An Automated Framework for General-Purpose Genetic Algorithms on FPGAs

Abstract:
FPGA-based Genetic Algorithms (GAs) can effectively optimise complex applications, but require extensive hardware architecture customisation. To promote these accelerated GAs to potential users without hardware design experience, this paper proposes an automated framework for creating and executing a general-purpose GA system on FPGAs. This framework is a scalable and customisable hardware architecture while providing a unified platform for different chromosomes. At compile-time, only a high-level input of the target application needs to be provided, without any hardware-specific code being necessary. At run-time, application inputs and GA parameters can be tuned, without time-consuming recompilation, for further good GA executions configuration to be found. The framework was tested on a high performance FPGA platform using six problems and benchmarks, including the Travelling Salesman problem, a locating problem and the NP-hard set covering problem. Experiments show the system's flexibility as well as an average speed-up of 29 times compared to a multi-core CPU.

Keywords: Genetic Algorithms; FPGA; Automated Framework.

1 Introduction

Genetic algorithms (GAs) are a common population-based generic meta-heuristic in artificial intelligence. These algorithms generate solutions to problems using bio-inspired techniques, including reproduction, selection, crossover and mutation, all of those which are similar to natural evolution. Genetic algorithms are effective in the optimisation problems where other methods experience difficulties, including combinatorial optimisation and real-valued parameter estimation.

With the increasing requirement for high-performance computing and low-energy consumption, genetic algorithms need to be accelerated or adapted in embedded systems. For example, complex problems often require a GA to evolve many generations to produce a satisfactory solution, meaning CPU-based GAs are often too slow to handle those problems. In addition, real-time applications also require GAs to produce solutions with low latency (31).

In order to accelerate GAs, researchers have adapted them to hardware, such as field programmable gate arrays (FPGAs) (3) and graphics processing units (GPUs) (34). GPUs involve a massive parallel processing elements and are suitable for the algorithms with high parallelism. FPGAs take advantages of flexibility of software and high-performance of hardware, becoming a promising platform for acceleration. The existing hardware-based GA systems are faster than CPU-based GAs in solving many real-world applications and benchmarks. In particular, FPGA-based GA systems are commonly used due to their flexibility (6).

Most existing hardware GA systems are written in low-level hardware description languages such as VHDL and Verilog which require an in-depth knowledge of FPGA architecture and hardware programming. Another issue present in the current state-of-the-art tools is that most existing FPGA-based GA systems only support one type of chromosome, either binary, real-valued or permutation, which limits their applicability. To address these issues, this paper proposes an automated unified framework to create and execute GA systems on FPGAs, with the following contributions:

- A framework which could create and execute general-purpose GA systems on FPGAs: based on the user-defined high-level description and hardware template, the framework automatically generates the whole system.
- A novel FPGA-based GA architecture: the design is both scalable and customisable, supporting binary, real-valued and permutation chromosomes, and also enabling a user to change the parallelism of the architecture.
- A run-time tuning framework: GA parameters and application inputs are changeable without time-consuming recompilation, thus a user can freely tune GA parameters to find good configuration for future executions.

We qualitatively compare our work with existing FPGA-based systems, showing improved flexibility in hardware architecture and functionality in run-time tuning, as well as increased ability to support complex applications. We also quantitatively compare the accelerated FPGA framework with a multi-core CPU and a third party GPU, including all I/O and initialisation costs, showing an average speed-up of 29 times over the CPU and 5 times over the GPU system. We find the same solutions for six different applications and benchmarks.

The arrangement of the paper is as follows: section 2 describes the background and related work; section 3 presents the proposed work of automated framework; section 4 demonstrates the user inputs for the framework; section 5 compares our work with previous FPGA-based systems quantitatively; and section 6 presents the experimental results of our work.

2 Background and Related Work

2.1 Genetic Algorithms

Genetic algorithms (GAs) are increasingly popular search-based algorithms which emulate the natural evaluation process making use of three genetic operators (*selection*, *crossover* and *mutation*) which follow the principles first laid down by Charles Darwin of *"survival of the fittest"*. While being randomised, GAs are not behaving in a random manner, but they make great use of historical information in order to generate better performance within the search space.

Genetic algorithms are used in solving many different complex problems. During recent research, it has been shown that in searching a large state-space or an n-dimensional surface, this heuristic technique may offer significant benefits over more typical search of optimisation techniques such as: depth-first/breath-first search or linear programming.

The genetic algorithm is based on iteratively updating a collection of individuals, called the population. During each iteration, all of the current population members are evaluated according to the selected fitness function.

A new population is being created by probabilistically selecting the best performing individuals from the current population. Those best performing individuals are then forming the basis for creating new offspring individuals by applying a number of different genetic operations, such as crossover, mutation, reproduction, on them. The individuals to be included in the new generation are being selected probabilistically and this can be done in a number of different ways. For example, the probability for an individual to be selected can be proportional to its own fitness while in the same time being inversely proportional to the other individuals' fitness from the current population.

Once the best performing individuals have been selected to be included in the next generation, additional members are generated using a crossover operation (this operation takes two parents, again using a probabilistic approach, from the current generation and creates two offspring individuals by recombining portions of both parents). At this point, some members are chosen and random mutations are performed on them, thus obtaining new altered individuals. Also, some of the best performing individuals are being selected to be cloned (a copy of them is being added in the next generation), again with a given probability.

Once the new generation has been created it is then becoming the new population which will go through the same process again (its individuals are evaluated, selected for the offspring and a new generation is being generated) until satisfying a termination condition, for example reaching maximal generation number (*gen_max*), or finding acceptable solutions.

The code for a typical genetic algorithm is the following:

```
Algorithm 1: A typical genetic algorithm

Input: random_seed //random nunmber seed
       gen_max //maximum generatio
       ni //number of individuals

1. gen_n = 0 //The number of generations
2. while(gen_n < gen_max)
     //evolution starts
3.   foreach i in [0  ni) begin
4.     oldpop[i] = (gen_n == 0)?
                   init[i]: newpop[i]
       //initial or new population
5.     fitness[i] = evaluate(oldpop[i])
       //fitness evaluation
7.   end
8.   foreach i in [0,(ni/2)) begin
9.     select(oldpop, oldtwo, fitness)
       //select 'parents' into oldtwo
10.    crossover(oldtwo)
       //exchange inforamtion
11.    foreach j in [0,2) begin
12.      mutation(oldtwo[j])
         //introduce diversity
13.      newpop[2*i + j] = oldtwo[j]
         //store the offsprings
14.    end
15.  end
16.  gen_n ++ //enter next generation
17. end
```

The data flow of the typical genetic algorithm is presented in Fig 1. When applying a genetic algorithm to a specific problem, a user should first design the structure of chromosomes and define a fitness evaluation function for the solution domain. The chromosome is often represented in a binary encoding format (a vector of bits), with the evaluation function re-interpreting sub-sequences of the binary chromosome as booleans, permutations, integers, or real components. Except for the problem-dependent parts (chromosome and evaluation function), the other parts in a typical GA such as genetic operators, are problem-independent. Therefore, it is possible to build a general-purpose library, which can offer various methods of genetic operators for different types of chromosomes.

The *generational* GAs work as follows: First, a population of individuals is randomly generated, so that each of those individuals can then be evaluated using a problem-specific fitness function (this step

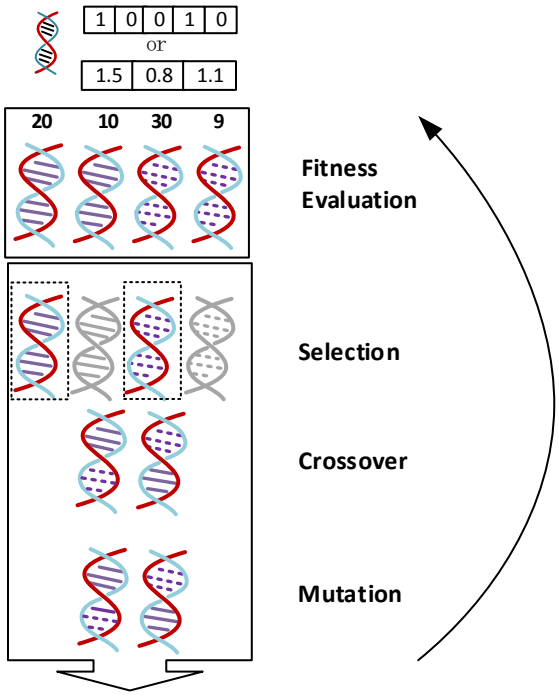

Figure 1 Data Flow of a Typical Genetic Algorithm

involves assigning fitness values to all individuals present in the population in a current time). Next, the genetic operators (crossover, mutation and selection) are performed in order 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 algorithms then stops when a convergence criterion is met.

2.2 Reconfigurable Computing

Many applications require high performance, and the current solutions come from Microprocessors, application-specific integrated circuites (ASICs) and FPGAs. Application-specific integrated circuits (ASICs) are customised based on a specific application for high performance, but they are inflexible after manufacture. Microprocessors provide high flexibility but the performance is limited because of the fetch-decode-execute process. FPGAs combine the flexibility of microprocessors and efficiency of ASICs. The FPGAs contain logic gates and small random-access memories. The platform can be configured many times for specific applications during the compilation process, including synthesis, map, place and route.

FPGA systems also have drawbacks and challenges. Unlike software, the compilation process is time consuming and often takes hours to finish, so it is not practical to frequently modify hardware designs. The programmability in FPGAs is also not easy, as all algorithms need to be written in a low-level hardware description language (HDL) such as VHDL or Verilog. The HDL programs are based on the registers and logic gates. Recently, high-level compilation tools are proposed to reduce the programming effort (18). Even with these tools, it is still difficult for a non-expert user to create fast and efficient FPGA systems. However, these tools can provide a good intermediate-level target for customisable frameworks, such as our GA system.

2.3 Previous hardware-based GAs

With the increasing demand for GA performance, researchers use hardware platforms such as FPGAs and GPUs to accelerate the evolution process. For example, a generational GPU-based GA system combined with local search for the maximum satisfiability (MAX-SAT) problem is proposed in (34). Pedemonte et al. propose a non-generational GA system called systolic genetic search, which places the genetic operators into a fixed array (35). The individuals in the system go through the network in a fixed way, thus the systems may not be suitable for all the problems. While GPU-based GA systems can be faster than software, there is a large communication and synchronisation cost between processing elements, and existing work is limited to a restricted set of problems.

FPGAs are thus a promising platform to improve performance. Existing FPGA-based GA systems are effective in real-time applications (22; 23; 20; 27; 7; 6; 3; 31). There are two types of FPGA-based GA systems: the *application-specific* ones tailored to one specific application, with fixed chromosomes and specific genetic operators and the *general-purpose* ones supporting a wide range of chromosomes and genetic operators for different applications. For example, a GA system for the set covering problem is demonstrated in (20), with problem-specific settings. The general-purpose GA systems are the focus of this paper, we describe some examples below.

The first FPGA-based GA system HGA was proposed in 1995 (3) and its design is based on a typical GA. The system has a small speedup over the CPU-based GA work. A general-purpose GA engine is demonstrated in (6), it has speed-ups of 5 times over CPU for several GA benchmarks with binary chromosomes. The work supports a limited number of run-time changeable parameters, and has a general-purpose engine which could be embedded in other systems. However, the tuning of parameters needs hardware experience and performance is limited. We summarise the features of these previous work in the first 7 rows of Table 2. However, these previous FPGA-based systems suffer from one or more limitations:

1. The FPGA-based GA systems require a user to have significant hardware architecture knowledge before applying them to an application;

2. The systems support only one type of chromosome, binary, real-valued or permutation, which is not suitable for different applications;

3. It is hard to tune the structure and resource usage, even for an expert, as most of the architecture is fixed;

4. Modifying either GA parameters or application parameters requires time-consuming recompilation, which usually needs many hours to complete.

To address these issues, we propose an automated unified framework for creating and executing FPGA-based general-purpose GA systems, with run-time parameters and compile-time architecture parameters.

3 Automated Framework for General-Purpose GAs

Genetic algorithms have the potential to process information faster on the FPGAs than CPUs, but the existing FPGA GA systems do not well support all the features of the algorithm. We develop the automated framework under the following considerations:

1. GA is slow for complex problems, as it needs many generation to produce good solutions. High performance is the first requirement for the system.

2. Genetic algorithm is a non-deterministic algorithm with various parameters. Frequent tuning of parameters should be not time-consuming in order to find a proper configuration in a short time.

3. The FPGA is a platform demanding a deep hardware knowledge, which is a barrier for software users. Easy customisation for the hardware is helpful to non-export users.

4. The resources in hardware are limited, thus the customisation of the architecture being necessary in order to make it possible to fit the complex problems in the platform.

Most existing work meets the first requirement, while our proposed work builds on all considerations. In our framework, a general-purpose GA architecture combines with high-level user-defined chromosome and fitness function. The framework then produces a custom GA system which can be executed in hardware, providing high performance while retaining functional flexibility. A user without hardware design experience can easily create an FPGA-based GA for an application with pre-defined chromosomes while changing both GA and application parameters at run-time without time-consuming recompilation.

3.1 Automated Framework

Figure. 2 provides an overview of the proposed automated framework, showing compile-time inputs with software on the left, and run-time inputs with hardware on the right. The compile-time inputs control the scale of architecture in the hardware, while the run-time inputs carry the application and GA parameters.

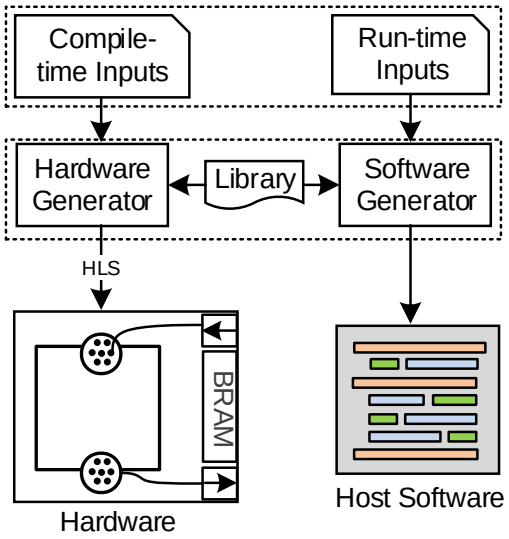

Figure 2 Automated Framework

The *framework* requires no hardware programming from users, who only need to provide the high-level inputs for application (App.) and GA.

The *customisation engine*, written in Python, automatically combines them and a number of existing templates into the following:

- Hardware code with the compile-time parameters;
- Software code with the run-time parameters.

In the hardware generation process, a high-level compilation tool Maxcompiler compiles the hardware code to low-level implementation. At run-time, the software transfers GA and application parameters to FPGAs, via input streams and on-chip memories. In this way all the requirements are met.

In the following sections, we will first describe the architecture of custom GA, and then demonstrate the reports of framework.

3.2 Custom GA

Our framework provides a scalable hardware architecture for FPGA-based GA systems improved from (20), which is referred to as a "custom GA" in this paper. Each custom GA contains all the hardware and configuration data in order to solve instances specific to a user's problem. Once compiled, the custom GA can repeatedly execute with different application parameters without time-consuming recompilation, and allows GA parameters, such as crossover rate, mutation rate and random seed, to be varied during execution.

Our custom GA is functionally flexible and scalable, allowing the user to customise the architecture for hardware resource constraints. In the custom GA, the steps of a typical GA are converted into

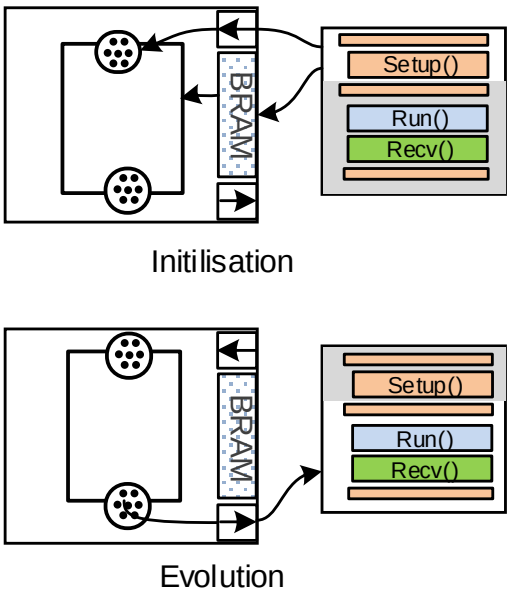

Figure 3 Custom GA

hardware functional units, which are then pipelined and parallelised.

The dataflow of a custom GA is shown in Fig. 3, labelled by the parameters described in Table 1.

3.2.1 Chromosome Configuration

To create a custom GA for a new application, a user needs to define the chromosome as an individual in one population, which can be constructed in many different forms for various applications, such as: integers, floating-point representations or bit-strings.

Unlike the previous FPGA-based systems supporting only one type of chromosomes, our custom GA system supports different kinds of chromosomes. The chromosome can be configured as a set of booleans, permutations, integers or real components. Although we could use a fixed point to represent a floating point parameter, it might be more natural to use a floating-point encoding. Our custom GA provides different genetic operators for the selected chromosome type.

3.2.2 Population Initialisation

The initial population represents the start of the evolutionary points in a genetic algorithm, as it may need many generations to produce high fitness individuals from low fitness ones. In our custom GA, there are two approaches to generate the initial population:

1. We can use random numbers, which means the initial fitness depends on the random seeds;
2. We can load a population previously generated by heuristics on a CPU, which is likely to mean a higher starting fitness.

The custom GA receives the initial population via on-chip RAM before execution begins.

3.2.3 Fitness Evaluation

The evaluation unit returns a fitness value for an individual which guides the exploring processes. In the custom GA, there are parallel evaluation units to reduce execution time. To simplify the hardware design effort, the fitness function is written in a high-level description language according to simple rules, described in section 4. Section 6 also gives some working examples for different functions.

3.2.4 Selection, Crossover and Mutation (SCM)

Selection, crossover and mutation units present in a custom GA link together into a SCM unit due to their close data coupling, making it easier to instantiate replicas for spatial parallelism. Our automated platform provides an extensive library containing different methods of selection, crossover and mutation, allowing a user to customise them for a specific application.

As shown in Table 1, a user can choose the selection method from random, roulette wheel or tournament selection. For binary chromosomes, a user can select one-point or multi-point crossover to combine different parts from parents, and then choose binary mutation inverting bits in a chromosome. For real-valued chromosomes, a user can select blending method, which generates a new value based on a linear mixture of two parents (29), and use real-valued mutation to generate a random value in the range of variables. For the permutation chromosome, a user can apply a permutation crossover and an swap mutation, which ensures to generate correct individuals.

It is usually necessary to tune crossover and mutation rates for high convergence speed, but in most previous FPGA-based GAs the adjustment requires hardware modification and recompilation, which takes many hours to complete. In contrast, our framework allows users to modify these rates at run-time without recompilation (see the examples in section 6).

3.2.5 Random Number Generator (RNG)

The Random Number Generator (RNG) plays an important role in many steps, including initial population generation, selection, crossover and mutation operations. Therefore, the quality of random numbers may affect the convergence of the algorithm.

There are two types of RNG: “True” random numbers generator (TRNG) which uses a non-deterministic source such as clock jitter in digital circuits; and a Pseudo-random number generator (PRNG), which uses a deterministic algorithm to generate random numbers.

It is common to use a PRNG on FPGA-based genetic algorithms as it is faster and smaller than a TRNG (6). Ref. (28) defines a combined Tausworthe generator, which provides a simple but effective way to generate pseudo-random numbers on the FPGAs. The random seeds can be configured by users at run-time, to explore different number sequences dynamically.

Table 1 The Parameters of User Input

	Architecture Parameters		GA Parameters
N_e	number of evaluation units	N_g	maximal generation
N_s	number of SCM units	R_u	mutation rate
M_c	crossover method: "one-point","multi-point", "blending","permutation"	R_c	crossover rate
		R_s	random seed
M_s	selection method: "roulette wheel", "random" "tournament"	I_p	initial population
		N_p	population size
M_u	mutation method: "bit-flip","constraint","swap"		

3.2.6 SCM Parallelism in Custom GA

Parallelism in architecture reduces the execution time of GAs. As shown in Fig. 3, the number of parallel evaluation units is controlled by N_E, while that of parallel SCM units is changed by N_S. A user can adjust the amount of parallelism by simply modifying the two parameters in the input, and the framework will automatically compile them to an appropriate hardware design.

In Fig. 3, N_p is the population size. If all the individuals in a population are evaluated in the cycle after filling the pipelines, there will be $N_E = N_p$ parallel evaluation instances. The feedback latency will be $L = L_E + L_S$, which is labelled on the left of Fig. 3. In this case, the resource usage will be very large when N_p is large. To solve this problem, we allow evaluation units to process a population over n cycles after filling the pipeline. Therefore, N_E is reduced to N_p/n, and the feedback latency for one generation will be $L' = L_E + L_S + n - 1$, which slightly decreases the performance if $(n-1)$ is far smaller than $L_E + L_S$.

In the same way, we can also adjust N_S to balance the resource usage of SCM units with performance. By tuning N_E and N_S according to the complexity of the evaluation and SCM units, our platform can support complex applications. An example in section 6 shows how this flexibility affects resources and performance.

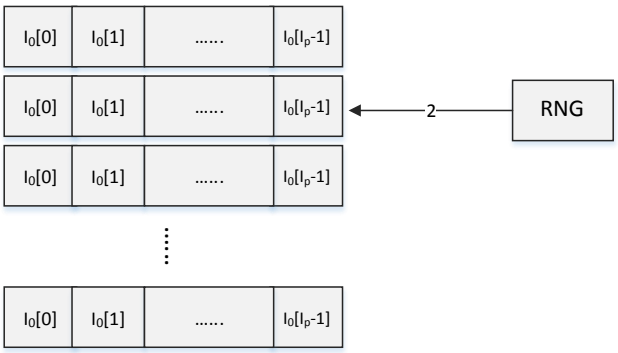

Figure 4 Hardware Construction of Random Selection

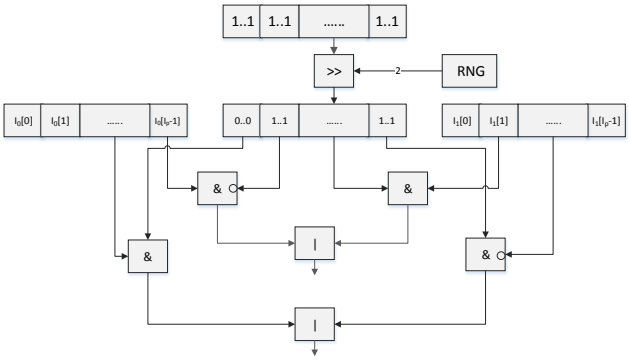

Figure 5 Hardware Construction of One-point Crossover

3.3 Hardware Construction

The hardware implementation is different with software systems and in this section we show the hardware construction of random selection, one-point crossover and exchange mutation.

The random selection is a simple selection method. First the random number generator produces a random number, and slices the first $log_2^{I_P}$ bits to generate a selection signal. We use the signal as the access address of the individual pool, as shown in Fig. 4.

In the one-point crossover, we cut one part of a chromosome and combine it with the remaining part of the other chromosome. On hardware, we use a bit-vector as the mask for the recombination, as shown in Fig. 5.

In the swap mutation, we exchange the two items containing different positions in a chromosome. On hardware, we use two bit-vectors as the masks, and do the logic computation for the exchange, as shown in Fig. 6.

3.4 Compilation and Execution Reports

The framework produces two reports automatically during the compilation and execution stages.

The *compilation report* presents the results of hardware implementation to the user, such as overall resource usages, clock frequency and any errors. Errors may be related to the syntax of the input specification, or due to resource exhaustion. Based on the report, the user can decide whether to change N_E or N_S, depending on whether there are any free resources.

During execution, the FPGA platform outputs the best fitness and solutions found over an FPGA-to-CPU stream, making the current best solution immediately available to the user. The *execution report* shows

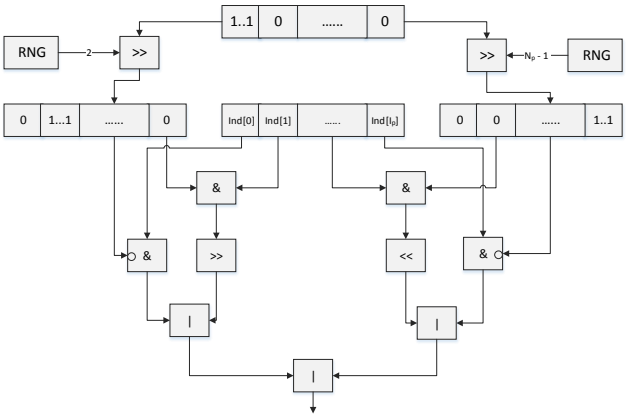

Figure 6 Hardware Construction of Swap Mutation

how different configurations and parameters affect performance. It is essentially the "answer" to the problem the user wants to solve, and also contains information suggesting the best configuration for future executions of the problems with different parameters. It is useful when solving a series of problems.

4 User defined Input

To promote the framework to non-expert users, we only require the users to provide specifications and parameters for their applications and genetic algorithm, as shown in Table 1. They are represented in a high-level domain specific language, which uses a sub-set of C. We describe the compile-time and run-time inputs in this section, and present examples in section 6.

4.1 Compile-time Inputs

The compile-time inputs contain two parts, namely application description and architecture parameters.

4.1.1 Application Description

There exists two compile-time sections in the application description:

- *CHROMOSOME* which contains the names and types of the data elements making up an chromosome. This section is declarative, describing data structures.
- *FITNESS*, which describes the fitness function used to evaluate individuals. This section contains imperative code, which uses input parameters and a chromosomes to calculate a fitness value.

The fitness function uses a sub-set of C which is then automatically compiled into a hardware implementation. The fitness function could contain all standard arithmetic operators (add, mul, etc.) as well as mathematical functions such as sin, log and exp. All components of the chromosome and application parameters are available as implicitly declared variables, which can be read either directly or as array look-ups, depending on their types.

A user can declare additional temporary variables within the fitness function of any type, and convert expressions between types, for example from fixed-point (int/uint) to floating-point (float/double). A user also can also customise the width of an integer for optimisation, for example, uint8 means unsigned 8-bit width integer.

The fitness function can contain multiple statements, which are executed sequentially. The statements can be simple assignment, if-else, or for-loops. Due to the underlying compilation strategy, we require that for-loop bounds are statically determined, so that they can later be converted into a streaming representation. These restrictions mean that certain behaviour cannot be expressed, but we show in section 6 that they can be used to capture various common problems.

4.1.2 Architecture Parameters

Our framework supports five compile-time parameters for the custom GA. At compile-time, the user can balance the resource usage of evaluation and SCM units, by changing N_E and N_S. Those architecture parameters can make the system easier to fit different scale problems. For the methods of selection, crossover and mutation, M_s, M_c and M_m can be configured for binary, permutation or real-valued chromosomes.

4.2 Run-time Inputs

Run-time inputs make it possible to change the GA system quickly without long-time recompilation. There are also two parts in run-time inputs, one are application parameters, the other are GA parameters.

4.3 Application Parameters

Most applications have input parameters passed to the fitness function, which represents a specific instance of the problem, but in the previous FPGA-based GAs, a user has to recompile the design to change any of them. To save the long compilation time, our single custom GA can support all input problems for an application by changing the *APP_PARAM* section at run-time.

4.3.1 GA Parameters

As seen from Table 1, we have six run-time parameters for the GA. At run-time, N_g controls the number of generations generated, while N_p controls the size of one population.

The framework can also try multiple combinations of run-time parameters, for example trying a list of R_c and R_m to maximise the convergence rate. Changing I_p is also useful if a good prior population exists. These GA parameters can help a user find a good configuration for

a type of application, and they all have sensible default values if a user does not specify them.

5 Qualitative Comparison

We compare the features of previous FPGA-based GAs to our framework in Table 2 (6). Our custom GA is more flexible and easier to use than other FPGA-based GA systems, with the following advantages:

1. Our framework provides a unified platform for binary, permutation, and real-valued chromosomes with a flexible structure. The selection, crossover and mutation methods (M_s, M_c, M_m) can be changed for different kinds of problems. There exists six parameters changeable at run-time, including crossover and mutation rates (R_c, R_m), the population size(N_p), generation number (N_g), the random seed (R_s), in particular application parameters and initial population (I_p), which are supported only in our platform. Specifically, the changeable application parameters make it possible to execute different inputs for an application without recompilation. Some platforms also support several run-time parameters, but require the user to have hardware knowledge to change them (6; 31).
2. The framework allows a user to decide the number of parallel evaluation (N_E) and SCM units (N_S) at a high level to balance the resource usage with performance, without any manual modification of hardware code. Furthermore, as described in subsection 3.2.6, the customisable parallelism makes it possible to support complex applications.
3. The chromosome and the fitness function of a new application are defined in a high level description language, making it easy for a user to apply our custom GA, without writing any low-level hardware code such as VHDL or Verilog.

6 Experiments

The proposed general purpose framework can deal with many problems. In our case, we choose just a number of applications, including the locating problem, maximum satisfiability problem, travelling saleman problem, and six standard benchmarks to prove our system's functionality and behaviour.

We select an FPGA-based acceleration platform containing Virtex 6-SX475T for hardware implementation (18). We compare our FPGA-based system with software and GPU-based solutions quantitatively. We implement software counterparts based on a third-party GA (25) on an 2.67GHz Dual Intel Xeon X5650 CPU system, which has 12 physical cores and 24 threads in total. The number of threads used is optimised based on the communication cost. The CPU code is well tuned with multi-threading techniques including Pthread and SIMD, and the code is compiled by Intel C compilier with highest optimisation level.

For the maximum satisfiability problem (MAXSAT), we also compare our custom GA with a third-party GPU-based work on an nVidia Tesla C1060 (21). We have compared our system with other FPGA-based systems qualitatively in the section 5, but it is nearly impossible to compare quantitatively as they use different platforms and do not provide enough details of their execution time.

6.1 Locating Problem

The locating problem is dealing with finding an emergency response unit, which has the best response time to reach any emergency that occurs in a city.

The study presented in (29) provides a complex example with a 10×10 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 cost function is:

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

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

In our case, we use floating-point chromosomes, and define high-level specification and parameters in Fig. 7 and Fig. 8. During compile-time, the chromosome and application are defined according to the rules described in section 4.

The number of parallel evaluation and SCM units can be changed via N_e and N_s. As shown in Fig. 9, we can tune those parameters in order to improve resource usage. For example, if we reduce the N_e complex evaluation unit from 4 to 2, we obtained a slightly decreased performance (5% slower) due to pipelined structure.

Fig. 8 shows the run-time parameters, array W is defined as the application parameters. By changing W, we can use the same custom GA to solve multiple input problems without recompilation, which always needs several hours to finish. Then we choose selection, crossover and mutation based on chromosome type. Here we also define a series of run-time parameters to tune convergence speed. Our framework will try a full combination of them, including lists of mutation rates (R_u), crossover rates (R_c), and random seeds (R_s).

The execution report helps users to find the best configuration for the problem. To compare execution time with the multi-core CPU, we let the custom GA run 1,000,000 generations for different population size (N_p).

As shown in Fig. 10, our custom GA is 24 times faster than on the multi-core CPU. Although the initial compilation of the custom GA is slow, future executions of the same GA with different parameters require no compilation, and start evaluating immediately.

Table 2 Qualitative Comparisons of FPGA-based GAs (Chrome: Chromosome)

	Year	Chrome.	Run-time GA param.	App. param.	Parallel param.	Initial pop.	App. level	Platform
(3)	1995	binary	-	fixed	-	rand	low	BORDG
(7)	1999	binary	-	fixed	-	rand	low	SFL
(22)	2001	binary	-	fixed	-	rand	low	AXB-MP3
(23)	2001	binary	-	fixed	-	rand	low	Xilinx V1000
(31)	2004	binary	N_p, N_g, R_c, R_m	fixed	-	rand	low	PCI System
(27)	2009	binary	N_p	fixed	-	rand	low	Virtex2 Pro
(6)	2010	binary	N_p, N_g, R_m, R_c, R_s	fixed	-	rand,	low	Virtex2 Pro
ours	2013	binary, real-valued permutation	$N_p, N_g, R_m, R_s, R_c, I_p$	run -time	N_E, N_S	rand, I_p	high	MAX3 (V6-SXT475)

Compile-time Input

```
Chromosome {float xf,yf;}
Fitness {
  float cost = 0.0;
  for ( int i = 0; i ≤ 9; i ++ ){
    for(int j = 0; j ≤ 9; j ++){
      float xn = i + 0.5;
      float yn = j + 0.5;
      cost += W[i][j]*sqrt((xn - xf)*
      (xn - xf)+(yn - yf)*(yn - yf)); }
  }
  return cost;
}
Arch_Param{
  Ne = 2; Ns = 8;
  Ms: "tournament selection"
  Mc: "blending crossover"
  Mu: "real-valued mutation"
}
```

Figure 7 The Compile-time User Input for Locating Problem

Run-time Input

```
App_Param{ int8 W[10][10] =
          {{0,6,...} {4...}...};
          }
GA_Param{
  Ng = 1,000,000
  Np = 32
  Rc = 0.6, 0.5, ...
  Ru = 0.01, 0.02, ...
  Rs = 0x1234, 0xffff
  Ip = {(3.1, 4.5),(2.3, 4.8)...}
}
```

Figure 8 The Run-time User Input for Locating Problem

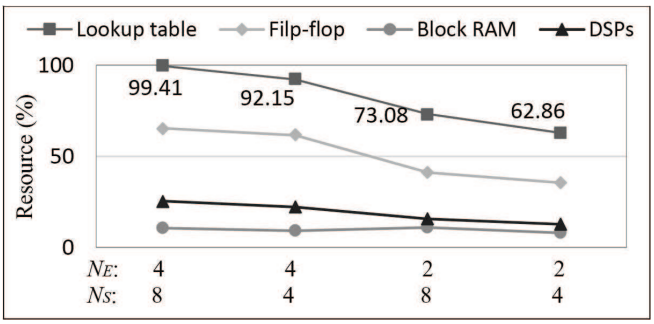

Figure 9 Resources for various N_e and N_s

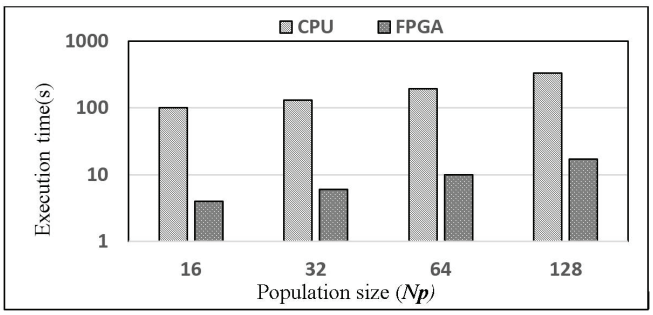

Figure 10 The execution time for various N_p

6.2 The Set Covering Problem

The set covering problem (SCP) is a classic NP-hard combinatorial optimisation problem which has many practical applications (26). For example in hardware verification, a suite with M programs can test or cover N functions, and every program has a cost. The relation between suite and functions can be represented by an $N \times M$ matrix. The aim of the SCP is to find the lowest cost sub-suite of programs which tests all N functions.

For a candidate suite, the fitness is computed as:

$$Fitness = p \times coverage(suite) - cost(suite) \quad (2)$$

where p adjusts the fitness scale, and *coverage* represents how many functions covered.

As shown in Fig. 11, in our case, the chromosome is designed as an m-bit binary, and the i-th bit of the binary determines the exist of i-th program in the suite. In the fitness section, we define a function like *countone*() to calculate the number of 1. The coverage array ($cov[M]$), cost array ($cost[M]$) and p are supplied at run-time. We also choose different methods of selection, crossover and mutation from the locating problem.

We test an instance of Steiner triple systems (STS_{27}), which is considered as a hard SCP with a matrix of 117×27 (24).

Our custom GA is 45 times faster than the CPU.

Compile-time Input

```
Chromosome { uintM suite; }
Fitness{
  uint cost = 0; uintN cov = 0;
  for ( int i = 0; i ≤ M-1; i ++ ){
    cov |= suite[i] ? covs[i]: 0;
    cost += suite[i] ? costs[i] : 0; }
  uint covs_n = countone (cov);
  return (p * covs_n - cost);
}
uint6 countone(uintN cov) {...}
Arch_Param{
  M_c: "multi-point crossover"
  M_s: "roulette wheel selection"
  M_u: "binary mutation"
}
```

Figure 11 The Compile-time User Inputs for Set Covering Problem

Run-time Inputs

```
APP_PARAM{ int p = 2,
           uintN covs[M]={...},
           uint cost[M]={...} }
GA_PARAM{
  ...
  I_p = {0xffff, 0x1, ...}
}
```

Figure 12 The Run-time User Inputs for Set Covering Problem

6.3 Maximum Satisfiability Problem

The Maximum Satisfiability Problem (MAXSAT) is another classic NP-hard problem used to determine an optimised assignment of a set of boolean variables $\mathbb{V} = \{V_1, V_2, \ldots, V_u\}$. A literal is a boolean variable or its negation, i.e. $L \in \{V, \neg V\}$. A clause C_k is the disjunction ("*or*") of m_k literals in the form

$$C_k = \bigvee_{i=1}^{m_k} L_{ki} \tag{3}$$

A formula F in the conjunctive normal form is defined as the conjunction ("*and*") of M clauses, i.e.

$$F = \bigwedge_{k=1}^{M} C_k = \bigwedge_{k=1}^{M} (\bigvee_{i=1}^{m_k} L_{ki}) \tag{4}$$

Let $\Xi = (V_1/v_1, V_2/v_2, ..., V_u/v_u)$ be an assignment of $\mathbb{V}$. Then the optimisation target of the MAX-SAT problem is

$$g(\Xi) = \max_{\Xi} \sum_{k=1}^{M} \delta(C_k|_{\Xi}) \tag{5}$$

where $\delta(C_k) = \begin{cases} 1 & \text{if } C_k = \text{true} \\ 0 & \text{otherwise} \end{cases}$

We define Ξ as a binary chromosome containing multiple booleans and describe the high-level inputs in Table 3. Here we apply the custom GA system to a hard instance uf100-430, which contains 100 variables and 430 clauses (1).

Compared with the GPU-based work (34) for same problem, our system obtaines a 29 times speedup.

6.4 Travelling Salesman Problem

The Travelling Salesman Problem is a well-known NP-hard problem in combinatorial optimisation 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 origin city.

In our case we test a 64 cities TSP. We use permutation chromosomes with 64 items, each of them representing a city index. The order of the chromosome represents the sequence in which the cities been visited. We also define the distance array as the application parameters.

For this problem, we use 'swap' mutation and 'permutation' crossover, to make sure all the new individuals are correct.

6.5 GA Benchmarks

To test the ability of dealing with numeric computation in our custom GA, we use three GA benchmarks from (6), including binary F6 (BF6), binary F7 (BF7) and 2-D Shubert function (2DS), and one benchmark called F11 from (29). As shown in Table 4, these functions have one or more parameters.

6.6 Experiments Summary

Our platform can output the results from the FPGA to the CPU for comparison. As shown in Table 4, our platform can effectively solve different applications with an average speed up of 29 times, including discrete combinatorics, numeric, real-valued and permutation computation.

Our custom GA can find a location with 96% of best fitness according to (29) for the locating problem, and find good solutions for all other applications. The clock frequency of the custom GA is set to a conservative default value of 75MHz, so with longer compile times higher speed-ups are possible. In the Table 4, the

Table 3 The User Inputs for MAXSAT

Compile-time Inputs	Run-time Inputs

`Chromesome { bool v[100]; }`
`Fitness{`
`  bool c[430]={false};`
`  uint count=0;`
`  c[0] = v[25]|(!v[98])|v(6);`
`  ...  // remove for space`
`  c[429] = v[59]|v[91]|(!v(72));`
`  for ( int i=0; i≤429; i++)`
`  count += c[i] ? 1:0;`
`  return count; }`
`}`
`Arch_Param {`
`  $N_e$=2; $N_s$=2;`
`  $M_c$:"multi-point crossover";`
`  $M_s$:"roulette wheel select";`
`  $M_u$:"bit-flip mutation"; }`
`}`
	`App_Param{ }`
	`GA_Param`
	`{`
	`  $N_p$ = 32;`
	`  $N_g$ = 1000000;`
	`  $R_u$ = 0.065;`
	`  $R_c$ = 0.65;`
	`  $R_s$ = 0xffffff;`
	`  $I_p$ = {b10...1`
	`  ,...}`
	`}`

Table 4 Resources (Res.), Solution Quality and Speed-ups

App.	N_p	N_E	N_S	Res.%	Speed-up	Quality	Description
Locating	32	2	8	73.08	24	96%	real-valued computation
STS_{27}	128	16	16	65.12	45	100%	discrete combinatorics
MAX-SAT	64	8	8	74.96	22	100 %	
TSP	32	1	1	77.54	28	90 %	permutation chromosomes
BF6	32	8	8	26.45	26	100%	
BF7	32	8	8	21.48	25	100%	numeric computation
2DS	32	8	8	68.89	31	100%	
F11	32	8	8	60.05	27	100%	real-valued computation
MEAN		-	-	-	29	-	-

resource usages vary by the complexity of an application, the number of evaluation units (N_E) and SCM units (N_S).

Reference (6) gives speed-ups of 5 times over BF6, BF7 and 2DS without reporting exact execution time, so we cannot directly compare its performance with ours. Although based on high-level inputs, our custom GA can still achieve high performance while retaining flexibility, with parallelled and pipelined units.

7 Conclusion and Future Work

Genetic algorithms are ideal candidates for FPGA acceleration due to their long execution time. To provide an easy way for users to create and execute FPGA-based GAs with binary, real-valued and permutation chromosomes, we propose an automated unified framework for whole general-purpose GA systems.

Our framework contains a scalable and customisable custom GA, which allows a user to tune the resource usage, without directly modifying hardware design. At compile-time, a user just needs to define a high-level specification of an application, including chromosome and fitness function, without writing any hardware code using VHDL. At run-time, the user can change GA and application parameters without waiting for recompilation.

When compared with existing FPGA-based GAs, our custom GA has more architectural flexibility, making it much easier for users to take advantage of FPGA acceleration. Compared with a multi-core CPU over six applications, the average speed-up of the custom GA is 29 times.

In the future, we will allow users to enhance the library in the framework to support more genetic operators. We will also improve the framework by supporting variable length chromosomes, automatic parameter decision and structure tuning due to the flexibility of FPGA platform.

References

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

[2] S. N. Sivanandam, and S. N. Deepa. Introduction to Genetic Algorithms. Springer Berlin Heidelberg. 2008.

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

[4] 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.

[5] 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.

[6] P. R. Fernando, 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.

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

[8] Y. Choi, and D. J. Chung. "VLSI processor of parallel genetic algorithm." IEEE Asia Pacific Conference on ASIC. pp. 143-146, 2000.

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

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

[11] 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.

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

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

[14] P. V. d. Santos, J. C. Alves, and J. C. Ferreira. "A framework for hardware cellular genetic algorithms: an application to spectrum allocation in cognitive radio." 23rd International Conference on Field Programmable Logic and Applications (FPL). pp. 1-4, 2013.

[15] J. M. P. Cardoso, P. C. Diniz, and M. Weinhardt. "Compiling for reconfigurable computing: a survey." ACM Computing Survey. Vol. 42, No. 4, pp. 1-65, 2010.

[16] T. J. Todman, G. A. Constantinides, S. J. Wilton, O. Mencer, W. Luk & P. Y. Cheung, "Reconfigurable computing: architectures and design methods." Proceedings on IEEE Computers and Digital Techniques, vol. 152, no. 2, pp. 193-207, 2005.

[17] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software." ACM Computing Surveys, vol. 34, no. 2, pp. 171-210, 2002.

[18] Maxeler Tech. "Programming MPC Systems White Paper." 2013.

[19] D. B. Thomas, and W. Luk. "The LUT-SR family of uniform random number generators for FPGA architectures." IEEE Transactions on Very Large Scale Integration Systems (VLSI) Vol. 21, No. 4, pp. 761-770, 2013.

[20] L. Guo, D. B. Thomas, and W. Luk. "Customisable Architectures for the Set Covering Problem." International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART). pp. 69-74, 2013.

[21] A. Munawar, et al. "Hybrid of genetic algorithm and local search to solve MAX-SAT problem using nVidia CUDA framework." Genetic Programming and Evolvable Machines. Vol. 10, No. 4, pp. 391-415, 2009.

[22] B. Shackleford, G. Snider, & R. Carter, "A high-performance, pipelined, FPGA-based genetic algorithm machine," *Genetic Programming and Evolvable Machines*, vol. 2, no. 1, pp. 33-60, 2001.

[23] C. Aporntewan & P. Chongstilivatana, "A hardware implementation of the compact genetic algorithm," *Proceedings of the 2001 Congress on Evolutionary Computation*, vol. 1, pp. 624-629, 2001.

[24] C. Plessl. & M. Platzner. "Custom computing machines for the set covering problem," *Proceedings of 10th IEEE Symposium on Field-Programmable Custom Computing Machines*. pp. 163-172, 2002.

[25] D. A. Coley, *"An introduction to genetic algorithms for scientists and engineers,"* Singapore: World Scientific Publishing, 2003.

[26] E. Balas, "A class of location, distribution and scheduling problems: Modeling and solution methods," 1982.

[27] M. Vavouras , K. Papadimitriou, & I. Papaefstathiou, "High-speed FPGA-based implementations of a genetic algorithm," *In Systems, Architectures, Modeling, and Simulation*, pp. 9-16, 2009.

[28] P. L Ecuyer, "Tables of maximally equidistributed combined LFSR generators," *Mathematics of computation*, vol. 68, no. 225, pp. 261-269, 1999.

[29] R. L. Haupt & S. E. Haupt, *Practical genetic algorithms*, John Wiley & Sons, 2004.

[30] S. N. Sivanandam & S. N. Deepa, *Introduction to genetic algorithms,* Springer, 2007.

[31] W. Tang & L. Yip, "Hardware implementation of genetic algorithms using FPGA," *47th IEEE Midwest Symposium on Circuits and Systems*, pp. 549-552, 2004.

[32] Blind for Review.

[33] Blind for Review.

[34] A. Munawar, et al. Hybrid of genetic algorithm and local search to solve MAX-SAT problem using nVidia CUDA framework. Genetic Programming and Evolvable Machines. Vol. 10, No. 4, pp. 391-415, 2009.

[35] M. Pedemonte, E. Alba, and F. Luna. Towards the design of systolic genetic search. IEEE Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), pp. 1778-1786. 2012.