Graphics Processing Unit–Enhanced Genetic Algorithms for Solving the Temporal Dynamics of Gene Regulatory Networks

Evolutionary Bioinformatics
Volume 14: 1–16
© The Author(s) 2018
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1176934318767889

Raúl García-Calvo[1], JL Guisado[1], Fernando Diaz-del-Rio[1], Antonio Córdoba[2] and Francisco Jiménez-Morales[2]

[1]Department of Computer Architecture and Technology, University of Seville, Seville, Spain.
[2]Department of Condensed Matter Physics, University of Seville, Seville, Spain.

ABSTRACT: Understanding the regulation of gene expression is one of the key problems in current biology. A promising method for that purpose is the determination of the temporal dynamics between known initial and ending network states, by using simple acting rules. The huge amount of rule combinations and the nonlinear inherent nature of the problem make genetic algorithms an excellent candidate for finding optimal solutions. As this is a computationally intensive problem that needs long runtimes in conventional architectures for realistic network sizes, it is fundamental to accelerate this task. In this article, we study how to develop efficient parallel implementations of this method for the fine-grained parallel architecture of graphics processing units (GPUs) using the compute unified device architecture (CUDA) platform. An exhaustive and methodical study of various parallel genetic algorithm schemes—master-slave, island, cellular, and hybrid models, and various individual selection methods (roulette, elitist)—is carried out for this problem. Several procedures that optimize the use of the GPU's resources are presented. We conclude that the implementation that produces better results (both from the performance and the genetic algorithm fitness perspectives) is simulating a few thousands of individuals grouped in a few islands using elitist selection. This model comprises 2 mighty factors for discovering the best solutions: finding good individuals in a short number of generations, and introducing genetic diversity via a relatively frequent and numerous migration. As a result, we have even found the optimal solution for the analyzed gene regulatory network (GRN). In addition, a comparative study of the performance obtained by the different parallel implementations on GPU versus a sequential application on CPU is carried out. In our tests, a multifold speedup was obtained for our optimized parallel implementation of the method on medium class GPU over an equivalent sequential single-core implementation running on a recent Intel i7 CPU. This work can provide useful guidance to researchers in biology, medicine, or bioinformatics in how to take advantage of the parallelization on massively parallel devices and GPUs to apply novel metaheuristic algorithms powered by nature for real-world applications (like the method to solve the temporal dynamics of GRNs).

KEYWORDS: Gene regulatory networks, evolutionary computing, parallel genetic algorithms, GPU

RECEIVED: November 23, 2017. **ACCEPTED:** February 28, 2018.

TYPE: Original Research

FUNDING: The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the following research projects of the Ministerio de Economía, Industria y Competitividad (MINECO), and the "Agencia Estatal de Investigación (AEI)" of Spain, co-financed by FEDER funds (EU): MABICAP (Bio-inspired machines on High Performance Computing platforms:a multidisciplinary approach, TIN2017-89842P), COFNET (Event-based Cognitive Visual and Auditory Sensory Fusion, TEC2016-77785-P) and TOP4COG (Topological Recognition of 4D Digital Images via HSF model, MTM2016-81030-P (AEI/FEDER,UE)).

DECLARATION OF CONFLICTING INTERESTS: The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

CORRESPONDING AUTHOR: José Luis Guisado, Department of Computer Architecture and Technology, University of Seville, E.T.S. Ingeniería Informática, Avda. Reina Mercedes s/n, 41012 Sevilla, Spain. Email: jlguisado@us.es

Introduction

The understanding of gene regulatory networks (GRNs)[1,2] is one of the main current challenges in biology. A crucial problem is the determination of laws that govern the temporal dynamics of a biologic system from a particular state to a later one, through activating and repressing actions of some genes or their products (proteins) over other ones in transcription and translation processes. The identification of these actions is very important for developmental biology, for determination of the causes of certain diseases, and for the signaling of therapeutic targets. The set of genes and their interactions, which determine a particular behavior, can be represented by a directed network in which the nodes are genes/proteins and the links represent the activating and repressing actions of some genes over other ones. Based on experimental data from 2 different network states, the dynamical processes that transform from one state into another through evolution rules can be inferred and their characteristic parameters can be evaluated.[3,4] As experimental data are often unclear or incomplete, this information can also help experimentalists in the search for new interactions or in the discard of false positives.

One of the most used inference method consists in using Boolean rules, characterizing node states by 2 binary values (activated/not activated) that can be marked as 1 and 0, and links associated to actions by activator/repressor do not exist (which can be marked as +1, −1, and 0). A set of possible simple rules can be established, which determines the state of a node at any given time by the nature of the incoming links to that node at a previous time. The state of the network nodes can be refreshed synchronously or asynchronously. Given an initial and a final state, the objective is to determine which rules and interactions are applicable to each node, with a particular configuration of links, to reach the final state (or a state very close to it) in a number of time steps, starting from the initial state. In other words, we would like to find the network links and the evolution rules of the network nodes. In the case that the result is not unique, the results can be interpreted with biologic criteria or by experiments that confirm one option against the other

Creative Commons Non Commercial CC BY-NC: This article is distributed under the terms of the Creative Commons Attribution-NonCommercial 4.0 License (http://www.creativecommons.org/licenses/by-nc/4.0/) which permits non-commercial use, reproduction and distribution of the work without further permission provided the original work is attributed as specified on the SAGE and Open Access pages (https://us.sagepub.com/en-us/nam/open-access-at-sage).

ones. To find the structure of the network and its evolution rules, a heuristic optimization method is applied. A fitness function is calculated by comparing the reached state with the preestablished final state. The most frequently applied methods are evolutionary algorithms,[5,6] a family of optimization methods that are inspired by the principles of natural evolution to find solutions to hard optimization problems. The method introduced in Aguilar-Hidalgo et al[3] can infer the dynamics of GRNs between known initial and ending network states by simple acting rules, using a genetic algorithm (GA). The method was also reviewed and its applicability to real biologic problems was further discussed in Aguilar-Hidalgo et al.[4] However, in those works, only a simply sequential version of the algorithm was used, thus needing several hours for a small testing GRN. Even for our optimized sequential implementation using a compiled language like C++ with all the optimizations enabled (used for comparison with a parallel version in this article), a runtime of the order of 30 min is needed, for example, to run a single-threaded version of the algorithm for a GRN including 32 genes with a GA population of 30 000 individuals on a high-end Intel i7 CPU. Moreover, this runtime increases rapidly with the number of genes of the GRN.

Therefore, it is very convenient to design and implement efficient parallel algorithms for this task. Genetic algorithms have an inherently parallel nature that makes them well suited to implementation in massively parallel hardware like graphics processing units (GPUs), but they have several sequential parts that are difficult to deal with, like the crossover or the selection procedure. Thus, it is important to fine-tune the parallel implementation to benefit from all the potential computational power of the GPU. To allow the practical application of this method to real-life problems (often involving large GRNs), we present in this work a comprehensive study on how to develop efficient parallel implementations of this method for the fine-grained parallel architecture of GPUs. To this end, we evaluate both performance and effectiveness for a wide range of possible elections for parallel GA schemes (master-slave, island, cellular, and hybrid models) and selection methods (roulette, elitist), and we suggest procedures to optimize the use of the GPU's resources.

Main contributions

Because of the intricacy of the GRN dynamics, we have proceeded in a systematic manner to unveil what the most promising GA and the most efficient GPU configurations may be. All along the article, our conclusions are described and justified with experimental data. As a summary, which can be a guide for practitioners and experimentalists to solve the temporal dynamics of GRNs, the working flow of our experiments was as follows:

1. Studying the transformation rules that can be applied to each network state and finding a universal encoding for them.
2. Defining a proper fitness function based on the number of differences in network node and network link values, between the current population individual and the target solution.
3. Implementing and testing most of the parallel versions of GAs, bearing in mind 2 main groups: nonstructured master-slave models and parallel structured GA models (island, cellular, and hybrid).
4. Confronting the behavior of 2 dissimilar selection methods: roulette and elitist.
5. Studying in depth the best model found. In particular, we have discovered a large variation in the results for the island model when varying the migration frequency (number of generations between each migration) and the migration rate (number of individuals migrated).
6. Determining whether the found fitness is near to the optimal or not. In fact, we have found the minimum possible fitness along our experiments.

Although there is much general advice when programming for GPGPUs (General-Purpose Computing on Graphics Processing Units), for the specific case of achieving maximum performance in the study of GA for GRN some important aspects have been unveiled all along this work. Several key points to achieve maximum performance have been found accordingly. In particular, we have determined a different thread pairing for each genetic operator as a parallelization strategy that GAs may benefit from. The complete list of those key points is as follows:

1. First, the proper selection of the number, type, and size of the data objects for the GPGPU is of essential priority.
2. Determination of the best thread pairing for each genetic operator (or any other computation). The most extended and intuitive thread selection in the current literature (pairing individuals with threads) does not usually produce the better results.
3. Reorganization of memory accesses to favor coalescence in the memory access patterns.
4. Careful estimate of GPU shared memory consumption so as the maximum number of threads could fit in it. This estimation usually determines the maximum number of individuals for some models of the GA, such as the cellular model.
5. Management of any type of bifurcation (branch, loops, etc.). On the whole, we have found that it is especially important to avoid thread divergence when applying the GRN rules.

The execution time of the algorithm has been reduced by applying development techniques which optimize the use of resources of the GPU with compute unified device archit (CUDA). A comparative study of the performance obtained by the different versions of the parallel implementation on GPU against a sequential implementation on a CPU has been made.

The proposed parallel algorithm has been implemented into the software cuEVOGENET and is available online at https://github.com/rxp90/cuEVOGENET.

Related Work

Evolutionary algorithms are based on creating a population of possible solutions of a problem and applying procedures that mimic natural evolution to find good candidate solutions within that population: creation of diversity, mechanism of selection, and genetic inheritance. Thus, evolutionary algorithms explore the search space to locate good solutions of the problem. Genetic algorithms[7,8] are a class of evolutionary algorithms in which a candidate solution to a given problem is encoded as a bit string, called individual or chromosome. A population of such strings is chosen at random and evolves over a number of generations under procedures inspired in natural evolution to find good candidate solutions: selection according to a fitness function, crossover to produce new offspring, and random mutation to increase diversity. Genetic algorithms and other types of evolutionary algorithms have been successfully applied to various bioinformatics problems, such as inferring GRNs from gene expression profile data,[9,10] identifying motifs in DNA sequences,[11,12] or generating DNA strands for DNA computing.[13]

Many of the GA operations can be executed in parallel in different processors or processing elements with a high computation to communication ratio. Therefore, GAs are good candidates to be executed in parallel and distributed computers with good speedup respect to sequential executions. This parallelization is very often necessary to obtain solutions of good quality in a reasonable time. Even though GAs are based on simple mechanisms, parallel GAs are complex nonlinear algorithms controlled by many parameters, so that it is difficult to preview which are the best choices for a good efficiency. Different models of parallelization of GA have been studied. A good early overview of techniques employed to implement GAs in parallel is the work by Alba and Troya.[14] Cantú-Paz[15] also presented a classification of parallel GAs and made a systematic study of how the different parameter values affect to the quality of solutions and to the efficiency of the parallel implementation. A recent survey of the state of the art in distributed evolutionary algorithms (including and up-to-date references) is offered by Gong et al.[16] Moreover, this work analyzes in detail the different models of parallel evolutionary algorithms and their characteristics. In addition to the runtime reduction, another advantage of the parallel implementation of evolutionary algorithms is that it can add intrinsic fault-tolerance mechanisms to the optimization algorithm, as shown by González et al[17] for the master-slave model and by Hidalgo et al[18] for the island model.

Parallel GAs have been implemented on GPU for many test problems using various parallelization models, demonstrating that good results can be obtained specially for large populations. Some examples are Luo and Liu,[19] who employed a cellular model; Li et al,[20] who used a master-slave model; and Luong et al,[21] where island model was preferred. Some GPU implementations of GAs are also reported for real-life biologic problems. For instance, Ben-Shalom et al[22] used a master-slave model for a computational neuroscience problem consisting in tuning the parameters of kinetics models for voltage-dependent ion channels of neurons, obtaining a considerable speedup for that problem as compared with a small Linux cluster. At last, a good review paper about parallel implementation of evolutionary algorithms is the work by Arenas et al,[23] which introduces the questions posed by the implementation of this kind of algorithms on GPU and presents a survey of solutions found in the literature.

In recent years, increasing attention has been paid to evolving GRNs. In consonance, a special issue of the journal *BioSystems* was devoted to this topic in 2009,[24] identifying the 3 areas that have been more studied since then. The first area is synthesis of GRNs using evolutionary algorithms. Esmaeili and Jacob[25] employed evolutionary algorithms to discover regulatory networks that optimize multiple stability indicators, including network sensitivity, cyclic length of the attractors, and number of attractors. Nicolau and Schoenauer[26] used evolution to find regulatory networks with specific patterns of connectivity, such as scale-free networks. In the same line and most recently, Noman et al[27] employed an evolutionary algorithm to find GRNs that include robust network modules, using a Monte Carlo method to quantify topologic robustness. They verified their algorithm by identifying GRNs showing 2 classic behaviors: oscillation and bistability. The second area is not related to this work (the reconstruction of biologic genetic networks using computational gene regulatory models and experimental data), so these references are not included here.

Finally, the third area is the application of gene regulatory models and artificial development to solving engineering problems and to build artificial simulated organisms intended to study the working principles of natural systems. In the last decades, *artificial GRNs*, which are dynamical systems inspired in their natural counterparts, have been employed with those purposes. Evolutionary algorithms play a crucial role in finding good solutions of these artificial GRNs. Eggenberger[28] proposed an artificial evolutionary system to evolve 3-dimensional shapes of simulated multicellular organisms. Banzhaf[29] introduced an artificial regulatory network and showed that it can reproduce phenomena found in natural ones, such as heterochrony (a variation in timing of expression) or different dynamical behaviors on protein concentration, ruled by changes in the genome. Knabe et al[30] proposed an artificial GRN to model biologic clocks and showed that it captures many of their characteristic properties. Guo et al[31] proposed a GRN-based algorithm to make an arbitrary number of robots to self-organize into different predefined shapes and even to self-reorganize adaptively under dynamic environments without the need of a centralized control. They employed a multiobjective GA to tune the parameters in the regulatory model to simultaneously minimize 2 objectives: the

total traveling distance of all the agents and the system convergence time. Joachimczak and Wróbel[32] used artificial GRNs evolved by a GA to generate or process signals in which information is encoded in chemical pulses, showing their capability to perform tasks such as doubling the frequency or pulse length of the input signal, acting as a low pass frequency filter, or doubling the number of input pulses. More recently, Cussat-Blanc et al[33] employed a special type of GA (with augmenting topologies) to evolve artificial GRNs that were used to control agents performing different problem domain tasks.

GA for the Temporal Dynamics of GRNs

In this section, we present a description of the GA that we have employed. A pure sequential, elementary implementation of it was introduced by Aguilar-Hidalgo et al.[3] We refer the reader to this reference for more details on the basics of the algorithm. Although this previous work suggested a modelization for this kind of GRN, it did not give an insight in the most efficient GA model nor in reaching the best solution for this network. Maybe this was due to its elevated computing times (several hours for a trial), which made prohibitive the testing of the different possibilities: GA models, parameter adjusting, best number of individuals, mutation probabilities, and so on. Thanks to the chromosome encoding and the GPU-oriented parallelization methods employed here, we verify that the proposed GRN temporal dynamics model provides complete efficiency for the solution of this kind of problem. Indeed, we investigate all of these possibilities and find the implementation that produces better results (both from the performance and the GA fitness perspectives). An important result of our exhaustive study is that a moderate population size is enough to get to very good fitness values. This allows us to compute many of the GA possibilities in less than a minute and finally to find a minimal fitness value. Indeed, we have demonstrated that this value corresponds to the best solution for the analyzed GRN, impossible to diminish.

Gene regulatory networks

A GRN is modeled, using a Boolean network model,[34] as a directed graph $G=(V,E)$, where V is the set of vertices or nodes, which represent genes or proteins, and E is the set of edges on V, which represent regulatory interactions between genes/proteins. An edge between V_i and V_j symbolizes a dependence between them. The state of a node can be described by a Boolean variable expressing that it is active (1) or inactive (0), and thus that its products are present or absent. The graph will be annotated so as to reflect the nature of the dependencies (eg, activator, repressor).

GRN temporal dynamics model

The time evolution of a GRN starts from an initial state A and reaches a final state B. It is expected that the transformation dynamics that turns network state A into state B will be ruled by a certain rule set. The objective of the GA is to find a rule set (1 rule for each network node) that, applied to an initial state A' which is close to A and differs from it on a number of links, transforms it into a final state (B^*) that is close to the desired final state B, as shown in Figure 1.

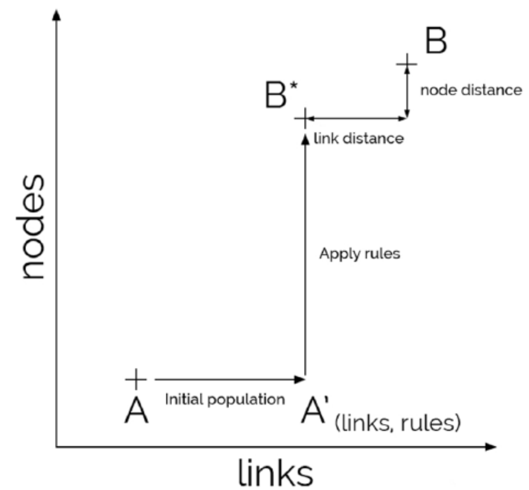

Figure 1. Evolutive transformation dynamics of the GRN. The objective of the genetic algorithm is to find a set of transformation rules that brings from A' (which is close to the initial state A and differs from it only on a number of links) to a final state B^* close to the known final state B.

Each chromosome represents a candidate solution of our problem, including the set of links of state B^* and a set of rules. A' is the state formed by the same node configuration as A and the same set of rules as B^*. The application of this set of rules to state A' (1 rule for each network node) transforms it into a network state corresponding to B^*.

A pseudocode of the GA is shown in Figure 2. At the start, a population of solutions is generated. Each individual is evaluated according to a fitness function. After the generation of the initial population, evolution is allowed to proceed in an iterative manner. In each iteration or generation, following the biologic evolution analogy, several genetic variation operators (crossover, mutation) are applied to introduce diversity in the population. Each resulting population individual is evaluated according to the fitness function. Then a selection procedure is applied, in which the population for the next generation is formed by a combination of some individuals of the previous generation and the offsprings resulting from the application of the variation operators. The procedure is iterated until some stop condition is fulfilled: a predefined number of generations is completed or the fitness value of some individual surpasses a predefined value.

Initial population

The initial population is composed of individuals whose links part is generated by making random permutations of the original network links, that is, by permuting a number of link elements randomly chosen. In this way, a new individual with

```
t ← 0
P(t) ← Generate initial population
Evaluate[P(t)]
while not stop_condition do
    P′(t) ← Variation[P(t)]
    Evaluate[P′(t)]
    P(t + 1) ← Selection[P(t) ∪ P′(t)]
    t ← t + 1
end while
```

Figure 2. Pseudocode of the sequential version of the genetic algorithm.

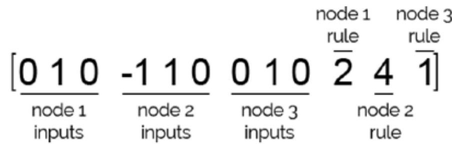

Figure 3. Chromosome example representing a particular possible solution of the problem.

different links is obtained, but the connectivity of the network number of links of the individual is maintained. The rules part of each individual is generated randomly.

Solution encoding

The GA chromosome, which represents a possible solution of the problem, is formed by the model parameters whose configuration is to be investigated: the links among the nodes and the rules applied to each node. The chromosome used in our study is composed of 32 nodes, but for clarity we have shown in Figure 3 the chromosome of an individual with only 3 nodes. The first part is composed of the links that each node has, and the second part includes the rules applied to each of the 3 nodes. The possible values of a link can be +1 (if the link is an activator), −1 (if the link is a repressor), or 0 (if there is no link). A numeric value represents the rule to be applied to each node (see next section).

Transformation rules

The possible rules that can be applied to each network state are as follows:

1. Rule of the most: the final state of the node is determined by the action of most links directed to it. If most of them are activators it will be activated; if most of them are repressors, it will be deactivated. If the number of activators is equal to the number of repressors, the state of the node will remain the same.
2. Absolute repressor: the presence of just 1 repressor link is enough to deactivate the node.
3. Joint action of 2 activators: the node will be activated only if, at least, 2 input links are activators.
4. Joint action of 2 repressors: the node will be deactivated only if, at least, 2 input links are repressors.

The set of rules is the specification of which one of these rules is applied to each particular network state in the GRN. Only links coming from active nodes are considered.

Solution evaluation

Each individual of the population is evaluated, and a score value is associated to it according to a fitness function, based on the number of differences in the values of the network nodes and the network links between the population individual (B) and the target solution (B^*). Hence, the fitness function ϕ is defined as,

$$\phi = \lambda \frac{d(B,B^*)}{\max\left[d(B,B^*)\right]} + \left(1-\lambda\right)\frac{d(L,L^*)}{\max\left[d(L,L^*)\right]},\ 0 \le \lambda \le 1 \quad (1)$$

where $d(B,B^*)$ is the distance between the values of the nodes in network states B and B^*, and $d(L,L^*)$ is the distance between the values of the links for B and B^*, considered both distances as the number of differences between the nodes (or links) values. The better the individual, the lower its numerical fitness value. Here, λ is a regulatory parameter that determines the relative importance of each term (nodes or links) in the global calculation of the evaluation. As in the article where the model was introduced,[3] the following value for the λ parameter has been chosen:

$$\lambda = 0.9 \quad (2)$$

The reason for using a high λ value is that the amount of link variability found in these kinds of biologic problems is usually much higher than that of nodes.

Genetic operators

Crossover. In each iteration of the GA, just the best 2 individuals from the previous generation are kept, that is, *elitism* is applied. The rest of the individuals of the population are generated by applying crossover to couples of the remaining individuals. To select the parent individuals of each crossover operation, the *roulette* method is applied: all the individuals are capable of being selected as parents with a probability that is directly proportional to the normalized fitness value of the individual. Thus, individuals with a higher fitness value have a higher probability of generating new individuals and keeping part of their genetic information in the population. As a result, the fitness value of the average individual should increase with the number of generations. However, the crossover operations also introduce some variability so that the capability of exploring the search space is increased.

As the chromosome is formed by 2 parts, rules and links, it is necessary to determine 2 related crossover points for each crossover operation. The links crossover point is determined randomly and the rules crossover point is determined according to the former election following the expression:

$$K = int\left(\frac{l}{N}\right)+1 \quad (3)$$

where K is the rules crossover point, l is the links crossover point, and N is the number of nodes in the network. In this way, we calculate how many nodes are involved (ie, with all of their inputs being crossed) in the crossover and select also the rules attached to them.

A low connectivity is frequently observed in biologic networks.[35] Thus, to keep the connectivity of the network within low values, each new individual generated by crossover has to pass a connectivity test before being accepted as part of the new population. Only the individuals with a number of inputs per node smaller or equal to 7 are accepted. Otherwise, the individual is rejected and a new individual is generated by crossover.

Mutation. After a new individual has been generated by crossover, it undergoes a mutation process in a probabilistic manner. There is a mutation probability for each one of the parts of a chromosome: μ_1, the mutation probability for the links portion of the chromosome; and μ_1, the mutation probability for the rules portion of the chromosome. Mutation is individually applied to each element of the chromosome according to its probability value. In the case that mutation has to be applied for an element, a new value for it is randomly chosen. All the possible final values after mutation have the same weight in this random choice: 1/2 for links (changing to 1 of the 2 remaining possibilities) and 1/3 for rules (changing to 1 of the 3 remaining possibilities).

Selection/replacement. The population of generation $t+1$ is formed by the best 2 individuals from generation t and the individuals obtained by applying the crossover and mutation operators to the remaining individuals from generation t. Thus, the total number of individuals in the population is unchanged.

GPU Architecture and Parallel Programming

During the last decade, several reasons have pushed up GPGPUs as an excellent platform to conduct heavy tasks with a high degree of parallelism. One of the reasons for their success is its inherent general-purpose manycore-based implementation, in contrast to other less flexible architectures like Application-Specific Integrated Circuit (ASICs) or Field-Programmable Gate Array (FPGAs). Moreover, several additional advantages have thrusted their success for an ample set of scientific applications. First, the very low cost per processing unit, which is supposed to persist, since data center and PC graphics markets subsidize GPUs. Second, the annual growth of performance and memory bandwidth is predicted to remain very high.[36] For example, many existing GPUs, with around 3000 floating-point arithmetic cores, can achieve a performance above 5 Tera FLoating point OPerations per Second (TFLOPS) and around 300 GB/s within 1 chip. One of the consequences of this advantage with respect to its competitors is that many top leading supercomputers are GPU-based machines.[37] Besides, with the advent of new architecture capabilities of GPGPUs, and the expansion of tools, libraries, and hardware abstraction mechanisms, GPU software development is not now a tricky and bizarre programming task like it was in its former stages.[38] Debugging and profiling tools like those from CUDA[39] are more and more reliable. Previous reasons have pushed the scientific community to incorporate GPUs in several disciplines, and it seems that this tendency will continue for the next decade. Compute unified device architecture is the NVIDIA programming platform that has been chosen to implement current work.

A GPGPU[39] includes an array of streaming multiprocessors (SMs). Nowadays, each SM consists of 8, 16, or 32 floating-point Scalar Processors (SPs); 1, 2, or 4 Special Function Units; 1 or 2 multithreaded instruction units; and several pieces of memory. Each memory type is intended for a different use; thus, if the programmer were conscious of memory management, it could extract any drop of performance. Each GPGPU has nowadays 1 or 2 *warp schedulers* that select at any cycle several groups of threads for execution on each SM in a round-robin fashion. A warp is simply a group of 32 hardware-managed threads. As the number and types of threads are enormous, a 5-dimension organization is supported by CUDA, with 2 important levels: a grid contains blocks that must not be very coupled while every block contains a relatively short number of threads (usually 512 or 1024), which can cooperate deeper (eg, by sharing data through on-chip shared memory and fast synchronization primitives). If a thread in a warp issues a costly operation (like an external memory access), then the warp scheduler switches to a new warp, to hide the latency of the other thread. To effectively use the GPGPU resources, each thread should operate on different scalar data, with a certain memory access pattern.

Implementation of the GRN Temporal Dynamics on GPUs

Parallel-oriented GA models

We have implemented different parallel versions of the GA: a *nonstructured* master-slave model and 3 parallel *structured* GA models—island, cellular, and hybrid model (Figure 4). In a nonstructured model, the GA uses a single population, and the GA procedures or operators (selection, crossover, mutation) are applied on them as a whole. In particular, all individuals can undergo crossover with one another (panmixia). In contrast, in a structured model, the population is divided in groups so that each individual can only interact with individuals within its group.

Master-slave model. In this model, the GA is composed of a single monolithic (panmictic) population. To use the huge parallelism of the GPU, different operations of the algorithm can be executed independently for each individual of the

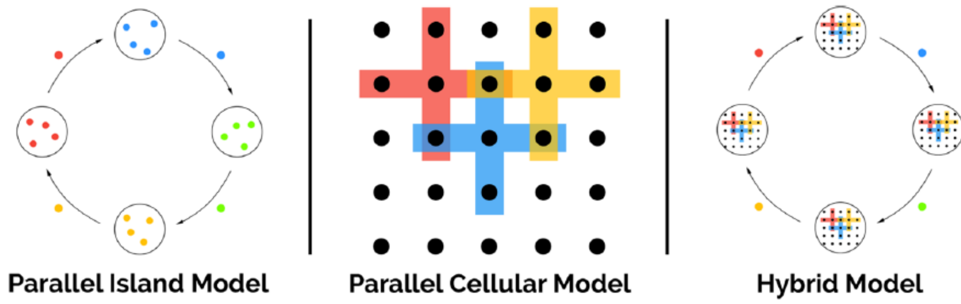

Figure 4. The 3 different models of parallel *structured* genetic algorithm that have been studied.

0.12	0.24	0.75	0.14	0.48	0.88	0.90	0.76	Initial population
1	2	3	4	5	6	7	8	Indexes
1	4	2	5	3	8	6	7	Indexes sorted by fitness
1	4	2	5	3	8	6	7	Couples determined by elitist selection
0.12	0.24	0.76	0.14	0.48	0.88	0.90	0.75	

Figure 5. Elitist selection implementation for the master-slave model. From top to bottom: fitness values of an initial population, increasing indexes, fitness sorting results expressed over the indexes, adjacent couples to be mated, and individuals to be mated according to this criterion.

$t \leftarrow 1$
$P(t) \leftarrow$ Generate initial population
Evaluate$[P(t)]$
while not `stop_condition` **do**
  $P'(t) \leftarrow$ Variation$[P(t)]$
  Evaluate$[P'(t)]$
  $P(t+1) \leftarrow$ Selection$[P(t) \cup P'(t)]$
  if t **mod** `migration_gap` $== 0$ **then**
    Select migrants
    Perform migrations
  end if
  $t \leftarrow t+1$
end while

Figure 6. Pseudocode of the parallel version of the genetic algorithm following the island model.

population on different processing elements of the GPU (see section "Optimizations for GPUs"). It is worth to mention that the implementation of the elitist selection has been targeted to reduce the GPU memory bandwidth consumption. Given the fitness values of an initial population (see Figure 5), sorting results are collected over their indexes. These indexes are then sorted by fitness values, so that adjacent couples are selected. The fourth row of Figure 5 shows an example of this selection, whereas its last row indicates which individuals (represented by their fitness values) are actually to be mated according to this criterion.

Parallel island model. In the island model, also called "coarse-grained" or "distributed" GA model,[15,40] the population is partitioned into a set of subpopulations or *islands*. An isolated GA is executed in each island and a number of individuals are copied to another island (migrations) periodically, as shown in Figure 4. This strategy has 2 advantages. First, a performance improvement can be obtained on a parallel computer. If we can execute each island concurrently on a different processor, the computing time of the algorithm can potentially be decreased substantially. Second, the presence of parallel populations and their interaction introduce a new dynamics that improve the numerical behavior and execution time of the algorithm by obtaining a higher genetic differentiation and a better sampling of the search space.

A pseudocode of our parallel version of the GA for GRN temporal dynamics using the island model is shown in Figure 6.

Different choices or parameter values are involved in the design of a parallel island GA: size and number of islands, migration topology (topology that interconnects the islands, for example, ring, star, or fully connected), migration frequency (number of generations between each migration), migration rate (number of individuals to migrate), and migration policy (selection of those individuals to migrate and those to be replaced in the destination island). As will be discussed in the section "Experimental results," the choice of parameter values can affect the efficiency of the parallel algorithm.

The methodology to implement migrations in the parallel island model is described in Figure 7. Information on islands was stored in the GPU memory. An elitist selection was carried out in each island. Then, the crossover operator is applied. To save computing time, the population was not sorted again after crossover: it is assumed that the crossing of good individuals will give rise also to good individuals. After that, the best n elements of each island are migrated to the destination island (being all the islands arranged in a ring topology).

Parallel cellular model. In a cellular model (Figure 4), the whole population is arranged in a grid with a certain topology, and each individual can only mate and compete with other individuals included in its neighborhood. The neighborhoods overlap, so that individuals with good fitness value can propagate through the entire population. A cellular model is thus fine-grained and spatially structured.

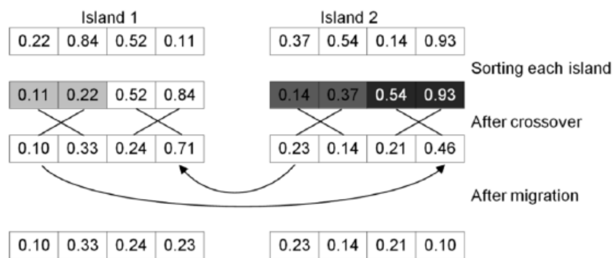

Figure 7. Description of the migration methodology used in the parallel island model.

We have implemented a cellular parallel model in which elitist selection is used: each element is crossed with its best fitness neighbor as shown in Figure 8.

Hybrid model. A hybrid model is composed of islands that include a cellular model inside each one (Figure 4). Hybrid models have not yet been used as frequently as island or cellular ones, because they involve a higher level of complexity, both in programming and in choosing the right values of the parameters of its more complex topologic structure. However, they can be good candidates to extract the potential power of parallel architectures with a highly hierarchical memory, such as GPUs.

Optimizations for GPUs

On the whole, GAs, as most bioinspired algorithms, are parallel in nature, which might make them good candidates for massively parallel hardware like GPUs.[23,41,42] However, there are several sequential parts that are difficult to deal with (like the selection procedure). Hints about GPU optimization of these parts are unveiled in this section.

One of the most important aspects that reduces time computation is executing everything directly at the GPU, thus avoiding transfer times between CPU and GPU. With current GPU architecture capabilities and software tools (mainly the all-purpose cuRAND, cuBLAS, and Thrust), this is perfectly possible and this policy has been followed all along this work.

Some authors have even published an ordered points checklist for maximizing GPUs' efficiency.[43] In this respect, outstanding soft computing methods and excellent researchers meet with a profound challenge when trying to program in highly parallel platforms.[44] On the other side, it can be easily understood that tuning GAs to be highly parallel and efficient over GPU platforms leads to implementations that are difficult to implement into other architectures.

Although previous studies serve to depict a first scene of the main optimizations and challenges that a GA designer must face to when programming for GPGPUs, for the specific case of GRN other important aspects have been unveiled in this work. A lot of preliminary tests were carried out to discover the impact on performance of several programming decisions. In some occasions, we have detected that the most proper choice depends even on the GA model. During this evaluation, monitoring internal GPU performance through the Compute Visual Profiler[45] for NVIDIA technology was used. Through the use of this tool, we have found several key points to achieve maximum performance.

First, the proper selection of the number, type, and size of the data objects for the GPGPU is essential. In our case, GRNs contain lots of state information that consists on binary variables in most occasions. Although compressing this information in binary format would have supposed a reduction of total amount of consumed memory, a set of initial test did advise us that this compression was not beneficial from the performance point of view. The implementation of bit management operations was the main cause of this. As a consequence, we comprised most of the individual information, but each state value was stored in a byte.

Special caution has been taken with shared memory consumption. The total amount of shared memory for the different parts of the GA has been carefully measured so as the maximum number of threads could fit in the 48 kB of the tested GPU (see section "Experimental results").

The second important decision is the proper launching of thousands or millions of very light threads. The most intuitive thread selection is pairing individuals with threads (all cited references here except one follow this method). Nevertheless, as some authors have pointed out, other possibilities can produce better results. For example, in Jaros,[46] the author used a per-warp assignment, because in this case (solving the knapsack problem) warp granularity did not limit the size of individual (using appropriate mapping). In our case, we have stretched out a little more the thread assignment, mainly with the aim of avoiding GDRAM accesses by fitting the used memory with that of shared memory. This implies a different thread pairing for each genetic operator or other computations:

- *Crossover.* Each block crosses only 2 individuals. A reduction process is applied to the 512 threads of a block to compute if the connectivity of successors preserves the correct number of links according to GRN hypothesis.
- *Mutation.* A thread-individual pairing is preferred because mutation rules do not need individual communication.
- *Selection/replacement.* The pairing for this step must be carefully decided for each genetic model. Our results clearly reveal some pairings for each model, which can be extended for other GAs. Briefly, we can state that (a) if roulette is chosen, each individual has no relation with the others, and only a GDRAM access to the random numbers is necessary; thus, a thread per individual selection pairing is preferred; and (b) for elitist selection, CUDA libraries (like Thrust[47]) propose a sorting by key with a reduced number of threads.

Moving pointers instead of moving the complete structure of individuals reduces the precious memory bandwidth, which yields to important benefit on CPU, which is even larger for GPU times.

Figure 8. An example showing how the best fitness neighboring element is chosen for each element in a 9-individual cellular population. Top: from left to right, first we depict the fitness value of 9 individuals arranged as a cellular model. On the next 3 diagrams, the Moore neighbors (considering a torus topology) for the individuals situated at (1, 1), (0, 0), and (2, 2) are marked with shadows. Bottom: left, coordinates of the best neighbors for the 9 individuals; right, a possible result after the crossover operation.

- *Rule application.* A node is paired with a thread. As the number of nodes of the analyzed GRN is 32, a warp contains an individual. For each node (thread), only that useful information (connected links and their state) is uploaded to the shared memory, thus reducing memory transferences.
- *Fitness computation.* Same pairing as rule application because similar reasons apply.

Previous thread assignment and data partitioning to fit on-chip shared memory reduce considerably main memory accesses. However, if this assignment wants to be preserved to keep previous benefits, some consequences on the maximum number of individuals appear for some GA models. Section "Study of fitness of cellular parallel model" gives details for this circumstance.

The next decisive factor on performance is the management of any type of bifurcation (branch, loops, etc.). On the whole, thread divergence must be avoided for most cases, meanwhile coalesce memory accesses should be reinforced to maximize GPU utilization. There are several conditional structures in GRN dynamics where CUDA compiler is not able to introduce predication. These were detected using the NVIDIA Visual Profiler[45] and considered of high priority by this tool. The most evident example is the choice of one of the 4 rules for each node (see section "Transformation rules"), when the rules are iterated 5 times for each individual. Other example is the selection of a random value after a mutation. These have been rewritten using lookup tables and logic bitwise operations, which introduce an important speedup.

Another issue that must receive attention is the pattern memory access. The CUDA manual[39] provides detailed algorithms to identify types of coalesced/uncoalesced memory accesses. Some reorganization of memory accesses was introduced in our codes to favor coalescing. Crossing operation requires memory accesses that may have a variable stride if not enough attention is paid. In our case, instead of taking the most evident decision, which is that any thread copies 2 consecutive nodes, the code was rewritten so that a thread copies 1 node and other with a distance of 512. This allows a stride equal to 1 between threads.

At any piece of code, some speedup can be achieved. A simple test with 1 kernel shows different speedups for a block when evaluating 1024 individuals (using elitist model). Avoiding thread divergence when applying rules can achieve a 1.44× speedup, an acceleration of 1.66× is obtained when vectorizing loads when accessing individuals (using casting operators to read 1 int4 elements at once instead of 16 independent bytes), and finally, the vectorization of node link accesses when computing fitness values decreases time by a factor of 1.26.

It is worth to mention that previous optimizations can be used in other parts of a GA, thus obtaining additional speedups. A relevant case occurred for migration of individuals when the island model is used (see section "Study of fitness of parallel island model"). Timing may vary significantly depending on the pairing between threads and individual information to be transferred. A comparison for 512 individuals (with a rate migration of 16 individuals each 16 generations) between 2 cases can give us an idea of the importance of this issue. In a first test, a thread was in charge of transmitting 4 bytes (1 int). In a second test, transferring was vectorized, so each thread transfers 16 bytes (1 int4). Because the resultant bandwidth is 10 times higher for the vectorized case, the execution is speeded up 3.5 times.

Experimental Results

A thorough study of various parallel implementations for the temporal dynamics of GRNs was carried out with a twofold aim: corroborating that GPUs are an excellent choice due to the huge achieved speedup with respect to a sequential CPU and selecting the best GA model (from fitness values evolution, scalability, and performance points of view). To achieve this goal, we have analyzed different parallel implementations for the GA which is the base of the studied algorithm: master-slave, island, cellular, and hybrid models. Moreover, we tested for GPUs the behavior of 2 different individual selection methods: roulette and elitist.

The tests were conducted for 2 GPUs shipped with a difference of around 5 years and 2 PCs with a similar age disparity. Speedup comparisons were done for machines of the same period, which allows confirming that achieved GPU speedup is being not only maintained but also improved with much modern GPUs. PC processors were Intel Core i7-4712MQ running at 3.3 GHz and Intel Core 2 Quad Q6600 at 2.40 GHz (2 GB RAM DDR2). The 2 GPUs were an NVIDIA GeForce 820M equipped with 96 CUDA Cores and an NVIDIA GeForce GTX 660 with 960 CUDA Cores. Five trials were executed for each experiment, and the mean fitness values are shown in the graphs of this article. Nevertheless, the differences between the several trials were minimal. The GA was applied to the GRNs of the eye of the mouse (*Mus musculus*) during the embryonic stages of days 9 and 10.5 after the fertilization. These stages are called E9 and E10.5, respectively, and are described in Aguilar-Hidalgo et al.[3]

Study of master-slave (panmictic) model and selection method

Apart from the parallelization and optimization for CUDA architecture explained in section "Optimizations for GPUs," other important decisions were taken in relation with the memory allocation and the random number selection. For both selection methods, elitist and roulette, memory management was the same. To reduce the memory-based operations, we generated a population-length array with the indices of each individual within the population and we applied the selection method to this array (instead of applying to the whole data of total population). More specifically in the roulette method, and with the aim of saving memory accesses, we employed an empty array whose positions were filled with the index of the individuals selected by a roulette wheel algorithm. Some important considerations of each selection follow. In the case of the elitist method, a Thrust function *sort_by_key()* was used, so we obtained a sorted array of indices corresponding to the current position of the individuals that they point to. For the roulette selection, we tested all the possibilities offered by the *curand_init()* function, which is included in the cuRAND library, to generate high-quality pseudorandom numbers in a simple and efficient manner.[48] The key to prevent slow random number initializations is setting cuRAND states initially with a different seed per each thread. Instead of initializing all the threads with a same seed (which preserves a very good distribution uniformity), we assign the current clock time plus the thread identifier (modified by a hash function) as a unique seed per thread (Figure 9). This allows a fast parallel initialization of all the random number states, whereas it prevents the use of the best quality random numbers. Although the number distribution is not as uniform for our approach as for the other one, in our case GA

An illustration of using the same seed and different sequence numbers for each thread

```
__global__void setup_kernel(curandState* state )
{
int id = blockIdx.x*blockDim.x + threadIdx.x;
unsigned intseed = 1234;
curand_init(seed, id, 0, &state[id]);
}
```

An illustration of using different seeds and the same sequence number for each thread.
A simple hash function is introduced to improve the seed disparity.

```
__global__void setup_kernel_V3(curandState* state )
{
int id = blockIdx.x*blockDim.x + threadIdx.x;
unsigned intseed = (unsigned int)clock64();
curand_init(WangHash(seed) + id, 0, 0, &state[id]);
}
```

Figure 9. Two manners of generating random numbers via *curand_init()* function.

executions do not suffer from noticeable fitness variations (along 5000 generations). Meanwhile, performance disparity grows in a huge manner. Specifically, our tests reveal that the second case is more than 200 times faster for 256 individuals and more than 3000 times for 1024 individuals.

When comparing the performance of selection methods for sequential processors, it is obvious that the roulette wheel selection requires less time than the elitist one, because the first one does not need to sort any indices. Of course, things are very different for multithreaded-oriented architecture thanks to the parallel processing ability of CUDA systems. Time complexity order of parallel GPU implementation (using many processors and additional scratch memory space) being proportional to the logarithm of the amount of numbers to be ordered implies that, for a relative small number of individuals (around 6000), the overhead time injected by the sorting is less than that of the random function calls (see Table 1).

When representing the speedup of the parallel GPU versus the sequential CPU version for both selection methods (Figure 10), it is discovered that GPUs can yield speedups with respect to CPUs. Comparing modern medium class CPUs with GPUs (both shipped around the same year), we can reach speedups of up to 13 times (for the elitist method with a population of 20 480 individuals; being a little inferior when using the roulette method).

In addition, multiple tests were launched to address the question of determining some reliable values for link and rule mutation probabilities. After this study, we concluded that excellent results can be obtained using low values for elitist selection (we chose .001 for both link and rule mutation probability values across this work). On the contrary, these values for roulette selection are not so clear (maybe because of the randomness introduced by the selection algorithm). The mutation probabilities that were chosen for roulette selection in our study were .001 for links and .06 for rules.

In spite of all the above performance reasons, the most important issue that confirms the superior behavior of elitist selection is the evolution of the fitness. In Figure 11, elitist

Table 1. Times required (in seconds) for the elitist and roulette wheel selection methods for the 2 tested architectures.

POPULATION	SEQUENTIAL I7. ELITIST SELECTION	SEQUENTIAL I7. ROULETTE WHEEL SELECTION	GTX660. ELITIST SELECTION	GTX660. ROULETTE WHEEL SELECTION
100	6.2	3.1	19.1	13
200	11.3	6.3	19.7	13.4
500	29.3	16.5	19.9	18.7
1024	60.3	32.4	21.6	17.3
2000	114.7	64.3	26.2	21.1
5000	298.7	190	34.5	31.1
7000	404.3	308.3	40.8	40.9
10240	550.5	470.1	51.1	58.5
20480	1078.7	1260.9	82.8	105.5
30720	1619.7	2430.3	144.9	229.7

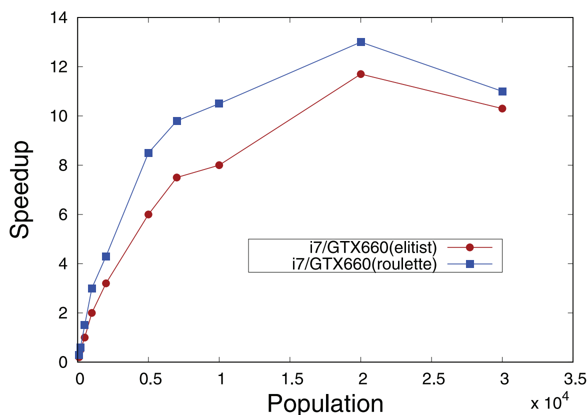

Figure 10. Speedup of sequential version running on an Intel i7 CPU versus parallel master-slave compute unified device architecture (CUDA) version running on a Nvidia GTX660 (without island or cellular model) using elitist or roulette selection.

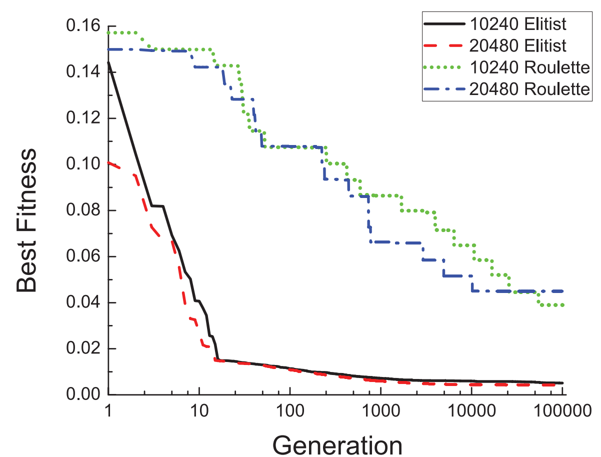

Figure 11. Evolution of best fitness values for different selections and populations. Each value represents *population selection*. Best fitness is here the mean of the minimum fitness values of 5 trials.

and roulette fitness values are represented versus the number of generations and for 2 different populations. It is patent that in a few generations, elitist method reaches fitness values that surpass even the minimum ones that roulette selection can achieve for many generations. Moreover, this figure also allows one to draw another important conclusion: a moderate number of individuals (a few thousands) is enough to get to minimal fitness values in a modest number of generations. To sum up, in this section we can ensure that a few thousands of individuals using elitist selection produce good results (from the performance, scalability, and GA fitness perspectives) for emulating the temporal dynamics of GRNs using parallel GAs. In the rest of the article, this selection method and number of individuals are to be employed with the aim of determining the best parallelization scheme. In fact, tests of the next sections are going to corroborate that this number is enough, yielding in some cases to the optimal fitness value (see section "Solution found").

Study of fitness of parallel island model

The design of an island parallel GA involves various choices: size and number of islands, migration topology, migration frequency (number of generations between each migration), migration rate (number of individuals migrated), and migration policy (which individuals migrate and which are replaced in the receiving island). The choice of values for those parameters is important and intricate, because each parameter can affect the quality of the search and the efficiency of the algorithm in nonlinear ways.[15,40] As shown theoretically and experimentally for one particular problem in Lässig and Sudholt,[49] migrations can make populations to gain information by communication, leading to a better fitness value, and periods of independent evolution can cause information to be lost, leading to an increase in diversity. Thus, the values of the parameters of the parallel island model can lead to important differences in the efficiency of the algorithm. Therefore, we

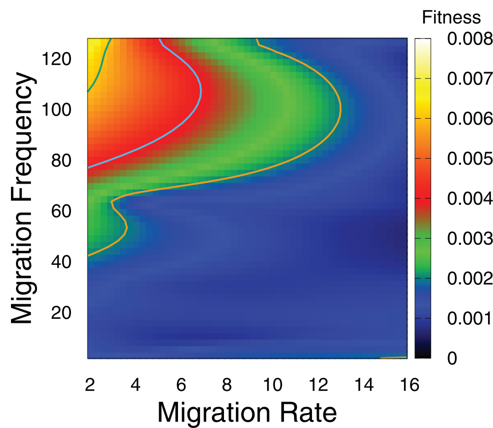

Figure 12. Effect of the migration frequency (number of generations between each migration) and migration rate (number of individuals migrated) on the fitness value, for 2 islands with a population of 512 individuals in each of them.

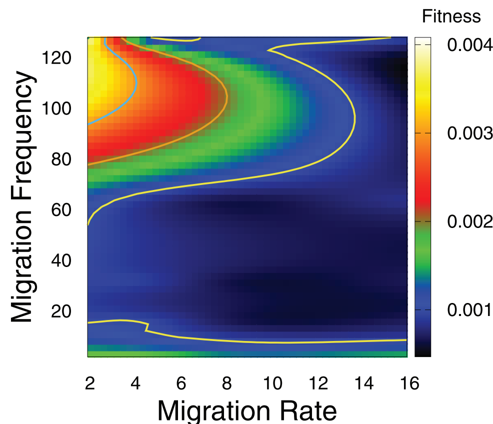

Figure 13. Effect of the migration frequency and migration rate on the fitness value, for 16 islands with a population of 64 individuals in each of them.

have carried out an experimental study to study the effect of the values of the most important parameters in the efficiency of our problem and to find the right elections for them.

Cantú-Paz[15] shows that fully connected topologies may be the best choice when the number of available processors is small, because they offer more potential sources of information leading to a better fitness value. This is not a restriction in our study, because GPUs are equiped with a large number of processors. Also, in a systematic study carried out by Muelas et al,[50] they found that topologies have minimal influence in the algorithm performance. Therefore, to simplify our study, we have considered a simple ring topology. Our results corroborate that this is a good choice for our algorithm on GPU.

We have made a systematic study to determine experimentally the best choices for the size and number of islands, the migration frequency, and the migration rate. In Figure 12, we can see a contour plot of the fitness values obtained by 2 islands of 512 individuals by modifying the migration frequency and the migration rate. The same is studied in Figure

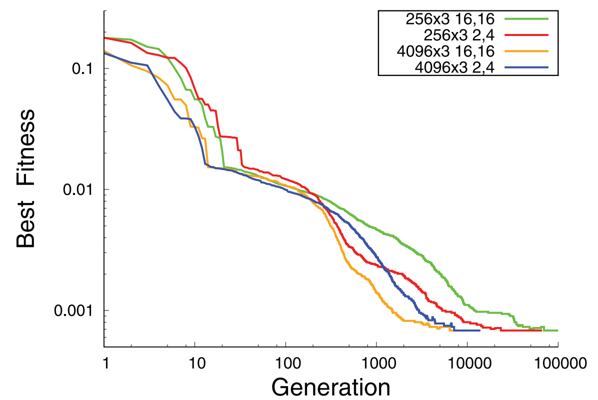

Figure 14. Fitness evolution for the parallel island model using different parameters. Notation: "population per island × number of islands migration rate, migration frequency." The number of islands is 3 in all cases. Log scale on vertical axis.

13 for 16 islands with a population of 64 individuals in each of them. It must be taken into account that a low number of generations between migration implies a high coupling, whereas a high number implies a low coupling. We can notice that there is a relationship between the number of individuals to migrate and the migration frequency, especially for a small number of islands: for frequent migrations, a better result is obtained by migrating few individuals, but for more infrequent migrations, it is necessary to migrate more individuals. Based on our tests, we chose 2 couples of values of migration frequency and migration rate that are opposed with respect to these parameters and offer in general excellent results for our model. These values are to be used in subsequent experiments:

- Migrating 2 elements every 4 generations.
- Migrating 16 elements every 16 generations.

The fitness value evolution for the parallel island model using those parameters is shown in Figure 14, and the number of generations needed to reach the optimal solution (see section "Solution found") for each case is presented in Table 2. We can notice that the efficiency of this implementation is very good, reaching the optimal solutions for all the studied cases by using a moderate number of individuals (around a few thousands) and a fairly small number of generations (from a few thousand to a few tens of thousand generations).

Study of fitness of cellular parallel model

We can notice that during the first generations, the master-slave model is able to find individuals with better fitness values. This is probably due to the strength of the global elitist selection for the total population used in this model, whereas cellular model uses a local selection strategy. However, after some hundreds of generations, the global strategy of the master-slave model begins to saturate, and hence, the cellular model with only 256 individuals (Figure 15) is able to obtain the same fitness value as the

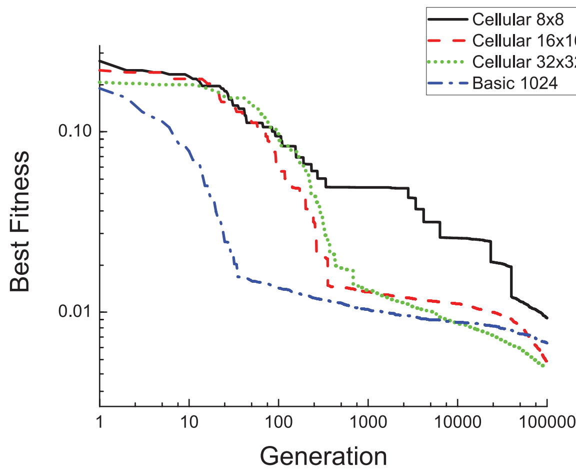

Figure 15. Fitness evolution of cellular models arranged as squares of 8×8, 16×16, and 32×32 individuals. Their results along generations is compared with a master-slave model with elitist selection (named in the caption as "Basic 1024 "). Log scale on vertical axis.

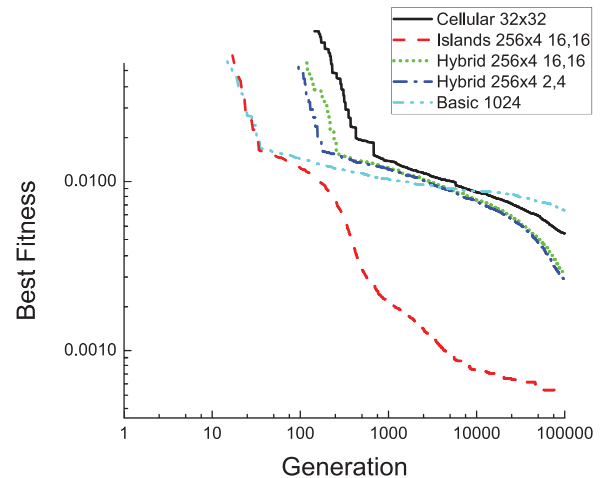

Figure 16. Fitness evolution along 100 000 generations. Cellular versus island and master-slave models are compared. In addition, results for 2 hybrid models are also contrasted (see section "Study of fitness of hybrid (island + cellular) parallel model"). Legend: Island model parameters: "population per island × number of islands." Number separated by commas are "migration rate, migration frequency," respectively. Both hybrid models use a square of 16×16 individuals and 4 islands. "Basic" refers to the master-slave model. Log scale on vertical axis.

Table 2. Number of generations needed to reach the optimal solution for the parallel island model using different parameters (island size, migration rate, and migration frequency). The number of islands is 3 in all cases.

ISLAND SIZE	GENERATIONS UNTIL SOLUTION (2/4)	GENERATIONS UNTIL SOLUTION (16/16)
256	68 464	58 970
512	58 382	36 575
2048	10 951	10 688
4096	6307	5125

master-slave model with 1024 individuals. We assume that the richer interchange of individuals inherent to cellular models is responsible for introducing more genetic diversity, and hence finding incessantly better individuals. Taking this into account, the question is whether we can gain the best of this beneficial behavior and the previously effective performance of the island model by fusing both methods in a hybrid model (see section "Study of fitness of hybrid (island + cellular) parallel model.")

Finally, we have investigated whether cellular model can outperform the island one. In Figure 16, we have compared the cellular versus the island model, for the same whole population. We can notice that the island model offers results that are clearly better than the cellular approach, both in terms of fast convergence and of best fitness value obtained.

Study of fitness of hybrid (island + cellular) parallel model

We have studied the performance of a hybrid parallel model employing islands that include a cellular model inside each one. To simplify the implementation of the hybrid model and to keep the runtime low enough, we have only employed elitist selection in the cellular algorithm inside each island, selecting randomly the individuals to be migrated between different islands. The results are shown in Figure 16. It can be noticed that the hybrid model outperforms the pure cellular one in terms of fitness value. It must be taken into account that the population has been restricted to 1024 because we could not execute a cellular model with a population higher than 32 × 32. The reason is that it would need more cache memory than available in the employed GPU. Using this relatively low population, a final comparison with a 4-island model can also be observed in Figure 16. As can be seen in the figure, the island model outperforms broadly the rest of models from the fitness perspective. Moreover, it is also better from a runtime viewpoint and is the only one that can reach the optimal fitness value for the evaluated genetic network (see next section) in less than 100 000 generations.

Following the same reasoning, one should think that island model comprises the 2 mighty factors for discovering the best solutions: first, an elitist selection for a few hundreds of individuals, capable of finding good individual in a short number of generations, and second, the introduction of genetic diversity, in this case, via a relatively frequent and numerous migration.

Study of fitness versus runtime

This section contrasts all the GA models from the runtime perspective. We selected those configurations that produced good fitness value results and compared them, using the same number of individuals. The aim was to answer the following question: which model produces the best fitness value in less

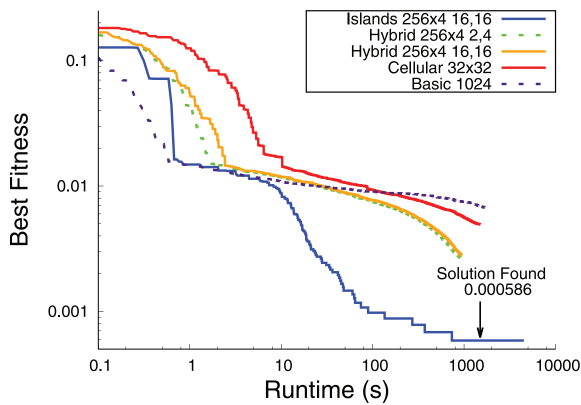

Figure 17. Comparison between all the considered implementations of the algorithm in terms of fitness value versus runtime along 100 000 generations for the same number of individuals. Legend: Island model parameters: "population per island × number of islands." Number separated by commas are "migration rate, migration frequency," respectively. Both hybrid models use a square of 16 × 16 individuals and 4 islands. "Basic" refers to the master-slave model. Log scale on vertical axis.

time? Thus, we ran for 100 000 generations those configurations and depicted the evolution of fitness versus time (Figure 17). The above commented memory restriction for the cellular model implies using 1024 individuals for it (a 32 × 32 square configuration). Thus, other models can adapt to this number: master-slave with 1024 individuals, hybrid (cellular with four 16 × 16 islands), and island models using 4 islands with 256 individuals each. Two opposed migration values (16 migrations each 16 generations, and 2 each 4) were also tested for hybrid models. Hybrid models were the fastest to execute 100 000 generations, but they provided fitness values much worse than those of the island model. In fact, results demonstrate that the island model clearly outperforms the rest of the models in fitness after 100 000 generations. In addition, it is the only one that could reach the optimal solution (a fitness value of 0.0005859375 as shown in section "Solution found"). In particular, a 1024-individual island model reached the optimal solution in less than 3000 s for this figure. However, we carried out other tests for the island model with a bigger number of

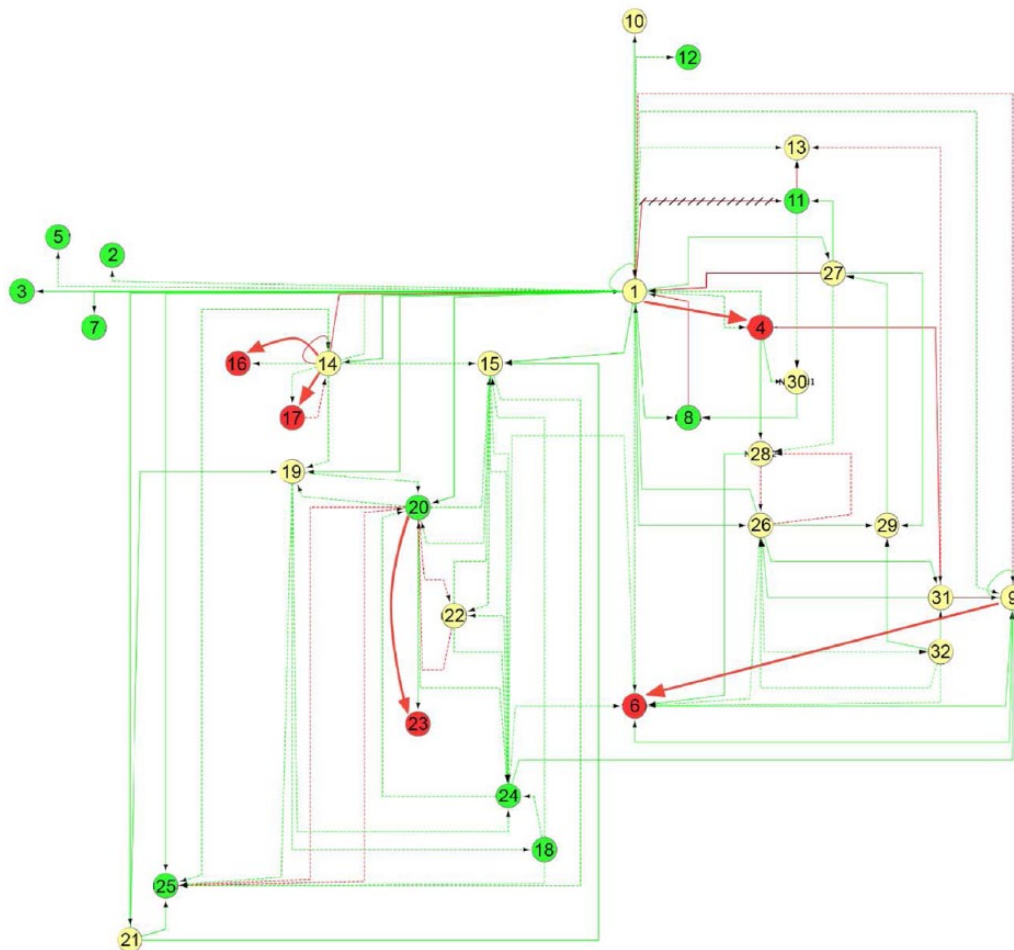

Figure 18. Best solution found by our algorithm. Green nodes: active in E10.5, red nodes: active in E9, and yellow nodes: active in both E9 and E10.5. Red links represent repressors, green links represent activators, solid links represent transcriptional interactions, and discontinuous links represent protein-protein interactions. Links added by our solution are highlighted with thick lines, and the only link deleted by our algorithm is crossed out.

individuals, and the optimal solution was reached even quicker (around 600 s using 12 000 individuals).

From another perspective, if we consider time as the main factor, thus interrupting the simulations at a certain time, it is discovered that island models reach always a much better result than the rest of the configurations. As a result, we can conclude that the implementation that produces better results (both from the performance and the GA fitness perspectives) is simulating a few thousands of individuals using elitist selection under the cellular island model.

Solution found

Along the several tests that we performed to discover the best solutions for the dynamics of the analyzed GRN, we found a minimal fitness value that was impossible to diminish. This is due to the acting rules that model the temporal dynamics, which imply that a lower bound on the fitness can exist. This is an additional achievement of our algorithm implementation, due to the fact that it allows one to run many generations in a few seconds, so that the time needed to reach a very low fitness value has been considerably reduced by using the GA island model over a modern GPU. The network that produced the best fitness value is represented in Figure 18. The initial active nodes (ie, genes that are expressed at the embryonic stage of day 9 after fertilization, or *network E9*, according to the notation of Aguilar-Hidalgo et al[3]) are depicted with red circles and the final active nodes (genes that are expressed at the embryonic stage of day 10.5 after fertilization, or *network E10.5*) with green ones. This involves a transformation dynamics of the GRN from the initial network E9 to the final one E10.5 (see Figure 1). The best solution needs that nodes 4, 6, 16, 17, and 23 change from active to inactive. However, all of these nodes receive activator links, and hence they cannot be made inactive. Additional repressor links (thick red links in Figure 18) were added by the best solution found to change the state of these nodes. This implies that the best network can have the same node state as E10.5, but inevitably 6 links must be different, with the optimal fitness value being the following:

$$fitness = 0.9 \times \frac{0}{32} + 0.1 \times \frac{6}{1024} = 0.000586 \quad (4)$$

Conclusions

Conclusions are twofold: first, we present a guide for practitioners and experimentalists to employ GPU technology as a tool to study the temporal dynamics of GRNs, that is, its temporal evolution between known initial and ending states by using Boolean rules and a GA. To conduct this study, a proper combination of 3 different fields that are at the forefront of current research in high performance computing has been required: parallelization, machine learning using metaheuristics or novel algorithms inspired by nature, and big challenge applications. This analysis has implied the study of a wide variety of parallel GA models: master-slave, islands, cellular, and hybrid. To extract all the potential power of the GPU, we show how to apply development procedures that optimize the use of the GPU resources using CUDA. A thorough comparative study between the performance of the different versions of the parallel implementation on GPU and the performance of a sequential implementation on a CPU has been made.

Second, previous systematic analysis led to the discovery that a moderate population size under the parallel island model is enough to get to very good fitness values. The excellent efficiency in terms of performance of the chosen model has allowed to compute many of the GA possibilities in around a minute and even finally to reach the optimal minimal fitness value for the studied network. In particular, the best results are obtained by simulating a few thousands of individuals using elitist selection under the island model. Our best implementation reduces the runtime to an order of 80 s for a population of 20 000 individuals, running on a Nvidia GTX660 GPU. Our results also show that the rest of the models can be ranked in terms of efficiency, in descending order, in this way: hybrid model (employing islands which include a cellular model inside each one), cellular model, and master-slave model.

The potential of our study is evident for scientific analysis of the temporal dynamics of other GRNs, which may help to determine the causes of certain diseases or to identify therapeutic targets. In particular, GRNs with a large number of genes would take a prohibitive runtime if implemented sequentially, if non optimized GPU code were used, or if an inadequate GA model were employed. However, GRN dynamics determination would be feasible if implemented in fitted parallel codes on GPU (or on multi-GPU clusters, because parallel island model matches perfectly in them) following the procedures presented in this work. Indeed, a future line of development will be verifying the scalability for multi-GPU clusters in terms of the number of GPUs for the different GA models.

Author Contributions

RGC, JLG, FDR, and AC conceived and designed the experiments. RGC implemented the programs and carried out the experiments. RGC, JLG, FDR, AC, and FJM analyzed the data. RGC, JLG, and FDR wrote the first draft of the manuscript. AC and FJM contributed to the writing of the manuscript. RGC, JLG, FDR, AC, and FJM agreed with manuscript results and conclusions. RGC, JLG, and FDR jointly developed the structure and arguments for the paper. RGC, JLG, FDR, AC, and FJM made critical revisions and approved final version. All authors reviewed and approved the final manuscript.

References

1. Buchanan M, Caldarelli G, Los Rios PD, Rao F, Vendruscolo M, eds. *Networks in Cell Biology*. Cambridge, UK: Cambridge University Press; 2010.
2. Bolouri H. *Computational Modeling of Gene Regulatory Networks: A Primer*. London, England: Imperial College Press; 2008.
3. Aguilar-Hidalgo D, Córdoba Zurita A, Lemos Fernández MC. Complex networks evolutionary dynamics using genetic algorithms. *Int J Bifurcat Chaos*. 2012;22:1250156.

4. Aguilar-Hidalgo D, Lemos M, Córdoba A. Evolutionary dynamics in gene networks and inference algorithms. *Computation*. 2015;3:99–113.
5. Bäck T, Fogel DB, Michalewicz Z. *Handbook of Evolutionary Computation*. Bristol, UK: IOP Publishing; 1997.
6. Simon D. *Evolutionary Optimization Algorithms*. Hoboken, NJ: Wiley; 2013.
7. Goldberg DE. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley; 1989.
8. Mitchell M. *An Introduction to Genetic Algorithms*. Cambridge, MA: The MIT Press; 1996.
9. Keedwell E, Narayanan A. Discovering gene networks with a neural-genetic hybrid. *IEEE/ACM Trans Comput Biol Bioinform*. 2005;2:231–242.
10. Lee WP, Tzou WS. Computational methods for discovering gene networks from expression data. *Brief Bioinform*. 2009;10:408–423.
11. Gonzalez-Álvarez DL, Vega-Rodríguez MA, Gómez-Pulido JA, Sanchez-Pérez JM. Predicting DNA motifs by using evolutionary multiobjective optimization. *IEEE T Syst Man Cy C*. 2012;42:913–925.
12. Fan Y, Wu W, Yang J, Liu R. An algorithm for motif discovery with iteration on lengths of motifs. *IEEE/ACM Trans Comput Biol Bioinform*. 2015;12:136–141.
13. Chaves-González JM, Vega-Rodríguez MA. DNA strand generation for DNA computing by using a multi-objective differential evolution algorithm. *Biosystems*. 2014;116:49–64.
14. Alba E, Troya JM. A survey of parallel distributed genetic algorithms. *Complexity*. 1999;4:31–52.
15. Cantú-Paz E. *Efficient and Accurate Parallel Genetic Algorithms*. Dordrecht, The Netherlands: Kluwer Academic Publishers; 2000.
16. Gong YJ, Chen WN, Zhan ZH, et al. Distributed evolutionary algorithms and their models: a survey of the state-of-the-art. *Appl Soft Comput*. 2015;34:286–300.
17. González DL, De Vega FF. On the intrinsic fault-tolerance nature of Parallel Genetic Programming. In: Proceedings of the 15th EUROMICRO International Conference on Parallel, Distributed and Network-based Processing; February 7-9, 2007:450–456; Naples. New York, NY: IEEE.
18. Hidalgo JI, Lanchares J, Fernández de Vega F, et al. Is the island model fault tolerant? In: Proceedings of the 2007 GECCO Conference Companion on Genetic and Evolutionary Computation; July 7-11, 2007:27–37; London, England. New York, NY: ACM.
19. Luo Z, Liu H. Cellular genetic algorithms and local search for 3-SAT problem on graphic hardware. In: Proceedings of the IEEE International Conference on Evolutionary Computation; July 16-21, 2006:2988–2992; Vancouver, BC, Canada. New York, NY: IEEE.
20. Li JM, Wang XJ, He RS, Chi Z-X. An efficient fine-grained parallel genetic algorithm based on GPU-accelerated. In: Proceedings of the IFIP International Conference on Network and Parallel Computing Workshops; September 18-21, 2007:857–864; Liaoning, China. New York, NY: IEEE.
21. Luong TV, Melab N, Talbi EG. GPU-based island model for evolutionary algorithms. In: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation; July 7-11, 2010:1089–1096; Portland, OR. New York, NY: IEEE.
22. Ben-Shalom R, Aviv A, Razon B, Korngreen A. Optimizing ion channel models using a parallel genetic algorithm on graphical processors. *J Neurosci Methods*. 2012;206:183–194.
23. Arenas MG, Romero G, Mora AM, Castillo PA, Merelo JJ. *GPU Parallel Computation in Bioinspired Algorithms: A Review*. Berlin, Germany: Springer; 2012: 113–134.
24. Jin Y, Hallinan J. Evolving gene regulatory networks. *Biosystems*. 2009;98: 2008–2009.
25. Esmaeili A, Jacob C. A multi-objective differential evolutionary approach toward more stable gene regulatory networks. *Biosystems*. 2009;98:127–136.
26. Nicolau M, Schoenauer M. On the evolution of scale-free topologies with a gene regulatory network model. *Biosystems*. 2009;98:137–148.
27. Noman N, Monjo T, Moscato P, Iba H. Evolving robust gene regulatory networks. *PLoS ONE*. 2015;10:1–22.
28. Eggenberger P. Evolving morphologies of simulated 3d organisms based on differential gene expression. In: Fourth European Conference on Artificial Life, 28–31 July 1997:205–213. Brighton, UK: The MIT Press.
29. Banzhaf W. On the dynamics of an artificial regulatory network. In: Advances in Artificial Life: European Conference on Artificial Life; September 14-17, 2003:217–227; Dortmund, Germany. New York, NY: Springer.
30. Knabe J, Nehaniv CL, Schilstra M, et al. Evolving biological clocks using genetic regulatory networks. In: Artificial Life X: Proceedings of the Tenth International Conference on the Simulation and Synthesis of Living Systems; 3–6 June 2006. Bloomington, IN: The MIT Press.
31. Guo H, Meng Y, Jin Y. A cellular mechanism for multi-robot construction via evolutionary multi-objective optimization of a gene regulatory network. *Biosystems*. 2009;98:193–203.
32. Joachimczak M, Wróbel B. Processing signals with evolving artificial gene regulatory networks. In: Artificial Life XII: Proceedings of the Twelfth International Conference on the Synthesis and Simulation of Living Systems; 19–23 August 2010:203–212. Odense, Denmark.
33. Cussat-Blanc S, Harrington K, Pollack J. Gene regulatory network evolution through augmenting topologies. *IEEE T Evolut Comput*. 2015;19:823–837.
34. de Jong H. Modeling and simulation of genetic regulatory systems: a literature review. *J Comput Biol*. 2002;9:67–103.
35. Leclerc RD. Survival of the sparsest: robust gene networks are parsimonious. *Mol Syst Biol*. 2008;4:213.
36. Keckler SW, Dally WJ, Khailany B, Garland M, Glasco D. GPUs and the future of parallel computing. *IEEE Micro*. 2011;31:7–17.
37. Strohmaier E, Dongarra J, Simon H, et al. TOP500 supercomputing sites. http://www.top500.org/
38. Halfhill TR. Parallel processing with CUDA. *Microprocessor Report*. 28 January 2008:1–8.
39. Nvidia. CUDA toolkit documentation. *Technical Report*. http://docs.nvidia.com/cuda/. Published 2015.
40. Luque G, Alba E. *Parallel Genetic Algorithms: Theory and Real World Applications* (Vol 367). Berlin, Germany: Springer; 2011.
41. Krömer P, Platoš J, Snášel V. Nature-inspired meta-heuristics on modern GPUs: state of the art and brief survey of selected algorithms. *Int J Parallel Prog*. 2014;42:681–709.
42. Cecilia JM, Nisbet A, Amos M, García JM, Ujaldón M. Enhancing GPU parallelism in nature-inspired algorithms. *J Supercomput*. 2013;63:773–789.
43. Wahib M, Munawar A, Munetomo M, Akama K. Optimization of parallel Genetic Algorithms for nVidia GPUs. In: Proceedings of the 2011 IEEE Congress of Evolutionary Computation (CEC); June 5-8, 2011:803–811; New Orleans, LA. New York, NY: IEEE.
44. Sonnenburg S, Braun ML, Ong CS, et al. The need for open source software in machine learning. *J Mach Learn Res*. 2007;8:2443–2466.
45. Nvidia. NVIDIA visual profiler. https://developer.nvidia.com/nvidia-visual-profiler. Published 2016.
46. Jaros J. Multi-GPU island-based genetic algorithm for solving the knapsack problem. In: Proceedings of the 2012 IEEE Congress on Evolutionary Computation; June 10-15, 2012:1–8; Brisbane, QLD, Australia. New York, NY: IEEE.
47. Nvidia. NVIDIA thrust library. https://developer.nvidia.com/thrust. Published 2016.
48. Nvidia. CUDA toolkit 4.2 CURAND guide. *Technical Report*. https://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CURAND_Library.pdf. Published 2012.
49. Lässig J, Sudholt D. Design and analysis of migration in parallel evolutionary algorithms. *Soft Comput*. 2013;17:1121–1144.
50. Muelas S, Peña JM, Robles V, et al. Machine learning to analyze migration parameters in parallel genetic algorithms. *Adv Soft Comp*. 2007;44:199–206.