Parallel Genetic Algorithms on Multiple FPGAs

Liucheng Guo
Dept. of EEE
Imperial College London
SW7 2AZ, London, UK
gl512@imperial.ac.uk

Andreea Ingrid Funie
Dept. of Computing
Imperial College London
SW7 2AZ, London, UK
andreea.funie09@imperial.ac.uk

David B Thomas
Dept. of EEE
Imperial College London
SW7 2AZ, London, UK
dt10@imperial.ac.uk

Haohuan Fu
Center for Earth System Science
Tsinghua University
Beijing, P.R. China
haohuan@tsinghua.edu.cn

Wayne Luk
Dept. of Computing
Imperial College London
SW7 2AZ, London, UK
w.luk@imperial.ac.uk

ABSTRACT

Genetic algorithms (GA) have been shown to be effective in the optimization of many large-scale real-world problems in a reasonable amount of time. Parallel GAs not only reduce the overall GA execution time, but also bring higher quality solutions due to parallel search in multiple parts of the solution space. This paper proposes a parallel GA system on hardware such as Field-Programmable-Gate-Arrays (FPGAs). Our approach targets multiple FPGAs by exploring different search areas of the same solution space with different behaviours. Each FPGA contains an optimised customisable GA which can be configured using run-time parameters, removing the need for expensive recompilation. This paper also explores adjustment of the migration gap, providing empirical guidance on good settings to users. Experiments on three problems show the high performance of our system, with a 30 times speedup achieved compared to a multi-core CPU-based implementation.

1. INTRODUCTION

Genetic algorithms (GAs) are effective in solving different types of optimization problems. Nevertheless, due to its heuristic nature, a genetic algorithm often has a long execution time, thus becoming vital to optimise and accelerate this well-known computation technique.

Various techniques are available to find better solutions in the search space, with one common method being the parallellisation of the GA technique itself [2]. *Parallel GA* (pGA) is a type of GA which contains multiple GA instances that are evolving and exchanging information in parallel. The search processes of different GA instances work on different areas of the search space in parallel. The pGAs can improve the solution's quality while decreasing the execution time. The use of multiple populations in pGAs is based on the idea that the isolation of populations can maintain a higher genetic diversity, while the communication between them can make GAs work together to find good solutions.

There exists previous work on finding an optimal way of adapting pGAs to hardware platforms such as FPGAs. However, several challenges appear while optimising pGAs for FPGAs. First, the FPGAs resource availability limits the problem size and the degree of parallelism in the platform; Second, when using multiple FPGAs to increase the degree and problem size, the connection complexity and synchronisation reduce performance; Third, recompilation can be time-consuming, making it impractical to frequently modify the hardware at run time.

This paper proposes a full system to solve the problem of creating and executing parallel genetic algorithms on multiple FPGAs. Our main contributions are:

- A fully parallel GA system on multiple FPGAs, avoiding the limitation of resources in one single FPGA.
- A connection structure with a low-cost communication, making the architecture extensible without significant cost.
- A system supporting a run-time changeable migration gap, making it possible to tune it for fast evaluation.
- An investigation about the impact of migration gap on optimisation quality, providing a guidance for setting the migration gap by experiments.

We apply our system to *Function 11*, the *Locating problem* and *the Travelling Salesman Problem*. The experiments on 4-FPGAs show an average of 30 times speedup over a multi-core CPU-based GA.

2. BACKGROUND AND RELATED WORK

2.1 Genetic Algorithms

Genetic algorithms are inspired by evolution in nature and they usually take the form of an iterative process. At the end of each iteration, a new population of individuals (generation) gets created [1]. Every individual in the iteration is represented as a chromosome and describes a possible solution in the search space of the specific problem.

There are different types of genetic algorithms such as: the *generational* GA or the *steady-state* GA. The *generational* GAs work as follows [2]. The first step is to randomly generate a population of individuals. The second step evaluates each of the individuals using a problem-specific fitness function, thus assigns fitness values to all individuals

This work was presented in part at the international symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2015) Boston, MA, USA, June 1-2, 2015.

in the population. The next step is to perform the genetic operators (crossover, mutation and selection) to produce a new generation of individuals (offspring). After offspring are produced, the new generation will replace the previous generation and the above steps are then repeated. The evolutionary process stops when a convergence criterion is met: the maximum number of iterations is reached or a good solution appears.

Our approach is based on the *steady-state* GA, which is a simplified version of the generational GA. The steady-state GA selects two parents, crosses them to produce two offspring, mutates the offspring, then inserts them back into the population. This process is repeated in a loop, so the population size remains constant, instead of a new population which contains the whole offspring being maintained. The steady-state GA requires less resources and has a reduced delay before entering the next evolutionary process. Meanwhile, the steady-state GA always requires more generations to find a good solution than a generational GA.

2.2 Parallel Genetic Algorithms

Parallel genetic algorithms have been created to reduce the execution times which are often associated with classic genetic algorithms when finding optimal solutions for large-scale domain specific search spaces [2].

One simple way of obtaining GA parallellisation is to simply execute multiple copies of the same GA instance, however each of these GA instances alone would have to start with different initial sub-populations, evolve, and stop independently of the other parallel GAs. Another method is the *distributed GA* (dGA): the independent GA instances now periodically exchange chromosomes between their populations, thus not only sharing high quality solutions, but potentially reducing the execution time through the periodic migration of the information. Chromosome migrations occur after a number of iterations, when each of the individual GA instances sends a copy of its locally best chromosome to the next GA instance at each of the migration steps and so on. Normally, the GA instance receiving a chromosome replaces the locally worst chromosome with the incoming one, unless an identical chromosome already exists in its local population.

2.3 FPGA-based Parallel GA

FPGAs are a promising platform to successfully provide execution time speedup as they combine the parallelism and great performance of hardware with the flexibility provided by a software tool.

Genetic algorithms were first introduced on FPGAs in 1995 [3]. There are a number of conventional GA systems mentioned in [4], [5], [6], [7], [8], [9]. For example, [9] developed a GA system for the travelling salesman problem.

Researchers also demonstrated a number of FPGA-based architectures for dGAs, and other kinds of pGAs. Some of them as mentioned in [10], [14], [12], [13], [17], [15] and [16]. For example, [16] proposes a multi-level parallel dGA system and targets one single FPGA with multiple GA instances, while [17] proposed a fine grained pGA system in one FPGA. We summarise the features of these systems in the first seven rows in Table 1. Due to the insufficient resources of one single FPGA the scale of the problems solved and the degree of parallelism is limited. To address the problem, researchers adapt pGAs to multiple FPGAs [20], [11]. Majority of the research so far involves the use of *generational* GA, instead of the *steady-state* GA, as the former is more common in the software GA. However, while the *steady-state* GA requires more generations, it is simpler to perform and needs less resources than a *generational* GA, thus being ideal for FPGA.

While parallel GAs on multiple FPGAs exist, they all suffer from a number of problems: 1) lack of flexibility of the architecture, resulting in time-consuming recompilation when modifying them; 2) the link of multiple FPGAs are complex, reducing the performance carried by parallel GAs; 3) they do not investigate the impact of migration settings, thus not providing any guidance of the settings to users.

This paper proposes a multiple FPGA GA system with an easy communication method, supporting run-time modification of the migration gap without time-consuming recompilation.

3. CUSTOM GA IN MULTIPLE FPGAS

Parallel GAs have a big potential for hardware implementation, however when designing a parallel GA system on multiple FPGAs, we need to consider the following challenges, as working with multiple FPGAs is different from working with multi-core CPUs:

- The connection and synchronisation between FPGAs are very complex. A simple method is needed to abstract the details.
- The compilation of FPGAs is time expensive, so it is impractical to frequently adjust GA parameters at compile-time.
- The migration parameters play an important role in the parallel GA system, but guidance is needed for its correct setting.

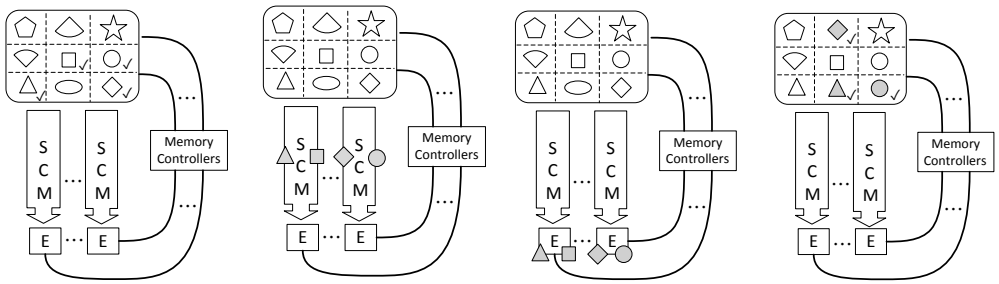
Our approach has three novel features that significantly improve its pipeline performance:

- It adopts the steady-state GA method, allowing a custom GA to continue to evolve without updating the entire population. In contrast, recent work [16] is based on generational genetic algorithms which have to wait for all the new offspring to be produced before entering the next generation, making it slow;
- It supports simple communication between multiple FPGAs, encapsulating the complex operation of connecting and synchronising multiple FPGAs together;
- It provides guidance to users about how to set the migration gap. In addition, users can change the migration gap and the other parameters at run-time.

There are several main units in the customisable GA architecture, described in the following sections. It mainly includes population memory, selection, crossover, mutation, evaluation and memory control units. The flow of our GA system is shown in Fig. 1.

3.1 Population Memory

The population memory stores both the individuals in one population as well as their associated fitness values. In Fig. 1, we show nine different individuals in the memory. Because we use steady-state GA, we only maintain this part of memory in our system.



(a) Selecting (b) Evolving (c) Evaluating (d) Updating

Figure 1: GA Flow

3.2 Selection-Crossover-Mutation and Evaluator Unit (SCM/E Unit)

In the steady state GA, we only make use of offspring generated from a mix of the parents and offspring individuals, instead of using the whole offspring population. On each iteration, the steady state GA only produces and inserts back two individuals, which we call an update.

The selection, crossover and mutation operators deal with two individuals sequentially, and then the evaluator assigns the fitness values to the new offspring. This results in a big potential for parallel processing of those operators for one update. For further parallelism and increased performance we combine those operators in an SCM/E unit, to finish the updates in parallel.

In hardware, every SCM/E unit consists of one selection, one crossover, two mutation operators, and one evaluator, thus processing two individuals at the same time, evaluating them one by one, and then sending them into the memory controller units without increasing the population size. Our system supports different configurations in SCM/E units for different problems. For example, the binary chromosomes can be changed by one-point crossover and flip-flop mutation, the permutation chromosomes can be evolved by multiple-point crossover and exchange mutation, while the real-valued chromosomes can be modified by blending crossover and constraint mutation [19]. The random number generator used in hardware is the same as the one described in [16] paper.

As shown in Fig. 1, the system has multiple parallel SCM/E units. In the selection stage, individuals in different locations are chosen by a selection method and a number of pairs of the individuals enter the SCM/E units. In this evolving stage, the multiple SCM/E units will crossover and mutate each of the pairs in parallel, finally producing new offspring. Then, in the the evaluating stage, offspring are evaluated and assigned the fitness values. In the Fig. 1, the shapes in gray are the new offspring.

3.3 Memory Controller

The memory controller manages the update of the populations. When the memory controller receives the offspring, it will produce a random address for the individuals to tell it where the offspring must go. If the fitness of the incoming individual is higher than the current one in that address, the individual will replace the current one. Otherwise, the lower fitness individual will be thrown away. For example, in the updating stage of Fig. 1, the offspring replaces three parents. The replacement method can also take a different form, such as elite replacement, or randomised replacement.

3.4 Inter-FPGA Migration Unit

The inter-FPGA migration unit controls all the migration parameters, including the migration gap and number of migrants. The user can customise all of those parameters from the inter-FPGA migration unit. The migration gap plays the most important part in the system and the settings for which is presented in the next section.

The migration unit has two input ports and two output ports and all the units can communicate with each neighbour FPGAs. Our platform's communication process can be divided into three stages: sending, receiving and preparing. As shown in Fig. 2, there are three FPGA-based GAs: GA1, GA2 and GA3. In the first stage the three FPGA-based GAs keep sending their individuals (a, b, c) to their neighbours, according to the links from Fig. 2, after previously placing them into their output ports. In the second stage, the input ports get updated after a fixed number of generations, to c, a, b. Before that, although receiving the data, the input ports will not update. Lastly, when our system reaches the next generation gap, the data in the output ports gets updated to a', b', c' and then reaches the preparing stage. To simplify the communication between multiple FPGAs, we allow the hardware to receive and send the migrants at different times.

3.5 Qualitative Summary

We summarise the features of our multiple-FPGA approach and compare it with other pGA approaches mentioned in the related work section. From the comparison shown in Table 1, it is clear that our approach is different from others, because we solve all problems encountered in the current research:

1. Our approach adopts steady-state GA while most of others use generational GA. The steady-state GA has a reduced latency and lower resource usage compared with the generational one because it maintains fixed memory and only updates a part of whole population.

2. Our approach maximises flexibility and parallelism in the architecture, the degrees of both inter-FPGAs and internal SCM/E parallelism. The multiple SCM/E units make it easy to update the steady-state GAs while they work in parallel.

3. Our approach supports a simple method for the inter-FPGA link, which abstracts the complexity and the

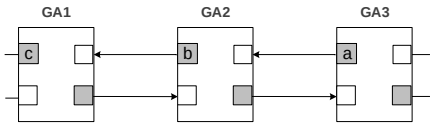
Table 1: Qualitative Comparisons of Multiple FPGA-based pGAs

Work	Year	# of FPGAs	Runtime Param.	Migration Param.	Parallel SCM/E	GA Type	Platform
[11]	1999	Multiple	N/A	No	No	Steady-State	SFL
[12]	2000	Single	N/A	N/A	No	Steady-State	PCIGEN10K
[20]	2005	Multiple	N/A	No	No	Generational	Spartan II-E
[15]	2006	Single	N/A	N/A	No	Generational	Stratix EPS1S10
[10]	2006	Single	N/A	N/A	No	Generational	Cyclone FPGA
[14]	2012	Single	N/A	N/A	No	Generational	Virtex 4 (LX25)
[17]	2013	Single	N/A	No	No	Generational	Virtex 6 (LX240T-1)
[16]	2014	Single	SCM rates	Run-time	Yes	Generational	Virtex 6 (SX475T)
Proposed	2015	Multiple	SCM rates	Run-time	Yes	Steady-State	Virtex 6 (SX475T)

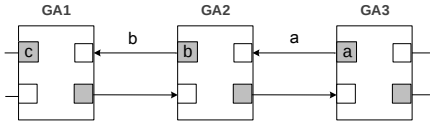
synchronisation between multiple FPGAs.

4. Our approach allows users to tune the migration gap at run-time, which saves significant compilation time.

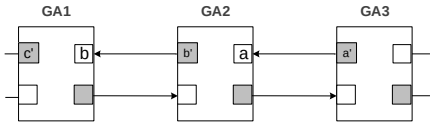
5. Our approach provides guidance of setting migration gaps in section 4.3, which helps to reduce the number of generations needed for good solutions.



(a) Stage1: Preparing



(b) Stage2: Sending



(c) Stage3: Receiving

Figure 2: Communication Strategy

4. PERFORMANCE EVALUATION

We implement the proposed solution on a workstation with four Maxeler MAX3A Vectis acceleration cards. Each card contains a Xilinx Virtex-6 SX475T FPGA. The CPU-based GA system [18] is based on a 12 physical Intel Xeon-X5650 CPU cores running at 2.67GHz. The CPU code is well tuned with multi-threading techniques such as Pthread, and is compiled with the Intel C/C++ compiler with level-3 optimisation. We use eight threads to test the code. The FPGA code is written using the OpenSPL language [21].

4.1 GA Benchmarks

Global optimization has many real-world applications, both of discrete and non-discrete nature. We know that a general global optimization algorithm is usually less efficient than specific versions tuned to the problem at hand, however it is still valuable to gauge the baseline performance of a global optimization scheme using benchmark problems.

Therefore, in our testing we make use of classic GA benchmarks, such as *Function* 11, the *Locating Problem* and the *Travelling Salesman Problem*. All of these benchmarks are considered to be complex to solve, but our system can find good solutions for them. Because our system uses multiple FPGAs in an attempt to converge faster to the optimum, the performance scales with the number of FPGAs used, as shown in subsection 4.3.

4.1.1 F11

Function 11 (F11) is a popular GA benchmark [18]. The objective of the optimisation problem is to maximise

$$f(\mathbf{x}) = 1 + \sum_{n=1}^{N} x_n^2/4000 - \prod_{n=1}^{N} cos(x_n) \qquad (1)$$

s.t. $-10 \leq x_n \leq 10.0$

We have tested our approach with a population of 64 individuals when N is 4. We use 4 x 32 bits fixed-point chromosome to represent the solution.

4.1.2 Locating Problem

The locating problem deals with finding a good location for an emergency response unit, which has the best response time for reaching any emergency that occurs in a city.

Reference [2] provides a complex example with a 10 km $\times$ 10 km city divided into 100 sections. The response unit can be put at any place in the city, so a solution (x_f, y_f) is a floating point coordinate. The objective of this problem is to minimise the following cost function:

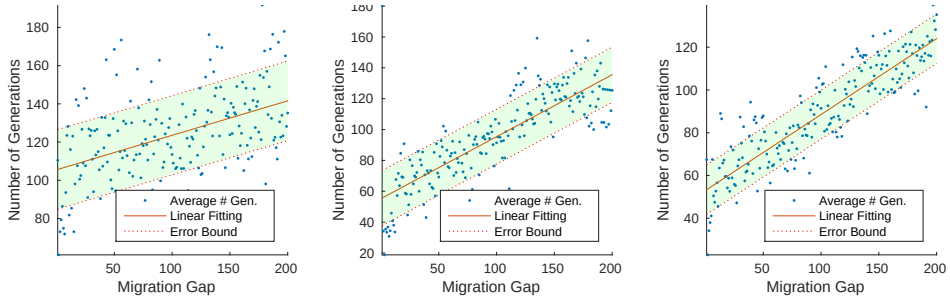
$$f(x_f, y_f) = \sum_{n=1}^{100} w_n \sqrt{(x_n - x_f)^2 + (y_n - y_f)^2} \qquad (2)$$

where (x_n,y_n) is the coordinate of the centre of square n and w_n is emergency frequency in square n.

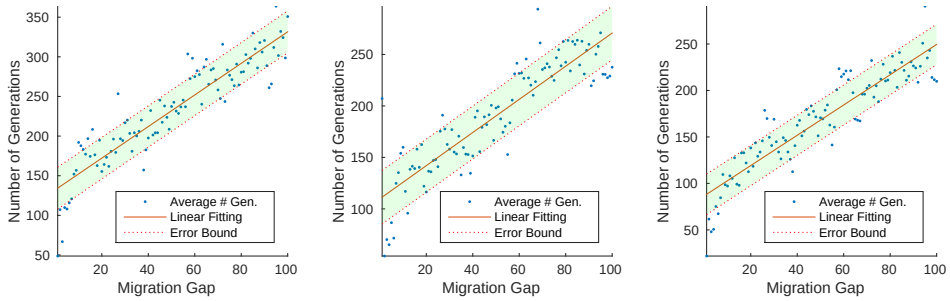
We have tested our approach on a number of 32 individuals. The bits representation used for the solution is a 2 x 64 bits floating-point variables.

4.1.3 The Travelling Salesman Problem

The Travelling Salesman Problem is a well-known NP-hard problem in combinatorial optimization which sorts the following problem: given a list of cities and the distances between each pair of cities we try to find the shortest possible route that visits each city exactly once and returns to the



(a) F11: 2 FPGAs (b) F11: 3 FPGAs (c) F11: 4 FPGAs



(d) LP: 2 FPGAs (e) LP: 3 FPGAs (f) LP: 4 FPGAs

Figure 3: Experimental Results with Different Migration Gap and Number of FPGAs

origin city. We have tested our approach on a number of 64 cities. The bits representation used for the TSP solution is a 64 x 6 bits permutation encoding.

4.2 Experimental Results

We have obtained some significant good results as well a large speedup for the problems we have tested our approach on (see Table 2). We will further analyse them in Section 4.3.

Table 2: Experimental Results (P: SCM/E Parallelism, I: Individuals, Freq.: Frequency, SP.C: Speedup over C-GA)

Fun.	# of FPGA	# of P	# of I	Freq.(MHz)	SP.C
TSP	4	1	32	100	30
Locating	4	32	64	150	27
F11	4	4	64	150	34

Our tool performs the best in the case of the F11 benchmark, obtaining a 34 times speedup, while using a 150 MHz clock frequency, when compared to an equivalent multi-core software implementation of the GA. We also test TSP and we can notice from the resources usage on one FPGA that we could try to further optimise our design and possibly fit 2 pipes instead of one (see Table 3).

Based on the resources left over, we can tailor the architectures according to the complexity of our evaluation and SCM/E unit (see Table 3).

Table 3: Resources in one FPGA for LP, TSP and F11

Fun.	LUTs(%)	FFs(%)	BRAMs(%)	DSPs(%)
TSP	75.85	59.69	19.17	6.25
Locating	72.18	40.71	9.68	26.98
F11	60.05	38.14	17.67	32.54

4.3 Migration Gap and Discussion

In this section, we examine how the migration gap affects the performance, and give users the guidance of setting it. We have done experiments for the first two problems and their results are shown in Figure 3. The first three graphs are the F11 under different generation gap, while the last three are for the locating problem (LP). LP-4FPGAs means using 4 FPGAs for the locating problem.

In Figure 3, each point represents the average number of generations to reach 90% of the best possible fitness in 100 repetitions using the corresponding generation gap. The straight line means linear regression results from points. This is to show the trend of the growth. The error bound is the regression result represented by one standard deviation of the regression error, showing the uncertainty of the linear regression.

Thus, as a result of our extensive experiments we can draw two major conclusions as follows:

- A system with more FPGAs takes fewer generations to obtain satisfactory fitness. One explanation of this observation is that having more FPGAs enables more individuals to evolve in parallel. In addition, with a fixed migration rate, the total number of migration cases is higher with more FPGAs.
- The number of generations to reach the satisfactory fitness level grows linearly against the migration gap. We also notice that, even if we increase the degree of the regression polynomial, we still obtain a linear trend. This trend reveals the strong correlation between the migration gap and the optimisation quality, but there does not seem to be any obvious reasons for the linearity of the correlation.

The results provide practical guidance for the determina-

tion of the migration gap. The user may determine a reasonable migration gap for a problem by trying different values. The experiments suggest that low values of migration gap are likely to result in high optimisation quality. As a result, we suggest users to give high priority to small values in the determination of migration gaps.

5. CONCLUSION

Parallel genetic algorithms have shown significant potential for large hardware acceleration in evolutionary computing [2], as demonstrated by previous work and our study.

This paper proposes a GA system which makes use of multiple FPGAs to speed up the search for optimised solutions. Our system is simple to use, providing increased flexibility not only in picking the SCM/E components and their rates, but also in selecting the migration parameters. Our approach exploits two levels of parallelism: fine-grained parallelism in SCM/E units, and the coarse-grained parallelism in inter-FPGAs. It enables the tuning of run-time parameters without time-consuming hardware recompilation. Our system is more flexible compared to existing FPGA systems for parallel genetic algorithms, while providing significant execution time speedup and converging faster to the target-problem optimum.

We have tested our system on a location problem, a GA benchmark called F11, as well as the well-known travelling salesman problem. After running experiments on these problems, we obtain 30 times speedup on average with a 4-FPGA system compared to a multi-core software implementation.

As a result of our extensive experiments, we are able to provide a guidance for setting a specific value for the migration gap. The migration gap parameter can be set at run-time, thus allowing users to change it without time-consuming hardware recompilation.

Current and future work include exploring methods to automatically create all hardware elements, and also to automatically pick the best configurations for the hardware.

6. ACKNOWLEDGMENT

Ce Guo makes a contribution. The first author is financially supported by the Imperial CSC Scholarship.

7. REFERENCES

[1] M. Mitchell, *"An Introduction to Genetic Algorithms"*, Bradford Book, 1998.

[2] G. Luque, and E.Alba, *"Parallel Genetic Algorithms: Theory and Real World Applications"*, Springer, Vol. 37, 2011.

[3] S. Scott, S. Ashok, and S. Shared, *"HGA: A hardware-based genetic algorithm"*, ACM International Symposium on Field-Programmable Gate Arrays, pp. 53-59, 1995.

[4] M. Vavouras, K. Papadimitriou, I. Papaefstathiou, *"High-speed FPGA-based implementations of a Genetic Algorithm"*, International Symposium on Systems, Architectures, Modeling, and Simulation (SAMOS), pp. 9-16, 2009.

[5] J. Pimery, and K. Pinit, *"Development of a flexible hardware core for genetic algorithm"*, Intelligent Computing and Intelligent Systems, Vol. 1, pp. 867-870, 2009.

[6] C. Effraimidis, K. Papadimitriou, A. Dollas, and I. Papaefstathiou, *"A self-reconfiguring architecture supporting multiple objective functions in genetic algorithms"*, International Conference on Field Programmable Logic and Applications (FPL), pp. 453-456, 2009.

[7] P.R. Fernando, R. Zebulum, and A. Stoica, *"Customisable 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] L. Guo, D. B. Thomas, and W. Luk, *"Customisable Architectures for the Set Covering Problem"*, HEART, pp. 69-74, 2013.

[9] M.A. Vega-Rodriguez, R. Gutierrez-Gil, J.M. Avila-Roman, J.M. Sanchez-Perez, and J.A. Gomez-Pulido, *"Genetic algorithms using parallelism and FPGAs: the TSP as case study"*, International Conference Workshops on Parallel Processing, pp. 573-759, 2005.

[10] T. Tachibana et. al., *"General architecture for hardware implementation of genetic algorithm"*, IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 291-292, 2006.

[11] N. Yoshida, and T. Yasuoka, *"Multi-gap: parallel and distributed genetic algorithms in VLSI"*, Systems, Man and Cybernetics, Vol. 5, pp. 571-576, 1999.

[12] Y. Choi, and D. J. Cheung, *"VLSI processor of parallel genetic algorithm"*, IEEE Asia Pacific Conference on ASIC, pp. 143-1462, 2000.

[13] Y. Jewajinda, and P. Chongstitvatana, *"FPGA implementation of a celullar compact genetic algorithm"*, NASA/ESA Conference on Adaptive Hardware and Systems (AHS), pp. 385-390, 2008.

[14] T. Kamimura, and A. Kanasugi, *"A parallel processor for distributed genetic algorithm with redundant binary number"*, 6th International Conference on New Trends in Information Science and Service Science and Data Mining (ISSDM), pp. 125-128, 2012.

[15] M. S. Jelodar et.al., *"SOPC-based parallel genetic algorithm"*, IEEE Congress on Evolutionary Computation, pp. 2800-2806, 2006.

[16] L. Guo, D. B. Thomas, C. Guo, and W. Luk,, *"Automated framework for FPGA-based parallel genetic algorithms"*, 24th International Conference on Field Programmable Logic and Applications (FPL), pp. 1-7, 2014.

[17] Dos Santos, P.V., Alves, J.C., Ferreira, J.C., *"A scalable array for Cellular Genetic Algorithms: TSP as case study"*, International Conference on Reconfigurable Computing and FPGAs (ReConFig), pp. 1-6, 2012.

[18] D. A. Coley, *"An Introduction to Genetic Algorithms for Scientists and Engineers"*, World Scientific, 1999.

[19] R. L. Haupt, and S. E. Haupt, *"Practical Genetic Algorithms"*, John Wiley and Sons, 2004.

[20] J. Newborough, and S. Stepney, *"A generic framework for population-based algorithms, implemented on multiple FPGAs"*, Artificial Immune Systems, Springer Berlin Heidelberg, pp. 43-55, 2005.

[21] OpenSPL Consortium, The OpenSPL Standard, v1.0, http://www.openspl.org/