An FPGA Framework for Genetic Algorithms: Solving the Minimum Energy Broadcast Problem

Pedro Vieira dos Santos, José Carlos Alves, João Canas Ferreira
INESC TEC - INESC Technology and Science
and FEUP - Faculty of Engineering, University of Porto
Porto, Portugal
email: pedro.vieira.santos@fe.up.pt, jca@fe.up.pt, jcf@fe.up.pt

***Abstract*—Solving complex optimization problems with genetic algorithms (GAs) with custom computing architectures is a way to improve the execution time of this metaheuristic, which is known to consume considerable amounts of time to converge to final solutions. In this work, we present a scalable computing array architecture to accelerate the execution of cellular GAs (cGAs), a variant of genetic algorithms which can conveniently exploit the coarse-grain parallelism afforded by custom parallel processing. The proposed architecture targets Xilinx FPGAs and is used as an auxiliary processor of an embedded CPU (MicroBlaze). To handle different optimization problems, a high-level synthesis (HLS) design flow is proposed where the problem-dependent operations are specified in C++ and synthesised to custom hardware, thus requiring a minimum knowledge of digital design for FPGAs. The minimum energy broadcast (MEB) problem in wireless ad hoc networks is used as a case study. An existing software implementation of a GA to solve this problem is ported to the proposed computing array to demonstrate its effectiveness and the HLS-based design flow. Implementation results in a Virtex-6 FPGA show significant speedups, while finding solutions with improved quality.**

I. Introduction

Genetic algorithms (GAs) are metaheuristic search methods inspired by the evolution of living species, where the principles of natural selection and genetics are applied to solve optimization problems. The algorithm have become one of the best known and widely accepted metaheuristics used to solve complex optimization problems, like applications in clustering in data mining and bioinformatics [1], path planning for autonomous navigation [2], antenna design [3], or energy minimization in wireless ad hoc networks [4]. Although GAs, can effectively explore a huge search space of a problem, as it happens with NP-hard problems, only by examining a small fraction of it, they usually suffer from long execution times as the evolutionary process of the algorithm needs to be repeated numerous times. Thus, solving optimization problems where time constraints apply or when the processing power is restricted can, in practice, constraint the utilization of GAs.

This paper presents a general framework for building a custom computing architecture that accelerates the execution of GAs using field-programmable gate arrays (FPGAs) devices. A scalable hardware architecture that supports the execution of cellular GAs (cGAs) is presented, where the population evolved by the metaheuristic is distributed over several independent memories shared by an array of processing elements (PEs). As a result, the level of parallelism of the engine can be increased with the size of the array without adding memory access bottlenecks that would compromise the gain of performance of the hardware.

Although GAs are often associated to a representation of solutions using a binary encoding, where each bit behaves as a gene, more complex representations usually need to be used, for example, to encode solutions for graph problems. As a result, the basic genetic operators (crossover and mutation) must be adapted for each specific representation of solutions. Additionally, it is often the case that the encoding scheme used to represent a valid solution requires the application of specialized algorithms to maintain the feasibility of solutions. To cope with this, we propose a design flow where the problem-dependent operations of the metaheuristic are specified in C++ and translated to digital hardware with high-level synthesis (HLS) tools. Therefore, the architecture can be rapidly customized to build a custom engine specialised to successfully solve different problems.

To demonstrate the effectiveness of the proposed engine and design methodology, the minimum energy broadcast (MEB) problem is explored as a case study. This is a relevant problem in the domain of wireless ad hoc networks and to construct the custom engine for this application we have ported an existing software implementation of a GA [4]. As referred in this work, solving the MEB problem with a genetic algorithm poses several challenges like the need for specific encoding of solutions or the need to complement the canonical GA with local search procedures for better performance of the metaheuristic. Our proposal of using a HLS-based design flow to implement GAs successfully deals with all these issues, where other approaches that rely on a predefined set of templates for the genetics-inspired operators may not be usable.

The complete hardware infrastructure that supports the execution of cGAs is named cGA processor (cGAP), and is implemented in a Xilinx Virtex-6 FPGA. Two versions of the cGAP are built using different intensities in the local search procedure and implementing arrays with 5×5 PEs and 4×4 PEs. We detail the algorithms implemented in the engines and compare the results obtained by the cGAPs with the original work. Accelerations between $21.8\times$ and $2.2\times$ are obtained while ensuring superior quality of solutions found by the

978-1-4673-8035-5/15 $31.00 © 2015 IEEE
DOI 10.1109/DSD.2015.81

engine.

The paper is organized as follows. Section II presents a background about GAs and relevant hardware architectures to implement this metaheuristic. In Section III the cGAP is presented together with its design flow. The MEB problem is introduced in Section IV and a combination of a genetic algorithm and a local search (a memetic algorithm), is presented in Section V including details of the mechanisms to encode and evolve the solutions. Hardware implementation and results are discussed in Section VI. The paper concludes with Section VII.

II. Background

The GA is a population-based metaheuristic where a population, which is a set of tentative solutions of the optimization problem to be solved, goes through an evolutionary process inspired in the biological evolution of living species [5]. The algorithm starts with a population P where all the solutions are evaluated by a *fitness* function that quantifies their quality according to the objective function of the optimization problem. The algorithm proceeds with an iterative process where genetics-inspired operations are applied. First, a *selection* of solutions in P is performed to elect solutions that will undergo some transformations to create new solutions. Typically, the selected solutions are called *parents* and are combined (usually in pairs) to generate new ones through a *crossover* operation. Then, the new solutions (*child*) may suffer a *mutation* operation that induces small changes to them. The generated solutions P' are then evaluated and the population for the next generation is elected among the solutions in P and P', according to a strategy that takes into account the fitness values to promote the evolution towards a better population.

A continuous research activity has been carried out over the last 20 years to implement custom computing architectures in FPGAs for GAs aiming to accelerate its execution. There are various approaches in what concerns the variant of the GA implemented, and the way the engines can be configured to handle different optimization problems.

Several attempts have been made to built a GA general engine that can be applied to different optimization problems. However, it is clear that at least one operation of the algorithm needs to be customized: the fitness evaluation. Moreover, it is often the case that a GA requires special representation of solutions, besides the usual binary codification, as for example to encode a valid path in a graph. Additionally, some optimization problems may require specific procedures to compensate infeasible solutions that may be generated by the genetic operators. Therefore, dedicated hardware architectures for GAs that aim to be general enough to handle any optimization problem, only succeed for problems where the straightforward binary solutions encoding can be used.

Most of the works target a *panmictic GA*, which is the best known GA where any given solution can interact with any other during the evolutionary process. Therefore, the population needs to be kept in a single memory that is accessed by the processing units that compute the operations of the algorithm. One approach is to generate a new population at each iteration of the algorithm (*generational GA*), where it is possible to have several parallel units that access to the population and generate new solutions [6], [7]. Nevertheless, the access to the memory that keeps the population may represent an important bottleneck when the level of parallelism increases. On the other hand, it is possible to generate a single solution in a generation of the GA (*steady-state GA*), for which a single processing unit is sufficient to compute the solution. This approach has led to efficient pipelined architectures, although the operations of the algorithm cannot be parallelized as in the generational GA [8], [9].

Contrasting to the panmictic GA, in a *structured GA* the population is somehow decentralized, resulting that a given solution can only be combined with a limited set of other solutions [10]. Therefore, it is possible to have a custom computing architecture where several processing units have their own memories that hold subsets of the population. With the *distributed GA* model the population is spread in islands that evolve autonomously, and occasionally exchange solutions among them. The work presented in [11] adopts this strategy, using 4 processing nodes to solve the travelling salesman problem and the knapsack problem. The *cellular GA* (cGA) is another structured GA where the solutions are distributed over a regular grid and the operations of the algorithm are only applied to certain overlapped subsets of solutions. Although the cGA exhibits a great potential to be accelerated with dedicated computing architectures, it has not been an active subject of research by other authors.

In this work, we use a previously developed scalable computing array architecture that supports the execution of cGAs with a performance that is directly proportional to the number of processing nodes [12]. We propose a HLS-based design flow, where the problem-specific operations of the algorithm are customized, thus allowing a rapid specification of the architecture with the desired level of parallelism, to solve different optimization problems. To demonstrate the effectiveness of the design flow, we port an existing software GA implementation to solve the minimum energy broadcast problem, which requires special procedures to encode a valid solution and additional search heuristics to further improve the optimization process.

III. A custom computing for accelerating cGAs

In this section we propose a custom computing machine for accelerating GAs, in particular cGAs, which we call cellular genetic algorithm processor (cGAP).

A. The cGAP architecture

Figure 1 depicts an overview of the cGAP architecture composed by the array of processing nodes and a controller that monitors the evolutionary process of the metaheuristic and implements the interface with the host processor.

1) cellular genetic algorithm array (cGAA): The cGAA is responsible to implement the execution of the cellular genetic algorithm and thus it is the core block of the complete processor. It is formed by a regular structure of processing

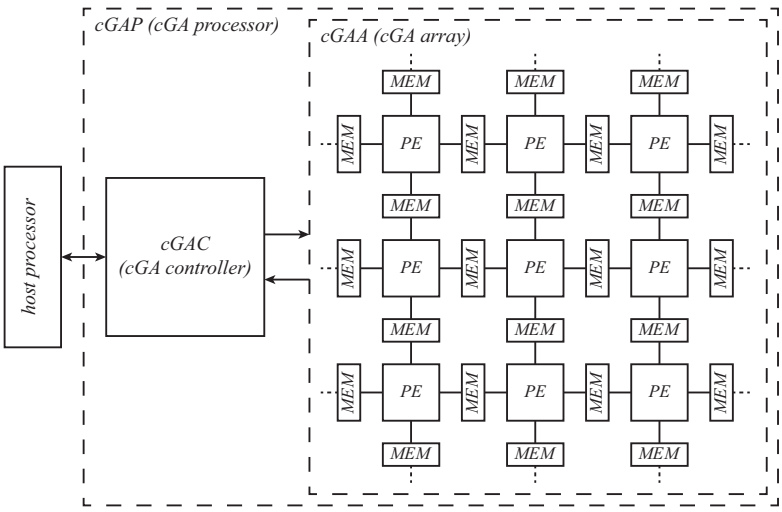

Fig. 1. Overview of the cGAP architecture.

elements (PEs) and memories, where each PE connects to four memories and each memory is shared by two PEs, as shown in Figure 1. A memory holds a subset of solutions (or subpopulation) of the algorithm, and is thus called subpopulation memory (spMEM). A PE is responsible to apply the genetics-inspired operations of the metaheuristic to the solutions presented in their local spMEMs, implementing a local genetic evolution with those solutions. This configuration of PEs and spMEMs, where each spMEM is shared by two PEs, implements a cellular genetic algorithm since the solutions' information is naturally spread throughout the whole population (all the spMEMs), those imposing an implicit mechanism of migration of solutions.

One of the main advantages of the proposed architecture is its scalability since the number of PEs (and associated spMEMs) can be adjusted to ensure a trade-off between throughput, and consequent acceleration of the algorithm, and the number of hardware resources required to implement it. Therefore, for a given population size, the parallelism level can be increased by introducing more PEs in the array while decreasing the number of solutions per spMEM. To keep the regularity of the array, and guarantee a regular shape, only entire rows or columns of PEs (and spMEMs) can be added to adjust the parallelism level, while not introducing relevant memory access bottleneck since a PE always accesses to its four local spMEMs. Indeed, this is a clear advantage over other custom computing architectures for GAs where the population is kept in a single memory, thus constraining the parallelism due to limited memory bandwidth.

Furthermore, the architecture of the cGAA is suitable for an implementation in current FPGAs as these devices possess large amounts of independent memory blocks, with dual-port capabilities that can be used to implement the spMEMs of the cGAP.

2) cellular genetic algorithm controller (cGAC): This block controls and monitors the execution of the cGA supported by the engine, and configures any parameter needed for the execution of the algorithm. To accomplish that, the cGAC performs two main tasks: it communicates with each PE through a dedicated communication infrastructure to send and receive commands that are used to monitor and configure them as desired; and it interprets and executes commands received from the exterior (e.g. a host processor) so that the cGAP can be controlled externally.

For example, the host can start by sending a set of configuration parameters that are interpreted by the cGAC to configure itself and the PEs and then to start the cGA in all the PEs. During the evolutionary process of the metaheuristic, the cGAC keeps track of which PE has the best solution, and how many generated solutions have been computed in all the PEs. At the end of the algorithm, the cGAC requests from the appropriate PE node in the array the best solution found, and sends it back to the host processor.

This module also contains a global random number generator that feeds all the PEs with a random sequence of bits from which the PEs extract the random numbers required for the execution of the genetic algorithm.

B. Design flow

We consider in the following examples a reference design that includes a MicroBlaze soft-core processor as the host processor, targeting a Xilinx ML605 board that integrates a Virtex-6 FPGA (XC6VLX240T-1) [13]. The design flow proposed in this work aims to organize the development process of the cGAP to ease its customization for different optimization problems. To achieve this goal, we use a high-level synthesis design flow (Fig. 2), where the custom digital systems to implement the problem-specific operations are specified with conventional programming languages (C++) and translated automatically to Verilog hardware description language (HDL). Therefore, the main core of the flow is the specification of the PE and the cGAC, which are the hardware

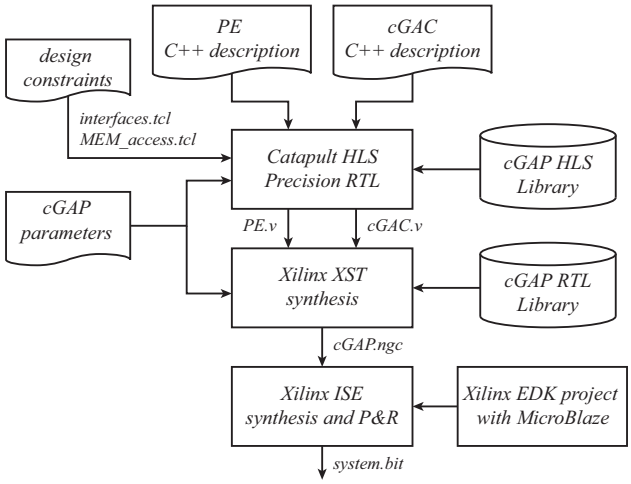

Fig. 2. Design flow of the cGAP.

blocks that need to be adapted to each problem's requirements, using HLS methodologies. The rest of the architecture, which is not specific of the optimization problem, is described as a parameterizable module in Verilog HDL.

A new design starts by defining a small set of parameters that configure the cGAP architecture as desired. This allows specifying the size and aspect ratio of the PE array (which sets the parallelism level of the engine), the configuration of the subpopulation memories (the spMEMs), the maximum number of solutions that a subpopulation memory can handle, and the definition of all the control commands used in the communication between the cGAC and the PEs.

The next step is the specification of the PE and cGAC using HLS techniques. Two independent projects must be created (for the cGAC and the PE), where the corresponding algorithms are specified in C++, following a design template and according to a set of coding guidelines imposed by the HLS tool. Both projects share a *cGAP HLS library* that contains two C++ classes created specifically to aid in the description of the hardware modules, mainly to handle the commands that are sent between the PEs and the cGAC, and to ensure a correct arbitration in the access to the shared spMEMs. We have used Catapult HLS version 2010a (University Version) from Calypto Design Systems, combined with RTL synthesis by Precision RTL version 2010a from Mentor Graphics.

After generating the cGAC and the PE hardware description files, the complete cGAP is built. To do so, a *cGAP RTL library* is used that contains all the remaining files that describe, in Verilog HDL, the cGAP and do not need to be customized for different problems. In this phase, the cGAP is synthesized with the XST tool from Xilinx (ISE version 13.4) to integrate all the hardware blocks required to build the complete cGAP.

In the last phase of the design flow, the cGAP is integrated with a Xilinx EDK design that includes a MicroBlaze processor running Linux, using the cGAP as a memory mapped peripheral. The ISE design flow is used to implement the complete project provided by EDK, including the netlist describing the cGAP block. In a typical session to use the cGAP, the MicroBlaze is remotely accessed from a personal computer using a remote shell and a software application that controls the cGAP.

IV. Case study: minimum energy broadcast

The minimum energy broadcast (MEB) problem is an optimization problem that appears with the use of the wireless networks. This problem consists in minimizing the global energy consumption of a set wireless devices (nodes) when one of them needs to broadcast a message to all the remaining nodes in that network. Unlike in a wired network, in a wireless network a single transmission can reach several nodes, which means that when a node i transmits to node j, all the other nodes located near to i will also receive the transmission. Therefore, it is possible to increase/decrease a transmission range of a node so that more/less nodes are covered, which reflects in the total energy used by the system. The MEB problem is NP-hard [4] and thus metaheuristic procedures are good approaches to solve it.

The problem is specified as follows: a direct graph $G = (V, E)$, represents a set of wireless nodes V, and the edges $e_{i,j} \in E$ represent the energy required to transmit from node i to node j; given a source node $s \in V$ that has to broadcast a message to all the other nodes in V, find the minimum transmission energy used by all nodes in the network, considering that all nodes can work as repeaters. This can be accomplished by creating an arborescence[1] of G rooted at s, where an edge $e_{i,j} \in E$ included in this arborescence defines that node i must transmit to node j with a energy requirement equal to the cost of that edge. If a node transmits to more than 1 node, the cost associated with that node is defined by the maximum of its adjacent edges considered in this arborescence. The MEB problem consists in minimizing the total energy required to broadcast a message from s to all nodes in V, what is equivalent to find an arborescence $T \subseteq E$ rooted at s that minimizes:

$$\sum_{i \in V} \max_{e_{i,j} \in T} d_{i,j}^{\alpha} \tag{1}$$

where $d_{i,j}$ is the Euclidean distance between nodes i and j, and α is the channel loss exponent that takes a value in the range of $2 \leq \alpha \leq 4$ depending on the characteristics of the communication medium. This function is the addition of the energies required for each node to reach the most distant node, among its adjacent nodes considered in that arborescence.

Considering $\alpha = 2$, as it is typically done, and knowing that the $d_{i,j}$ is given by:

$$d_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{2}$$

[1] An arborescence in graph theory is a directed graph in which a vertex u, called the root, and any other vertex v there is exactly one directed path from u to v.

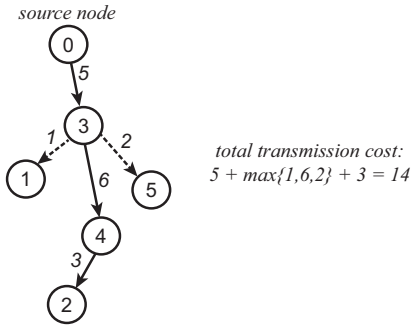

Fig. 3. Example of a MEB solution with 6 nodes where solid edges represent the transmission costs and dashed edges represent implicit transmissions.

where (x_i, y_i) and (x_j, y_j) are the coordinates of nodes i and j respectively, the objective function of the MEB is now to minimize:

$$\sum_{i \in V} \max_{e_{i,j} \in T} (x_i - x_j)^2 + (y_i - y_j)^2 \quad (3)$$

Figure 3 presents an example of a MEB solution with 6 nodes. As it can be seen, the source node (root) is node 0 and it transmits to node 3. In turn, node 3 transmits to node 4, and since nodes 1 and 5 are closer to 3 than node 4, they are implicitly covered by the transmission of node 3, as it is depicted by the dashed edges. Finally, node 4 transmits no node 2 to complete the arborescence of the MEB solution.

V. A GENETIC ALGORITHM FOR THE MEB PROBLEM

The algorithm used to solve the minimum energy broadcast problem has been adapted from [4], where a hybrid genetic algorithm is used. This consists in a GA complemented with a local search procedure (or heuristic) that tries to improve the quality of all generated solutions during the evolutionary process of the algorithm. (Some authors refer to this metaheuristic as *memetic algorithm*.) In this section we will describe the algorithm used and how it has been adapted to the cGAP to solve the MEB problem.

A. Codification of solutions

As in any GA, a solution must be encoded to represent a tentative solution of the optimization problem, so that the genetics-inspired operators are applied. In [4] the path representation (or permutation encoding) is used to represent an MEB solution. Since this encoding scheme consists in a list of unique elements, it cannot be used directly to represent a solution (an arborescence), and therefore a decoder is used to transform the solution representation to a valid MEB solution.

Figure 4 depicts an example of a MEB solution coded with the path representation and the corresponding arborescence with the physical placement of the nodes. The decoder algorithm starts by introducing the source node (node 0) in the arborescence as the first leaf node. This is not included in the path representation because the source node is always the first node in all solutions. Then, the first node in the list (node

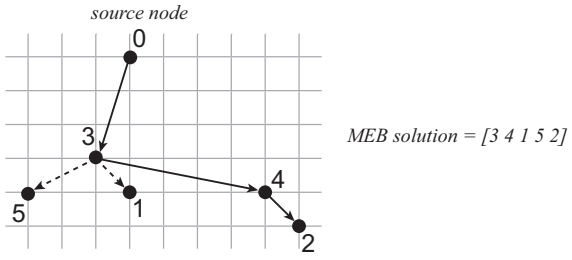

Fig. 4. GA solution representation and codification used in the MEB problem.

Algorithm 1 Pseudo-code of a MEB solution decoder and fitness evaluation

```
solution                          ▷ GA's MEB solution to be decoded and evaluated
leaves ← [source_node]            ▷ list with all non-transmitting nodes
non_leaves ← [ ]                  ▷ list with all transmitting nodes
fitness ← 0
i ← 0
while not all nodes of solution processed do
    sel_node ← solution[i]
    sel_leaf ← select in leaves node that leads lower transmission
    update leaves and non_leaves
    fitness ← fitness+ transmission cost from sel_node to sel_leaf
    for all nodes in solution not processed do
        if node covered by transmission from sel_node to sel_leaf then
            update leaves
        end if
    end for
    increment i to next node in solution not processed
end while
```

3) is introduced in the arborescence that, at the beginning, is formed only by a leaf node represented by the source node. Node 3 is then removed from the list and in this case there are no more nodes implicitly covered by the transmission $0 \rightarrow 3$. Then the first node of the remaining list is chosen (node 4) and it is inserted in the arborescence so that a current leaf node (node not yet transmitting) will transmit to the selected node with the minimum cost, which is node 3 (node 4 is closer to node 3 than node 0). The transmission $3 \rightarrow 4$ is formed and implicitly covered nodes are just removed from the list (for the example nodes 1 and 5, as the transmission $3 \rightarrow 4$ already covers the transmissions $3 \rightarrow 1$ and $3 \rightarrow 5$). Finally the transmission from node 4 to 2 is formed with the same procedure. Algorithm 1 presents the pseudo-code of the MEB solution decoder that has been implemented in the PEs of the cGAP. This algorithm is also used to evaluate the fitness value of the solution by adding the transmission costs of the nodes.

B. Local search heuristic: r-shrink

The local search heuristic applied to the solutions generated by the crossover and mutation operations is the *r*-shrink, which is based on *shrinking* the transmission energy of a node so that the arborescences disconnected from the original solution (nodes to which the broadcast is not performed due to the shrinking) are reassigned to other nodes by increasing their transmission energy. If the reduced energy is superior to the incremented energy for the resulting assignment, a better solution is obtained. The *r* in the name *r*-shrink means the number

Algorithm 2 Pseudo-code of 1-shrink local search for the MEB problem

```
1: non_leaves ▷ list with all transmitting nodes (created by Alg. 1)
2: fitness ▷ MEB fitness value (created by Alg. 1)
3: number_nodes ▷ Total number of nodes
4: total_fitness_gain ← 0
5: while true do
6:     improvement_found ← 0
7:     for all nodes in non_leaves (from last element to first) do
8:         node_reduce ← node from non_leaves
9:         node_sel ← node to which node_reduce transmits
10:        node_improvement_found ← 0
11:        for i ← 0, number_nodes − 1 do
12:            if can node i transmit to node_sel then ▷ avoid cycles
13:                calculate dec_cost and inc_cost
14:                if inc_cost < dec_cost then
15:                    node_improvement_found ← 1
16:                    if best gain cost of all i nodes then
17:                        node_gain ← dec_cost − inc_cost
18:                    end if
19:                end if
20:            end if
21:        end for
22:        if node_improvement_found then
23:            improvement_found ← 1
24:            total_fitness_gain ← total_fitness_gain + node_gain
25:        end if
26:    end for
27:    if not improvement_found then
28:        break
29:    end if
30:    update non_leaves
31: end while
32: fitness ← fitness − total_fitness_gain
```

of reduction steps performed by the algorithm. Therefore, in the 1-shrink and 2-shrink for each transmitting node a reduction on the energy is performed so that, respectively, 1 or 2 nodes (with their corresponding arborescences) are left for reassigning.

Algorithm 2 shows the pseudo-code of the 1-shrink procedure implemented in the cGAP. The algorithm starts by selecting one node that is transmitting and then it finds the best possible move that leads to the minimum energy increase. At the same time, it verifies if this move improves the solution fitness. If an improvement occurs, the algorithm starts again; if not, another transmitting node is selected for the same procedure. The algorithm stops when it cannot find any possible better solution.

The 2-shrink algorithm is identical to the 1-shrink, but instead of moving 1 node from one place to another in the MEB solution, now 2 nodes are moved. As proposed in [4], the 2-shrink starts first with the 1-shrink procedure to improve the solution.

C. The cGA operations

Table I shows all the operations used by the cellular genetic algorithm, and thus implemented in the PEs, that we have used to solve the MEB problem. Both the selection and mutation operators are identical to the original work, as well as the local search heuristic described above. Since the cGAP supports the execution of cellular GAs, we propose to use for the replacement operation a known strategy in this class of GAs,

TABLE I
GENETIC OPERATIONS ADOPTED IN THE PES FOR THE MEB PROBLEM.

Parent selection	probabilistic binary tournament (75 % to accept best solution)
Crossover	maximal preservative crossover (MPX)
Mutation	swap 2 nodes (maximum 3 swaps with probability 75 % each)
Local search	1-shrink or 2-shrink
Replacement	select random solution and replace if better

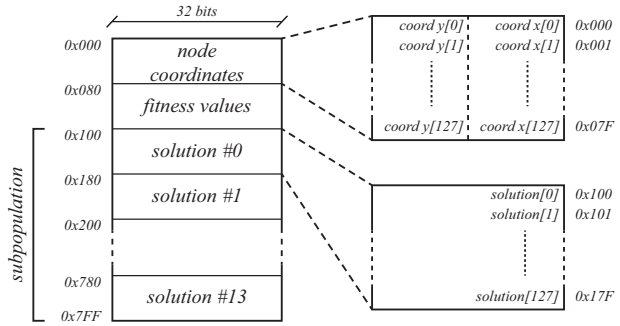

Fig. 5. Subpopulation memory (spMEM) organization in the cGAP.

which consists in randomly select a solution (from all the subpopulations connected to the PE) and replace it with the new generated solution if this has a better fitness value [10].

To evaluate the performance of two different crossover operators adequate for the path representation used to encode solutions, we have implemented the GA in software using the maximal preservative crossover (MPX) [14], and the cycle crossover used originally in [4]. We concluded that the MPX has consistently obtained better results and this was adopted for the crossover operation of the PEs.

VI. IMPLEMENTATIONS AND RESULTS

The PE algorithm has been described in C++ for the Catapult HLS tool. Figure 5 depicts the subpopulation memory organization used to solve the MEB problem, both for the 1- and 2-shrink implementations, which is identical in all the four subpopulation memories that connect to a PE. As it can be seen, we have kept the nodes' coordinates of the MEB problem as additional data besides the solutions. With the node's coordinates replicated in the 4 spMEMs accessed by a PE, we can parallelize the calculation of the squared distance between two nodes (see Eq. (3)), required to compute the fitness function. To keep the subpopulation memories with two memory blocks of the target FPGA (BRAMs of the Virtex-6 family), we have limited our implementation to a maximum of 128 nodes and subpopulations with a maximum of 14 solutions.

Additionally, we have used several C++ arrays (mapped to additional memory blocks) to describe the PE algorithm that are used, for example, to manipulable the MEB solution (an arborescence), to compute the MPX, and to avoid recalculating the distance between two nodes in the 1- and 2-shrink heuris-

TABLE II
CHARACTERISTICS OF THE PROJECTS USED TO IMPLEMENT THE CGAP-1S AND CGAP-2S. TARGET FPGA IS A VIRTEX-6 (XC6VLX240T-1).

	cGAP-1s			cGAP-2s		
	Precision RTL (PE)	ISE synthesis (cGAP - 5×5 PEs)	ISE (cGAP + MicroBlaze)	Precision RTL (PE)	ISE synthesis (cGAP - 4×4 PEs)	ISE (cGAP + MicroBlaze)
Registers	3060 (1.0 %)	90214 (29.9 %)	115759 (38.4 %)	3738 (1.2 %)	68974 (22.9 %)	94695 (31.4 %)
LUTs	3996 (2.7 %)	116093 (77.0 %)	130943 (86.9 %)	5992 (4.0 %)	106437 (70.6 %)	114090 (75.7 %)
Slices	999 (2.7 %)	-	37212 (98.8 %)	1498 (4.0 %)	-	37374 (99.2 %)
BRAMs	5 (1.2 %)	246 (59.1 %)	207 (49.8 %)	7 (1.7 %)	185 (44.5 %)	165 (39.7 %)
DSPs	17 (2.2 %)	425 (55.3 %)	431 (56.1 %)	4 (0.5 %)	64 (8.3 %)	70 (9.1 %)
Frequency	87.6 MHz	80.3 MHz	$>$ 75 MHz	104.8 MHz	95.8 MHz	$>$ 75 MHz

tics. The high-level synthesis performed by Catapult HLS was optimized by pipelining all the innermost loops, resulting in most of the pipelined circuits able to work with an initiation interval equal to 1.

Table II provides the implementation details of the main projects used for implementing the MEB with 1-shrink and 2-shrink local search heuristics, called respectively cGAP-1s and cGAP-2s. The two designs implemented were chosen to maximize the number of PEs (level of parallelism), which was constrained by the logic resources available in the target FPGA. As presented in Table II, the cGAP-1s achieved a maximum of 5×5 PEs, while for the cGAP-2s the maximum array size was 4×4 PEs. Although both implementations support a clock frequency slightly higher than 75 MHz, the final implementation was run with this clock frequency which is convenient for interfacing with the MicroBlaze processor.

The two cGAPs implementations were used to solve the same instances of the MEB problem reported in [4], which consist of two sets of instances, one with 20 nodes and other with 50 nodes, each with a total of 30 different instances. As in the original work, the results are averaged over 30 independent runs for each instance.

The initial populations used in all the experiments have been built offline and have been loaded to the subpopulation memories during the initialization phase of the algorithm. Furthermore, the execution times reported in this work do not include the initialization phase. This configuration was used to evaluate the cGAP and thus it does not reflect a real scenario application, where the initial population can be kept in the on-chip memories of the FPGA, thus avoiding the need to create an initial population.

The original algorithm reported in [4] uses 400 solutions for the population, and a stop criterion of more than $1000 \cdot n$ generated solutions, where n is the number of nodes, without improving the best solution found. Therefore, for the cGAP-1s, which uses an array of 5×5 PEs, a total of 7 solutions is used per subpopulation, which results in a population of 420 solutions. The cGAP-2s, which has 4×4 PEs, requires 10 solutions per subpopulation to achieve exactly a total of 400 solutions. To implement the stop criterion in the cGAPs, the PEs notify the cGAC when they found a local best solution and, periodically, when a certain number of generations has been elapsed. With this information, the cGAC knows what is the best solution in the cGAA and, approximately, how many generations have elapsed in all the PEs since the best solution was found. Therefore, the cGAC realizes when the stop criterion has been met and broadcasts a command to stop all the PEs, thus concluding the iterative process.

TABLE III
RESULTS PERFORMANCE OF THE CGAP-1S AND CGAP-2S. DATA IS COMPARED AGAINST SOFTWARE GA DEVELOPED IN [4].

		20-node	50-node
GA-1s	Excess (%)	0	0.81
	Found	30/30	19.27/30
	Time (s)	0.56	7,49
cGAP-1s	Excess (%)	0	0.63
	Found	30/30	17.37/30
	Time (s)	0.026	0.53
	Acceleration	21.8	15.4
GA-2s	Excess (%)	0	0.25
	Found	30/30	22.9/30
	Time (s)	0.86	13.71
cGAP-2s	Excess (%)	0	0.10
	Found	30/30	25.87/30
	Time (s)	0.134	6.49
	Acceleration	6.7	2.23

Table III presents the results obtained for the two groups of instances of the MEB problem. We provide the average excess which measures, in percentage, the distance that the solutions obtained are from the known optimum, the number of times the optimum solution is found out of 30 trials, and the average execution time. All the results are compared against the figures reported in the original work (GA-1s and GA-2s, respectively for the 1- and 2-shrink), which were implemented in C and executed on a Pentium 4 system running at 3.0 GHz. Additionally, we present the acceleration of the cGAPs with respect to the corresponding software version of the GAs.

As observed from these results, for the 20-node problems the cGAP-1s executes in average $21.8\times$ faster than the GA-1s, whereas the cGAP-2s achieves an acceleration of $6.7\times$, both finding always the optimum solution. For the instances with 50 nodes, the cGAP-1s founds in average the optimum 17.37 times out of 30, while the original algorithm provides a slightly better result of 19.27 out of 30. However, the excess figure is better for the cGAP-1s, with $0.63\,\%$ against $0.81\,\%$ for the software version GA-1s. Therefore, even though the cGAP-1s

achieves less times the optimum, in average it has shown to find solutions with better quality. For the cGAP-2s both the excess and the number of times the optimum is found is in average better than the GA-2s. In this case, the acceleration figures are between $6.7\times$ and $2.2\times$.

To conclude, for the instances analysed our custom processor has demonstrated to achieve solutions with superior quality when compared to the software versions of the GAs. Additionally, the cGAP executes faster than the original software version, running in a CPU of a personal computer. Although these acceleration figures are naturally worst if we consider the same software running in today's personal computers, the potential of acceleration of the proposed cGAP increases significantly if we compare the performance of the custom processor to the software genetic algorithm running in an embedded processor, as for example the soft-core MicroBlaze implemented in the same FPGA technology. Moreover, the scalability of the cGAP allows to take advantage of the larger FPGAs presently available by enlarging the array of processing nodes, thus increasing the level of parallelism and consequently the global throughout of the engine.

VII. Conclusions

In this paper we presented a custom computing array architecture to accelerate the execution of cellular genetic algorithms. The computing engine is formed by a regular array of identical problem-specific processing nodes, sharing a set of independent memory blocks that hold the pool of solutions evolved by the genetic algorithm. To facilitate the implementation and customization of the processing nodes, a high-level synthesis design flow is proposed to synthesize its digital implementation from a C++ functional specification, using commercial high-level synthesis and FPGA back-end tools. The proposed array architecture has demonstrated a performance for solving different genetic algorithms that is almost directly proportional to the number of processing nodes. This is mainly due to the scalability of the distributed memory architecture that provides to each processing node a set of local memories that are only shared with the PEs in its neighbourhood.

Using this architecture, we have successfully ported an existing software implementation of a variant of a genetic algorithm (combined with a local search heuristic, usually called a *memetic algorithm*) to solve an energy minimization problem in wireless ad hoc networks. Results have shown interesting speedup figures compared to the original software version of the same genetic algorithm. Besides, the cellular GA approach implemented by our architecture has also shown to obtain slightly better results in terms of the quality of the solutions found. Although this framework has been designed originally for cellular genetic algorithms, the flexibility of the processor array and the high-level specification of the core processing units make this interesting for building custom processing arrays for other population-based optimization metaheuristics, like particle swarm optimization or differential evolution.

Acknowledgment

This work has been partially funded by the FCT – Fundação 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) within project FCT/DAAD 2010/2011 under reference daad124412622223295.

References

[1] U. Maulik, S. Bandyopadhyay, and A. Mukhopadhyay, *Multiobjective Genetic Algorithms for Clustering: Applications in Data Mining and Bioinformatics*. Springer, 2011.

[2] C.-C. Tsai, H.-C. Huang, and C.-K. Chan, "Parallel elite genetic algorithm and its application to global path planning for autonomous robot navigation," *IEEE Transactions on Industrial Electronics*, vol. 58, no. 10, pp. 4813–4821, 2011.

[3] G. S. Hornby, A. Globus, D. S. Linden, and J. D. Lohn, "Automated antenna design with evolutionary algorithms," in *Proc. AIAA Space Conference*, 2006, p. 8.

[4] A. Singh and W. N. Bhukya, "A hybrid genetic algorithm for the minimum energy broadcast problem in wireless ad hoc networks," *Applied Soft Computing*, vol. 11, no. 1, pp. 667–674, 2011.

[5] J. H. Holland, *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* University Michigan Press, 1975.

[6] L. Guo, D. B. Thomas, and W. Luk, "Automated framework for general-purpose genetic algorithms in FPGAs," in *Applications of Evolutionary Computation*. Springer, 2014, pp. 714–725.

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

[8] M. Vavouras, K. Papadimitriou, and I. Papaefstathiou, "High-speed FPGA-based implementations of a genetic algorithm," in *International Symposium on Systems, Architectures, Modeling, and Simulation.* IEEE, 2009, pp. 9–16.

[9] 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.

[10] E. Alba and B. Dorronsoro, *Cellular genetic algorithms*. Springer, 2008, vol. 42.

[11] 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.* IEEE, 2006, pp. 291–292.

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

[13] Xilinx, "Virtex-6 FPGA ML605 evaluation kit," http://www.xilinx.com/products/boards-and-kits/ek-v6-ml605-g.html.

[14] P. Larranaga, C. M. H. Kuijpers, R. H. 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.