U.Porto Journal of Engineering, 2:2 (2016) 2-13
ISSN 2183-6493
DOI 10.24840/2183-6493_002.002_0002

Received: 12 October, 2015
Accepted: 29 March, 2016
Published: 29 July, 2016

A Reconfigurable Custom Machine for Accelerating Cellular Genetic Algorithms

P. V. Santos[1], José Carlos Alves[2], João Canas Ferreira[3]
INESC TEC – INESC Technology and Science and FEUP - Faculty of Engineering, University of Porto, Porto, Portugal ([1]pmsantos@inesctec.pt, [2]jca@fe.up.pt, [3]jcf@fe.up.pt)

Abstract
In this work we present a reconfigurable and scalable custom processor array for solving optimization problems using cellular genetic algorithms (cGAs), based on a regular fabric of processing nodes and local memories. Cellular genetic algorithms are a variant of the well-known genetic algorithm that can conveniently exploit the coarse-grain parallelism afforded by this architecture. To ease the design of the proposed computing engine for solving different optimization problems, a high-level synthesis design flow is proposed, where the problem-dependent operations of the algorithm are specified in C++ and synthesized to custom hardware. A spectrum allocation problem was used as a case study and successfully implemented in a Virtex-6 FPGA device, showing relevant figures for the computing acceleration.

Subject Headings. Microelectronics, Electrical Engineering
Author Keywords. Reconfigurable Computing, FPGAs, High-Level Synthesis, Metaheuristics, Cellular Genetic Algorithms, Spectrum Allocation

1. Introduction

Reconfigurable custom computing machines are an effective mean to accelerate critical algorithms that require to be executed under strict timing constraints. In this regard, field-programmable gate arrays (FPGAs) devices provide high potential for accelerating computing tasks by configuring a very flexible logic fabric to implement the desired computing operations. However, programming these devices requires building the design of a custom digital circuit capable of handling efficiently the desired tasks, which is usually done with hardware description languages like Verilog or VHDL.

This paper presents a framework for implementing custom computing machines capable of accelerating cellular genetic algorithms (cGAs) in FPGA devices. The cGA, a variant of the genetic algorithm (GA) metaheuristic, is inspired by the evolution of living species, where principles inspired in the natural selection and genetics are applied to solve complex optimization problems. Although GAs are well-known for effectively exploring a huge search space by sensing only a small fraction of it, they still require a considerable amount of execution time as the iterative process of the metaheuristic needs to be repeated numerous times. Therefore, a dedicated computing engine specifically designed to implement this metaheuristic can allow the utilization of genetic algorithms when timing constraints apply or when processing power is limited.

We present a scalable processor array, named cellular genetic algorithm processor - cGAP, built by assembling a regular matrix of independent memories shared by problem-specific processing elements (PEs) that execute the genetics-inspired operations. The population of solutions evolved by the metaheuristic (or the GA population) is thus distributed over these memories and the array of processing elements evolve that population in parallel.

To facilitate the design of the cGAP we propose a design flow where the problem-dependent operations of the algorithm are specified in C++ and automatically translated to digital hardware using high-level synthesis (HLS) tools. This allows customizing the operations of the metaheuristic according to the requirements of the problem, open this to users familiar with C++ programming but having limited knowledge on digital design. Therefore, the problem-specific constraints, the special genetics-inspired operations, and the fitness evaluation function (or the objective function) can be *rapidly* and *easily* described to reconfigure the cGAP for different target problems.

A spectrum allocation (SA) problem in cognitive radio networks has been used to demonstrate the effectiveness of the engine and its design methodology. This relevant problem in the domain of wireless ad hoc networks has special constraints that must be satisfied and thus our HLS-based design flow can successfully deal with the implementation of them. The cGAP is implemented with different levels of parallelism ranging from 1 PE to 5×5 PEs, targeting a Virtex-6 FPGA from Xilinx. Results show a throughput that is directly proportional to the number of PEs while ensuring similar quality solutions found by the metaheuristic for the different levels of parallelism. The hardware speedup observed when comparing the 5×5 PE array with a similar GA implemented in software are between 7× for a i7 Intel processor (3.4 GHz) and more than 5000× for an embedded MicroBlaze (150 MHz).

The paper is organized as follows. Section 2 presents the genetic algorithm and reviews dedicated hardware implementations of this metaheuristic. In Section 3 the proposed processor array (the cGAP) is presented together with the main infrastructures required to support the execution of the metaheuristic. The design flow used to implement the cGAP is introduced in Section 4 and Section 5 presents the genetic algorithm formulation for an optimization case study in the domain of cognitive radio networks. The implementation and results obtained with this example are presented in Section 6, followed by the final conclusions in Section 7.

2. Background and State of the Art

The genetic algorithm (GA) is a population-based metaheuristic where a population constituted by a set of feasible solutions of an optimization problem goes through an evolutionary process inspired by the biological evolution of living species, in order to improve their quality (Holland 1975). Therefore, this metaheuristic relies on mimicking operations that are observed in nature, as the genetics-inspired operations crossover, mutation, and natural selection. The solutions cooperate and compete among them so that during the evolutionary process only the most fit solutions survive to propagate their genetic information to future generations.

Figure 1 illustrates the iterative process of the genetic algorithm. The algorithm starts with a random population *P*, where its elements (represented by dots in the figure) represent valid solutions of the optimization problem. The algorithm proceeds with the evolution of the current population by applying genetics-inspired operations. First, a *selection* of solutions in *P* is performed to designate a subset of solutions that will undergo transformations for creating new solutions. The selected solutions are called *parents* and are combined (usually in pairs) to generate new ones through a *crossover* operation. The new solutions may then suffer a *mutation* that induces small changes to them. This builds a new population *P′* so that the population for the next generation is created through a replacement strategy that elects solutions among *P* and *P′*. Both the selection and replacement operations choose solutions according to an evolution strategy that takes into account their fitness values to promote the progression towards a better population. This process is repeated iteratively leading to an improvement of the global population fitness quality and the algorithm stops when a given criterion is met.

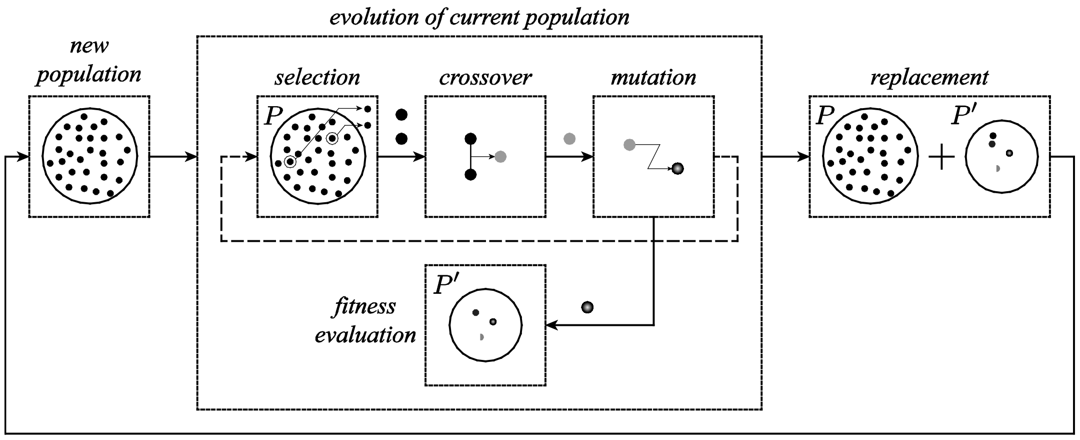

Figure 1: Iterative process of a GA

2.1. Decentralized Genetic Algorithms

While most of GAs evolve a single population using a global selection procedure (called *panmictic*), other categories of genetic algorithms use decentralized sets of solutions that develop autonomously, while allowing some form of migration of solutions among them, therefore spreading the genetic information throughout the whole population (Figure 2). The cellular genetic algorithm addressed in this work belongs to this late category. All the solutions that form the population are distributed in a regular grid and one solution can only interact with the solutions belonging to its neighbourhood.

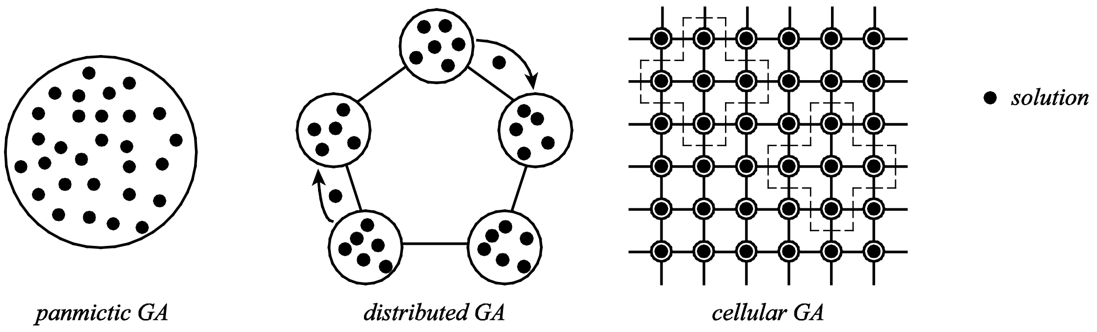

Figure 2: Panmictic vs. decentralized GAs

2.2. Dedicated Hardware Implementations of GAs

Since the 1990's there has been a continuous research activity to implement custom computing machines that support GAs, mainly motivated by the acceleration potential of FPGA devices. Nevertheless, the majority of the works focus on the implementation of the basic and general purpose genetics-inspired operations and often disregard the organization of the population of the GA or the evaluation of the fitness function. Indeed, it is often the case that the basic operations of the algorithm do not constrain the overall performance of the algorithm, while the access to a single memory holding the whole population may represent a severe bottleneck when exploiting parallel processing. This is particularly true for the variants of the algorithm evolving a single population (*panmictic*). However, the cellular genetic algorithm (cGA) raises the possibility of distributing the solutions in independent memories that are only accessed locally by a limited number of associated processing units.

This can lead to an improvement in the parallelism level without critical memory access bottlenecks.

Work	Optimization problem	Hardware platform	Acceleration	GA architecture
(Guo, Thomas, and Luk 2014)	Locating; set covering; mathematical functions	Virtex-6	30× (PC 2.67 GHz)	gGA
(Fernando et al. 2010)	Mathematical functions	Virtex-II Pro	5.1× (PowerPC in FPGA)	gGA
(Deliparaschos, Doyamis, and Tzafestas 2008)	Mathematical functions; TSP	Spartan-3	11× (PC 3.2 GHz)	gGA
(Nambiar et al. 2013)	Mathematical functions; parameter tuning in finger-vein biometrics	Stratix II	102× (soft-processor Nios II in FPGA)	ssGA
(Ahmadi et al. 2011)	Mathematical functions	Spartan-3	218x (PC)	ssGA
(Vavouras, Papadimitriou, and Papaefstathiou 2009)	Mathematical functions	Virtex-II Pro	1.2-60× (others FPGA implementations)	ssGA
(Tsai, Huang, and Chan 2011)	Path planning	Altera FPGA	90× (PC 3.4 GHz)	dGA (2 nodes)
(Jelodar et al. 2006)	OneMax; Mathematical function	Altera Stratix	50×; 20× (soft-processor Nios)	dGA (2 nodes)
(Tachibana et al. 2006)	TSP; Knapsack	Altera Cyclone	-	dGA (4 nodes)

Table 1: Review of dedicated FPGA implementations of GAs

Table 1summarizes recent works that implement GAs in FPGA devices. The table is organized according to the structure of the population of the algorithm: generational GA (gGA) that replaces the whole population in each iteration, steady-state GA (ssGA) that evolves the population one solution at a time, and distributed GA (dGA). Although decentralized genetic algorithms exhibit a great potential for exploiting hardware acceleration with dedicated hardware implementations, this has only been addressed with a few processing nodes and targeting specific optimization problems. To the best of our knowledge, there has not been other custom implementations of the cellular genetic algorithms combining scalability and flexibility for handling different problems than the proposed in this work (dos Santos, Alves, and Ferreira 2012).

3. A Scalable Array Processor for Accelerating Cellular Genetic Algorithms (cGA)

A cellular GA uses a pool of solutions distributed over a regular grid and evolves the population by combining solutions selected within partially overlapped neighbourhoods. We have elected this class of GAs to be supported by a custom computing machine built with an array of processing nodes capable of evolving concurrently the population.

Figure 3 depicts the overall organization of the proposed architecture, called cellular genetic algorithm processor (cGAP), which is constituted by replicating two main blocks: subpopulation memories (spMEMs) and processing elements (PEs). Each spMEM holds a subset of solutions (or subpopulation) of the algorithm and is shared between two adjacent PEs. In turn, each PE accesses to the subpopulations residing in its four neighbour memories and it is responsible to apply the genetic algorithm to those solutions. This architecture implements a cGA since each solution can only interact with a predefined number of solutions in its neighbourhood. Additionally, throughout the array there is an overlap of different solutions' neighbourhoods which imposes an implicit mechanism of migration of the solutions' information along the whole population.

One of the main advantages of this architecture is its scalability since the number of PEs can be selected to fulfil the desired performance or area requirements. From one side, a higher number of PEs leads to an improved throughput but it requires more hardware resources. Situations where harsh real-time constraints exist may take advantage of an increased parallelism by using a large number of PEs.

It is known that the population size is determinant for the speed of convergence because larger populations require more iterations to reach a certain convergence criterion. On the other hand, the quality of the final solutions tends to be better when using larger populations. Because of this, the population size is usually chosen based on the knowledge of the solution space and constraints of the problem, and also in the computing budget available for executing the optimization process. In addition, in our implementation the minimum population size is also dictated by the number of spMEMs, what happens when each memory only holds a single solution shared by two adjacent PEs.

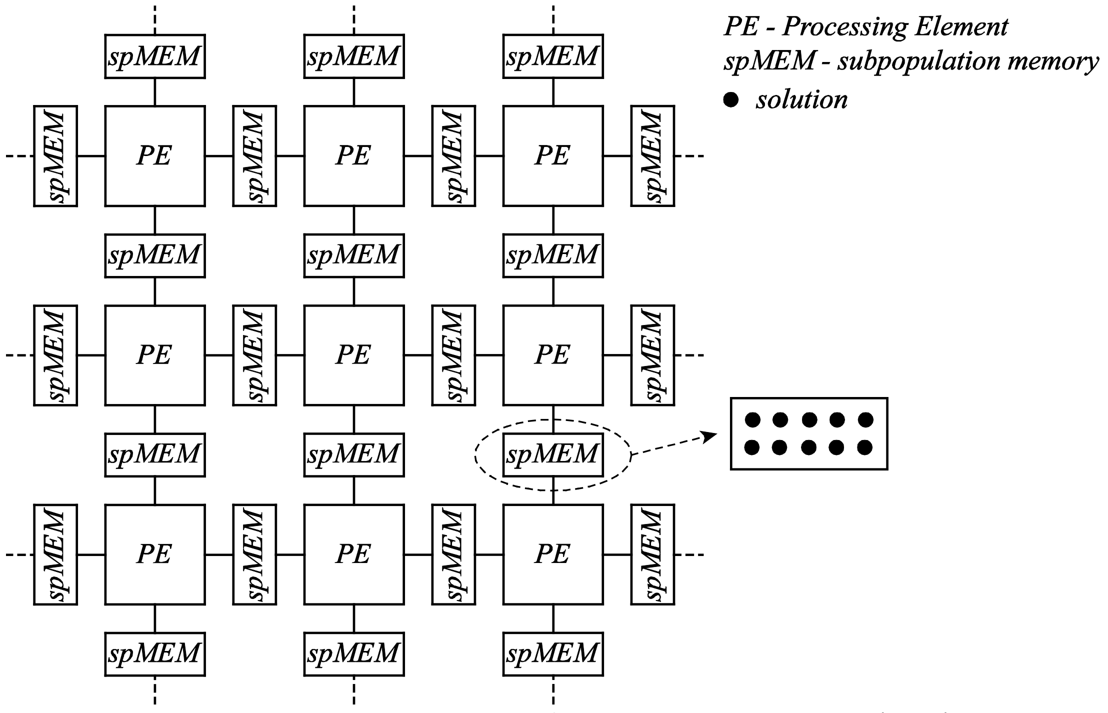

Figure 3: A scalable processor array for cellular genetic algorithms (cGAP)

Below we summarize the main characteristics of the proposed cGAP.

- *Scalable*: The number of PEs can be selected to achieve different levels of parallelism.
- Nevertheless, the maximum parallelism level is constrained by the global population size.
- *Regular*: The cGAP is built by replicating the PE and associated spMEMs in a regular array, mainly using only local connections among the PEs and the spMEMs.
- *High memory bandwidth*: Changing the level of parallelism does not introduce any potential memory access bottleneck since the number of spMEMs increases in proportion with the number of PEs.
- *Shared memories*: Although the main goal of the spMEMs is to keep the problem's solutions, they can also be used to store additional data that may be needed by the two PEs accessing that spMEM. Typical examples are problem specific read-only parameters required to compute the fitness function.
- *Concurrent PEs*: The operation of the array of PEs is globally asynchronous. Each PE works at its own pace and does not need to synchronize with the processing running in the other PEs.
- *Configurable*: The cGAP is configured by defining a small number of parameters to define the size and organization of the spMEMs, and the number of rows and columns of the array of PEs.

3.1. Support Infrastructure

To control the execution of the cGA metaheuristic in the array and also provide a convenient interface with a host computer, additional infrastructures are implemented that are detailed in the following sections.

3.1.1. cGA Controller (cGAC)

A block named cGA controller (cGAC) communicates with all the PEs, so that the GA running in each of them can be individually controlled and monitored by sending/receiving commands to/from all the PEs through a dedicated communication infrastructure specifically built for that purpose. Additionally, the cGAC can communicate with an external host processor, acting this way as a bridge between the array of PEs running the cGA and a software application running in a host computer. This is used, for example, to configure parameters during the set-up phase, to start/stop the PEs based on a global stop criterion, or to retrieve the best solution at the end of the algorithm. These functionalities are essential for the cGAP to support the execution of a cGA, while providing effective mechanisms to configure the PEs and monitor the algorithm evolution.

3.1.2. spMEM Access Control

The spMEMs of the cGAP are implemented with dual-port memories available in current FPGA devices. Therefore, simultaneous access to a spMEM by the two PEs connected to it can exist and, in particular, to the same memory contents (a solution). In such cases and to avoid corruption of data, when a PE is updating a solution the other PE cannot be accessing to that particular solution at the same time. To ensure this, we have implemented a memory arbitration mechanism to prevent undesired memory access conflicts.

3.1.3. Global Random Number Generator

Since GAs rely heavily in random numbers to compute their genetics-inspired operations, we have introduced a random number generator (RNG) infrastructure where a global RNG feeds all the PEs with the required numbers, thus avoiding the need to replicate RNGs blocks in all the PEs. This infrastructure is continuously carrying random numbers throughout the array of PEs, providing this way the required numbers to these blocks when they are needed. The global RNG adopted in this work is based in cellular automata techniques, and adjusted for hardware implementation (Shackleford et al. 2002).

4. Design Flow using High-Level Synthesis

Although GAs are well-known for using a binary encoding scheme to represent the solutions of the optimization problem, other representations exist that better map the solutions' encoding, as it is the case of graph problems. Additionally, it is often the case that problem-specific constraints apply, and therefore the encoding scheme must be tailored by applying specialized algorithms to represent valid solutions. Furthermore, the fitness evaluation function is always problem dependent and has to be designed according to the encoding chosen for the solutions.

With this in mind, we adopted a high-level synthesis (HLS) design methodology to construct the problem-specific blocks (PEs and cGAC) by specifying the hardware behaviour using a high-level programming language (e.g. C++) and translating it to digital circuits with HLS tools. Therefore, to customize the cGAP for a given optimization problem the designer has to program the GA in C++, using a function template and a set of classes and methods that provide the necessary interfaces to the spMEMs and the communication with the cGAC and RNG infrastructures. Although the correct usage of the HLS tools still requires a relevant knowledge of the underlying target hardware (namely, the number and type of resources available and the organization of memory blocks), the designer does not need to design these blocks at the usual logic/register-transfer level (RTL), thus accelerating the design process and opening the customization of the architecture to non-digital designers.

Figure 4 represents an overview of the complete design flow to build the cGAP. Bellow we summarize the main phases of the flow.

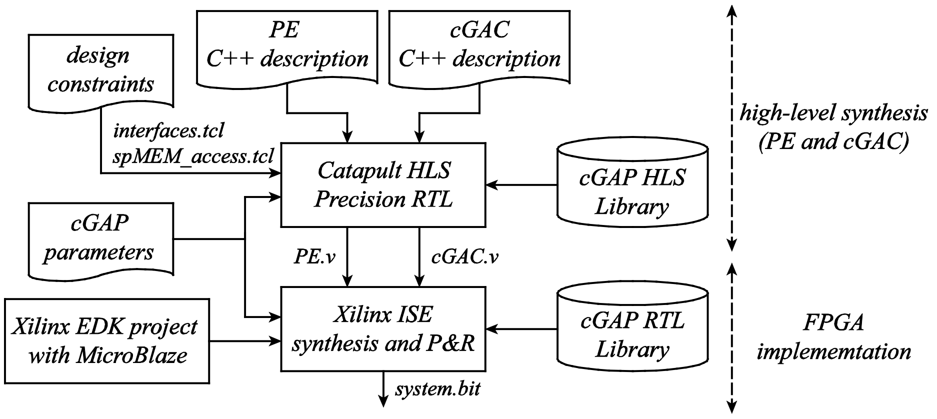

Figure 4: Design flow for building the cGAP

4.1. HLS and FPGA implementation

Starting with a C++ description of the PE and the cGAC, which is done with the help of the *cGAP HLS Library*, the result of the high-level synthesis produces two digital blocks as Verilog RTL modules (PE.v and cGAC.v) that are then integrated in the Xilinx design flow, along with a set of other blocks that are not problem-dependent and are available in the *cGAP RTL Library* (e.g. spMEMs, the communication infrastructure, the global RNG). To complete the design, the user must also define a small set of configuration parameters to set the number of rows and columns of the PE array, the organization of the subpopulation memories, and the number of solutions assigned to each subpopulation. Additionally to the cGAP, the design includes a MicroBlaze soft-core processor that runs a Linux operating system and accesses to the cGAP as a memory mapped device.

The tools used in this work are Catapult HLS version 2010a (University Version) combined with RTL synthesis by Precision RTL version 2010a for the HLS phase, and ISE Design Suite Embedded Edition version 13.4 for the FPGA implementation phase. The target platform is a Xilinx ML605 board that integrates a Virtex-6 FPGA (XC6VLX240T-1).

5. Case study: the Spectrum Allocation Problem

In this work we have used the spectrum allocation (SA) problem in cognitive radio networks to demonstrate the effectiveness of the proposed computing engine and its design methodology. This problem appears in the context of wireless ad hoc networks to improve the usage of the available spectrum by exploiting opportunistically and in real-time unused spectrum.

5.1. Problem formulation

The SA model adopted is described in (Peng, Zheng, and Zhao 2006). This problem consists of a set of N secondary users that try to access to M non-overlapping orthogonal channels. Each secondary user can utilize any channel, but limited to interference constraints that may appear among primary users, which have dedicated channels, and secondary users.

A channel availability binary matrix $L = \{l_{n,m}\}_{N \times M}$ defines when channel m is available to user n by setting $l_{n,m} = 1$. This constraint ensures that a given secondary user does not interfere with a primary user when it is in its range of transmission. A binary matrix $C = \{c_{n,k,m}\}_{N\times N\times M}$ informs when users n and k cannot use simultaneously channel m by setting $c_{n,k,m} = 1$, which happens when the two users are in the range of each other and thus interfere if using the same channel. The SA problem consists on finding the best channel assignment binary matrix $A = \{a_{n,m}\}_{N\times M}$, where $a_{n,m}$ is 1 if channel m is assigned to user n, and 0 otherwise.

The constraints of the problem are defined by the following equations:

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

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

Each element in matrix $B = \{b_{n,m}\}_{N\times M}$ represents the reward to user n when using channel m. These values are related with the maximum distance that a user can transmit using a given channel. The objective of this problem is to maximize a given utility function U. In this work, we consider the maximum sum of rewards, defined by the following equation which also maximizes the spectrum utilization of the system:

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

It has been demonstrated that the previous formulation of the SA leads to a NP-hard problem (Peng, Zheng, and Zhao 2006), and therefore genetic algorithms are a good approach for solving them (Zhao et al. 2009).

5.2. GA operations

Table 2 lists the genetic operations implemented in each PE. Since a solution of the SA problem is defined by the binary matrix A, the binary representation is used to encode the solutions in the GA. Each solution represents a feasible assignment of channels to secondary users. During the evolutionary process, the application of the genetic operators to the assignment matrix A will generate solutions that can change previous channel allocations and thus improve the utility function U.

Therefore, we have used the well-known uniform crossover and bit-flip mutation, applied to the binary representations of solutions. For the selection we have elected two binary tournaments to select two solutions, and in the replacement stage a solution is randomly selected and replaced if the new generated solution has a better fitness value. These operations are often used in cGAs (Alba and Dorronsoro 2008).

Parent selection	binary tournament
Crossover	uniform
Mutation	bit-flip (probability < 5%)
Replacement	select random solution and replace if better

Table 2: Genetic operations used in the PEs to solve the SA problem

Additionally to the genetic operations, the two constraints defined by equations (1) and (2) are properly applied to every new generated solution to make it feasible.

6. Implementation and Results

We have implemented several cGAPs with the PEs forming square arrays, with different levels of parallelism (number of PEs). We then have evaluated the behaviour of the convergence of the algorithm for the different configurations and measured the speedup achieved by the different organizations of the cGAP.

To build the cGAP we start by describing the PE functionality in C++ of the operations needed to generate a new solution, which are described in Section 5.2, which then will be synthesized with the HLS flow referred above. The cGAC (the controller) is built in a similar way, and its behaviour comprises the definition of simple configuration parameters during the set-up phase of the algorithm (e.g. number of secondary users, channels, solutions per memory), controlling the run/stop state of each PE, and retrieve the best solution at the end of the execution of the algorithm. With the PE and cGAC customized for solving this problem, the complete cGAP can then be synthesized and implemented to the target FPGA.

Table 3 shows the implementation results of the cGAP (after placement and routing) for the different array configurations, excluding the MicroBlaze processor. As expected, the different implementations increase the amount of hardware resources required by the cGAP in proportion with the number of PEs. The final design used to evaluate the cGAP includes the MicroBlaze processor using the cGAP as a peripheral and can accommodate an array with a maximum of 5×5 PEs. Although the frequencies reported in Table 3 represent the maximum clock speed supported for each configuration of the cGAP, we decided to run all the designs with a 75 MHz clock to maintain the same clocking synchronization circuit that interfaces with the MicroBlaze processor. In spite of the decrease of clock frequency with the increase of the array size, the performance improvement afforded by the parallelism is clearly advantageous as the number of PEs increases.

	1×1	2×2	3×3	4×4	5×5
Registers	3606 (1.2%)	11159 (3.7%)	23786 (7.9%)	41458 (13.8%)	64239 (21.3%)
LUTs	4097 (2.7%)	13294 (8.8%)	28666 (19.0%)	50348 (33.4%)	78021 (51.8%)
Slices	1740 (4.6%)	4727 (12.5%)	11665 (31.0%)	21056 (55.9%)	29218 (77.5%)
BRAMs	5 (1.2%)	13 (3.1%)	25 (6.0%)	41 (9.9%)	61 (14.7%)
Frequency	120.3 MHz	83.5 MHz	83.0 MHz	82.9 MHz	77.1 MHz

Table 3: Characteristics of the implementations of the cGAP (1×1 PE to 5×5 PEs).
Target FPGA is a Xilinx Virtex-6 (XC6VLX240T-1)

The cGAP is prepared to solve SA instances up to 32 secondary users and 32 channels, with a population of approximately 200 solutions that is obtained by fixing an integer number of solutions per spMEM. Since no known available instances of this problem exist, we have generated a set of instances according to the algorithm for modelling a network conflict graph provided by (Peng, Zheng, and Zhao 2006). The generated instances are labelled with the name *N_M* where *N* is the number of secondary users and *M* the number of available channels. All the results reported with these instances are averaged over 100 independent runs.

Figure 5 illustrates the convergence of the algorithm for a *20_24* SA instance, while running 10×10^6 new generated solutions in all the PEs, thus not reflecting the actual acceleration obtained with the parallel processing by the PEs. It can be observed that the quality of the final solution found by the cGAP is not degraded as the level of parallelism increases, and therefore the cGA supported by our engine is effective as it preserves the quality of the solutions found by the metaheuristic. This behaviour has been also observed in the other instances of this problem and in other problems targeted by this architecture.

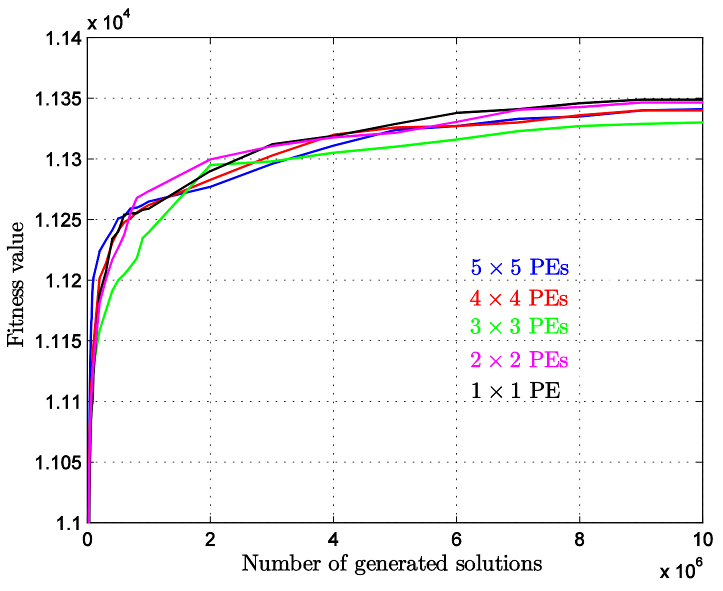

Figure 5: Fitness evolution with the number of generated solutions for an instance 20_24 of the SA problem

To measure the potential acceleration of the cGAP for different levels of parallelism, we have measured the normalized throughput considering 1 PE as reference. **Error! Reference source not found.** presents these results for a SA instance *5_6*. As it can be seen, the throughput of a cGAP is almost directly proportional with the number of PEs. The results show a very small degradation (<0.4%) for the cGAP with the highest number of PEs, which is due to the subpopulation memory collisions that happen during the access to the shared spMEM blocks. Additionally, larger instances are less affected by this, since the cGA selection algorithm, where the collisions occur, has a less computational weight in a generation of a solution. For a *32_32* instance the maximum degradation measured was 0.05% for 5×5 PEs.

Number of PEs	1	4	9	16	25
Normalized acceleration	1	3.999	8.992	15.967	24.906

Table 4: Normalized speedup achieved by the cGAP for the 5_6 instance

We have also measured the quality of the solution obtained by the cGAP at the end of 10^6 generations and compared it against the results obtained with a known heuristic based on a variant of the graph colouring problem, named colour-sensitive graph colouring (CSGC) (Peng, Zheng, and Zhao 2006). Table 5 presents the results of the fitness values obtained by the different arrays together with the CSGC heuristic. Although this comparison was done at a number of iterations far from settling the fitness value (see Figure 5), the cGAP achieves a similar or even superior quality when compared to the results obtained with the heuristic (a higher value means a better result). Although it is not our goal to tune the genetic algorithm parameters for this particular problem, as the crossover and mutation strategies, the results observed already show promising results as the algorithm provides good quality results.

	5_6	*8_16*	*16_16*	*16_32*	*20_24*	*32_32*
CSGC	1130	6090	6959	15197	11209	22882
1×1 CGAP	1130	6101	6965	15236	11268	22584
2×2 CGAP	1130	6117	6951	15234	11275	22645
3×3 CGAP	1130	6121	6939	15207	11254	22754
4×4 CGAP	1130	6112	6942	15192	11232	22746
5×5 CGAP	1130	6113	6946	15198	11265	22757

Table 5: Fitness results obtained for different SA instances with the cGAP and the CSGC heuristic. cGAPs stop at 106 generated solutions

Finally, we have measured the speedup obtained for the 5×5 cGAP when compared to a panmictic (single node) GA running on different processors, including the MicroBlaze soft-core processor implemented together with the cGAP in the same FPGA, an ARM Cortex A8 (in the BeagleBone Black embedded computer), and a desktop computer with an Intel i7 processor. All the software versions were compiled with `gcc -O3`. The results are summarized in Table 6, where the execution times shown for the cGAP where obtained running the algorithm for 10^6 iterations. Even for the most powerful i7 processor the cGAP runs 7 times faster and the acceleration factor is in the range of a few thousands comparing to the embedded MicroBlaze. These results show a clear acceleration potential of custom designed parallel processing for genetic algorithms, even using a high-level synthesis design methodology to build digital circuits from C++ specifications.

	exec. time (ms)	cGAP acceleration		
	cGAP 5×5 PEs 75 MHz	MicroBlaze 150 MHz	ARM Cortex A8 1GHz	Intel i7 3.4GHz
5_6	82	2086	163	7
8_16	139	3657	323	13
16_16	331	3050	292	16
16_32	349	5296	484	32
20_24	464	3909	373	24
32_32	955	4781	403	32

Table 6: Execution times in the cGAP and acceleration figures for a 5×5 cGAP array

7. Conclusions

In this paper we presented a reconfigurable custom processor array that supports the execution of cellular genetic algorithms. The engine is formed by a cellular-like structure where various identical processing elements are interconnected in a 2D matrix, sharing independent memory blocks among them. The genetic operations of the metaheuristic are applied in parallel by each PE to subsets of solutions distributed by the shared memories. To facilitate the configuration of the cGAP for solving different optimization problems, a high-level synthesis design flow is used where the problem-specific operations of the algorithm are synthesized to logic hardware from a C++ functional specification, using commercial HLS tools.

Using the proposed engine and its design methodology, we have presented the implementation of a genetic algorithm metaheuristic for solving a spectrum allocation problem. Results have shown relevant acceleration figures and a throughput that is (almost) directly proportional to the number of PEs, while ensuring that the quality of the generated solutions is not degraded with the parallelization of the algorithm.

Besides the potential for accelerating the execution of genetic algorithms, the framework presented in this work also eases the task of configuring the hardware platform to different optimization problems, thus making the design process accessible to software engineers with limited background on digital design.

References

Ahmadi, Fariborz, Reza Tati, Soraia Ahmadi, and Veria Hossaini. 2011. “New Hardware Engine for Genetic Algorithms.” In *International Conference on Genetic and Evolutionary Computing.*, 122–26.

Alba, Enrique, and Bernabé Dorronsoro. 2008. *Cellular Genetic Algorithms*. Vol. 42. Springer.

Deliparaschos, K M, G C Doyamis, and S G Tzafestas. 2008. “A Parameterised Genetic Algorithm IP Core: FPGA Design, Implementation and Performance Evaluation.” *International Journal of Electronics* 95 (11). Taylor & Francis: 1149–66.

dos Santos, Pedro Vieira, José Carlos Alves, and João Canas Ferreira. 2012. "A Scalable Array for Cellular Genetic Algorithms: TSP as Case Study." In *International Conference on Reconfigurable Computing and FPGAs (ReconFig)*, 1–6.

Fernando, Pradeep R., Srinivas Katkoori, Didier Keymeulen, Ricardo Zebulum, and Adrian Stoica. 2010. "Customizable FPGA IP Core Implementation of a General-Purpose Genetic Algorithm Engine." *IEEE Transactions on Evolutionary Computation.* 14 (1). IEEE: 133–49.

Guo, Liucheng, David B. Thomas, and Wayne Luk. 2014. "Automated Framework for General-Purpose Genetic Algorithms in FPGAs." In *Applications of Evolutionary Computation*, 714–25. Springer.

Holland, John H. 1975. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* University Michigan Press.

Jelodar, Mehdi Salmani, Mehdi Kamal, Sied Mehdi Fakhraie, and Majid Nili Ahmadabadi. 2006. "SOPC-Based Parallel Genetic Algorithm." In *IEEE Congress on Evolutionary Computation.*, 2800–2806.

Nambiar, Vishnu P., Sathivellu Balakrishnan, Mohamed Khalil-Hani, and Muhammad N. Marsono. 2013. "HW/SW Co-Design of Reconfigurable Hardware-Based Genetic Algorithm in FPGAs Applicable to a Variety of Problems." *Computing* 95 (9). Springer: 863–96.

Peng, Chunyi, Haitao Zheng, and Ben Y Zhao. 2006. "Utilization and Fairness in Spectrum Assignment for Opportunistic Spectrum Access." *Mobile Networks and Applications* 11 (4). Springer: 555–76.

Shackleford, Barry, Motoo Tanaka, Richard J. Carter, and Greg Snider. 2002. "FPGA Implementation of Neighborhood-of-Four Cellular Automata Random Number Generators." In *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-Programmable Gate Arrays*, 106–12.

Tachibana, Tatsuhiro, Yoshihiro Murata, Naoki Shibata, Keiichi Yasumoto, and Minoru Ito. 2006. "General Architecture for Hardware Implementation of Genetic Algorithm." In *IEEE Symposium on Field-Programmable Custom Computing Machines.*, 291–92.

Tsai, Ching-Chih, Hsu-Chih Huang, and Cheng-Kai Chan. 2011. "Parallel Elite Genetic Algorithm and Its Application to Global Path Planning for Autonomous Robot Navigation." *IEEE Transactions on Industrial Electronics.* 58 (10). IEEE: 4813–21.

Vavouras, Michalis, Kyprianos Papadimitriou, and Ioannis Papaefstathiou. 2009. "High-Speed FPGA-Based Implementations of a Genetic Algorithm." In *International Symposium on Systems, Architectures, Modeling, and Simulation.*, 9–16.

Zhao, Zhijin, Zhen Peng, Shilian Zheng, and Junna Shang. 2009. "Cognitive Radio Spectrum Allocation Using Evolutionary Algorithms." *IEEE Transactions on Wireless Communications* 8 (9). IEEE: 4421–25.