the convergence rate slows down because of the small number of solutions

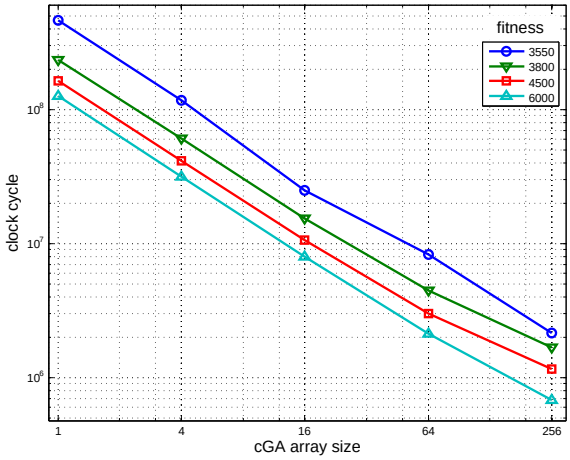


Figure 3. Number of clock cycles needed to achieve a given fitness value for different cGA sizes (population of 1024 solutions).

present in each subpopulation of the cGA. It is not the size of the array of the cGA that directly affects the convergence but, instead, it is the subdivision of the global population into small subpopulations. To show this, a second experiment was performed now with a total population of 1024 solutions applied to arrays of 1×1, 2×2, 4×4, 8×8 and 16×16. These configurations lead to subpopulations of 512, 128, 32, 8 and 2 solutions respectively. 20 million generated solutions were needed to make the algorithm converge for this population size. The convergence rate of the cGA with 16×16 processors behaves in the same way as the 8×8 case of the previous experiment when compared to the corresponding 1×1 array. Both cases have 2 solutions per subpopulation. The same happens for the 8×8 array with a total of 1024 solutions and the 4×4 array with 256 solutions, both with 8 solutions per subpopulation.

Figure 3 depicts (in log-log scale) the number of clock cycles needed to achieve a given value of fitness for the different array configurations of the second experiment. The results, once again, show that the proposed cGA does not degrade the quality of the final solution found by larger arrays and the execution time of the algorithm decreases almost linearly with the number of processors of the cGA.

4. Analysis for a FPGA Implementation

By exploring the dual-port block memories present in modern FPGAs, the proposed cGA fits well these programmable devices. The algorithm, by spreading the populations into small subpopulations and by having several GA processors capable of running at the same time, becomes a scalable hardware architecture. This section discusses the implementation of the proposed cGA in current FPGAs.

4.1. The GA Processor

A custom hardware FPGA was developed that implements a simple GA [11]. As in the previous section, the TSP was chosen as a case study and the same genetic operators of the GA were adopted. The proposed architecture can process nearly one city of the problem on each clock cycle by implementing an efficient hardware pipeline in the operations of the GA. The algorithm was described using Verilog language and successfully implemented in a Virtex-4 FPGA (XC4VLX80-10) running at 96MHz for solving a 1002 city problem [10]. Comparing to the same algorithm coded in C and running in the embedded PowerPC processor of a Virtex-4FX FPGA, a speedup of 20x was observed. Synthesis results showed that for this instance of the TSP and with a GA with a population of 62 solutions, this architecture uses 810kbit of RAM, that represents 22% of the total dual port memory blocks (BRAMs) available in the FPGA, and less than 1% of the LUTs and flip-flops. With a TSP with 4000 cities, the logic resources needed to implement the GA are kept almost the same and below 1%, however the BRAMs of the FPGA are almost fully utilized.

A GA processor of the proposed cGA is similar to the one presented previously since the operations of the GA are the same. It is clear that the parallelization of several GA processors is possible to be implemented into a FPGA as they require a reduced amount of logic resources.

4.2. The Subpopulation

In the cGA presented in this work each subpopulation connects to two different GA processors that can access to the same memory at the same time. For simplicity and optimization of the digital circuits it is desired the use of a dual port RAM. Nowadays FPGAs provide a considerable number of BRAMs with such feature. As an example, the FPGA used to implement the simple GA described over the last subsection (XC4VLX80) has 200 BRAMs each one with a capacity of 18kbit. This means that a maximum of 200 subpopulations can be implemented together with 100 GA processors (half of the number of subpopulations) of the cGA. Nevertheless, the 18kbit of a BRAM limits the number of solutions capable to be stored in a single memory.

$$n_{sol\_subpop} \cdot n_c \cdot \lceil log_2(n_c) \rceil \leq n_{BRAM\_subpop} \cdot 18 \cdot 2^{10} \quad (1)$$

Equation 1 describes the relation among the number of solutions in a subpopulation (n_{sol_subpop}); the maximum number of cities of the TSP (n_c); and the number of BRAMs needed to store a subpopulation (n_{BRAM_subpop}). This equation assumes a capacity of 18kbit for a BRAM and that a solution of the TSP in the path representation (a list of unique cities) needs $n_c \cdot \lceil log_2(n_c) \rceil$ bit to be represented. In table 1, some numerical examples (for the XC4VLX80 FPGA) are presented for these variables together with the number of GA processors and the size of the population of the cGA.

4.3. FPGA Resources

The proposed cGA is well suited for a FPGA implementation as it can increase/decrease the number of GA processors and the subpopulations. Since the operations of a GA

#BRAMs p/ subpop	#GA processors	#sol p/ subpop	pop size	#cities
1	100	1	200	1675
		2	400	921
		4	800	512
		8	1600	256
2	50	1	100	3072
		2	200	1675
		4	400	921
		8	800	512
3	33	1	66	4253
		2	132	2304
		4	264	1256
		8	528	691

Table 1. Maximum array size of the cGA applied to the TSP for a XC4VLX80 FPGA.

processor are simple, it is possible to have a considerable amount of parallel processing elements. Nevertheless, for optimization problems requiring a large amount of memory to store the solutions, it may happen that a single subpopulation needs more than one BRAM to store the information. This leads to a decrease of the number of subpopulations in the cGA and, consequently, to less GA processors. To overcome this, a decrease of the overall population size must be performed.

5. Conclusions

This paper presents an architecture of a cellular genetic algorithm suitable to be implemented in modern FPGA devices. The cGA is scalable since the number of GA processors and subpopulations can be increased to speedup the algorithm. Nevertheless, the size of the array of the cGA is limited by the number of available BRAMs in the FPGA since a single subpopulation requires at least one of these blocks. For problems requiring large amounts of memory the number of subpopulation of the cGA needs to decrease, leading to a decrement of the number of GA processors.

Simulations performed to evaluate the cGA as an optimization technique have shown the effectiveness of the algorithm. This means that despite the spreading of solutions used in the proposed cGA, the algorithm does not degrade the quality of the final solution. The parallelism obtained by having more GA processors clearly shows effective speedup which grows proportionally with the number of processing nodes. The result is a cGA that guarantees the same solution quality as the simple GA and it can be more efficient as it may run faster due to parallelization.

Acknowledgments

This work has been partially funded by Fundação para a Ciência e a Tecnologia (FCT) through the Ph.D scholarship SFRH/BD/41259/2007 and by the bilateral cooperation between FCT and Deutscher Akademischer Austausch Dienst (DAAD) through the project FCT/DAAD 2010/2011 under reference daad124412622223295.

References

[1] J. H. Holland, *Adaptation in Natural and Artificial Systems*. The MIT Press, 1992.

[2] P. Larranaga, C. Kuijpers, R. Murga, I. Inza, and S. Dizdarevic, "Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators," *Artificial Intelligence Review*, vol. 13, no. 2, pp. 129–170, 1999.

[3] E. Cantu-Paz, *Efficient and Accurate Parallel Genetic Algorithms*. Springer Netherlands, 2000, vol. 1.

[4] P. Graham and B. Nelson, "A Hardware Genetic Algorithm for the Traveling Salesman Problem on Splash2," in *Field-Programmable Logic and Applications*. Springer-Verlag, 1995, pp. 352–361.

[5] O. Kitaura, H. Asada, M. Matsuzaki, T. Kawai, H. Ando, and T. Shimada, "A Custom Computing Machine for Genetic Algorithms without Pipeline Stalls," vol. 5, pp. 577–584, 12–15 Oct. 1999.

[6] B. Shackleford, G. Snider, R. J. Carter, E. Okushi, M. Yasuda, K. Seo, and H. Yasuura, "A High-Performance, Pipelined, FPGA-Based Genetic Algorithm Machine," *Genetic Programming and Evolvable Machines*, vol. 2, no. 1, pp. 33–60, 2001.

[7] T. Tachibana, Y. Murata, N. Shibata, K. Yasumoto, and M. Ito, "General Architecture for Hardware Implementation of Genetic Algorithm," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2006 14th Annual)*, 2006, pp. 291–292.

[8] N. Sitkoff, M. Wazlowski, A. Smith, and H. Silverman, "Implementing a Genetic Algorithm on a Parallel Custom Computing Machine," in *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, 19–21 April 1995, pp. 180–187.

[9] Open SystemC Initiative (OSCI). [Online]. Available: http://http://www.systemc.org/home/

[10] G. Reinelt. Tsplib. [Online]. Available: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

[11] P. Santos and J. Alves, "FPGA Based Engines for Genetic and Memetic Algorithms," in *2010 International Conference on Field Programmable Logic and Applications*. IEEE, 2010, pp. 251–254.