Automated Framework for FPGA-Based Parallel Genetic Algorithms

Liucheng Guo, David Thomas
Dept. of EEE
Imperial College London, UK
Email: {gl512, dt10}@ic.ac.uk

Ce Guo, Wayne Luk
Dept. of Computing
Imperial College London, UK
Email: {ce.guo10, w.luk}@ic.ac.uk

***Abstract*—Parallel genetic algorithms (pGAs) are a variant of genetic algorithms which can promise substantial gains in both efficiency of execution and quality of results. pGAs have attracted researchers to implement them in FPGAs, but the implementation always needs large human effort. To simplify the implementation process and make the hardware pGA designs accessible to potential non-expert users, this paper proposes a general-purpose framework, which takes in a high-level description of the optimisation target and automatically generates pGA designs for FPGAs. Our pGA system exploits the two levels of parallelism found in GA instances and genetic operations, allowing users to tailor the architecture for resource constraints at compile-time. The framework also enables users to tune a subset of parameters at run-time without time-consuming recompilation. Our pGA design is more flexible than previous ones, and has an average speedup of 26 times compared to the multi-core counterparts over five combinatorial and numerical optimisation problems. When compared with a GPU, it also shows a 6.8 times speedup over a combinatorial application.**

I. Introduction

Genetic algorithms (GAs) have been shown to be effective in solving optimization problems. Parallel GA (pGA) is a type of GA which contains a number of GA instances evolving and exchanging information in parallel. As the search processes of GA instances are conducted in different areas of the search space, pGAs can both decrease execution time and improve the quality of solutions [1]. However, CPU-based pGAs may need acceleration for real-time problems, and are not suitable for embedded systems due to the high power consumption [14].

There have been previous attempts to adapt pGAs to FPGAs for acceleration or low power consumption [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]. However, designing an FPGA-based pGA is not as easy as implementing multiple hardware blocks supporting a set of GA instances. Major challenges behind a good design include the following. First, the information exchange between GA instances requires communication. The design of the commutation scheme is closely related to hardware resource consumption as well as the optimisation power. Second, pGA systems are more complex than single-thread ones because the user needs to control the parallelism in addition to the optimisation routine. This complexity brings difficulties to users especially for unprofessional ones. Third, it might be necessary to recompile the system after modification. As a compilation may take a long time, a useful pGA design should be able to avoid unnecessary recompilation. To address these challenges and make the pGA designs accessible to non-expert users, we proposes a pGA framework with the following contributions:

1) A general-purpose pGA framework for creating and running multiple GA instances with an effective communication scheme. (Section III.D)
2) A customisable pGA architecture which exploits both fine and coarse-grained parallelism, allowing non-expert users to specify problem settings and architecture options in a high level without hardware design expertise. (Section III and IV.B)
3) A fine-tuning mechanism for the pGA architecture, enabling users to tune a set of run-time parameters without recompiling the design. (Section IV.C)

After a qualitative comparison with existing FPGA-based designs, our work shows improved flexibility in hardware architecture and run-time tuning. When compared with a multi-core CPU, our design shows an average speedup of 26 times for an NP-hard combinatorial application and four numerical benchmarks; when compared with a GPU, it also shows a 6.8 times speedup over the combinatorial application.

II. Background and Related Work

A. Genetic Algorithms

Genetic algorithms emulate the natural evaluation process with genetic operators, including *selection*, *crossover* and *mutation*. A typical GA evolves an *initial population* of tentative solutions (called *individuals*) towards better ones. Every individual is a genetic representation (*chromosome*) of a candidate solution in the search space, and associated with a fitness value by an *evaluation* function. Based on their fitness values, pairs of individuals are chosen by the selection operation. Then two individuals in each pair exchange their information by crossover, and are modified by mutation operators to gain diversity. The evolution process is repeated for many iterations (called *generations*) until a termination condition is reached, for example maximal generation number is reached. GAs can perform well when has a proper trade-off between exploration on search space and the exploitation of good solutions [1].

When applying GA to a problem, a user should first design the chromosomes and define a fitness evaluation function for the solution domain. The chromosome is often defined as a binary encoding (a vector of bits), with the evaluation function re-interpreting sub-sequences of the binary chromosome as booleans, permutations, integers, or real components. Except

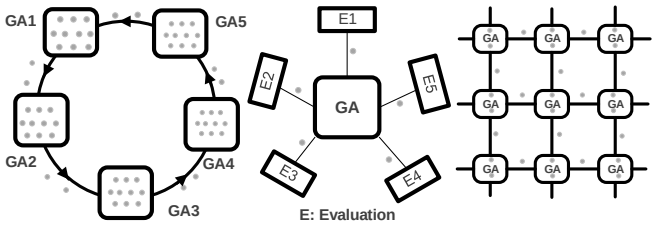

Fig. 1. Distributed (left), Master-slave (middle) and Cellular (right) pGAs

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.

B. Parallel GA

Genetic algorithms may get stuck in a local optimum, because the population will tend to cluster around the best individual, so natural selection limits exploration of the surround space. To explore a wider search space, parallel GAs are proposed to invoke multiple GA instances evolving with a group of sub-populations in parallel. Each GA instance can carry identical or divergent parameters, and the divergent parameters may result in different behaviours in each GA instance [2].

The simplest pGA structure is the independent pGA (iGA), which runs multiple parallel GA instances starting from different initial populations, and with no mixing or communication between instances during execution. A more popular and effective way is to establish communication between GA instances, so we have coarse-grained pGAs and fine-grained pGAs (see Fig. 1). A typical coarse-grained pGA is the distributed GA (dGA), in which every GA instance maintains a sub-population and the communication of instances is conducted by the infrequent migration. Master-slave GA is a popular fine-grained pGA, contains multiple evaluation units valuing individuals in parallel. Another type of fine-grained pGA is the cellular GA (cGA), when every GA instance maintains several (usually two or three) individuals and frequently communicates with its neighbours in a net architecture. All these pGAs have potential for hardware acceleration due to their natural parallelism.

C. FPGA-based Parallel GA

FPGAs balance the high performance and parallelism of hardware with the flexibility of software, and so are a promising platform for acceleration. With the requirements of real-time computing and low energy consumption, GAs were first adapted for FPGAs in 1995 [3]. In addition to a considerable number of conventional GA systems mentioned in [4], [5], [6], [20], FPGA-based master-slave GAs and dGAs have been demonstrated [7], [8], [9], [10], [11]. FPGA-based cGAs are also proposed in [12], [13], [14]. In the top 6 rows of Table II we summarise the features of these existing pGA systems. Master-slave GAs and dGAs are likely to be easier for FPGA place and route because of the simple and few connections between GA instances, so there seem to be more systems for them in FPGAs. For example, Kamimura implement a dGA processor consisting of a migration unit and four GA instances [11]. Their system is applied to three simple numeric functions. In the paper we also focus on the pGA with master-slave and distributed models, to exploit both fine and coarse-grained parallelism.

Existing pGA systems suffer from at least one of the following problems: 1) The system neglects the flexibility of architecture, even an expert cannot easily tune resource consumption against performance; 2) The system is designed for specific problems in a low-level hardware language, such as VHDL or Verilog, which is almost impossible for non-expert users to modify to support new applications; 3) Tuning GA or application parameters requires recompilation, which always takes hours to complete.

To address these issues, we proposes a general-purpose framework to create flexible pGA systems based on a high-level description. To simplify the design process, a streaming high-level synthesis tool Maxcompiler is used as an intermediate-level target [15], [16], [17], [18]. We will show how our proposed work addresses these problems in section III and IV.

III. Customisable Architecture of GA Kernel

Each GA instance in our pGA system is executed in a customisable kernel, which is called GA kernel in this paper. The overall and data processing of a GA kernel are shown in Fig. 2. In the kernel, the basic steps of a typical GA mentioned in the background have been implemented as hardware units, which are fully pipelined and highly parallelised. The details of these basic units are described in this section.

A. Initialisation Unit

Before using a GA to optimise a function, it is necessary to define the representation of chromosomes and to generate an initial population. The chromosome can be a set of booleans, permutations, integers or real components. The GA kernel will support each possible set with different sets of genetic operations.

The initial population (I_p) determines the number and positions of start points in the search space. To spread the starting points of the GA kernels over the search space, the initialisation unit offers two ways to specify the initial population: a random number generator; or loading pre-computed sub-populations.

B. Evaluation Unit

The evaluation unit returns a fitness value for an individual, which guides the exploring processes. Master-slave GAs instantiate multiple evaluation units in parallel to reduce execution time, so we also use this model and define the degree of parallelism for the evaluation units as N_e. By customising N_e, we can balance the resource usage against performance. For example, in Fig. 2, the sub-population size (N_p) is four. If we have four evaluation units ($N_e = 4$), it will need ($N_p/N_e = 1$) cycle to finish all the four evaluation operations after filling the pipeline, but when we reduce N_e by half ($N'_e = N_e/2 = 2$), it will need only one extra cycle in the pipeline ($N_p/N'_e - N_p/N_e = 1$), which just slightly reduces the performance.

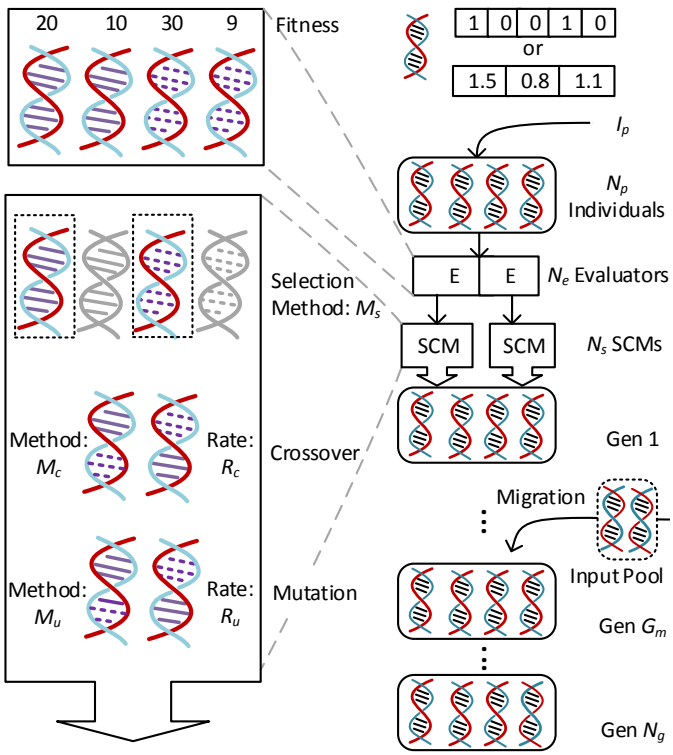

Fig. 2. Parameters and Data Processing in a GA Kernel

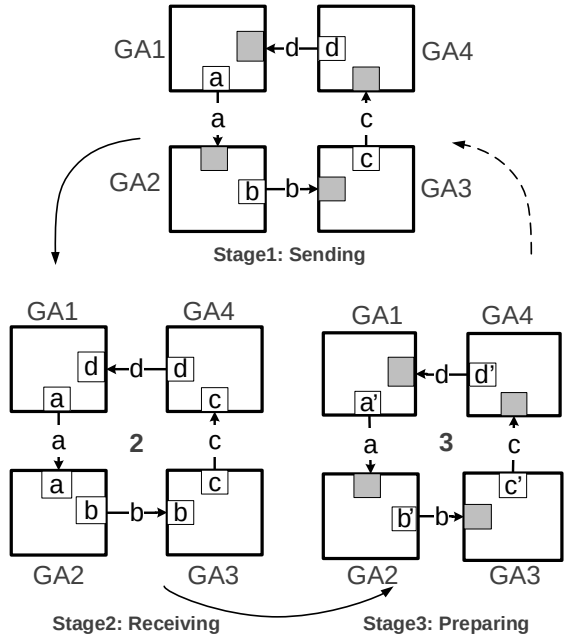

Fig. 3. Inter-Kernel Communication for Migration

C. SCM Unit

As shown in Fig. 2, after evaluation, the three genetic operators, namely selection, crossover and mutation, will deal with two individuals sequentially. This close relationship results in a great potential for processing pairs in parallel. Therefore, we combine these operators into a single SCM (selection, crossover and mutation) unit and control the number of SCM units by a compile-time parameter N_s. We can tune N_s independently of N_e, allowing us to balance resource utilisation and performance. Every SCM unit consists of one selection, one crossover and two mutation operators, processing two individuals concurrently. The specific methods (M_s, M_c, M_u) used for the three genetic operators as well as the rates for crossover and mutation (R_c, R_u) are customisable.

D. Migration Unit

Migration units control how the data are exchanged between GA kernels. A migration unit has an input pool and an output pool for receiving and sending individuals. We use a unidirectional ring to transfer a part of sub-population to one of its neighbours. Three parameters in a pGA system decide the migration policy:

1) Migration Gap and Rate: The migration gap (G_m) controls the frequency of data exchange and the migration rate R_m decides the probability of exchanging the data.

2) Number of Migrants: The parameter N_m determines the number of individuals to be exchanged during one migration.

3) Method of Migrants Selection: This parameter (M_m) decides how to select the migrants, either the current elite or random individuals.

Those parameters are all customisable in the migration unit. To avoid waiting for data from the neighbour kernel, we continually stream data in the links and control when to accept them. The three stages of inter-kernel communication are shown in Fig. 3. For example, at the initial stage, the four kernels GA1, GA2, GA3 and GA4 place *a*, *b*, *c* and *d* in their output pools (depth = N_m), then keep sending them to their neighbours respectively. However, although receiving the data, their neighbours do not update pools, until G_m generations have been evolved. Afterwards, the data in input pools are updated to *d*, *a*, *b* and *c*, while signals emitted from all GA kernels still remain unchanged. At the end of next generation: the data in output pools will be updated to *a'*, *b'*, *c'* and *d'*, which reaches preparing stage. In this way, we avoid the unnecessary waiting for new data as the streaming links always have data.

During the data processing, the random number generator (RNG) produces random numbers for initial population, genetic operators and migration. We implement an RNG based on a method [19]. The sequence of random number generated can be customised by a random seed R_s.

IV. Automated Framework

The framework incorporates both hardware and software, improved from a conventional GA system in an earlier work [20]. Our framework involves the customisable GA kernels in section III to provide a general-purpose system. An overview of the design and execution flow is shown in Fig. 4, while a summary of its parameters is listed in Table I.

A. Problem-Independent Library

As mentioned in the background, genetic operators are essentially problem independent, so it is possible to implement them in advance, before knowing the specific application a user will use them in. We have built a library consisting

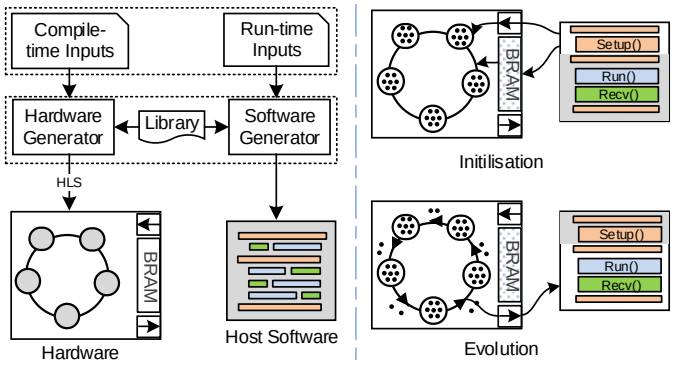

Fig. 4. Design Flow (left) and Execution Flow (right)

of different genetic operators, parameterisable GA kernels, and host software. Various methods of performing genetic operations are available for use in different circumstances, selectable by the user. For selection, roulette wheel and tournament methods are offered. For mutation operation, we support: bit-flip, which randomly inverts bits within the chromosome; swap, to exchange integers within a permutation chromosome; and constraint, which generates random numbers within some set bounds. Obviously, the swap method is suitable for the chromosomes of sequential ordering problems, and the constraint method is suitable for real-valued problems. For the crossover operation, we support: one-point and multi-point binary exchange; blending between real values [22]; and reordering for permutations. Although a real value can be mapped to integers or booleans, it may be more natural to use the real-valued encoding and related methods. With this library, our framework can support both combinatorial and numerical optimisation.

B. Design Flow

In the design flow shown in Fig. 4, the inputs of the framework are compile-time and run-time specifications, while the outputs are the hardware implementation and host software executable. Compile-time inputs include hardware-dependent parameters and a high-level C-like description of the optimisation problem. The problem-independent library provides the architecture template and methods of genetic operators. The hardware generator will combine them together to create a full FPGA implementation by calling a high-level synthesis tool Maxcompiler [18]. The generated hardware also has input and output channels, and BRAM for communication. The run-time inputs are hardware-independent parameters that control the behaviour of the algorithm. After reading the inputs from file or console, the software generator loads these parameters into the host software, which will send them to the hardware.

The compile-time inputs include two following parts:

1) Architecture Parameters (Arch_Param): The architecture can be easily configured by the user as we offer seven compile-time architecture parameters, including the degrees of parallelism through the numbers of parallel kernels (N_k), SCM units (N_s) and evaluation units (N_e), methods of genetic operations (M_s, M_c, M_u) and migrants selection (M_m).

In this way, users gain the ability to control the hardware layout via the parametrisation of external and internal parallelism, without needing to understand how to modify the FPGA architecture directly. In a kernel, N_e = N_p and N_s = $N_p/2$ imply maximal parallelism and resource utilisation, but minimal execution time per generation. Naturally, if the whole population size $N_P = N_k \times N_p$ exceeds the available hardware resources then multiple rounds of N_k, N_s and N_e variation might be necessary. Our framework enables a user to select suitable values of these parameters to maximize FPGA resource utilisation or performance.

TABLE I. THE PARAMETERS OF USER INPUT

	Architecture Parameters		GA Parameters
N_k	number of GA kernels	N_p	sub-population size
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”,“reordering”	R_c	crossover rate
		R_s	random seed
M_s	selection method: “roulette wheel” “tournament”	I_p	initial population
		N_m	number of migrants
M_u	mutation method: “bit-flip”,“constraint”,“swap”	G_m	migration gap
M_m	migrants selection method: “elite”, “random”	R_m	migration rate

2) Application Description: The input optimisation problem is written in a C-like programming language, in order to simplify the design process for a non-expert user. A description contains two parts: 1) chromosome structure, which defines the items making up the genetic representative; 2) fitness function, which can contain for-loop statements (with non data-dependent bounds), mathematical functions and any necessary variables including application parameters and temporary variables. Examples will be provided in section V to show the descriptions for different applications.

C. Execution Flow

The host software in our framework controls the operation of the FPGA system and divides the execution flow into two steps shown in Fig. 4: initialisation and evolution. During the initialisation step, the host software loads run-time parameters into hardware by streaming them into the input channel or storing them in the BRAM. In the evolution step, the hardware pGA system evolves and outputs solutions to the host software.

The run-time user inputs contain application parameters and GA parameters. Although the time for the first compilation is limited by place-and-route speed, the system can be modified without recompilation by updating these parameters, including:

1) Application Parameters (App_Param): A specific problem can have various configurations but they only differ in terms of some parameters. For example, the function F7 in section V.B has a parameter a, so we can tune the value of a at run-time after the system is created.

2) GA Parameters (GA_Param): Nine GA parameters are available for each kernel, including maximal generation number (N_g), sub-population size (N_p), initial population (I_p), random number seed (R_s), crossover and mutation rates (R_c, R_u), number of migrants (N_m), migration gap (G_m) and rate (R_m). It is quite common for a user to tune these parameters at run-time, either globally to improve efficiency, or locally to give each individual GA kernel different parameters. Our system is more flexible than existing ones because in those systems, these parameters are fixed across all GA kernels, and the tuning of them also needs hours because of recompilation.

TABLE II. QUALITATIVE COMPARISONS OF FPGA-BASED PGAS

Work	Year	Operation Rates	Operation Methods	Migration Parameters	Application Parameters	Kernels' Parameters	Parallel Evaluation	Parallel SCM	Initial Population	Custom Language	Platform
[7]	1999	fixed	one type	none	none	identical	yes	no	–	VHDL	SFL
[8]	2000	fixed	one type	none	none	identical	yes	no	–	VHDL	PCIGEN10K
[9]	2006	fixed	one type	none	none	identical	yes	no	rand	C-like	Stratix EPS1S10
[10]	2006	fixed	one type	none	none	identical	no	no	rand	VHDL	Cyclone FPGA
[11]	2012	fixed	one type	none	none	identical	no	no	rand	VHDL	Virtex 4 (XC4VLX25)
[14]	2013	fixed	one type	none	none	identical	no	no	rand	C-like	Virtex 6 (XC6VLX240T-1)
Ours	2014	run-time R_u, R_c	multiple types	M_m, N_m R_m, G_m	run-time App_Parm	identical, divergent	N_e	N_s	run-time I_p, rand	C-like	Virtex 6 (SX475T)

D. Qualitative Comparison

We summarise the features of our framework and compare with other systems mentioned in the related work. In Table II, it is clear that our pGA system is easier to use and more flexible than others because: 1) Many parameters in Table I are modifiable at run-time, enabling a user to effectively tune them in any kernel without time-consuming recompilation. 2) We gain the maximal flexibility and parallelism in the architecture, the degrees of both external and internal parallelism can be tuned by non-expert users. 3) Our work is a general-purpose platform for different chromosomes based on a problem-independent library. 4) All the problem-dependent parts of a typical GA can be specified in a high level programming language, requiring little knowledge about the underlying hardware or tools from the users.

V. EXPERIMENTS

The proposed framework can handle many problems, but for space reasons here we choose an NP-hard combinatorial optimisation problem and four numeric ones. We select an FPGA-based acceleration platform containing Virtex 6-SX475T for hardware implementation [18]. We compare our FPGA-based pGA with software and GPU quantitatively. We implement software counterparts based on a third-party GA [21] 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 number of GA kernels and 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 pGA system with a third-party GPU-based work on an nVidia Tesla C1060 [23]. We have compared our design with other FPGA-based systems qualitatively in section IV, but it is nearly impossible to compare quantitatively as they use different platforms and do not provide enough details of their execution time.

A. Maximum Satisfiability Problem

MAXSAT is a classic NP-hard problem 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

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

TABLE III. 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_k`=4;`
`  K1{ `N_e`=2; `N_s`=2; `M_m`="elite";`
`  `M_c`:"multi-point crossover";`
`  `M_s`:"roulette wheel select";`
`  `M_u`:"bit-flip mutation"; }`
`  K2{...} ... K4{...} //kernel `N_k
`}`
	`App_Param{ }`
	`GA_Param{`
	`  K1{`
	`  `N_p` = 32;`
	`  `N_g` = 1000000;`
	`  `R_u` = 0.065;`
	`  `R_c` = 0.65;`
	`  `R_s` = 0xffffff;`
	`  `N_m` = 4;`
	`  `G_m` = 8;`
	`  `R_m` = 0.8;`
	`  `I_p` = {b10...1`
	`  ,...}`
	`  }`
	`  K2{...}//kernel 2`
	`  ...`
	`  K4{...}//kernel `N_k
	`}`

TABLE IV. RESOURCE USAGE AND COMPILATION TIME

Configuration	GA1	pGA2	pGA4/iGA4	pGA8
kernel number N_k	1	2	4	8
N_e in one kernel	8	4	2	1
N_s in one kernel	8	4	2	1
Resource Usage (%)	74.96	50.46	41.44	32.46
Compilation Time	5h38m	4h02m	2h28m	1h49m

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

$$F = \bigwedge_{k=1}^{M} C_k = \bigwedge_{k=1}^{M} (\bigvee_{i=1}^{m_k} L_{ki}) \tag{2}$$

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 MAXSAT problem is

$$g(\Xi) = \max_{\Xi} \sum_{k=1}^{M} \delta(C_k|_{\Xi}) \tag{3}$$

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 III, then a hardware implementation with four kernels will be created. As shown, the parameters can be divergent for every kernel. Here we apply the pGA system to a hard instance uf100-430, which contains 100 variables and 430 clauses [1].

1) Resource Usage and Compilation Time: We use our framework to generate FPGA-based pGA systems with different compile-time configurations, namely GA1, pGA2, pGA4, pGA8, and iGA4. GA1 is a conventional architecture with

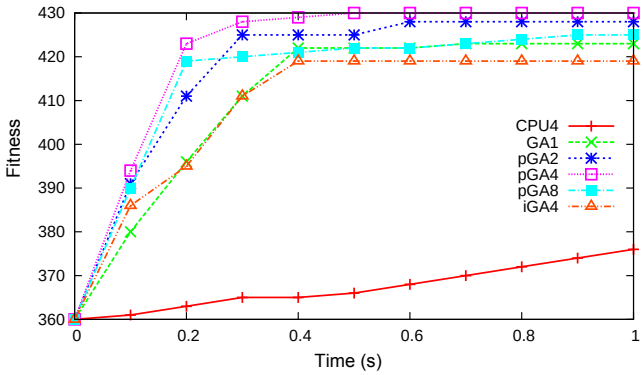

Fig. 5. Best Fitness Found in Scale of Time

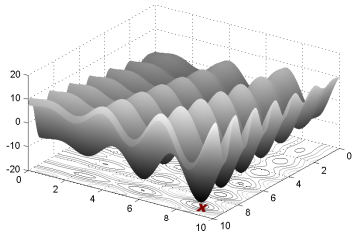

Fig. 6. Search Space of F7

```
Chromosome { float x, y; }
Fitness {
    float result = 0.0;
    result = x * sin (4 * y) +
             α * y*sin(2*y);
    return result;
}
Arch_Param{ Mu ="constraint";
            ... , Mc ="blending" }
App_Param{  float α = 1.1; ...}
```

Fig. 7. User Inputs of F7

one kernel. iGA4 has the same hardware as pGA4, but is configured as an independent parallel GA with a very large migration gap, so there is not communication between the GA kernels. The differences of resource usages and compilation time are shown in Table IV with various numbers of kernels (N_k), internal evaluation units (N_e) and SCM units (N_s). As shown, when maintaining the same total population, pGA systems can not only have smaller resource usages, due to reduced fan-out and minimised control complexity, but also have shorter compilation time, as it is easier to place-and-route multiple smaller kernels than a few large ones. The change of resource usages is non-linear, as the streaming links also occupy resources.

2) Computational Effort: We record the execution time of the FPGA-based pGA designs and their CPU-based counterparts, i.e. CPU4 is the counterpart of pGA4. As shown in Table V, our systems have an average speedup of 24 times over the multi-core CPU. Moreover, pGA4 system gains 6.8 times speedup over the GPU-based system proposed in [23] on the nVidia Tesla C1060, when comparing the wall clock time taken to find the optimum solution.

We also observe the best fitness found with these configurations, and the times when the solutions are achieved, which are shown in Fig. 5. Overall the pGA systems can achieve better results than the iGA or CPU-based ones, and we can also make the following three observations:

First, the conventional GA1 and independent iGA4 reach plateaus, but the systems with multiple GA kernels and migration reach better solutions. This is because GA1 and iGA4 are vulnerable to local optimal solutions while other configurations avoid similar circumstance by exploring different sub-spaces and by sharing information. The observation demonstrates the importance of accelerating pGAs rather than iGAs for solution quality, and the need to include communication channels between GA kernels.

Second, FPGA-based systems generated by our framework offer high quality solutions in less time than CPU-based systems. This is because the FPGA-based systems can compute many more generations per unit time. This observation suggests the practical usefulness of the systems, as a user always expects satisfactory results within shorter time.

Third, it is not clear which particular configuration achieves the best result. This is a common observation in GA systems.

TABLE V. RESOURCES USAGE (RES.) AND SPEEDUPS OVER A CPU

App.	N_k	N_p	N_e	N_s	Res.%	Speedup	Chromosome
MAXSAT	4	32	2	2	41.44	24	100 x 1-bit
BF6	4	16	2	2	23.74	28	16-bit integer
2DS	4	16	2	2	60.42	34	2 x 8-bit integers
F7	4	16	2	2	27.95	21	2 x floating points
F11	4	16	2	2	55.34	23	2 x floating points
Average	-	-	-	-	-	26	

Practically, this observation suggests the need for flexibility of a GA system, because inflexible systems may prevent the users from fine-tuning them. In contrast, we pay special attention to the flexibility in our framework, allowing users to specify various properties of the system to pursue better solutions.

B. Numerical Optimisation Problems

To test the ability of dealing with numeric computation in our pGA systems, we solve four GA benchmarks with different encodings. Binary F6 (BF6) and 2-D Shubert function (2DS) are from [6], while F7 and F11 are from [21]. All these benchmarks are considered to be complex to solve, but our system can easily find their optimum. For example, Fig. 6 shows the solution search space of F7, which contains many local optimum that may trap the search algorithm. The red cross in the figure shows the best solution. Fig. 7 demonstrates the user inputs for the real-valued function F7; here we choose constraint mutation and blending crossover for this task. An application parameter a is defined in the run-time inputs, allowing users to change it for a different scale without recompilation. The resource usages and speedups are shown in Table V, when the frequencies of the systems are 160 MHz. Based on the resources left over, we can tailor the architectures according to the complexity of an evaluation unit.

VI. CONCLUSION AND FUTURE WORK

As a variant of genetic algorithms, parallel GA has great potential for hardware acceleration. To simplify the design and execution flow of FPGA-based pGA systems for non-expert users, this paper proposes an automated framework to generate them based on a high-level description of a target problem. This general-purpose framework mainly contains customisable GA kernels and a problem-independent library for different types of chromosomes. The flexible architecture created by our framework exploits two levels of parallelism, allowing users to customise the degrees of both fine-grained and coarse-grained parallelism. Once created, the framework also enables a user to tune run-time parameters without time-consuming recompilation, including algorithmic and application parameters. Our system is more flexible compared with existing pGA work; when compared with a multi-core software implementation,

our system also shows an average speedups of 26 times for an NP-hard problem and four benchmarks; our system is also 6.8 times faster than a GPU over the NP-hard problem.

In the future, we will enhance the framework to generate more variants of pGAs such as cellular pGAs, and extend the library to support more genetic operators, for example the mutation using Gaussian random number.

ACKNOWLEDGMENT

The first and third authors are financially supported by the CSC-Imperial Scholarships which are co-founded by Imperial College London and the China Scholarship Council (CSC).

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] D. A. Coley. An Introduction to Genetic Algorithms for Scientists and Engineers. World Scientific. 1999.

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

[23] 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.