Pipelined Genetic Propagation

Liucheng Guo[†], Ce Guo[‡], David B Thomas[†], Wayne Luk[‡]
[†]Department of EEE, [‡]Department of Computing
Imperial College London, United Kingdom
{gl512, ce.guo10, dt10, w.luk}@imperial.ac.uk

***Abstract*—Genetic Algorithms (GAs) are a class of numerical and combinatorial optimisers which are especially useful for solving complex non-linear and non-convex problems. However, the required execution time often limits their application to small-scale or latency-insensitive problems, so techniques to increase the computational efficiency of GAs are needed. FPGA-based acceleration has significant potential for speeding up genetic algorithms, but existing FPGA GAs are limited by the generational approaches inherited from software GAs. Many parts of the generational approach do not map well to hardware, such as the large shared population memory and intrinsic loop-carried dependency. To address this problem, this paper proposes a new hardware-oriented approach to GAs, called Pipelined Genetic Propagation (PGP), which is intrinsically distributed and pipelined. PGP represents a GA solver as a graph of loosely coupled genetic operators, which allows the solution to be scaled to the available resources, and also to dynamically change topology at run-time to explore different solution strategies. Experiments show that pipelined genetic propagation is effective in solving seven different applications. Our PGP design is 5 times faster than a recent FPGA-based GA system, and 90 times faster than a CPU-based GA system.**

I. Introduction

Genetic algorithms (GAs) are a family of nature-inspired search algorithms for general-purpose optimisation. A genetic algorithm finds good solutions to a problem by mimicking the natural evolutionary process, using mutation, crossover, and selection to improve the overall fitness of a pool of candidate solutions. However, natural selection is a slow process, so the GA execution time is significant when the problem is large-scale or has a complicated fitness surface. To allow the use of GAs in a wider range of practical problems, researchers have explored many ways of improving efficiency, including FPGA-based accelerators.

Most existing FPGA-based GAs are adaptations of classic software-based genetic algorithms, which model evolution using consecutive distinct generations. They introduce three significant problems in hardware. First, to compute a new generation of candidate solutions, the algorithm needs to wait for the evolution results of the previous generation, forming a hard data dependency. Second, the generational algorithm requires a large shared memory space to store a pool of candidate solutions, presenting a memory-access bottleneck in FPGAs. Third, when a genetic algorithm in FPGAs reaches a local optimum, it is difficult for users to re-tune hardware architecture because a modification usually results in hours of recompilation.

To address these problems, this paper proposes a novel GA approach called Pipelined Genetic Propagation (PGP), designed specifically for reconfigurable hardware, with the following contributions:

- A hardware-oriented way of structuring GAs as a graph of parallel pipelined components with local candidate storage, breaking the data dependency found in generational genetic algorithms, and eliminating the shared memory bottleneck.
- A topology generator for producing PGP solver instances, allowing multiple candidate graphs to be created for a given FPGA.
- A resource-efficient concrete architecture for PGP, demonstrating high performance compared to existing FPGA solvers, and dynamic changing of solver topology at run-time.

According to the experiments on two classical problems (TSP and MAX-SAT) and five benchmark problems, our pipelined genetic propagation are 5 times faster than a recent hardware-based GA system, 30 times faster than a GPU-based work, and 90 times faster than a multi-core GA system.

II. Background and Related Work

A. Genetic Algorithms

GAs belong to a larger class of evolutionary algorithms, all inspired by natural evolution. The type of evolution in GA is usually an iterative process, with a population of individuals in every iteration called a *generation*. Every individual in a generation has a genetic *chromosome* to represent a candidate solution in the search space for a problem. A generational GA evolves candidate individuals towards the individuals with better qualities as follows [1]. First, the GA creates an initial generation with a population of random individuals. Then an *evaluation* function assesses the quality of all the individuals and assigns them *fitness* values. After that, three genetic operators evolve the current individuals to produce new candidates: 1) *selection* operators choose pairs of individuals as parents based on the fitness values; 2) *crossover* operators combine the chromosomes of the selected parents to produce offspring; 3) *mutation* operators modify the chromosome of the offspring to gain diversity. Finally, after all the offspring are produced, the new generation of offspring replaces the old generation of their parents. The individuals evolve from generation to generation, so GA has a strong data dependence between generations [2]. The evolutionary process stops when a termination condition is satisfied, such as time running out, or finding an acceptably good solution.

978-1-4799-9969-9/15 $31.00 © 2015 IEEE
DOI 10.1109/FCCM.2015.64

B. Genetic Operators

The three types of genetic operators, selection, crossover and mutation, represent natural evolutionary processes, but there are multiple artificial operators which can be chosen for a given problem. Selection variants include roulette-wheel, truncation, and tournament selection, and stochastic universal sampling. Crossover operations include one-point crossover, multi-point crossover, blending crossover (for real-valued chromosomes) [2], and reordering (for permutation chromosomes). Mutations include random bit-flip, constrained mutation (for real-valued chromosomes) [2], and swap mutations (for permutation chromosomes). Different operators may be needed for each problem, though some operators are essentially universal. Supporting multiple operator choices is important, as it allows the solver to be tailored and optimised for the characteristics of a given problem.

C. Existing Hardware-Accelerated GA Approaches

With the increasing demand for GA performance, researchers have tried to use hardware such as FPGAs and GPUs to accelerate the evolution process. For example, a generational GPU-based GA combined with local search for the maximum satisfiability (MAX-SAT) problem has been evaluated [3]. Pedemonte et al. propose a non-generational genetic algorithm called systolic genetic search [4], which replaces the shared generations with small pools distributed on a spatial grid. While GPU-based GAs can be faster than software, there is a large communication and synchronisation cost between processing elements, and existing work is limited in to a restricted set of problems.

FPGAs can provide the parallelism of hardware and the flexibility of software [5], so are a promising platform for acceleration of genetic algorithms. Scott et al. proposed the first FPGA-based GA system in 1995, using a generational approach called HGA [6]. A considerable number of problem-specific FPGA-based generational GAs were then proposed [7], [8], [9], [10], [11], [12]. More flexible problem-indendent approaches have also been considered [10], showing speed-ups of about 5 times over multi-core CPU across several GA benchmarks. One reason for the low speedups is due to the communication overhead and pipeline stalls between generations.

Variants of classic generational GAs have also been considered for FPGAs, such as Parallel GAs (PGAs) and pipeline-effective GAs. PGAs use multiple GA instances with scheduled communication and exchange between the instances, which enables a large amount of parallelism and the use of local memories [13], [14]. A recent example is a PGA supporting a wide class of problems using a framework for automatically creating solvers [15]. Their work explores multi-level parallelism and supports modification of some GA parameters at run-time, but the pipelines still have stalls due to communication and synchronisation between GA instances. To reduce the pipeline stalls, [16] and [17] propose pipeline-effective GAs in FPGAs, but these still need large globally shared memories to store the population.

Most FPGA-based GA systems share one or more common problems in the algorithm mapping and tuning: 1) lack of run-time tuning when the solver gets stuck in the local optimum; 2) the necessity for a large single piece of memory to store all the individuals; 3) the pipeline stalls to wait all the new offspring to be produced before entering the next generation. These issues limit the performance of FPGA-based GA systems, so to address them this paper proposes a novel non-generational genetic algorithm on FPGAs called PGP.

III. Pipelined Genetic Propagation

Pipelined Genetic Propagation is an optimisation algorithm, architecture, and strategy, allowing us to create a hardware-specific graph-based GA optimiser for a chosen objective function. A PGP system optimises the objective function by propagating and circulating a group of individuals in a directed graph structure $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ [18], where $\mathbb{V}$ is a set of genetic nodes that perform genetic operations on data, and $\mathbb{E}$ is a set of directed edges between the nodes that transport data in a pipelined or streaming manner. All nodes and edges are continually active, with no pauses or dependencies, allowing good utilisation of the logic.

A. Genetic Nodes

Genetic nodes are the elementary compute components of a PGP solver, and are represented by the nodes $\mathbb{V}$ in the logical structure. A primary concern in the design of the nodes is an efficient hardware implementation, so we propose the three requirements below.

1) Minimise memory consumption. On-chip RAM is limited in a reconfigurable hardware platform, so we try to eliminate explicit RAM storage and rely on implicit storage in pipelines, only using on-chip memory when random accesses are needed.
2) Distribute local memories. The large shared storage in generational GAs results in high contention and significant fan-out. We only allow local RAMs which are accessible within a genetic operator, ensuring locality and exploiting the distributed nature of FPGA storage.
3) Minimise per-node resource consumption. To get the maximum speed-up, we want as many genetic operators as possible, so we design small operators that map easily to hardware, allowing us to scale up to complex solver topologies and large FPGAs.

Based on the above requirements, we map all the data-dependent genetic operators of genetic algorithms into hardware as distributed nodes, which are shown in Fig. 1. We divide the genetic nodes into three classes, namely selection nodes, crossover nodes and mutation nodes. The exact behaviour and implementation of these nodes (e.g. the specific version of selection) is not specified at this level, only the overall behaviour in terms of inputs and outputs.

A selection node consists of an input port, an output port, some amount of local memory, a fitness evaluator, and an unspecified selection process. The local memory stores a fixed number N_i of individuals along with their corresponding fitness values. Every cycle, the input port receives one incoming individual I_{in} and sends it to the evaluator, while the output port sends out one individual I_{out} chosen by the selector from the local internal memory. When the fitness value of the

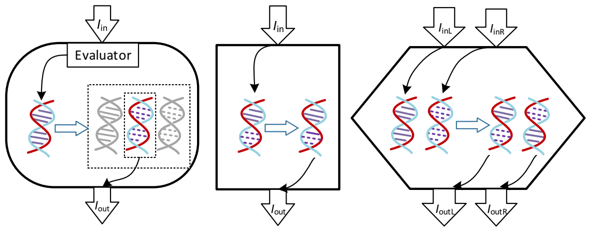


Fig. 1: Genetic Nodes: Selection, Mutation and Crossover

incoming individual I_{in} becomes available, the node compares the fitness values of I_{in} and I_{out}. If the incoming individual I_{in} has higher quality than the outgoing individual I_{out}, it replaces I_{out} in the local memory. Meanwhile, its fitness value replaces that of the selected individual as well. In other words, the selection node monitors the quality of the individuals evolving and stores the high-quality individuals in the local memory.

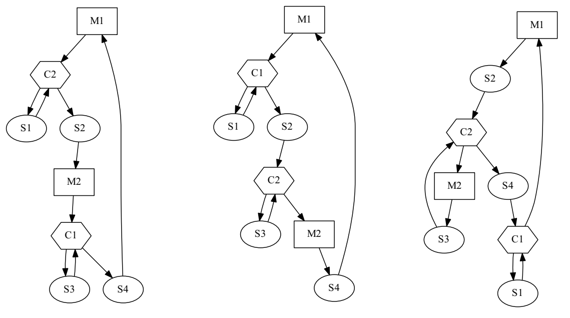
A mutation node contains an input port, an output port and some sort of mutation process, but does not contain any local memory or fitness evaluator. Every cycle, the input port receives one incoming individual I_{in}, then mutates the individual (for example using a bit-flip or constraint mutation). Finally, the output port sends out a mutated individual to other nodes.

A crossover node consists of two inputs, two output ports and some sort of crossover process. Similar to the mutation node, a crossover node contains neither local memory nor fitness evaluator. The crossover node exchanges the information between the incoming individual I_{inL} from one input port and I_{inR} from the other port. After receiving the two individuals, the crossover executor cuts off a piece of information from each individual, and recombines the individuals by transplanting the information taken from one individual to the other, producing two new individuals I_{outL} and I_{outR}. The two output ports then send out I_{outL} and I_{outR} to other nodes. Note that we disallow any output of a crossover node to directly connect to its inputs, because a crossover node configured in this way might introduce a combinatorial loop.

We divide the genetic nodes into two categories, namely non-blind nodes and blind nodes. As the selection nodes have the evaluators to monitor the quality of individuals, we call them *non-blind* nodes; the mutation and crossover nodes change the individuals without considering their quality, so we call them *blind* nodes. For ease of discussion, we denote the set of all selection, mutation and crossover nodes respectively as $\mathfrak{S}, \mathfrak{M}$, and $\mathfrak{C}$. Let the set of blind nodes $\mathfrak{B} = \mathfrak{M} \cup \mathfrak{C}$ and the set of all non-blind nodes as $\mathfrak{N} = \mathfrak{S}$. A evolution process should have at least one blind node and one non-blind node, because blind nodes carry out the evolutionary genetic operations which increases diversity, while the non-blind nodes ensure that there is a trend towards higher fitness individuals.

B. Logical Connection Between Nodes

We consider the logical connections between the nodes, which are captured as the set of edges $\mathbb{E}$ between nodes. In the design of logical connections, we keep the following considerations in mind.



(a) Topology 1 (b) Topology 2 (c) Topology 3

Fig. 2: Three Topologies

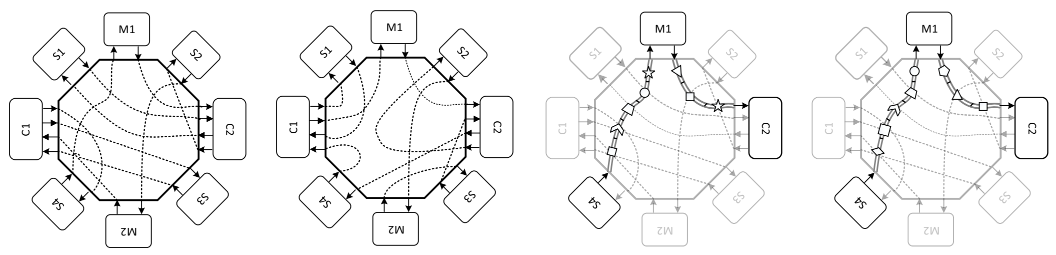
1) Minimise the amount of recompilation. Hardware compilation of FPGA devices is time-consuming, and it is not always clear what the best topology is, so we support a single compilation of the architecture which allows multiple possible topologies.
2) Dynamic changes of connection topology at run-time. Run-time reconfiguration of the topological structure can enhance the optimisation power of the system, but we do not want to interrupt the current evolutionary process. During the reconfiguration of the topological structure, we keep the data in all unrelated blocks unchanged, retaining the fitness already achieved.
3) Configurable set of selection, mutation, and crossover nodes. Users can specify a set of nodes to use, and may also choose to use a sub-set in certain configurations. PGP will then create valid topologies using all the user-specified nodes to allow exploration of different patterns of connectivity between them.

Compared with the fixed and restricted connection of genetic operators in generational genetic algorithms, the proposed connection scheme is much more flexible and powerful. Given the number and type of genetic nodes, we can create many different topologies. For example, given 2 mutation nodes, 4 selection nodes and 2 crossover nodes, we have 10 inputs and 10 outputs in total, a large set of possible interconnections.

However, only a subset of topologies are reasonable, and can be expected to produce good results. We restrict the set of topologies using the following rules:

- **Rule of full connectivity**: for all pairs of nodes $(v_\dagger, v_\ddagger) \in \mathbb{V}^2$ in a topology, there exists a path from $v_\dagger$ to $v_\ddagger$ with finite length.
- **Rule of striped serialisation**: for all edges $\langle v, v' \rangle \in \mathbb{E}$, if $(v, v') \in \mathfrak{B}^2$, then $v'' \in \mathfrak{N}$ for all v'' such that $\langle v', v'' \rangle \in \mathbb{E}$; for all edges $\langle v, v' \rangle \in \mathbb{E}$, if $(v, v') \in \mathfrak{N}^2$, then $v'' \in \mathfrak{B}$ for all v'' such that $\langle v', v'' \rangle \in \mathbb{E}$.
- **Rule of fan-out control**: $\deg^+(v_i) = 1$ for all nodes $v_i \in \mathfrak{M} \cup \mathfrak{S}$; $\deg^+(v_{ii}) = 2$ for all nodes $v_{ii} \in \mathfrak{C}$ where $\deg^+(v)$ is the outdegree of node v.

The rule of full connectivity ensures all individuals are able (though not guaranteed) to eventually reach all nodes in the topology. The rule of striped serialisation ensures at most two



(a) Mapping of Two Topologies

(b) Pipelined Flow

Fig. 3: Physical-Level Connections between Genetic Nodes

nodes of the same type connect directly. In particular, a chain of blind nodes may throw out or replace high-quality solutions because the blind nodes do not monitor the fitness. Similarly, a chain of non-blind nodes keep the individuals unchanged and result in additional resource consumption. The rule of fan-out control avoids duplication of individuals to keep the divergence of the population.

It is a non-trivial task to generate legitimate topologies satisfying all the proposed rules, particularly with large numbers of nodes. We tackle the topology generation problem and present our solution in section IV. In the rest of this section, we assume the availability of legitimate topologies. For instance, Fig. 2 shows three topologies satisfying all the three rules. In each topology, $\mathbb{V}$ includes two mutation nodes (M1 and M2), two crossover nodes (C1 and C2), and four selection nodes (S1, S2, S3 and S4).

C. Physical Connection Between Nodes

The pipelined genetic propagation scheme has a substantial advantage over generational genetic computing architectures in terms of pipeline efficiency. In a generational system, the next generation cannot be started till the previous one has finished, forcing a synchronisation point amongst all genetic operators. This data dependence results in pipeline stalls and reduces the overall efficiency of computation.

In contrast, our system does not divide the evolution process into generations. Therefore, all genetic operations are able to work simultaneously in a fully pipelined manner without incurring stalls between generations. For instance, the physical connections between the genetic nodes of the two example topologies in Fig. 2 are shown in Fig. 3a. The two stages of the pipelined flow of the first topology are demonstrated in Fig. 3b, showing the individuals flowing in the pipeline as streams.

To allow the propagation and circulation of individuals in the topology, we need to map the logical structure to a physical reconfigurable device, which we solve using a *topology mapping unit* to manage the connection of genetic nodes. The mapping unit connects all the inputs and outputs of genetic nodes, using multiplexers to support dynamic reconfiguration of topology without interrupting executing. A direct and simple solution is to connect all the inputs from each genetic nodes to all genetic node outputs, using a *full crossbar*.

Full crossbar supports a wide range of topologies, however it is resource-consuming, especially for a large number of genetic nodes. It also supports many illegal connections of nodes, as well as multiple realisations of what is essentially the same topology. To reduce resource consumption, we limit the inputs of each multiplexer to support only the topologies needed. If a user generates M topologies for a structure, we only need at most M input sources for each multiplexer. Because some of the topologies share a subset of inputs, the number of required inputs for a multiplexer may be even less than M, resulting in a *sparse crossbar*. We will discuss the empirical benefits of sparse crossbars over full crossbars in section V.

D. Dynamic Change of Topology

Generational genetic algorithm sometimes gets stuck into a local optimum, or a particular configuration is found to work poorly for a specific problem. In software the algorithm and parameters can be tweaked, but in hardware any modification of the architecture requires a time-consuming recompilation. In PGP, we can switch between multiple topologies, in order to escape from the current local optimum found by one topology. For instance, Fig. 4 shows a case where we support three topologies for a problem, each of which contains two genetic operators. In this example, we assume that the two genetic operators are conventional and smooth, such that one execution of the operator slightly modifies the individual. A point in the space represents an individual, and the x axis and y axis show respectively the directions that the two genetic operators drive individuals along. The z axis stands for the fitness value of the individual. All points in the space form a surface corresponding to the three-dimensional projection of the high-dimensional individual fitness mapping. The surfaces in the three topologies are different as genetic operations in the three topologies move individuals on different planes in the high-dimensional space.

The genetic operations modify an individual slightly in each execution, resulting in small changes in the solution space. The small scale of changes brings both benefits and drawbacks. The small steps enable an individual to move smoothly in the solution space of a topology without mistakenly skipping high-quality solutions. Nevertheless, when an individual gets stuck in a local optimum, the genetic operators may have insufficient power to help the individual to escape due to the limited step size.

An occurrence of dynamic change of the topology corresponds to an *inter-topology crossing* for all individuals. If a individual is unable to move in the original topology due to a local optimum, it may move again in the destination topology.

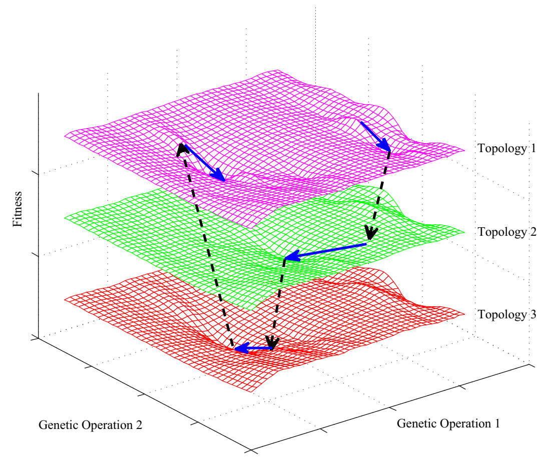


Fig. 4: Inter-Topology Crossing

By moving back and forth between topologies, an individual has a better chance to escape from a local optimum.

IV. Topology Generation Algorithm

We propose a randomised topology generation algorithm to produce legitimate topological structures for the connection problem based on the three rules in section III.B. This algorithm consists of four major parts namely *balancing*, *threading*, *dispatch* and *collapsing*.

A. Balancing

The balancing algorithm discovers legitimate configuration of a chosen number of blind and non-blind nodes by finding integer solutions to a set of equations. According to the Rule of striped serialisation, let x_b and x_{2b} be the number of blind nodes and blind pairs respectively; let x_{nb} and x_{2nb} be the number of non-blind nodes and non-blind pairs respectively. By definition, the following must hold:

$$x_b + 2x_{2b} = 2c + m \tag{1}$$
$$x_{nb} + 2x_{2nb} = s \tag{2}$$

Here c is the number of crossover nodes, m is the number of mutation nodes, and s is the number of selection nodes.

Because blind components and non-blind component alternate in the chain, the number of blind components equals that of the non-blind ones, such that:

$$x_b + x_{2b} = x_{nb} + x_{2nb} \tag{3}$$

Therefore, the vector $\mathbf{x} = (x_b, x_{2b}, x_{nb}, x_{2nb})'$ must satisfy

$$\begin{pmatrix} 1 & 2 & 0 & 0 \\ 0 & 0 & 1 & 2 \\ 1 & 1 & -1 & -1 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 2c+m \\ s \\ 0 \end{pmatrix} \tag{4}$$

The rank of the coefficient matrix is less than the number of its rows. Therefore, the linear equation has an infinite number of solutions if there are no additional constraints. However, we know that x_b, x_{2b}, x_{nb} and x_{2nb} must be all positive integers. To obtain the positive solutions, we first represent x_b, x_{nb} and x_{2nb} in terms of x_{2b}:

$$x_b = 2c + m - 2x_{2b} \tag{5}$$
$$x_{nb} = 4c + 2m - 2x_{2b} - s \tag{6}$$
$$x_{2nb} = s - 2c - m + x_{2b} \tag{7}$$

We then determine x_{2b} by enumerating through a number of integers in the range $[x_\perp, x_\top]$ where

$$x_\perp = \lceil 2c + m - s \rceil \tag{8}$$
$$x_\top = \left\lfloor \min\left(\frac{m}{2} + c, m, m + 2c - \frac{s}{2}\right) \right\rfloor \tag{9}$$

Equation 8 and 9 remain unchanged regardless the value of m, c and s. Therefore, no integer linear programming is required to solve the equations.

B. Threading

The threading algorithm arranges the units along a directed ring. The threading algorithm first makes a local arrangement for blind components and non-blind ones separately, and then merges the two sequences together. To make a local arrangement for blind components, the algorithm creates a sequence of blind components where the number of blind units and pairs follow the constraints from the previous stage. To merge the two local arrangements together, the algorithm takes entries alternately from the two local arrangements.

The algorithm first produces a sequence list of blind nodes L^B and a sequence of non-blind ones L^N. Let $\mathcal{B}$ be an abstract blind node, which will become a crossover node or a mutation node in later stages of the generation algorithm; let $\mathcal{N}$ be a non-blind node, which in this work is always a selection node. For ease of explanation, we use $2\mathcal{B}$ and $2\mathcal{N}$ to represent respectively a combination of two blind nodes, and a combination of two non-blind nodes.

$$L^B = (\underbrace{\mathcal{B}, \mathcal{B}, \ldots, \mathcal{B}}_{x_b \text{ times}}, \underbrace{2\mathcal{B}, 2\mathcal{B}, \ldots, 2\mathcal{B}}_{x_{2b} \text{ times}}) \tag{10}$$
$$L^N = (\underbrace{\mathcal{N}, \mathcal{N}, \ldots, \mathcal{N}}_{x_{nb} \text{ times}}, \underbrace{2\mathcal{N}, 2\mathcal{N}, \ldots, 2\mathcal{N}}_{x_{2nb} \text{ times}}) \tag{11}$$

The algorithm then shuffles L^B and L^N into a random order called Λ^B and Λ^N.

Finally, the algorithm returns a sequence Λ containing all the entries in Λ^B and Λ^N. Entries in Λ with odd indexes are entries in Λ^B in their original order, i.e.

$$\Lambda_{2k+1} = \Lambda^B_{k+1} \tag{12}$$

for all $k \in \mathbb{Z} \cap [0, x_b + x_{2b} - 1]$. Similarly, entries in Λ with even indexes are entries in Λ^N in their original order, i.e.

$$\Lambda_{2k} = \Lambda^N_k \tag{13}$$

for all $k \in \mathbb{Z} \cap [1, x_{nb} + x_{2nb}]$. By linking up the head and tail of Λ, we may build a ring satisfying all the rules discussed in section III.B.

C. Dispatching

The dispatch algorithm expands the sequence produced by the threading algorithm by dispatching mutation (M), half-crossover (hC), and selection (S) nodes. A half-crossover node is a temporary unit to simplify the computation. It contains one input and one output. In a later stage, two half-crossover nodes merge to a crossover node.

The algorithm first instantiates NB and 2NB units, as they must be selection nodes. Let $\mathcal{E}$ be the result of dispatching.

$$\mathcal{E}_{i_N} = (S),\ \forall i_N \in \{i : \Lambda_i = \mathcal{N}\} \tag{14}$$

$$\mathcal{E}_{i_{2N}} = (S, S),\ \forall i_{2N} \in \{i : \Lambda_i = 2\mathcal{N}\} \tag{15}$$

The next task is to dispatch M nodes. As we have discussed in section III.A, we avoid the situation where an output of a crossover node connects to its own inputs. As a result, we avoid two hC nodes from occupying a 2B unit. To achieve this, the algorithm dispatches at least one M node to each 2B unit. Therefore, each 2B unit either contains exactly one or two M nodes. In other words, the algorithm fixes a total of x_{2b} M nodes to 2B units, leaving $(m - x_{2b})$ to dispatch elsewhere.

It is trivial to assign an M node to each 2B unit, so we fix these M nodes while dispatching other nodes. To dispatch the $(m - x_{2b})$ M nodes in the ring, we first find all possible dispatching locations by computing the index set $\mathcal{I}_{B+}$ of B and 2B nodes:

$$\mathcal{I}_{B+} = \{i : \Lambda_i = \mathcal{B} \vee \Lambda_i = 2\mathcal{B}\} \tag{16}$$

We randomly select a subset with $(m - x_{2b})$ members, i.e.

$$\mathcal{I}^*_{B+} = \text{randomsubset}(\mathcal{I}_{B+}, m - x_{2b}) \tag{17}$$

Then for each $i^*_{B+} \in \mathcal{I}^*_{B+}$, we dispatch an M node using

$$\mathcal{E}_{i^*_{B+}} = \begin{cases} (M), & \Lambda_{i^*_{B+}} = \mathcal{B} \\ (M, M), & \Lambda_{i^*_{B+}} = 2\mathcal{B} \end{cases} \tag{18}$$

Next, we fix the hC nodes to the remaining locations in $\mathcal{I}_{B+}$. Specifically, for all $i^\#_{B+} \in (\mathcal{I}_{B+} - \mathcal{I}^*_{B+})$

$$\mathcal{E}_{i^\#_{B+}} = \begin{cases} (hC), & \Lambda_{i^\#_{B+}} = \mathcal{B} \\ (hC, M), & \Lambda_{i^\#_{B+}} = 2\mathcal{B} \end{cases} \tag{19}$$

To introduce extra diversity to the topology, when $\Lambda_{i^\#_{B+}} = 2\mathcal{B}$, we randomly flip (hC, M) to (hC, M) with a probability of 0.5. The dispatch algorithm finally returns a list of nodes Ξ by linking up all list in $\mathcal{E}$, i.e.

$$\Xi = \mathcal{E}_1 \parallel \mathcal{E}_2 \parallel \ \dots \ \parallel \mathcal{E}_{|2(x_b + x_{2b})|} \tag{20}$$

where $\parallel$ is the list concatenating operator.

D. Collapsing

The collapsing algorithm transforms the outcome of the dispatch algorithm by randomly pairing half-crossover (hC) nodes. To achieve this goal, the algorithm first extracts locations of hC nodes in Ξ, i.e.

$$\mathcal{I}_{hC} = \{i : \Xi_i = hC\} \tag{21}$$

Then it partitions $\mathcal{I}_{hC}$ into two subsets with equal size c. Next, the algorithm places the members in each subset into a sequence in random order, called C^L and C^R. Finally, we obtain the C nodes by combining the hC nodes in the two sequence with identical index, i.e.

$$C_i = (C^L_i, C^R_i) \tag{22}$$

The topology generator finally returns the ring Ξ along with C. Ξ contains the logical positions of all M, S and hC nodes, while C specifies the combination of hC nodes.

The user may then deploy the result of the topology generator in a reconfigurable device using multiplexers following the instructions in section III.C. Experimental results related to the generator are available in section V.

V. Experiments

To evaluate the PGP approach, we apply it to two optimisation problems, the maximum satisfiability (MAX-SAT) and travelling salesman problems (TSP), and five numerical optimisation benchmarks [10], [19]. These seven problems have different kinds of chromosomes, including fixed-point, permutation and floating-point chromosomes, and vary significantly in their scale and difficulty.

We compare the proposed work to a recent FPGA-based accelerator [15], a GPU-based accelerator [3], and a CPU-based software based on the generational genetic algorithm [19]. The CPU-based system runs on an Intel Xeon X5650 CPU using 8 of 12 cores because more cores reduce the performance. The GPU version targets the nVidia Tesla C1060 platform. The FPGA-based accelerator targets a Virtex 6 (SX475T) FPGA. Our proposed system also targets the same platform for fair comparison. A reference software model and experimental setting are available online.[1] The CPU-based PGP is slow due to communication and synchronisation cost, so here we only compare our work with the standard CPU-based GA system.

If the best individual fitness of a GA system remains unchanged for 10,000 cycles, we consider the GA system to have reached its plateau fitness. When a CPU-based GA system reaches its plateau, we calculate the speedup by dividing CPU-based GA system time by the time for the FPGA-based systems to reach the same fitness.

A. Maximum Satisfiability Problem (MAX-SAT)

The maximum satisfiability problem (MAX-SAT) is a classic NP-hard combinational optimisation problem [3]. We apply three different topologies (PGP-1, PGP-2 and PGP-3) in our proposed work, and denote the inter-topology crossing technology between these three topologies as PGP-X. When our work reaches a plateau with unchanged fitness for 10,000 clock cycles, we carry out the inter-topology crossing. Inter-topology crossing never reduces the highest quality found using previous topology, because the optimum is still in local memory when switching topologies.

1) Performance Results: As GA is a stochastic algorithm, we run the experiments multiple times. The average values of the best fitness found over time are shown in Fig. 5. Our proposed work clearly outperforms the FPGA-based GA system (F-GA) and CPU-based system (C-GA), and when the

[1]http://guoliucheng.info/pgp

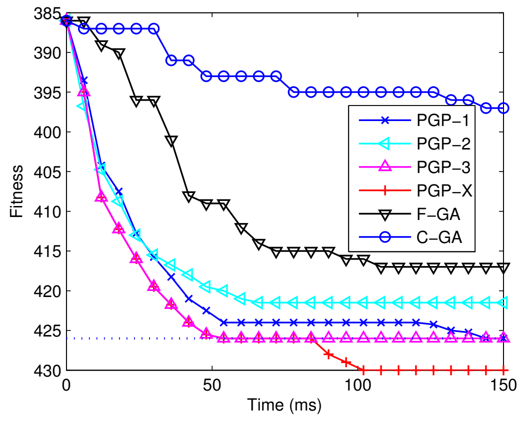


Fig. 5: Fitness Found by Different Optimisers

C-GA system reaches plateau fitness (shown as the dotted line in the figure), our proposed work has a speedup of 90 times over the multi-core C-GA, and 5 times over the F-GA for the same fitness. For the same problem, our work is also 30 times faster than the GPU-based GA [3].

We can make the following three observation from Fig. 5.

1) All the three topologies provide significantly better results than the CPU and generational FPGA architecture in terms of convergence speed and solution fitness. This observation shows the increased optimisation power of the PGP-X approach.
2) The gaps in terms of the optimisation power between three different topologies are small. This observation demonstrates the stable quality of topological structures produced by the proposed topology generator with the three rules.
3) The results for PGP-X with inter-topology crossing are better than all fixed topologies. This observation shows the usefulness of dynamic switching between topologies.

2) Resource Optimisation: We also test the reduction of resource consumption using sparse crossbars. We build three systems consisting of 5, 10 and 20 genetic nodes respectively. In each system, we generate five different topologies using the generator proposed in section IV. The resource usages are shown in Table I, and in all cases the sparse crossbars save resources, sometimes significantly.

B. Travelling Salesman Problem and Benchmarks

We apply our proposed work to the travelling salesman problem (TSP) with 64 cities, and five different functional benchmarks, including binary function 6 (BF6) [10], binary function 7 (BF7) [10], function 11 (F11) [19], 2-D Shubert function (2DS) [10] and Rosenbrock function (RF). After running multiple times, we compare the average of the best fitness versus time for the following three approaches: the FPGA-based generational GA system (F-GA), pipelined genetic propagation using inter-topology crossing (PGP-X), and the CPU-based standard GA (C-GA). Table II and Fig. 6 show the combined results and PGP-X information, including clock

TABLE I: Resource Usages of MAX-SAT

# of Nodes	Crossbar	LUTs(%)	FFs(%)	BRAMs(%)
5	Full	6.17	5.11	3.85
5	Sparse	5.98	5.01	3.85
5	Saving	3.08%	1.96%	0.00%
10	Full	12.90	9.54	6.95
10	Sparse	10.31	8.49	6.20
10	Saving	20.08%	11.01%	10.79%
20	Full	26.38	17.86	13.44
20	Sparse	21.50	16.96	13.35
20	Saving	18.50%	5.04%	0.67%

TABLE II: Experimental Results (SP.F: Speedup over F-GA, SP.C: Speedup over C-GA, Res.: Resources, Freq.: Frequency)

Fun.	# of M	# of C	# of S	Freq. (MHz)	Res. (%)	SP.F	SP.C
TSP	2	1	2	110	77.17	-	91
BF6	3	3	6	160	20.18	6	98
BF7	3	3	6	160	21.22	5.8	91
F11	3	3	6	160	46.24	3.8	83
2DS	3	3	6	160	38.73	3.1	103
RF	3	3	6	160	23.16	3.7	76
Avg	-	-	-	-	-	4.5	90

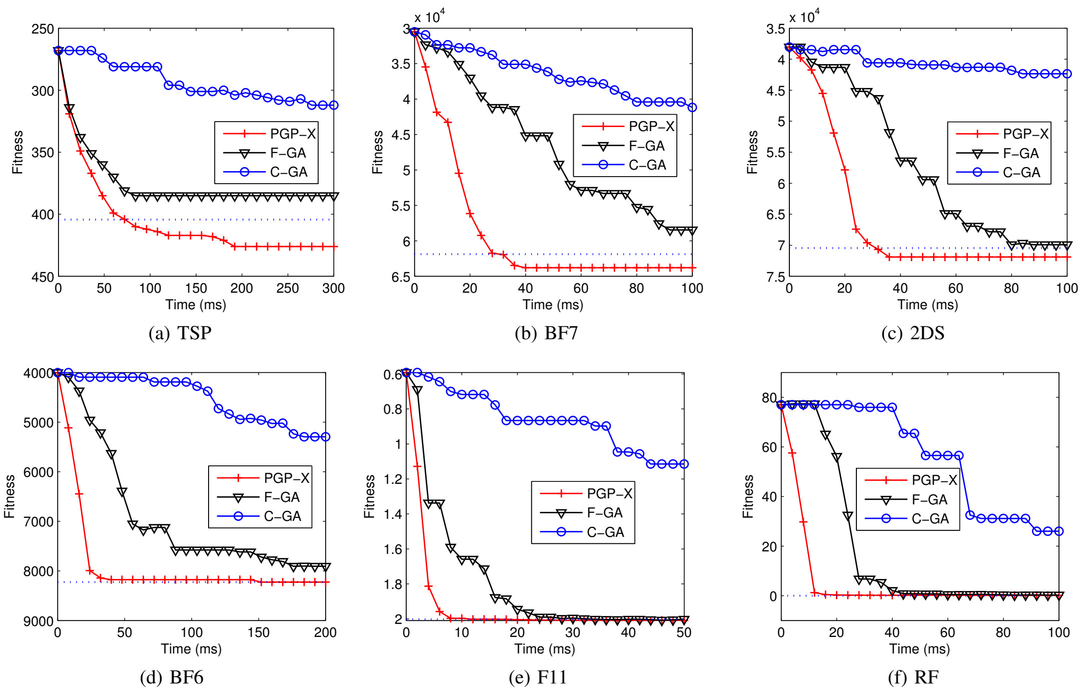
frequency, resources and the numbers of genetic nodes (S, C, M). After the CPU-based work reaches plateau labelled as dotted line, our proposed work has an average speedup of around 5 times than the generational FPGA-based GA system, and an average speedup of 90 times over the multi-core CPU-based GA. The blank in the table means F-GA cannot reach the fitness plateau of C-GA.

We can make the following observations from Fig. 6. 1) The PGP approach provides better results versus time spent than the other systems for different types of objective functions, across a wide variety of problems. 2) The PGP architecture is less likely to get stuck in a local plateau. 3) The fitness of the pipelined genetic propagation architecture is smoother than those of the generational GA architecture.

To sum up, the pipelined genetic propagation outperforms the other two systems in terms of optimisation power and computational efficiency. We offer three possible explanations for this improvement: 1) the inter-topology crossing may help escape from a local optimum, although there is no guarantee it will fall into the global optimum; 2) the non-trivial topological structure may encourage higher genetic diversity in the population; 3) the distributed memory and reduced fan-out enable our system to achieve a higher clock frequency than the generational FPGA-based GA. For instance, the maximal frequency of generational FPGA-based GA system for TSP is 75MHz, while that of our proposed work is 110MHz.

VI. Conclusion and Future Work

This paper proposes a new approach for genetic algorithms called pipelined genetic propagation, which is optimised for reconfigurable hardware. It has many loosely coupled genetic nodes with distributed local memory, and supports dynamic switch of topologies to escape from a local optimum, along with a generator to produce the topologies automatically. Experiments on two applications and five GA benchmarks demonstrate the high performance and flexibility of our proposed work.



(a) TSP (b) BF7 (c) 2DS

(d) BF6 (e) F11 (f) RF

Fig. 6: Fitness Results (dotted line: plateau fitness of C-GA)

In the future, we will develop a new topology generation algorithm to support user-defined rules. Another direction is to devise strategies to perform inter-topology crossing in a non-random manner using reinforcement learning [20] to improve its effectiveness.

ACKNOWLEDGMENT

The first and second authors are financially supported by the CSC-Imperial Scholarships 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] R. L. Haupt, and S. E. Haupt. Practical genetic algorithms. John Wiley & Sons, 2004.

[3] 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.

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

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

[6] S. Scott, et al. "HGA: A hardware-based genetic algorithm." ACM International Symposium on FPGA. pp. 53-59, 1995.

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

[8] L. Guo, D. B. Thomas, and W. Luk. "Automated Framework for General-Purpose Genetic Algorithms in FPGAs." Applications of Evolutionary Computation. Springer Berlin Heidelberg. pp. 714-725, 2014.

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

[10] 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.

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

[12] 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.

[13] 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.

[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] 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. pp. 1-7, 2014.

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

[17] O. Kitaura, et al. "A custom computing machine for genetic algorithms without pipeline stalls." IEEE International Conference on Systems, Man, and Cybernetics. Vol. 5, pp. 577-584, 1999.

[18] S. Dasgupta, C. H. Papadimitriou, and U. Vazirani. Algorithms. McGraw-Hill, Inc. 2006.

[19] D. A. Coley. An introduction to genetic algorithms for scientists and engineers. World Scientific Publishing, 2003.

[20] R. S. Sutton, et al. "Fast gradient-descent methods for temporal-difference learning with linear function approximation." International Conference on Machine Learning. pp. 993-1000, 2009.