A FRAMEWORK FOR HARDWARE CELLULAR GENETIC ALGORITHMS: AN APPLICATION TO SPECTRUM ALLOCATION IN COGNITIVE RADIO

Pedro Vieira dos Santos, José Carlos Alves, João Canas Ferreira

INESC TEC (formerly INESC Porto)
Faculdade de Engenharia, Universidade do Porto
Porto, Portugal
email: pedro.vieira.santos@fe.up.pt, jca@fe.up.pt, jcf@fe.up.pt

ABSTRACT

The genetic algorithm (GA) is an optimization metaheuristic that relies on the evolution of a set of solutions (population) according to genetically inspired transformations. In the variant of this technique called cellular GA, the evolution is done separately for subgroups of solutions. This paper describes a hardware framework capable of efficiently supporting custom accelerators for this metaheuristic. This approach builds a regular array of problem-specific processing elements (PEs), which perform the genetic evolution, connected to shared memories holding the local subpopulations. To assist the design of the custom PEs, a methodology based on high-level synthesis from C++ descriptions is used. The proposed architecture was applied to a spectrum allocation problem in cognitive radio networks. For an array of 5×5 PEs in a Virtex-6 FPGA, the results show a minimum speedup of 22× compared to a software version running on a PC and a speedup near 2000× over a MicroBlaze soft processor.

1. INTRODUCTION

Genetic algorithms (GAs) have been widely used for solving complex optimization problems with proven success in a wide range of applications [1]. This class of algorithms mimics the natural evolution of living species by iteratively applying genetically inspired operations to a *population* of solutions with the goal of evolving towards good, but not necessarily optimal, solutions. However, the large number of iterations usually required represents a drawback when targeting embedded systems, particularly if real-time constraints apply.

In the last few years, cellular genetic algorithms (cGAs) have become a promising area of research in the field of evolutionary algorithms [2], exhibiting interesting opportunities for parallel custom computing. This variant of GAs spreads the solutions over a regular grid, and constrains the application of the *genetic* interactions among the solutions to local subpopulations. This leads to a natural parallelization of the algorithm, as the evolution of the overall population can be handled in parallel by independent processors, each one dealing with its local subpopulation.

The goal of the work presented in this paper is to build custom hardware accelerators for cGAs. Besides building application-specific units capable of efficiently performing the genetic operations, we also exploit the inherent parallelism of cGAs to accelerate the optimization process. This is done by spreading the evolutionary process over a regular array of problem-specific processing elements (PEs) that operate on partially overlapped sub-sets of the population. The proposed architecture is built around a scalable and parameterized generic array framework that implements the memory system, the communication and control infrastructure, and a set of problem-specific PEs that are specified in C++ and translated to hardware by high-level synthesis (HLS).

This architecture and design methodology have been evaluated with an implementation of a cGA for solving a spectrum allocation problem in cognitive radio networks. An embedded system with a MicroBlaze attached to the cGA array has been implemented in a Xilinx Virtex-6 FPGA. A 25 node processor array has shown significant speedups compared to a software version running on a PC. The final optimization results have better quality than the ones obtained by a heuristic procedure.

This work makes the following contributions: *(a)* Describes a scalable and parameterized processor array for the execution of cGAs, including a distributed control and communication infrastructure to access the array components from an host processor; *(b)* Presents implementation and execution results for an NP-hard problem. Moreover, a high-level design development methodology for creating the cGA processing element is presented, which eases the implementation of other optimization problems.

This work has been partially funded by National Funds through the FCT - Fundaçao para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) 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.

2. STATE OF THE ART

The genetic algorithm is a population-based optimization metaheuristic inspired by the evolutionary refinement of living beings [1]. A pool of solutions (the *population*) evolves iteratively by replacing existing solutions with new ones that are created by operations like *selection*, *crossover* and *mutation*. With an appropriate encoding of solutions and genetic operators, and using replacement strategies based on the evaluation of an objective function (or *fitness*), the GA has proven to be an effective optimization metaheuristic for a variety of NP-hard problems.

When a single population exists and any solution can interact with any other, the GA is called *panmictic* [2]. The work reported in [3] proposes a general-purpose GA engine where several parameters can be specified together with an interface to a custom fitness function. The work is based on the *generational* GA, where a new complete population is created at each generation completely replacing the existing one. Another approach is the *steady-state* GA, where the population is evolved by generating a single solution per iteration to replace an existing one. This approach requires less memory than the generation GA, and it can lead to efficient pipelined architectures [4].

In contrast to the panmictic GA, a *structured* GA divides its population into smaller subpopulations. Two main classes of structured GAs are usually considered: the *distributed GA* (dGA) and the *cellular GA* (cGA) [2]. In the dGA the population is divided into separate subpopulations, which evolve independently with sporadic interaction among them. In [5] a dGA is proposed with 2 processing units (each one implemented in different FPGAs) for solving the set covering problem. The results show a speedup of two times when compared to a single processor. The work developed in [6] integrates 4 GA processors in an Altera Cyclone FPGA device for solving a traveling salesman problem.

In the cGA, solutions are distributed in a regular array and one solution can only be combined with others residing in the vicinity. This shows an interesting potential for acceleration with coarse-grain parallel processing architectures, as several solutions can be evolved simultaneously. In previous work we validated the concept of the proposed array architecture with the traveling salesman problem [7]. In this paper we propose a flexible and scalable hardware framework for supporting this category of GAs and present implementation and execution results for a relevant optimization problem.

3. THE CGA HARDWARE ARCHITECTURE

3.1. General cGA array architecture

The cellular GA hardware architecture proposed in this work is depicted in Figure 1. A basic cell formed by a PE attached to two dual-port memories is replicated to build a 2D regular array. The whole population handled by the algorithm is spread over multiple memories (subpopulations). A single PE has only direct access to its four local memories and each memory is shared by two adjacent PEs.

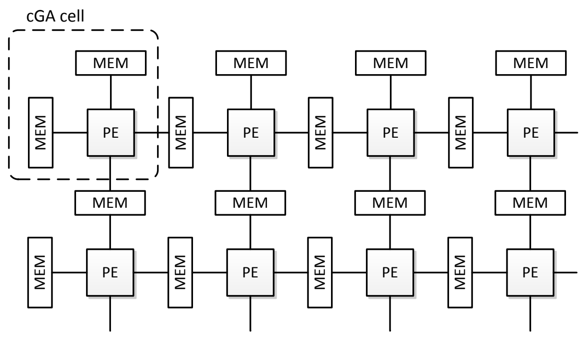

Fig. 1. Hardware architecture of the cellular genetic algorithm array.

The architecture exhibits a regular structure and it can thus be constructed as a rectangular array with varying number of PEs and different aspect ratios. By connecting the opposite sides of the array (both for top-bottom and left-right) the architecture acquires a toroidal shape. In alternative, non-toroidal 2D shapes can be easily built by including additional memories so that all PEs connect to four memories.

3.2. Interface and control

The proposed cGA array is intended as a peripheral of an embedded processor that controls the algorithm setup and execution. Figure 2 shows the overall architecture of such system, consisting of an embedded processor, a cGA controller unit, and the cGA array itself. The control unit is mainly responsible for the interface between the software part (the processor) and the cGA array.

PE interface: To enable the access to each cGA cell by the host processor, a network infrastructure that traverses all the array is included. The network uses a simple routing mechanism, using as address the row/column indexes of each PE. The network enables sending commands to each PE for controlling its execution state and accessing its local memories, and also receiving status information.

Subpopulation memory access control: As simultaneous accesses from two neighboring PEs to the same memory region (holding one solution) can occur, read and write operations must guarantee memory coherency. This is done by a dedicated collision avoidance circuit associated with each subpopulation memory; it is responsible for tracking which solutions are being used by each PE and for informing them if a conflicting access occurs.

Global RNG: The operation of the genetic algorithm relies on random numbers. To avoid the replication of random number generators (RNGs) in all the PEs, a global RNG feeds a dedicated 1-bit network that traverses all the PEs. A shift-

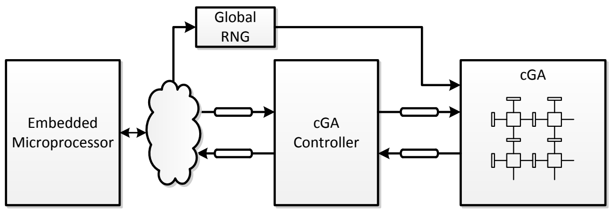

Fig. 2. cGA hardware overall architecture.

register, local to each PE, acquires the necessary random numbers for the operation of the algorithm.

4. HARDWARE DESIGN FLOW

The customization of this architecture for a new problem includes two main phases: configuring the array size and shape, and creating the PE and the cGA array controller. The organization of the cGA array is defined by a small set of parameters that configure a synthesizable Verilog RTL model. The PE must be designed for each new problem, as the behavior of the operations of the genetic algorithm (selection, crossover, mutation and fitness evaluation) are highly specific of each particular application.

In order to simplify the design task of the PE, a high-level synthesis from C++ design flow was adopted. It starts from a template that matches the interface between the PE and the surrounding memories and control infrastructure. This template also includes a set of C++ classes for accessing the PE local memories and the control bus, manage memory collisions and access the global RNG. This way, a PE functionality is described in C++ where a set of methods are called when needed to perform the subpopulation memory access control and to access the control bus and the global RNG. The high-level synthesis tool used was Catapult combined with RTL synthesis by Precision RTL. The result of this process is a netlist that is included in a ISE/EDK project previously created for a given FPGA-based platform.

5. CASE STUDY: SPECTRUM ALLOCATION PROBLEM

The spectrum allocation problem consists of finding an optimal assignment (satisfying some criteria) of the available radio spectrum channels to secondary users (e.g., smartphones, laptops), while a set of primary users already have assigned radio channels. In the model adopted in this work, a set of N secondary users try to access M non-overlapping orthogonal channels. Each secondary user can utilize any channel, but limited by interference constraints among all users as defined by two binary matrices L and C. The spectrum assignment problem consists in finding the best channel assignment binary matrix A. The constraints of the problem are defined as:

$$a_{n,m} \leq l_{n,m}, \quad \forall n < N, m < M \tag{1}$$

Table 1. Results for different spectrum allocation instances with a cGA of 5×5 PEs.

		cGA hardware		Speedup	
instance	CSGC fitness	best fitness	time for 10^6 generations	Micro-Blaze	Intel PC
5_6	1130	1130	0.086 s	1937	22
8_16	6090	6129	0.149 s	3036	44
16_16	6959	6985	0.350 s	2847	43
16_32	15197	15220	0.384 s	4423	74
20_24	11209	11278	0.498 s	3708	56
32_32	22882	22978	1.023 s	4115	64

$$a_{n,m} + a_{k,m} \leq 1, \text{ if } c_{n,k,m} = 1, \forall n, k < N, m < M \tag{2}$$

and the objective of the problem is to maximize a given utility function:

$$U_{sum} = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} a_{n,m} \cdot b_{n,m} \tag{3}$$

where the matrix B represents a channel quality metric. Further details of this mathematical model can be found in [8].

6. IMPLEMENTATION AND RESULTS

The proposed cGA array hardware architecture together with the methodology described in Sec. 4 has been configured to solve the spectrum allocation problem.

A solution of the problem is encoded with a N×M bitmap representing the channel assignment matrix A. By design, we decided to use a single BRAM of the target Virtex-6 FPGA per subpopulation memory and constrain N and M to a maximum of 32. Additionally, each memory uses half of the space to keep all the data necessary to configure a spectrum allocation problem, while the other half keeps the solutions of the population.

Two binary tournaments are performed to select two solutions as parents. These go through uniform crossover and mutation (bit-flip with probability $< 5\%$) to generate a new solution. A random solution is chosen that will be replaced by the new one if it is better than the one selected for replacement. The crossover and mutation operations may generate solutions that do not satisfy the problem constraints defined by equations 1 and 2. In this case a set of corrections are performed to eliminate the unfeasibility.

Catapult HLS version 2010a (University Version) and Precision RTL 2010a were used to synthesize the PE and cGA controller, and Xilinx ISE and EDK 13.4 to implement the complete embedded system. The global RNG has been implemented with a cellular automata ring network as in [7]. The target device is a Xilinx Virtex-6 FPGA (XC6VLX240T-1) on a ML605 board.

Six instances of the problem have been created, based on the pseudo code provided in [8]. The cGA hardware speedup is measured by comparison with a software version of the

Table 2. Results for different cGA configurations for the *20_24* instance.

cGA array	solutions per subpop.	population size	best fitness	time for 10^6 generations
2×2	16	192	11205	3.10 s
3×3	8	192	11221	1.38 s
4×4	5	200	11246	0.78 s
5×5	3	180	11248	0.50 s

same algorithm coded in C and executed on a PC and on the MicroBlaze processor. Additionally, the quality of the solutions found is compared with the results of an heuristic named colour-sensitive graph colouring (CSGC) [8].

Table 1 presents the results obtained for a cGA hardware configuration of 5×5 PEs with non-toroidal shape. The instances are named N_M and each subpopulation memory accommodates 8 solutions, resulting in a population of 480 individuals. Results are averaged over 100 independent runs. The results show that the best averaged fitness obtained with the cGA surpasses the existing CSGC heuristic. Regarding execution time, the hardware cGA array achieves a speedup ranging from 22 to 74× over a PC with an Intel T8100 processor at 2.1 GHz; compared to the MicroBlaze processor running at 150 MHz the speedup exceeds 1937×. Table 2 presents the results of running instance *20_24* with different cGA array configurations, ranging from 2×2 to 5×5. The results were obtained with the same procedure, but now considering a population with approximately 200 solutions. The number of PEs ranges from 4 to 25 and the results show a throughput (number of generations per time unit) that increases linearly with the number of PEs.

The proposed cGA hardware architecture can be tailored by choosing the number of PEs, impacting not only on the execution time, but also on the hardware resources needed. Table 3 shows the FPGA resources used for different array configurations, including the MicroBlaze processor with MMU and FPU. The designs were implemented for a MicroBlaze clock frequency of 150 MHz, with the cGA hardware running at 75 MHz.

The same problem is addressed for an instance *5_5* in [9], where the generation of a single solution takes an average time of 547 µs (0.093 s reported for running 10 generations with 17 solutions each) running on a PC at 1.66 GHz in Matlab. With our 5×5 cGA array and a slightly larger *5_6* instance, the average time for generating one solution is only 86 ns, which represents a 6360× speedup over the reported implementation.

7. CONCLUSIONS

This paper presents a flexible framework for implement cellular genetic algorithms in custom hardware. The array is formed by problem-specific processing elements that implement the genetic evolution on a subpopulation stored in a set of local memories, shared by adjacent PEs. By specifying the functionality of the PEs and a controller module, the cGA architecture can be easily customized for different optimization problems. To ease the development of a PE, its algorithmic behavior is specified in C++ and a high-level synthesis design flow is used. The results obtained by building a specific cGA array for a spectrum allocation problem in cognitive radio networks have shown significant speedups. For an array with 5×5 PEs, a minimum speedup of 22× was measured, when comparing to a C code for the same algorithm running on a PC, and near 2000× compared to the embedded MicroBlaze processor running at 150 MHz.

Table 3. Characteristics of the cGA implementations on a Virtex-6 (XC6VLX240T-1) for different array sizes.

Parameter	2×2	3×3	4×4	5×5
Registers	35848 (11%)	47092 (15%)	62807 (20%)	83111 (27%)
LUTs	36266 (24%)	51664 (34%)	72653 (48%)	100790 (66%)
Slices	15782 (41%)	21949 (58%)	31262 (82%)	35507 (95%)
BRAMs	49 (11%)	61 (14%)	77 (18%)	97 (23%)

8. REFERENCES

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

[2] E. Alba and B. Dorronsoro, *Cellular Genetic Algorithms*. Springer Verlag, 2008, vol. 42.

[3] P. Fernando, S. Katkoori, D. Keymeulen, R. Zebulum, and A. Stoica, "Customizable FPGA IP core implementation of a general-purpose genetic algorithm engine," *IEEE Trans. Evol. Comput.*, vol. 14, no. 1, pp. 133–149, 2010.

[4] P. Santos and J. Alves, "FPGA based engines for genetic and memetic algorithms," in *Proc. Field-Program. Logic Appl.* IEEE, 2010, pp. 251–254.

[5] Y.-H. Choi and D. J. Chung, "VLSI processor of parallel genetic algorithm," in *Proc. Second IEEE Asia Pacific Conf. ASICs*, 2000, pp. 143–146.

[6] T. Tachibana, Y. Murata, N. Shibata, K. Yasumoto, and M. Ito, "General architecture for hardware implementation of genetic algorithm," in *14th Annual IEEE Symp. Field-Program. Custom Comput. Machines*, 2006, pp. 291–292.

[7] P. V. dos Santos, J. C. Alves, and J. C. Ferreira, "A scalable array for cellular genetic algorithms: TSP as case study," in *Int. Conf. on Reconfigurable Computing and FPGAs*. IEEE, 2012, pp. 1–6.

[8] C. Peng, H. Zheng, and B. Y. Zhao, "Utilization and fairness in spectrum assignment for opportunistic spectrum access," *Mobile Networks and Applicat.*, vol. 11, no. 4, pp. 555–576, 2006.

[9] Z. Zhao, Z. Peng, S. Zheng, and J. Shang, "Cognitive radio spectrum allocation using evolutionary algorithms," *IEEE Trans. Wireless Commun.*, vol. 8, no. 9, pp. 4421–4425, 2009.