A Scalable Array for Cellular Genetic Algorithms: TSP as Case Study

Pedro Vieira dos Santos, José Carlos Alves, João Canas Ferreira
INESC TEC (formerly INESC Porto)
Faculdade de Engenharia, Universidade do Porto
Porto, Portugal
pedro.vieira.santos@fe.up.pt, jca@fe.up.pt, jcf@fe.up.pt

***Abstract*—Cellular Genetic Algorithms (cGAs) exhibit a natural parallelism that makes them interesting candidates for hardware implementation, as several processing elements can operate simultaneously on subpopulations shared among them. This paper presents a scalable architecture for a cGA, suitable for FPGA implementation. A regular array of custom designed processing elements (PEs) works on a population of solutions that is spread into dual-port memory blocks locally shared by adjacent PEs.**

A travelling salesman problem with 150 cities was used to verify the implementation of the proposed cGA on a Virtex-6 FPGA, using a population of 128 solutions with different levels of parallelism (1, 4, 16 and 64 PEs). Results have shown that an increase of the number of PEs does not degrade the quality of the convergence of the iterative process, and that the throughput increases almost linearly with the number of PEs. Comparing with a software implementation running in a PC, the cGA with 64 PEs has shown a 45x speedup.

I. Introduction

The Genetic Algorithm (GA) is one of the best known techniques for solving complex combinatorial optimization problems. The algorithm is based on the principles of natural selection, where genetic operations are applied to candidate solutions of a problem, so that better solutions are generated [1]. Although it can produce good quality solutions, the algorithm usually requires a large number of iterations. Solving such problems in embedded systems, especially under real-time constraints, suggests the need of custom hardware architectures for accelerating the GA without compromising the quality of the final solutions found. Applications examples are dynamic frequency selection for efficient utilization of the spectrum by cognitive radios [2], and image matching in a recognition system of a workpiece [3].

This paper presents a new hardware architecture based on the cellular Genetic Algorithm (cGA) model, which has been pointed out as a promising area of research in the field of evolutionary algorithms [4]. By splitting the population of the algorithm (the pool of solutions) into subpopulations located in separate memories, several GA processing elements (PEs) can operate in parallel on these subpopulations. Each population memory is accessed by two different PEs, and each PE connects to four neighbour memories. This creates a regular 2D architecture, which efficiently uses the true dual-port memory blocks available in modern Field-Programmable Gate Arrays (FPGAs).

The approach followed in this work exploits the reconfigurability of FPGAs to avoid the trade-offs involved in general-purpose hardware cGAs; instead, a scalable hardware cGA architecture is proposed, which can be tailored to different problems by designing appropriate PEs. The proposed architecture is scalable, as the number of processing elements can be varied, leading to different speedups and hardware resource utilization. To the best knowledge of the authors, this is the first implementation of a canonical cGA on FPGA, although implementations of many other architectures are reported [5].

The paper presents a first evaluation of the proposed hardware architecture by applying it to the well-known Travelling Salesman Problem (TSP) [6]. The design of the problem-specific PE is also described and implementation issues are highlighted.

The paper is organized as follows. Section II presents a review of the genetic algorithm together with related work about hardware implementations. The proposed cGA architecture is presented in Section III, and a description of a GA processing element for the TSP is given in Section IV. Results are presented and discussed in Section V. Section VI presents the concluding remarks.

II. State of the art

A. The Genetic Algorithm

The GA is a population-based metaheuristic where a set of candidate solutions are competing and cooperating with the goal of generating improved solutions for a given problem. A solution is coded in a problem-specific way, so that it can be combined with others to generate a new one. The algorithm repeatedly goes through a *selection* procedure, where solutions from the population (typically two) are chosen so that they can be combined via genetically inspired operators like *crossover* and *mutation* to produce a new child solution. A *replacement* strategy decides whether the child substitutes

This work has been partially funded by National Funds through 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) through the project FCT/DAAD 2010/2011 under reference daad124412622223295.

one of the parents in the population. For that, the quality of a solution is measured by an objective function that quantifies the new solution's *fitness*, The iterative process must ensure that, on average, better solutions are generated during the evolution of the algorithm.

When a single population exists and any solution can interact with any other, the GA is called *panmictic* [7]. In contrast, the *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) [7]. In the first one, the population is divided into separate subpopulations that evolve independently, while in the second one, the solutions are distributed in a regular structure. As the dGA and cGA have several subpopulations/solutions that can evolve simultaneously, they are very attractive for parallel implementations.

B. Hardware for Genetic Algorithms

A considerable number of dedicated hardware architectures for GAs can be found in literature. The work presented in [5] proposes a general-purpose GA engine where several parameters of the algorithm can be specified together with an interface to a custom fitness function. This implementation is based on the *generational* GA, where a new population is created in a single iteration (generation), so that it replaces the old one. Because of that, this approach requires two memories: one for the current population and another other for the new solutions.

Different hardware architectures can be explored by generating a single solution that replaces an existing one in the population at each generation [8]. This approach, called *steady-state* GA, requires less memory and leads to more efficient hardware pipelined architectures [9].

Both generational and steady-state GAs belong to the class of panmictic GAs, which are difficult to parallelize as only one population exists. In contrast, a structured GA leads to a natural parallelization of the algorithm and to an increase in memory access rates as several independent memories can be accessed by different PEs. The distributed GA has been implemented on FPGA [10], but not with a considerable degree of parallelism. As an example, in [11] only a maximum of 4 PEs can run in parallel.

More recently, Graphic Processor Units (GPUs) have been used to implement GAs for solving mixed integer non-linear programming problems [12]. As GPUs have highly parallel structures, they become attractive for implementations of a cGA [13], where a processing element can exist for each solution, thus maximizing parallelization.

Although cGAs represent a great opportunity for speeding up the algorithm execution, they have not been explored regarding their implementation in FPGAs. This paper proposes to address this gap by introducing a new hardware architecture for cGAs on FPGA.

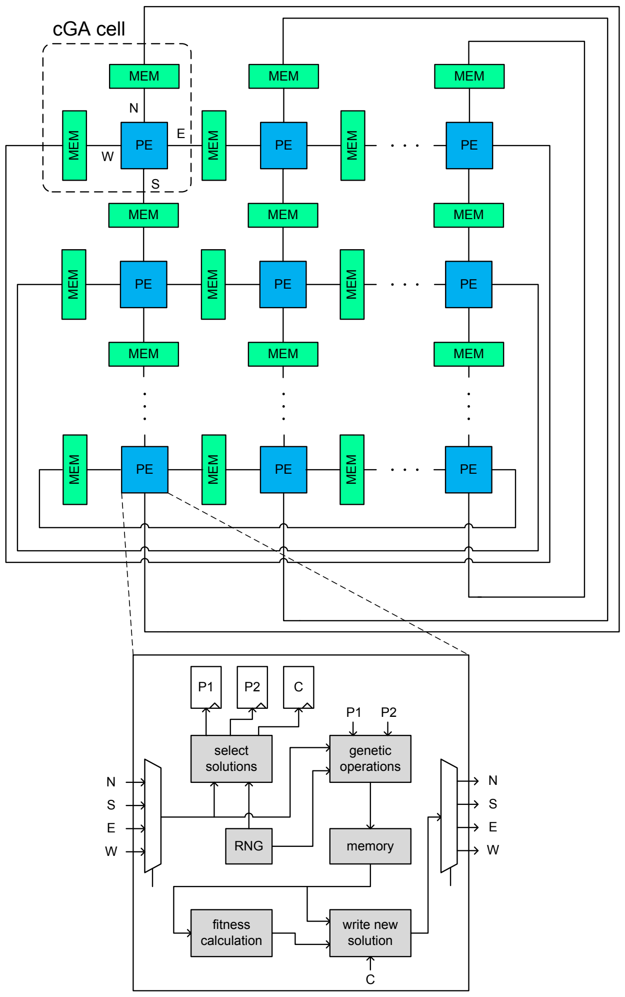


Figure 1. Hardware cellular GA.

III. Hardware Architecture for cGA

A. Main architecture

The proposed overall architecture of the system is shown in Fig. 1. The basic element of the hardware cGA architecture consists of a single PE connected to two memories. This cGA cell is replicated in a rectangular array to build the complete architecture. Each memory keeps a subpopulation of the algorithm and is connected to two different PEs. The architecture assumes a toroidal shape by connecting the opposites sides of the array (both for top-bottom and left-right). The number of subpopulations is twice the number of PEs and since each memory is accessed by two adjacent PEs, it can be efficiently mapped in the dual-port RAM blocks available in current FPGAs.

The proposed architecture is scalable as the number of cGA cells can be increased, subject to resource availability, in order to increase the amount of parallelism. Each PE only accesses its local memories and does not communicate directly with the other PEs, thus the memory access bandwidth is not degraded with the number of PEs.

Increasing the number of cells also allows the total size of the population to increase. However, it is well established that an increase of the size of the population of a GA slows down its convergence, although it may improve the quality of the final solution [1]. If this is not desired, the number of members of each subpopulation can be decreased as the number of cells is increased. This imposes a limit on the parallelism of the hardware as a minimum of one solution can exist in each subpopulation. As an example, for a GA with a population of 32 solutions, the proposed architecture can have a maximum of 32 memories (each with a single solution) and 16 PEs.

B. The GA processing element

The GA processing element hardware should be customized to adapt its functionality to the problem that is being solved. It is possible to have general-purpose GAs that are suitable for a set of optimization problems [5]. However, embedded systems typically solve only one type of problem, since they address a specific set of applications. In addition, resource and time restrictions usually apply. If an efficient and effective optimization procedure is desired, it is not only necessary to adapt the fitness function to the problem in the GA, but also the codification of solutions, as well as performing an adequate selection of the genetic operations [6].

```
while (!(stop criteria)) {
  select solutions
    - 2 for reading // (parents: 'P1' and 'P2')
    - 1 for writing // (child: 'C')
  apply genetic operations // crossover/mutation
  fitness calculation
  write new solution
}
```

Listing 1. Pseudo-code of a PE.

Listing 1 gives the pseudo-code of the operations performed by a PE. An iteration of the algorithm starts by selecting from the 4 subpopulations two solutions for reading (that will be mated) and one for writing (to be replaced by the new generated solution). A proper selection and replacement strategy must be used here. After that, genetic operators are applied to produce a new solution and its fitness is calculated. Finally, the solution is written into the subpopulation memory. These operations are repeated in each PE until a global stop criteria to the cGA is met. Fig. 1 depicts a possible hardware architecture of a PE.

C. Memory access control

Since subpopulation memories are shared between two PEs, simultaneous access to the same solution must be handled carefully. If a PE is writing new data, the other cannot access it, as the contents of that solution will be temporarily incoherent. However, read operations can be performed simultaneously.

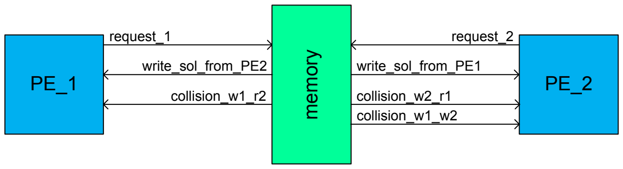


Figure 2. Memory access control among two GA PEs and a subpopulation of the cGA.

In our design, subpopulation memories and PEs collaborate to avoid access collisions. Each memory module keeps track of which of its solutions are being read and written by the two neighbour PEs. This way, each memory informs the corresponding PEs if the selected solutions can be used or not. Figure 2 shows the signals used for memory access control and Listing 2 describes the behaviour of these signals.

```
if (request) { // valid during 'select solutions'
  // requests for reading and writing
  if (read request)
    select solution for read
  if (write request)
    select solution for write
}
if (collision_w_w) {
  // valid during 'select solutions'
  // 2 simultaneous write requests on same solution
  select new solution for writing
}
if (write_sol_from_PE == selected solution) {
  // valid during 'select solutions'
  // selected solution is been written by other PE
  select another solution
}
if (collision_w_r) {
  // valid during 'write new solution'
  // opposite PE is reading from that solution
  write operation must wait till signal deasserts
}
```

Listing 2. Pseudo-code for memory access control signals.

IV. A GA Processing Element for the TSP

This section describes the PE developed for use in the scalable array to solve instances of the symmetric TSP: given a list of cities and their coordinates, determine the shortest tour(s) that visits each city exactly once. A solution is encoded as an ordered list of n cities that represent a tour [6]. The strategy adopted for selection and replacement is based on the steady-state GA presented in [8], where one solution replaces an existing one at each iteration of the algorithm.

At each generation a PE starts by *selecting* two solutions (parents), combine them to generate a new one (child) by applying a *crossover/mutation* operation, evaluate its *fitness*, and *update the population* by substituting the worst parent by the new child if a better fitness has been achieved. These phases are performed in sequence, as shown in Fig. 3, where

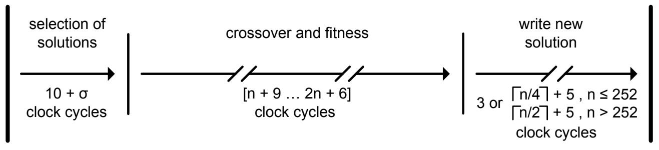


Figure 3. Sequence of operations of a PE in the cGA during a generation of a new solution for solving a TSP with n cities.

the number of clock cycles required for each is presented. The remainder of this section details these operations.

A random number generator (RNG) exists in each PE to feed their selection and crossover modules with random numbers required for their operations. In this work, a 64-bit RNG based on cellular automata (CA) techniques has been adopted (with a ring network with connectivity {-7,0,11,17} and rule 50745, as described in [14]). To promote diversity, the RNGs are initialized with different seeds for each PE.

A. Selection

Using data from the RNG, two solutions are randomly chosen as parents. Their fitness values (which are kept in the subpopulation memory, together with the solution) are read and the solution with the worst fitness is also pre-selected for writing. This write is conditional: the child is only written back to the population if it is better than the worst parent, otherwise the new solution is discarded. Simultaneously, the presence of collisions is checked. If a collision is detected, the process restarts and two new solutions are randomly chosen. The procedure repeats until reservation succeeds.

Although this approach may result in successive restarts of the selection process, it simplifies the hardware design, ensuring that selection is fast when no collisions occur, which is expected to happen more frequently. The experimental results of Sec. V show that collisions do not impact significantly the performance of the algorithm. Selection takes 10 clock cycles when no collisions occur, otherwise it will have an additional overhead (shown as σ in Fig. 3) which accounts for the time lost due to the occurrence of collisions.

B. Crossover and Fitness Calculation

The Maximal Preservative Crossover (MPX) was chosen to implement the crossover and mutation operations [6]. This operator creates connections between two cities of the TSP which are not present on both parents, therefore also acting as a mutation. The operator starts by selecting a random set of contiguous cities from the first parent which are copied to the child. Subsequently, the missing cities in the child are filled in from the second parent, preserving their relative order. The hardware implementation of the MPX operator is based on a 4-stage pipelined architecture that is able to process a city each clock cycle [9].

The evaluation of the fitness function is performed in parallel with crossover as soon as a city of the new solution

Table I
CHARACTERISTICS OF cGA IMPLEMENTATIONS ON A VIRTEX-6 (XC6VLX240T-1) FOR DIFFERENT ARRAY SIZES.

Parameter	1×1	2×2	4×4	8×8
Registers	543 (0.2%)	2020 (0.7%)	7485 (2.5%)	27591 (9.2%)
LUTs	663 (0.4%)	2605 (1.7%)	9306 (6.2%)	34855 (23.1%)
Slices	229 (0.6%)	913 (2.4%)	3727 (9.9%)	13316 (35.3%)
BRAMs	9 (2.2%)	12 (2.9%)	48 (11.5%)	192 (46.2%)
Frequency	186MHz	179MHz	152MHz	122MHz

is written (also using a 4-stage pipelined architecture). The sum of the absolute differences of the coordinates is used as a metric to calculate the value according to equation 1, where (x_k, y_k) represents the coordinates of city k.

$$fitness = \sum_{i=1}^{n-1} \Big(|x_i - x_{i-1}| + |y_i - y_{i-1}| \Big) + \\ + |x_{n-1} - x_0| + |y_{n-1} - y_0| \quad (1)$$

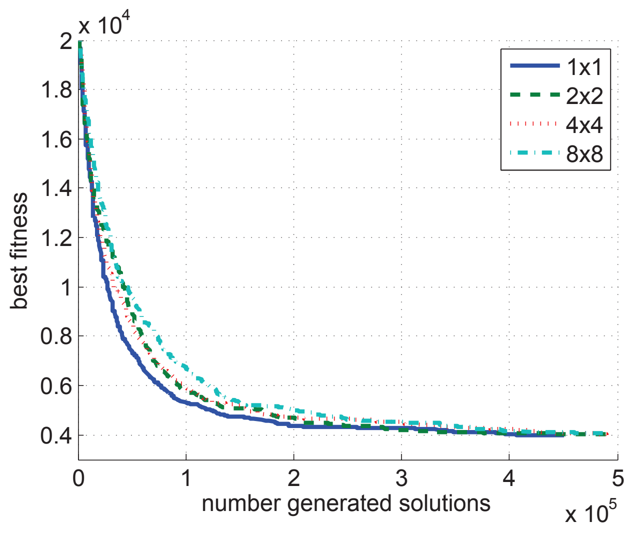
To perform the crossover and fitness calculation, a local dual-port memory is used for saving the new solution generated, and to be used as an auxiliary memory during the MPX operation. Additionally, this memory keeps the cities coordinates for the fitness computation. No stalls exist in the hardware pipelines of the MPX and fitness calculation operations, as all the memory accesses can be performed in parallel.

C. Updating the Population

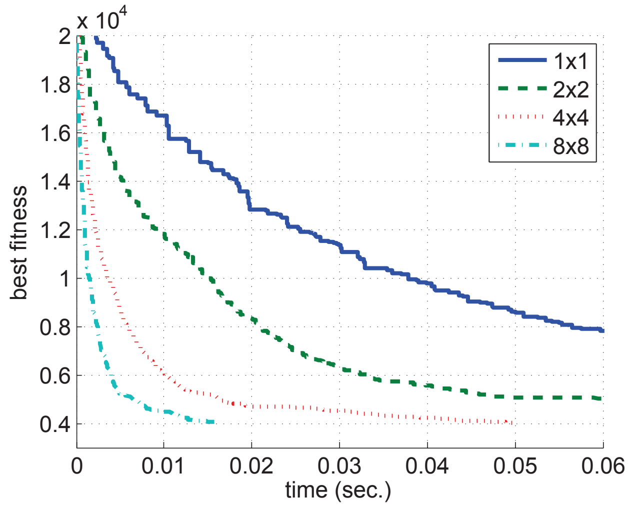
The fitness value of the new solution is compared to the worst parent fitness value determined in the selection step. If the new generated solution has a better fitness, then it is written back to the subpopulation memory, substituting the worst parent; otherwise it is discarded. When the new solution is discarded the update task takes only 3 clock cycles; when a copy is performed, the whole new solution is transferred to the subpopulation memory.

V. IMPLEMENTATION AND RESULTS

The proposed architecture was implemented as a parameterized Verilog HDL model, synthesized using Xilinx ISE 12.4, and successfully run on a ML605 board with a Virtex-6 FPGA (XC6VLX240T-1). This section presents implementation details for an instance capable of solving problems up to 252 cities of the TSP and results of its evaluation with the benchmark *ch150* (with 150 cities) from [15]. For a total population with 128 solutions, PE arrays of size 1×1, 2×2, 4×4 and 8×8 have been evaluated, corresponding to subpopulations of 64, 16, 4 and



(a) Fitness evolution with number of generated solutions.



(b) Fitness evolution with time.

Figure 4. Fitness evolution for different array sizes of the cGA over 500 thousand new generated solutions.

1 solution/memory, respectively. The $1{\times}1$ cGA has only one PE and is equivalent to the panmictic GA, even though it works with two separate subpopulation memories.

Table I presents the implementation results for different array dimensions. As expected, the number of occupied resources increases linearly with the number of PEs. The maximum clock frequency reported by the timing analyzer decreases from 186 MHz (1 PE) to 122 MHz (64 PEs). This happens due to the delays associated with the connections used to build the toroidal shape of the cGA, which become more critical as the number of PEs increases.

Figure 4(a) depicts the fitness evolution over 500 thousand new generated solutions (over all PEs). Although the figure does not reflect the parallelism afforded by the cellular array, it shows that the fitness converges for similar values,

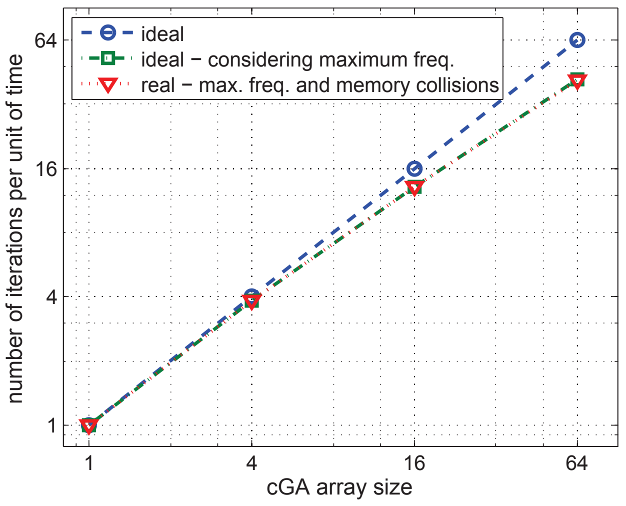


Figure 5. Throughput of the cGA for different arrays sizes, normalized to the throughput of a single PE.

regardless the number of PEs. As can be seen, the arrays with higher number of PEs, especially the $8{\times}8$ case, reveal a slower convergence rate when compared to the simple GA ($1{\times}1$ case). Figure 4(b) shows the fitness evolution as a function of the actual execution time required for the different arrays, using the maximum clock frequency supported by each implementation. This graph presents the overall time improvements obtained by exploiting the parallelism of the cGA, as it includes the effects of a slower convergence rate for larger arrays and clock frequency degradation.

The experimental results show that the $1{\times}1$ array produces an average of 0.75×10^6 solutions/second and for the $8{\times}8$ array the throughput is 31.09×10^6 solutions/second, which represents a $41\times$ throughput increase.

Taking the $1{\times}1$ array as reference, Fig. 5 shows the normalized throughput estimated by considering a proportional increase of the array size (ideal situation) and accounting for the effect of maximum clock frequency degradation, and the actual measured throughput that includes the effects of memory collisions. The data shows that it is mainly the degradation of the clock frequency for larger cGA arrays that limits the speedup, and that there is no significant impact due to memory collisions. The maximum throughput decrease due to this collisions occurs for the $8{\times}8$ cGA array and is less than 0.6 iterations per unit of time.

An equivalent genetic algorithm was developed in C language and run on a single processor personal computer (PC). This software implementation does not explore parallelism and is thus equivalent to the 1×1 cGA. The program was compiled with `GCC -O3` and executed on an Intel T8100 processor running at 2.1 GHz, to solve the same TSP instance. The PC achieved a throughput of only 0.69×10^6 solutions/second, which translates to a $45\times$ speedup for the $8{\times}8$ cGA FPGA implementation.

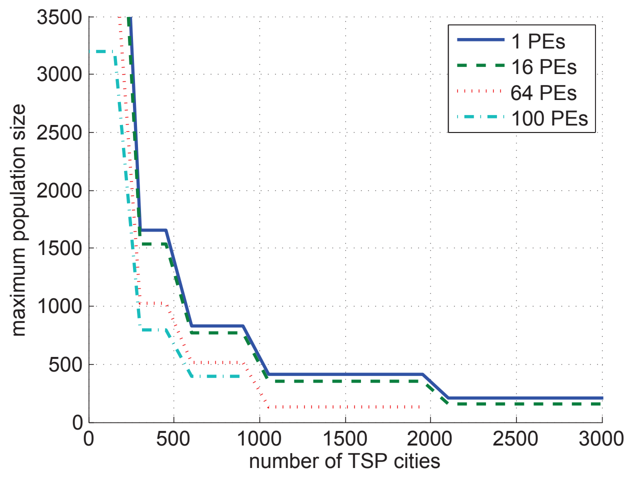


Figure 6. Maximum population size and number of TSP cities allowed in a Virtex-6 FPGA (XC6VLX240T) for different numbers of PEs in the cGA. Results are constrained by the maximum number of available BRAMs.

The same procedure was also executed on a MicroBlaze processor running in the same Virtex-6 FPGA. Results have shown a throughput of 4.59×10^3 solutions/second. This means that our cGA architecture can achieve an impressive speedup of more than 6700× compared to a software version running on the same hardware platform.

The proposed cGA can be customized for TSP problems with different number of cities and population sizes. The available dual-port memory blocks (BRAMs) in the Virtex-6 FPGA (416) limit the number of PEs, population size, and number of cities, since all the information needs to be kept in internal memory. Fig. 6 plots the maximum population size and number of cities in arrays with 1, 16, 64 and 100 PEs. This data was calculated constraining the width and depth of the BRAMs (subpopulation and internal PE memories) to integer powers of two and taking into account the space required for each solution. These numbers show that, for example, a 16 PE cGA array can accommodate TSP instances up to 3000 cities and a population size of 160 solutions. A 10×10 cGA with a population of 200 solutions was also synthesized: it occupies 53.5 % of the slices and 72.1 % of the BRAMs available on the FPGA, and it can achieve a maximum frequency of 112 MHz.

VI. Conclusions

We have presented a scalable hardware architecture for a cellular genetic algorithm targeting FPGA devices. By distributing the entire population over smaller subpopulations that can be accessed simultaneously by independent processing elements, the algorithm can be parallelized without critical bottlenecks due to memory accesses. Each subpopulation of the cGA is shared between two adjacent PEs and can thus be easily implemented in FPGA devices by the use of the native dual-port memory blocks.

The architecture was implemented in a Virtex-6 FPGA to solve a TSP instance and results have shown that, in spite of the spreading of the solutions throughout the various subpopulations, the convergence is not degraded. Additionally, the throughput of the cellular array increases almost linearly with the number of PEs.

The cGA proposed in this work can be applied to other optimization problems by designing custom PEs with appropriate implementations of selection, replacement, crossover, mutation, and fitness evaluation. We are now developing a library of generic templates for building a PE from a C code specification using high-level synthesis tools in order to accelerate the implementation of the processing genetic element to be applied to other optimization problems.

References

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

[2] I. Hong, S. Sohn, J. Lee, and J. Kim, "DFS algorithm with ranking based on genetic algorithm in UHF portable system," in *Proc. 9th Int. Conf. Comm. Inf. Technol.* IEEE Press, 2009, pp. 454–459.

[3] W. Yan, F. Weiping, Z. Chensheng, and W. Hongtao, "Image matching for workpiece based on genetic algorithm," in *Proc. 2009 Int. Conf. Artif. Intell. Comput. Intell.*, vol. 3. IEEE Computer Society, 2009, pp. 152–157.

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

[5] 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.

[6] P. Larranaga, C. Kuijpers, R. Murga, I. Inza, and S. Dizdarevic, "Genetic algorithms for the travelling salesman problem: A review of representations and operators," *Artif. Intell. Review*, vol. 13, no. 2, pp. 129–170, 1999.

[7] G. Luque and E. Alba, *Parallel Genetic Algorithms: Theory and Real World Applications*. Springer, 2011, vol. 367.

[8] B. Shackleford, G. Snider, R. 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, pp. 33–60, 2001.

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

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

[11] 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.

[12] A. Munawar, M. Wahib, M. Munetomo, and K. Akama, "Advanced genetic algorithm to solve MINLP problems over GPU," in *IEEE Congr. Evol. Comput.*, 2011, pp. 318–325.

[13] P. Vidal and E. Alba, "A multi-GPU implementation of a cellular genetic algorithm," in *IEEE Congr. Evol. Comput.*, 2010, pp. 1–7.

[14] B. Shackleford, M. Tanaka, R. Carter, and G. Snider, "FPGA implementation of neighborhood-of-four cellular automata random number generators," in *Proc. 2002 ACM/SIGDA Tenth Int. Symp. Field-Program. Gate Arrays*, 2002, pp. 106–112.

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