Tesis de Doctorado en Informática

Systolic Genetic Search, A Parallel Metaheuristic for GPUs

Martín Pedemonte

Director de Tesis: Prof. Enrique Alba Torres
Co-Director de Tesis: Prof. Francisco Luna Valero
Director Académico: Prof. Héctor Cancela

PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Setiembre 2017

“Life is what happens to you,
while you’re busy making other plans”
John Lennon, *Beautiful Boy (Darling Boy)*

To Juan Pablo and Isabel, two by-products of this thesis.

Agradecimientos

Finalmente ha llegado el día y el Doctorado está terminando. Este momento nunca hubiera sido posible sin la colaboración de un conjunto de personas, que me apoyaron a lo largo de este proceso, a las que quiero expresar mi agradecimiento.

En primer lugar, quiero agradecer a mis directores de Tesis Enrique y Paco por su guía académica, su aporte a este trabajo, y la confianza en mí y en mi trabajo. También agradezco a Héctor, quien fue mi director académico, por su ayuda y su estímulo.

A Pablo y Ernesto por aportarme otras perspectivas, por los intercambios de ideas y por sus valiosos comentarios que contribuyeron con este trabajo. A Eduardo por sus sugerencias e ideas para el diseño de una nueva grilla.

Agradezco a mi compañera Leonella por su apoyo incondicional y su paciencia infinita, y también por haberme facilitado las cosas para que pudiera dedicarle tiempo a esta tesis. También agradezco a mis hijos Juan Pablo e Isabel por todos los ratos que les tuve que robar. A mi madre Graciela y a mi hermano Gerardo por su continuo aliento y por su cariño.

También agradezco a mis compañeros de la sala 048 del INCO por el buen ambiente de trabajo que hemos creado y a los integrantes del grupo NEO de la UMA por el buen rollo y por tratarme como uno más.

Por último agradezco a aquellas instituciones que brindaron soporte económico para realizar tareas en el marco de esta investigación. En particular, a la Agencia Nacional de Investigación e Innovación que a través del fondo María Viñas financió el proyecto “Metaheurísticas paralelas en GPU”, y a la RED CYTED - CADING (Computación de Alto Desempeño en Ingeniería), al PEDECIBA Informática y a CSIC que financiaron parcialmente pasantías, asistencias a congresos y compras de equipamiento.

Abstract

The use of Graphics Processing Units (GPUs) for general purpose computing has experienced a tremendous growth in recent years, based on its wide availability, low economic cost and inherent parallel architecture, as well as on the emergence of general purpose programming languages that have eased the development of applications onto this parallel computing platforms. In this context, the design of new parallel algorithms that profit from the GPUs platform is certainly a promising and interesting research line.

Metaheuristics are stochastic algorithms which are able to provide optimization problems with very accurate (many times, optimal) solutions in a reasonable amount of time. However, as many optimization problems involve tasks demanding large computational resources, and/or problem instances in today's research are becoming very large, even metaheuristics may be highly computationally expensive. In this situation, parallelism comes out as a successful strategy for speeding up the search of those kind of algorithms. Parallel metaheuristics do not only allow to reduce the runtime of the algorithms, but also often improve the quality of the results obtained by traditional sequential algorithms.

Although the use of GPUs has also represented an inspiring domain for the research in parallel metaheuristics, most previous works were aimed at porting existing family of algorithms onto this new kind of hardware. As a consequence, many published material is targeted to show the time savings of running different parallel types of metaheuristics on GPUs. In spite of this considerable work on the parallelization of metaheuristics in GPUs, there are few novel ideas for designing new algorithms and/or parallel models that explicitly exploit the high degree of parallelism available in modern GPU architectures.

This thesis addresses the design of an innovative proposal of a parallel optimization family of algorithms, called Systolic Genetic Search (SGS), that merges ideas from the fields of metaheuristics and systolic computing. SGS, as well as systolic computing, are inspired by the same biological phenomenon: the systolic contraction of the heart that makes possible blood circulation. In SGS, solutions circulate synchronously through a grid of processing cells. When two solutions meet in a cell, adapted evolutionary operators are applied to generate new solutions that continue moving through the grid. The implementation of this new proposal takes special advantage of the specific features of the massively parallel GPU platforms.

An extensive experimental analysis, which considers several classical benchmark problems and two real-world problems from the Software Engineering field, shows that the newly proposed algorithm is highly effective, finding optimal or near optimal solutions in short execution times. Moreover, the numerical results obtained by SGS are competitive with the state-of-the-art results for the two real-world problems also targeted in this PhD thesis. On the other hand, the parallel GPU implementation of SGS has achieved a high performance, obtaining a large runtime reduction from the sequential implementation and showing that it scales properly on instances of increasing size.

A theoretical analysis of the search capabilities of SGS has been also performed for understanding how some aspects of the behaviour of the algorithm affect the numerical results of SGS. This analysis gives an important insight in the behavior of SGS that can be used to improve the design of future variants of the algorithm.

Keywords: Evolutionary Algorithms, Systolic Computing, Parallel Metaheuristics, GPU, Search-Based Software Engineering.

Resumen

La utilización de unidades de procesamiento gráfico (GPUs) para la resolución de problemas de propósito general ha experimentado un crecimiento vertiginoso en los últimos años, sustentado en su amplia disponibilidad, su bajo costo económico y en contar con una arquitectura inherentemente paralela, así como en la aparición de lenguajes de programación de propósito general que han facilitado el desarrollo de aplicaciones en estas plataformas. En este contexto, el diseño de nuevos algoritmos paralelos que puedan beneficiarse del uso de GPUs es una línea de investigación prometedora e interesante.

Las metaheurísticas son algoritmos estocásticos capaces de encontrar soluciones muy precisas (muchas veces óptimas) a problemas de optimización en un tiempo razonable. Sin embargo, como muchos problemas de optimización involucran tareas que exigen grandes recursos computacionales y/o el tamaño de las instancias que se están abordando actualmente se están volviendo muy grandes, incluso las metaheurísticas pueden ser computacionalmente muy costosas. En este escenario, el paralelismo surge como una alternativa exitosa con el fin de acelerar la búsqueda de este tipo de algoritmos. Además de permitir reducir el tiempo de ejecución de los algoritmos, las metaheurísticas paralelas a menudo son capaces de mejorar la calidad de los resultados obtenidos por los algoritmos secuenciales tradicionales.

Si bien el uso de GPUs ha representado un dominio inspirador también para la investigación en metaheurísticas paralelas, la mayoría de los trabajos previos tenían como objetivo portar una familia existente de algoritmos a este nuevo tipo de hardware. Como consecuencia, muchas publicaciones están dirigidas a mostrar el ahorro en tiempo de ejecución que se puede lograr al ejecutar los diferentes tipos paralelos de metaheurísticas existentes en GPU. En otras palabras, a pesar de que existe un volumen considerable de trabajo sobre este tópico, se han propuesto pocas ideas novedosas que busquen diseñar nuevos algoritmos y/o modelos de paralelismo que exploten explícitamente el alto grado de paralelismo disponible en las arquitecturas de las GPUs.

Esta tesis aborda el diseño de una propuesta innovadora de algoritmo de optimización paralelo denominada Búsqueda Genética Sistólica (SGS), que combina ideas de los campos de metaheurísticas y computación sistólica. SGS, así como la computación sistólica, se inspiran en el mismo fenómeno biológico: la contracción sistólica del corazón que hace posible la circulación de la sangre. En SGS, las soluciones circulan de forma síncrona a través de una grilla (rejilla) de celdas. Cuando dos soluciones se encuentran en una celda se aplican operadores evolutivos adaptados para generar nuevas soluciones que continúan moviéndose a través de la grilla (rejilla). La implementación de esta nueva propuesta saca partido especialmente de las características específicas de las GPUs.

Un extenso análisis experimental que considera varios problemas de benchmark clásicos y dos problemas del mundo real del área de Ingeniería de Software, muestra que el nuevo algoritmo propuesto es muy efectivo, encontrando soluciones óptimas o casi óptimas en tiempos de ejecución cortos. Además, los resultados numéricos obtenidos por SGS son competitivos con los resultados del estado del arte para los dos problemas del mundo real en cuestión. Por otro lado, la implementación paralela en GPU de SGS ha logrado un alto rendimiento, obteniendo grandes reducciones de tiempo de ejecución con respecto a la implementación secuencial y mostrando que escala adecuadamente cuando se consideran instancias de tamaño creciente.

También se ha realizado un análisis teórico de las capacidades de búsqueda de SGS para comprender cómo algunos aspectos del diseño del algoritmo afectan a sus resultados numéricos. Este análisis arroja luz sobre algunos aspectos del funcionamiento de SGS que pueden utilizarse para mejorar el diseño del algoritmo en futuras variantes.

Palabras clave: Algoritmos Evolutivos, Computación Sistólica, Metaheurísticas Paralelas, GPU, Ingeniería de Software Basada en Búsqueda.

Table of Contents

List of Figures

List of Tables

Nomenclature

Acronyms / Abbreviations

ACO	Ant Colony Optimization
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ARM	Advanced RISC Machine
COW	Clusters Of Workstations
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DPX	Double Point Crossover
EA	Evolutionary Algorithm
ES	Evolution Strategies
FPGA	Field-Programmable Gate Array
GA	Genetic Algorithm
GPC	Graphics Processing Cluster
GP	Genetic Programming
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphic Processing Unit
GRASP	Greedy Randomized Adaptive Search Procedure

HPC	High Performance Computing
ILS	Iterated Local Search
KP	Knapsack Problem
MIMD	Multiple Instruction streams - Multiple Data streams
MISD	Multiple Instruction streams - Single Data stream
MMDP	Massively Multimodal Deceptive Problem
MPMD	Multiple Programs - Multiple Data streams
MPP	Massively Parallel Processors
NP	Non-deterministic Polynomial-time
NRP	Next Release Problem
PEA	Parallel Evolutionary Algorithm
PGP	Pipelined Genetic Propagation
PSO	Particle Swarm Optimization
SA	Simulated Annealing
SBSE	Search-Based Software Engineering
SGS	Systolic Genetic Search
SIMD	Single Instruction stream - Multiple Data streams
SIMT	Single-Instruction Multiple-Threads
SISD	Single Instruction stream - Single Data stream
SM	Streaming Multiprocessor
SNS	Systolic Neighborhood Search
SPMD	Single Program - Multiple Data streams
SPX	Single Point Crossover
TSMP	Test Suite Minimization Problem

TS	Tabu Search
VLSI	Very-Large-Scale Integration
VNS	Variable Neighborhood Search

Chapter 1

Introduction

1.1 Motivation

One of the most important open questions in computation theory is whether the complexity class P is equal to complexity class NP. The complexity class P is the set of decision problems that can be solved in polynomial time by a deterministic Turing machine, while the complexity class NP contains the problems that can be solved in polynomial time by a non-deterministic Turing machine [6]. Equivalently, the NP class can be conceptualized as the set of problems verifiable in polynomial time, i.e., any given instance of the problem can be checked if it is a solution to the problem or not in polynomial time. In other words, P problems can be solved efficiently, while NP problems can be verifiable efficiently.

From these definitions, it is possible to conclude that P $\subseteq$ NP, but the question "Is P = NP?" has not been answered in more than 40 years [24]. P $\neq$ NP is nowadays the most important conjecture of the theory of computation. This conjecture has many practical implications for solving optimization problems, since many optimization problems belong to the NP-hard class. A problem is NP-hard when all NP problems can be reduced to it in polynomial time. That is, NP-hard problems are as hard as any problem in NP.

When a problem belongs to the class P, i.e., there is a polynomial time algorithm for its resolution, it is considered that it can be efficiently solved in practice. On the other hand, when a problem belongs to the NP class, and therefore there is no such algorithm, it is considered that the problem can not be efficiently resolved when instances of increasing size are used. As a consequence, the resolution of NP-hard optimization problems [39] in reasonable times is limited in practice due to their high computational complexity.

In this scenario, metaheuristic methods have emerged as an alternative since, although they do not guarantee to obtain an optimal solution in many cases, they are able to find very accurate solutions efficiently. Metaheuristics [14, 43, 111] are general schemes of heuristics that make it able to address a wide range of problems, adapting to the particularities of each one. Evolutionary Algorithms (EAs) are one of the most popular types of metaheuristics. EAs explore the search space evaluating candidate solutions of the optimization problem at hand. During the exploration, EAs generate new candidate solutions in the decision space. This process is guided by a metric related to the objective space, which is known as the *fitness* function.

Research in metaheuristics is nowadays consolidated, motivated by the excellent results obtained in their application to the resolution of problems in search, optimization, and machine learning [46, 111]. However, as the problem instances in today's research are becoming very large, even metaheuristics may be highly computationally expensive.

The design of parallel metaheuristics [1], due to the increase on the capabilities of modern hardware architectures, is a natural line of research with the goal of substantially reducing the runtime of the algorithms, specially when solving problems of a high dimension or severely restricted, or when the time available for solving the problem is very limited.

Parallel metaheuristics often exploit new search patterns that are different from traditional sequential algorithms. The truly interesting point is that parallel metaheuristics not only speed up the runtime of the algorithms, but also allow to improve the quality of results obtained by traditional sequential algorithms due to their enhanced search engine [1]. As a consequence, research in parallel metaheuristics has substantially grown in the last two decades.

In the last ten years, computing platforms have undergone revolutionary changes [52]. Parallel hardware is no longer an infrastructure reserved for a few research laboratories, but it is widely available for the general public. On the one hand, the architecture of CPU processors has changed, and are now multi-core, i.e., a single computing unit composed of at least two independent processors. As a consequence, modern desktop computers are currently quad-core or octa-core. On the other hand, the parallel hardware that can be used for computation has diversified notably. Nowadays, it is possible to use parallel computing devices like multi-core processors with ARM architecture [38] for general purpose, which have become massively available in smart cell phones and tablet computers, as well as hardware accelerators, like Graphics Processing Units (GPUs) [40, 63] and Xeon Phi processors [57]. As a consequence, the design of parallel algorithms able to profit from the new capabilities available in modern hardware is certainly indispensable.

It is clear that the computing platform used for execution is a crucial aspect in the implementation of parallel algorithms in general and of parallel metaheuristics in particular. The hardware platforms that are traditionally available, such as supercomputers or clusters with a large number of CPUs, imply a very high economic cost caused by the equipment cost, the building cost associated to the equipment installation and the administration, and maintenance cost of such infrastructure. A more cost-effective alternative is represented by multi-core processors but they only have a relatively small number of processing units. In contrast, GPUs are today an extremely appealing option for their computing power, as well as for their economic cost and energy consumption, specially in countries like Uruguay, where the access to large computing infrastructures is restricted due to economic reasons.

General-purpose computing on graphics processing units, which is known as GPGPU, has experienced a tremendous growth in recent years, based on its wide availability, low economic cost, low energy consumption and inherent parallel architecture, and also on the emergence of general purpose programming languages, like CUDA [92] and OpenCL [40]. This fact has also motivated many scientists from different fields to take advantage of the use of GPUs in order to tackle general problems in various fields [93] like numerical linear algebra [33], databases [17], model order reduction [9], scientific computing [108], etc.

The use of GPUs have also represented an inspiring domain for the research in parallel metaheuristics. As might be expected, the first works on this subject have gone in the direction of taking a standard existing family of algorithms and porting them to this new kind of hardware [66, 71]. Thus, many results show the time savings of the implementation of master-slave [78], island [122, 123], and cellular [109, 114] models of parallel metaheuristics on GPU. The metaheuristics that have been mainly used for such implementation on GPUs are Genetic Programming [50, 71–73, 79] and Genetic Algorithms [78, 123], but also other types of techniques like Ant Colony Optimization [21, 22, 112], Differential Evolution [113], Particle Swarm Optimization [124], etc.

In spite of this considerable work in the use of GPUs for reducing the execution time of metaheuristics, the changes in modern hardware like the GPUs, which are massively parallel and have thousands of cores, were not matched by changes in the conception of the development of parallel metaheuristics. As a consequence, there are few innovative proposals for the design of new algorithms and/or parallelism models that exploit the particular features and massive parallelism offered by the architecture of the GPU.

GPUs can be seen as a revival of the concept of SIMD machines, opening the door to a vast existing knowledge on massively parallel and vectorial computers ready to be used to create new generations of unseen numerical algorithms. And this merits a different approach, consisting in creating new fresh algorithms targeted to this architecture. For this reason, the course of action followed in this work is different, and consists in using *Systolic Computing* as a source of inspiration for designing a new optimization algorithm.

The concept *Systolic Computing* was developed at Carnegie-Mellon University [67, 69]. The basic idea focuses on creating a network of different simple processors or operations that rhythmically compute and pass data through the system. Systolic computation offers several advantages, including simplicity, modularity, and repeatability of operations. This kind of architecture also offers transparent, understandable and manageable, but still quite powerful parallelism. However, this architecture had difficulties in the past, since building systolic computers was not easy and, especially, because programming high level algorithms on such a low-level hardware was hard, error prone, and too manual. Now, the behavior of systolic computing can be mimicked by software on the GPUs, avoiding such problems and getting only the advantages.

1.2 Objectives

This thesis deals with the study of the use of GPUs as a general-purpose computing platform, in particular for the implementation of parallel metaheuristics applied to the resolution of optimization problems. The main goal of this thesis lies in designing and proposing new parallel metaheuristic techniques and/or new parallel models that explicitly exploit the high degree of parallelism available in modern GPU architectures. The objective is to develop algorithms that can profit from the characteristics of the GPU architecture and that they are both effective and efficient, i.e., that are able to obtain near optimal solutions in short execution times.

To achieve the main purpose of this thesis, the work is divided into several specific objectives that are summarized as follows:

Design of an innovative proposal of parallel metaheuristics on GPU for solving optimization problems: The source of inspiration for designing a new optimization algorithm comes from systolic computing. In particular, the new optimization algorithm proposed combines the use of a systolic architecture, in which the solutions circulate through the cells, with the application of adapted evolutionary operators.

Develop an efficient implementation on GPU of the newly proposed optimization algorithm: The implementation of the new proposal has to take special advantage of the specific features of the massively parallel GPU platform.

Validation of the optimization algorithm proposed: In order to validate the new optimization algorithm, an experimental evaluation has been conducted on classical benchmark problems to asses both the search capabilities of the algorithm and the parallel performance of the algorithm when deployed on a GPU card. Since GPUs architectures have significant changes in each generation, it is desirable that the experimental evaluation involves GPUs from different generations and with different features.

In-depth study of one real-word problem: As part of this work, a real-world problem has been tackled in-depth as an application case of the newly proposed algorithm. In particular, the use of metaheuristics for solving optimization problems that arise in the Software Engineering field, which is known as Search-Based Software Engineering (SBSE) [51], has gained great interest among researchers. For this reason, it is attractive to address a real-world problem from this field.

Study of theoretical properties of the newly proposed optimization algorithm: Besides the experimental evaluation, it is also important to understand how the features of the algorithm affect the numerical results obtained by the new optimization algorithm. For this reason, a theoretical study has been conducted for analyzing some aspects of the behaviour of the algorithm if this PhD is to be a solid contribution to the field.

In order to fulfill these objectives, the following stages have been followed. In the first place, the features of the GPUs hardware platforms have been studied, as well as the state-of-the-art in parallel metaheuristics implementation on GPUs, and in the implementation of evolutionary algorithms using systolic computing devices. From this survey, the design of a new algorithm is made, combining ideas from systolic computing and evolutionary computation. The new algorithm is named Systolic Genetic Search (SGS) since it uses the solution representation and the evolutionary operators of the genetic algorithms. A direct antecedent of SGS is Systolic Neighborhood Search (SNS) [115, 116], which also uses a systolic architecture but with a more simple search strategy than SGS. Then, a first empirical analysis of the proposed algorithm is conducted using as a testbed two classical benchmark problems and a real-world problem from the field of software engineering. From the feedback of the first experimental evaluation, the original design of the new algorithm is expected to be adjusted.

An analysis of relevant real-world problems from the field of software engineering is conducted in order to identify and select an application case of the newly proposed algorithm. This problem is used for a second experimental evaluation of SGS. This evaluation includes the experimental comparison with algorithms previously proposed in the literature that are the state-of-the-art for the problem under consideration. Finally, since the behavior of the new algorithm proposed has been studied only empirically, a particular aspect of the algorithm (the flow of solutions) is theoretically and experimentally analyzed in detail, yielding valuable lessons that will undoubtedly be useful to consolidate the proposed algorithm.

1.3 Contributions

The contributions of this thesis are related to the study of the possibilities offered by new hardware platforms, like the GPUs, in order to design new algorithms that are both effective and efficient for the resolution of optimization problems. These contributions can be summarized in the following points:

- Design of the new parallel metaheuristic algorithm *Systolic Genetic Search*, specially tailored for GPU platforms, for solving optimization problems based on hybridizing ideas of genetic algorithms and systolic computing.
- Efficient GPU implementation of SGS able to both reduce the runtime of the algorithm significantly and scale properly on instances of increasing size.
- Exhaustive and detailed analysis of the numerical efficiency and parallel performance of SGS in both benchmark and real-world problems.
- Application of SGS to address two real-world problems from the field of Software Engineering, namely the Next Release Problem [8] and the Test Suite Minimization Problem [121], with competitive results to the state-of-the-art.
- Theoretical analysis of the search capabilities of SGS based on the trajectories described by the solutions on the grid. This analysis gives an important insight into the behaviour of the algorithm that can be used in future works to improve the design of the algorithm.

1.4 Structure of the Document

This PhD thesis is structured as a compendium of articles instead of a traditional monograph. The thesis is supported by three articles that have been published in international journals. This document is divided into two parts. The first part, which is composed by six chapters, provides general background on the subject of this thesis, introduces the main contributions of this work, a summary of each one of the three articles, and the bibliography. The second part is composed by three appendixes that correspond to the three articles supporting this PhD thesis.

A brief introduction to each chapter is provided next:

Chapter 1 serves as introduction to the subject of the thesis. It provides the motivation and the objectives of the thesis and the outline of the document.

Chapter 2 gives background knowledge on computing platforms that it is indispensable for understanding the algorithm proposed in this thesis. It describes the fundamental ideas of Systolic Computing, which is the source of inspiration for Systolic Genetic Search, and the architecture of the GPUs, which is the platform for which the proposed algorithm is conceived.

Chapter 3 provides elementary notions of metaheuristics that support the rest of this work. It describes Evolutionary Algorithms, and Genetic Algorithms in particular, which are closely related to the proposal of Systolic Genetic Search. It also presents the parallelization strategies used for EAs and the population models that have emerged as a result of these strategies.

Chapter 4 serves as a presentation of the Systolic Genetic Search algorithm proposed in this thesis. It presents the fundamental aspects of the algorithm, as well as, the description of its GPU implementation. The chapter also includes a brief discussion of related work with similar ideas to the algorithm proposed and the methodology followed for analyzing the experimental results of the algorithm.

Chapter 5 summarizes each one of the three articles that are the main outcomes of the research of this PhD thesis. In addition to those articles that provide support for this thesis, this chapter also presents other peer-reviewed publications that have also been produced in this work.

Chapter 6 serves as a general conclusion of this thesis and establishes the lines of future research.

Appendix 1 presents the article "Systolic genetic search, a systolic computing-based metaheuristic" that has been published in *Soft Computing* journal. This article sets the foundations of SGS algorithm.

Appendix 2 presents the article "A Systolic Genetic Search for reducing the execution cost of regression testing" that has been published in *Applied Soft Computing* journal. In this article, a real-world problem is tackled with SGS; the results obtained are not only relevant for the SGS but also are relevant for the problem.

Appendix 3 presents the article “A theoretical and empirical study of the trajectories of solutions on the grid of Systolic Genetic Search” that has been published in *Information Sciences* journal. This article provides a theoretical analysis on the trajectories described by the solutions on the grid of SGS that offers a valuable insight on the behavior of the algorithm.

Chapter 2

Systolic Computing and Graphics Processing Units

Parallel computing is a high performance computing (HPC) technique in which several processing units work simultaneously and coordinated, i.e., at the same physical instant, to solve a common problem [80]. In this computational paradigm, a problem is divided into several subtasks that can be computed independently and whose results are combined for obtaining the result of the whole problem.

Since the implementation of parallel algorithms heavily relies on the features of the computing platform used for execution, this chapter is devoted to introducing Graphics Processing Units (GPUs), which are the parallel computing devices that are used in this work. The rest of this chapter is organized as follows. The next section presents a description of existing parallel computer architectures. Then, we introduce Systolic Computing, which is the source of inspiration for the algorithm proposed in this thesis, namely Systolic Genetic Search. Finally, the chapter ends with the description of the architecture of the GPUs.

2.1 Architecture of Parallel Computers

In the last ten years, parallel-capable hardware has turned into the dominant paradigm in computer architecture [7]. Several taxonomies have been proposed to classify parallel computers. The most commonly used was proposed by Flynn [36] and it is characterized by considering independently the instruction and data streams. For both types of streams, the taxonomy distinguishes according to the number of streams available between single and multiple, thus determining four different categories. The characteristics of each category are briefly described below:

- Single Instruction stream - Single Data stream (SISD): This category corresponds to the traditional von Neumann architecture, which consists in a single processing unit that has no parallelism in either the data or the instructions. An example of this architecture is the traditional single uni-core processor that was massively available in older PCs. The SISD architecture is shown in Figure 2.1a.

- Single Instruction stream - Multiple Data streams (SIMD): This category is characterized by having multiple processing units that execute the same instruction on different data at the same instant of time. The processing units have no processing autonomy and are centrally controlled. Originally, because of the constraints caused by synchronism and data distribution, computers with this architecture were typically used for purpose-specific applications [3]. Later, SIMD instructions were incorporated in the instruction set of common PC processors, like Intel's MMX, SSE and AVX instruction set extensions to the x86 architecture. The SIMD architecture is shown in Figure 2.1b.

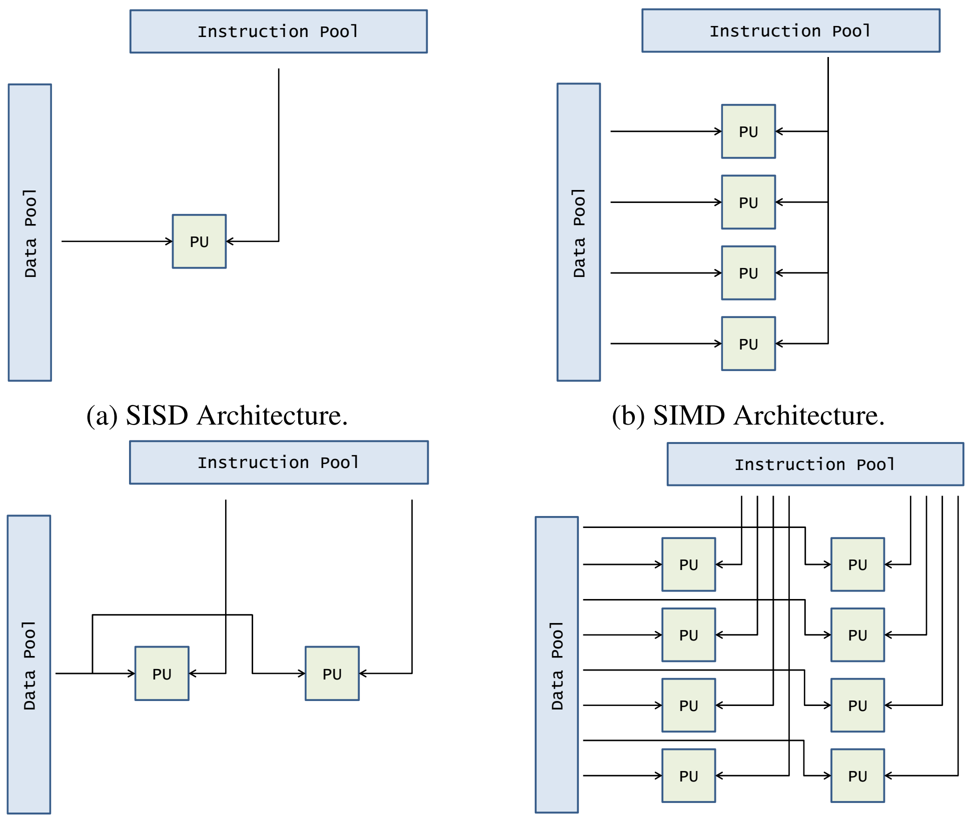
- Multiple Instruction streams - Single Data stream (MISD): This type of architecture executes multiple instructions on a single data stream. It is a parallel architecture very uncommon in practice. It can be used for implementing fault tolerance systems, where each processing unit operates on the same data and the results obtained by each unit have to be the same. In general, systolic arrays are usually classified into this category, although, as it will be seen in Section 2.2, the classification of such devices is unclear. The MISD architecture is shown in Figure 2.1c.

- Multiple Instruction streams - Multiple Data streams (MIMD): In the architectures of this category, several processing units execute different instructions on different data at any instant of time. The processing units are autonomous and the control is completely decentralized. The MIMD architecture is shown in Figure 2.1d.

Several refinements to this classification have been proposed, taking into account aspects that were not included in the original taxonomy.

The SIMD category can be subdivided between array processors and vector processors. In array processors, each operation is performed in a single step over multiple data (the elements of the vectors) at the same time. On the other hand, in vector processors, each operation is performed in several consecutive steps over multiple data. Array processors are based on the use of multiple ALUs, while vector processors use pipeline parallelism, in which each stage of the pipeline works on a different element of the vectors, for implementing the vectorial instructions.

The MIMD category can be subdivided into two subcategories, depending on whether the memory is composed of a single address space or there are several different spaces. In



(a) SISD Architecture.

(b) SIMD Architecture.

(c) MISD Architecture.

(d) MIMD Architecture.

Fig. 2.1 Flynn's Taxonomy.

multiprocessors, all processing units have access to the same memory space, so these devices are also known as shared memory computers. The communication and synchronization of processes is done through the reading and writing of the global memory. On the other hand, in multicomputers, each processing units has a separate memory that can only be accessed by the processor itself, without a global memory to the entire system. Multicomputers are also known as distributed memory machines. In this case, the communication and synchronization of processes is made through the explicit passage of messages between the processing units. Examples of distributed memory include Clusters of Workstations (COW) and Massively Parallel Processors (MPP).

Another alternative for subdividing the MIMD category is to distinguish whether all processing units execute the same program or not. In particular this refinement distinguishes between the Single Program Multiple Data streams (SPMD) and the Multiple Programs Multiple Data streams (MPMD) paradigms. In SPMD, processing units run the same program on multiple parts of the data, but do not have to be executing the same instruction at the same time [25, 26]. On the other hand, in MPMD, the processing units are running at least two independent programs at the same time.

2.2 Systolic Computing

The idea of Systolic Computing emerged in the late 70's [58, 67, 69, 70]. It was conceived as a general methodology for mapping high-level computations into low-level hardware structures, specially for the design of high-performance special-purpose computer systems used for off-loading computations from a general-purpose computer. It is a design style that was proposed for the design and implementation of Very-Large-Scale Integration (VLSI) systems [81], which combine electronic components in an integrated circuit or microchip.

The basic idea of systolic computing consists in organizing the processing units using a systolic architecture. In a systolic architecture, the processing units lie in a network connected in a simple and regular fashion allowing data flow between neighboring units. The network is also connected to the computer memory. Data rhythmically flows from the computer memory through the units before it returns to memory. The units, which are also known as cells, are capable of performing simple operations to data, for instance the inner product of the inputs of the unit [69], that is then passed through the next unit in the topology. The behaviour of systolic computing systems resemble the systolic contraction of the heart that makes possible that ventricles eject blood rhythmically. Each processing unit can be seen as a heart that regularly pumps data in and out, performing a simple operation over the data in order to maintain a regular flow of data [69]. Figure 2.2 shows the architecture of a one dimensional systolic computing system.

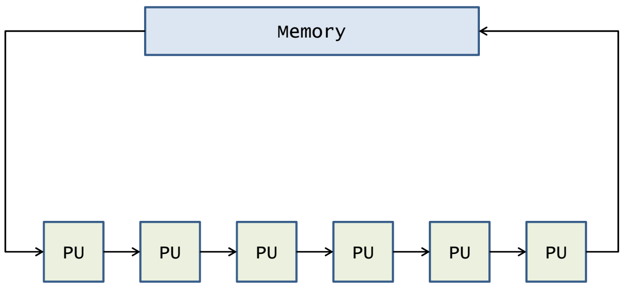
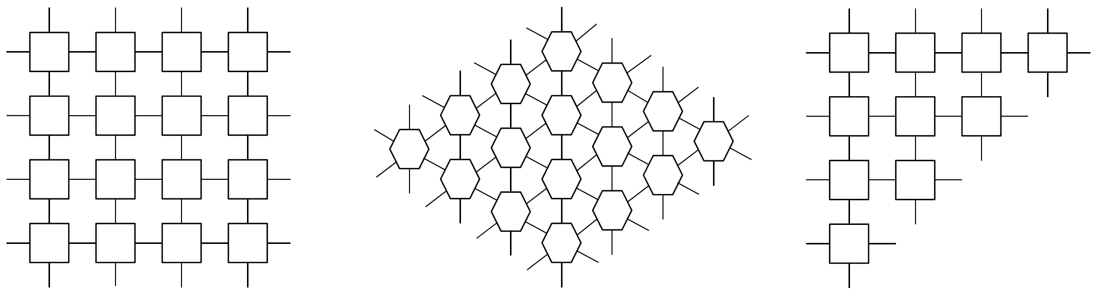


Fig. 2.2 Architecture of a one dimensional systolic system.

The architecture of systolic computation offers several advantages. The advantages include a simple and regular design, local communication between the cells, and the modularity and repeatability of the operations computed by the cells. This architecture offers understandable and manageable, but still quite powerful parallelism through the use of pipelining and multiprocessing. For this reason, the use of systolic architectures have transcended from special-purpose implementations of algorithms on VLSI technology to general-purpose systolic arrays using Field-Programmable Gate Array (FPGA) technology [58].

The one dimensional linearly connected architecture is the most common topology used for systolic arrays. This architecture has been used for implementing, among others, the computation of lineal polynomial GCD [16], the resolution of triangular linear systems [69], and the computation of the Matrix-Vector multiplication [69].

Two-dimensional systolic arrays have also been proposed and extensively used, like the orthogonally connected or *type R* architecture (e.g., used for the calculation of relational database operations [68] and the implementation of dynamic programming algorithms [47] and the transitive closure of graphs [47]), the hexagonally connected or *type H* architecture (e.g., the calculation of the matrix multiplication [69], the LU-decomposition of a band matrix [69], and the calculation of a band matrix-matrix multiplication [117]), and the rectangular grid on a triangle or *type T* architecture (e.g., the solution of dense systems of linear equations [15] and the calculation of matrix triangularization [42]). These architectures are shown in Figure 2.3.



(a) Type R. (b) Type H. (c) Type T.

Fig. 2.3 2-dimensional systolic arrays.

It is a controversial issue in which category of the Flynn's Taxonomy systolic arrays should be classified. In general, systolic arrays tend to be classified as MISD, but it can be argued that they are not part of that category since the processing units are not working on the same data because data is transformed when it circulates through the units. Some authors, like Johnson et al. [58], categorize systolic arrays as SIMD or MIMD. The SIMD systolic arrays are characterized by executing the same instruction on all the processing units on different data [58] but data streams do not maintain their independence as it happens in traditional SIMD architectures. On the other hand, the MIMD systolic arrays execute different programs on each of the cells [58] but they also do not seem to be completely well classified by this category. In any case, none of the categories of the Flynn's Taxonomy captures the interconnection of the data flow between the processing units, which is one of the key aspects of a systolic computing system.

2.3 Graphics Processing Units

Graphics Processing Units (GPUs) are computation devices originally designed for graphics processing. The term GPU was coined by Nvidia in 1999, when the Nvidia GeForce 256 was launched on the market. The GeForce 256 was the first consumer-level video card that implemented the entire graphics pipeline in hardware, i.e., it integrated the transformation and lighting calculations in the hardware of the GPU instead of using the CPU for performing such calculations. In consequence, the workload of the CPU is lightened and the CPU can perform other computations while the graphics processing calculations are performed on the graphic device.

Two important milestones in the design of the architecture of the GPUs should be highlighted. The first one was the replacement of the fixed-function graphics pipeline by programmable vertex shader units (to perform transformations and lighting operations on vertices) and pixel shader units (to determine the final pixels color). The second one was the adoption of the unified shader model in 2005/2006. In the unified shader model, the same set of unified processing units is used for both vertex and pixel computation. This is considered one of the most important breakthroughs in the design of these devices since it included for the first time general-purpose cores in the GPU.

The architecture of GPUs follows a design philosophy that is radically different to CPU. While GPUs are designed with the idea of devoting most of the transistors to computation, in a CPU a large part of the transistors are dedicated to other tasks such as branch prediction, out-of-order execution, etc. In consequence, current GPUs have a large number of small cores and are usually considered many-core devices. The number of cores available in modern GPUs is growing steadily and will undoubtedly continue to do so in the foreseeable future. For instance, Nvidia has recently launched its new generation of GPUs with Pascal microarchitecture [87, 88], with up to 3,584 CUDA cores at 1,480 MHz and a single precision floating point peak performance of 10.61 TFlops (i.e., 10^{12} floating point operations per second) in the GeForce GTX 1080 Ti. As a consequence, the number of threads that recent GPUs can run in parallel is in the order of thousands and is expected to continue growing rapidly; what makes these devices a very powerful platform for implementing massively parallel algorithms.

Regarding the programming capabilities, the progress in the design of GPUs was not initially accompanied by an advance in the software for programming these devices. The shaders had to be written originally in assembly language, so the resulting programs were not portable because there were several languages depending on the GPU models. An alternative was to directly use computer graphics application programming interfaces (APIs), such as OpenGL [61] or DirectX . Hence, this was an important limitation for the development of general-purpose computing programs since the programmer had to do esoteric mappings between data structures of the program and computer graphics datatypes. For this reason, different higher level programming languages were developed to solve this problem, like NVIDIA's Cg [35], as well as languages following the stream processing computer programming paradigm [59], such as Brook [18, 19]. However, each programming language was still highly dependent on the GPU architecture, model, etc.

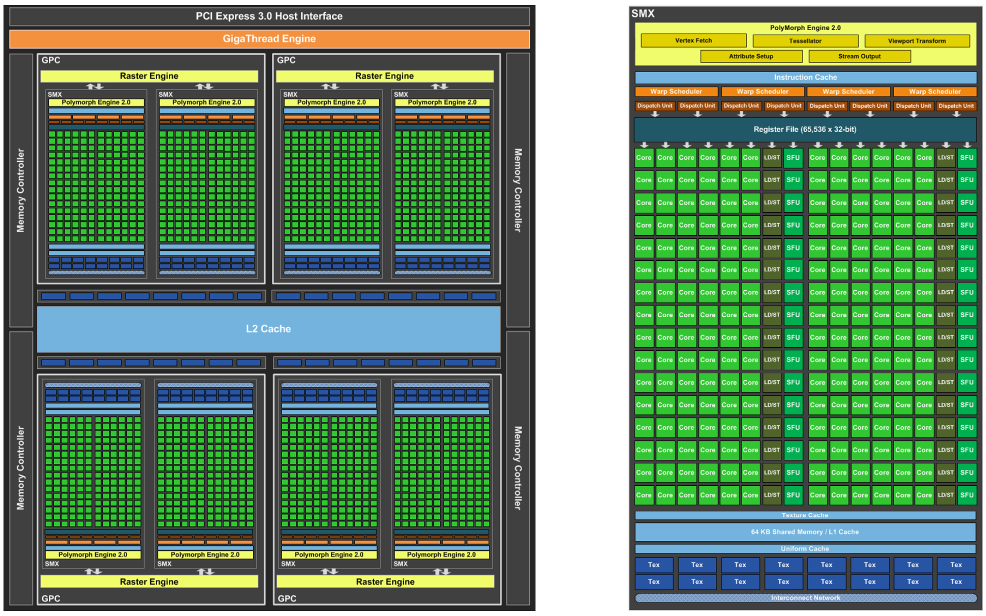
Finally, Compute Unified Device Architecture (CUDA) [63, 92] and OpenCL [62] emerged as general-purpose programming languages in order to solve these drawbacks. CUDA is the general framework that enables to use Nvidia's GPUs for general-purpose computing, while OpenCL is an open standard proposed by the Khronos Group for supporting programming on heterogeneous platforms including CPUs, GPUs and FPGAs, among others. These languages have enabled to unleash the power of GPUs for a wide range of users, including researchers. They are a key aspect of the widespread adoption of these devices for general-purpose computing.

Since in this work we use Nvidia's GPUs, the next section describes the main features of CUDA. OpenCL is rather similar to CUDA and the concepts, which are explained in the next section about CUDA, can be mapped almost directly to OpenCL.

2.3.1 CUDA (Compute Unified Device Architecture)

CUDA is a general-purpose parallel computing platform and programming model [92]. The cores of CUDA-enabled GPUs are organized in Streaming Multiprocessors (SMs) that are grouped in Graphics Processing Clusters (GPCs). Figure 2.4a shows the block diagram of the architecture of the GeForce GTX 680. This GPU has four GPCs, eight SMs, four global memory controllers, and a 512KB L2 cache. Figure 2.4b presents the structure of each SM of the GeForce GTX 680. Each SM is composed of 192 cores, which are usually known as CUDA cores, totalling 1,536 cores in the whole GPU. The SMs also include four warp schedulers, which are responsible of dispatching up to two instructions to the cores on every clock cycle, and an on-chip memory that could be used as a L1 cache for global memory.

CUDA abstracts the GPU as a set of shared memory multiprocessors (MPs) that are able to execute a large number of threads in parallel. Each MP follows the SIMT (Single-Instruction Multiple-Threads) parallel programming paradigm [63, 80, 92]. SIMT is similar to SIMD but in addition to data-level parallelism (when threads are coherent and are executing the same



(a) Block Diagram (taken from [86]). (b) SM (from [86]).

Fig. 2.4 GeForce GTX 680 Architecture.

instruction) allows thread-level parallelism (when threads are divergent and have to execute different instructions). The thread-level parallelism is transparently achieved by CUDA using masking and executing the different instructions in different clock cycles [80, 92].

CUDA provides an extension of the C/C++ programming language that acts as interface to the CUDA parallel computing platform. CUDA allows to define C/C++ functions, called *kernels*, that could be run in parallel on the GPU. When a kernel is called in CUDA, a large number of threads are generated on the GPU. The group of all the threads generated by a kernel invocation is called a *grid*, which is partitioned into *blocks*. The threads from a block are executed concurrently on a single multiprocessor. There is no fixed order of execution between blocks. If there are enough multiprocessors available on the GPU, they are executed in parallel. Otherwise, a time-sharing strategy is used [92]. The blocks are divided for their execution into *warps* that are the basic scheduling units in CUDA and consist of 32 consecutive threads.

Threads can access data on multiple memory spaces during their life time. CUDA architecture has six different memory spaces: global memory, registers, shared memory, local memory, constant memory and texture memory [92]. All the threads running on a GPU have access to the same global memory on the card that is one of the slowest memories on the GPU. However, access to global GPU memory is usually more than one order of magnitude faster than data transfers between CPU and GPU. In fact, the transfers between CPU and GPU are usually one of the most important bottlenecks on CPU-GPU heterogeneous computing. Registers are the fastest memory available on the card but they are entirely managed by the compiler. They are only accessible by each thread independently and the total memory space for registers has a very limited size. Shared memory is almost as fast as registers and it can be accessed by any thread of a block; its lifetime is equal to the lifetime of the block. Each thread has its own local memory but is one of the slowest memories on the card. Local memory is also entirely managed by the compiler. The compiler places variables in local memory when register spilling occurs, i.e., the kernel needs more registers than available. Constant memory is a read-only memory space for the device that is accessible for all threads. Texture memory has similar features than the constant memory, but it is optimized for certain access patterns. In the last years, the GPUs have incorporated two levels of cache for accessing to global memory, but both caches are quite small.

For designing a GPU implementation of an algorithm several aspects should be considered [91, 92] like minimizing data transfer between the CPU and the GPU, maximizing multiprocessor occupancy, coalescing memory accesses to global memory of the GPU, reducing the impact of memory latency, and avoiding different execution paths within threads of the same warp.

2.3.2 GPUs Used in this PhD Thesis

Throughout the development of this thesis six different GPUs were used. Table 2.1 shows the main features of the GPUs used in this thesis including the microarchitecture of the GPU, the CUDA compute capability (which allows to know some basic features of the GPU capacities), the number of cores of the GPU, the clock rate of the cores, and the theoretical peaks of the computational performance and the memory bandwidth of the global memory of GPU.

In the articles “Systolic genetic search, a systolic computing-based metaheuristic” (presented in Appendix A) and “A Systolic Genetic Search for reducing the execution cost of regression testing” (presented in Appendix B), which are two of the three core articles of this thesis, we use a GeForce GTX 480 and a GeForce GTX 780, respectively.

Table 2.1 Main features of GPUs used in this thesis.

	GeForce 9800 GTX+	Tesla C 1060	GeForce GTX 285	GeForce GTX 480	GeForce GTX 680	GeForce GTX 780
Microarchitecture	Tesla	Tesla	Tesla	Fermi	Kepler	Kepler
CUDA Compute Capability	1.1	1.3	1.3	2.0	3.0	3.5
CUDA Cores	128	240	240	480	1,536	2,304
Frequency (MHz)	1,836	1,296	1,476	1,401	1,006	863
Performance Theoretical Peak (GFlops)	705.02	933.12	1062.72	1344.96	3090.43	3976.70
Memory Bandwidth Theoretical Peak (GB/s)	70.40	102.40	159.00	177.40	192.26	288.00

Chapter 3

Evolutionary Algorithms and Their Parallelization

The resolution of an optimization problem lies in finding an optimal solution from a set of possible solutions (known as solution space or search space). The element is considered optimal in the sense that satisfies a certain criterion or objective. In order to measure the satisfaction of the objective, the notion of objective function is introduced and thus the goal of the optimization problem can be stated as the minimization or maximization of the objective function. These problems have been of great interest to the scientific community for its almost straightforward application to real world problems and the simplicity of its formulation [105].

The term *Metaheuristics* was coined in 1986 by Fred Glover to refer to heuristics with a higher level of abstraction [43]. Metaheuristics, as well as heuristics techniques, are able to find high quality solutions (many times optimal) to optimization problems in reasonable execution times. Unlike heuristic techniques that involve designing a specific procedure for each optimization problem, metaheuristics are general schemes of optimization algorithms that can be used for addressing a wide range of problems [14, 43, 111]. In general, metaheuristic techniques are considered as a general pattern of algorithms, which can be applied to the particularities of each problem with relatively few modifications.

This chapter provides the reader with the elementary notions of metaheuristics that will help for a better understanding of the rest of this PhD thesis. The structure of the chapter is as follows. Background on metaheuristics and some taxonomies proposed for classifying metaheuristics are presented in Section 3.1. Then, Section 3.2 introduces the family of Evolutionary Algorithms (EAs) techniques and Genetic Algorithms (GAs) in particular, since the evolutionary operators of GAs are used in the design of Systolic Genetic Search. Finally, Section 3.3 discusses the parallelization strategies that have been proposed for Evolutionary Algorithms.

3.1 Metaheuristics

Metaheuristics techniques [14, 111] are approximate and, in general, stochastic algorithms that can be abstracted with a high-level description of its basic components. They can be seen as general strategies for the design of underlying heuristics through their adaptation to the particularities of each specific problem. These algorithms can even incorporate specific information of the problem or a subordinated problem-specific heuristic.

Metaheuristics explore the search space efficiently, with the goal of finding high quality feasible solutions, i.e., with values of the objective function close to the optimal. The design of a metaheuristic algorithm should take into account a proper balance between diversification (i.e., the exploration of the search space) and intensification (i.e., the exploitation of the best solutions found). A metaheuristic should also explore the solution space without getting stuck in particular regions of the search space, specially avoiding getting trapped in local optima.

A large number of new metaheuristics have been proposed due to the increasing interest in the use of these techniques for the resolution of NP-hard optimization problems. However, only a small group of these proposals has consolidated in practice, demonstrating a broad spectrum of application and showing great maturity to be considered as an alternative when solving an optimization problem. Some of the most popular metaheuristic techniques are Iterated Local Search (ILS) [56, 75], Simulated Annealing (SA) [64, 85], Tabu Search (TS) [41, 43], Variable Neighborhood Search (VNS) [84], Greedy Randomized Adaptive Search Procedure (GRASP) [34], the family of Evolutionary Algorithms methods [37, 44, 104], Ant Colony Optimization (ACO) [32] and Particle Swarm Optimization (PSO) [60]. Several taxonomies have been proposed that classify metaheuristic techniques according to their characteristics [14].

One of the classification criteria distinguishes whether the source of inspiration of the algorithm comes from nature (nature-inspired techniques) or not (non-nature inspired techniques). This criteria is intuitive but there are difficulties in classifying some algorithms. According to this criteria, SA, EAs and ACO are nature-inspired algorithms, while ILS and TS are non-nature inspired techniques.

Another possible criterion distinguishes between algorithms that use memory or not. Metaheuristics can incorporate short-term or long-term memory mechanisms. Short-term memory is often used to recognize recently visited solutions, while long-term memory usually is used to accumulate the overall experience gained during the search process. According to this criteria, TS and ACO are algorithms that use memory, while SA, EAs and ILS are algorithms that do not use memory.

The most popular taxonomy distinguishes whether the algorithm uses a single candidate solution (trajectory-based techniques) or a set of candidate solutions (population-based techniques).

Trajectory-based metaheuristics consider a single point of the search space in each iteration of the algorithm. The algorithms in this category start from an initial point of the search space and update the position of the candidate solution by exploring the neighborhood of the solution (usually through the use of local search operators). Thus, the candidate solution describes a trajectory in the search spaces. The search process finishes when a maximum number of iterations is reached, a solution with an acceptable quality is found, or the search process is stagnated. ILS, SA, TS, VNS and GRASP are trajectory-based algorithms.

Population-based metaheuristics are characterized by using a set of solutions (population of solutions) in each iteration of the algorithm instead of a single solution. For this reason, these algorithms provide an intrinsic mechanism for the exploration of the search space. The numerical efficiency of a metaheuristics of this category mainly depends on how the algorithm manipulates the population in each iteration. EAs, ACO and PSO are population-based metaheuristics.

3.2 Evolutionary Algorithms

Evolutionary algorithms [44] are stochastic search methods inspired by the natural process of evolution of species. The origin of these methods dates back to the 1960s when John Holland considered the possibility of incorporating the Darwinian mechanisms [27] of selection and *survival of the fittest* to the resolution of problems in artificial intelligence [55]. This research on the simulation of processes of natural evolution of the species lead to the design of a new search and optimization algorithm that was later called evolutionary algorithms.

EAs iteratively evolve a population of individuals representing candidate solutions of the optimization problem at hand. The evolution process is guided by a *survival of the fittest* principle applied to the candidate solutions and it involves the probabilistic application of evolutionary operators to find better solutions. The whole process involves the use of a *fitness function* that is a metric closely related to the objective function of the optimization problem being solved.

Algorithm 1 presents the pseudocode of an Evolutionary Algorithm. The initial population is usually randomly generated. Then, every initial solution in the population is associated with a fitness value that measures the quality of the candidate solution. Each iteration of the algorithm can be divided into three main stages: selection, reproduction and replacement. In the selection stage, a temporary population is created in which the solutions are selected based on their fitness value, usually giving higher priority to higher quality solutions. After that, in the reproduction stage, new solutions are constructed applying evolutionary operators to the selected solutions. Typically, the evolutionary operators used are crossover (recombination of parts of individuals) and mutation (random changes in a single individual). Then, in the final stage, the new population is created, by replacing the worse adapted individuals of the population with new solutions generated in the iteration. The iteration loop is repeated until a certain stop criterion is reached. Finally, EAs return the best solution found during the whole search process (see Algorithm 1).

```
1  $t = 0$
2  $P(t)$ = generateInitialPopulation()
3  evaluate($P(t)$)
4  **while** *not stopCondition()* **do**
5      $t = t + 1$
6      select $P'$ from $P(t-1)$
7      apply evolutionary operators to $P'$ forming $P''$
8      evaluate($P''$)
9      form P(t) using $P''$ and/or $P(t-1)$ % replace
10 **end**
11 **return** *the best solution found*
```

Algorithm 1: Evolutionary Algorithm.

Evolutionary Algorithms are a family of techniques, among which Genetic Programming (GP), Evolution Strategies (ES), and Genetic Algorithms (GAs) stand out. GP [65, 102] is an evolutionary computation algorithm aimed for evolving computer programs. ES [10, 104] are in general aimed for solving optimization problems in continuous search spaces and are characterized by having a deterministic selection process. The next subsection is devoted to presenting GAs, as they are the algorithms from which SGS has adopted several components.

3.2.1 Genetic Algorithms

This subsection presents the classical GA with binary encoding and generational replacement since the evolutionary operators of this GA are used in SGS. GAs are one of the most popular types of EAs due to their adaptability to a wide range of problems. GAs are based on the general sketch of an Evolutionary Algorithm presented in Algorithm 1. The evolutionary operators used by GAs are the crossover and the mutation. The former is the main evolutionary operator of GAs and it allows to recombine the existing genetic material in the solutions, while the latter allows to modify the genetic information of a solution, introducing new genetic material in the solution. Both mutation and crossover operators are applied probabilistically. In general, the application rate of the crossover operator is high, while the application rate of the mutation operator is very low.

In GAs, the candidate solutions, which are also known as individuals, are usually represented using binary strings. The binary string with the genetic information of an individual is known as a chromosome. Each chromosome is composed by genes and the possible values of a gene are called alleles. The genotype of an individual is formed by the set of chromosomes that define its characteristics. GAs usually have a single chromosome per individual, but there are proposals with more than one chromosome per individual, such as diploid GA [120]. On the other hand, the phenotype of an individual is the point of the search space of the problem that is represented by the candidate solution. The GA uses a *codification* and a *decoding* function that are used for obtaining the genotype from the phenotype of an individual and vice versa, respectively. Each chromosome defines a single point of the search space, but each solution can be encoded by more than one different chromosomes. Both functions are of vital importance for the algorithm since selection operates on the phenotype, while reproduction operates on the genotype.

In the binary string representation, the solutions are traditionally encoded as a string of bits of a fixed length. The length of the string is associated with the features of the problem instance. Binary encoding was originally considered as the standard representation of GA. In fact, binary strings can be used for encoding integers, reals, graphs or sets. Binary strings are also easy to handle, which greatly facilitates the design of crossover and mutation operators. However, other encodings are also possible like real encoding [54] or permutational encoding [118].

GAs use a selection mechanism that encourages the selection of the best adapted individuals of the population. The probability of an individual of being selected for reproduction is related to its fitness value, giving a greater probability to individuals with a fitness value close to the fitness value of the optimal solution. In traditional sequential GAs (and EAs), the population is organized into a single group, which is known as panmixia. In panmixia, the population is not structured in groups of individuals, and therefore there are no restrictions in the selection of individuals for reproduction. One of the most accepted selection mechanisms is the binary tournament selection. In the binary tournament selection, two individuals are chosen at random from the population, and the winner of the tournament, i.e., the one with higher fitness value, is selected for reproduction.

The crossover operator uses the information of the search space gathered in current solutions to find better solutions through the recombination of genetic material. The crossover operator takes two parent solutions and produces one or two child solutions, depending on the characteristics of the crossover operator used. GAs originally used the Single Point Crossover (SPX) for the binary string representation. SPX randomly generates a crossover point and exchanges the genetic information between the two chromosomes from the crossover point to the end of the string, thus generating two potentially new individuals. SPX is the crossover operator used by David Golberg in the seminal Simple Genetic Algorithm [44]. Figure 3.1 graphically shows how the SPX operates.

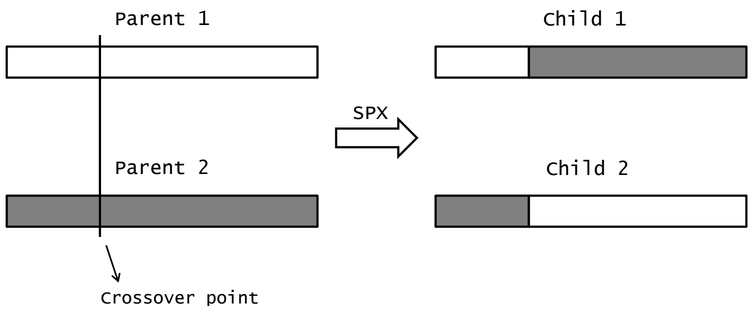


Fig. 3.1 Single Point Crossover.

One of the most popular crossover operators for GAs is the Double Point Crossover (DPX). In DPX, two crossover points are chosen randomly. This crossover exchanges the genetic information between the two chromosomes from the first crossover point to the second crossover point. Figure 3.2 graphically shows how the DPX operates. The SPX and DPX operators can be generalized in the *N-point* crossover [28, 110], in which N random crossover points are generated and then the swaps are made between the $N+1$ segments that are determined by the crossover points.

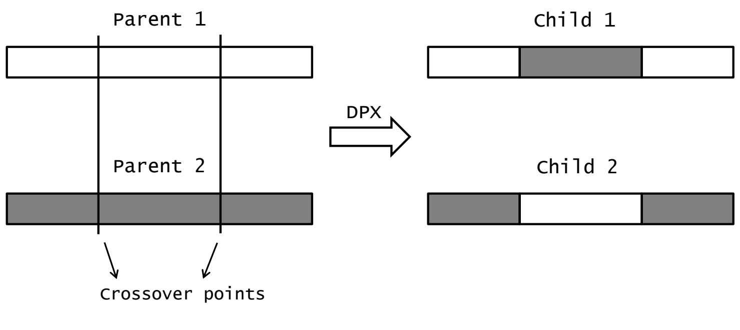


Fig. 3.2 Double Point Crossover.

The mutation operator helps to maintain the genetic diversity through the execution of the GA, incorporating new genetic material into the solutions or reincorporating genetic material that has been lost during the search process. The mutation operator takes one solution and randomly modifies the values of the genes of the solution.

The mutation operator most commonly used in GAs for the binary string representation is the *bit-flip* mutation. In the bit-flip mutation, some genes of the chromosome are chosen randomly, and the value of the genes is inverted, i.e., if the value is one, it is changed to zero and vice versa.

A sketch of a GA that is used in this thesis for the comparison with SGS is presented in Algorithm 2. The initial population is randomly generated and then the algorithm iterates until the maximum number of iterations is reached. In each iteration, a new generation of *popSize* individuals is produced by the selection, recombination, and mutation loop, and then replaces the old population, i.e., it is a generational GA. Two parents solutions p1 and p2 are selected from the population by binary tournament based on their previously computed fitness value. Then, two new solutions p1 and p2 are created by applying the DPX to the parents with a given probability *cp*. Finally, the bit-flip mutation is applied to the offspring solutions with a probability *mp*.

3.3 Parallel Evolutionary Algorithms

Among parallel metaheuristics, Parallel Evolutionary Algorithms (PEAs) [4, 76] have been extensively adopted and are nowadays quite popular, mainly because EAs are naturally prone to parallelism. For this reason, the study of parallelization strategies for EAs have laid the foundations for working in parallel metaheuristics. The most usual criterion for categorizing PEAs distinguishes three main different categories [20, 77]: the master-slave model, the distributed or island model, and the cellular model. In the master-slave model the population is panmitic as in the traditional sequential implementation, while the other two categories correspond to two population models that are markedly different from panmixia. Some authors also identify a fourth category that corresponds to the parallel independent run of the same sequential EA [20, 77]. However, these implementations have no interaction between the independent executions and they can be considered *embarrassingly parallel* algorithms [53].

The master-slave model corresponds to the functional distribution of the algorithm [20, 77]. The master process executes the main loop of the EA and controls the search procedure, computing the selection mechanism, the replacement of the population, and in most cases the reproduction of the individuals. On the other hand, the slave processes only

```
input  : The population size $popSize$, the crossover probability $cp$ and the mutation
         probability $mp$
output : The best solution found
$generation = 0$;
$pop = generateRandomPopulation(popSize)$;
$evaluate(P(t))$;
while not maximum number of iterations reached do
    for $i$=1 to $\frac{popSize}{2}$ do
        $p1 = selectBinaryTournament(pop)$;
        $p2 = selectBinaryTournament(pop)$;
        $(p1', p2') = doublePointCrossover(p1, p2, cp)$;
        $p1'' = bitFlipMutation(p1', mp)$;
        $p2'' = bitFlipMutation(p2', mp)$;
        $newPop[2*i-1] = p1''$;
        $newPop[2*i] = p2''$;
    end
    $pop = newPop$;
    $evaluate(pop)$;
    $generation = generation + 1$;
end
return the best solution found
```

Algorithm 2: Genetic algorithm

perform subordinate tasks that are usually associated with the evaluation of fitness of the solutions. The master process sends the candidate solutions to several slaves processes, and the slaves return the corresponding fitness values. Figure 3.3 illustrates the master-slave model for EAs. This parallelization strategy for EAs when is synchronously implemented has exactly the same behaviour than the sequential implementation of the EA. As a consequence, this model can only reduce the runtime of the sequential implementation, profiting from the additional hardware resources available.

In the distributed or island model [20, 77], the population is partitioned into a small number of subpopulations, called islands or demes, that evolve in semi-isolation. Each subpopulation works independently as a sequential EA and they can even apply different evolutionary operators. As a consequence, the selection for reproduction and the application of the evolutionary operators is local for each island. For this reason, each subpopulation usually explores different regions of the search space. The only exchange of information among the subpopulations is produced through a migration operator that exchanges individuals between the islands, thus increasing the diversity of the subpopulation. The island model for EAs is illustrated in Figure 3.4.

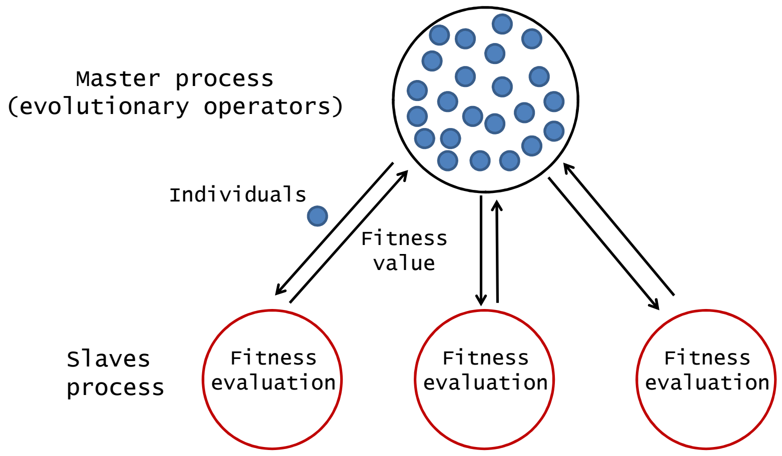


Fig. 3.3 Master-slave model for EAs.

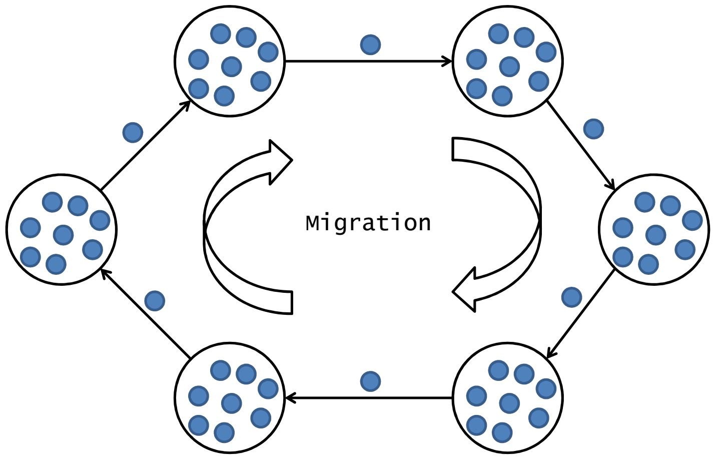


Fig. 3.4 Island model for EAs.

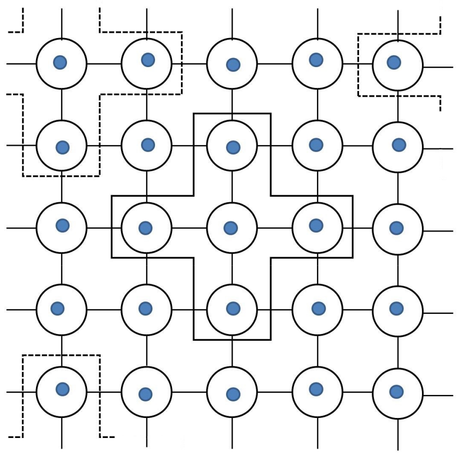


Fig. 3.5 Cellular model for EAs.

Finally, the cellular model [2] works with a single population structured in many small overlapped neighborhoods. Each individual is placed in a cell on a toroidal n-dimensional grid and belongs to several neighborhoods. The interactions between individuals are limited since the selection of parents for reproduction is local to each neighborhood. Therefore, mating is restricted to each of the neighborhoods. The effect of finding high-quality solutions gradually spreads to other neighborhoods along the grid due to the use of a *diffusion model* that is a consequence of the neighborhoods overlapping [2]. The cellular model for EAs on a 2D grid is illustrated in Figure 3.5.

As it was already stated in the motivation of this thesis presented in Chapter 1, all the parallelization strategies for EAs have already been ported to GPU architectures in previous works. For instance, the cellular model GA is studied in [109, 114], the island model GA is discussed in [123], while the master-slave implementation is addressed in [78].

Chapter 4

Systolic Genetic Search

Systolic Computing is inspired by the behavior of the cardiovascular system [49, 74]. In physiology, the term *systole* is used to refer to the cyclic contraction of the heart that pumps blood through the body. In each cardiac cycle, the heart first relaxes in the diastole phase to refill with circulating blood and then it contracts in the systole phase. Due to the systolic contraction, oxygenated blood is ejected from the heart into the arterial system with a regular cadence or rhythmically to meet the metabolic needs of the tissues [49, 74].

Systolic computing architectures [67, 69] are composed by data processing units (also known as cells) that are connected through a network enabling a data flow between neighboring units. The processing units are able to compute relatively simple operations to data received from neighboring units, that is then passed through the next cell in the topology. Each processing unit of a systolic computing architecture is analogous to the heart, regularly receiving data, processing and pumping data out, in order to keep a constant flow of data in the system [68].

In this thesis, the optimization algorithm Systolic Genetic Search is proposed. SGS adapts the operation of genetic algorithms to a systolic computing architecture. The algorithm is characterized by the flow of solutions through data processing units following a synchronous and structured plan. Each processing unit applies evolutionary operators to the circulating tentative solutions in order to obtain new solutions that continue moving across the units. In this way, solutions are refined again and again by simple low complexity search operators.

The rest of this chapter is structured as follows. First, in the next section the SGS algorithm is described. Then, and since SGS is specially conceived for GPUs, Section 4.2 provides a brief description of the GPU implementation of SGS. Section 4.3 discusses related papers with similar ideas to SGS. Finally, Section 4.4 presents the methodology followed in this work for conducting the evaluation of the numerical and computational performance of the algorithms.

4.1 Systolic Genetic Search Algorithm

In a SGS algorithm, the solutions are synchronously pumped through a bidimensional grid of cells, circulating through an horizontal and a vertical data streams. At each step of SGS, two solutions enter to the cell at position (i,j), $s_{\mathrm{H}}^{\mathrm{i,j\text{-}1}}$ from the horizontal data stream (cell at position $(i,j-1)$) and $s_{\mathrm{V}}^{\mathrm{i\text{-}1,j}}$ from the vertical data stream (cell at position $(i-1,j)$). Then, the cell computation is performed generating two (potentially new) solutions that continue moving through the grid, $s_{\mathrm{H}}^{\mathrm{i,j+1}}$ through the horizontal data stream (cell at position $(i,j+1)$) and $s_{\mathrm{V}}^{\mathrm{i+1,j}}$ through the vertical data stream (cell at position $(i+1,j)$), as it is shown in Figure 4.1.

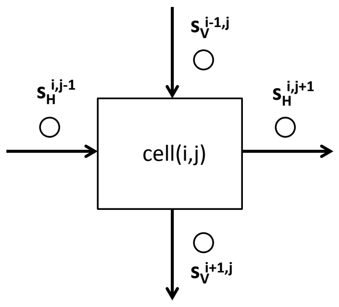


Fig. 4.1 Ingoing and outgoing solutions from cell at position(i,j).

The computation performed by the cells is described next. At the very beginning of the operation, each cell generates two random solutions which are aimed at moving horizontally and vertically, respectively. At each step of SGS, which is known as systolic step, two solutions enter to each cell, one from the horizontal data stream and one from the vertical data stream. Then, adapted evolutionary/genetic operators (crossover and mutation) are applied to the incoming solutions in order to generate two potentially new solutions. SGS is commonly used with crossover operators that take two parent solutions and produce two children solutions. However, it is easy to adapt the cell computation for using crossover operators that only produce one child solution, e.g., applying the crossover operator twice. Later, the cell uses elitism to determine the outgoing solutions that continue moving through the grid, choosing for each data stream between the incoming solution and a newly generated one. The use of elitism is critical, as there is no global selection process like in standard EAs/GAs. Finally, each cell sends the outgoing solutions to the next cells of each of the data streams. The pseudocode of SGS is presented in Algorithm 3.

It is important to highlight that the general idea of the SGS algorithm described can be adapted to any solution representation and any particular operator. Since in this thesis the problems addressed are all binary problems, the solutions are encoded as binary strings, as in

```
foreach cell c do
    $s_H$ = generateRandomSolution()
    $s_V$ = generateRandomSolution()
    sendSolutionThroughHorizontalDataStream($S_H, c$)
    sendSolutionThroughVerticalDataStream($S_V, c$)
end
for $i = 1$ to maxGeneration do
    foreach cell c do
        $s_H$ = receiveSolutionFromHorizontalDataStream($c$)
        $s_V$ = receiveSolutionFromVerticalDataStream($c$)
        $(new_H, new_V)$ = crossover($s_H, s_V$)
        $new_H$ = mutation($new_H$)
        $new_V$ = mutation($new_V$)
        $new_H$ = elitism($s_H, new_H$)
        $new_V$ = elitism($s_V, new_V$)
        sendSolutionThroughHorizontalDataStream($new_H, c$)
        sendSolutionThroughVerticalDataStream($new_V, c$)
    end
end
```

Algorithm 3: Systolic Genetic Search.

classical GAs. For this reason, we use the bit-flip mutation and the two-point crossover as evolutionary search operators of SGS.

Several aspects have to yet be precisely defined to fully characterize the SGS algorithm. These aspects include: the flow of solutions through the grid (i.e., how is the complete interconnection topology of the systolic structure and how do the solutions move through this structure), the size and the dimensions (i.e., the number of rows and columns) of the grid, and how the crossover points and the mutation point of each cell are calculated.

Throughout this thesis and specially in the three articles compiled in this document, different alternatives for each of these aspects were studied. In [98] (presented in Appendix A), four different strategies for the flow of solutions through the grid are analyzed. Regarding the selection of the crossover points and the mutation point, in [98] the points are preprogrammed at fixed positions of the tentative solutions (according to the location of the cell in the grid), in [100] (presented in Appendix C) the points are chosen randomly for each cell, and in [99] (presented in Appendix B) an hybrid approach is used. Finally, four different grid sizes with different dimensions where studied in [98–100]. This is presented in greater detail in Chapter 5 and in the referred articles.

4.2 GPU Implementation of SGS

This section is devoted to presenting the general idea of the implementation of SGS on GPU. Algorithm 4 presents the pseudocode of SGS for the host side (CPU). Initially, the seed for the random number generation is transferred from the CPU to the global memory of the GPU and the constant data associated with the problem required for computing the fitness values is also transferred from the CPU to the GPU memory. Then, the population is initialized on the GPU (Step 3) and the fitness of the initial population is computed afterwards (Step 4). At each iteration, the crossover and mutation of the solutions of each cell of the grid are executed (Step 6), and the systolic step is completed by calculating the fitness evaluation and applying the elitist replacement (Step 8). Finally, when the algorithm reaches the stop condition, the results are transferred from the GPU to the CPU.

```
1  transfer seed for random number generation to GPU
2  transfer problem data to GPU's memory
3  call initPopulation kernel to initialize population
4  call fitnessCalculation kernel to calculate fitness of the population
5  for $i = 1$ to $maxGeneration$ do
6      call crossoverAndMutation kernel to compute crossover and mutation operators
7      call fitnessCalculation kernel to calculate fitness of the population
8      call elitism kernel to calculate elitism
9  end
10 transfer final results from GPU to CPU
```
Algorithm 4: SGS Host Side Pseudocode

Now, the organization of the data on the GPU memory is detailed. As it is shown in Figure 4.2, two independent memory spaces of the GPU global memory are used to store the population and its associated fitness value. While the memory space that contains the population/the fitness values in generation t is read, the new solutions/fitness values from generation $t+1$ can be written in the other memory space allowing concurrent access to the data (disjoint storage). Each memory space stores an struct containing an array with the solutions moving horizontally, an array with the solutions moving vertically, an array with the fitness values corresponding to the solutions moving horizontally, and an array with the fitness values corresponding to the solutions moving vertically, following the well known software pattern for vectorization, struct of array (SoA) [80]. Depending on the characteristics of the problem and the size of the instance, the problem data can be stored in texture memory or global memory (in Figure 4.2 it is shown allocated in global memory). Finally, a memory space for auxiliary data is also allocated in the global memory that is used for storing the states of the random number generators, intermediate calculations of the computation of the fitness evaluation, etc.

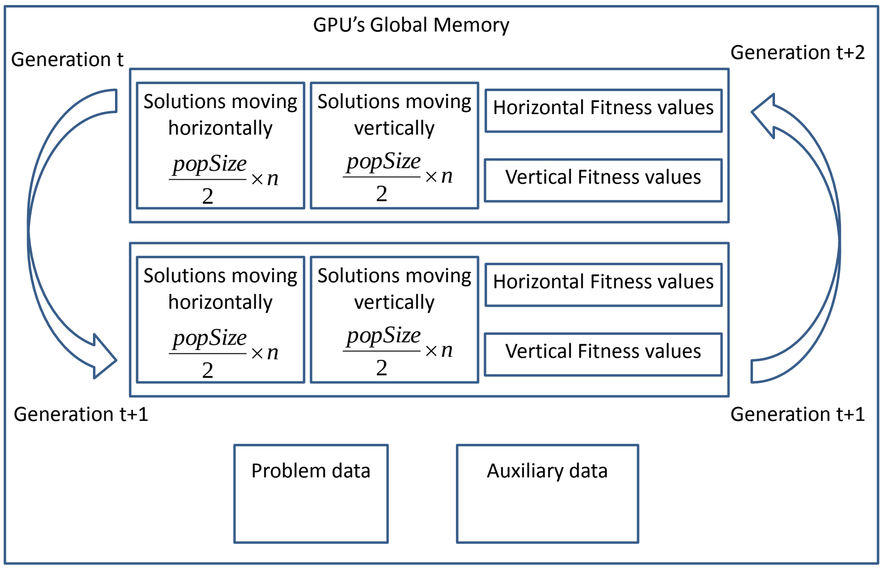
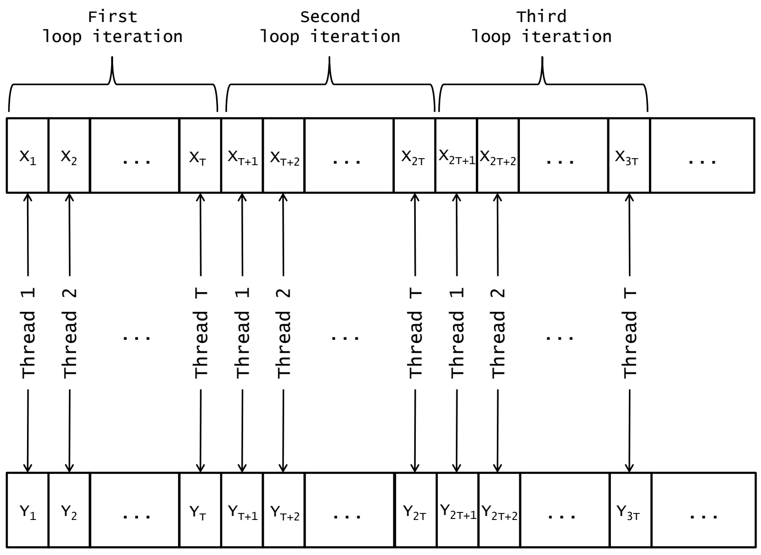

GPU's Global Memory
Generation t
Generation t+2
Solutions moving horizontally
$\frac{popSize}{2} \times n$
Solutions moving vertically
$\frac{popSize}{2} \times n$
Horizontal Fitness values
Vertical Fitness values
Solutions moving horizontally
$\frac{popSize}{2} \times n$
Solutions moving vertically
$\frac{popSize}{2} \times n$
Horizontal Fitness values
Vertical Fitness values
Generation t+1
Generation t+1
Problem data
Auxiliary data

Fig. 4.2 Data organization on the GPU.

The kernel operation is explained next. The `initPopulation` kernel initializes the population in the GPU using the CUDA CURAND Library [90] to generate random numbers. The kernel is launched with a configuration that depends on the total number of bits that have to be initialized, following the guidelines recommended in [91].

The `crossoverAndMutation`, `fitnessCalculation` and `elitism` kernels are implemented following the idea used in [94], in which operations are assigned to a whole block and all the threads of the block cooperate to perform a given operation. As a consequence, these kernels are launched with as many blocks as the number of cells of the grid of SGS, i.e., each block processes one cell of the grid. If the solution length is larger than the number of threads in the block, each thread processes more than one element of the solution but the elements used by a single thread are not contiguous. Thus, each operation is applied to a solution in chunks of the size of the thread block (T in the following figure), as it is shown in Figure 4.3. Depending on the characteristics of the problem and the size of the instance, these three kernels can be merged into a single kernel in order to increase the performance. This allows to temporarily store the two solutions being constructed in shared memory, which reduces the accesses to global memory, even though it restricts the size of the instances that could be resolved.

The `crossoverAndMutation` kernel initially calculates the global memory location of the two ingoing (the two solutions that have to read and processed by the cell) and the two



First loop iteration
Second loop iteration
Third loop iteration
X_1 X_2 ... X_T X_{T+1} X_{T+2} ... X_{2T} X_{2T+1} X_{2T+2} ... X_{3T} ...
Thread 1
Thread 2
Thread T
Y_1 Y_2 ... Y_T Y_{T+1} Y_{T+2} ... Y_{2T} Y_{2T+1} Y_{2T+2} ... Y_{3T} ...

Fig. 4.3 Threads organization.

resulting solutions (the memory location where the solutions have to be stored) from the block identifiers and determines the crossover points and the mutation point. These values are calculated by thread zero of the block and they are stored in shared memory in order to make them available for the rest of the threads of the block. Then, the kernel applies the crossover operator, processing the solution components in chunks of size of the thread block (as it was explained above). Finally, the thread zero of the block mutates the two intermediate solutions.

The `fitnessCalculation` kernel first calculates the global memory location of the two solutions of the cell and the global memory location where the fitness values have to be stored from the block identifiers. Then, partial fitness values are computed by each thread in chunks of size of the thread block, using the solutions and the problem data. The kernel uses the shared memory of the GPU to temporarily store the partial fitness values computed by each thread. Then, the kernel applies the well-known reduction pattern [80] to these values to calculate the full fitness value of each of the solutions. The fitness evaluation, when the problem data is a matrix, could involve irregular computations that are not suitable for GPU. In such case, the evaluation can be transformed into a matrix-matrix multiplication operation that follows a more structured pattern of computation and it is well suited for GPU. This transformation involves the use of the auxiliary data memory space for storing intermediate results and external libraries that compute linear algebra operations efficiently [89].

Finally, the `elitism` kernel determines the best solution for each data stream, considering the fitness values from the new solutions and from the ingoing solutions. If a new solution is better than the ingoing solution, the new solution and its fitness value are copied to the memory space of the next generation. Otherwise, the ingoing solution and its fitness value are the ones that are copied. The copy is made in chunks of size of the thread block.

4.3 Related Work

Besides the context of this thesis already presented in Chapter 1, there are some additional relevant works that should be briefly commented. This section analyzes published material which is related to the SGS algorithm presented in this thesis.

Few efforts have been devoted to design optimization algorithms based on systolic computing-like architectures [67, 69, 70]. Indeed, only in [23] and in the works of Bland and Megson [11–13, 82, 83] an implementation of a GA on VLSI and FPGA architectures in a systolic fashion is proposed. However, this research line was early discarded due to the complexity of translating the GA operations into the recurrent equations required for the hardware definition.

A direct antecedent of SGS is Systolic Neighborhood Search (SNS) [5, 115, 116]. As a matter of fact, SGS can be seen as an advanced version of SNS, exploring a more sophisticated approach and involving more diverse operations. Both algorithms share the arrangement of solutions into a grid, but SNS only circulates solution through an horizontal data stream, whereas SGS moves solutions not only through an horizontal data stream but also through a vertical data stream. This means that SNS manages a single solution on each cell instead of a pair of solutions as in SGS. As a consequence, SNS uses a more simple search strategy than in SGS. Indeed, SNS is based on using a local search as the working operation in the cells, while SGS is based on using the crossover and the mutation operators of GAs.

During the development of this thesis, a hardware-oriented GA for FPGA architectures was presented in [48, 106] with matching points with the proposal of this thesis. The algorithm introduced in these works is called Pipelined Genetic Propagation (PGP), and is based on propagating and circulating a group of individuals in a directed graph structure. Data is transported in a pipelined manner between the nodes of the graph that perform genetic operations on the circulating data. However, in PGP there are different types of nodes, which perform a different specific operation. There are selection nodes, crossover nodes and mutation nodes, which substantially distinguishes PGP with respect to SGS.

SGS have similarities with the cellular model of EAs [2], but there are strong conceptual design goals that make the two underlying search models fairly different. Although in a first impression the models look alike, the only point of contact of both models is that the solutions are placed in a structured grid. Two main differences emerge.

In the first place, the information flow in both models is quite different. While the solutions remain static in the same position of the grid and all the exchange of information among solutions is caused by the overlapping of neighborhoods in the cellular model, SGS is based on the flow of solutions. That is, the constant movement of all the solutions through the grid produces the exchange of information between the solutions. As a consequence, the solutions that could be mated in SGS are dynamic during the execution of the algorithm, while in the cellular model the mating is in general static, i.e., a given solution can only be mated with the same set of solutions for the whole execution. Secondly, each cell applies the evolutionary operators to produce new solutions independently of the other cells in SGS, i.e., when a cell is applying those operators it can be considered isolated from the rest of the grid, while in the cellular model each cell needs the neighboring cells to be able to produce new solutions.

SGS also differs from the two other parallelization strategies for EAs (besides the cellular model). In the first place, SGS restricts mating to the pair of solutions that are on a cell, while in the master-slave model the population is panmitic and there are no restrictions for mating. Second, the population of SGS is divided into two subpopulations, the solutions that are moving horizontally and the solutions that are moving vertically. In SGS, mating is limited by the subpopulations, being only possible interactions between two solutions that belong to the two different subpopulations, while in the island model the mating process is local to each of the subpopulations.

Finally, a major difference with the three classical parallelization strategies for EAs is that SGS does not have an explicit selection process. As a matter of fact, the selection of parents for reproduction in SGS is implicit, and it is determined by the flow of solutions through the grid, while the other parallelization strategies have a explicit selection process.

4.4 Methodology for the Evaluation of the Numerical and Computational Performance

The experiments conducted in this thesis include the study of the numerical efficiency of SGS and the study of the computational performance of the parallel GPU implementation of SGS. Since the algorithm proposed in this thesis and most of the algorithms used for comparison purpose are stochastic algorithms, statistical tests are used to assess the significance of the experimental results obtained. First, at least fifty independent runs for each algorithm and each problem instance have been performed [103]. Two different statistical procedures are considered, one analyzing the statistical difference for each problem and instance independently, and one involving the statistical differences for the algorithms across the multiple instances. The criterion for deciding which approach is used is related to the number of instances available for the problem considered.

For the former approach, the following statistical procedure is used [107] to determine if the distribution of a particular metric for each algorithm and each instance independently is statistically different. First a Kolmogorov-Smirnov test and a Levene test are performed in order to check, respectively, whether the samples are distributed according to a normal distribution and whether the variances are homogeneous (homocedasticity). If the two conditions hold, an ANOVA I test is performed; otherwise a Kruskal-Wallis test is performed. Since more than two algorithms are involved in the study, a post-hoc testing phase consisting in a pairwise comparison of all the cases compared using a correction method (either Bonferroni-Dunn or Holm) on either the Student's t-test (if the samples follow a normal distribution and the variances are homogeneous) or the Wilcoxon-Mann-Whitney test (otherwise) is also performed.

For the latter approach, the Friedman's test is used for ranking the algorithms according to some particular metric [31, 107]. This test is used to check if the differences in the metric are statistically significant among the algorithms for the whole set of instances of a particular problem. Since more than two algorithms are involved in the study, a multiple comparison using the Holm's post-hoc procedure is performed.

Chapter 5

Articles Supporting this PhD Thesis

The results of the research of this PhD thesis have been validated with several peer-reviewed publications both in international journals and conferences, and a book chapter. In particular, three articles have been published in international journals indexed in ISI-JCR, which serve as support for this thesis. In addition to this, three papers have been presented at international conferences and one book chapter has also been published. The structure of the chapter is as follows. The next section presents a summary of each one of the three articles compiled in this thesis. Then, Section 5.2 describes the other publications that have been produced in this work.

5.1 Articles Compiled in this PhD Thesis

This section presents the summary of the three articles that support this thesis. All of them address the subject of the thesis and as a whole give coherence to the research work developed. Thus, in the first place, a new algorithm is designed combining ideas from systolic computing and evolutionary computation and it is validated over three different problems. Then, from the feedback of the first article, the original design is adjusted, and SGS is used for solving a real-world problem from the field of software engineering. Finally, as the behavior of the new algorithm proposed has been studied only empirically, a theoretical and experimental analysis on the flow of solutions of SGS is conducted. A brief summary of the articles is presented next.

[98] Pedemonte, M., Luna, F., and Alba, E. (2015). Systolic genetic search, a systolic computing-based metaheuristic. *Soft Computing*, 19(7):1779–1801

This article is included in Appendix A. The starting point for the design of SGS is the proposal of SNS in [5, 115, 116]. On the basis of the interesting results obtained by SNS, the purpose of our research has been to extend the idea of SNS to more complex search strategies. With this goal in mind, the design of SGS has incorporated a bidimensional grid, and the crossover and the mutation operators of GAs within each cell, thus allowing the algorithm to manipulate two solutions on each cell. The proposal of SGS is consolidated in this work, after some preliminary explorations of the idea in [95, 96].

In order to validate the proposal of SGS, two classic benchmark problems, the Knapsack Problem (KP) [101] and the Massively Multimodal Deceptive Problem (MMDP) [45], and one real-world problem, the Next Release Problem [8] from the field of software engineering, have been used as testbed. The experimental evaluation has also included two GAs and a Random Search to be used as a basis for comparison. The GAs have been chosen because they share the same basic search operators and use a panmitic population, so the underlying search engine of the techniques can be compared.

The experimental evaluation have shown that three instantiations of SGS have a great potential regarding the quality of the solutions obtained. With respect to the performance of the parallel implementation of SGS on GPU, even though the three aforementioned instantiations of SGS have achieved large runtime reductions from the sequential implementation and exhibited a good scalability with high dimensional instances, the instantiation named SGS_B has systematically obtained the shortest runtime for all the problems and instances considered.

[99] Pedemonte, M., Luna, F., and Alba, E. (2016). A systolic genetic search for reducing the execution cost of regression testing. *Applied Soft Computing*, 49:1145 – 1161

This article is included in Appendix B. After the validation of the SGS in the previous work, the focus of this new paper has been addressing a real-world problem and evaluating whether the proposed algorithm is competitive with the state-of-the-art for the problem or not. For this, the Test Suite Minimization Problem (TSMP), a software testing problem that arises in regression testing, has been selected as an application case. A first approach based on a simpler formulation of this problem using SGS was made in [97]. The original design of SGS has been adjusted in this paper from the feedback of the first article [98].

The experimental evaluation has been conducted on instances generated for eight real-world software programs and it also included two GAs (similar to the ones used in the previous work [98]) and four heuristics specially designed for the TSMP that are the state-of-the-art for this problem. The experimental results confirm the virtues of SGS, being worthy of mentioning that SGS is the algorithm with the best numerical performance (outperforming the state-of-the-art algorithm for this problem) and that the GPU implementation of SGS has the best computational performance among the EAs studied.

[100] Pedemonte, M., Luna, F., and Alba, E. (2018). A theoretical and empirical study of the trajectories of solutions on the grid of systolic genetic search. *Information Sciences*, 445-446:97 – 117

This article is included in Appendix C. The two previous articles were centered in the design of the newly proposed algorithm and in the application of SGS for solving a real-world problem. For this reason, the behavior of the new algorithm proposed has been studied only empirically. The focus of this new work lies in studying the flow of solutions through the theoretical analysis of the trajectories described by the solutions along the grid of SGS.

The theoretical analysis has found that, in the grids used thus far, there are cells in which the two incoming solutions are direct descendants of a pair of solutions that have already been mated in another cell of the grid. For this reason, a new grid that does not have this limitation is designed.

In addition to the theoretical analysis, an experimental evaluation has been also conducted to examine how the different features of the grids impact in the effectiveness of the algorithm. The experimental evaluation has been performed on three deceptive problems (the MMDP [45], the six-bit fully deceptive subfunction of Deb and Goldberg [29, 30], and the four-bit fully deceptive subfunction of Whitley [119]) and it has also included two GAs with similar features to the ones used in the two previous works.

The experimental results has confirmed that SGS is able to outperform the GAs, which use a panmitic population, corroborating that the underlying search engine of SGS is highly effective. The results have also shown that the use of grids limiting the mating of descendants of pairs of solutions that have already been mated benefits the search engine of SGS. Indeed, the grids so-designed guarantee a better diffusion of highly fitted genetic material through the grid, which produces a higher diversity on the cells of the grid that avoid the algorithm to get stuck and, thus, reaching better solutions.

5.2 Other Peer-reviewed Publications

The following subsections list the rest of the publications grouped according to the type of publication, providing a brief summary of their contents.

5.2.1 Proceedings of International Conferences

[94] Pedemonte, M., Alba, E., and Luna, F. (2011). Bitwise operations for GPU implementation of genetic algorithms. In *Genetic and Evolutionary Computation Conference, GECCO'11 - Companion Publication*, pages 439 – 446

This work addresses the influence on the performance of a GA of how the population of the GA is stored in the memory of the GPU. Individuals are often represented using binary strings in GAs and thus there are two alternatives for storing the binary strings in the memory. The former approach consists in using the boolean data type, which is easier to implement but wastes memory (seven out of eight bits) that is a valuable and limited resource in GPUs. The latter approach consists in packing multiple bits in a non-boolean data type, which almost does not waste memory but involves working at bit level (using bitwise operations) that is more complicated.

The GA used in this work have similar features to that used in the rest of the thesis: binary tournament, either SPX or DPX, bit-flip mutation and generational replacement. The results obtained in the experimental evaluation have shown that the use of bit packing for storing binary strings is able to reduce the execution time up to 50% on the GPU.

Although bit packing was not finally used in the GPU implementation of SGS, the thread organization that processes the solution components in chunks (assigning the operation to a whole block and making all the threads of the block cooperate to perform the operation) was conceived in this work. This strategy has been then used throughout the entire thesis.

[95] Pedemonte, M., Alba, E., and Luna, F. (2012). Towards the design of systolic genetic search. In *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1778–1786. IEEE Computer Society

This work is the first approach to the design of Systolic Genetic Search algorithm. The basic idea of the algorithm outlined in this work, remained unchanged along the entire thesis.

The experimental evaluation was conducted on instances of the KP and it included two panmitic GAs and a Random Search algorithm. SGS showed promising results both in the quality of the solutions obtained and in the performance of the GPU implementation.

[97] Pedemonte, M., Luna, F., and Alba, E. (2014). Systolic genetic search for software engineering: The test suite minimization case. In Esparcia-Alcázar, A. I. and Mora, A. M., editors, *Applications of Evolutionary Computation - 17th European Conference, EvoApplications 2014, Granada, Spain, April 23-25, 2014, Revised Selected Papers*, volume 8602 of *Lecture Notes in Computer Science*, pages 678–689. Springer

After setting the foundations of the SGS algorithm in [98], a survey on relevant real-world problems from the field of software engineering was conducted to identify and select an application case for SGS. The problem selected was the TSMP.

This work presents the first approach followed to tackle the TSMP with SGS. The formulation used in this work for the TSMP is more simple than the formulation that was then used in [99]. In this case, the goal is to find a set of test cases that maximizes the number of test requirements covered, while in the other work the set of test cases not only has to cover all the set of test requirements but also has to minimize a cost associated with the execution of the test cases.

The experimental evaluation was conducted on instances of seven real-world software programs and it also included two panmitic GAs. The experimental results showed that SGS was both effective and efficient for tackling the TSMP. This effort was later consolidated in [99].

5.2.2 Book Chapters

[96] Pedemonte, M., Luna, F., and Alba, E. (2013). New ideas in parallel metaheuristics on GPU: Systolic genetic search. In Tsutsui, S. and Collet, P., editors, *Massively Parallel Evolutionary Computation on GPGPUs*, Natural Computing Series, chapter 10, pages 203–225. Springer

This book chapter is chronologically the second publication on SGS. The focus of the work is analyzing how the features of different GPUs impact on the performance of the parallel GPU implementation of SGS. The characteristics of both the SGS algorithm and the GPU implementation of SGS used in this work are exactly the same than in [95].

The experimental evaluation was conducted on instances of the KP and it included two panmitic GAs and a Random Search algorithm. The experiments were executed in four different GPUs.The evaluation showed that the changes produced in GPUs in a period of less than two years impacted on a runtime reduction of the algorithm of more than 26$\times$ for the largest instances considered. Taking into account, the results in a broader perspective, it also shows the impressive progress on the design and computing power of the GPUs.

Chapter 6

Conclusions

The main goal of this thesis has been the design, implementation and evaluation of new parallel metaheuristic algorithms and/or new parallel models for GPUs, taking advantage of the high degree of parallelism present in modern GPU architectures. This objective was motivated by the challenge that represents the current and future scenarios of application of metaheuristic techniques and by the disruptive emergence of the massively parallel, widely available and low-cost GPUs. In order to address such a scenario, in this thesis we have engineered a new optimization algorithm, which combines ideas from Systolic Computing and Evolutionary Algorithms. The optimization algorithm is aimed at being both effective and efficient, taking specially advantage of the particular features of the GPU architecture.

The structure of this chapter is described next. First, Section 6.1 presents the general conclusions of this thesis and, second, the lines of future work that have been identified during this research are included in Section 6.2.

6.1 Concluding Remarks

The very first conclusion drawn from this work developed in this thesis is that the newly proposed algorithm is highly effective for tackling both benchmark and real world problems, finding optimal or near optimal solutions in short execution times. In particular, SGS has outperformed two competitive Evolutionary Algorithms. These EAs use a panmitic population and evolutionary operators that are similar to those used by SGS. This highlights the importance of the underlying search engine of SGS, which has shown to be highly accurate.

Second, the results obtained by SGS for the two real-world problems are relevant. In the NRP, the SGS algorithm has been competitive with the state-of-the-art algorithms for this problem, finding the set of Pareto optimal solutions. These results are achieved even though the problem is multi-objective, whereas SGS is a mono-objective algorithm. In the TSMP, SGS is able to provide better solutions than four heuristics specially designed for this problem. In particular, the results obtained by SGS are superior than the results obtained by GREEDY_{E}, which is the previous state-of-the-art algorithm for solving such problem.

It is interesting to note that the quality of the numerical results obtained by SGS is quite independent of the execution platform used, i.e., the CPU and GPU implementations present similar results for all the problems studied as the behaviour of the algorithm is basically the same (if there were significant differences, they could be produced by the use of different random number generators and/or by the parallelism). As a consequence, and despite being conceived for its execution in GPU, the CPU implementation of SGS is also an interesting alternative as an optimization algorithm.

In the third place, a general strategy for the thread organization of the GPU implementation has been conceived based on assigning the operations to a whole block, and processing the solutions components in chunks and cooperatively among the threads of the block. This strategy is aligned with the guidelines for efficient GPU implementation [91, 92]. The GPU implementation of SGS has achieved a high performance, obtaining large runtime reductions with respect to the CPU counterparts for solutions with similar quality. The strategy for thread organization has not only allowed SGS to achieve a high performance, but also to the two other EAs implemented on GPU. However, the GPU implementation of SGS is the best performing algorithm among the EAs studied, having systematically the shortest runtime for all the instances of all the problems considered. This can be explained because SGS has a more regular computation (all the cells do basically the same computation), whereas in the other EAs there are other aspects that drop the efficiency down, such as the selection operation. Moreover, the GPU implementation of SGS has also shown an excellent scalability behavior when solving instances of increasing size.

Throughout this thesis, two other interesting ideas for GPU-based implementations have been evaluated that deserve to be highlighted. First, when the fitness evaluation involves irregular computation patterns that are not suitable for GPU, this evaluation can be transformed into a matrix-matrix multiplication operation. This operation has a more structured pattern of computation that is well suited for the GPU. Also, there are available linear algebra libraries able to compute this operation efficiently like CUBLAS [89]. Second, the use of bit packing in non-boolean data types for storing the population on the memory can additionally reduce the execution time of GPU implementations up to 50%. This approach helps to avoid wasting memory, but it is more difficult to implement as it involves the use of bitwise operations.

Finally, the theoretical analysis has identified the issue in which there are cells in the grids where the two incoming solutions are direct descendants of a pair of solutions that have already been mated in another cell of the grid. The experimental evaluation of a new grid, which prevents mating descendants of pairs of solutions that have already been mated, shows that the use of this grid benefits the search engine of SGS. Moreover, this grid guarantees a better diffusion of highly fitted genetic material, thus producing a higher diversity on the cells that makes SGS to be able to find better solutions.

6.2 Open Research Lines and Future Work

This thesis have fulfilled the general initial goal of designing, implementing and evaluating a new parallel metaheuristic algorithm and/or new parallelism model for GPUs. During this work several issues has been identified, which could become lines of future work of this thesis. The following list details some of the research lines that deserve further study.

Further study of the TSMP and similar problems: In the TSMP, there are available heuristics with acceptable numerical results and short execution times. The approach followed thus far for addressing the TSMP can be considered purist as SGS does not incorporate any knowledge of the problem in its operation. It is interesting to study the effect of initializing some solutions of the population of SGS with these heuristics. SGS can also incorporate a local search mechanism designed from such heuristics. This could help to speed up the search of the SGS, specially considering the resolution of larger instances, and even lead to reach new best-known solutions.

In addition to this, there are other optimization problems from software testing that are similar to the TSMP, for instance the Test Case Selection Problem [121]. This problem consists in choosing a subset of the test suite based on which test cases are relevant for testing the changes between the previous and the current version of a piece of software. Due to the matching points between this problem and TSMP and the excellent results obtained by SGS for the TSMP, it is also attractive to address this problem with SGS.

Study of additional problems with SGS: The results of SGS have already shown to be promising, but it is desirable to extend the analysis by solving additional problems with SGS to increase the existing evidence of the benefits of this line of research. This can also serve to obtain more feedback on SGS and thus revealing potential shortcomings in the algorithm.

Enhancing the design of the search engine of SGS: The theoretical study on the flow of solutions has provided valuable information on the behavior of the algorithm, in particular with regard to the diversity of the solutions in the cells. Even though the numerical results obtained by SGS are noteworthy, it would be interesting to incorporate a mechanism to detect a loss of diversity in the solutions managed by the algorithm and thus enabling SGS to react by introducing new genetic material (e.g., generating completely new solutions throughout the grid). Using this type of ideas can potentially help to speed up the search of the SGS and even enhance the search engine of the algorithm, leading to better solutions.

Other studies can be devised to analyze several aspects of SGS that have not been studied in depth during this work, for instance, the use of elitism on the cells of the grid. It is interesting to understand how often elitism is used in each cell. If there is an overuse of elitism, i.e., many new solutions are discarded, a memory mechanism could be incorporated into SGS so that each cell stores its best generated solutions and use them afterwards to reduce or directly remove elitism.

Extension of SGS algorithm to other encodings: The general idea of SGS is not tied to a particular representation, however the use of other representations can constitute a new challenge for the GPU implementation of SGS: real encoding could present performance problems if the problem requires to use the double precision floating point data type as the performance with this precision is much slower than single precision. Also, it is not trivial to use the idea of processing a solution by chunks for the permutation encoding.

Design and development of a library with generic components for GPU implementations of metaheuristics: The development of a framework for implementing metaheuristics on GPU has some practical difficulties since the implementation of the fitness function on the GPU has to be provided by the user of the framework. The implementation of this function on the GPU is critical and it could compromise the overall performance of the algorithm. This issue partly explains why there are few metaheuristic frameworks that support GPU implementations. However, there are components that can be abstracted in a library for providing generic operations that could be useful for other efforts. Two of such components can be the use of bit packing for the binary encoding and the use of the matrix-matrix multiplication for the fitness evaluation.

It is interesting to note that the use of the matrix-matrix multiplication is not compatible with the use of bit packing. The libraries that provide efficient matrix-matrix multiplication store data using float or double data types. As a consequence, it is not possible to directly combine in a GPU implementation the use of bit packing in non-boolean data types for storing the population on the memory and the use of the matrix-matrix multiplication for fitness evaluation. For this reason, it is interesting to conduct an experimental study to determine in which cases is better to use one idea or the other. In addition to this, a matrix-matrix multiplication that support working with bit packing in non-boolean data types can also be developed as a future work.

References

[1] Alba, E., editor (2005). *Parallel Metaheuristics: A New Class of Algorithms*. Wiley.

[2] Alba, E. and Dorronsorso, B. (2008). *Cellular Genetic Algorithms*. Springer.

[3] Alba, E. and Nebro, A. J. (2005). New Technologies in Parallelism. In Alba, E., editor, *Parallel Metaheuristics*, pages 63–78. Wiley.

[4] Alba, E. and Tomassini, M. (2002). Parallelism and evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 6(5):443 – 462.

[5] Alba, E. and Vidal, P. (2011). Systolic optimization on GPU platforms. In *13th International Conference on Computer Aided Systems Theory (EUROCAST 2011).*

[6] Arora, S. and Barak, B. (2009). *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition.

[7] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. (2009). A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67.

[8] Bagnall, A., Rayward-Smith, V., and Whittley, I. (2001). The next release problem. *Information and Software Technology*, 43(14):883 – 890.

[9] Benner, P., Ezzatti, P., Kressner, D., Quintana-Ortí, E. S., and Remón, A. (2011). A mixed-precision algorithm for the solution of Lyapunov equations on hybrid CPU-GPU platforms. *Parallel Computing*, 37(8):439–450.

[10] Beyer, H.-G. and Schwefel, H.-P. (2002). Evolution strategies - a comprehensive introduction. 1(1):3–52.

[11] Bland, I. M., Megson, G., et al. (1996). Implementing a generic systolic array for genetic algorithms. In *Proceedings of the First Online Workshop on Soft Computing (WSC1)*, pages 268–273.

[12] Bland, I. M. and Megson, G. M. (1996). Systolic random number generation for genetic algorithms. *Electronics Letters*, 32(12):1069–1070.

[13] Bland, I. M. and Megson, G. M. (1998). The systolic array genetic algorithm, an example of systolic arrays as a reconfigurable design methodology. In *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*, pages 260–261.

[14] Blum, C. and Roli, A. (2003). Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Computing Surveys*, 35(3):268–308.

[15] Bojanczyk, A., Brent, R. P., and Kung, H. (1984). Numerically stable solution of dense systems of linear equations using mesh-connected processors. *SIAM journal on scientific and statistical computing*, 5(1):95–104.

[16] Brent, R. P. and Kung, H. T. (1984). Systolic VLSI arrays for polynomial GCD computation. *IEEE Trans. Computers*, 33(8):731–736.

[17] Breß, S., Heimel, M., Siegmund, N., Bellatreche, L., and Saake, G. (2014). *GPU-Accelerated Database Systems: Survey and Open Challenges*, pages 1–35. Springer Berlin Heidelberg, Berlin, Heidelberg.

[18] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. (2004a). Brook for GPUs: Stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA. ACM.

[19] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. (2004b). Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786.

[20] Cantu-Paz, E. (2000). *Efficient and Accurate Parallel Genetic Algorithms*. Kluwer Academic Publishers.

[21] Cecilia, J. M., García, J. M., Nisbet, A., Amos, M., and Ujaldón, M. (2013). Enhancing data parallelism for ant colony optimization on GPUs. *Journal of Parallel and Distributed Computing*, 73(1):42 – 51.

[22] Cecilia, J. M., García, J. M., Ujaldón, M., Nisbet, A., and Amos, M. (2011). Parallelization strategies for ant colony optimisation on GPUs. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Workshop Proceedings*, pages 339–346.

[23] Chan, H. and Mazumder, P. (1995). A systolic architecture for high speed hypergraph partitioning using a genetic algorithm. In *Progress in Evolutionary Computation*, volume 956 of *Lecture Notes in Computer Science*, pages 109–126. Springer Berlin / Heidelberg.

[24] Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM.

[25] Darema, F. (2001). *The SPMD Model: Past, Present and Future*, pages 1–1. Springer Berlin Heidelberg, Berlin, Heidelberg.

[26] Darema, F., George, D., Norton, V., and Pfister, G. (1988). A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, 7(1):11 – 24.

[27] Darwin, C. (1859). *On the Origin of Species by Means of Natural Selection or the Preservation of Favored Races in the Struggle for Life*. John Murray, London.

[28] De Jong, K. A. and Spears, W. M. (1992). A formal analysis of the role of multi-point crossover in genetic algorithms. *Annals of Mathematics and Artificial Intelligence*, 5(1):1–26.

[29] Deb, K. and Goldberg, D. E. (1992). Analyzing Deception in Trap Functions. In *Foundations of Genetic Algorithms*, pages 93–108.

[30] Deb, K. and Goldberg, D. E. (1994). Sufficient conditions for deceptive and easy binary functions. *Annals of Mathematics and Artificial Intelligence*, 10(4):385–408.

[31] Derrac, J., García, S., Molina, D., and Herrera, F. (2011). A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. *Swarm and Evolutionary Computation*, 1(1):3–18.

[32] Dorigo, M. and Stützle, T. (2004). *Ant Colony Optimization*. MIT Press, Cambridge, MA, USA.

[33] Ezzatti, P., Quintana-Ortí, E. S., and Remón, A. (2011). Using graphics processors to accelerate the computation of the matrix inverse. *The Journal of Supercomputing*, 58(3):429–437.

[34] Feo, T. and Resende, M. (1995). Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, 6:109–133.

[35] Fernando, R. and Kilgard, M. J. (2003). *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[36] Flynn, M. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960.

[37] Fogel, L., Owens, A., and Walsh, M. (1966). *Artificial Intelligence Through Simulated Evolution*. John Wiley and Sons, New York.

[38] Furber, S. (2000). *ARM System-on-Chip Architecture*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition.

[39] Garey, M. and Johnson, D. (1979). *Computers and Intractability: A Guide to the Theory of NP–Completeness*. Freeman, San Francisco.

[40] Gaster, B., Howes, L., Kaeli, D., Mistry, P., and Schaa, D. (2012). *Heterogeneous computing with OpenCL, 2nd Edition*. Morgan Kaufmann.

[41] Gendreau, M. and Potvin, J.-Y. (2010). Tabu search. In Gendreau, M. and Potvin, J.-Y., editors, *Handbook of metaheuristics*, pages 41–60. Springer.

[42] Gentleman, W. M. and Kung, H. (1982). Matrix triangularization by systolic arrays. In *25th Annual Technical Symposium*, pages 19–26. International Society for Optics and Photonics.

[43] Glover, F. (1986). Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers & Operations Research*, 13(5):533–549.

[44] Goldberg, D. (1989a). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

[45] Goldberg, D., Deb, K., and Horn, J. (1992). Massively multimodality, deception and genetic algorithms. In *Proceedings of the International Conference on Parallel Problem Solving from Nature II (PPSNII)*, pages 37–46.

[46] Goldberg, D. E. (1989b). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

[47] Guibas, L., Kung, H., and Thompson, C. (1979). Direct VLSI implementation of combinatorial algorithms. In *Proc. Conf. Very Large Scale Integration: Architecture, Design, Fabrication*, pages 509–525. California Institute of Technology.

[48] Guo, L., Guo, C., Thomas, D., and Luk, W. (2015). Pipelined genetic propagation. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 103–110.

[49] Guyton, A. C. and Hall, J. E. (2006). *Textbook of medical physiology*. Elsevier Saunders, 11 edition.

[50] Harding, S. and Banzhaf, W. (2011). Implementing cartesian genetic programming classifiers on graphics processing units using GPU.NET. In *13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Companion Material*, pages 463–470.

[51] Harman, M., Mansouri, S. A., and Zhang, Y. (2012). Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11:1–11:61.

[52] Hennessy, J. and Patterson, D. (2011). *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann.

[53] Herlihy, M. and Shavit, N. (2012). *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[54] Herrera, F., Lozano, M., and Verdegay, J. L. (1998). Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis. *Artif. Intell. Rev.*, 12(4):265–319.

[55] Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA.

[56] Hoos, H. H. and Stützle, T. (2004). *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[57] Jeffers, J. and Reinders, J. (2013). *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[58] Johnson, K. T., Hurson, A. R., and Shirazi, B. (1993). General-purpose systolic arrays. *Computer*, 26(11):20–31.

[59] Kapasi, U. J., Rixner, S., Dally, W. J., Khailany, B., Ahn, J. H., Mattson, P. R., and Owens, J. D. (2003). Programmable stream processors. *IEEE Computer*, 36(8):54–62.

[60] Kennedy, J. and Eberhart, R. C. (1995). Particle swarm optimization. In *Proceedings of the 1995 IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948.

[61] Khronos Group (2017). OpenGL Website. http://www.opengl.org. Accessed: February, 2017.

[62] Khronos OpenCL Working Group (2015). *The OpenCL Specification, version 2.0*. Editors: Lee Howes and Aaftab Munshi.

[63] Kirk, D. and Hwu, W. (2012). *Programming Massively Parallel Processors, Second Edition: A Hands-on Approach*. Morgan Kaufmann.

[64] Kirkpatrick, S., Gelatt, C., and Vecchi, M. (1983). Optimization by Simulated Annealing. *Science*, 220:671–680.

[65] Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

[66] Krömer, P., Platoš, J., and Snášel, V. (2014). Nature-inspired meta-heuristics on modern GPUs: State of the art and brief survey of selected algorithms. *Int. J. Parallel Program.*, 42(5):681–709.

[67] Kung, H. T. (1982). Why systolic architectures? *Computer*, 15(1):37 – 46.

[68] Kung, H. T. and Lehman, P. L. (1980). Systolic (VLSI) arrays for relational database operations. In Chen, P. P. and Sprowls, R. C., editors, *Proceedings of the 1980 ACM SIGMOD International Conference on Management of Data, Santa Monica, California, May 14-16, 1980.*, pages 105–116. ACM Press.

[69] Kung, H. T. and Leiserson, C. E. (1978). Systolic arrays (for VLSI). In *Sparse Matrix Proceedings*, pages 256 – 282.

[70] Kung, S. Y. (1987). *VLSI Array Processors*. Prentice-Hall, Inc.

[71] Langdon, W. B. (2011). Graphics processing units and genetic programming: an overview. *Soft Computing*, 15(8):1657 – 1669.

[72] Langdon, W. B. and Banzhaf, W. (2008). A SIMD interpreter for genetic programming on GPU graphics cards. In *Genetic Programming, 11th European Conference, EuroGP 2008, Naples, Italy, March 26-28, 2008. Proceedings*, volume 4971 of *Lecture Notes in Computer Science*, pages 73–85. Springer.

[73] Lewis, T. E. and Magoulas, G. D. (2009). Strategies to minimise the total run time of cyclic graph based genetic programming with GPUs. In *Genetic and Evolutionary Computation Conference, GECCO 2009*, pages 1379–1386.

[74] Libby, P., Bonow, R., Mann, D., and Zipes, D. (2007). *Braunwald's Heart Disease: A Textbook of Cardiovascular Medicine*. Elsevier Health Sciences.

[75] Lourenço, H., Martin, O., and Stützle, T. (2002). Iterated Local Search. In Glover, F. and Kochenberger, G., editors, *Handbook of Metaheuristics*, pages 321–353. Kluwer Academic Publisher.

[76] Luque, G. and Alba, E. (2011). *Parallel Genetic Algorithms: Theory and Real World Applications*, volume 367 of *Studies in Computational Intelligence*. Springer.

[77] Luque, G., Alba, E., and Dorronsoro, B. (2005). *Parallel Metaheuristics: A New Class of Algorithms*, chapter 5. Parallel Genetic Algorithm, pages 107–126. Wiley Series on Parallel and Distributed Computing. Wiley.

[78] Maitre, O., Krüger, F., Querry, S., Lachiche, N., and Collet, P. (2012). EASEA: specification and execution of evolutionary algorithms on GPGPU. *Soft Computing*, 16(2):261–279.

[79] Maitre, O., Lachiche, N., and Collet, P. (2010). Fast evaluation of GP trees on GPGPU by optimizing hardware scheduling. In *Genetic Programming, 13th European Conference, EuroGP 2010, Istanbul, Turkey, April 7-9, 2010. Proceedings*, volume 6021 of *Lecture Notes in Computer Science*, pages 301–312. Springer.

[80] McCool, M., Reinders, J., and Robison, A. (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[81] Mead, C. and Conway, L. (1979). *Introduction to VLSI Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[82] Megson, G. and Bland, I. (1998). Synthesis of a systolic array genetic algorithm. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998*, pages 316 –320.

[83] Megson, G. M. and Bland, I. M. (1997). Generic systolic array for genetic algorithms. In *Computers and Digital Techniques, IEE Proceedings-*, volume 144, pages 107–119. IET.

[84] Mladenović, N. and Hansen, P. (1997). Variable Neighborhood Search. *Computers & Operations Research*, 24(11):1097–1100.

[85] Nikolaev, A. G. and Jacobson, S. H. (2010). Simulated annealing. In Gendreau, M. and Potvin, J.-Y., editors, *Handbook of metaheuristics*, pages 1–39. Springer.

[86] Nvidia Corporation (2012). *NVIDIA GeForce GTX 680 Whitepaper*. Nvidia Corporation.

[87] Nvidia Corporation (2016a). *NVIDIA GeForce GTX 1080 Whitepaper*. Nvidia Corporation.

[88] Nvidia Corporation (2016b). *NVIDIA Tesla P100 Whitepaper*. Nvidia Corporation.

[89] Nvidia Corporation (2017a). *CUDA Toolkit 8.0 CUBLAS Library User Guide*. Nvidia Corporation.

[90] Nvidia Corporation (2017b). *CUDA Toolkit 8.0 CURAND Library Programming Guide*. Nvidia Corporation.

[91] Nvidia Corporation (2017c). *NVIDIA CUDA C Best Practices Guide Version 8.0*. Nvidia Corporation.

[92] Nvidia Corporation (2017d). *NVIDIA CUDA C Programming Guide Version 8.0*. Nvidia Corporation.

[93] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A., and Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113.

[94] Pedemonte, M., Alba, E., and Luna, F. (2011). Bitwise operations for GPU implementation of genetic algorithms. In *Genetic and Evolutionary Computation Conference, GECCO'11 - Companion Publication*, pages 439 – 446.

[95] Pedemonte, M., Alba, E., and Luna, F. (2012). Towards the design of systolic genetic search. In *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1778–1786. IEEE Computer Society.

[96] Pedemonte, M., Luna, F., and Alba, E. (2013). New ideas in parallel metaheuristics on GPU: Systolic genetic search. In Tsutsui, S. and Collet, P., editors, *Massively Parallel Evolutionary Computation on GPGPUs*, Natural Computing Series, chapter 10, pages 203–225. Springer.

[97] Pedemonte, M., Luna, F., and Alba, E. (2014). Systolic genetic search for software engineering: The test suite minimization case. In Esparcia-Alcázar, A. I. and Mora, A. M., editors, *Applications of Evolutionary Computation - 17th European Conference, EvoApplications 2014, Granada, Spain, April 23-25, 2014, Revised Selected Papers*, volume 8602 of *Lecture Notes in Computer Science*, pages 678–689. Springer.

[98] Pedemonte, M., Luna, F., and Alba, E. (2015). Systolic genetic search, a systolic computing-based metaheuristic. *Soft Computing*, 19(7):1779–1801.

[99] Pedemonte, M., Luna, F., and Alba, E. (2016). A systolic genetic search for reducing the execution cost of regression testing. *Applied Soft Computing*, 49:1145 – 1161.

[100] Pedemonte, M., Luna, F., and Alba, E. (2018). A theoretical and empirical study of the trajectories of solutions on the grid of systolic genetic search. *Information Sciences*, 445-446:97 – 117.

[101] Pisinger, D. (1999). Core problems in knapsack algorithms. *Operations Research*, 47:570 – 575.

[102] Poli, R., Langdon, W. B., and McPhee, N. F. (2008). *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd.

[103] Rardin, R. L. and Uzsoy, R. (2001). Experimental evaluation of heuristic optimization algorithms: A tutorial. *Journal of Heuristics*, 7(3):261–304.

[104] Rechenberg, I. (1973). *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Fromman-Holzboog, Stuttgart, Germany.

[105] Schrijver, A. (2005). On the history of combinatorial optimization (till 1960). *Handbooks in Operations Research and Management Science*, 12:1 – 68.

[106] Shao, S., Guo, L., Guo, C., Chau, T., Thomas, D., Luk, W., and Weston, S. (2015). Recursive pipelined genetic propagation for bilevel optimisation. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 1–6.

[107] Sheskin, D. J. (2011). *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman and Hall/CRC, fifth edition edition.

[108] Silva, J. P., Hagopian, J., Burdiat, M., Dufrechou, E., Pedemonte, M., Gutiérrez, A., Cazes, G., and Ezzatti, P. (2014). Another step to the full GPU implementation of the weather research and forecasting model. *The Journal of Supercomputing*, 70(2):746–755.

[109] Soca, N., Blengio, J. L., Pedemonte, M., and Ezzatti, P. (2010). PUGACE, a cellular evolutionary algorithm framework on GPUs. In *IEEE Congress on Evolutionary Computation*, pages 1–8.

[110] Spears, W. M. and Jong, K. A. D. (1990). An analysis of multi-point crossover. In Rawlins, G. J. E., editor, *Proceedings of the First Workshop on Foundations of Genetic Algorithms. Bloomington Campus, Indiana, USA, July 15-18 1990.*, pages 301–315. Morgan Kaufmann.

[111] Talbi, E.-G. (2009). *Metaheuristics: From Design to Implementation*. Wiley Publishing.

[112] Tsutsui, S. and Fujimoto, N. (2011). Fast QAP solving by ACO with 2-opt local search on a GPU. In *2011 IEEE Congress of Evolutionary Computation, CEC 2011*, pages 812–819.

[113] Veronese, L. D. P. and Krohling, R. A. (2010). Differential evolution algorithm on the GPU with C-CUDA. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010*, pages 1–7.

[114] Vidal, P. and Alba, E. (2010). Cellular genetic algorithm on graphic processing units. In *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*, pages 223 – 232.

[115] Vidal, P., Alba, E., and Luna, F. (2016). Solving optimization problems using a hybrid systolic search on GPU plus CPU. *Soft Computing*, pages 1–19.

[116] Vidal, P., Luna, F., and Alba, E. (2014). Systolic neighborhood search on graphics processing units. *Soft Computing*, 18(1):125–142.

[117] Weiser, U. and Davis, A. (1981). *A Wavefront Notation Tool for VLSI Array Design*, pages 226–234. Springer Berlin Heidelberg, Berlin, Heidelberg.

[118] Whitley, D. and wook Yoo, N. (1995). Modeling simple genetic algorithms for permutation problems. In *in Foundations of Genetic Algorithms*, pages 163–184. Morgan Kaufmann.

[119] Whitley, L. D. (1990). Fundamental Principles of Deception in Genetic Search. In *Foundations of Genetic Algorithms*, pages 221–241.

[120] Yang, S. (2006). On the design of diploid genetic algorithms for problem optimization in dynamic environments. In *2006 IEEE International Conference on Evolutionary Computation*, pages 1362–1369.

[121] Yoo, S. and Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120.

[122] Zhang, S. and He, Z. (2009). Implementation of parallel genetic algorithm based on CUDA. In *ISICA 2009*, LNCS 5821, pages 24 – 30.

[123] Zheng, L., Lu, Y., Guo, M., Guo, S., and Xu, C. (2014). Architecture-based design and optimization of genetic algorithms on multi- and many-core systems. *Future Generation Comp. Syst.*, 38:75–91.

[124] Zhou, Y. and Tan, Y. (2009). GPU-based parallel particle swarm optimization. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2009*, pages 1493–1500.

Appendix A

Systolic Genetic Search, a Systolic Computing-Based Metaheuristic

Soft Comput
DOI 10.1007/s00500-014-1363-0

METHODOLOGIES AND APPLICATION

Systolic genetic search, a systolic computing-based metaheuristic

Martín Pedemonte · Francisco Luna · Enrique Alba

Abstract In this paper, we propose a new parallel optimization algorithm that combines ideas from the fields of metaheuristics and Systolic Computing. The algorithm, called Systolic Genetic Search (SGS), is designed to explicitly exploit the high degree of parallelism available in modern Graphics Processing Unit (GPU) architectures. In SGS, solutions circulate synchronously through a grid of processing cells, which apply adapted evolutionary operators on their inputs to compute their outputs that are then ejected from the cells and continue moving through the grid. Four different variants of SGS are experimentally studied for solving two classical benchmarking problems and a real-world application. An extensive experimental analysis, which considered several instances for each problem, shows that three of the SGS variants designed are highly effective since they can obtain the optimal solution in almost every execution for the instances and problems studied, as well as they outperform a Random Search (sanity check) and two Genetic Algorithms. The parallel implementation on GPU of the proposed algorithm has achieved a high performance obtaining runtime reductions from the sequential implementation that, depending on the instance considered, can arrive to around a hundred times, and have also exhibited a good scalability behavior when solving highly dimensional problem instances.

Keywords Systolic genetic search · Evolutionary algorithms · Systolic computing · Parallel computing · Graphics processing units · CUDA · GPGPU

Communicated by V. Loia.

M. Pedemonte ( )
Instituto de Computación, Facultad de Ingeniería,
Universidad de la República, Julio Herrera y Reissig 565,
11300 Montevideo, Uruguay
e-mail: mpedemon@fing.edu.uy

F. Luna
Depto. de Ingeniería de Sistemas Informáticos y Telemáticos,
Centro Universitario de Mérida, Universidad de Extremadura,
Santa Teresa de Jornet, 28, 06800 Mérida, Spain
e-mail: fluna@unex.es

E. Alba
Departamento de Lenguajes y Ciencias de la Computación,
Universidad de Málaga, E.T.S. Ingeniería Informática,
Campus de Teatinos, 29071 Málaga, Spain
e-mail: eat@lcc.uma.es

1 Introduction

In the last ten years, computing platforms have undergone revolutionary changes (Hennessy and Patterson 2011). Parallel hardware is no longer an infrastructure reserved for a few research laboratories, but it is widely available for the general public. On one hand, the architecture of CPU processors has changed, being now multi-core (a single computing unit composed of at least two independent processors). As a consequence, modern desktop computers are at least dual-core (the more common hardware configuration) and even hexa-core. On the other hand, the parallel hardware that can be used for computation has diversified notably. Nowadays, it is possible to use devices like multi-core processors with ARM architecture (Furber 2000), which have become massively available in smart cell phones and tablet computers, as well as Graphics Processing Units (GPUs) as general-purpose parallel platforms (Owens et al. 2007).

The number of cores available in modern hardware is growing steadily and will undoubtedly continue to do so in the foreseeable future. For instance, Nvidia has launched

Published online: 19 July 2014

its new generation of GPUs, Kepler (Nvidia Corporation 2012d), with up to 2,688 CUDA cores at 732 MHz and a single precision floating point peak performance of 3.95 TFlops; while Intel has released the Xeon Phi coprocessor (Intel Corporation 2013a,b), with up to 61 cores operating at least 1 GHz and a single precision floating point peak performance of 2.15 TFlops. As a consequence, the design of parallel algorithms able to exploit the new capabilities available in modern hardware is indispensable.

In this context, in which remarkable changes are taking place in the devices used for computing and device capabilities will stay growing, the design of new parallel algorithms that profit from them is certainly an interesting, promising research line. This is specially relevant in the field of metaheuristics (Blum and Roli 2003) and their parallelization (Alba 2005). Unlike exact methods, metaheuristics are stochastic algorithms which are able to provide optimization problems with very accurate solutions in a reasonable amount of time. However, as the problem instances in today's research are becoming very large, even metaheuristics may be highly computationally expensive. This is where parallelism comes out as an actual, reliable strategy to speed up the search of those kind of optimizers. The truly interesting point is that parallel metaheuristics do not only allow the runtime of the algorithms to be reduced, but also allow to improve the quality of results obtained by traditional sequential algorithms due to their (often new) enhanced search engine (Alba 2005). As a consequence, research on this topic has grown substantially in the last years, motivated by the excellent results obtained in their application to the resolution of problems in search, optimization, and machine learning.

In particular, the use of GPUs has represented an inspiring domain for the research in parallel metaheuristics, experiencing a tremendous growth in the last five years. This growth has been based on its wide availability, low economic cost, and inherent parallel architecture, and also on the emergence of general-purpose programming languages, such as CUDA (Kirk and Hwu 2012) and OpenCL (Gaster et al. 2012).

The first works on parallel metaheuristics on GPUs have gone in the direction of taking a standard existing family of algorithms and porting them to this new kind of hardware (Langdon 2011). Thus, many results show the time savings of running master–slave (Maitre et al. 2012), distributed (Zhang and He 2009), and cellular (Soca et al. 2010; Vidal and Alba 2010a) metaheuristics on GPU (mainly Genetic Algorithms (Maitre et al. 2012; Pedemonte et al. 2011) and Genetic Programming (Harding and Banzhaf 2011; Langdon and Banzhaf 2008; Lewis and Magoulas 2009) but also other types of techniques like Ant Colony Optimization (Cecilia et al. 2011; Tsutsui and Fujimoto 2011), Differential Evolution (Veronese and Krohling 2010), Particle Swarm Optimization (Zhou and Tan 2009), etc.).

A different approach, followed in this paper, lies in proposing and designing new techniques that explicitly exploit the high degree of parallelism available in modern GPU architectures. Following this course of action, two new optimization algorithms (Systolic Neighborhood Search (Alba and Vidal 2011; Vidal et al. 2013) and Systolic Genetic Search (Pedemonte et al. 2012, 2013)) have recently been proposed that are based on combining ideas from the fields of metaheuristics and Systolic Computing. The concept *Systolic Computing* was coined at Carnegie-Mellon University by Kung (1982), and Kung and Leiserson (1978). The basic idea of this concept focuses on creating a network of different simple processors or operations that rhythmically compute and pass data through the system. Systolic computation offers several advantages, including simplicity, modularity, and repeatability of operations. This kind of architecture also offers transparent, understandable and manageable, but still quite powerful parallelism. In Systolic Genetic Search (SGS) solutions circulate synchronously through a grid of cells. When two solutions meet in a cell, adapted evolutionary operators are applied to generate new solutions that continue moving through the grid. SGS has shown its potential for tackling the Knapsack Problem finding optimal solutions in short execution times in Pedemonte et al. (2012, 2013).

The goal of the present work is to characterize the general SGS optimization algorithm and study four novel variants that follow the premises of this general optimization algorithm. An exhaustive experimental evaluation has been undertaken to provide the reader with insights on both the search capabilities of the SGS algorithms and their parallel performance when deployed on a GPU card. The results have shown that three out of the four systolic algorithms devised are highly effective as they are able to reach the optimal solution in almost every execution for the instances and problems studied, outperforming the other algorithms involved in the experiment (namely, Random Search, and two Genetic Algorithms). The testbed is composed of different instances of two classical benchmark problems, Knapsack Problem (Pisinger 1999) and Massively Multimodal Deceptive Problem (Goldberg et al. 1992), and a real-world application, the Next Release Problem (Bagnall et al. 2001). On the other hand, the parallel implementation on GPU of SGS has achieved a high performance obtaining runtime reductions from the sequential implementation that, depending on the problem and the instance considered, can scale up to around a hundred times.

Finally, it should be highlighted that parallel SGS also exhibits a good scalability behavior when solving high-dimensional problem instances.

This article is organized as follows. The next section introduces the SGS algorithm and the four different variants studied. Section 3 describes the implementation of the SGS on a GPU. Then, Sect. 4 presents the experimental study considering the three different problems aforementioned. Section 5 shows how our approach differs from the existing population models for evolutionary algorithms. Finally, in Sect. 6, we outline the conclusions of this work and suggest future research directions.

2 Systolic genetic search

Systolic computing-based metaheuristics, as well as Systolic Computing, are inspired by the same biological phenomenon. The idea is to mimic the systolic contraction of the heart that makes it possible to inject blood back on the body rhythmically according to the metabolic needs of the tissues. This biological phenomenon is briefly described next.

The cardiovascular system consists of the heart, that is responsible for pumping blood with each beat, and a specialized conduction system composed of arteries, which transport and distribute blood from the heart to the body, and the veins that transport the blood back to the heart. The cardiovascular system can be seen as two pumps that work in parallel; the first pump corresponds to the right heart, which receives deoxygenated blood from the tissues and sends it to the lungs to be oxygenated (pulmonary circulation). On the other hand, the second pump corresponds to the left heart that receives oxygenated blood from the lungs and sends it to all tissues to distribute oxygen to the different parenchyma (systemic circulation). The human heart pumps through the body more than 6,000 l of blood daily (Guyton and Hall 2006; Libby et al. 2007).

In each cardiac cycle, the heart first relaxes to refill with circulating blood (this phase is known as diastole) and then it contracts (this phase is known as systole), increasing the pressure inside the cavities. As a consequence, the heart ejects blood into the arterial system. Due to the systolic contraction, the blood is ejected from the heart with a regular cadence or rhythmically according to the metabolic needs of the tissues. If the cardiac cycle is regular, i.e., there are no pathological situations (called cardiac dysrhythmia), we can define as a normal heart rate for an adult a value between 60 and 100 cycles/min (Guyton and Hall 2006; Libby et al. 2007).

In Systolic Computing-based metaheuristics, solutions flow across processing units (cells) according to a synchronous, structured plan. When two tentative solutions (or one single solution in Systolic Neighborhood Search algorithm) meet in a cell, operators are applied to obtain new solutions that continue moving across the processing units. In this way, solutions are refined again and again by simple low complexity search operators.

In the leading work on Systolic Computing-based metaheuristics, the Systolic Neighborhood Search algorithm has been proposed that is based on using a local search as the working operation in cells (Alba and Vidal 2011; Vidal et al. 2013). In subsequent works, we have explored a more sophisticated approach that involves more diverse operations: the Systolic Genetic Search algorithm (Pedemonte et al. 2012, 2013). Closely related works are further analyzed in Sect. 5.

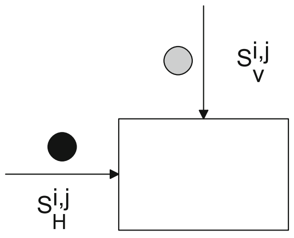
The rest of this section is structured as follows. First, in the next subsection the SGS algorithm is described. Then, the four different instantiations or flavors of SGS that are used in the experimental evaluation are introduced.

2.1 Systolic genetic search algorithm

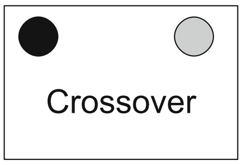
In a SGS algorithm, the solutions are synchronously pumped through a bidimensional grid of cells. At each step of SGS, two solutions enter each cell, one from the horizontal data stream ($S_H^{i,j}$) and one from the vertical data stream ($S_V^{i,j}$), as it is shown in Fig. 1a. Then, adapted evolutionary/genetic operators (crossover shown in Fig. 1b and mutation shown in Fig. 1c) are applied to generate two new solutions. Later, the cell uses elitism (shown in Fig. 1d) to determine which solutions continue moving through the grid, one through the horizontal data stream ($S_H^{i,j+1}$) and one through the vertical data stream ($S_V^{i+1,j}$), as it is shown in Fig. 1e.

The pseudocode of SGS is presented in Algorithm 1. At the very beginning of the operation, each cell generates two random solutions which are aimed at moving horizontally and vertically, respectively. Then, it applies the basic evolutionary search operators (crossover and mutation) but to different, preprogrammed fixed positions of the tentative solutions that circulate throughout the grid. This is the major contribution of SGS: it performs a stochastic yet structured exploration of the search space. The cells use elitism to pass on the best solution (between the incoming solution and the newly generated one by the genetic operators) to the next cells. The incorporation of elitism is critical, as there is no global selection process like in standard Evolutionary Algorithms (EAs). Each cell sends the outgoing solutions to the next cells of the data streams, which have been computed previously.

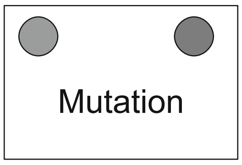
We want to remark that the idea of the SGS algorithm can be adapted to any solution representation and any particular operator. In this work, we address binary problems, for this reason we encoded the solutions as binary strings, and use bit-flip mutation and two-point crossover as evolutionary search operators. In this case (binary representation), the positions in which operators are applied in each cell are defined by considering the location of the cell in the grid, thus avoiding the generation of random numbers during the execution. Some key aspects of the algorithm such as the size of the grid and the calculation of the crossover points and the mutation point are discussed next.



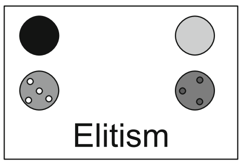
(a) Stage 1: receiving inputs.



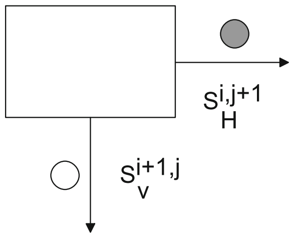
(b) Stage 2: internal operation.



(c) Stage 3: internal operation.



(d) Stage 4: internal operation.



(e) Stage 5: pumping outputs.

Fig. 1 SGS processing at cell (i, j)

2.1.1 Size of the grid

The length and width of the grid considered, respectively, as the number of cells in a row and in a column, should allow the algorithm to achieve a good exploration, but without increasing the population size up to values that compromise performance. To generate all possible mutation point values at each single row and considering that each cell uses a different mutation point value, the grid length is l (the length of the tentative solutions, i.e., the size of the problem instance). As a consequence, each cell in a given row modifies a different position of the arriving solutions.

If a similar strategy would have been used for the columns (generate all possible mutation point values at each single column), the natural value for the width of the grid is also l. However, that would lead SGS to use a population with $2 \times l \times l$ (2 solutions per cell) for solving problem instances of size l. For this reason, and to keep the total number of solutions of the population within an affordable value, the width of the grid has been reduced to $\tau = \lceil \lg l \rceil$. Therefore, the number of solutions of the population is $2 \times l \times \tau$ (2 solutions per cell).

Algorithm 1 Systolic Genetic Search

```
1: for all $c$ Cell do
2:     $c.h$ =generateRandomSolution();
3:     $c.v$ =generateRandomSolution();
4: end for
5: for $i = 1$ to $maxGeneration$ do
6:     for all $c$ Cell do
7:         $(temp_H, temp_V)$ =crossover($c.h, c.v$);
8:         $temp_H$ =mutation($temp_H$);
9:         $temp_V$ =mutation($temp_V$);
10:        $c_1$ =calculateNextHorizontalCell($c$);
11:        $c_2$ =calculateNextVerticalCell($c$);
12:        $temp_H$ =elitism($c.h, temp_H$);
13:        $temp_V$ =elitism($c.v, temp_V$);
14:        moveSolutionToCell($temp_H, c_1.h$);
15:        moveSolutionToCell($temp_V, c_2.v$);
16:    end for
17: end for
```

2.1.2 Crossover operator

As the crossover operator used is the two-point crossover, two different crossover point values (preprogrammed at fixed positions of the tentative solutions) have to be calculated for each cell. In each row, to sample different sections of the individuals, the second crossover point is calculated increasing the distance to the first crossover point with the column, and two different values for the first crossover point are used. Figure 2 shows the general idea followed to distribute the crossover points over the entire grid, using different crossover points in each cell to exchange different sections of the solutions through the grid.

For the first crossover point, two different values are used in each row, one for the first $\frac{l}{2}$ cells and another one for the last $\frac{l}{2}$ cells. These two values differ by $\mathrm{div}(l, 2\tau)$, while cells of successive rows in the same column differ by $\mathrm{div}(l, \tau)$. This allows using a large number of different values for the first crossover point following a pattern known a priori. If $x \geq 0$ and $y > 0$, then $\mathrm{div}(x, y) = \lfloor \frac{x}{y} \rfloor$, so we use div in the text but we prefer to use floor notation in the equations for the sake of clarity. Figure 3 illustrates the first crossover point calculation.

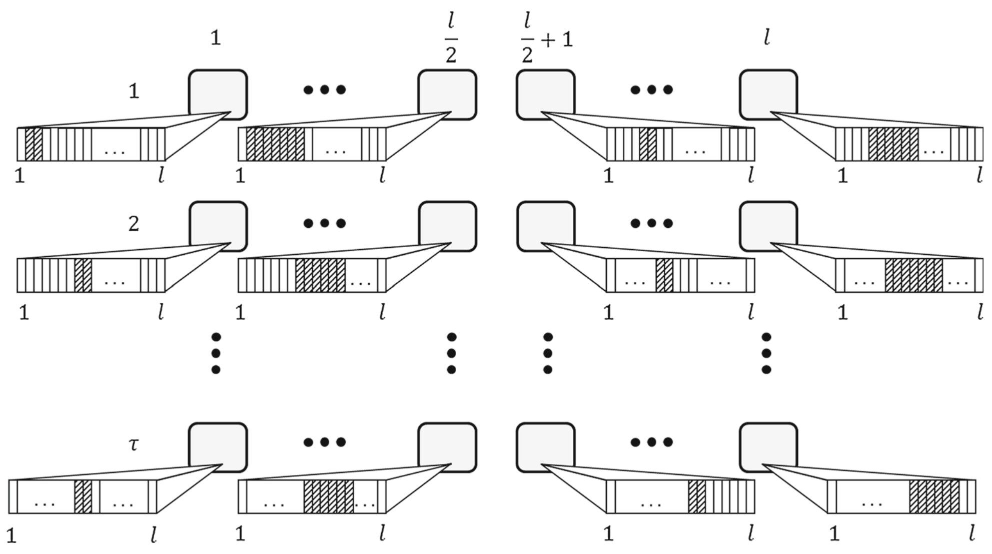


Fig. 2 Distribution of crossover points across the grid

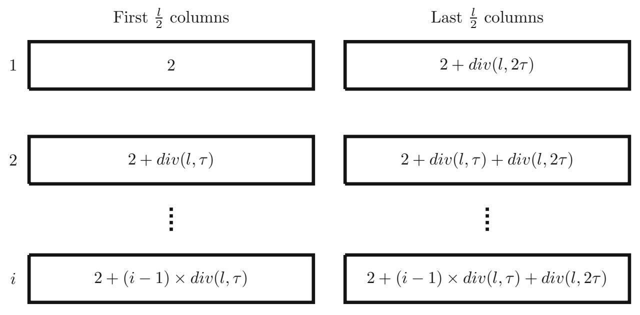

First $\frac{l}{2}$ columns
Last $\frac{l}{2}$ columns
1
2
$2 + div(l, 2\tau)$
2
$2 + div(l, \tau)$
$2 + div(l, \tau) + div(l, 2\tau)$
i
$2 + (i-1) \times div(l, \tau)$
$2 + (i-1) \times div(l, \tau) + div(l, 2\tau)$

Fig. 3 First crossover point calculation

The general expression for calculating the first crossover point at cell (i, j) is:

$$2 + \left\lfloor \frac{l}{\tau} \right\rfloor (i-1) + \left\lfloor \frac{j-1}{\left\lfloor \frac{l}{2} \right\rfloor} \right\rfloor \left\lfloor \frac{l}{2\tau} \right\rfloor \tag{1}$$

For the second crossover point, the distance to the first crossover point increases with the column index, from a minimum distance of two positions to a maximum distance of $\text{div}(l, 2) + 1$ positions. In this way, cells in contiguous columns exchange a larger portion of the solutions. Figure 4 illustrates the second crossover point calculation, being F_1 the first crossover point for the first $\frac{l}{2}$ cells and F_2 the first crossover point for the last $\frac{l}{2}$ cells. If the value of second crossover point is smaller than the first one, the values are swapped.

The general formula for calculating the second crossover point for the cell (i, j) is presented in Eq. 2, where mod is the modulus of the integer division.

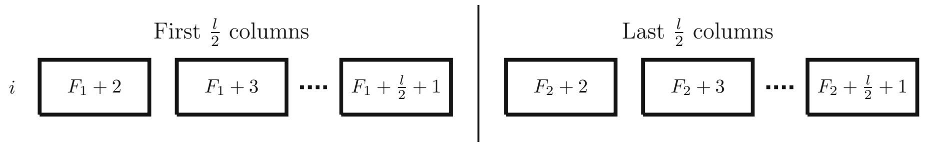
$$1 + \left(3 + \left\lfloor \frac{l}{\tau} \right\rfloor (i-1) + \left\lfloor \frac{j-1}{\left\lfloor \frac{l}{2} \right\rfloor} \right\rfloor \left\lfloor \frac{l}{2\tau} \right\rfloor + \left((j-1) \mod \left\lfloor \frac{l}{2} \right\rfloor\right)\right) \mod l \tag{2}$$

2.1.3 Mutation

The mutation operator flips a single bit in each solution. Figure 5 shows the general idea followed to distribute the points of mutation over the entire grid, using different mutation points in each cell in order to change different bits of the solutions through the grid.

Each cell mutates a different bit of the solutions in the horizontal data stream in order to generate diversity by encouraging the exploration of new solutions. On the other hand, cells in the same vertical data stream should not mutate the same bit in order to avoid deteriorating the search capability of the algorithm. For this reason, the mutation points on each row are shifted $\text{div}(l, \tau)$ places. Figure 6 shows an example of the mutation points for the cells of column j.

Fig. 4 Second crossover point calculation



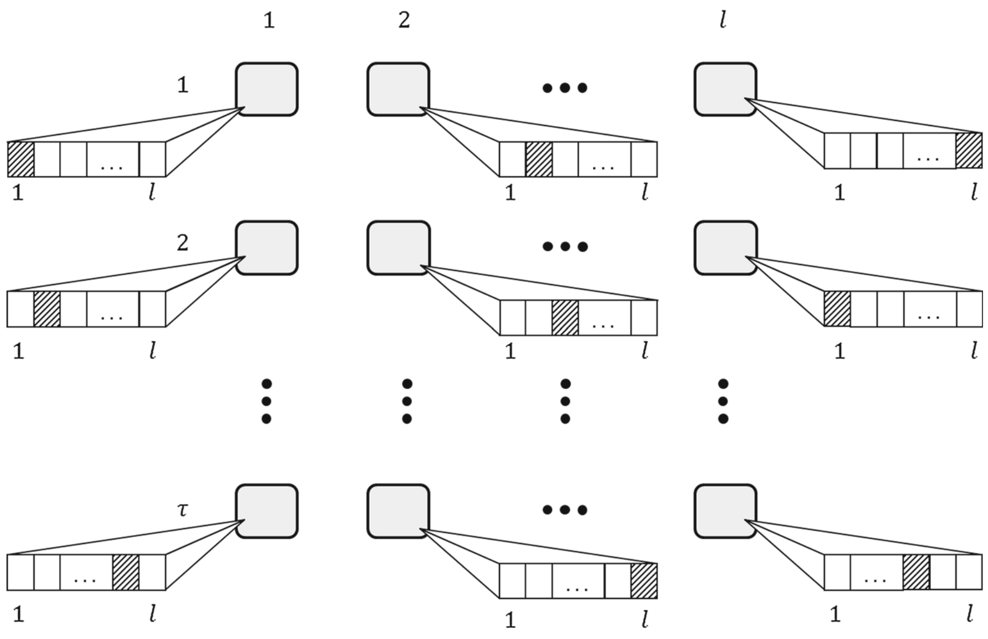


Fig. 5 Distribution of mutation points across the grid

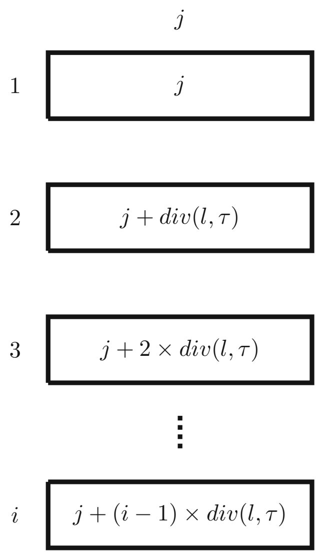


Fig. 6 Mutation points for column j

The general formula for calculating the mutation point of the cell (i, j) is:

$$1 + \left((i-1)\left\lfloor \frac{l}{\tau} \right\rfloor + j - 1\right) \mod l, \tag{3}$$

where mod is the modulus of the integer division.

2.2 SGS flavors

So far, we have described the complete algorithm of SGS, but one important detail is still missing: what happens when a solution reaches the end of the grid either horizontally (horizontal outgoing solution from a cell of the last column) or vertically (vertical outgoing solution from a cell of the last row)? Four different flavors have been devised attending to this design decision.

The first alternative is to use a bidimensional toroidal grid of cells (first subsection below). However, we quickly identify a major issue with this approach as solutions moving vertically lack diversity (remind that the width of the grid is lower than the length, i.e., $l > \tau$) because they are only mutated in τ positions. Three enhanced versions have then been engineered aiming at overcoming this issue. They are presented in the last three subsections.

Toroidal Systolic Genetic Search (SGS$_T$). The solutions flow across a bidimensional toroidal grid (as it is shown in Fig. 7a) either horizontally, moving always in the same row, or vertically, moving always in the same column. The horizontal outgoing solutions from the cells of the last column of the grid are passed on to the cells of the first column of the grid in the same row. In the same way, the vertical outgoing solutions from the cells of the last row of the grid are passed on to the cells of the first row of the grid in the same column.

Toroidal Systolic Genetic Search with Exchange of directions (SGS$_E$). The solutions flow across a bidimensional toroidal grid as it is shown in Fig. 7a). As the length of the grid is larger than the width of the grid, the solutions moving through the columns would be limited to only τ different mutation and crossover points, while those moving horizontally use a wider set of values. To avoid this issue, every τ iterations the two solutions being processed in each cell exchange their directions. That is, the solution received through the horizon-

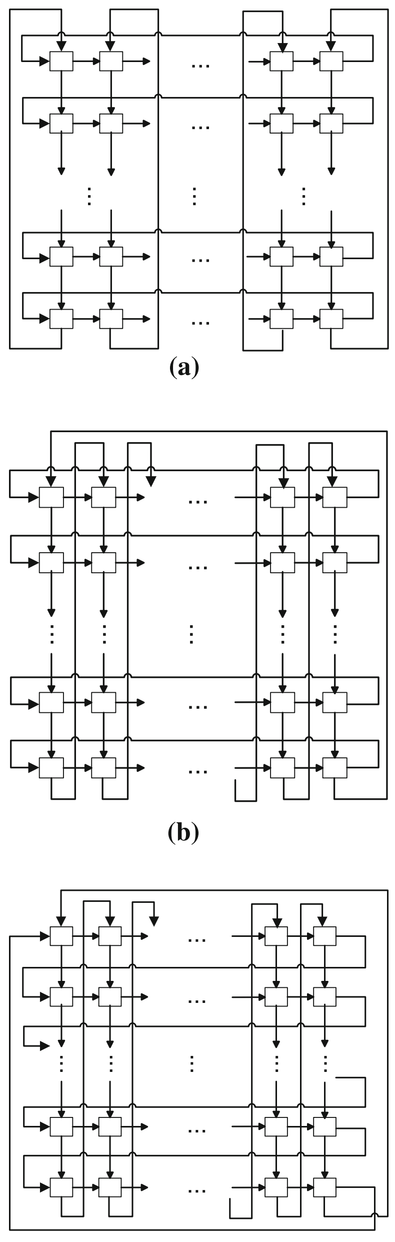


Fig. 7 Interconnection topology for the different flavors. **a** Toroidal grid. **b** Grid with horizontal toroidal flow and vertical flow to the next column. **c** Grid with vertical flow to the next column and horizontal flow to the next row

tal input leaves the cell through the vertical output, while the one moving vertically continues through the horizontal. This flavor has already been used in previous works (Pedemonte et al. 2012, 2013).

Systolic Genetic Search with horizontal toroidal flow and vertical flow of solutions to the next column (SGS_V). In SGS_V, the grid is toroidal regarding the horizontal axis, but to avoid the low diversity in the mutation points of the solutions moving vertically, a vertical outgoing solution from a cell of the last row of the grid is passed on to the cell of the first row of the next column of the grid. The interconnection topology of the cells is shown in Fig. 7b.

Systolic Genetic Search with vertical flow of solutions to the next column and horizontal flow of solutions to the next row (SGS_B).[1] In SGS_B, together with the modification of the vertical flow that happens in SGS_V with respect to SGS_T, a horizontal outgoing solution from a cell of the last column of the grid is passed on to the cell of the first column of the next row of the grid. The interconnection topology of the cells is shown in Fig. 7c.

3 SGS implementation on GPU

This section is devoted to presenting how SGS has been deployed on a GPU. First, we provide a general snapshot of GPU devices and highlight some relevant features of the card used in this work (Nvidia's GeForce GTX 480). Then, all the implementation details are thoroughly explained.

3.1 CUDA graphics processing units

In recent years, GPUs have significantly diversified their field of application because they are no longer just specialized fixed-function graphics platforms. At present, GPUs have become general computing devices composed by highly parallel programmable cores. The architecture of GPUs is designed by following the idea of devoting more transistors to computation than traditional CPUs (Kirk and Hwu 2012). As a consequence, current GPUs have a large number of small cores and are usually considered as *many-core* processors.

CUDA is the general framework that enables to work with Nvidia's GPUs. The CUDA architecture abstracts GPUs as a set of shared memory multiprocessors (MPs) that are able to run a large number of threads in parallel. Each MP follows the SIMT (Single Instruction Multiple Threads) parallel programming paradigm. SIMT is similar to SIMD (Single Instruction Multiple Data) but in addition to data-level parallelism (when threads are coherent) it allows thread-level parallelism (when threads are divergent, see Kirk and Hwu (2012), and Nvidia Corporation (2012c)). The number of threads that modern GPUs can execute in parallel is in the order of thousands and is expected to continue growing rapidly; what makes these devices a powerful and low cost platform for implementing parallel algorithms.

When a *kernel* is called in CUDA, a large number of threads are generated on the GPU. The group of all the threads generated by a kernel invocation is called a *grid*, which is partitioned into many *blocks*. Each block groups threads that are executed concurrently on a single MP. There is no fixed order of execution between blocks. If there are

[1] The B stands for Both flows.

enough multiprocessors available on the card, they are executed in parallel. Otherwise, a time-sharing strategy is used. The blocks are divided for their execution into *warps* that are the basic scheduling units in CUDA and consist of 32 consecutive threads.

Threads can access data on multiple memory spaces during their life time. CUDA architecture has six different memory spaces: registers, shared memory, local memory, global memory, constant memory and texture memory (Kirk and Hwu 2012).

Registers are the fastest memory on the card and are only accessible by each thread. Shared memory is almost as fast as registers and can be accessed by any thread of a block; its lifetime is equal to the lifetime of the block. Each thread has its own local memory but is one of the slowest memories on the card, because it is located in the device memory. Local memory and registers are entirely managed by the compiler. The compiler places variables in local memory when register spilling occurs, i.e., the kernel needs more registers than available. All the threads executing on the GPU have access to the same global memory on the card that is one of the slowest memory on the GPU. Constant memory is a read-only space with only 64 kB accessible by all threads that is located in the device memory. Each multiprocessor has a constant cache of 8 kB that makes access to constant memory space faster. Finally, the texture memory has the same features that of constant memory, but it is optimized for certain access patterns (Nvidia Corporation 2012c).

In this work, we use a GeForce GTX 480 (Compute Capability 2.0 Nvidia Corporation 2012c), which has a Fermi architecture (CUDA's third-generation architecture, Nvidia Corporation 2009). Each multiprocessor on the Fermi architecture consists of 32 CUDA cores that are organized into two blocks with 16 CUDA cores each. Moreover, each MP has two warp schedulers that could handle two warps at once, one for each block of CUDA cores. Figure 8 shows the architecture of the GeForce GTX 480 card, as well as the maximum bandwidth of the access to global GPU memory, CPU memory and transfers between CPU and GPU of the infrastructure used in this work. It should be noted that access to global GPU memory is more than sixteen times faster than access to CPU

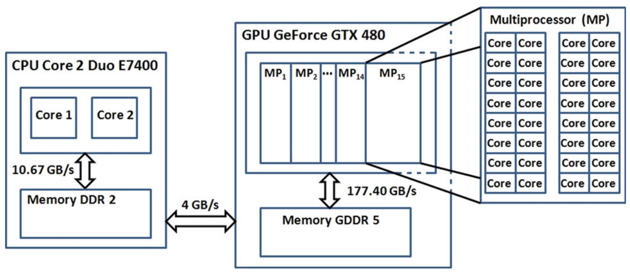


Fig. 8 CPU–GPU system used in this work

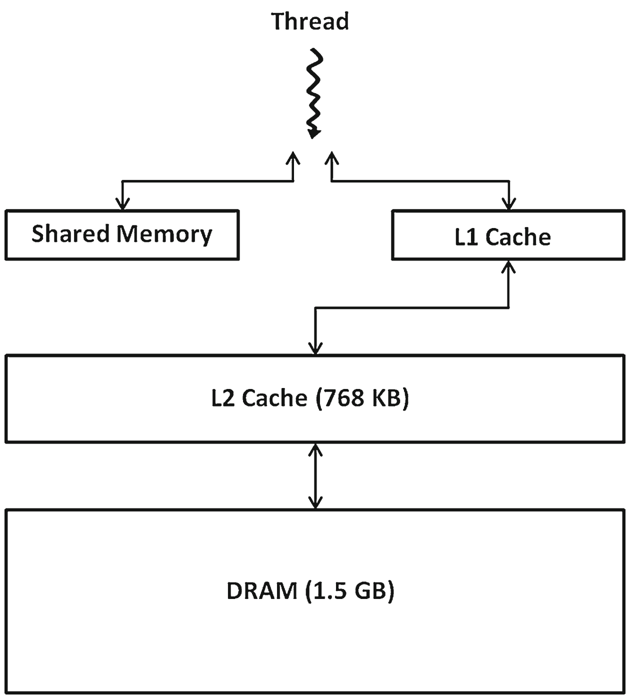


Fig. 9 GeForce GTX 480 (Fermi architecture) memory hierarchy

memory and more than forty times faster than data transfers between CPU and GPU. In fact, the transfers between CPU and GPU are usually one of the most important bottlenecks on CPU–GPU heterogeneous computing.

Each multiprocessor has also an on-chip memory of only 64 kB. A portion of this memory is used as shared memory and the rest is used as a first-level cache for global memory. It can be divided as 16–48 kB or 48–16 kB between cache and shared memory. The Fermi architecture also incorporates a second-level cache with 768 kB shared among all multiprocessors to access the global memory. Figure 9 presents the memory hierarchy of the GeForce GTX 480.

3.2 Implementation details

The approach followed for the GPU implementation of SGS in previous works (Pedemonte et al. 2012, 2013) was targeted to validating the algorithmic proposal, but without neglecting performance. However, little attention was paid in the development of a highly optimized code. For this reason, several design decisions have been reconsidered for this work. One of the most important improvements lies in the kernel design. In the first SGS implementations, each step of the search loop was computed using three different kernels (namely, `crossoverAndMutation`, `evaluate` and `elitism` kernels), while in the present implementation the code of the kernels has been merged into a single kernel to increase the performance. Another important difference is that the pseudorandom number generation has been moved from the

CPU[2] to the GPU. The source code of SGS is publicly available in http://www.fing.edu.uy/~mpedemon/SGS.html. The GPU implementation details are commented next.

Algorithm 2 presents the pseudocode of the SGS algorithm for the host side (CPU). Initially, the seed for the random number generation is transferred from the CPU to the global memory of the GPU and the constant data associated with the problem required for computing the fitness values are transferred from the CPU to texture memory of the GPU. Then, the population is initialized on the GPU (`initPop` kernel) and the fitness of the initial population is computed afterwards (`fitness` kernel). At each iteration, the crossover and mutation operators, the fitness function evaluation, and the elitist replacement are executed on the GPU in a single kernel (`systolicStep` kernel). Additionally, in the SGS_E flavor the exchange of directions operator (`exchange` kernel) is applied on the GPU in given iterations (when div(generation, τ) $==$ 0). Finally, when the algorithm reaches the stop condition, the results are transferred from the GPU to the CPU.

Algorithm 2 SGS Host Side Pseudocode

```
1: transfer seed for random number generation to GPU
2: transfer constant data to GPU's texture memory
3: invoke initPop kernel to initialize population
4: invoke fitness kernel to calculate fitness of the population
5: for $i = 1$ to $maxGeneration$ do
6:     invoke systolicStep kernel to compute systolic step
7:     if $div(generation, \tau) == 0$ then                % only in $SGS_E$
8:         invoke exchange kernel to exchange directions
9:     end if
10: end for
11: transfer results from GPU to CPU
```

3.2.1 Data organization

Two independent memory spaces of the GPU global memory are used to allow concurrent access of data. While the memory space that contains the population in generation t is read, the new solutions from generation $t + 1$ can be written in the other memory space without requiring any type of concurrency control (disjoint storage). Each memory space stores a struct, containing an array with the solutions moving horizontally, an array with the solutions moving vertically, an array with the fitness values corresponding to the solutions moving horizontally, and an array with the fitness values corresponding to the solutions moving vertically.

[2] The random number generation on the CPU guarantees that, using the same seed, the results obtained by a stochastic algorithm in a CPU and in a GPU are the same.

3.2.2 Kernel operation

The `initPop` kernel initializes the population in the GPU using the CUDA CURAND Library (Nvidia Corporation 2012b) to generate random numbers. The kernel is launched with a configuration that depends on the total number of bits that have to be initialized, following the guidelines recommended in Nvidia Corporation (2012a).

The `fitness`, `systolicStep` and `exchange` kernels are implemented following the idea used in Pedemonte et al. (2011), in which operations are assigned to a whole block and all the threads of the block cooperate to perform a given operation. If the solution length is larger than the number of threads in the block, each thread processes more than one element of the solution but the elements used by a single thread are not contiguous. Thus, each operation is applied to a solution in chunks of the size of the thread block (T in the following figure), as it is shown in Fig. 10.

The `systolicStep` kernel is launched with $l \times \tau$ blocks, i.e., each block processes one cell of the grid. Initially, the global memory location of the two solutions that have to be processed by the cell,[3] the global memory location where the resulting solutions should be stored,[4] the crossover points and the mutation point are calculated from the block identifiers by thread zero of the block. These values are stored in shared memory to make them available for the rest of the threads of the block.[5] This kernel uses shared memory to temporarily store the two solutions being constructed and partial fitness values computed by each thread. The amount of shared memory used by each kernel ($8 \times threadsPerBlock + 2 \times l$) ensures that at least four blocks can work concurrently in a multiprocessor with solutions of up to 3,800 bit length. The use of shared memory has the advantage that reduces the accesses to global memory, which is a costly operation, even though it restricts the size of the instances that could be resolved.

Initially, `systolicStep` kernel applies the crossover operator, processing the solution components in chunks of size of the thread block (as it was explained above), taking the two solutions from the first memory space of the GPU global memory and storing the intermediate solutions in the shared memory. The thread zero of the block mutates the two inter-

[3] The two solutions are read from the first memory space of the GPU global memory, one from the array that stores the solutions moving horizontally and the other from the array that stores the solutions moving vertically.

[4] It should be noted that the two solutions are written in the second memory space of the GPU global memory, one in the array that stores the solutions moving horizontally and the other in the array that stores the solutions moving vertically.

[5] We made this decision, rather than making each thread calculate these values redundantly, in order to reduce the number of registers used by the block.

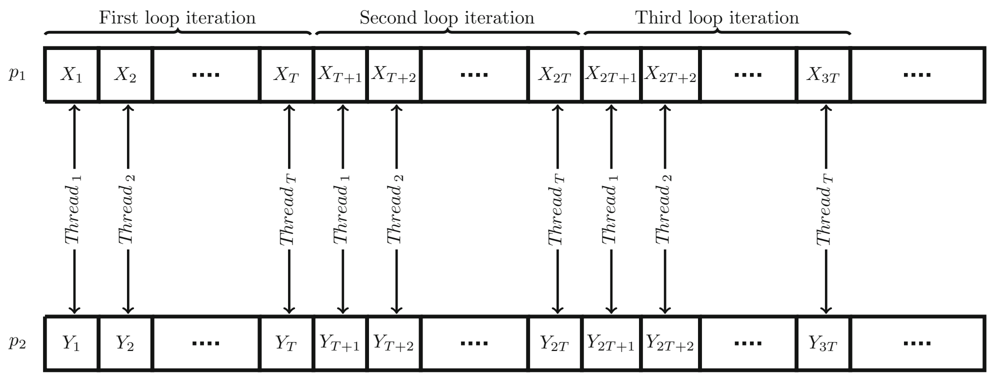

First loop iteration
Second loop iteration
Third loop iteration
p_1
X_1 X_2 .... X_T X_{T+1} X_{T+2} .... X_{2T} X_{2T+1} X_{2T+2} .... X_{3T} ....
$Thread_1$ $Thread_2$ $Thread_T$
p_2
Y_1 Y_2 .... Y_T Y_{T+1} Y_{T+2} .... Y_{2T} Y_{2T+1} Y_{2T+2} .... Y_{3T} ....

Fig. 10 Threads organization

mediate solutions. Then, partial fitness values are computed by each thread using the data from the texture memory of the GPU and those values are stored in shared memory. Then, the kernel applies the well-known reduction pattern (McCool et al. 2012) to these values to calculate the full fitness value of each intermediate solution. Finally, the best solutions for each flow are copied to the second memory space of the GPU global memory, considering the fitness values calculated for the intermediate solutions and the fitness values from the original solutions. If an intermediate solution is better than the original solution in one cell, the intermediate solution is directly copied from the shared memory to the global memory. Otherwise, the original solution is copied from the first memory space of the global memory to the second one.

The `fitness` and `exchange` kernels follow the same idea regarding the thread organization and behavior than `systolicStep` kernel, and are also launched for execution organized in $l \times \tau$ blocks.

4 Experimental study

This section describes the problems used for the experimental study, the parameters setting, and the execution platforms. Then, the results obtained are presented and analyzed.

4.1 Test problems

For the experimental evaluation of SGS, we use two classical benchmark problems, Knapsack Problem and Massively Multimodal Deceptive Problem, plus a real-world application, the Next Release Problem. These problems and the test instances used are briefly introduced next.

4.1.1 Knapsack problem

The Knapsack Problem (KP) is a classical combinatorial optimization problem that belongs to the class of $\mathcal{NP}$-hard problems (Pisinger 1999). It is defined as follows. Given a set of n items, each of them having associated an integer value p_i called profit or value and an integer value w_i known as weight, the goal is to find the subset of items that maximizes the total profit keeping the total weight below a fixed maximum capacity (W) of the knapsack or bag. It is assumed that all profits and weights are positive, that all the weights are smaller than W (items heavier than W do not belong to the optimal solution), and that the total weight of all the items exceeds W (otherwise, the optimal solution contains all the items of the set).

The most common formulation of the KP is the integer programming model presented in Eqs. 4a, 4b, and 4c, being x_i the binary decision variables of the problem that indicate whether the item i is included or not in the knapsack.

$$\text{(KP)} \quad \text{maximize} \quad f(\mathbf{x}) = \sum_{i=1}^{n} p_i x_i \tag{4a}$$

$$\text{subject to:} \quad \sum_{i=1}^{n} w_i x_i \leqslant W \tag{4b}$$

$$x_i \in \{0, 1\}, \forall i = 1, \ldots, n \tag{4c}$$

Table 1 presents the instances used in this work. These instances have been generated with no correlation between the weight and the profit of an item (i.e., w_i and p_i are chosen randomly in $[1, R]$) using the generator described in Pisinger (1999). The Minknap algorithm (Pisinger 1997), an exact method based on dynamic programming, was used to find the optimal solution for each of the instances.

All the algorithms studied use a penalty approach to manage infeasibility. In this case, the penalty function subtracts W to the total profit for each unit of the total weight that exceeds the maximum capacity. The formula for calculating the fitness with penalty is:

$$f(\mathbf{x}) = \sum_{i=1}^{n} p_i x_i - \left(\sum_{i=1}^{n} w_i x_i - W \right) \times W. \tag{5}$$

Table 1 Knapsack instances used in the experimental evaluation and their exact optimal solutions

Instance	n	R	W	Profit of Opt. Sol	Weight of Opt. Sol
100–1,000	100	1,000	1,001	5,676	983
100–10,000	100	10,000	10,001	73,988	9,993
200–1,000	200	1,000	1,001	10,867	1,001
200–10,000	200	10,000	10,001	100,952	9,944
500–1,000	500	1,000	1,001	19,152	1,000
500–10,000	500	10,000	10,001	153,726	9,985
1,000–1,000	1,000	1,000	1,001	27,305	1,000
1,000–10,000	1,000	10,000	10,001	231,915	9,996

Table 2 MMDP basic deceptive subfunction

Number of ones (unitation)	Subfunction value
0	1.000000
1	0.000000
2	0.360384
3	0.640576
4	0.360384
5	0.000000
6	1.000000

4.1.2 Massively multimodal deceptive problem

The Massively Multimodal Deceptive Problem (MMDP) is a problem that has been specifically designed to make EAs converge to regions of the search space where the optimal solution cannot be found (Goldberg et al. 1992). MMDP is made up of k deceptive subproblems of 6 bits each one. The function value of each of these subproblems is independent from each other and only depends on the number of ones it has (Unitation), following Table 2. The optimal solution of a MMDP with k subproblems is accomplished if every subproblem has either zero or six ones, and in that case the function value and the fitness value are k. We use for the experimental evaluation instances with strings of 300, 600, 900, 1,200 and 1,500 bits and therefore, the optimal solutions are 50, 100, 150, 200 and 250, respectively.

4.1.3 Next release problem

The Next Release Problem (NRP) is a real-world problem that arises in the software development industry (Bagnall et al. 2001). In NRP, a company involved in the development of a large software system has to determine which requirements should be targeted in the next release of the software. The set of costumers has different requirements that provide some value to the company, while fulfilling each requirement has an associated cost for the company.

NRP can be stated in the following terms (Durillo et al. 2011). There is a set C of m customers and a set R of n requirements. The economical cost of satisfying each requirement is denoted by r_j. Each customer has associated a value c_i that reflects the importance of the customer to the company. There is also a value associated with each costumer and each requirement (v_{ij}) that represents the importance for the customer i of the requirement j.

NRP was originally formulated as a single-objective problem using an integer programming model that is closely related with the knapsack problem (Bagnall et al. 2001). The formulation of the single-objective NRP is presented in Eqs. 6a, 6b, and 6c, being x_j the binary decision variables of the problem that indicate whether the requirement j is satisfied or not and B a given bound for the total cost.

$$\text{(NRP)} \quad \text{maximize} \quad f(\mathbf{x}) = \sum_{i=1}^{m} c_j \sum_{j=1}^{n} x_j v_{ij} \tag{6a}$$

$$\text{subject to:} \quad \sum_{j=1}^{n} x_j r_j \leqslant B \tag{6b}$$

$$x_j \in \{0, 1\}, \forall j = 1, \ldots, n \tag{6c}$$

Later on, NRP was reformulated as a bi-objective problem to avoid imposing the artificial constraint presented in Eq. 6b. The formulation of the bi-objective NRP (Durillo et al. 2011; Zhang et al. 2007) is presented in Eqs. 7a, 7b, and 7c.

$$\text{(NRP) minimize} \quad f_1(\mathbf{x}) = \sum_{j=1}^{n} x_j r_j \tag{7a}$$

$$\text{maximize} \quad f_2(\mathbf{x}) = \sum_{i=1}^{m} c_j \sum_{j=1}^{n} x_j v_{ij} \tag{7b}$$

$$\text{subject to:} \quad x_j \in \{0, 1\}, \forall j = 1, \ldots, n \tag{7c}$$

Since SGS is a single-objective algorithm, we followed a similar approach that Zhang et al. (2007), who also solved the NRP using a single-objective GA.

To that end, the authors transform the first-objective function in a maximization, as shown in Eq. 8, and use the weighted sum method (Deb 2001; Marler and Arora 2004) that combines both objective functions into a single-objective using w as a weighting factor ($0 \leq w \leq 1$), as shown in Eq. 9.

$$\text{maximize} \quad f_1(\mathbf{x}) = -\sum_{j=1}^{n} x_j r_j \tag{8}$$

$$\text{maximize} \quad F(\mathbf{x}) = (1-w)\cdot f_1(\mathbf{x}) + w \cdot f_2(\mathbf{x}) \tag{9}$$

However, there is a great difference between the magnitudes of f_1 and f_2, so we normalized both objective functions (Deb 2001; Marler and Arora 2004) to map them in [0,1]. The formula for normalization in a maximization is:

$$f_i^{\text{Trans}}(\mathbf{x}) = \frac{f_i(\mathbf{x}) - z_i^{\text{Nadir}}}{z_i^{\text{Ideal}} - z_i^{\text{Nadir}}}, \tag{10}$$

being z^{Nadir} the Nadir point, i.e., the point with the worse (minimal) value for each f_i and z^{Ideal} the Ideal or utopian point, i.e., the point with the best (maximal) value for each f_i.

Since $\forall \mathbf{x} f_1(\mathbf{x}) \leq 0$ and $f_2(\mathbf{x}) \geq 0$, then $z_1^{\text{Ideal}} = 0$ and $z_2^{\text{Nadir}} = 0$, thus resulting in the objective functions shown in Eqs. 11a and 11b.

$$f_1^{\text{Trans}}(\mathbf{x}) = \frac{f_1(\mathbf{x}) - z_i^{\text{Nadir}}}{-z_i^{\text{Nadir}}} \tag{11a}$$

$$f_2^{\text{Trans}}(\mathbf{x}) = \frac{f_2(\mathbf{x})}{z_i^{\text{Ideal}}} \tag{11b}$$

To obtain a better distribution on the Pareto Front of the solutions obtained with a single-objective algorithm, we preferred to use the Tchebycheff approach (Marler and Arora 2004; Miettinen 1999; Zhang and Li 2007) rather than using the weighted sum method. This will avoid the usual issue of not being able to solve non-convex problems. Thus, the resulting problem formulation is:

$$\text{minimize } g(\mathbf{x}) = \max(f_1^{\text{Tch}}(\mathbf{x}), f_2^{\text{Tch}}(\mathbf{x})) \tag{12a}$$

$$\text{where: } f_1^{\text{Tch}}(\mathbf{x}) = (1-w)\cdot(1 - f_1^{\text{Trans}}(\mathbf{x})) \tag{12b}$$

$$f_2^{\text{Tch}}(\mathbf{x}) = w \cdot (1 - f_2^{\text{Trans}}(\mathbf{x})) \tag{12c}$$

Since the original problem has been transformed in a minimization and $g(\mathbf{x}) \leq 1$, the fitness function is defined as follows:

$$f(\mathbf{x}) = 1 - g(\mathbf{x}). \tag{13}$$

Additionally, as it is possible that $\mathbf{x}$ dominates $\mathbf{y}$ and $g(\mathbf{x}) = g(\mathbf{y})$ (Zhang and Li 2007), when two different solutions with the same fitness value are compared (e.g., when elitism is applied), it is checked whether a solution dominates the other.

The instances used in this work for the experimental evaluation of the NRP are taken from Durillo et al. (2011), and Zhang et al. (2007). The instance name indicates the number of costumers and requirements ($m-n$ stands for m costumers and n requirements). The instances used are 100–20, 100–25, 35–35 (real-world instance from Durillo et al. 2011), 15–40, 50–80, 100–140 and 2–200. The optimal value for each instance and weighting factor w is unknown since the approach followed in the previous work (Zhang et al. 2007) for solving the NRP using single-objective algorithms is different from the one used in this work.

4.2 Algorithms

In addition to the SGS algorithms proposed in this paper, we have included two algorithms, a Random Search and a simple Genetic Algorithm (GA) with and without elitism, to compare the quality of the solutions obtained. The former is used as a sanity check, just to show that our algorithmic proposals are more intelligent that a pure random sampling. On the other hand, the GAs have been chosen because of their popularity in the literature and also because they share the same basic search operators so we can properly compare the underlying search engine of the techniques. Briefly, the details of these algorithms are:

- Random Search (RS): The RS algorithm processes each bit of the solution vector sequentially. Each bit is set to 1 at random with probability 0.5, except for the KP. In the KP, if including an item in the knapsack exceeds the maximum capacity, it is discarded. Otherwise, the item is included in the knapsack at random with probability 0.5.
- Simple Genetic Algorithm (SGA): It is a generational GA with binary tournament, two-point crossover, and bit-flip mutation.
- Elitist Genetic Algorithm (EGA): It is similar to SGA but with elitist replacement, i.e., each child solution replaces its parent solution only if it has a better (higher) fitness value.

Each of the algorithms studied has been implemented both on CPU and GPU, except RS and SGS$_\text{T}$ that have only been implemented on CPU since they use a rather simple search engine with low numerical efficiency. The CPU implementation is straightforward, so no further details are provided. The SGA and EGA implementation on GPU follows the same guidelines that the implementations of the SGS algorithms.

4.3 Parameters setting and test environment

The SGA and EGA parameter values used are 0.9 for the crossover probability and $1/l$ for the mutation probability, where l is the length of the tentative solutions. The population size and the number of iterations are defined by considering the features of SGS, using exactly the same values for the two GA versions. In this study, the population size is $2 \times l \times \tau$ and the number of iterations is $l \times \tau$ (recall that $\tau = \lceil \lg l \rceil$). This number was chosen so that each solution returns to its original cell in SGS_B after that number of iterations. Finally, $2 \times l^2 \times \tau^2$ solutions are generated by RS to perform a fair comparison.

In the NRP, we use eleven different weight coefficients w ranging from 0 to 1 with a step size of 0.1 to analyze the importance of the two internal goal functions. Each execution reported in the article consists of eleven consecutive and independent runs with the different possible values of w to obtain different solutions within a single experiment. A similar approach was previously used in Zhang et al. (2007), but using only nine different values (ranging from 0.1 to 0.9 with a step size of 0.1).

It is still a controversial issue how to make a fair comparison between traditional CPUs and modern GPUs. The selection of the execution platforms tries to follow the guidelines suggested in Hennessy and Patterson (2011). The execution platform for the CPU versions is a PC with a Quad Core Xeon E5530 processor at 2.40 GHz with 48 GB RAM using Linux operating system. The CPU versions have been compiled using the `-O3` flag and are run as single thread applications. The execution platform for the GPU versions is a Nvidia's GeForce GTX 480 (480 CUDA Cores) connected to a PC with a Core 2 Duo E7400 at 2.80 GHz with 2 GB RAM using Linux operating system. The GPU versions were also compiled using the `-O3` flag.

All the results reported are mean values rounded to two figures over 50 independent runs. The transference times of data between CPU and GPU are included in the reported total runtime of the GPU version.

4.4 Experimental analysis

This section describes the experimental analysis conducted to validate SGS. The experiments include a study of the numerical efficiency of the algorithm proposed and a study of the performance of the parallel GPU implementation of SGS.

All the algorithms in this work are stochastic algorithms, therefore, the results have to be provided with statistical significance. The following statistical procedure has been used. First, fifty independent runs for each algorithm and each problem instance have been performed. The following statistical analysis has been carried out (Sheskin 2011). First, a Kolmogorov–Smirnov test and a Levene test are performed to check, respectively, whether the samples are distributed according to a normal distribution and whether the variances are homogeneous (homocedasticity). If the two conditions hold, an ANOVA I test is performed; otherwise we perform a Kruskal–Wallis test. All the statistical tests are performed with a confidence level of 95 %. Since more than two algorithms are involved in the study, a post hoc testing phase which allows for a multiple comparison of samples has been performed. The result is a pairwise comparison of all the cases compared using the Bonferroni–Dunn method on either the Student's t test (if the samples follow a normal distribution and the variances are homogeneous) or the Wilcoxon–Mann–Whitney test (otherwise). The results are displayed in tabular form (see below), where '▲' states that the configuration of the row has statistically lower values (i.e., it is better) than the column and '▽' states that the opposite is true. When no statistically significant differences are found, the '—' symbol is used.

4.4.1 Numerical efficiency

Let us first analyze the numerical efficiency for KP. Table 3 presents the experimental results regarding the quality of the solutions (measured in terms of distance to the optimal solution) obtained for the KP, while Table 4 presents in which instances the statistical confidence has been achieved.

The results obtained show that SGS_E, SGS_V and SGS_B are the best performing algorithms for the KP, as they are far superior than RS, SGA and SGS_T in all the instances considered in this study. They are also superior than EGA in five out of eight instances (the instances with more items). It should also be noted that both SGS_V and SGS_B find the optimal solution on every run for all the instances, while SGS_E reaches the optimal solution on every run for six out of eight instances. EGA also performs well, having a small mean error and being superior to RS, SGA and SGS_T in all the instances studied. Although the results obtained by SGS_T are not satisfactory due to the rather high mean error, SGS_T performs better than RS and SGA. SGA presents non-competitive results and it is only better than the (non-intelligent) random search. It is also remarkable the ability of SGS_V and SGS_B to scale properly with the size of the KP instances: they have consistently reached the optimal solutions regardless of the number of items (which ranges from 100 to 1,000).

Now, we analyze the numerical efficiency for the MMDP. Table 5 presents the experimental results regarding the quality of the solutions obtained for MMDP, while Table 6 presents in which instances the statistical confidence has been achieved.

The results obtained show that SGA, SGS_E, SGS_V and SGS_B are the best performing algorithms for MMDP. They all reach the optimal solution in every independent run for all the considered instances. EGA and SGS_T have a similar performance, having a small mean error and only being superior to RS. It is interesting that EGA performs worse than SGS_E, SGS_V and SGS_B. Since MMDP is a deceptive problem, it is reasonable that an algorithm with elitism is especially attracted to local optima. However, the systolic

Table 3 Numerical efficiency of CPU versions for KP (mean error ± std. dev.)

Instance	RS	SGA	EGA	SGS_T	SGS_E	SGS_V	SGS_B
100–1,000	1.59e3 ± 8.56e1	4.79e2 ± 1.65e2	5.14e0 ± 2.60e1	2.56e2 ± 1.15e2	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**
100–10,000	1.96e4 ± 1.78e3	7.15e3 ± 2.24e3	**0.00e0 ± 0.00e0**	3.66e3 ± 2.03e3	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**
200–1,000	5.27e3 ± 1.65e2	1.77e3 ± 3.07e2	3.64e0 ± 1.46e1	9.23e2 ± 2.71e2	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**
200–10,000	2.96e4 ± 1.77e3	1.12e4 ± 2.05e3	1.36e1 ± 3.86e1	8.15e3 ± 2.42e3	0.20e0 ± 1.41e0	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**
500–1,000	1.19e4 ± 1.74e2	4.32e3 ± 4.12e2	2.21e1 ± 3.74e1	2.04e3 ± 3.73e2	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**
500–10,000	7.06e4 ± 2.20e3	3.44e4 ± 2.56e3	1.55e2 ± 1.65e2	1.55e4 ± 3.39e4	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**
1,000–1,000	1.93e4 ± 2.89e2	7.93e3 ± 4.65e2	5.27e1 ± 4.19e1	6.59e3 ± 6.87e2	5.52e0 ± 1.71e1	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**
1,000–10,000	1.34e5 ± 2.63e3	6.55e4 ± 4.02e3	4.00e2 ± 6.36e2	5.73e4 ± 4.23e3	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**

The best results are in bold

Table 4 Statistical significance for instances 100–1,000, 100–10,000, 200–1,000, 200–10,000, 500–1,000, 500–10,000, 1,000-1,000, 1,000–10,000

	SGA	EGA	SGS_T	SGS_E	SGS_V	SGS_B
RS	▽ ▽ ▽ ▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽ ▽ ▽ ▽
SGA		▽ ▽ ▽ ▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽ ▽ ▽ ▽
EGA			▲ ▲ ▲ ▲ ▲ ▲ ▲ ▲	− − − ▽ ▽ ▽ ▽ ▽	− − − ▽ ▽ ▽ ▽ ▽	− − − ▽ ▽ ▽ ▽ ▽
SGS_T				▽ ▽ ▽ ▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽ ▽ ▽ ▽
SGS_E					− − − − − − − −	− − − − − − − −
SGS_V						− − − − − − − −

Table 5 Numerical efficiency of CPU versions for MMDP (mean error ± std. dev.)

Instance	RS	SGA	EGA	SGS_T	SGS_E	SGS_V	SGS_B
300	2.07e1 ± 3.88e−1	**0.00e0 ± 0.00e0**	1.44e−2 ± 7.11e−2	1.44e−2 ± 7.11e−2	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**
600	4.63e1 ± 5.20e−1	**0.00e0 ± 0.00e0**	1.29e−1 ± 2.02e−1	3.95e−1 ± 2.32e−1	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**
900	7.26e1 ± 5.37e−1	**0.00e0 ± 0.00e0**	9.92e−1 ± 6.01e−1	7.69e−1 ± 2.81e−1	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**
1,200	9.95e1 ± 6.82e−1	**0.00e0 ± 0.00e0**	1.89e0 ± 9.49e−1	1.20e0 ± 4.63e−1	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**
1,500	1.27e2 ± 8.42e−1	**0.00e0 ± 0.00e0**	4.08e0 ± 1.11e0	3.52e0 ± 1.37e0	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**

The best results are in bold

Table 6 Statistical significance for instances 300, 600, 900, 1,200, 1,500

	SGA	EGA	SGS_T	SGS_E	SGS_V	SGS_B
RS	▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽
SGA		− ▲ ▲ ▲ ▲	− ▲ ▲ ▲ ▲	− − − − −	− − − − −	− − − − −
EGA			− ▲ − ▽ −	− ▽ ▽ ▽ ▽	− ▽ ▽ ▽ ▽	− ▽ ▽ ▽ ▽
SGS_T				− ▽ ▽ ▽ ▽	− ▽ ▽ ▽ ▽	− ▽ ▽ ▽ ▽
SGS_E					− − − − −	− − − − −
SGS_V						− − − − −

variants, which also use elitism, have managed to avoid getting stuck in unpromising regions of the search space. SGS_E, SGS_V and SGS_B have been able to scale with the size of the instances, showing the promising search engine devised.

Finally, we analyze the numerical efficiency for the NRP. In a previous work, Zhang et al. (2007) used two single-objective algorithms for solving the NRP, but they used a different approach than the one used in this work. For this reason, the optimal value for each instance and weighting factor w is unknown. Table 7 presents the best solution found on all the executions for each pair instance weighting factor considered. These solutions will be considered the best known solutions of the NRP for the experimental analysis.

Table 8 presents the experimental results regarding the quality of the solutions obtained for the NRP. For each instance, we measure the Euclidean distance between the

Table 7 Best solution found for NRP

Instance	Weight										
	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
100–20											
Cost	0	9	18	25	33	44	54	65	79	90	107
Value	0	495.55	762.19	1,004.11	1,238.43	1,472.86	1,599.37	1,816.68	1,964.12	2,181.44	2,408.56
100–25											
Cost	0	4	8	12	18	22	27	33	40	48	58
Value	2,433	4,791	6,016	7,157	8,458	9,124	10,301	11,451	12,620	13,741	14,998
35–35											
Cost	0	440	770	1,080	1,430	1,830	2,240	2,840	3,440	4,640	6,740
Value	0	33	42	47	52	56	60	64	68	72	78
15–40											
Cost	0	15	29	42	55	69	84	101	124	147	185
Value	0	167.19	254.35	322.51	378.34	429.51	478.18	525.47	572.54	626.96	688.36
50–80											
Cost	0	34	66	98	131	165	201	240	284	337	404
Value	0	1,061.73	1,614.07	2,049.91	2,446.10	2,821.81	3,184.76	3,552.58	3,925.87	4,323.95	4,761.10
100–140											
Cost	0	58	110	162	216	271	332	397	471	562	572
Value	0	3,936.08	5,923.49	7,541.37	8,962.41	10,299.10	11,595.75	12,920.88	14,264.14	15,734.58	17,289.44
2–200											
Cost	0	80	149	216	283	353	427	509	600	716	987
Value	0	128.29	185.24	228.57	265.37	298.49	329.66	360.84	392.05	424.53	461.04

Table 8 Numerical efficiency of CPU versions for NRP (mean error ± std. dev.)

Instance	RS	SGA	EGA	SGS_T	SGS_E	SGS_V	SGS_B
100–20	6.15e2 ± 4.31e2	0.02e0 ± 0.25e0	0.09e0 ± 0.60e0	0.11e0 ± 0.52e0	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**
100–25	3.71e3 ± 2.49e3	0.11e0 ± 1.72e0	0.87e0 ± 2.03e1	1.15e0 ± 7.28e0	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**
35–35	2.03e3 ± 1.24e3	0.01e0 ± 0.06e0	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**
15–40	1.90e2 ± 1.14e2	0.28e0 ± 1.07e0	0.39e0 ± 1.38e0	0.24e0 ± 1.12e0	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**
50–80	1.22e3 ± 7.99e2	5.43e0 ± 4.44e0	0.65e0 ± 2.15e0	0.98e0 ± 1.94e0	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**
100–140	4.41e3 ± 2.84e3	2.44e1 ± 2.81e1	3.84e0 ± 1.08e1	5.07e0 ± 1.00e1	3.39e0 ± 1.01e1	2.23e0 ± 6.53e0	**0.09e0 ± 0.28e0**
2–200	3.01e2 ± 1.53e2	0.94e0 ± 0.91e0	0.05e0 ± 0.18e0	1.77e0 ± 1.94e0	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**	**0.00e0 ± 0.00e0**

The best results are in bold

solution obtained for each run and the best solution found, using the same weighting factor, calculating the mean error as the average of these distances.

Table 9 presents in which instances the statistical confidence has been achieved.

The results obtained show that SGS_E, SGS_V and SGS_B are among the best performing algorithms for the NRP, as their solutions are much closer to the best known than RS, SGA, EGA and SGS_T in most of the instances considered in this study; SGS_B even outperforms SGS_E and SGS_V in one instance. It should also be noted that those three algorithms are able to find the best known solution in every run in six out of seven instances, while EGA and SGS_T find the best known solution in every run in one of the instances. EGA also performs well, having a small mean error and being superior to SGA and SGS_T in most instances and to RS in all the instances studied. SGA and SGS_T have a similar performance, having an acceptable mean error in most instances and only being superior to RS.

From these results, it is clear that the structured search of SGS performs an intelligent exploration of the search space, allowing three out of four flavors of SGS to identify the region where the optimal solution is located for the considered problem and instances. The key aspect that explains why SGS_T

Table 9 Statistical significance for instances 100–20, 100–25, 35–35, 15–40, 50–80, 100–140, 2–200

	SGA	EGA	SGS_T	SGS_E	SGS_V	SGS_B
RS	▽ ▽ ▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽ ▽ ▽	▽ ▽ ▽ ▽ ▽ ▽ ▽
SGA		– – – ▲ ▽ ▽ ▽	▲ ▲ – ▽ ▽ ▽ ▲	– – – ▽ ▽ ▽ ▽	– – – ▽ ▽ ▽ ▽	– – – ▽ ▽ ▽ ▽
EGA			– ▲ – – ▲ ▲ ▲	▽ – – ▽ ▽ – ▽	▽ – – ▽ ▽ – ▽	▽ – – ▽ ▽ ▽ ▽
SGS_T				▽ ▽ – ▽ ▽ ▽ ▽	▽ ▽ – ▽ ▽ ▽ ▽	▽ ▽ – ▽ ▽ ▽ ▽
SGS_E					– – – – – – –	– – – – – ▽ –
SGS_V						– – – – – ▽ –

Table 10 Runtime in seconds of the CPU versions for KP (mean ± std. dev.)

Instance	SGA	EGA	SGS_T	SGS_E	SGS_V	SGS_B
100–1,000	1.27e0 ± 0.01e0	1.36e0 ± 0.01e0	0.69e0 ± 0.04e0	0.71e0 ± 0.04e0	0.66e0 ± 0.04e0	**0.65e0 ± 0.03e0**
100–10,000	1.35e0 ± 0.04e0	1.42e0 ± 0.04e0	0.76e0 ± 0.05e0	0.73e0 ± 0.02e0	**0.66e0 ± 0.03e0**	0.70e0 ± 0.04e0
200–1,000	1.30e1 ± 0.41e0	1.36e1 ± 0.09e0	6.07e0 ± 0.36e0	6.12e0 ± 0.33e0	**5.54e0 ± 0.21e0**	5.70e0 ± 0.13e0
200–10,000	1.29e1 ± 0.15e0	1.38e1 ± 0.10e0	6.25e0 ± 0.37e0	6.08e0 ± 0.20e0	5.85e0 ± 0.36e0	**5.75e0 ± 0.04e0**
500–1,000	2.52e2 ± 2.54e0	2.63e2 ± 4.57e0	1.09e2 ± 3.21e0	1.08e2 ± 2.97e0	**1.03e2 ± 3.44e0**	1.06e2 ± 3.69e0
500–10,000	2.48e2 ± 4.81e0	2.68e2 ± 8.05e0	1.08e2 ± 0.75e0	1.03e2 ± 0.20e0	**9.82e1 ± 1.09e0**	1.03e2 ± 1.38e0
1,000–1,000	2.42e3 ± 6.75e1	2.56e3 ± 4.84e1	9.10e2 ± 5.23e0	9.30e2 ± 2.54e1	**8.82e2 ± 5.35e0**	9.08e2 ± 1.04e1
1,000–10,000	2.39e3 ± 6.80e1	2.52e3 ± 4.54e1	8.98e2 ± 3.90e0	9.12e2 ± 3.27e1	**8.66e2 ± 5.60e0**	8.92e2 ± 1.72e1

The shortest runtimes are in bold

Table 11 Runtime in seconds of CPU versions for MMDP (mean ± std. dev.)

Instance	SGA	EGA	SGS_T	SGS_E	SGS_V	SGS_B
300	5.62e1 ± 1.67e0	5.91e1 ± 2.08e0	2.64e1 ± 0.03e0	2.74e1 ± 1.29e0	**2.50e1 ± 1.34e0**	2.71e1 ± 0.66e0
600	5.37e2 ± 2.39e0	5.73e2 ± 7.70e0	2.44e2 ± 6.30e0	2.46e2 ± 2.45e0	2.40e2 ± 1.36e1	**2.36e2 ± 1.20e1**
900	1.85e3 ± 4.9e1	1.91e3 ± 2.78e1	7.63e2 ± 1.24e1	7.85e2 ± 7.83e0	7.72e2 ± 4.21e1	**7.43e2 ± 1.95e1**
1,200	5.25e3 ± 1.64e2	5.47e3 ± 1.74e2	2.13e3 ± 5.43e1	2.17e3 ± 5.78e1	2.11e3 ± 9.36e1	**2.10e3 ± 9.38e1**
1,500	1.02e4 ± 1.73e2	1.08e4 ± 3.39e2	4.10e3 ± 1.05e2	4.23e3 ± 1.36e2	4.04e3 ± 1.14e2	**4.00e3 ± 4.64e1**

The shortest runtimes are in bold

is not competitive with the other flavors of SGS is that SGS_T has low diversity in the mutation and crossover points of the solutions moving through the vertical data stream. Within the context of the experimental evaluation of the KP, a deceptive problem, like the MMDP, and a real-world problem like the NRP, it has been shown the potential of SGS_E, SGS_V and SGS_B regarding the quality of the obtained solutions.

4.4.2 Parallel performance

In this section, we begin our study with the performance analysis of the CPU versions of the algorithms studied. Tables 10, 11 and 12 show the mean runtime in seconds and the standard deviation of the algorithms executed on CPU for the KP, MMDP and NRP, respectively. The runtime of RS is not included due to its poor numerical results.

The results show that SGS algorithms are the best performing algorithms. In particular, SGS_V is the algorithm with the shortest runtime in most instances of the KP, while SGS_B is the best performing algorithm in most instances of the MMDP. This is mainly caused because the crossover and mutation points of each cell are calculated according to the coordinates of the cell on the grid, thus avoiding the generation of random numbers during the execution of the algorithm. In NRP, the behavior is somewhat different to the behavior observed in the other two problems as SGS_B and SGS_V are among the algorithms that take the longest runtime to finish in several instances. This fact may be mainly provoked by two reasons. On one hand, the fitness function is the most computationally costly of the whole experimental evaluation. On the other hand, in NRP each execution consists of eleven consecutive runs with the different values of the weighting factor. These two facts, as well as the reduced number of requirements of the instances used, i.e., the number of decision variables, seem to compensate the possible gain in performance that could be achieved by avoiding the random number generation during its execution.

Table 12 Runtime in seconds of CPU versions for NRP (mean ± std. dev.)

Instance	SGA	EGA	SGS_T	SGS_E	SGS_V	SGS_B
100–20	4.54e−1 ± 2.16e−2	4.71e−1 ± 8.38e−3	4.19e−1 ± 2.54e−2	**4.02e−1 ± 5.55e−4**	4.81e−1 ± 4.90e−4	5.52e−1 ± 7.52e−3
100–25	9.12e−1 ± 4.80e−2	9.65e−1 ± 2.44e−2	**8.43e−1 ± 1.86e−2**	8.44e−1 ± 2.50e−2	9.80e−1 ± 1.16e−3	1.12e0 ± 3.38e−2
35–35	1.77e0 ± 8.03e−2	1.88e0 ± 3.62e−2	**1.55e0 ± 4.43e−2**	**1.55e0 ± 1.83e−2**	1.71e0 ± 8.06e−4	1.95e0 ± 3.48e−2
15–40	1.58e0 ± 7.16e−2	1.69e0 ± 2.72e−2	1.22e0 ± 7.29e−2	**1.16e0 ± 1.07e−3**	1.28e0 ± 1.00e−3	1.43e0 ± 4.27e−2
50–80	2.96e1 ± 1.91e0	3.15e1 ± 1.56e0	**2.45e1 ± 1.28e0**	2.52e1 ± 7.57e−1	3.08e1 ± 9.70e−1	3.47e1 ± 5.98e−1
100–140	3.64e2 ± 7.15e0	3.69e2 ± 2.08e0	**3.22e2 ± 3.18e−1**	3.42e2 ± 7.09e0	4.19e2 ± 7.09e0	4.28e2 ± 1.79e1
2–200	1.88e2 ± 1.12e0	2.01e2 ± 3.98e0	**1.00e2 ± 8.04e−2**	1.07e2 ± 1.91e0	1.06e2 ± 2.30e0	1.07e2 ± 5.84e0

The shortest runtimes are in bold

Table 13 Best TPB configuration of GPU versions for KP

Instances	SGA	EGA	SGS_E	SGS_V	SGS_B
100–1,000	32	32	64	64	64
100–10,000	32	32	64	64	64
200–1,000	32	32	64	64	64
200–10,000	32	32	64	64	64
500–1,000	64	64	128	128	128
500–10,000	64	64	128	128	128
1,000–1,000	64	64	128	128	128
1,000–10,000	64	64	128	128	128

Table 14 Best TPB configuration of GPU versions for MMDP

Instances	SGA	EGA	SGS_E	SGS_V	SGS_B
300	32/64	64	64	64	64
600	64	64	64	64	64
900	64	64	128	128	128
1200	64	64	128	128	128
1500	64	64	128	128	128

Table 15 Best TPB configuration of GPU versions for NRP

Instances	SGA	EGA	SGS_E	SGS_V	SGS_B
100–20	32	32	32	32	32
100–25	32	32	32	32	32
35–35	32	32	32/64	64	64
15–40	32	32	32/64	64	64
50–80	32	32	32	32	32/64
100–140	32	32	64	64	64
2–200	32	64	64	64	64

Now, we analyze the performance of the GPU versions. Considering the features of the GPU platform used in this work, executions with 32, 64, 128 and 256 Threads Per Block (TPB) were made. Tables 13, 14, and 15 show the best configuration of TPB of the algorithms studied for each problem and instance, i.e., the TPB configuration with the shortest execution time. The numerical efficiency of the GPU implementations was also studied, reaching similar conclusions as those drawn for the CPU versions, but these results are not included in this article because of its huge extension.

Tables 16, 17, and 18 show the mean runtime in seconds and the standard deviation of the algorithms implemented on GPU using the best TPB configuration on each problem and instance for the KP, MMDP and NRP, respectively. The results show that the SGS algorithms are also the best performing algorithms when implemented on GPU. In particular, SGS_B is the algorithm with the shortest runtime in all the instances of the three problems studied, e.g., SGS_B needs only 14.70 s for instance 100–10,000 of the KP, 35.65 s for instance 1,500 of MMDP and 1.32 s for instance 100–140 of NRP, which is more than 2× faster than both GAs.

Let us now analyze the improvement in performance of GPU over CPU implementations. To this end, we use the ratio between the wall-clock time of the CPU and the GPU executions of each algorithm. Even though some authors make reference to this metric as speedup, we prefer to refer to this ratio as runtime reduction. The use of the term speedup can give a misleading idea on how parallelizable is the GPU implementation of an algorithm since the execution times are measured in two different platforms. Tables 19, 20 and 21 show the runtime reduction of GPU versions vs. the CPU versions for KP, MMDP and NRP.

The runtime reduction of SGS algorithms is up to 62.86× (SGS_B in 1,000–1,000) for the KP, 112.74× (SGS_V in 1,500) for the MMDP and 324.08× (SGS_B in 100–140) for the NRP, while for GAs it is up to 72.22× (EGA in 1,000–1,000), 110.86× (EGA in 1,500) and 110.94× (EGA in 100–140), respectively.

The tendency is clear, the larger the instance, the higher the time reduction. The reason is twofold. On the one hand, larger tentative solutions allow all the algorithms to better profit from the parallel computation of the threads and, on the other hand, the algorithms use larger populations when the size of the instances increases (the grid has to be enlarged to meet the SGS structured search model), so a higher num-

Table 16 Runtime in seconds of GPU versions for KP (mean ± std. dev.)

Instance	SGA	EGA	SGS_E	SGS_V	SGS_B
100–1,000	7.54e−2 ± 5.48e−4	7.02e−2 ± 4.31e−4	3.70e−2 ± 1.41e−4	3.44e−2 ± 4.99e−4	**3.40e−2 ± 2.47e−4**
100–10,000	7.68e−2 ± 4.60e−4	7.24e−2 ± 4.90e−4	3.70e−2 ± 3.48e−4	3.46e−2 ± 4.99e−4	**3.41e−2 ± 3.73e−4**
200–1,000	4.48e−1 ± 6.30e−4	4.41e−1 ± 4.63e−4	2.26e−1 ± 4.05e−4	2.11e−1 ± 4.63e−4	**2.10e−1 ± 4.43e−4**
200–10,000	4.52e−1 ± 5.55e−4	4.31e−1 ± 4.99e−4	2.22e−1 ± 4.31e−4	2.06e−1 ± 5.01e−4	**2.05e−1 ± 4.71e−4**
500–1,000	4.58e0 ± 2.17e−3	4.51e0 ± 6.13e−4	2.28e0 ± 7.51e−4	2.15e0 ± 8.18e−4	**2.13e0 ± 7.42e−4**
500–10,000	4.59e0 ± 2.20e−3	4.51e0 ± 5.28e−3	2.29e0 ± 6.52e−4	2.15e0 ± 6.69e−4	**2.14e0 ± 8.48e−4**
1,000–1,000	3.43e1 ± 1.67e−2	3.40e1 ± 9.52e−4	1.53e1 ± 2.39e−3	**1.45e1 ± 2.70e−3**	**1.45e1 ± 2.87e−3**
1,000–10,000	3.42e1 ± 1.71e−2	3.49e1 ± 1.53e−3	1.55e1 ± 3.33e−3	1.48e1 ± 3.18e−3	**1.47e1 ± 3.19e−3**

The shortest runtimes are in bold

Table 17 Runtime in seconds of GPU versions for MMDP (mean ± std. dev.)

Instance	SGA	EGA	SGS_E	SGS_V	SGS_B
300	1.15e0 ± 6.58e−4	1.12e0 ± 7.12e−4	5.43e−1 ± 3.25e−3	5.01e−1 ± 2.35e−3	**4.97e−1 ± 1.72e−3**
600	8.00e0 ± 9.86e−4	7.88e0 ± 4.10e−3	3.06e0 ± 3.02e−3	2.83e0 ± 1.37e−3	**2.81e0 ± 2.68e−3**
900	2.34e1 ± 1.58e−3	2.26e1 ± 4.77e−2	8.85e0 ± 4.68e−2	8.26e0 ± 2.47e−3	**8.21e0 ± 1.63e−3**
1,200	5.48e1 ± 4.96e−2	5.37e1 ± 8.02e−2	2.10e1 ± 4.60e−2	1.96e1 ± 2.89e−3	**1.95e1 ± 2.60e−3**
1,500	9.70e1 ± 7.10e−2	9.70e1 ± 1.89e−1	3.81e1 ± 2.88e−3	3.58e1 ± 3.08e−3	**3.57e1 ± 5.65e−3**

The shortest runtimes are in bold

Table 18 Runtime in seconds of GPU versions for NRP (mean ± std. dev.)

Instance	SGA	EGA	SGS_E	SGS_V	SGS_B
100–20	6.04e−2 ± 5.81e−4	5.72e−2 ± 5.66e−4	5.25e−2 ± 6.08e−4	**5.15e−2 ± 5.38e−4**	**5.15e−2 ± 5.74e−4**
100–25	7.68e−2 ± 5.39e−4	7.31e−2 ± 8.29e−4	6.36e−2 ± 5.38e−4	**6.15e−2 ± 5.80e−4**	**6.15e−2 ± 5.42e−4**
35–35	1.16e−1 ± 7.60e−4	1.11e−1 ± 4.04e−4	8.79e−2 ± 1.02e−3	8.46e−1 ± 5.35e−4	**8.36e−2 ± 5.98e−4**
15–40	1.30e−1 ± 1.09e−3	1.24e−1 ± 5.15e−4	9.52e−2 ± 5.35e−4	9.11e−1 ± 3.96e−4	**9.07e−2 ± 6.00e−4**
50–80	5.31e−1 ± 5.85e−4	5.11e−1 ± 4.20e−3	3.56e−1 ± 4.22e−4	3.37e−1 ± 5.75e−4	**3.35e−1 ± 5.25e−4**
100–140	3.79e0 ± 6.95e−3	3.66e0 ± 9.09e−3	1.40e0 ± 7.40e−4	1.33e0 ± 5.80e−4	**1.32e0 ± 7.85e−3**
2–200	6.71e0 ± 1.21e−2	6.42e0 ± 7.35e−3	4.65e0 ± 9.11e−3	4.20e0 ± 5.65e−3	**4.18e0 ± 8.12e−3**

The shortest runtimes are in bold

ber of blocks have to be generated and the algorithm takes advantage of the computing capabilities offered by the GPU architecture. The reductions obtained by SGS for the MMDP and the NRP are larger than the ones obtained by GAs except for instance 2–200. In KP, the reductions obtained by GAs are slightly larger than the ones obtained by SGS, since both implementations follow a similar scheme, while sequential GAs are computationally more expensive than sequential SGS and have some additional features that can be parallelized (e.g., mutation, random number generation).

Finally, we study the comparative performance among CPU and GPU implementations.

To this end, we analyze the normalized number of solutions built and evaluated for each algorithm per second (the unit is the number of solutions constructed by the slower algorithm for each instance). Figures 11 and 12 graphically show the normalized number of solutions built and evaluated by CPU implementations for KP and MMDP, while Figs. 13 and 14 graphically show the normalized number of solu-

Table 19 Runtime reduction of GPU vs CPU versions for KP

Instance	SGA	EGA	SGS_E	SGS_V	SGS_B
100–1,000	16.84	**19.37**	19.19	19.19	19.12
100–10,000	17.58	19.61	19.73	19.08	**20.53**
200–1,000	29.11	**30.79**	27.08	26.26	27.14
200–10,000	28.58	**31.93**	27.39	28.40	28.05
500–1,000	54.95	**58.41**	47.32	48.10	49.78
500–10,000	54.04	**59.47**	45.16	45.67	48.28
1,000–1,000	70.50	**75.14**	60.77	60.92	62.86
1,000–10,000	69.81	**72.22**	58.78	58.68	60.69

The best values are in bold

Table 20 Runtime reduction of GPU versions vs CPU versions for MMDP

Instance	SGA	EGA	SGS_E	SGS_V	SGS_B
300	48.83	52.76	50.50	49.96	**54.61**
600	67.18	72.75	80.28	**84.64**	83.89
900	79.09	84.46	88.75	**93.45**	90.50
1,200	95.81	102.02	103.60	**107.64**	107.47
1,500	104.91	110.86	110.91	**112.74**	112.12

The best values are in bold

Table 21 Runtime reduction of GPU vs CPU versions for NRP

Instance	SGA	EGA	SGS_E	SGS_V	SGS_B
100–20	7.52	8.23	7.66	9.34	**10.72**
100–25	11.88	13.20	12.27	15.93	**18.21**
35–35	15.26	16.94	17.63	20.21	**23.33**
15–40	12.15	13.63	12.18	14.05	**15.77**
50–80	55.82	61.59	70.81	91.34	**103.52**
100–140	96.10	100.94	244.36	315.26	**324.08**
2–200	28.03	**31.34**	23.05	25.12	25.60

The best values are in bold

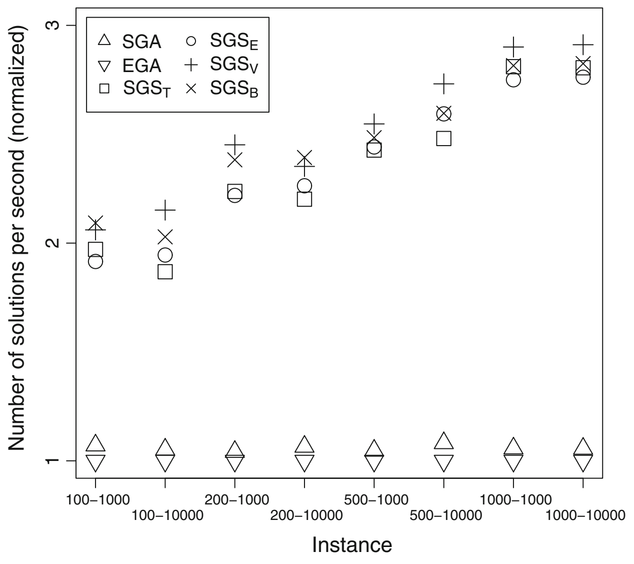


Fig. 11 Performance of CPU implementations for KP

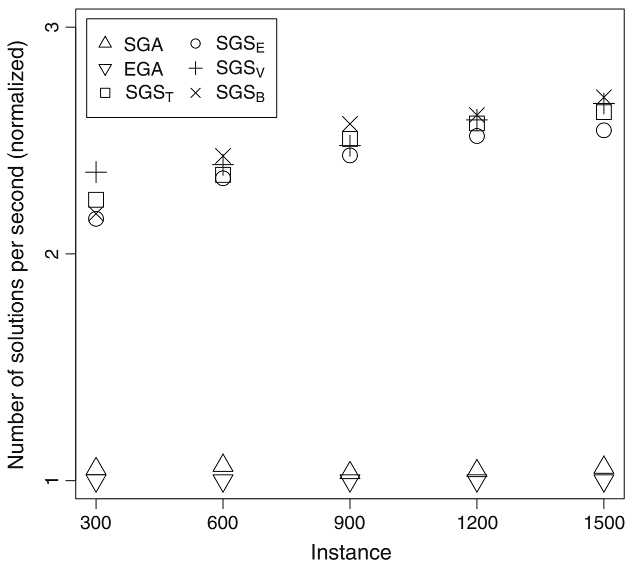


Fig. 12 Performance of CPU implementations for MMDP

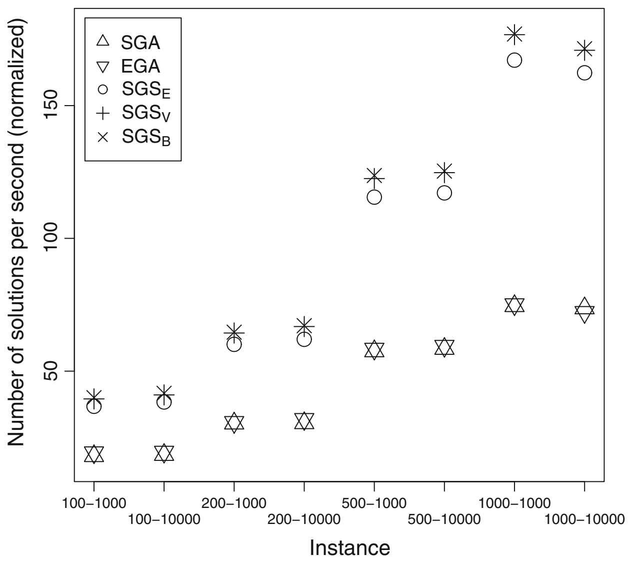


Fig. 13 Performance of GPU implementations for KP

tions built and evaluated by GPU implementations for KP and MMDP. For NRP, as the results of the CPU versions are irregular and there are large differences between the magnitudes of the results of the GPU versions, we have chosen to not include plots, and the results are summarized in Table 22.

The results obtained show that the CPU implementation of SGS can build and evaluate solutions more than two times faster than both GAs for almost all the instances considered in the experimental evaluation of the KP and MMDP. Additionally, the trend can be clearly seen in the figure: the larger the instance considered, the larger the improvement over the number of solutions build and evaluated by both GAs. In NRP, the results obtained regarding the number of solutions built and evaluated per second for the CPU implementations are somewhat irregular, having a great variability. The behavior is fairly different from the behavior seen in the first two problems analyzed. The improvement of the best performing algorithm (SGS_T in five out of seven instances and SGS_E in four out of seven instances) over the algorithm with the longest execution time for each instance ranges from 1.26 (35–35) to 2.01 (2–200).

On the other hand, the results obtained show that the GPU implementation of SGS can build and evaluate solutions up to more than 150 and 260 times faster than the CPU imple-

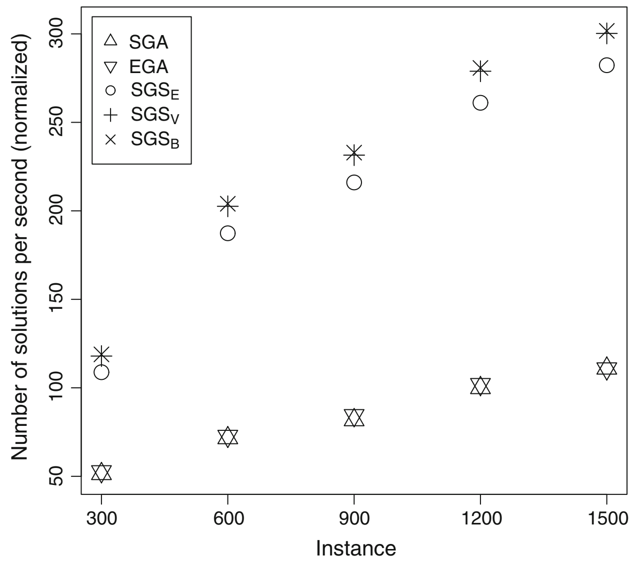


Fig. 14 Performance of GPU implementations for MMDP

mentation of both GAs for the largest instances considered in the experimental evaluation of the KP and MMDP, respectively. In particular, SGS_B, the best performing algorithm, improves upon the sequential SGA up to 166× and 284×, and upon the sequential EGA up to 176× and 301× for KP and MMDP, respectively. Additionally, the clear trend shown in the figures is that, when the larger the instance considered, the larger the improvement over the number of solutions build and evaluated by both sequential GAs and, what is more relevant, the larger the difference between the improvement of the GPU implementation of SGS and the improvement of the GPU implementations of both GAs.

Finally, to compare the improvement in performance for different instances of the NRP, it should be taken into account that the complexity of an instance is not only influenced by the number of requirements but also is influenced by the number of customers. For this reason, it is not easy to exactly establish which instances are larger than others. To analyze the scalability of the GPU implementations, we consider that the size of the instance can be inferred from the value of the normalized number of solutions, i.e., the larger the value, the larger the instance. Taking as a criterion the order of the instances determined by the normalized number of solutions of the GPU implementation of SGA, the tendency is similar than the one obtained for the first two problems, when the larger the instance considered, the larger the improvement over the sequential algorithm with the longest execution time on CPU and the larger the difference between the improvement of the SGS over both GAs on GPU.

The results obtained in the study of the performance of the GPU implementation of SGS are superior than reductions that are often found in the literature of metaheuristics on GPUs (10–20×). Additionally, the idea of SGS has proven that is highly scalable, making these algorithms an interesting alternative to unleash the potential of GPU platforms for new applications.

5 Related work

This section analyzes published material which is related to the SGS algorithm presented in this work. First of all, to the best of our knowledge, the SGS algorithm is a newly fresh research line developed by the authors which has been preliminary explored in Pedemonte et al. (2012), Pedemonte et al. (2013). Besides the seminal works of Systolic Computing by Kung (1982), and Kung and Leiserson (1978), only few subsequent trials have been devoted to engineer optimization algorithms based on this paradigm. Indeed, only Chan and Mazumder (1995) and Megson and Bland (1998) implemented a GA on VLSI and FPGA architectures in a systolic fashion, but this lines were early discarded due to the complexity of translating the GA operations into the recurrent equations required to define the hardware. More recently, Alba and Vidal (2011), and Vidal et al. (2013) have proposed SNS (*Systolic Neigborhood Search*). SGS can be seen as an

Table 22 Normalized number of solutions constructed per second for NRP

Instance	CPU versions						GPU versions				
	SGA	EGA	SGS_T	SGS_E	SGS_V	SGS_B	SGA	EGA	SGS_E	SGS_V	SGS_B
100–20	1.22	1.17	1.32	**1.37**	1.15	1.00	9.14	9.65	10.51	**10.72**	**10.72**
100–25	1.23	1.16	**1.33**	**1.33**	1.14	1.00	14.58	15.32	17.61	**18.21**	**18.21**
35–35	1.10	1.04	**1.26**	**1.26**	1.14	1.00	16.81	17.57	22.18	23.05	**23.33**
15–40	1.07	1.00	1.39	**1.46**	1.32	1.18	13.00	13.63	17.75	18.55	**18.63**
50–80	1.17	1.10	**1.41**	1.38	1.13	1.00	65.31	67.87	97.42	102.91	**103.52**
100–140	1.17	1.16	**1.33**	1.25	1.02	1.00	112.87	116.88	305.56	321.65	**324.08**
2–200	1.07	1.00	**2.01**	1.88	1.91	1.88	29.99	31.34	43.28	47.91	**48.14**

The best values are in bold

advanced version of SNS. They share the arrangement of solutions into a grid, but SNS only circulates solutions in a row fashion, whereas SGS moves solutions not only horizontally but also vertically. This means that each cell has to manage pairs of solutions and thus more complex search strategies can be devised.

Finally, SGS (and SNS as well) have similarities with the cellular model of EAs (Alba and Dorronsorso 2008), but there are strong conceptual design goals that make the two underlying search models fairly different. In the cellular model, the population is structured in overlapping neighborhoods with interactions between individuals limited to those neighborhoods. Although in a first impression the models look alike, the only point of contact of both models is that the solutions are placed in a structured grid. Two main differences emerge.

First, the information flow in both models is quite different. While the solutions remain static in the same position of the grid and all the exchange of information among solutions is caused by the overlap of neighborhoods in the cellular model, SGS is based on the flow of solutions. That is, the constant movement of all the information through the grid produces the communication between the solutions. As a consequence, the solutions that could be mated in SGS is dynamic during the execution of the algorithm, while in the cellular model the mating is static, i.e., a given solution can only be mated with the same set of solutions for the whole execution. This clearly introduces a higher diversity in the search.

Second, each cell applies the evolutionary operators to produce new solutions independently of the other cells in SGS, i.e., when a cell is applying those operators it can be considered isolated from the rest of the grid, while in the cellular model each cell needs the neighboring cells to be able to produce new solutions. Also, the SGS search aims at being systematic: the search operators in each cell has an structured pattern rather than the pure stochastic approach of cellular EAs. As a final remark, we would like to point out that the cellular model has been ported to CUDA too, so as to allow its deployment on GPU cards (Vidal and Alba 2010a,b).

6 Conclusions and future work

In this work, we have presented a new parallel optimization algorithm that combines ideas from the fields of metaheuristics and Systolic Computing, the Systolic Genetic Search algorithm. The algorithm is inspired by the systolic contraction of the heart that makes possible that it pumps blood rhythmically according to the metabolic needs of the tissues and is designed to explicitly exploit the high degree of parallelism available in modern devices such as Graphics Processing Units. An exhaustive experimental evaluation was conducted using four different instantiations of SGS, a Random Search and two GAs for solving two classical benchmarking problems (including one deceptive problem) and a real-world application on twenty different instances.

The experimental evaluation shows that $\mathrm{SGS_E}$, $\mathrm{SGS_V}$ and $\mathrm{SGS_B}$ flavors have a great potential. These algorithms have shown to be highly effective for solving the three problems considered as they are able to find the optimal solution in almost every run for each instance. Additionally, these three instantiations of SGS outperform the remaining algorithms involved in the experiment for KP and NRP, as well as RS and EGA for MMDP.

The parallel implementation on GPU of these three algorithms has achieved a high performance obtaining runtime reductions from their corresponding sequential implementation that, depending on the instance considered, can arrive to hundred times. In particular, parallel GPU-based implementation of $\mathrm{SGS_B}$ is the best performing algorithm of the whole experimental evaluation, having systematically the shortest runtime for all the instances of all the problems considered. The runtime reduction of the parallel GPU-based implementation of $\mathrm{SGS_B}$ with respect to its sequential implementation is up to $62\times$ for the KP, $112\times$ for the MMDP and $324\times$ for the NRP. Additionally, if the performance is evaluated using the improvement in the number of solutions constructed and evaluated relative to the sequential algorithm with the longest runtime on CPU, the parallel GPU-based implementation of $\mathrm{SGS_B}$ improvement in performance is up to $176\times$ for the KNSP, $301\times$ for the MMDP and $324\times$ for the NRP. Finally, it should be highlighted that $\mathrm{SGS_E}$, $\mathrm{SGS_V}$ and $\mathrm{SGS_B}$ on GPU have shown a good scalability behavior when solving high-dimension problem instances.

Three main areas that deserve further study are identified. A first issue is to customize the GPU implementation of SGS to the new Kepler architecture to assess the improvement in performance that can be obtained in these new devices. A second line of interest is to study theoretically and empirically the impact of the values of the crossover and mutation points, and how these values are distributed in the systolic grid, in the quality of the solutions obtained by SGS. Given the results obtained, we also want to go for an accurate scalability study of this search model. Finally, we aim to perform a wider impact analysis by solving additional problems to extend the existing evidence of the benefits of this line of research.

Acknowledgments Martín Pedemonte acknowledges support from Programa de Desarrollo de las Ciencias Básicas, Universidad de la República, and Agencia Nacional de Investigación e Innovación, Uruguay. Francisco Luna and Enrique Alba acknowledge partial support from the Spanish Ministry of Economy and Competitiveness and FEDER under contract TIN2011-28194. Francisco Luna also acknowledges partial support from TIN2011-28336. The authors would like to thank to M.Sc. Leonella Luzardo for her valuable comments and suggestions to improve the description of the biological phenomenon that inspires Systolic Computing and systolic based metaheuristics. The authors would also like to thank to the anonymous reviewers for their insightful and constructive suggestions.

References

Alba E (ed) (2005) Parallel metaheuristics: a new class of algorithms. Wiley, New York

Alba E, Dorronsorso B (eds) (2008) Cellular genetic algorithms. Springer, New York

Alba E, Vidal P (2011) Systolic optimization on GPU platforms. In: 13th international conference on computer aided systems theory (EUROCAST 2011)

Bagnall A, Rayward-Smith V, Whittley I (2001) The next release problem. Inf Softw Technol 43(14):883–890

Blum C, Roli A (2003) Metaheuristics in combinatorial optimization: overview and conceptual comparison. ACM Comput Surv 35(3):268–308

Cecilia JM, García JM, Ujaldon M, Nisbet A, Amos M (2011) Parallelization strategies for ant colony optimisation on gpus. In: 25th IEEE international symposium on parallel and distributed processing, IPDPS 2011, workshop proceedings, pp 339–346

Chan H, Mazumder P (1995) A systolic architecture for high speed hypergraph partitioning using a genetic algorithm. In: Yao X (ed) Progress in evolutionary computation, vol 956., Lecture Notes in Computer ScienceSpringer, Berlin, pp 109–126

Deb K (2001) Multi-objective optimization using evolutionary algorithms. Wiley, New York

Durillo JJ, Zhang Y, Alba E, Harman M, Nebro AJ (2011) A study of the bi-objective next release problem. Empirical Softw Eng 16(1):29–60

Furber S (2000) ARM system-on-chip architecture, 2nd edn. Addison-Wesley Longman Publishing Co., Inc.

Gaster B, Howes L, Kaeli D, Mistry P, Schaa D (2012) Heterogeneous computing with OpenCL, 2nd edn. Morgan Kaufmann

Goldberg D, Deb K, Horn J (1992) Massively multimodality, deception and genetic algorithms. In: Proceedings of the international conference on parallel problem solving from nature II (PPSNII), pp 37–46

Guyton AC, Hall JE (2006) Textbook of medical physiology, 11th edn. Elsevier Saunders

Harding S, Banzhaf W (2011) Implementing cartesian genetic programming classifiers on graphics processing units using gpu.net. In: 13th annual genetic and evolutionary computation conference, GECCO 2011, companion material, pp 463–470

Hennessy J, Patterson D (2011) Computer architecture: a quantitative approach. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann

Intel Corporation (2013a) Intel xeon phi core micro-architecture. White paper, Intel Corporation. http://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture

Intel Corporation (2013b) Intel xeon phi product family: performance brief. White paper, Intel Corporation. http://www.intel.com/content/www/us/en/benchmarks/xeon-phi-product-family-performance-brief.html

Kirk D, Hwu W (2012) Programming Massively parallel processors. A hands-on approach. 2nd edn. Morgan Kaufmann

Kung HT (1982) Why systolic architectures? Computer 15(1):37–46

Kung HT, Leiserson CE (1978) Systolic arrays (for VLSI). In: Sparse matrix proceedings, pp 256–282

Langdon WB (2011) Graphics processing units and genetic programming: an overview. Soft Comput 15(8):1657–1669

Langdon WB, Banzhaf W (2008) A simd interpreter for genetic programming on gpu graphics cards. In: Genetic programming, 11th European conference, EuroGP 2008. Proceedings, Springer, Lecture Notes in Computer Science, vol 4971, pp 73–85

Lewis TE, Magoulas GD (2009) Strategies to minimise the total run time of cyclic graph based genetic programming with gpus. Genetic and evolutionary computation conference, GECCO 2009, pp 1379–1386

Libby P, Bonow R, Mann D, Zipes D (2007) Braunwald's heart disease: a textbook of cardiovascular medicine. Elsevier Health Sciences

Maitre O, Krüger F, Querry S, Lachiche N, Collet P (2012) Easea: specification and execution of evolutionary algorithms on gpgpu. Soft Comput 16(2):261–279

Marler R, Arora J (2004) Survey of multi-objective optimization methods for engineering. Struct Multidiscip Optim 26(6):369–395

McCool MD, Robison AD, Reinders J (2012) Structured parallel programming, patterns for efficient computation. Morgan Kaufmann

Megson G, Bland I (1998) Synthesis of a systolic array genetic algorithm. In: Parallel processing symposium, 1998. IPPS/SPDP 1998, pp 316–320

Miettinen K (1999) Nonlinear multiobjective optimization. International series in operations research and management science. Kluwer Academic Publishers

Nvidia Corporation (2009) NVIDIA's next generation CUDA compute architecture: fermi. Nvidia Corporation, Whitepaper

Nvidia Corporation (2012a) CUDA C Best Practices Guide Version 5.0. Nvidia Corporation

Nvidia Corporation (2012b) CUDA Toolkit 5.0 CURAND Guide. Nvidia Corporation

Nvidia Corporation (2012c) NVIDIA CUDA C Programming Guide Version 5.0. Nvidia Corporation

Nvidia Corporation (2012d) NVIDIA's next generation CUDA compute architecture: Kepler GK110. Whitepaper, the fastest, most efficient HPC architecture ever built. Nvidia Corporation

Owens JD, Luebke D, Govindaraju N, Harris M, Krnger J, Lefohn A, Purcell TJ (2007) A survey of general-purpose computation on graphics hardware. Comput Graphics Forum 26(1):80–113

Pedemonte M, Alba E, Luna F (2011) Bitwise operations for gpu implementation of genetic algorithms. In: Genetic and evolutionary computation conference, GECCO'11. Companion Publication, pp 439–446

Pedemonte M, Alba E, Luna F (2012) Towards the design of systolic genetic search. In: IEEE 26th international parallel and distributed processing symposium workshops and PhD Forum. IEEE Computer Society, pp 1778–1786

Pedemonte M, Luna F, Alba E (2013) New ideas in parallel metaheuristics on gpu: systolic genetic search. In: Tsutsui S, Collet P (eds) Massively parallel evolutionary computation on GPGPUs, Natural Computing Series, chap 10. Springer, Berlin, pp 203–225

Pisinger D (1997) A minimal algorithm for the 0–1 knapsack problem. Oper Res 45:758–767

Pisinger D (1999) Core problems in knapsack algorithms. Oper Res 47:570–575

Sheskin DJ (2011) Handbook of parametric and nonparametric statistical procedures, 5th edn. Chapman and Hall/CRC

Soca N, Blengio J, Pedemonte M, Ezzatti P (2010) PUGACE, a cellular evolutionary algorithm framework on GPUs. In: 2010 IEEE world congress on computational intelligence. WCCI 2010–2010 IEEE Congress on Evolutionary Computation, CEC 2010, pp 1–8

Tsutsui S, Fujimoto N (2011) Fast qap solving by aco with 2-opt local search on a gpu. In: 2011 IEEE congress of evolutionary computation, CEC 2011, pp 812–819

Veronese LDP, Krohling RA (2010) Differential evolution algorithm on the gpu with c-cuda. In: Proceedings of the IEEE congress on evolutionary computation, CEC 2010, pp 1–7

Vidal P, Alba E (2010a) Cellular genetic algorithm on graphic processing units. In: Nature inspired cooperative strategies for optimization (NICSO 2010), pp 223–232

Vidal P, Alba E (2010b) A multi-gpu implementation of a cellular genetic algorithm. In: IEEE congress on evolutionary computation, pp 1–7

Vidal P, Luna F, Alba E (2013) Systolic neighborhood search on graphics processing units. Soft Computing, pp 1–18

Zhang Q, Li H (2007) Moea/d: a multiobjective evolutionary algorithm based on decomposition. IEEE Trans Evol Comput 11(6):712–731

Zhang S, He Z (2009) Implementation of parallel genetic algorithm based on CUDA. In: ISICA 2009, LNCS 5821, pp 24–30

Zhang Y, Harman M, Mansouri SA (2007) The multi-objective next release problem. In: Proceedings of the 9th annual conference on genetic and evolutionary computation, ACM, GECCO '07, pp 1129–1137

Zhou Y, Tan Y (2009) Gpu-based parallel particle swarm optimization. In: Proceedings of the IEEE congress on evolutionary computation, CEC 2009, pp 1493–1500

Appendix B

A Systolic Genetic Search for Reducing the Execution Cost of Regression Testing

Applied Soft Computing 49 (2016) 1145–1161

Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

A Systolic Genetic Search for reducing the execution cost of regression testing

Martín Pedemonte[a,*], Francisco Luna[b], Enrique Alba[b]

[a] Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Julio Herrera y Reissig 565, 11300 Montevideo, Uruguay
[b] Depto. de Lenguajes y Ciencias de la Computación, Univ. de Málaga, E.T.S. Ingeniería Informática, Campus de Teatinos, 29071 Málaga, Spain

ARTICLE INFO

Article history:
Received 31 December 2015
Received in revised form 16 June 2016
Accepted 4 July 2016
Available online 15 July 2016

Keywords:
Regression testing
Evolutionary algorithms
Parallel metaheuristics
GPU
CUDA

ABSTRACT

The Test Suite Minimization Problem (TSMP) is a $\mathcal{NP}$-hard real-world problem that arises in the field of software engineering. It consists in selecting a minimal set of test cases from a large test suite, ensuring that the test cases selected cover a given set of requirements of a piece of software at the same time as it minimizes the amount of resources required for its execution. In this paper, we propose a Systolic Genetic Search (SGS) algorithm for solving the TSMP. SGS is a recently proposed optimization algorithm capable of taking advantage of the high degree of parallelism available in modern GPU architectures. The experimental evaluation conducted on a large number of test suites generated for seven real-world programs and seven large test suites generated for a case study from a real-world program shows that SGS is highly effective for the TSMP. SGS not only outperforms two competitive genetic algorithms, but also outperforms four heuristics specially conceived for this problem. The results also show that the GPU implementation of SGS has achieved a high performance, obtaining a large runtime reduction with respect to the CPU implementation for solutions with similar quality. The GPU implementation of SGS also shows an excellent scalability behavior when solving instances with a large number of test cases. As a consequence, the GPU-based SGS stands as a state of the art alternative for solving the TSMP in real-world software testing environments.

1. Introduction

Software testing is one of the core activities of the software development process. It involves the execution of a piece of software to gather information for evaluating the quality of that software. In general, this execution uses test cases that are designed for exercising at least one feature (functional or non-functional) of the piece of software under evaluation. Although initially testing was considered necessary evil, it has become a key aspect of software development process. Testing has been reported to represent more than fifty percent of the cost of software development [1], while the total labor resources spent in testing range from 30 to 90% [2].

Regression testing is a testing activity performed to ensure that changes made to an existing piece of software do not introduce errors. During the evolution of a piece of software, the software tends to grow in size and complexity. This evolution also provokes that new test cases are continually generated and added to the test suite to validate the latest modifications to the piece of software. For this reason, the execution of the entire test suite can be impracticable. For instance, in a real-world test suite for regression testing from Cisco containing 2320 test cases [3], the runtime of each test case takes about 10–100 min, yielding a final total execution time of the test suite of around 5 weeks. In consequence, different approaches have been proposed in order to reduce the effort devoted to regression testing [4].

Search-based software engineering (SBSE) [5] is one recent field in Software Engineering that is based in applying search-based optimization techniques (as Evolutionary Algorithms, EAs) to software engineering problems. SBSE has been successfully used to solve problems from all the phases of the software development process, being software testing one of the most addressed issues [5]. In particular, the Test Suite Minimization Problem (TSMP) is a $\mathcal{NP}$-hard real-world software testing problem that arises in regression testing. It is based on reducing a large test suite by removing redundant test cases, ensuring that a set of test goals are satisfied [4]. The goal is to find a reduced test suite that minimizes the overall cost of the suite and that covers a given set of elements of the piece of software that is being tested.

As realistic software programs involve thousands of lines of code (and so the test suites used for their testing have thousands of test cases) exact algorithms are discarded for this problem because they could lead to huge computing times. Even metaheuristics may

* Corresponding author. Tel.: +598 27114244x1048.
E-mail addresses: mpedemon@fing.edu.uy (M. Pedemonte), flv@lcc.uma.es (F. Luna), eat@lcc.uma.es (E. Alba).

http://dx.doi.org/10.1016/j.asoc.2016.07.018
1568-4946/© 2016 Elsevier B.V. All rights reserved.

be highly computationally expensive when addressing real-world TSMP instances. In order to tackle this problem properly, we make use of parallel metaheuristics [6]. These algorithms do not only allow to reduce the runtime of the algorithms, but also usually provide new enhanced search engines that could lead to improve the quality of results obtained by traditional sequential algorithms. Despite their advantages, there are very few works that use parallel metaheuristics for solving SBSE problems [7].

Systolic Genetic Search (SGS) is a new optimization algorithm that merges ideas from systolic computing and metaheuristics [8]. SGS was conceived to exploit the high degree of parallelism available in modern GPU architectures. It has already shown its potential for the knapsack problem, the massively multimodal deceptive problem, and the next release problem, finding optimal or near optimal solutions in short runtimes [8].

In the present article we investigate the use of a SGS algorithm for solving the cost-aware Test Suite Minimization Problem. Our first concern is to show whether SGS is effective for this problem. With this in mind, a comparative study on the numerical performance is conducted between SGS, two EAs and four heuristics specially designed for the problem at stake. Since SGS is explicitly designed for GPU architectures, a second issue is to evaluate if the GPU implementation of the proposed SGS is efficient and what kind of performance benefits can be obtained by such implementation with respect to a CPU implementation. To this end, we comparatively analyze the performance of the implementation on GPU of SGS and two evolutionary algorithms. Finally, our third concern is how well the number of test cases and test goals impact in the performance of the CPU and GPU implementations of SGS. We can summarize the contributions of this work as follows:

- It presents a new success of SGS for solving an optimization problem in a unexplored domain. The results obtained are not only relevant for the SGS but also are relevant for the cost-aware TSMP. Because SGS is highly effective for solving instances from seven real-world programs and a case study from a real-world program, without using any problem-specific knowledge, outperforming four heuristics specifically designed for this problem.
- It shows that the GPU implementation of SGS is able to achieve a high performance, obtaining a large runtime reduction compared to the sequential implementation for similar solutions. Moreover, the GPU-based SGS is the EA with the best performance of this study. In consequence, the GPU-based SGS is able to both obtain excellent quality solutions and execute in a short runtime.
- It shows that the GPU-based SGS has an excellent scalability behavior when solving instances with a larger number of test cases. On the other hand, when a larger number of test goals is considered, the performance of the GPU-based SGS is just minimally degraded.

This article is organized as follows. Section 2 reviews the preliminaries of this work. Then, in Section 3, we describe the SGS algorithm and how it is instantiated for tackling the TSMP. Section 4 presents state of the art heuristic techniques for the cost-aware TSMP that are used for comparison in this work. Then, Section 5 describes the details of the empirical study and analyzes the results of the experimental evaluation. Threats to validity are discussed on Section 6. Then, Section 7 discusses the related papers in the literature. Finally, in Section 8, we outline the conclusions of this work and suggest future research directions.

2. Preliminaries

In this section we present some background on the TSMP and on the architecture of the GPUs.

2.1. Test Suite Minimization Problem

Three different problem formulations have been proposed for test suite reduction in regression testing [4]. The Test Case Selection Problem consists in choosing a subset of the test suite based on which test cases are relevant for testing the changes between the previous and the current version of the piece of software. Another approach is known as the Test Case Prioritization Problem, which consists in finding an ordering of the test cases according to some specific criteria, such that test cases ordered first should be run first. In this formulation, it is assumed that all the test cases could be executed but the testing process could be stopped at some arbitrary point. As a consequence, if the test processing is stopped, the test cases that maximize the specific criteria used for ordering them have already been executed.

In the present work, we adopt the problem formulation known as Test Suite Minimization Problem (TSMP) [4,9]. TSMP belongs to the class of $\mathcal{NP}$-hard problems since it is equivalent to the Minimal Hitting Set Problem. It lies in reducing a test suite by eliminating redundant test cases, and the goal is to select a minimal set of test cases that cover a set of test goals and minimizes the amount of resources required for its execution. It is formally defined as follows.

Let $T=\{t_1, \ldots, t_n\}$ be a test suite with n test cases for a piece of software and $R=\{r_1, \ldots, r_m\}$ be the set of m test goals (requirements) that has to be covered with the test cases. Each test case covers several test goals and this relation is represented by a coverage matrix $M=[m_{ij}]$ of dimension $n \times m$, whose entries are either 0 or 1. If $m_{ij}=1$ the test case i covers the test goal j, otherwise it does not covers the test goal. Also, each test case t_i has associated a positive cost c_i that measures the amount of resources required for its execution.

The single objective TSMP consists in finding a subset of test cases of the original test suite that covers all the test goals (100% of coverage) and minimizes its overall cost. The single objective TSMP can be formulated as the integer programming model presented in Eqs. (1)–(3), being x_i the binary decision variables of the problem that indicate whether the test case t_i is included or not in the reduced test suite.

$$\text{minimize} \quad \sum_{i=1}^{n} c_i x_i \tag{1}$$

$$\text{subject to:} \quad \sum_{i=1}^{n} m_{ij} x_i \geqslant 1, \quad \forall j = 1, \ldots, m \tag{2}$$

$$x_i \in \{0, 1\}, \quad \forall i = 1, \ldots, n \tag{3}$$

There are several alternatives that can be considered for the cost associated to the test cases. A classical option [10–15] is to consider all costs equal to one ($c_i=1$, $\forall i=1, \ldots, n$). In such case, the problem is equivalent to find a reduced test suite with a minimum number of test cases. However, the cost is often associated with a specific metric that is related to the cost of executing the test case, as the number of virtual code instructions that are run in a profiling tool [16] or the runtime measured in a particular platform [17]. This formulation is also known as the cost-aware TSMP.

Recently, the research community has also paid attention to the multi-objective TSMP. It has been proposed a bi-objective formulation [7,16] in which the conflicting objectives are the number of virtual code instructions executed in a profiling tool and the percentage of coverage of the test goals.

In spite of the existence of this multi-objective formulation, in our opinion the single-objective TSMP is still a relevant and important problem. For this reason, in this paper we adopt the classical single-objective formulation of the TSMP. In particular, we use a cost-aware formulation in which the cost of a test case corresponds to the wall-clock time of execution of the test case. The goal is minimizing the total runtime of the test suite ensuring that all the test goals of the piece of software under test are covered. This allows us to compare the SGS proposed in this work with classical specific heuristics previously proposed for this problem.

2.2. *Graphics processing units*

Since 2006, when Nvidia launched its first graphics card with an architecture with unified shaders, the rise in the use of GPUs as general purpose parallel platforms has never stopped growing. A key aspect for the widespread adoption of these devices is the emergence of general purpose programming languages, such as CUDA and OpenCL. These languages have enabled to unleash the power of GPUs for a wide range of users, including researchers. Other aspects that have contributed to the rise of general purpose computing on GPU are that GPUs are widely available, have a low economic cost, and have an inherent parallel architecture.

The GPU architecture follows a design philosophy that is radically different to CPU. While GPUs are designed with the idea of devoting most of the transistors to computation, in a CPU a large part of the transistors are dedicated to other tasks such as branch prediction, out-of-order execution, etc. In consequence, GPUs have a large number of small cores and are usually considered *many-core* platforms. The number of cores available in modern GPUs is growing steadily and will undoubtedly continue to do so in the foreseeable future. The number of threads that recent GPUs can run in parallel is in the order of thousands and is expected to continue growing rapidly, what makes these devices a powerful and low cost platform for implementing parallel algorithms.

CUDA is the general framework that enables to work with Nvidia's GPUs. The CUDA architecture abstracts GPUs as a set of shared memory multiprocessors (MPs) that follow the Single Instruction Multiple Threads (SIMT) parallel programming paradigm. SIMT is similar to Single Instruction Multiple Data but in addition to data-level parallelism (when threads are coherent) it allows thread-level parallelism (when threads are divergent, see [18]).

CUDA allows to define C/C++ functions, called *kernels*, that could be run in parallel by a large number of threads on the GPU. The group of all the threads generated by a kernel invocation is called a *grid*, which is partitioned into *blocks*. Each block groups threads that are run concurrently on a single MP. There is no fixed order of execution between blocks. The blocks are divided for their execution into *warps* that are the basic scheduling units and consist of 32 consecutive threads.

Threads can access data on multiple memory spaces during their life time [18]. All the threads running have access to the same global memory on the GPU that is one of the slowest memories. However, access to global GPU memory is usually more than one order of magnitude faster than data transfers between CPU and GPU. Registers are the fastest memory available on the card but are entirely managed by the compiler. They are only accessible by each thread independently. Shared memory is almost as fast as registers and can be accessed by any thread of a block; its lifetime is equal to the lifetime of the block. In the last years, the GPUs have incorporated two levels of cache for accessing to global memory, but both caches are really small.

In this work, we use a GeForce GTX 780 (Kepler architecture) that has a single precision floating point peak performance of 3977 GFlops.

3. Systolic computing based metaheuristics

The *Systolic Computing* [19,20] architecture emerged in the late 1970s. This architecture is composed by simple data processing units that are connected in a simple and regular fashion allowing a data flow between neighboring units. The units, which are also called cells, are capable of performing simple operations to data that is then passed through the next cell in the topology. This kind of architecture offers an understandable and manageable, but still quite powerful parallelism.

The source of inspiration of systolic computing architecture is the behaviour of the cardiovascular system. As part of the cardiac cycle, in the systole phase, the heart contracts, thus increasing the pressure inside the cavities. As a consequence, the heart ejects oxygenated blood into the arterial system to meet the metabolic needs of the tissues. Due to the systolic contraction, the blood is ejected from the heart with a regular cadence [21].

Systolic computing based metaheuristics adapt the concept of systolic computing to optimization. This family of algorithms are characterized by the flow of solutions through data processing units following a synchronous and structured plan. Each cell applies operators to the circulating tentative solutions in order to obtain new solutions that continue moving across the processing units. In this way, the circulating solutions are refined again and again by means of simple search operators. In particular, Systolic Genetic Search (SGS) [8,14,22,23] uses adapted evolutionary operators in the cells for refining the tentative solutions.

3.1. *Systolic Genetic Search Algorithm*

Several aspects have to be precisely defined to characterize a systolic computing based optimization algorithm as the interconnection topology of the systolic structure, the data flow of solutions, the size of the grid, and the computation of the cells.

SGS uses a bidimensional grid of cells in which solutions circulate synchronously through an horizontal and a vertical data flow. In this work, we use the SGS_B data flow (B stands for *both flows*), in which a solution moving through the vertical data flow that reaches the last row of the grid is passed on to the cell of the first row of the next column of the grid, while a solution moving through the horizontal data flow that reaches the last column of the grid is passed on to the cell of the first column of the next row of the grid. The interconnection topology and solution flows of SGS_B are shown in Fig. 1. Other data flows have been studied in [22,23]. From now on, we will refer to SGS_B as SGS since it is the only SGS algorithm that it is used in this work.

Although the idea is to have a relatively large number of cells to allow SGS to achieve a good exploration and to take advantage of the parallel computation capabilities offered by GPUs, the number of cells should not increase up to values that compromise

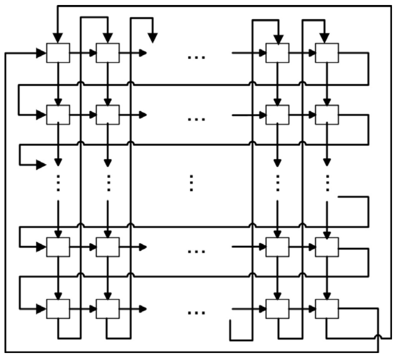


Fig. 1. Interconnection topology and solution flows of SGS_B.

performance. We consider that a proper balance is to have at least l cells (the length of the tentative solutions). The minimum grid with that number of cells is a $\lceil\sqrt{l}\rceil \times \lceil\sqrt{l}\rceil$ grid, but, in that case, every pair of solutions that coincide in a cell will coincide every $\lceil\sqrt{l}\rceil + 1$ steps. For this reason, in this work, we use a $\lceil\sqrt{l}\rceil \times \left(\lceil\sqrt{l}\rceil + 1\right)$ grid.

The computation performed by the cells is described next. Initially, each cell generates two random solutions which are aimed at moving horizontally and vertically, respectively. At each step of SGS, two solutions enter each cell, one from the horizontal data flow and one from the vertical data flow. Then, adapted genetic operators (crossover and mutation) are applied to generate two new solutions. SGS is commonly used with crossover operators that take two parent solutions and produce two children solutions. However, it is easy to adapt the cell computation for using crossover operators that only produce one child solution, e.g., applying the crossover operator twice. Later, the cell uses elitism to determine which solution continues moving through the grid for each flow, choosing between the incoming solution and the newly generated one. The use of elitism is critical, as there is no selection process like in standard genetic algorithms. Finally, each cell sends the outgoing solutions to the next cells of the data flows. Algorithm 1 presents the pseudocode of the SGS algorithm.

Algorithm 1. Systolic Genetic Search

```
1  foreach cell c do
2      $S_H$ =generateRandomSolution()
3      $S_V$ =generateRandomSolution()
4      sendSolutionThroughHorizontalFlow($S_H, c$)
5      sendSolutionThroughVerticalFlow($S_V, c$)
6  end
7  for $i$ = 1 to maxGeneration do
8      foreach cell c do
9          $S_H$ =receiveSolutionFromHorizontalFlow($c$)
10         $S_V$ =receiveSolutionFromVerticalFlow($c$)
11         $(new_H, new_V)$ =crossover($S_H, S_V$)
12         $new_H$ =mutation($new_H$)
13         $new_V$ =mutation($new_V$)
14         $new_H$ =elitism($S_H, new_H$)
15         $new_V$ =elitism($S_V, new_V$)
16         sendSolutionThroughHorizontalFlow($new_H, c$)
17         sendSolutionThroughVerticalFlow($new_V, c$)
18     end
19 end
```

3.2. A SGS for TSMP

SGS can be adapted to any solution representation and any particular operator. Since we are addressing a binary problem, we encode the solutions as binary strings. The length of the tentative solutions is equal to the number of test cases of the original test suite ($l = n$). The evolutionary search operators are the two-point crossover and the bit-flip mutation.

The two-point crossover selects two points from the binary strings of both parents. The two children solutions are calculated by swapping the binary string segments from the first crossover point to the second crossover point between the two parents. As the two-point crossover is applied on each cell, two different crossover point values are chosen randomly for each cell.

The bit-flip mutation operator flips a single bit in each solution of each cell. The mutation point for each cell is preprogrammed at fixed positions of the tentative solutions, which is defined by considering the location of the cell in the grid. In order to change different bits of the solutions through the grid, the formula for calculating the mutation point of the cell (i, j) is $i \times \lceil\sqrt{l}\rceil + j \bmod l$, where *mod* is the modulus of the integer division.

The fitness function used for the TSMP is described next. Since it is possible to build solutions that are not feasible, i.e., that do not cover all the test goals, the fitness function has to deal with this issue. Our approach applies a penalty term for each test goal that it is not satisfied. Eq. (4) shows the fitness function, where n is the number of test cases of the test suite, m is the number of test requirements that has to be covered with the test cases, c_i is the amount of resources required for the execution of the test case i (i.e., the wall-clock time of execution of the test case), x_i is the binary decision variables of the problem that indicate whether the test case i is included or not in the test suite, and r is the number of test goals covered by $(x_1, \ldots, x_n)$. We use a multiplicative penalty function $(1/(m - r + 1))$ that divides the original fitness value $(\sum_{i=1}^{n} c_i - \sum_{i=1}^{n} c_i x_i)$ by the number of test goals that are not covered plus one, reducing the fitness value with each uncovered test goal.

$$f(\vec{x}) = \frac{\sum_{i=1}^{n} c_i - \sum_{i=1}^{n} c_i x_i}{m - r + 1}. \tag{4}$$

3.3. CPU implementation of SGS

The CPU implementation of SGS is straightforward, so no further details are provided. However, the function for computing the fitness value in CPU has a peculiarity that deserves attention.

Algorithm 2 implements the fitness function on CPU. The algorithm iterates through a loop in Step 6 processing, in each iteration, a different test case i. If the test case is part of the solution (Step 7), another loop is used in Step 9 for iterating in the number of test goals m. This loop is used for calculating how many additional test goals are covered by the test case. As a consequence, the computation time of the fitness function evaluation is variable and highly sensitive to number of test cases that are part of the solution (i.e., the bit value is one) since for each of these tests it has to be computed which test goals are covered. For this reason, the computation time of the fitness function is long when a solution has many test cases, while it is short when the solution has fewer test cases.

Algorithm 2. Fitness function calculation on CPU

```
   input : Solution $s$, coverage matrix $M$, set of costs
           $c$, and cost $totalCost$ of the original suite
   output: Fitness value $f$ associated to solution $s$
1  for $j$ = 1 to $m$ do
2      $goal[j] = false$
3  end
4  $cost = 0$
5  $covered = 0$
6  for $i$ = 1 to $n$ do
7      if $s[i] == 1$ then
8          $cost = cost + c[i]$
9          for $j$ = 1 to $m$ do
10             if $M[i, j] == 1$ and not $goal[j]$ then
11                 $goal[j] = true$
12                 $covered = covered + 1$
13             end
14         end
15     end
16 end
17 $f = (totalCost - cost)/(m - covered + 1)$;
```

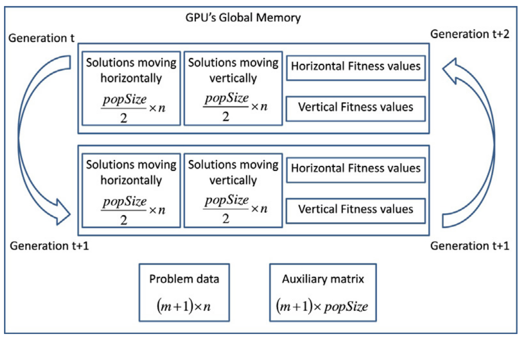


Fig. 2. Data organization on the GPU.

3.4. GPU implementation of SGS

A straightforward implementation of the fitness function would not be suitable for GPU since the presence and absence of test cases in a solution would produce thread divergence [18] within the warps and would not properly exploit the massive parallelism available on the GPU. For this reason, we followed an idea previously proposed in [7,16] that transforms the evaluation into a matrix–matrix multiplication operation. In this way, even though the multiplication requires a larger number of computations than the direct implementation, it follows a more structured pattern of computation with a better data access that it is well suited for GPU.

Algorithm 3 presents the pseudocode of the SGS algorithm for the host side. Initially, the seed for the random number generation and the constant data associated with the TSMP (i.e., the coverage matrix and the set of costs) are transferred from the CPU to the global memory of the GPU. Then, the population is initialized on the GPU (Step 3) and the fitness of the initial population is computed afterwards. The fitness evaluation involves two steps, the matrix–matrix multiplication (Step 4) that calculates which test goals are covered and the cost of the solution for all the population, and a reduction (Step 5) that calculates the number of test goals covered and the fitness value for all the population. At each iteration, the crossover and mutation of the solutions of each cell of the grid are executed (Step 7), and the systolic step is completed by computing the matrix-matrix multiplication, and completing the fitness calculation and applying the elitist replacement (Step 9). Finally, when the algorithm reaches the stop condition, the results are transferred from the GPU to the CPU.

Algorithm 3. SGS host side pseudocode

```
1  transfer seed for random numbers to GPU
2  transfer constant data to GPU's global memory
3  call initPopulation kernel
4  call matrix-matrix multiplication operation
5  call fitnessReduction kernel
6  for $i = 1$ to $maxGeneration$ do
7      call crossoverAndMutation kernel
8      call matrix-matrix multiplication operation
9      call fitnessReductionAndElitism kernel
10 end
11 transfer results from GPU to CPU
```

Fig. 2 shows how data is organized on the GPU. Two independent memory spaces of the GPU global memory are used to store the population and its associated fitness value. While the memory space that contains the population in generation t is read, the new solutions from generation $t+1$ can be written in the other memory space allowing concurrent access to the data (disjoint storage). Each memory space independently stores an array with the solutions moving horizontally, an array with the solutions moving vertically, an array with the fitness values corresponding to the solutions moving horizontally, and an array with the fitness values corresponding to the solutions moving vertically. Problem data is stored in a single $(m+1)\times n$ matrix, where in the $m\times n$ matrix is placed the coverage matrix and in the additional row is stored the cost of each test case. An auxiliary $(m+1)\times popSize$ matrix is used to store the matrix–matrix multiplication result.

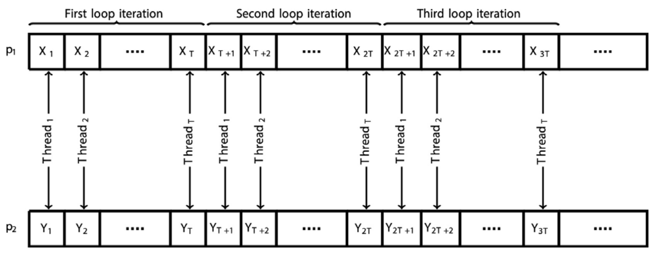

First loop iteration
Second loop iteration
Third loop iteration
p1
p2
X_1 X_2 .... X_T X_{T+1} X_{T+2} .... X_{2T} X_{2T+1} X_{2T+2} .... X_{3T} ....
Y_1 Y_2 .... Y_T Y_{T+1} Y_{T+2} .... Y_{2T} Y_{2T+1} Y_{2T+2} .... Y_{3T} ....
$Thread_1$ $Thread_2$ $Thread_T$ $Thread_1$ $Thread_2$ $Thread_T$ $Thread_1$ $Thread_2$ $Thread_T$

Fig. 3. Threads organization.

Now, we explain the kernel operation. `initPopulation` kernel initializes the population in the GPU using the CURAND Library to generate random numbers. The kernel is launched with a configuration that depends on the total number of bits that have to be initialized, following the guidelines recommended in [18]. The remaining kernels are implemented following the idea used in [8], in which operations are assigned to a whole block and all the threads of the block cooperate to perform a given operation, i.e., each block processes one cell of the grid. If the solution length is larger than the number of threads in the block, each thread processes more than one element but the elements used by a single thread are not contiguous. Thus, each operation is applied in chunks of the size of the thread block (T in the following figure), as it is shown in Fig. 3.

In the previous works from the literature [7,16] the matrix–matrix multiplication was programmed by hand by the authors. However, since there are already available libraries that compute linear algebra operations efficiently, we decided to use the matrix–matrix multiplication routine from CUBLAS library. The multiplication is invoked with the problem data matrix and the population matrix ($n\times popSize$ matrix composed by the array of the solutions moving horizontally and the array of the solutions moving vertically). The result is stored in an auxiliary $(m+1)\times popSize$ matrix that is then used by the kernels (`fitnessReduction` and `fitnessReductionAndElitism`) for completing the fitness calculation.

4. Heuristics for the cost-aware TSMP

Several specific heuristics for the TSMP with costs equal to one have been proposed in the literature, as a greedy algorithm [10], the HGS algorithm [11] and the GRE algorithm [12,15]. These algorithms, as well as their adaptations for the cost-aware TSMP, are described next.

4.1. Greedy algorithm

In the greedy algorithm, initially, all tests goals are marked as not covered and the reduced test suite is empty. While there is at least one test goal that is marked as not yet covered, the algorithm finds the test case that covers most uncovered test goals and that it is not yet included in the reduced test suite. This test case is then included in the test suite and all the test goals that are covered by the test case are now marked as covered. The algorithm ends when there are no uncovered test goals, returning the reduced test suite. The time complexity of the greedy algorithm is $O(mn\cdot min(m, n))$.

4.2. GRE algorithm

The GRE algorithm [12,15] involves the concepts of essential test case and 1-to-1 redundant test case. A test case is essential if a test goal is only covered by that test case. Essential test cases should be inserted as soon as possible in the reduced test suite for reducing redundancy. A test case is 1-to-1 redundant if there exists another test case that covers all the test goals of the former test case. 1-to-1 redundant test cases should be removed and not considered for the reduced test suite since there are other test cases that have a better coverage.

It should be noted that 1-to-1 redundant test cases should be removed one at a time because if it is done otherwise it is possible to left test goals uncovered. For this reason, this reduction has to be applied repeatedly until there are no 1-to-1 redundant test cases. Also, this reduction can turn a test case in essential since it could be left as the only test case able to cover a test goal. When an essential test case is inserted in the reduced test suite, it can turn a test case in 1-to-1 redundant. In consequence, the identification of essential and 1-to-1 redundant test cases are applied alternately.

The GRE algorithm is described next. Initially, all tests goals are marked as not covered and the reduced test suite is empty. Then, all the essential test cases are identified and included in the reduced test suite, marking as covered all the test goals that are covered by such test cases. The algorithm iterates processing the set of available test cases. In each iteration, first, 1-to-1 redundant test cases are identified and removed from the set. Then, if there is any essential test case, it is included in the reduced test suite and its associated test goals are marked as covered. Otherwise, a greedy heuristic is used for finding the test case with most uncovered test goals from the set of available test cases. The test case is included in the reduced test suite and its associated test goals are marked as covered. The loop ends when there are no test goals marked as uncovered. The time complexity of the GRE algorithm is $O(min(m, n) \cdot (m + n^2 k))$, where k is the maximum number of test goals that are covered by a single test case.

4.3. HGS algorithm

The HGS algorithm [11] is based on a completely different idea. Let R_i denote the set of all the test cases in the original test suite that cover the test goal r_i. Initially, the algorithm includes in the reduced test suite all the test case that belong to any R_i of cardinality one. The test cases included are also used for marking as covered all the R_i sets of whom they are a part. Then, the unmarked R_i sets with cardinality two are considered. The test case that belongs to the maximum number of such sets is chosen to be included in the reduced test suite, and all the unmarked R_i sets containing this test case are marked. This process is repeated until there are no unmarked R_i sets with cardinality two. In the same way, the process is repeated for the unmarked R_i sets with cardinalities 3, ..., *maxCard*, being *maxCard* the maximum cardinality of all the R_i sets. The time complexity of the HGS algorithm is $O(m \cdot (m + n) \cdot maxCard)$.

4.4. Cost-aware heuristics

Recently, Lin et al. [17] studied two different cost-aware test case metrics for being incorporated to the heuristics for test suite minimization. These metrics are the Ratio [24] (shown in Eq. (5), where $Coverage(t)$ is the number of uncovered test goals that are covered by t and $Cost(t)$ is the execution cost of the test case t), and the EIrreplaceability (shown in Eq. (6) and Eq. (7), where $Covers(r_s)$ is the number of test cases that cover r_s).

$$Ratio(t) = \frac{Coverage(t)}{Cost(t)} \tag{5}$$

$$EIrreplaceability(t) = \begin{cases} \infty, & t \text{ is essential} \\ \dfrac{\sum_{s=1}^{m} Contribution(t, r_s)}{Cost(t)}, & \text{otherwise} \end{cases} \tag{6}$$

$$Contribution(t, r_s) = \begin{cases} 0, & \text{if } t \text{ do not cover } r_s \\ \dfrac{1}{Covers(r_s)}, & \text{if } t \text{ covers } r_s \end{cases} \tag{7}$$

Lin et al. integrated this metrics to the greedy, GRE and HGS algorithm, obtaining six different algorithms. These algorithms, as well as the three original heuristics where extensively evaluated on real world programs. The results show that the four best performing algorithms, according to the percentage of suite cost reduction, are the greedy algorithm with the EIrreplaceability metric (from now on we will refer to it as GREEDY$_E$), the greedy algorithm with the Ratio metric (from now on we will refer to it as GREEDY$_R$), the HGS algorithm with the EIrreplaceability metric (from now on we will refer to it as HGS$_E$) and the GRE algorithm with the EIrreplaceability metric (from now on we will refer to it as GRE$_E$). Since these four specific heuristics have proved to be highly effective for reducing the cost and size of the test suite, we include them in our experimental analysis in order to provide a testbed for our proposal.

5. Experimental analysis

This section reports on the results of the experiments performed to evaluate the SGS proposed for the TSMP. First, we present the experimental setup and the research questions addressed in this work. Then, we detail the experimental results and discussion. Finally, we evaluate the proposed approach on larger instances as a case study.

5.1. Experimental setup

This subsection describes the subject programs and the methodology used for generating test suites, as well as the algorithms compared in the empirical studies.

5.1.1. Subject programs

The subject programs used in this work are real world programs that belong to the Siemens benchmark suite [25]. It is a well-known benchmark that has been often used to evaluate test suite reduction algorithms [9,13,16,17]. The Siemens suite is publicly available at the SIR website [26]. The suite includes an aircraft collision avoidance system (`tcas`), a statistic computation program (`totinfo`), two priority schedulers (`schedule` and `schedule2`), two lexical analyzers (`printtokens` and `printtokens2`) and a program that performs pattern matching and substitution (`replace`). Table 1 presents the Siemens programs including the number of test cases, the number of test goals and the number of source lines of code (LOCs). In this case, the test goals came from structural coverage. Test goals are already included in the source codes available in the repository.

For conducting the experiments, we measured the runtime of each test case of each subject programs. For this reason, the source

Table 1
TSMP subject programs used in the evaluation.

Subject	Test pool size	Test goals	LOCs
`tcas`	1608	54	162
`totinfo`	1052	117	346
`schedule`	2650	126	299
`schedule2`	2710	119	287
`printtokens`	4130	195	378
`printtokens2`	4115	192	366
`replace`	5542	208	514

Table 2
Execution cost of all the subject programs.

Subject program	Total execution cost in milliseconds	Minimum test case execution cost in microseconds	Mean test case execution cost in microseconds (mean ± std. dev.)	Maximum test case execution cost in microseconds
`tcas`	15.991	5.72	9.94 ± 0.58	11.47
`totinfo`	77.583	7.30	73.75 ± 42.36	359.66
`schedule`	125.487	5.45	47.35 ± 8.25	74.21
`schedule2`	145.086	0.06	53.54 ± 12.17	93.89
`printtokens`	152.267	5.49	36.87 ± 5.35	109.39
`printtokens2`	166.806	5.49	40.54 ± 6.34	117.42
`replace`	59.770	5.39	10.78 ± 4.21	98.63

code of the subject programs was modified to record their runtime. The source codes were compiled using the gcc 4.8.2 compiler with the `-O3` flag. The execution platform was a PC with a Quad Core Intel i7 4770 processor at 3.40 GHz. with 16 GB RAM using the CentOS Linux 7.0 operating system. The cost of each test case was computed as the average runtime of 1000 independent runs of the test case. Table 2 presents the cost of executing in milliseconds all the test cases of each subject program, as well as the minimum, the mean and the maximum cost of executing a test case for each subject program.

It should be noted that the set of test cases that takes more time (`printtokens2`) can be run in less than 200 ms. The reason is that the subject programs must accomplish simple tasks and the programs are relatively small. In spite of this, they are real-world scenarios that reflect actual interactions between test cases and test requirements from real programs, and they have been widely used in the literature. Therefore, in our opinion, it is justified to use these scenarios as a benchmark for evaluating a new proposal for the TSMP.

It is important to stress that the algorithms studied work independently of the units of the execution cost. Therefore, the conclusions drawn in this experimental evaluation can be extrapolated to scenarios with similar complexity but with a longer execution time (e.g., programs that involve tasks computationally more expensive, such as access to databases, interaction with remote servers or telecommunication networks). For instance, the execution of each test case takes about 10–100 min in a real-world test suite for regression testing from Cisco containing 2320 test cases [3]. The test pool of the Cisco suite is smaller than the test pool of some subject programs included in this work, but has a total execution time of the test suite of around 5 weeks.

5.1.2. *Methodology for generating the test suites*

In order to generate the test suites for each subject program, we followed the same methodology used by several authors [9,17,27] for conducting experimental evaluation over the Siemens suite. For each subject program, 1000 test suites were randomly generated following the next steps:

1. Randomly select a number c of test cases from the pool ($1 \le c \le 0.5 \times LOCs$).
2. If the test suite cannot satisfy all of the test requirements, randomly select a test case that at least satisfies an additional test requirement.
3. Repeat Step 2 until the test suite satisfies all of the test requirements.

Table 3 presents the range of sizes of the test suites generated, the average number of test cases of the 1000 test suites and the total execution cost in milliseconds of all the test suites for all subject programs. From now on, we will use the term *instance* interchangeably with the term *test suite*.

Table 3
Average size and total execution cost of all the test suites for all the subject programs.

Subject	Range of sizes of the suites	Average suite size	Total cost of all suites
`tcas`	3–81	41.40	411.609
`totinfo`	4–173	88.21	6515.621
`schedule`	3–149	76.85	3559.028
`schedule2`	3–143	71.98	3803.050
`printtokens`	8–189	96.54	3638.289
`printtokens2`	7–183	93.86	3851.476
`replace`	6–257	132.49	1429.501

5.1.3. *Algorithms and test environment*

In addition to the SGS algorithm proposed in this work and the four specific heuristics already proposed for the TSMP (GRE_E, HGS_E, GREEDY_R and GREEDY_E), we have also included two evolutionary algorithms, a simple genetic algorithm with and without elitism (EGA and SGA, respectively), in order to set an actual comparison basis. These evolutionary algorithms have been chosen because they share the same basic search operators (crossover and mutation) as SGS so we can properly evaluate the underlying search engine of the techniques. The details of the evolutionary algorithms used in the experimental evaluation are:

- Simple genetic algorithm (SGA): It is a generational genetic algorithm with binary tournament, two-point crossover and bit-flip mutation.
- Elitist genetic algorithm (EGA): It is similar to SGA but children solutions replace parent solutions only if they have a better fitness value.

The SGA and EGA parameter values used are 0.9 for the crossover probability and $1/l$ for the mutation probability, where l is the length of the tentative solutions (the number of test cases). The population size and the stopping criterion of both GAs were defined by considering the parameter settings of SGS. For this reason, the population size of both GAs is $2 \times \lceil\sqrt{l}\rceil \times \left(\lceil\sqrt{l}\rceil + 1\right)$, which is equal to the number of solutions of the grid of SGS. The stopping criterion used for both SGS and GAs is to reach a maximum number of generations fixed a priori. The number of generations is ten times the size of the grid and it was chosen to ensure that each solution circulates ten times over the grid in SGS.

The specific heuristics for the TSMP were only implemented on CPU, while the EAs (SGA, EGA and SGS) were implemented both on CPU and GPU.

The execution platform for the CPU implementations is the same used in Section 5.1.1. All CPU implementations were executed as single-threaded applications. The GPU implementations were run in an Nvidia's GeForce GTX 780 (2304 CUDA cores at 863 MHz., Kepler architecture) connected to the PC used for the CPU executions. CPU and GPU implementations were compiled using the `-O3` flag.

5.2. Research questions

This section presents the three research questions (RQs) studied in this paper. RQ1 concerns the numerical efficiency of the algorithm proposed in this work, SGS. Then, RQ2 deals with the performance of the GPU-based implementation of SGS. Finally, RQ3 is concerned with how the size of test suite and the number of test requirements affect the performance of SGS.

Research Question 1 (RQ1) – numerical efficiency of SGS: *Is SGS able to provide better solutions than the other six algorithms included in the study?*

To answer this question we have to study the quality of the solutions obtained by the algorithms. The quality of a solution is measured as the cost of executing the reduced test suite that is represented by the solution. For this reason, we analyze the execution cost of the reduced test suites produced for each instance of each subject program by all the algorithms. Since the scenarios are randomly generated and some of the algorithms involved in the evaluation are stochastic algorithms, statistical tests are used to assess the significance of the experimental results obtained.

The following statistical procedure has been used [28,29]. First, a single run for GRE_E, HGS_E, GREEDY_R and GREEDY_E and each instance of each subject program has been performed since they are deterministic algorithms. Then, fifty independent runs for each evolutionary algorithm and each test suite of each subject program have been performed. For the evolutionary algorithms, the execution cost of the reduced test suite for every instance is computed as the average of the execution cost over the fifty runs.

The Friedman's test has been used to rank the algorithms, for each subject program independently, attending to the execution cost of the reduced test suite obtained by the algorithm. This test is used to check if there are differences that are statistically significant among the algorithms for each subject program. Since more than two algorithms are involved in the study, a post-hoc testing phase which allows for a multiple comparison of samples has also been performed. The result is a multiple $1 \times N$ comparison with the control method, SGS in this case, using the Holm's post-hoc procedure for each subject program independently. All the statistical tests are performed with a confidence level of 99%.

Research Question 2 (RQ2) – parallel performance of SGS: *Is the computational performance of SGS on GPU better than the GPU-based implementations of SGA and EGA?*

In order to address this question, we analyze the performance of the CPU and GPU implementations of the three evolutionary algorithms using the wall-clock time of execution. We also use the Friedman's test for each subject program independently, attending to the average runtime of the algorithms for each instance. If the test finds that there are statistically significant differences between the algorithms, a $1 \times N$ comparison with SGS is performed (with Holm's post-hoc procedure) for each subject program independently. All the statistical tests are performed with a confidence level of 99%.

Additionally, we consider the improvement in performance of GPU over CPU implementations through the ratio between the wall-clock time of the CPU and the GPU executions of each algorithm. Even though some authors make reference to this metric as *speedup*, we prefer to refer to this ratio as runtime reduction. The use of the term speedup can give a misleading idea on how parallelizable is the GPU implementation of an algorithm since the execution times are measured in two completely different platforms.

Research Question 3 (RQ3) – scalability of the performance of SGS: *How does the size of the test suite and the number of test requirements impact in the performance of both implementations of SGS?*

To answer this question regarding the scalability of SGS, we analyze the incidence of both factors in the execution time of SGS. The number of test requirements it is determined by the subject program. As a consequence, there are only seven different values, while the sizes of the test suite have a much wider range of values. Since the size of the population and the number of iterations of SGS depends on the instance size, a direct analysis based on the runtime of the algorithm can be misleading. For this reason, we adopt a different metric for our analysis of the scalability of the performance, the number of genes processed by SGS per second. Although this measure does not entirely capture the complexity of the algorithm because there are some aspects that are not taken into account, like the generation of random numbers, it gives a general idea of the actual complexity of the computation performed by SGS.

5.3. Experimental results and analysis

In this subsection we present the numerical and performance results of our experiments, and, in the light of these results, we respond to the three research questions stated previously.

5.3.1. RQ1 – numerical efficiency of SGS

It is impossible to include the complete results obtained for the 7000 different instances (1000 instances for each subject program) and for each algorithm because of its huge extension. For this reason, we summarize the results obtained using two different metrics: the mean execution cost of the reduced test suites and the median of the execution cost of the reduced test suites. It should be noted that the total execution cost of the reduced test suites (i.e., the sum of the execution costs obtained by an algorithm for each of the instances of the subject program) can be easily calculated from the mean execution cost since the number of instances is 1000.

Table 4 presents the median of the execution cost in microseconds of the reduced test suites obtained by each of the algorithms, while Table 5 presents the mean execution cost in microseconds of the reduced test suites obtained by each of the algorithms. The numerical results of SGA, EGA and SGS included on both tables come from the GPU implementations. The numerical efficiency of the CPU implementations of these three algorithms was also studied, allowing us to verify that there are no statistically significant differences between the results on CPU and GPU of each algorithm.

The results obtained show that SGS is consistently the best performing algorithm for all the subject programs considering both the mean and the median execution cost. GREEDY_E and EGA also perform well, being in general the second and the third best performing algorithms. Even though GRE_E, HGS_E, GREEDY_R and SGA have a poor numerical performance compared to the other algorithms considered in the study, it should be highlighted that they are able to obtain reductions from the total execution cost of all the test suites of more than 93% for all the subject programs.

Table 6 presents the mean Friedman's ranking for each subject program independently. SGS is consistently the algorithm with the best ranking, while GREEDY_E and EGA are in general the second and the third best ranked algorithms (except for `tcas` where EGA is the second best and GREEDY_E is the third best). The Friedman's test shows that there are significant differences between the algorithms considered with a level of significance of 0.01. The Holm's post-hoc procedure for multiple comparison ($1 \times N$) with SGS as control method has obtained p-values, for each algorithm and each subject program, that are lower than 0.01, showing that the differences are statistically significant.

From these results, it is notorious that SGS outperforms the other two EAs that are based in the same basic search operators, making clear that the search engine of SGS is advantageous for solving the TSMP. Moreover, SGS does not uses any problem-specific knowledge and it is able to numerically outperform four different specific heuristics that were specially designed to tackle the TSMP.

Table 4
Median of the execution cost in microseconds of the reduced test suites.

Subject	GRE_E	HGS_E	$GREEDY_R$	$GREEDY_E$	SGA	EGA	SGS
`tcas`	25.830	25.865	25.790	25.710	29.800	25.650	**20.210**
`totinfo`	132.555	74.025	73.300	73.295	100.73	73.915	**72.155**
`schedule`	110.400	107.520	103.735	102.880	117.935	95.940	**84.150**
`schedule2`	96.310	88.140	83.270	80.980	101.520	78.315	**72.600**
`printtokens`	171.590	175.095	200.755	168.160	198.615	170.360	**150.420**
`printtokens2`	159.095	162.460	188.725	154.880	188.725	161.755	**150.520**
`replace`	39.230	39.185	36.260	35.245	40.110	34.715	**30.365**

The best results are in bold.

Table 5
Mean execution cost in microseconds of the reduced test suites.

Subject	GRE_E	HGS_E	$GREEDY_R$	$GREEDY_E$	SGA	EGA	SGS
`tcas`	25.824	26.455	26.666	25.508	28.158	25.635	**23.450**
`totinfo`	152.382	112.804	105.433	102.085	122.897	100.197	**93.547**
`schedule`	109.301	105.296	102.114	99.367	118.735	97.139	**90.059**
`schedule2`	104.081	91.739	90.336	87.792	101.713	85.026	**78.676**
`printtokens`	168.971	175.971	196.303	164.628	195.011	170.935	**155.902**
`printtokens2`	164.402	170.924	191.625	160.071	189.151	164.047	**151.897**
`replace`	41.008	41.841	39.035	36.763	41.941	37.382	**33.387**

The best results are in bold.

Table 6
Mean Friedman's ranking ($\alpha = 0.01$).

Subject	GRE_E	HGS_E	$GREEDY_R$	$GREEDY_E$	SGA	EGA	SGS
`tcas`	5.2190 ▲	4.5675 ▲	4.4380 ▲	3.9005 ▲	4.8610 ▲	3.1765 ▲	1.8375
`totinfo`	6.1175 ▲	4.0685 ▲	3.6895 ▲	3.3795 ▲	5.7245 ▲	3.4480 ▲	1.5725
`schedule`	5.3320 ▲	4.1375 ▲	3.7590 ▲	3.4290 ▲	6.1470 ▲	3.5645 ▲	1.6310
`schedule2`	5.6485 ▲	4.0275 ▲	3.8710 ▲	3.5425 ▲	5.6260 ▲	3.5465 ▲	1.7380
`printtokens`	3.9295 ▲	4.3215 ▲	5.8750 ▲	2.6900 ▲	6.0515 ▲	3.8220 ▲	1.3105
`printtokens2`	4.1190 ▲	4.2665 ▲	5.8175 ▲	2.8125 ▲	6.0375 ▲	3.4535 ▲	1.4935
`replace`	5.0765 ▲	5.0755 ▲	4.2815 ▲	3.1490 ▲	5.6055 ▲	3.5295 ▲	1.2825

'▲' states that the ranking of SGS has statistically lower value than the algorithm of the column for the subject program of the corresponding row.

Table 7
Runtime in seconds of the CPU versions.

Subject	SGA			EGA			SGS		
	min	median	max	min	median	max	min	median	max
`tcas`	**0.0001**	0.0687	0.4960	**0.0001**	0.0312	0.1478	**0.0001**	**0.0211**	**0.1010**
`totinfo`	**0.0002**	1.5663	9.9203	**0.0002**	0.3075	1.6525	**0.0002**	**0.2586**	**1.4201**
`schedule`	**0.0002**	0.7601	5.2867	**0.0002**	0.1794	1.0544	**0.0002**	**0.1583**	**0.8525**
`schedule2`	**0.0002**	0.6121	3.3032	**0.0002**	0.1615	0.7470	**0.0002**	**0.1287**	**0.5334**
`printtokens`	0.0021	3.5521	19.6361	0.0022	**0.5417**	**2.6400**	**0.0020**	0.7925	3.7750
`printtokens2`	0.0019	3.3179	18.3198	0.0020	**0.5058**	**2.4602**	**0.0018**	0.6691	3.3028
`replace`	**0.0018**	10.1598	56.0602	0.0021	**1.3678**	**7.0298**	0.0020	2.1527	10.1788

The best results are in bold.

5.3.2. *RQ2 – parallel performance of SGS*

For each instance of each subject program, the average runtime over fifty independent runs for the CPU and GPU implementations of the evolutionary algorithm is computed. To summarize the performance results of each evolutionary algorithm over the whole set of instances of each subject program, we consider the minimum (i.e., the minimal average runtime of an instance), the median (i.e., the average runtime of an instance that is larger than the average runtime of the 50% of the instances) and the maximum (i.e., the maximal average runtime of an instance) runtime. The most important indicators to analyze are the median and maximum.

Table 7 shows the minimum, the median and the maximum of the average runtime of the CPU implementations for each subject program. In the four smallest subjects, according to the number of test requirements, SGS is the algorithm with the shortest runtime, while in the three biggest subjects, EGA is the best performing algorithm. SGA is consistently the worst performing algorithm for all the subject programs. The fundamental reason for such results is that the fitness function evaluation has a variable runtime. For each test case included in a solution, it has to be computed which test requirements are covered. As a consequence, the fitness value computation is strongly dependent on the number of test cases of the solution, i.e., a higher number of test cases of the solution implies a longer runtime. Since SGA is the numerically worst performing algorithm, it constructs and evaluates many solutions with a large number of test cases, which is penalized in its runtime.

Table 8 shows the minimum, the median and the maximum of the average runtime of the GPU implementations for each subject program. While differences in the runtimes are not relatively large, SGS is the best performing algorithm, while SGA is in general the second best. In this case, the incidence of the fitness function evaluation in the runtime is equivalent for all the algorithms since the fitness values are computed using the same matrix–matrix multiplication operation, whose runtime mostly depends on the sizes of the involved matrices. For this reason, the differences in performance are strongly related to the underlying search mechanism of

Table 8
Runtime in seconds of the GPU versions.

Subject	SGA			EGA			SGS		
	min	median	max	min	median	max	min	median	max
`tcas`	0.0622	0.0807	0.0997	0.0623	0.0815	0.1017	**0.0617**	**0.0783**	**0.0954**
`totinfo`	0.0621	0.1065	0.2180	0.0623	0.1082	0.2257	**0.0618**	**0.1025**	**0.1952**
`schedule`	0.0622	0.0959	0.1646	0.0621	0.0971	0.1685	**0.0619**	**0.0928**	**0.1502**
`schedule2`	0.0622	0.0958	0.1437	0.0622	0.0970	0.1473	**0.0618**	**0.0926**	**0.1327**
`printtokens`	0.0638	0.1126	0.2423	0.0640	0.1142	0.2503	**0.0637**	**0.1081**	**0.2279**
`printtokens2`	0.0638	0.1125	0.2457	0.0639	0.1143	0.2537	**0.0634**	**0.1078**	**0.2254**
`replace`	0.0639	0.1525	0.4702	0.0642	0.1568	0.4889	**0.0636**	**0.1435**	**0.4208**

The best results are in bold.

Table 9
Mean Friedman's ranking for CPU and GPU runtimes ($\alpha = 0.01$).

Subject	CPU			GPU		
	SGA	EGA	SGS	SGA	EGA	SGS
`tcas`	2.9435 ▲	2.0295 ▲	1.0270	2.0750 ▲	2.9250 ▲	1.0000
`totinfo`	2.9550 ▲	2.0255 ▲	1.0195	2.0165 ▲	2.9835 ▲	1.0000
`schedule`	2.9640 ▲	2.0175 ▲	1.0185	2.0110 ▲	2.9865 ▲	1.0025
`schedule2`	2.9560 ▲	2.0245 ▲	1.0195	2.0145 ▲	2.9835 ▲	1.0020
`printtokens`	2.9845 ▲	1.0700 ▽	1.9455	1.9900 ▲	2.9980 ▲	1.0120
`printtokens2`	2.9775 ▲	1.0900 ▽	1.9325	1.9955 ▲	2.9970 ▲	1.0080
`replace`	2.9630 ▲	1.0615 ▽	1.9755	1.9955 ▲	3.0000 ▲	1.0045

'▲' states that the ranking of SGS runtime has statistically lower value than the algorithm of the column for the subject program of the corresponding row.
'▽' states that the ranking of SGS runtime has statistically higher value than the algorithm of the column for the subject program of the corresponding row.

Table 10
Runtime reduction of GPU versions vs. CPU versions.

Subject	SGA			EGA			SGS		
	min	median	max	min	median	max	min	median	max
`tcas`	–	–	**5.06**	–	–	1.46	–	–	1.09
`totinfo`	–	**14.66**	**45.57**	–	2.84	7.33	–	2.52	7.33
`schedule`	–	**7.93**	**32.35**	–	1.84	6.34	–	1.71	5.71
`schedule2`	–	**6.35**	**23.04**	–	1.66	5.09	–	1.39	4.02
`printtokens`	–	**31.93**	**81.07**	–	4.78	10.55	–	7.43	16.93
`printtokens2`	–	**29.93**	**74.67**	–	4.47	9.89	–	6.21	14.94
`replace`	–	**66.90**	**130.00**	–	8.77	16.21	–	15.17	27.52

The best results are in bold.
'–' states that there is no reduction in the runtime, i.e., CPU-based implementation executes faster than GPU-based implementation.

each algorithm. It should be noted that the differences in the performance of the parallel GPU implementations of the algorithms grow with the size of the instances (the differences are larger for maximum and the median than for the minimal). This shows that SGS exhibits a good scalability behavior when solving highly dimensional problem instances.

Table 9 presents the mean Friedman's ranking for each subject program independently. The Friedman's test shows that there are significant differences between the algorithms considered with a level of significance of 0.01. The Holm's post-hoc procedure for multiple comparison ($1 \times N$) with SGS as control method has obtained p-values, for each algorithm and each subject program, that are lower than 0.01, showing that the differences are statistically significant.

Table 10 presents the improvement in performance of GPU over CPU implementations measured as the runtime reduction of GPU versions vs. CPU versions. It can be noticed that SGA is the algorithm that presents the best reductions, and is able to obtain improvements of up to 66× for the median and up to 130× for the maximum in the largest instances (`replace` is the subject program with more test requirements and the test sets with more test cases were generated for `replace`). SGS has a also a good performance, specially for larger instances, obtaining reductions of up to 15× and 27× for the median and the maximum, respectively. Finally, even though EGA presents acceptable reductions, these improvements are the smallest for the larger instances.

In our opinion, in this particular case, and because of the impact of the variable runtime of the fitness evaluation in the total runtime of the algorithms, the use of the runtime reduction as a metric of the comparative performance can be misleading. SGA is the algorithm that is able to obtain the best runtime reductions, but as it has already been shown in Tables 7 and 8, the reduction is not produced by a particular additional improvement in the GPU implementation of SGA. It is just because the CPU-based implementation of SGA has a rather poor performance.

To make this point clear, we graphically analyze the runtime of the algorithms. Fig. 4 plots the average runtime (in logarithmic scale) of each algorithm for all the test suites generated for `replace`. The tendency is pretty clear, the difference between the runtime of SGA implemented on CPU and the other CPU-based implementation increases with the size of the test suite. It can also be appreciated that, even though SGS has the shortest execution time for the GPU implementations, all the GPU-based implementations of the three algorithms have runtimes of the same order.

A detail that deserves further attention is why the graph of the runtime has steps, i.e., the runtime is almost constant for some intervals of the values of the size of the test suite and there are jumps in the value of the runtime between test suites whose size belong to two adjacent intervals. The existence of such jumps for SGS is related with the fact that the size of the population and the number of iterations of the algorithms depends directly on the size of the test suite that is being solved. In particular, the size of the

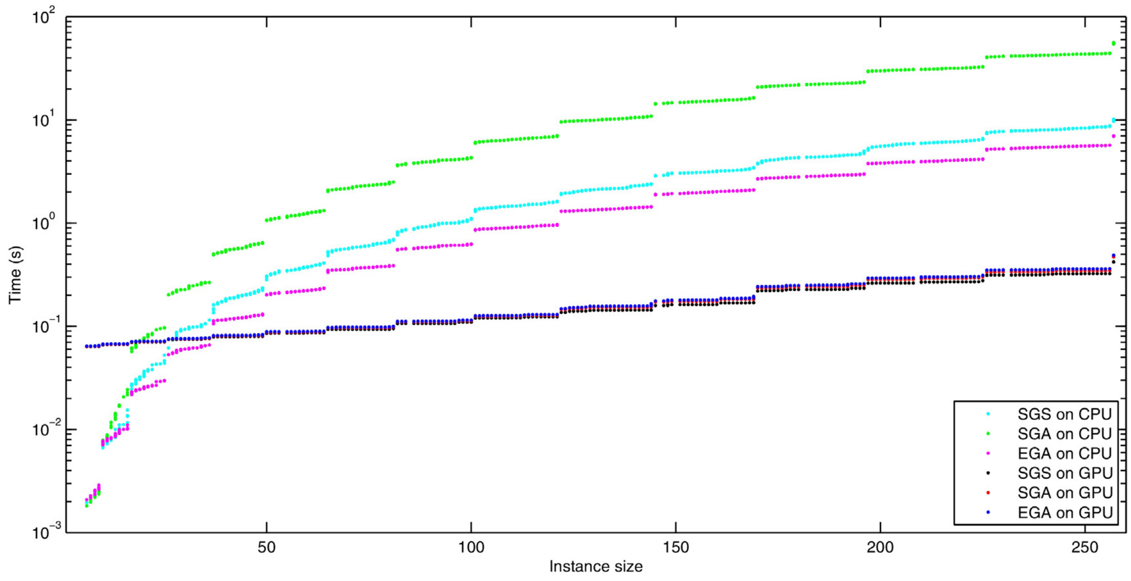


Fig. 4. Average runtime of the algorithms for all the instances of `replace`.

grid is $\lceil\sqrt{l}\rceil \times \left(\lceil\sqrt{l}\rceil + 1\right)$, where l is the size of the test suite that it is being solved. As a consequence, the grid size, as well as the size of the population and the number of iterations, used for several different test suite sizes is exactly the same (the only difference is in the size of the tentative solutions). For instance, the grid size is 10×11 for instances with size between 82 and 100 and it is 11×12 for instances whose size is between 101 and 121. The jump in the execution time is caused by the change in the size of the grid, e.g., there is a jump in runtime between instances with size 100 and 101 (see Fig. 4). Since SGA and EGA are configured with the same population size and number of iterations than SGS, this effect is also present in these algorithms.

From the analysis performed, it can be concluded that SGS is the best performing algorithm for the implementations on GPU and that the differences in runtime with the other GPU-based implementations are caused by the underlying search engine of SGS. The runtime reduction of the GPU-based implementation over the CPU-based implementation of SGS is up to 27 and the reduction is larger for the larger instances.

We do not explicitly make a comparison between the runtime of the evolutionary algorithms and the specific heuristics. Both type of algorithms have really different natures, what makes a direct comparison unfair. However, the runtime of the specific heuristics is one or two orders of magnitude smaller than the runtime of the evolutionary algorithms, so it can be considered negligible. In order to be more advantageous to use an evolutionary algorithm (in particular SGS) that the best performing specific heuristic (GREEDY$_E$), it must be satisfied that $t_{SGS} < t^s_H - t^s_{SGS}$, where t_{SGS} is the execution time of the SGS algorithm, t^s_{SGS} is the execution time of the reduced test suite obtained by SGS, and t^s_H is the execution time of the reduced test suite obtained by the specific heuristic. For instance, for test suite with similar features than the largest instance from `replace`, the difference between the reduced test suite obtained by GREEDY$_E$ and SGS have to be at least 0.4208 s, so that is advantageous to use SGS instead of the heuristic.

5.3.3. RQ3 – scalability of the performance of SGS

As it was highlighted in Section 5.3.2, the graph of the runtime of SGS as a function of the number of test cases of the suite has several discontinuity jumps. Since we are only interested in analyzing the general trend of the performance of SGS, we adopt the following criterion for dealing with the discontinuity jumps. Our analysis considers only the instance sizes that are in the middle of a step of the function.

A direct analysis based on how the runtime of the algorithm varies with the instance size would not properly reflect the scalability of the performance of SGS because the instance size affects the size of the population and the number of iterations of SGS. In consequence, we use the number of genes that the algorithm can process in 1 s of execution for our analysis. The number of genes processed by SGS per second for an instance can be calculated as the total number of genes processed by the algorithm ($population \times generations \times instancesize$) divided by the runtime of SGS.

Table 11 presents the number of genes processed in genes per second by the CPU-based SGS for each subject program. The subject programs are ordered in the table by the number of test requirements.

We begin our analysis with the incidence of the number of requirements in the performance of the CPU-based SGS. The tendency is clear, the larger the number of requirements, the lower the number of genes processed per second by SGS. This is justified by the fact that when an instance with a larger number of requirements is being solved, a greater amount of work has to be done for each gene that is processed during the fitness evaluation process. In fact, the increase in the amount of work is produced by the number of ones in the solutions and not just by the number of genes. This explains why there is an exception in the tendency, `tot-info`, which has 117 test requirements, shows a worse performance than `schedule2` (119 test requirements) in 10 out of 10 instances and than `schedule` (126 test requirements) in 8 out 10 instances. The highest performance degradation, regarding the number of test requirements, is produced in instances with size 57, SGS is able to process 9.06× genes per seconds for `tcas` than for `replace`.

Now we analyze the influence of the instance size in the performance of the CPU-based SGS. There is also a clear tendency, the larger the instance size, the larger the number of genes processed per second by SGS. The reason is twofold. On the one hand, the incidence of the rest of the algorithm (the fraction of the algorithm that does not deal with gene processing) in the total runtime of the algorithm is mitigated when considering larger instances. On the other hand, when SGS is solving larger instances, and therefore using a larger population, SGS makes a better use of the memory hierarchy due to data locality. This suggests that the CPU implementation is

Table 11
Number of genes processed by SGS on CPU (in genes per second).

Instance	Subject program						
Size	`tcas`	`totinfo`	`schedule2`	`schedule`	`printtokens2`	`printtokens`	`replace`
7	3.352e7	1.745e7	1.784e7	1.693e7	9.692e6	–	9.377e6
13	5.658e7	2.664e7	2.987e7	2.777e7	1.575e7	1.440e7	1.174e7
21	8.131e7	3.860e7	4.288e7	3.786e7	1.654e7	1.370e7	9.856e6
31	9.983e7	4.587e7	5.341e7	4.682e7	1.953e7	1.746e7	1.171e7
43	1.268e8	5.456e7	6.696e7	5.752e7	2.212e7	1.984e7	1.441e7
57	1.481e8	6.439e7	7.996e7	6.819e7	2.562e7	2.271e7	1.635e7
73	1.699e8	7.311e7	9.091e7	7.895e7	2.969e7	2.668e7	1.908e7
91	–	8.004e7	1.034e8	9.020e7	3.471e7	2.937e7	2.280e7
111	–	9.258e7	1.204e8	1.020e8	3.988e7	3.394e7	2.753e7
133	–	1.025e8	1.344e8	1.114e8	4.485e7	4.098e7	3.087e7
157	–	1.097e8	–	–	4.918e7	4.254e7	3.378e7
183	–	–	–	–	5.303e7	4.905e7	3.607e7
211	–	–	–	–	–	–	4.070e7
241	–	–	–	–	–	–	4.458e7

'–' state size for which no instance is available

Table 12
Number of genes processed by SGS on GPU (in genes per second).

Instance	Subject program						
Size	`tcas`	`totinfo`	`schedule2`	`schedule`	`printtokens2`	`printtokens`	`replace`
7	3.167e5	3.165e5	3.164e5	3.162e5	3.155e5	–	3.159e5
13	1.573e6	1.565e6	1.566e6	1.571e6	1.565e6	1.562e6	1.560e6
21	5.408e6	5.411e6	5.402e6	5.374e6	5.371e6	5.367e6	5.390e6
31	1.485e7	1.489e7	1.484e7	1.486e7	1.470e7	1.473e7	**1.465e7**
43	3.441e7	3.437e7	3.426e7	3.424e7	**3.403e7**	**3.397e7**	**3.390e7**
57	6.984e7	**6.965e7**	6.969e7	**6.957e7**	**6.885e7**	**6.867e7**	**6.843e7**
73	1.274e8	**1.277e8**	**1.275e8**	**1.276e8**	**1.264e8**	**1.266e8**	**1.263e8**
91	–	**2.141e8**	**2.147e8**	**2.143e8**	**2.058e8**	**2.068e8**	**2.063e8**
111	–	**3.379e8**	**3.375e8**	**3.371e8**	**3.194e8**	**3.211e8**	**3.205e8**
133	–	**4.905e8**	**4.905e8**	**4.890e8**	**4.544e8**	**4.523e8**	**4.506e8**
157	–	**6.959e8**	–	–	**6.456e8**	**6.428e8**	**6.407e8**
183	–	–	–	–	**7.165e8**	**7.096e8**	**7.082e8**
211	–	–	–	–	–	–	**9.023e8**
241	–	–	–	–	–	–	**1.099e9**

'–' state size for which no instance is available.
bold indicates that the performance of the GPU-based implementation is better than CPU-based implementation.

memory-bounded and it is not compute-bounded, i.e., the limiting factor of the performance is the memory access speed instead of the processor utilization. There are also some minor fluctuations in the tendency for the smaller instances of `printtokens2` and `replace` that are also caused by a relative greater number of genes with value one.

Table 12 presents the number of genes processed in genes per second by the implementation on GPU of SGS for each subject program. The subject programs are ordered in the table by the number of test requirements.

We continue the analysis with the incidence of the number of requirements in the performance of the GPU-based SGS. In this case, while there are some small fluctuations, it can also be seen that there is the same tendency than in the CPU-based implementations, the larger the number of requirements, the lower the number of genes processed per second by SGS. This is caused because the number of requirements is one of the dimensions of one of the matrices involved in the matrix–matrix multiplication operation.

From these results, it is clear that the impact of this factor in the degradation of the performance is considerably lower than for CPU, e.g., there is almost a 90% of performance reduction in the CPU-based implementation from 54 (`tcas`) to 208 (`replace`) test requirements when the instance size is 73, while in the GPU-based implementation is merely 1%.

Finally, we analyze the influence of the instance size in the performance of the GPU-based SGS. It can also be appreciated that there is the same tendency than in the CPU-based implementations, a larger number of genes processed per second by SGS is obtained when solving the instances with larger size. The reason is threefold. On the one hand, larger instances allow SGS to better profit from the parallelism of the GPUs. In particular, larger instances imply larger tentative solutions that are able to make the most out of the parallel computation of the threads. Larger instances also cause that SGS uses a larger population, generating a higher number of blocks and thus taking advantage of the computing capabilities offered by the GPU architecture. Also, the two reasons, that produce this tendency in the CPU implementation, help to explain this tendency in the GPU implementation, i.e., the reduction of the impact of the rest of the algorithm in the runtime of SGS and a better use of data locality.

It is clear that the impact of the instance size in the improvement of the performance is significantly higher for the GPU implementation than for the CPU-based implementation, e.g., the improvement in performance for `replace` between instance size 7 and 241 is almost $5\times$ in the CPU-based implementation, while in the GPU implementation is more than $3478\times$. It can also be seen that the instance size from which the implementation in GPU has better performance than the CPU implementation (indicated in bold in Table 12) is smaller when considering a larger number of test requirements.

From these results, it can be concluded that while both SGS implementations show a good scalability when considering larger instances, GPU implementation scalability is far superior than CPU-based implementation scalability. The results also allow to conclude that the performance of the CPU implementation of SGS is highly degraded when considering a larger number of test

Table 13
Execution cost in microseconds of the reduced test suites.

Instance size	GRE_E	HGS_E	$GREEDY_R$	$GREEDY_E$	SGA	EGA	SGS
255	1867.03	1920.45	1887.69	1714.57	1793.47 ± 72.16	1678.65 ± 52.08	**1605.59 ± 29.42**
501	3714.00	3931.92	3879.62	3671.89	3777.51 ± 59.90	3641.24 ± 51.78	**3550.33 ± 36.20**
1000	1379.63	1861.29	1811.85	1335.96	1574.41 ± 83.02	1381.47 ± 50.93	**1293.67 ± 28.44**
1502	1727.16	2074.31	2092.51	1775.42	1917.80 ± 70.81	1739.18 ± 54.40	**1694.19 ± 27.35**
2000	1487.15	1790.04	1795.33	1416.71	1588.55 ± 70.80	1394.91 ± 44.51	**1335.75 ± 30.96**
2505	1533.45	1563.24	1578.14	1433.35	1559.05 ± 62.24	1393.74 ± 29.79	**1363.25 ± 29.30**
3003	1363.69	1647.08	1528.54	1312.06	1541.00 ± 81.09	1314.32 ± 41.39	**1247.50 ± 24.01**

The best results are in bold.

Table 14
Mean Friedman's ranking ($\alpha = 0.01$).

Subject	GRE_E	HGS_E	$GREEDY_R$	$GREEDY_E$	SGA	EGA	SGS
`space`	3.7143 ▲	6.5714 ▲	6.2857 ▲	2.8571 –	5.0000 ▲	2.5714 –	1.0000

'▲' States that the ranking of SGS has statistically lower value than the algorithm of the column.
'–' States that the ranking of SGS has lower value than the algorithm of the column but it is not statistically significant.

requirements. Even though there is a degradation in the performance of the GPU implementation of SGS when considering a larger number of test requirements, this degradation is minimal.

5.4. *Case study: space*

In this subsection, we extend our experimental evaluation of SGS including larger scenarios than in the previous subsections. For this purpose, we use the subject program `space` that is also publicly available at SIR website [26]. `space` [30] is an interpreter for an Array Definition Language that consist of 6200 LOCs in C (more than 12× larger than the subject program with more LOCs from the Siemens benchmark). It has a test pool of 13,585 test cases (almost 2.5× larger than the subject program with the largest test pool from the Siemens benchmark) and 2728 test goals (more than 13× larger than the subject program with more test goals from the Siemens benchmark).

We also measured the runtime of each test case of `space` following the same procedure described in subsection 5.1.1. The resulting total cost of executing all the test cases of the test suite is 2403.989 ms, while the the minimum, the mean (± std. dev.) and the maximum cost of executing a test case for each subject program is 8.72, 176.96 (± 248.526) and 6256.04 microseconds, respectively.

We also followed the methodology described in Section 5.1.2 for randomly generating 1000 test suites. The size of the randomly generated test suites ranges from 33 to 3099 test cases, while the average number of test cases of the 1000 test suites is 1581.72. The total execution cost of all the test suites is 279916.135 ms.

As a consequence of the resulting instances size, the runtime is high for the CPU implementations of the evolutionary algorithms. For this reason, we have selected seven instances from the 1000 available for the experiments. Six instances were chosen considering sizes evenly distributed within the range of available sizes and another instance was included because its number of test cases is the most similar to the largest test suite generated for the Siemens benchmark. The number of test cases of the instances selected for the experimental evaluation is 255, 501, 1000, 1502, 2000, 2505, and 3003. We structure the analysis of the experimental results of the case study in the same three research questions responded for the Siemens benchmark. We follow exactly the same methodology, but for the number of runs of the evolutionary algorithms that is thirty in this case (instead of fifty).

5.4.1. *RQ1 – numerical efficiency of SGS*

Table 13 presents the execution cost in microseconds of the reduced test suites obtained by each of the algorithms for each of the instances. For the evolutionary algorithms, the execution cost of the reduced test suite is computed as the average of the execution cost over the thirty runs of the GPU implementation (mean ± std. dev.). Table 14 presents the mean Friedman's ranking for the seven instances.

The results obtained for the large instances of the case study are similar to the results obtained for the small instances of the Siemens benchmark. SGS is consistently the best performing algorithm for all the instances considered, and clearly outperforms the other two EAs and the four specific heuristics. Although it is not possible to demonstrate that the differences between the numerical performance of SGS and $GREEDY_E$ (p-value is 0.0539), and SGS and EGA (p-value is 0.1735) are statistically significant with a confidence level of 99%, this is caused by the small number of instances considered.

5.4.2. *RQ2 – parallel performance of SGS*

Table 15 presents the average runtime over the thirty independent runs for the CPU and GPU implementations of the evolutionary algorithms, as well as the improvement in performance of GPU implementation over CPU implementation measured as the runtime reduction of GPU versions vs. CPU versions.

The results obtained confirm the tendencies appreciated for the smaller instances. EGA is the best performing algorithm among the CPU versions, SGS is the best performing algorithm among the GPU versions, and SGA is the algorithm with the best runtime reductions. It it clear from the results that the runtime reduction of SGA is explained by the rather poor performance of the CPU version of SGA.

The overall best performing algorithm is the GPU implementation of SGS. Moreover, in this case, the differences in the runtime of the GPU implementation between SGS and the other two evolutionary algorithms is more than a 10% for the larger instances. Although the runtime reduction of SGS decreases as the size of the instance is larger, SGS is still able to obtain reductions of up to 48× for instances with 3003 test cases and and 2728 test goals.

5.4.3. *RQ3 – scalability of the performance of SGS*

Table 16 presents the number of genes processed in genes per second by the CPU and GPU versions of SGS.

The results corroborate the tendency in the influence of the instance size in the performance of the CPU-based SGS, the larger the instance size, the larger the number of genes processed per second by SGS. On the other hand, while the same tendency is appreciated for the influence of the instance size in the performance of the GPU implementation of SGS in the five smaller

Table 15
Runtime in seconds of the CPU and GPU versions and runtime reduction of GPU versions vs. CPU versions.

Instance	SGA			EGA			SGS		
Size	CPU	GPU	Red.	CPU	GPU	Red.	CPU	GPU	Red.
255	749.58 ± 5.84	1.47 ± 0.01	**509.92**	**214.53 ± 4.93**	1.49 ± 0.01	144.17	253.71 ± 6.81	**1.39 ± 0.01**	182.53
501	3731.14 ± 18.41	8.55 ± 0.10	**436.39**	**919.24 ± 25.71**	8.62 ± 0.12	106.70	1105.21 ± 24.15	**8.23 ± 0.11**	134.31
1000	15954.49 ± 214.88	54.16 ± 0.02	**294.57**	**3646.58 ± 128.57**	54.88 ± 0.02	66.45	4816.82 ± 53.72	**52.62 ± 0.02**	91.53
1502	36070.20 ± 1565.16	176.70 ± 0.07	**204.13**	**8494.88 ± 275.76**	178.77 ± 0.04	47.52	11719.31 ± 39.80	**168.12 ± 0.03**	69.71
2000	66436.25 ± 217.04	404.53 ± 0.07	**164.23**	**15950.64 ± 422.25**	409.41 ± 0.18	38.96	23390.66 ± 201.40	**375.19 ± 0.16**	62.34
2505	N/A	865.41 ± 0.32	**> 99.84**	**27319.37 ± 610.95**	876.87 ± 0.09	31.16	41968.11 ± 530.05	**781.67 ± 0.24**	53.69
3003	N/A	1387.51 ± 2.48	**> 62.27**	**38406.65 ± 574.53**	1405.61 ± 2.49	27.32	60560.02 ± 520.11	**1254.89 ± 0.14**	48.26

The best results are in bold.
'N/A' states that the runs have timed out (runtime is more than >86400 s).

Table 16
Number of genes processed by SGS (in genes per second).

Instance size	CPU version	GPU version
255	1.487e6	2.715e8
501	2.762e6	3.710e8
1000	4.630e6	4.238e8
1502	6.238e6	4.348e8
2000	7.328e6	4.568e8
2505	8.396e6	4.508e8
3003	9.408e6	4.540e8

instances studied for the case study, this trend is reversed for the remaining two instances. This result indicates that the GPU has already reached the peak of performance of the algorithm for the computing capacity of the device used in the experiments. But even so, the GPU implementation is still able to process almost two orders of magnitude more genes per second than the CPU implementation, while solving really large scenarios.

The results also corroborate the tendencies in the incidence of the number of requirements in the performance of the CPU and GPU implementations of SGS. This can be borne out by comparing the results for the instance with 255 test cases with the results for the instance with 241 test cases of the subject program `replace` from Tables 11 and 12. Both instances have a similar number of test cases but the instance from the subject program `space` has more than 13× more test goals than the instance from the subject program `replace`. Comparing the number of genes processed for the two different instances, it can be appreciated that the performance of the CPU implementation of SGS is highly degraded (29.98×), while the degradation of the performance of the GPU implementation of SGS is merely 4.05×

6. Threats to validity

Threats to validity are concerned with potential risks that could jeopardize the trustworthiness of the conducted empirical study [31,32]. The identification and mitigation of such threats prevents the introduction of a bias in the study by the researcher subjectivity. Threats to validity can be classified in four categories: internal, external, conclusion and construct [31,32].

Internal validity is related to other factors that could have affected the results instead of the factors studied in the experiment. The specific heuristics for the TSMP used in the experiments where implemented by ourselves since there are no publicly available implementations. To control this threat, we have carefully performed code inspection and step-by-step execution for small cases in order to check that the results obtained are correct. It should be noted that the results obtained are coherent with the results presented in [17].

External validity is concerned with factors that could prevent the generalization of the results of the experiments. In the first place, the subject programs used in the experimental analysis are real world programs from the Siemens benchmark suite and the `space` program. Even though, they have been widely used in the literature, they are relatively small programs. For this reason, they might not be representative of large size programs. We plan to extend our experiments in a future work, including larger subject programs in order to minimize this threat. Secondly, parallel performance is highly sensitive to the features of the hardware platform used. However, since the gap in performance between GPUs and CPUs keeps widening day after day, it is reasonable to infer that new GPUs will produce even better performance results than the presented in this work.

Conclusion validity is related to the possibility to draw correct conclusions from the outcomes of an experiment. Since we are dealing with stochastic algorithms, we executed fifty independent runs of each instance and subject program for each of such algorithms. Then, statistical tests with a high confidence level (99%) have been used to assess the statistical significance of the experimental results obtained.

Construct validity is concerned with when measurements used in the experiments do not adequately capture the concepts they are supposed to measure. The execution time of the test cases and of the algorithms was measured as the wall-clock time using the system clock, which it is clear that captures the speed of execution as it is perceived by the user.

7. Related work

Related work can be classified into other approaches for the Test Suite Minimization Problem, similar ideas to the SGS algorithm, and SBSE and metaheuristics implementations on GPU.

Test Suite Minimization Problem Besides the context of the problem provided in Section 2.1, and the previous works already described in Section 4, there are some additional relevant papers that are commented next.

Arito et al. [13] presented an interesting methodology for obtaining the exact solution of the TSMP when all cost are equal to one, i.e., when reducing the number of test cases of the suite. The methodology consists in automatically transforming a TSMP instance in a Boolean satisfiability (SAT) problem and then solving it using a exact SAT solver. Since the runtime of the methodology proposed is still very high, the authors reduced the original instances using a highly aggressive strategy that consist in removing all test cases whose coverage is contained in another test case. The authors did not report the runtime of neither the transformation between TSMP and SAT nor the reduction of the instances. Although the methodology proposed is interesting, it can not be applied directly to the cost-aware TSMP.

There are also other approaches for finding the test suite with minimum cardinality that satisfies a set of goals. A modification of the HGS heuristic was proposed including additional coverage information from the test cases with the goal of improving the fault detection effectiveness of the reduced suites [27]. The performance of several test suite reduction algorithms, including the GRE and HGS heuristics and a genetic algorithm was experimentally evaluated [33]. The GA [34] uses binary strings, the one-point crossover, the bit-flip mutation and both a penalty function and a feasibilization strategy. In the experimental evaluation, the GA was outperformed by both classical heuristics. Finally, a reduction with a tie-breaking mechanism was proposed and incorporated in the GRE and HGS heuristics in order to avoid random selection when there are ties between test cases [35].

Also, evolutionary algorithms have been used for solving variants of the TSMP considering a cost associated to the test cases. A genetic algorithm is used for solving a variant of TSMP that includes the cost of the test case and a weight for each of the goals that have to be covered [36]. The GA uses binary strings, the one-point crossover and the bit-flip mutation. In the experimental evaluation, the GA outperformed a modified greedy algorithm that takes into account the cost and the coverage of the test cases.

Systolic Genetic Search Algorithm SGS algorithm is a recent research line which has been proposed in [8] after some preliminary explorations in [22,23]. Since the seminal works of systolic computing by [19,20], a few efforts have been devoted to devising optimization algorithms based on this paradigm. As a matter of fact, only in [37] and in [38] an implementation of a GA on VLSI and FPGA architectures in a systolic fashion is proposed. However, this research lines were early discarded for the reason that it was very complex to translate the GA operations into the recurrent equations required for the hardware definition.

A direct antecedent of SGS is Systolic Neighborhood Search (SNS) [39]. Both algorithms share the arrangement of solutions into a grid, but moves solutions horizontally and vertically, whereas SNS only circulates solutions horizontally. As a consequence, SNS manages a single solution on each cell, using a more simple search strategy than in SGS. For this reason, SGS can be considered as an advanced version of SNS.

More recently, in [40,41] a hardware-oriented GA for FPGA architectures was presented. The proposal, called Pipelined Genetic Propagation (PGP), is based on propagating and circulating a group of individuals in a directed graph structure. Data is transported in a pipelined manner between the nodes of the graph that perform genetic operations on the data. Each node performs a specific operation. There are selection nodes, crossover nodes and mutation nodes, which distinguishes substantially PGP with respect to SGS.

SBSE and Metaheuristics on GPU Although SBSE is usually a computationally demanding area because most problems has to be solved within a tight schedule and the instances used have a large size, the application of parallelism to SBSE has been scarce [7]. There are just a few related works that use GPUs to solve SBSE problems.

The multi-objective TSMP was addressed using a GPU to speed up the fitness calculation of a multi-objective EA [7,16]. In the proposed implementation in each generation the entire population is transferred from the CPU to the GPU, the fitness function is evaluated on the GPU and the results are transferred back to the CPU. It should be noted that transfers between CPU and GPU in both directions are one of the most costly operation for an hybrid CPU-GPU platform. The novelty of this proposal is how the fitness values are calculated, transforming the fitness evaluation of the population into a matrix–matrix multiplication operation. This multiplication was programmed by hand by the authors instead of using a linear algebra library already available on GPU. Then, this idea is adapted for an hybrid CPU-GPU implementation of the NSGA-II algorithm is used for tackling the multi-objective test case prioritization [42]. In this proposal, besides computing the fitness evaluation on the GPU as a matrix–matrix multiplication operation, the crossover operation is also computed on the GPU.

It is not possible to make a completely fair comparison between the performance of the parallel implementation presented by Yoo et al. [7,16] and this work since there are many differences between both approaches: they use a multi-objective algorithm, while we use a single-objective algorithm; in their proposal, the GPU is only used for computing the fitness function evaluation, while, in our proposal, the whole algorithm is implemented in the GPU; they transfer the population from the CPU to the GPU in each iteration, while we do not do that; the number of fitness evaluations and the population size of both approaches are different; etc. Additionally, it is very difficult to evaluate if both CPU-GPU platforms have a similar relative performance. Regardless of these caveats, it should be highlighted that the maximal runtime reduction of the GPU version vs. the CPU version of SGS achieved in this work (182.53× for the instance with 255 test cases for the subject program `space`) is far superior than the maximal speedup reported in previous works (25.09×) for instances with similar features.

In a completely different line of work, a process for automatically improving existing software systems using genetic programming was proposed [43]. This methodology, known as Genetic Improving, has been successfully applied for optimizing a CUDA stereo image processing system [44] and a CUDA 3D medical image registration software [45].

On the other hand, the use of GPUs has represented an inspiring domain for the research in parallel metaheuristics, experiencing a tremendous growth in the last years. Currently, most of the standard existing family of algorithms have already been ported to this new kind of devices [46]. Several authors have tackled the GPU implementation of genetic algorithms [47,48], Ant Colony

Optimization [49,50], etc. Also, most popular models of parallel EAs have already been addressed on GPU, like master–slave [47], distributed [51], and cellular [52,53] models, showing the time savings that could be attained using such devices.

8. Concluding remarks and future work

In this work, we have proposed a Systolic Genetic Search Algorithm for solving the cost-aware TSMP. We have performed an exhaustive experimental evaluation of SGS, as well as two EAs, a simple genetic algorithm and an elitist genetic algorithm, and four specific heuristics for the TSMP, namely GRE_E, HGS_E, $GREEDY_R$ and $GREEDY_E$. This evaluation is aimed at understanding if the algorithm proposed is able to provide better solutions that the other algorithms. Second, we have evaluated the computational performance of SGS on GPU in comparison with the GPU-based implementations of SGA and EGA. Then, we have evaluated how the size of test suite and the number of test requirements affect the performance of the CPU and GPU implementation of SGS. The most important findings of this experimental evaluation can be summarized as follows:

- SGS is the algorithm with the best numerical performance for all the subject programs considered in the study. It is clear that the underlying search engine of SGS is advantageous for solving the cost-aware TSMP since the basic search operators are common to the other evolutionary algorithms evaluated. Additionally, SGS is able to outperform four specific heuristics for this problem, without using any problem-specific knowledge.
- Even though SGA is the algorithm with the best runtime reduction between the CPU implementation and the GPU implementation, this reduction is explained by the poor performance of the CPU-based implementation rather than by a particular improvement in the GPU implementation.
- The GPU-based SGS is the algorithm with the best computational performance. The differences in the runtime of SGS with the other GPU-based implementations are caused by the underlying search engine of SGS.
- Although both SGS implementations show a good scalability behavior when solving larger instances (instances with more test cases), the scalability of the GPU-based implementation is far superior than the scalability of the CPU-based implementation.
- The performance of the CPU-based SGS is highly degraded when considering a larger number of test requirements, while the performance of the GPU-based SGS is just minimally degraded.

In summary, the GPU implementation of SGS is not only able to reach excellent quality solutions for the cost-aware TSMP, but it is also able to do it fast. Moreover, GPU-based SGS shows a good scalability behavior when solving high dimension problem instances. This makes it possible to solve real-world software testing environments in which decisions have to be taken within a tight schedule.

Four main areas that deserve further study are identified. A first issue is to extend our experiments with other real-world benchmarks, specially including test suites that include test cases with larger execution times than the used in this work. Secondly, since the cost model of the test case could influence the numerical performance of the proposed algorithm, we aim to conduct a study on the sensitivity of SGS to different cost models by creating synthetic instances that include scenarios with similar test costs and with great variability test cost. Then, given that the numerical results obtained by $GREEDY_E$ are acceptable and that it has a really short execution time, a third line of interest is to study the effect of initializing some solutions of the SGS population using this specific heuristic or even using all the specific heuristics evaluated. This may help to speed up the search of the SGS and even lead to better solutions. Finally, we aim to widen the perspective including theoretical and empirical studies of the impact of the distribution of the crossover and mutation points throughout the systolic grid in the quality of the solutions obtained by SGS and in its speed of convergence.

Acknowledgements

M. Pedemonte acknowledges partial support from Programa de Desarrollo de las Ciencias Básicas, Sistema Nacional de Investigadores and Agencia Nacional de Investigación e Innovación, Uruguay. E. Alba is partially funded by the Spanish MINECO and FEDER project TIN2014-57341-R (http://moveon.lcc.uma.es). F. Luna acknowledges support from Consejería de Ciencia e Innovación, Gobierno de Extremadura, and FEDER under contract PRI13113.

References

[1] M.J. Harrold, Testing: a roadmap, in: Proceedings of the Conference on the Future of Software Engineering, ICSE'00, ACM, 2000, pp. 61–72.
[2] G. Tassey, The Economic Impacts of Inadequate Infrastructure for Software Testing, National Institute of Standards and Technology, RTI Project 7007 (011), 2002.
[3] Y.-D. Lin, C.-H. Chou, Y.-C. Lai, T.-Y. Huang, S. Chung, J.-T. Hung, F.C. Lin, Test coverage optimization for large code problems, J. Syst. Softw. 85 (1) (2012) 16–27.
[4] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: a survey, Softw. Test. Verif. Reliab. 22 (2) (2012) 67–120.
[5] M. Harman, S.A. Mansouri, Y. Zhang, Search-based software engineering: trends, techniques and applications, ACM Comput. Surv. 45 (1) (2012), 11:1–11:61.
[6] E. Alba (Ed.), Parallel Metaheuristics: A New Class of Algorithms, Wiley, 2005.
[7] S. Yoo, M. Harman, S. Ur, Highly scalable multi objective test suite minimisation using graphics cards, in: Proceedings of the Third International Conference on Search Based Software Engineering, 2011, pp. 219–236.
[8] M. Pedemonte, F. Luna, E. Alba, Systolic genetic search, a systolic computing-based metaheuristic, Soft Comput. 19 (7) (2015) 1779–1801.
[9] G. Rothermel, M.J. Harrold, J. von Ronne, C. Hong, Empirical studies of test-suite reduction, Softw. Test. Verif. Reliab. 12 (4) (2002) 219–249.
[10] J. Offutt, J. Pan, J. Voas, Procedures for reducing the size of coverage-based test sets, in: Proc. of the Twelfth Int. Conf. on Testing Computer Software, 1995, pp. 111–123.
[11] M.J. Harrold, R. Gupta, M.L. Soffa, A methodology for controlling the size of a test suite, ACM Trans. Softw. Eng. Methodol. 2 (3) (1993) 270–285.
[12] T. Chen, M.F. Lau, Heuristics towards the optimization of the size of a test suite, in: Proc. of the 3rd Int. Conf. on Software Quality Management, 1995, pp. 415–424.
[13] F. Arito, J.F. Chicano, E. Alba, On the application of sat solvers to the test suite minimization problem, in: SSBSE, Vol. 7515 of LNCS, Springer, 2012, pp. 45–59.
[14] M. Pedemonte, F. Luna, E. Alba, Systolic genetic search for software engineering: the test suite minimization case, in: Applications of Evolutionary Computation, Vol. 8602 of Lecture Notes in Computer Science, Springer, 2014, pp. 678–689.
[15] T.Y. Chen, M.F. Lau, A new heuristic for test suite reduction, Inf. Softw. Technol. 40 (5) (1998) 347–354.
[16] S. Yoo, M. Harman, S. Ur, GPGPU test suite minimisation: search based software engineering performance improvement using graphics cards, Empir. Softw. Eng. 18 (3) (2013) 550–593.
[17] C.-T. Lin, K.-W. Tang, G.M. Kapfhammer, Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inf. Softw. Technol. 56 (10) (2014) 1322–1344.
[18] D. Kirk, W. Hwu, Programming Massively Parallel Processors, Second Edition: A Hands-on Approach, Morgan Kaufmann, 2012.
[19] H.T. Kung, Why systolic architectures? Computer 15 (1) (1982) 37–46.
[20] H.T. Kung, C.E. Leiserson, Systolic arrays (for VLSI), in: Sparse Matrix Proceedings, 1978, pp. 256–282.
[21] A.C. Guyton, J.E. Hall, Textbook of Medical Physiology, 11th ed., Elsevier Saunders, 2006.
[22] M. Pedemonte, E. Alba, F. Luna, Towards the design of systolic genetic search, in: IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IEEE Computer Society, 2012, pp. 1778–1786.
[23] M. Pedemonte, F. Luna, E. Alba, New ideas in parallel metaheuristics on GPU: systolic genetic search, in: S. Tsutsui, P. Collet (Eds.), Massively Parallel Evolutionary Computation on GPGPUs, Natural Computing Series, Springer, 2013, pp. 203–225, Ch. 10.

[24] A.M. Smith, G.M. Kapfhammer, An empirical study of incorporating cost into test suite reduction and prioritization, in: Proceedings of the 2009 ACM Symposium on Applied Computing, ACM, 2009, pp. 461–467.
[25] M. Hutchins, H. Foster, T. Goradia, T. Ostrand, Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria, in: Proc. of the 16th Int. Conf. on Software Engineering, 1994, pp. 191–200.
[26] H. Do, S. Elbaum, G. Rothermel, Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact, Empir. Softw. Eng. 10 (4) (2005) 405–435.
[27] D. Jeffrey, N. Gupta, Improving fault detection capability by selectively retaining test cases during test suite reduction, IEEE Trans. Softw. Eng. 33 (2) (2007) 108–123.
[28] J. Derrac, S. García, D. Molina, F. Herrera, A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms, Swarm Evol. Comput. 1 (1) (2011) 3–18.
[29] D.J. Sheskin, Handbook of Parametric and Nonparametric Statistical Procedures, 5th ed., Chapman and Hall/CRC, 2011.
[30] F.I. Vokolos, P.G. Frankl, Empirical evaluation of the textual differencing regression testing technique, in: International Conference on Software Maintenance, 1998. Proceedings, 1998, pp. 44–53, http://dx.doi.org/10.1109/ICSM.1998.738488.
[31] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer Science & Business Media, 2012.
[32] R.K. Yin, Case Study Research: Design and Methods, Sage publications, 2013.
[33] H. Zhong, L. Zhang, H. Mei, An experimental study of four typical test suite reduction techniques, Inf. Softw. Technol. 50 (6) (2008) 534–546.
[34] N. Mansour, K. El-Fakih, Simulated annealing and genetic algorithms for optimal regression testing, J. Softw. Maint. Res. Pract. 11 (1) (1999) 19–34.
[35] J.-W. Lin, C.-Y. Huang, Analysis of test suite reduction with enhanced tie-breaking techniques, Inf. Softw. Technol. 51 (4) (2009) 679–690.
[36] X. Ma, Z. He, B. Sheng, C. Ye, A genetic algorithm for test-suite reduction, in: 2005 IEEE International Conference on Systems, Man and Cybernetics, vol. 1, IEEE, 2005, pp. 133–139.
[37] H. Chan, P. Mazumder, A systolic architecture for high speed hypergraph partitioning using a genetic algorithm, in: Progress in Evolutionary Computation, vol. 956 of Lecture Notes in Computer Science, Springer, Berlin/Heidelberg, 1995, pp. 109–126.
[38] G. Megson, I. Bland, Synthesis of a systolic array genetic algorithm, in: Parallel Processing Symposium, 1998. IPPS/SPDP 1998, 1998, pp. 316–320.
[39] P. Vidal, F. Luna, E. Alba, Systolic neighborhood search on graphics processing units, Soft Comput. (2013) 1–18.
[40] L. Guo, C. Guo, D. Thomas, W. Luk, Pipelined genetic propagation, in: 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2015, pp. 103–110.
[41] S. Shao, L. Guo, C. Guo, T. Chau, D. Thomas, W. Luk, S. Weston, Recursive pipelined genetic propagation for bilevel optimisation, in: 2015 25th International Conference on Field Programmable Logic and Applications (FPL), 2015, pp. 1–6.
[42] Z. Li, Y. Bian, R. Zhao, J. Cheng, A fine-grained parallel multi-objective test case prioritization on GPU, in: Search Based Software Engineering, Springer, 2013, pp. 111–125.
[43] W.B. Langdon, M. Harman, Optimising existing software with genetic programming, IEEE Trans. Evol. Comput. 19 (1) (2015) 118–135.
[44] W.B. Langdon, M. Harman, Genetically improved CUDA c++ software, in: Genetic Programming, Springer, 2014, pp. 87–99.
[45] W.B. Langdon, M. Modat, J. Petke, M. Harman, Improving 3D medical image registration CUDA software with genetic programming, in: Proceedings of the 2014 Conference on Genetic and Evolutionary Computation, ACM, 2014, pp. 951–958.
[46] W.B. Langdon, Graphics processing units and genetic programming: an overview, Soft Comput. 15 (8) (2011) 1657–1669.
[47] O. Maitre, F. Krüger, S. Querry, N. Lachiche, P. Collet, EASEA: specification and execution of evolutionary algorithms on GPGPU, Soft Comput. 16 (2) (2012) 261–279.
[48] M. Pedemonte, E. Alba, F. Luna, Bitwise operations for GPU implementation of genetic algorithms, in: Genetic and Evolutionary Computation Conference, GECCO'11 – Companion Publication, 2011, pp. 439–446, http://dx.doi.org/10.1145/2001858.2002031.
[49] J.M. Cecilia, J.M. García, M. Ujaldon, A. Nisbet, M. Amos, Parallelization strategies for ant colony optimisation on GPUs, in: 25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Workshop Proceedings, 2011, pp. 339–346.
[50] M. Pedemonte, S. Nesmachnow, H. Cancela, A survey on parallel ant colony optimization, Appl. Soft Comput. 11 (8) (2011) 5181–5197.
[51] S. Zhang, Z. He, Implementation of parallel genetic algorithm based on CUDA, in: ISICA 2009, LNCS 5821, 2009, pp. 24–30.
[52] N. Soca, J. Blengio, M. Pedemonte, P. Ezzatti, PUGACE, a cellular evolutionary algorithm framework on GPUs, in: 2010 IEEE World Congress on Computational Intelligence, WCCI 2010 – 2010 IEEE Congress on Evolutionary Computation, CEC 2010, 2010, pp. 1–8, http://dx.doi.org/10.1109/CEC.2010.5586286.
[53] P. Vidal, E. Alba, Cellular genetic algorithm on graphic processing units, in: Nature Inspired Cooperative Strategies for Optimization (NICSO 2010), 2010, pp. 223–232.

Appendix C

A Theoretical and Empirical Study of the Trajectories of Solutions on the Grid of Systolic Genetic Search

Information Sciences 445–446 (2018) 97–117

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

A theoretical and empirical study of the trajectories of solutions on the grid of Systolic Genetic Search

Martín Pedemonte[a,*], Francisco Luna[b], Enrique Alba[b]

[a] Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Julio Herrera y Reissig 565, Montevideo 11300, Uruguay
[b] Depto. de Lenguajes y Ciencias de la Computación, Univ. de Málaga, E.T.S. Ingeniería Informática, Campus de Teatinos, Málaga 29071, Spain

A R T I C L E I N F O

Article history:
Received 4 February 2017
Revised 31 August 2017
Accepted 15 February 2018
Available online 15 March 2018

Keywords:
Systolic Genetic Search
Evolutionary algorithms
Parallel Metaheuristics
Systolic Computing
GPU

A B S T R A C T

Systolic Genetic Search (SGS) is a recently proposed optimization algorithm based on the circulation of solutions through a bidimensional grid of cells and the application of evolutionary operators within the cells to the moving solutions. Until now, the influence of the solutions flow on the results of SGS has only been empirically studied. In this article, we theoretically analyze the trajectories of the solutions along the grid of SGS. This analysis shows that, in the grids used so far, there are cells in which the incoming solutions are descendants of a pair of solutions that have been previously mated. For this reason, we propose a new variant of SGS which uses a grid that guarantees that, given a pair of solutions that coincide in any cell, a pair of ancestors of these two solutions have not been previously mated. The experimental evaluation conducted on three deceptive problems shows that SGS has a better numerical efficiency when it uses grids that limit the mating of descendants of pairs of solutions that have already been mated. It also shows that this property helps to keep a larger diversity in the pairs of solutions that are mated in each cell.

1. Introduction

Evolutionary Algorithms (EAs) are stochastic search methods inspired by the natural process of evolution of species. EAs are guided by the *survival of the fittest* principle applied to candidate solutions through the selection of the best fitted individuals for reproduction, and it involves the probabilistic application of evolutionary operators to find better solutions. In the traditional sequential EA, the population is organized into a single group, mating individuals without limitations. Due to the emergence of Parallel EAs [3,22], two different population models have gained great popularity: the distributed or island model and the cellular model. In the island model [1,5], the population is partitioned into subpopulations that evolve semi-independently and the selection of parents for reproduction is limited to individuals that belong to the same subpopulation. In the cellular model [2], the population is structured in many small overlapping neighborhoods and the selection of parents for reproduction is local to each neighborhood.

Systolic Genetic Search (SGS) [26,29] is a recently proposed optimization algorithm that merges ideas from *Systolic Computing* and *Genetic Algorithms*. The algorithm was explicitly designed to exploit the high degree of parallelism available in

* Corresponding author.
E-mail addresses: mpedemon@fing.edu.uy (M. Pedemonte), flv@lcc.uma.es (F. Luna), eat@lcc.uma.es (E. Alba).

https://doi.org/10.1016/j.ins.2018.02.033
0020-0255/© 2018 Elsevier Inc. All rights reserved.

modern GPU architectures. It has already shown its potential for tackling benchmark and real world problems, finding optimal or near optimal solutions in short execution times. SGS optimizes a set of solutions that flows through a bidimensional grid of cells following a synchronous and structured plan through a horizontal and a vertical data flow. In each cell, adapted evolutionary operators are applied to the tentative circulating solutions in order to obtain better solutions that continue moving across the cells of the grid.

SGS differs from the population models mentioned above because the interactions in SGS are limited by the subpopulations, being only possible interactions between two solutions that belong to the two different data flows. Also, the selection of parents for reproduction in SGS is implicit (SGS does not have an explicit selection process), and it is determined by the flow of solutions through the grid. In spite of the success of SGS for solving hard binary optimization problems, little is known about how the underlying search engine affects the general behavior of the algorithm. This motivates us to conduct a formal analysis of the trajectories described by the solutions along the grid, in order to gain insight into the operating mechanism of SGS.

In this article, we investigate the relation between the pairing of solutions of the subpopulations of SGS_B (one of the most effective flows for SGS) and the numerical efficiency of the algorithm. With this in mind, we theoretically analyze the trajectories described by the solutions for two different grids that have been used experimentally, examining especially the pairing of solutions of the two subpopulations that are produced along the grid. In this analysis, we found that there are cells in which both incoming solutions are direct descendants of a pair of solutions that have already been mated in another cell of the grid. This could prevent that highly fitted genetic material of the best circulating solutions of a data flow can be shared with a large part of the solutions of the other data flow, potentially compromising the exploitation of good regions of the search space. Because of this, we design a new variant of SGS that uses an original grid specially conceived to overcome this limitation. The novel variant of SGS has better theoretical properties than the variants used so far. In addition to the theoretical analysis, an experimental evaluation is also conducted to examine how the different grids impact on the effectiveness of SGS for solving three deceptive problems. We can summarize the contributions of this work as follows:

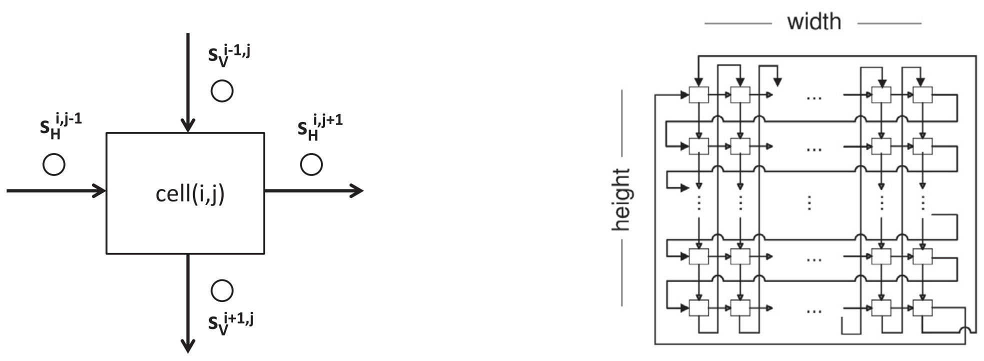
- It presents a theoretical analysis of the trajectories described by the solutions in two different grids that have been previously used for SGS [28,30]. As a result of this analysis, a new variant of SGS is designed, introducing a new grid with better theoretical properties that prevents the mating of descendants of pairs of solutions that have already been mated.
- It shows that the new variant proposed of SGS is the best performing algorithm in the experimental evaluation, being able to obtain optimal or almost optimal solutions for the three deceptive problems studied in this article. It also shows that the second best performing algorithm also uses a grid topology in which the mating of descendants of solution pairs that have already been mated is limited.
- It shows that SGS consistently outperforms two competitive genetic algorithms with the same evolutionary operators as the SGS algorithms for the three deceptive problems considered. This result corroborates that the success of SGS for solving these problems is caused by the underlying search engine of SGS.
- It reveals that the diversity in each cell of the grid of SGS (how different are the individuals that are being mated in each cell) is larger when the mating of descendants of pairs of solutions that have already been mated is prevented or limited, as in the newly proposed grid.

This article is organized as follows. Section 2 presents the main features of the SGS algorithm used in this work and it also discusses related papers from the literature. Then, in Section 3, we provide a theoretical analysis of the trajectories of the solutions on two different grids already used, as well as, we introduce a new grid that produces different pairing of solutions for mating in each cell of the grid in each step of the algorithm. The experimental design used for evaluating the three different grids and the results of the experimental evaluation are presented in Section 4. Finally, in Section 5, we outline the conclusions of this work and suggest future research directions.

2. Systolic genetic search

Systolic Computing is inspired by the physiology of the cardiovascular system [16,21]. In particular, in the systole phase, the heart contracts, increasing the pressure inside the cavities. As a result, the heart ejects blood into the arterial system with a regular cadence to meet the metabolic needs of the tissues. Systolic computing architectures are composed of simple data processing units, which are usually called cells, that are connected through a network. The cells are able to compute relatively simple operations to data received from neighboring units. The network allows a data flow between neighboring cells with a regular cadence, as in the systole phase of the cardiac cycle.

SGS [26–30] is a recently proposed optimization algorithm that adapts the operation of genetic algorithms to a systolic computing architecture. Several aspects of the SGS algorithm have to be defined such as the flow of solutions through the grid (how is the interconnection topology of the systolic structure and how do the solutions move through this structure), the dimension of the grid, and the computation of the cells (which operations are applied to the tentative solutions in each cell). In the rest of this section, we describe those aspects of the SGS used in this work, and we also discuss related papers from the literature.



(a) Ingoing and outgoing solutions from *cell*(i, j). (b) Interconnection topology in SGS_B.

Fig. 1. Flow of solutions in SGS_B.

2.1. Flow of solutions

SGS algorithm uses a bidimensional grid of cells in which the solutions circulate synchronously through an horizontal and a vertical data streams. At each step of SGS, two solutions enter each *cell*(i, j), $s_\text{H}^{i,j-1}$ from the horizontal data stream ($cell(i, j-1)$) and $s_\text{V}^{i-1,j}$ from the vertical data stream ($cell(i-1, j)$). Then, the cell computation is performed generating two (potentially new) solutions that continue moving through the grid, $s_\text{H}^{i,j+1}$ through the horizontal data stream ($cell(i, j+1)$) and $s_\text{V}^{i+1,j}$ through the vertical data stream ($cell(i+1, j)$), as it is shown in Fig. 1a.

In this work, we adopt the SGS_B data flow since it has shown empirically to be highly effective, as well as it has outperformed in terms of execution time other three data flows studied [26,27,29]. In SGS_B, the solutions flow either horizontally (moving in the same row), or vertically (moving in the same column), when a solution reaches the last cell of the last row/column of the grid is passed on to the cell of the first row/column of the next column/row. The interconnection topology and the solution flow of SGS_B are shown in Fig. 1b.

2.2. Grid of the SGS

In [26,27,29] a rectangular grid was used, which was designed to allow SGS to achieve an adequate exploration and to take advantage of the parallel computation capabilities offered by GPUs. In this grid, the width is l (the length of the tentative solutions) and the height is $\lceil \lg l \rceil$. As a consequence, the number of solutions of the population is $2 \times l \times \lceil \lg l \rceil$ (2 solutions per cell) for solving problem instances of size l. Although the idea is to have a relatively large number of cells to take advantage of the parallel computation capabilities offered by GPUs, this large number of cells can compromise the computational performance.

A proper balance could be obtained using grids with at least l cells [28,30]. In [28] a square $\lceil \sqrt{l} \rceil \times \lceil \sqrt{l} \rceil$ grid was used. If $\lceil \sqrt{l} \rceil$ is an integer, the grid has exactly l cells, otherwise it has some additional cells. As a consequence, the population size is $2 \times \lceil \sqrt{l} \rceil \times \lceil \sqrt{l} \rceil$. Additionally, in [30] a rectangular $\lceil \sqrt{l} \rceil \times \left(\lceil \sqrt{l} \rceil + 1\right)$ grid was used instead of the square grid. In Section 3, we present a theoretical study on the trajectories of solutions for both aforementioned grids.

2.3. Cell computation

Initially, each cell generates two random solutions, one for each data stream. At each step of SGS, which is known as systolic step, one solution enters each cell from the horizontal data stream and one solution enters each cell from the vertical data stream. Adapted genetic operators (crossover and mutation) are applied to the incoming solutions in order to generate two potentially new solutions. Then, the cell uses elitism to determine the outgoing solutions that continue moving through the grid, choosing for each data stream between the incoming solution and a newly generated one. The use of elitism is essential since SGS has no selection process. Finally, each cell sends the outgoing solutions to the next cells of each of the data streams. Algorithm 1 presents the pseudocode of the algorithm.

SGS can be adapted to any solution representation and any particular operator. Since we are dealing with binary problems in this work, we encode the solutions as binary strings. The evolutionary search operators are the two-point crossover and the bit-flip mutation. As the two-point crossover is applied on each cell, two different crossover point values are chosen randomly for each cell. The bit-flip mutation operator flips a single bit in each solution of each cell. In previous works [28,30], with the aim of reducing the generation of random numbers during the execution of SGS, the mutation point was preprogrammed at fixed positions of the tentative solutions according to the position of the cell on the grid. In this work, we analyze a single factor of SGS (the trajectories of the solutions through the grid) in isolation to understand its effect on the algorithm, so we have preferred to follow a more traditional approach and change a bit chosen randomly for each cell.

Algorithm 1: Systolic Genetic Search.

```
foreach cell c do
    $s_H$ = generateRandomSolution()
    $s_V$ = generateRandomSolution()
    sendSolutionThroughHorizontalDataStream($S_H, c$)
    sendSolutionThroughVerticalDataStream($S_V, c$)
end
for $i = 1$ to maxGeneration do
    foreach cell c do
        $s_H$ = receiveSolutionFromHorizontalDataStream($c$)
        $s_V$ = receiveSolutionFromVerticalDataStream($c$)
        $(new_H, new_V)$ = crossover($s_H, s_V$)
        $new_H$ = mutation($new_H$)
        $new_V$ = mutation($new_V$)
        $new_H$ = elitism($s_H, new_H$)
        $new_V$ = elitism($s_V, new_V$)
        sendSolutionThroughHorizontalDataStream($new_H, c$)
        sendSolutionThroughVerticalDataStream($new_V, c$)
    end
end
```

2.4. Related works

This section analyzes published material which is related to similar ideas to the SGS algorithm. Most of the research in the use of GPUs for implementing parallel metaheuristics have followed the approach of porting an existing family of algorithms to these devices [17,20]. As a consequence, several works show the benefits in time savings of implementing master-slave [23], island [39], and cellular [34,35] models of parallel metaheuristics on GPU, including most popular techniques like Genetic Algorithms (GAs) [23,39], Ant Colony Optimization [6,7], Particle Swarm Optimization [40], etc. On the other hand, SGS is a new research line specially conceived for exploiting the high degree of parallelism available in modern GPU architectures.

SGS algorithm has been proposed in [29] after some preliminary ideas that have been explored [26,27]. Few efforts have been devoted to designing optimization algorithms based on systolic computing-like architectures [18,19]. In [8] and in [24] an implementation of a GA on VLSI and FPGA architectures in a systolic fashion is proposed. However, this research line was early discarded since it was hard to translate the GA operations into the recurrent equations required for the hardware configuration.

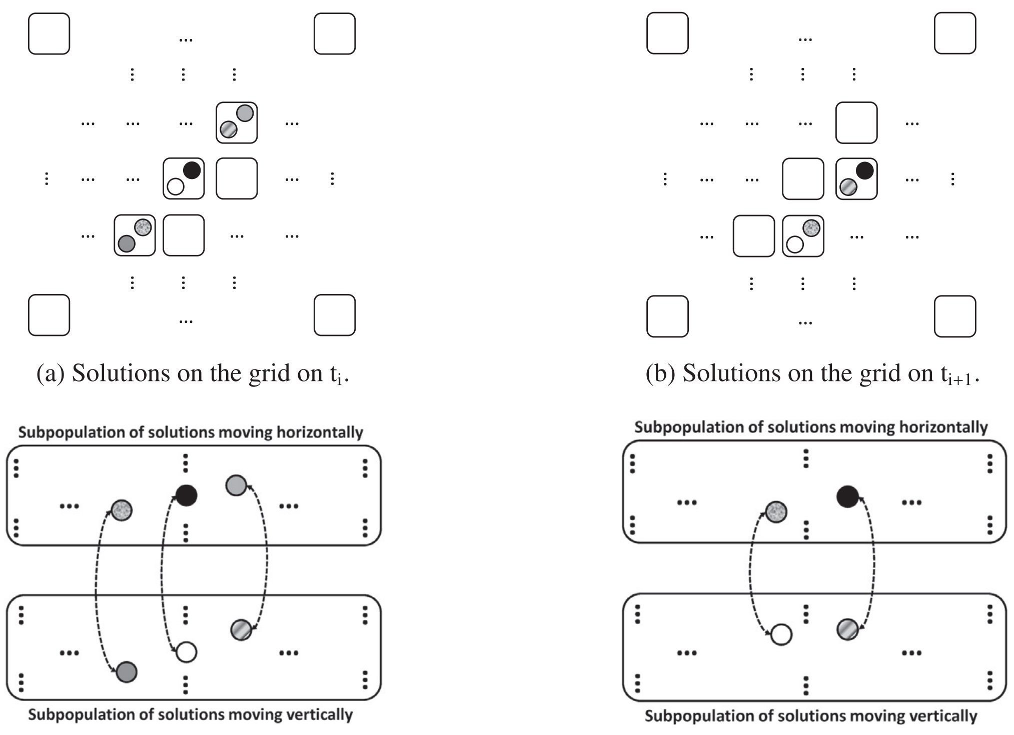
A direct antecedent of SGS is Systolic Neighborhood Search (SNS) [4,36,37]. As a matter of fact, SGS can be seen as an advanced version of SNS. Both algorithms share the arrangement of solutions into a grid, although SNS only circulates solution through a horizontal data stream, while SGS moves solutions not only through a horizontal data stream but also through a vertical data stream. Indeed, SNS manages a single solution in each cell, while SGS manages pairs of solutions in each cell, thus allowing to devise more complex search strategies than in SNS. For this reason, SGS can be considered as an advanced version of SNS.

More recently, in [15,32] a hardware-oriented GA for FPGA architectures was presented. The proposal, called Pipelined Genetic Propagation (PGP), is based on propagating and circulating a group of individuals in a directed graph structure. Data is propagated in a pipelined manner between the nodes of the graph, which are able to perform genetic operations on the circulating data. Each type of node performs a particular operation. There are selection nodes, crossover nodes and mutation nodes, which distinguishes substantially PGP from SGS.

3. Theoretical analysis of the trajectories of solutions on the grid

In SGS_B, the population is divided into two non-overlapping subpopulations, one subpopulation is composed of the solutions that are moving horizontally and the other subpopulation corresponds to the solutions that are moving vertically. At every step of SGS_B, each of the solutions moving horizontally mates with a different solution moving vertically. Therefore, mating is limited by the subpopulations, being only possible to produce interactions between two solutions that belong to the two different subpopulations. The pairing of solutions between subpopulations is not static and it changes in every iteration with the movement of the solutions over the grid, as it is shown in Fig. 2.

The population model of SGS_B is radically different from the other population models used in evolutionary algorithms. In SGS_B, two solutions are mated, one from each subpopulation, while the mating process is local to each subpopulation in the island model and each neighborhood in the cellular model. Additionally, in SGS_B, the pairings for mating are changing


Subpopulation of solutions moving horizontally
Subpopulation of solutions moving vertically
Subpopulation of solutions moving horizontally
Subpopulation of solutions moving vertically

(a) Solutions on the grid on t_i.

(b) Solutions on the grid on t_{i+1}.

(c) Mating of SGS_B on t_i.

(d) Mating of SGS_B on t_{i+1}.

Fig. 2. Population model of SGS_B.

dynamically as the solutions move through the grid, while in the other models the subpopulations considered for mating are in general static.

The selection pressure of both panmictic (traditional sequential) and cellular model EAs has been studied through the analysis of the *takeover time* [12,13,31]. The *takeover time* is defined as the time that takes for the best individual to take over the entire population. This metric can be estimated experimentally by measuring the propagation of the best individual only considering the effect of selection (omitting the effect of the crossover and mutation operators). Since SGS_B does not have an explicit selection process, such study cannot be performed in our case. In SGS_B, the selection of parents for mating is determined by the flow of solutions through the grid. Let S_H and S_V be the subpopulation of solutions moving horizontally and vertically, respectively, and $s_H \in S_H$ and $s_V \in S_V$ be two solutions that are located in the same cell at a given time. We define the *meeting time* of the cell as the minimum number of steps or movements that take for s_H and s_V to coincide again in a cell. Eq. 1 formally defines the *meeting time* of cell (i, j), where *moveStepsHorizontal*(x, y, s) and *moveStepsVertical*(x, y, s) calculate the final position of a solution that starts at cell (x, y) after s horizontal and vertical steps, respectively. For this definition, we only consider the effect of movements through the grid, without taking into account the application of variational operators (as in the *takeover time*). The *meeting time* of a cell can take values from one to the total number of cells of the grid.

$$meetingTime(x,y) = s \iff s \text{ is the minimal value such that:} \begin{cases} s > 0 \\ (x',y') = moveStepsHorizontal(x,y,s) \\ (x',y') = moveStepsVertical(x,y,s) \end{cases} \tag{1}$$

The *meeting time* is a useful metric for analyzing the trajectories described by the solutions through the grid, in order to gain insight into the interactions that are produced along the grid between both subpopulations in SGS_B. It is desirable that the best individual could interact in reproduction with as many as possible solutions from the other subpopulation (especially since the cells use elitism), assuring that the highly fitted genetic material is diffused along the grid. The larger the value of *meeting time* of the cells, the larger the number of different solutions with which the best individual would interact.

In the next two sections, we analyze the trajectories described by the solutions in the square and rectangular grids previously proposed. In this analysis, we divide the cells of the grid into different groups and independently calculate the *meeting time* of the cells of these groups. For this purpose, we show for each group that two solutions starting in the same cell, coincide again in a cell after the same number of horizontal and vertical steps. Then, we present a new grid in which the *meeting time* of all the cells of the grid is equal to the number of cells of the grid. After that, we compare the main theoretical features of the three grids. In Section 4, we experimentally evaluate the grids to understand how the different *meeting time* affects the behavior of SGS especially regarding its numerical and computational efficiency.

3.1. The square $\lceil\sqrt{l}\rceil \times \lceil\sqrt{l}\rceil$ grid

Let h be the height of the grid, i.e., the number of cells in a column, and w be the width of the grid, i.e., the number of cells in a row. In the square grid used in [28], $h = w = \lceil\sqrt{l}\rceil$. For the sake of simplicity we consider a 0-based index grid, i.e., the index of the cells ranges from 0 to $h-1$: $0 \le i \le h-1$ and $0 \le j \le h-1$. Let us consider three different groups of cells: the cells that do not belong to the last row or the last column of the grid, the cells from the last row of the grid, and the cells from the last column of the grid.

Theorem 3.1. *The* meeting time *of the cells that do not belong to the last row or the last column of the grid (i.e., cell(i, j) with $i < h-1$ and $j < h-1$) is $h+1$.*

Proof. It should be noted that $i+1 < h$ and $j+1 < h$ since $i < h-1$ and $j < h-1$. After $h+1$ steps through the horizontal flow, s_H is located in $cell(i+1, j+1)$, i.e., the next cell of the diagonal[1], since[2]:

$$(i + ((j + h + 1) \text{ div } h)) \text{ mod } h = (i+1) \text{ mod } h = i+1, \text{ and} \tag{2}$$

$$(j + (h+1)) \text{ mod } h = j+1. \tag{3}$$

After $h+1$ steps through the vertical flow, s_V is also in the $cell(i+1, j+1)$, since:

$$(i + (h+1)) \text{ mod } h = i+1, \text{ and} \tag{4}$$

$$(j + ((i + h + 1) \text{ div } h)) \text{ mod } h = (j+1) \text{ mod } h = j+1. \tag{5}$$

□

Theorem 3.2. *The* meeting time *of the cells that belong to the last row of the grid (i.e., $cell(h-1, j)$) is $h^2 - jh - j$.*

Proof. After $h^2 - jh - j$ steps through the horizontal flow, s_H is in the first cell of the diagonal ($cell(h-j-1, 0)$) since:

$$(h - 1 + ((h^2 - jh - j) \text{ div } h)) \text{ mod } h = (h + (h - j - 1)) \text{ mod } h = h - j - 1, \text{ and} \tag{6}$$

$$(j + (h^2 - jh - j)) \text{ mod } h = (h \times (h - j)) \text{ mod } h = 0. \tag{7}$$

After $h^2 - jh - j$ steps through the vertical flow s_V is also in the $cell(h-j-1, 0)$, since:

$$(h - 1 + (h^2 - jh - j)) \text{ mod } h = ((h \times (h - j)) + (h - j - 1)) \text{ mod } h = h - j - 1, \text{ and} \tag{8}$$

$$(j + ((h^2 - jh - j) \text{ div } h)) \text{ mod } h = (j + (h - j)) \text{ mod } h = 0 \tag{9}$$

□

Equivalently to Theorem 3.2, it can be proved that after $h^2 - ih - i$ movements from the cells of the grid that belong to the last column (i.e., $cell(i, h-1)$), both s_H and s_V return to the first cell of the diagonal (i.e., $cell(0, h-i-1)$), i.e., the *meeting time* of these cells is $h^2 - ih - i$.

As a consequence of these theorems, two solutions s_H and s_V, which coincide in a cell, will coincide in all the cells of the diagonal. Fig. 3 shows with dashed lines the coincidences of solutions moving horizontally and vertically through the grid. For this reason, each solution moving horizontally/vertically would only interact in reproduction with a fixed number of solutions moving vertically/horizontally that is equal to the number of diagonals of the grid, i.e., $w + h - 1 = 2h - 1$. The ratio of solutions from one subpopulation with which any given solution of the other subpopulation is mated can be computed as the number of diagonals of the grid divided by the total number of solutions of the subpopulation, i.e., $\frac{2h-1}{h \times h}$.

3.2. A rectangular $\lceil\sqrt{l}\rceil \times \left(\lceil\sqrt{l}\rceil + 1\right)$ grid

In the rectangular grid used in [30], $h = \lceil\sqrt{l}\rceil$ and $w = \lceil\sqrt{l}\rceil + 1$, i.e., $w = h+1$. We also use a 0-based index grid, i.e., the index of the cells ranges from 0 to h horizontally ($0 \le j \le h$) and from 0 to $h-1$ vertically ($0 \le i \le h-1$). Let us consider five different groups of cells: the single cell at position $(h-1, h)$, the single cell at position (0, 0), the cells at position (i, h) with $0 \le i \le h-3$, the cells at position $(i, 0)$ with $2 \le i \le h-1$, and the rest of the cells of the grid.

It holds trivially that *meeting time* of $cell(h-1, h)$ is 1 since the next cell through both the horizontal and the vertical flows is $cell(0, 0)$.

[1] The grid can be seen as a matrix. The concept of diagonal is proper of matrices.

[2] We note the integer division with the operand `div`. We note the remainder of the integer division with the operand `mod`.

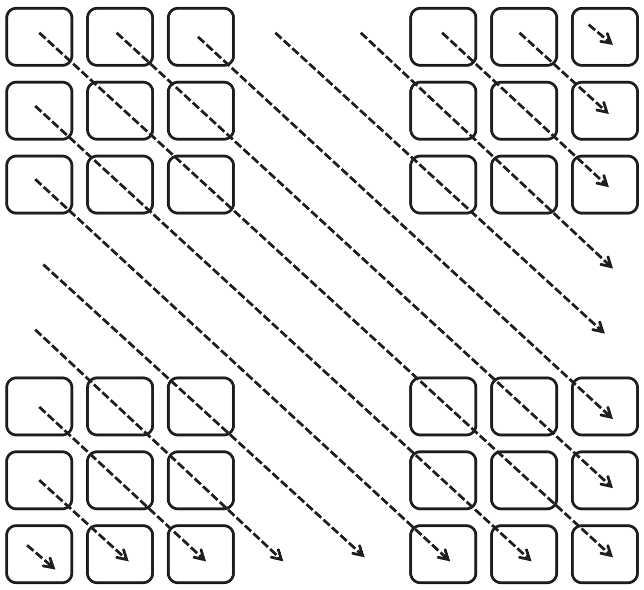


Fig. 3. Coincidences of solutions moving horizontally and vertically on the square grid.

Theorem 3.3. *The* meeting time *of cell*(0, 0) *is* $h^2 + h - 1$.

Proof. $h^2 + h - 1$ is equal to $(h-1) \times (h+1) + h$. After $h^2 + h - 1$ steps through the horizontal flow, s_H is in $cell(h-1, h)$ since:

$$(0 + (((h-1) \times (h+1) + h) \text{ div } (h+1))) \text{ mod } h = (h-1) \text{ mod } h = h-1, \text{ and} \tag{10}$$

$$(0 + ((h-1) \times (h+1) + h)) \text{ mod } (h+1) = h. \tag{11}$$

After $h^2 + h - 1$ steps through the vertical flow, s_V is also in $cell(h-1, h)$ since:

$$(0 + (h^2 + h - 1)) \text{ mod } h = h-1, \text{ and} \tag{12}$$

$$(0 + ((h^2 + h - 1) \text{ div } h)) \text{ mod } (h+1) = h \text{ mod } (h+1) = h. \tag{13}$$

□

Theorem 3.4. *The* meeting time *of cell*(*i*, *h*) *with* $0 \le i \le h - 3$ *is* $h + 2$.

Proof. After $h+2$ movements through the horizontal flow, s_H is in $cell(i+2, 0)$, since:

$$(i + ((h + h + 2) \text{ div } (h+1))) \text{ mod } h = (i+2) \text{ mod } h = i+2, \text{ and} \tag{14}$$

$$(h + (h+2)) \text{ mod } (h+1) = 0. \tag{15}$$

It should be noted that $i + 2 < h$ since $i \le h - 3$. After $h+2$ moves through the vertical flow, s_V is in $cell(i+2, 0)$, since:

$$(i + (h+2)) \text{ mod } h = i+2, \text{ and} \tag{16}$$

$$(h + ((i + h + 2) \text{ div } h)) \text{ mod } (h+1) = (h+1) \text{ mod } (h+1) = 0. \tag{17}$$

□

Theorem 3.5. *The* meeting time *of cell*(*i*, 0) *with* $2 \le i \le h - 1$ *is* $h^2 - 2$.

Proof. $h^2 - 2$ is equal to $(h-2)(h+1) + h$. After $h^2 - 2$ steps through the horizontal flow, s_H is in $cell(i-2, h)$, since:

$$(i + (((h-2) \times (h+1) + h) \text{ div } (h+1))) \text{ mod } h = (i + (h-2)) \text{ mod } h = i-2, \text{ and} \tag{18}$$

$$(0 + ((h-2) \times (h+1) + h)) \text{ mod } (h+1) = h. \tag{19}$$

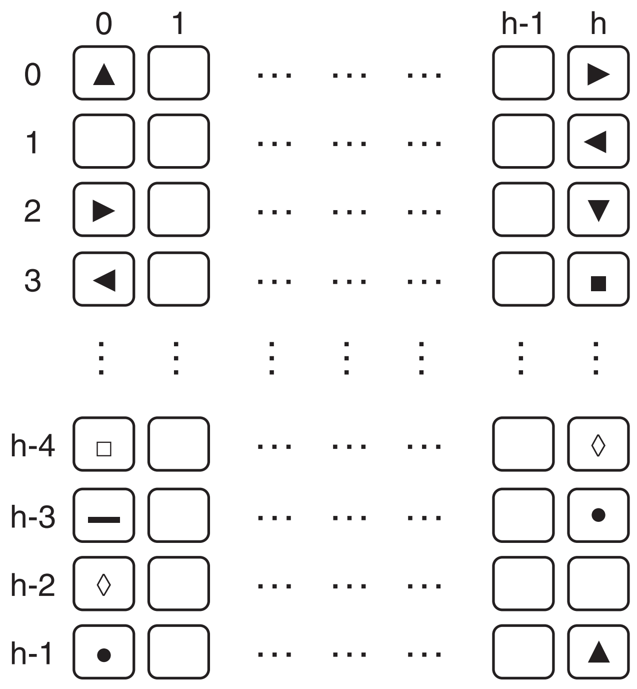


Fig. 4. Coincidence of trajectories of solutions moving horizontally and vertically on the rectangular grid.

It should be noted that $i-2 \geq 0$ since $2 \leq i$. After h^2-2 steps through the vertical flow, s_V is in $cell(i-2, h)$, since:

$$(i + (h^2-2)) \bmod h = i-2, \text{ and} \tag{20}$$

$$(0 + ((i + (h^2-2)) \operatorname{div} h)) \bmod (h+1) = h \bmod (h+1) = h. \tag{21}$$

□

Theorem 3.6. *The* meeting time *of the rest of the cells is* $h \times (h+1)$.

Proof. The rest of the cells are formed by *cell*(i, j) with $0 \leq i \leq h-1$ and $0 \leq j \leq h$ that do not belong to the other four group of cells. After $h \times (h+1)$ steps from a *cell*(i, j) through the horizontal flow, s_H is again in *cell*(i, j), since:

$$(i + ((h \times (h+1)) \operatorname{div} (h+1))) \bmod h = (i+h) \bmod h = i, \text{ and} \tag{22}$$

$$(j + (h \times (h+1))) \bmod (h+1) = j. \tag{23}$$

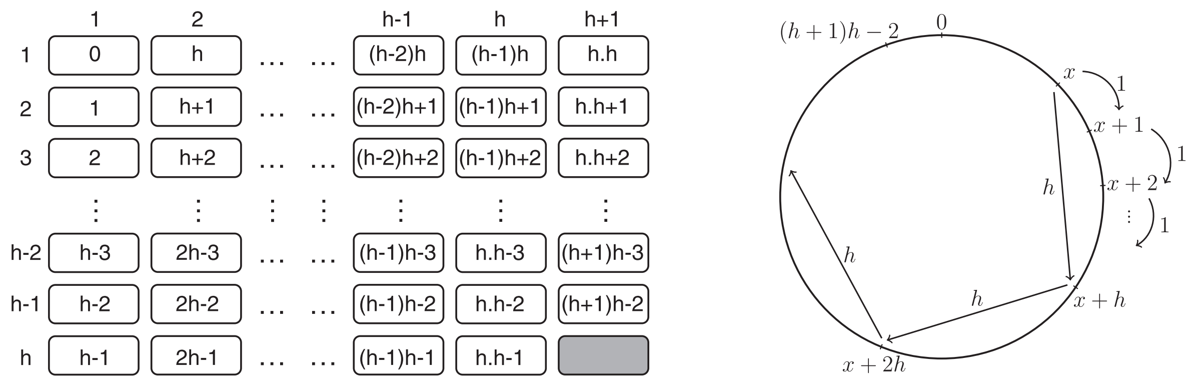
After $h \times (h+1)$ steps through the vertical flow, s_V is also again in *cell*(i, j), since:

$$(i + (h \times (h+1))) \bmod h = i, \text{ and} \tag{24}$$

$$(j + ((h \times (h+1)) \operatorname{div} h)) \bmod (h+1) = (j+h+1) \bmod (h+1) = j. \tag{25}$$

□

Fig. 4 represents the cells in which pairs of solutions moving horizontally and vertically coincide using the same symbol. Each solution moving horizontally/vertically would only interact in reproduction with a number of solutions moving vertically/horizontally that is equal to the number of cells from the fifth group ($h \times (h-1) + 2$) plus $h-2$ (the solutions from *cell*(i, h) with $0 \leq i \leq h-3$ coincide again in *cell*(i, 0) with $2 \leq i \leq h-1$), and plus one (the solutions from $cell(h-1, h)$ coincide in *cell*(0, 0)). As a consequence, the ratio of solutions from one subpopulation with which any given solution of the other subpopulation is mated is $\frac{h^2+1}{h^2+h}$. Even though this ratio is larger than the one obtained for the square grid, it is possible to design a new grid in which any solution is paired for mating with a different solution of the other subpopulation in each cell of the grid, i.e., the *meeting time* of each cell of the grid is equal to the number of cells of the grid. We present this new grid in the next subsection.



(a) The grid. (b) The idea of the proof.

Fig. 5. $h \times (h+1)$ grid with maximum *meeting time*.

3.3. *A* $\lceil\sqrt{l}\rceil \times \left(\lceil\sqrt{l}\rceil + 1\right)$ *grid with maximum meeting time*

Let us consider a $h \times (h+1)$ grid (i.e., $w = h+1$) without the $cell(h, h+1)$ (as it is shown in Fig. 5a). This grid has exactly $(h+1)h-1$ cells, and both subpopulations are composed of $(h+1)h-1$ solutions. Since we are considering a SGS$_\text{B}$ data flow, the vertically outgoing solution from $cell(h-1, h+1)$ passes to $cell(1, 1)$, while the horizontally outgoing solution from $cell(h, h)$ passes to $cell(1, 1)$. In order to prove that the *meeting time* for each cell of this grid is $(h+1)h-1$, we demonstrate an auxiliary lemma that will be used as part of the demonstration.

Lemma 3.1. *Let* $h \in \mathbb{Z}$ *and* $h \geq 0$, *it holds that* $h-1$ *and* $(h+1)h-1$ *are coprime.*

Proof. If $h-1$ and $(h+1)h-1$ were not coprime, then $h-1$ and $(h+1)h-1$ would have a prime factor in common, i.e., $\exists a$ prime such that $h-1 = a \times r_1$ and $(h+1)h-1 = a \times r_2$.

$$(h+1)h-1 = (a \times r_1 + 2) \times (a \times r_1 + 1) - 1 = a^2 \times r_1^2 + 3a \times r_1 + 1 = a(a \times r_1^2 + 3 \times r_1) + 1. \tag{26}$$

In other words, the modulo of $(h+1)h-1$ by a is 1. This is contradictory with the fact that a was a prime factor of $(h+1)h-1$. As a consequence $h-1$ and $(h+1)h-1$ do not have prime factors in common. □

Theorem 3.7. *The* meeting time *of all the cells of the grid is* $(h+1)h-1$.

Proof. Let us number the cells of the grid between 0 and $(h+1)h-2$, corresponding 0 to $cell(1, 1)$, 1 to $cell(2, 1)$, and so on, as it is shown in Fig. 5a. One step through the vertical data stream of a cell corresponds to adding one in modulo $(h+1)h-1$ to the number associated with the cell. One step through the horizontal data stream of a cell is equivalent to adding h in modulo $(h+1)h-1$ to the number associated with the cell. Two solutions located in the same cell, coincide again in a cell when the same number of additions (i.e., the number of steps) with a 1 and a h step in modulo $(h+1)h-1$ obtain the same result. The idea of the proof is illustrated in Fig. 5b.

Let x be a cell $(0 \leq x \leq (h+1)h-2)$ and p be *meeting time* of x $(1 \leq p \leq (h+1)h-1)$. Then,

$$(x + p \times h) \quad mod \quad ((h+1)h-1) = (x + 1 \times p) \quad mod \quad ((h+1)h-1)\,. \tag{27}$$

Since the remainders of the integer division have to be equal, it holds that

$$x + p \times h = \alpha \times ((h+1)h-1) + r \Rightarrow r = x + p \times h - \alpha \times ((h+1)h-1), \tag{28}$$

$$x + 1 \times p = \beta \times ((h+1)h-1) + r \Rightarrow r = x + p - \beta \times ((h+1)h-1), \tag{29}$$

with $\alpha \in \mathbb{Z}$, and $\beta \in \mathbb{Z}$. From Eq. 28 and Eq. 29, it follows that

$$p \times (h-1) = (\alpha - \beta) \times ((h+1)h-1). \tag{30}$$

Let $k = \alpha - \beta$, $k \in \mathbb{Z}$ since $\alpha \in \mathbb{Z}$ and $\beta \in \mathbb{Z}$. Eq. 30 can be rewritten as:

$$k = \frac{p \times (h-1)}{(h+1)h-1}. \tag{31}$$

p has to be divisible by $(h+1)h-1$ since $k \in \mathbb{Z}$, and $h-1$ and $(h+1)h-1$ do not have any prime factor in common by Lemma 3.1. As $1 \leq p \leq (h+1)h-1$, the only possible value for p is $(h+1)h-1$. □

As a consequence, all solutions moving through one data flow coincide in a single cell of the grid with each of the solutions moving through the other data flow. In each systolic step of SGS$_\text{B}$, the pairing of solutions between subpopulations for mating is different. In other words, each of the solutions of a subpopulation is paired with all the solutions of the other subpopulation, interacting for mating with a different solution in each cell of the grid.

Table 1
Main features of the grids.

	Square Grid	Rectangular Grid	New Grid
Cells of the grid	h^2	h^2+h	h^2+h-1
Average *meeting time*	$2h-1$	h^2+1	h^2+h-1
Mating Ratio	$\frac{2h-1}{h^2}$	$\frac{h^2+1}{h^2+h}$	1

3.4. Comparison of the different grids

Table 1 summarizes the main characteristics of the different grids analyzed. The table presents the number of cells in the grid, the average *meeting time*, and the mating ratio. The average *meeting time* is calculated as the sum of the *meeting time* of each cell in the grid divided by the total number of cells in the grid. The mating ratio is calculated as the number of different solutions from one subpopulation that can be mated with a solution from the other subpopulation (according to the *meeting time*) divided by the number of solutions of the subpopulation. Both the average *meeting time* and the mating ratio provide insight into the number of repetitions of pairings of solutions for reproduction between subpopulations when only the effect of the movements through the grid is considered. On the other hand, when the effect of the application of variational operators is taken into account, these metrics give an idea of the number of solutions paired for mating that are direct descendants of pairs of solutions that have already been mated. The square grid has the smallest average *meeting time* and mating ratio of the grids, while, on the other hand, the newly proposed grid has the largest value of these metrics. In the next section, we perform an empirical analysis of the three grids on three deceptive problems with the goal of understanding how the different features of the grids affect the behavior of SGS.

4. Experimental design

This section describes the experimental design used for evaluating the effect of the average *meeting time* and the mating ratio of the grid on SGS results including the problems used for the experimental study, the parameters setting, the execution platforms, and the experimental procedure. Then, the experimental results grouped in research questions are presented and discussed.

4.1. Test problems

For the experimental evaluation, we use three deceptive problems that are particularly hard for EAs since they converge to regions of the search space where the optimal solution cannot be found.

4.1.1. Massively multimodal deceptive problem

The Massively Multimodal Deceptive Problem (MMDP) is both deceptive and multimodal [14]. It is composed of subfunctions of six bits each one whose function value only depends on the number of ones of the six bits substring (unitation). The subfunction has two global optima located at both ends of the range and a local deceptive attractor located at the halfway point. The subfunction values are $f(0) = 1.000000$, $f(1) = 0.000000$, $f(2) = 0.360384$, $f(3) = 0.640576$, $f(4) = 0.360384$, $f(5) = 0.000000$, and $f(6) = 1.000000$. We use instances with strings of 300, 600, 900, 1200, 1500 and 1800 bits and so the optimal values are 50, 100, 150, 200, 250, and 300, respectively.

4.1.2. Six-bit fully deceptive subfunction of deb and goldberg

A six-bit fully deceptive subfunction was proposed by Deb and Goldberg [9,10] whose function value is also computed according to the unitation. It has one global optimum when unitation is six and one local deceptive attractor when unitation is zero. It was constructed assuring that all schemas containing the local attractor are superior to the ones that contain the global optimum for schemas of order up to 5. The subfunction values are $f(0) = 0.90$, $f(1) = 0.45$, $f(2) = 0.35$, $f(3) = 0.30$, $f(4) = 0.30$, $f(5) = 0.25$, and $f(6) = 1.00$. We use instances with the same size as in the MMDP, and so the optimal values are also 50, 100, 150, 200, 250, and 300. We will refer to this problem as D&G.

4.1.3. Four-bit fully deceptive subfunction of whitley

A four-bit fully deceptive subfunction was proposed by Whitley [38] following a different approach. All the four-bit strings are partitioned into subgroups according to their relative Hamming distance, i.e., each subgroup is composed of strings with the same number of 0s and 1s. An arbitrary order is chosen for each subgroup, e.g., in ascending order. Then, all the strings are sorted. The first string is the global optimum and the last string is the local deceptive attractor. Then, the function and fitness values are assigned in such a way that all order-3 and lower hyperplanes lead the search to the deceptive optimum. The subfunction values are presented in Fig. 6. We also use instances with strings of 300, 600, 900, 1200, 1500 and 1800 bits and so the optimal values are 2250, 4500, 6750, 9000, 11250, and 13500 respectively. We will refer to this problem as Whitley.

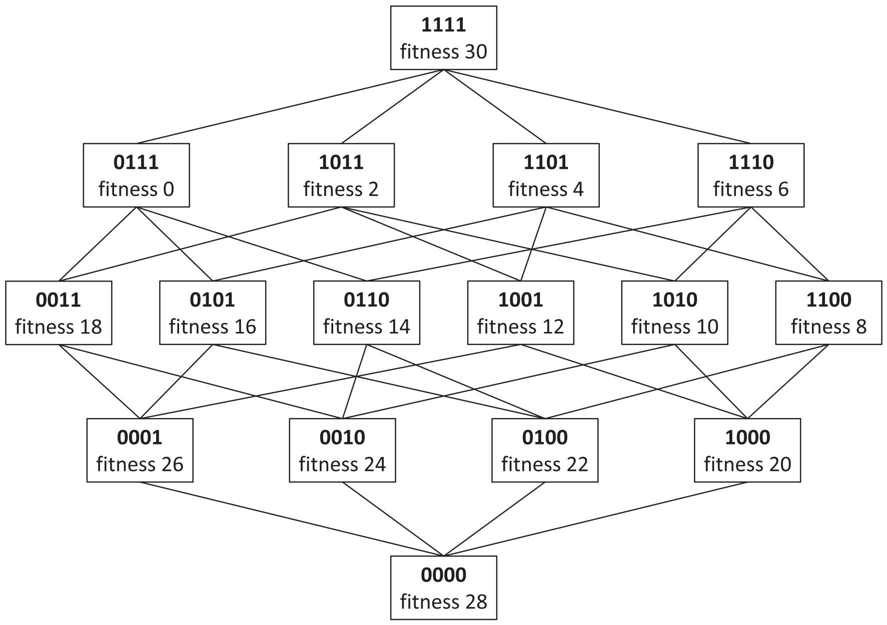


Fig. 6. Whitley's 4-bit fully deceptive subfunction.

Table 2
Parameters of the SGS algorithms for the instance sizes considered.

	300			600			900			1200			1500			1800		
	Sq	Rect	RwoC	Sq	Rect	RwoC	Sq	Rect	RwoC	Sq	Rect	RwoC	Sq	Rect	RwoC	Sq	Rect	RwoC
Number of cells	324	342	341	625	650	649	900	930	929	1225	1260	1259	1521	1560	1559	1849	1892	1891
Population size	648	684	682	1250	1300	1298	1800	1860	1858	2450	2520	2518	3042	3120	3118	3698	3784	3782
Generations	1805	1710	1715	3380	3250	3255	4805	4650	4655	6480	6300	6305	8000	7800	7805	9680	9460	9465

4.2. Algorithms, parameters setting, and test environment

In the experimental evaluation, we use three SGS algorithms. We have also included two EAs, a simple genetic algorithm with and without elitism (EGA and SGA, respectively), in order to set an actual comparison basis. These EAs have been chosen because they share the same basic search operators (crossover and mutation) as SGS so we can properly evaluate the underlying search engine of the techniques. The details of the algorithms are:

- Sq: SGS_B that uses the square grid presented in Section 3.1.
- Rect: SGS_B that uses the rectangular grid presented in Section 3.2.
- RwoC: SGS_B that uses the newly proposed rectangular grid without the $cell(l, l+1)$ presented in Section 3.3.
- SGA: It is a generational genetic algorithm with binary tournament, two-point crossover and bit-flip mutation.
- EGA: It is similar to SGA but children solutions replace parent solutions only if they have a better fitness value.

The crossover probability used for SGA and EGA is 0.9, the bit-flip mutation flips a single bit chosen randomly. Each of the SGS algorithms has a different population size that is determined by the size of the grid. The population size of both GAs was defined of the same size than Rect.

The stopping criterion used for the algorithms is to reach a maximum number of generations fixed a priori. In order to perform a fair comparison among the algorithms, the number of generations was chosen for the rectangular grid, and then the number of generations for the other algorithms was calculated guaranteeing that the number of solutions generated by every algorithm for each instance is exactly the same. The number of generations for the rectangular grid is five times the size of the grid and it was chosen to ensure that each solution circulates five times over the grid.

Table 2 presents the number of cells in the grid, the size of the population and the number of generations of the SGS algorithms for the six different instance sizes considered in the experimental study. The number of cells and the size of the population of SGS is automatically determined from the length of the tentative solutions, as it is explained in Section 2.2.

Each of the algorithms studied has only been implemented on CPU since the focus of the experimental evaluation is studying how the different grids affect the behavior of the algorithm. The execution platform for the algorithms evaluated is a PC with a Quad Core Intel i7 4770 processor at 3.40 GHz. with 16 GB RAM using a GNU/Linux O.S. All implementations have been compiled using the `-O3` flag and are run as single-thread applications.

Table 3
Percentage of the runs that obtained the optimal solution.

Problem	Instance	Hit Rate(%)				
	Size	SGA	EGA	Sq	Rect	RwoC
MMDP	300	0	0	81	**90**	89
	600	0	0	87	82	**94**
	900	0	0	78	94	**96**
	1200	0	0	85	97	**98**
	1500	0	0	88	93	**99**
	1800	0	0	89	98	**100**
D&G	300	0	0	0	47	**48**
	600	0	0	1	60	**62**
	900	0	0	1	46	**49**
	1200	0	0	1	49	**62**
	1500	0	0	3	35	**58**
	1800	0	0	3	50	**67**
Whitley	300	0	0	0	0	**1**
	600	0	0	0	0	**1**
	900	0	0	0	0	0
	1200	0	0	0	0	0
	1500	0	0	0	0	0
	1800	0	0	0	0	0

The best results are in bold.

4.3. Experimental procedure

Since the algorithms used in the evaluation are stochastic, statistical tests are used to assess the significance of the experimental results obtained. The following statistical procedure has been used [11,33]. First, one hundred independent runs for each algorithm and each instance have been performed. Then, the different metrics that are studied are computed. Two different statistical procedures are considered, one involving the statistical differences for the algorithms across the multiple problems and instances, and one analyzing the statistical difference for each problem and instance independently.

For the former approach, we use the Friedman's test according to the ranking of the algorithms for some particular metric. The test is used to check if the differences in a particular metric among the algorithms are statistically significant. Since more than two algorithms are involved in the study, a multiple $N \times N$ comparison using the Holm's post-hoc procedure is performed. These statistical tests are performed with a confidence level of 95%.

For the latter approach, the statistical procedure described next is followed [33] to determine if the distribution of a particular metric for each algorithm and each instance independently is statistically different. First, a Kolmogorov-Smirnov test and a Levene test are performed in order to check, respectively, whether the samples are distributed according to a normal distribution and whether the variances are homogeneous (homoscedasticity). If the two conditions hold, an ANOVA I test is performed; otherwise a Kruskal-Wallis test is performed. A post hoc testing phase consisting in a pairwise comparison of all the cases compared using the Holm's correction method on either the Student's *t*-test (if the samples follow a normal distribution and the variances are homogeneous) or the Wilcoxon-Mann-Whitney test (otherwise) is also performed. These tests are also performed with a confidence level of 95%.

4.4. Experimental results

In this section, we present the experimental results. We group our experiments in three research questions (RQs). RQ1 concerns the numerical efficiency of the algorithms studied in this work, especially the SGS algorithm using different grids. Then, RQ2 deals with the performance of the algorithms studied. Finally, RQ3 is concerned with understanding how highly fitted genetic material is diffused through the different grids.

4.4.1. Research question 1 - Numerical efficiency

In this subsection we address the following questions: *is SGS able to provide better solutions than the other EAs included in the study?*, and *do the grids with large average meeting time and mating ratio make SGS provide better solutions than the other grids?* To answer these questions, we consider two different metrics: the hit rate and the relative error between the best solution found by the algorithms and the optimal solution.

The hit rate is computed as the percentage of the runs of each algorithm that is able to obtain the optimal solution for each instance. Table 3 presents the hit rate of each of the algorithms for each of the 18 instances considered, while Table 4 presents the mean Friedman's ranking according to the hit rate.

The results show that RwoC is the best performing algorithm, being able to obtain more times the optimal solution in 13 out of 18 instances. RwoC is excellent for solving the MMDP and a highly competitive algorithm for solving D&G, obtaining the optimal solution in more than 50% of the trials. Although Rect is the best performing algorithm in only one instance,

Table 4
Mean Friedman's ranking.

Algorithm	Ranking
SGA	4.03
EGA	4.03
Sq	3.06
Rect	2.39
RwoC	1.50

Table 5
Statistical assessment of algorithms.

Comparison	p_{value}	Stat. assesment
SGA vs EGA	1.00e1	-
SGA vs Sq	0.33e1	-
SGA vs Rect	0.15e-1	✓
SGA vs RwoC	0.16e-4	✓
EGA vs Sq	0.33e1	-
EGA vs Rect	0.15e-1	✓
EGA vs RwoC	0.16e-4	✓
Sq vs Rect	0.41e1	-
Sq vs RwoC	0.19e-1	✓
Rect vs RwoC	0.33e1	-

the hit rates obtained by Rect are similar to the hit rates of RwoC in most of the instances studied. Sq is only a competitive algorithm for the MMDP. On the other hand, both SGA and EGA are the worst performing algorithms, and they could not find the optimal solution in any of the trials of the three considered problems. If the results obtained for each problem are globally analyzed, it can be seen that the different problems pose different challenges for SGS, being MMDP an easy problem, D&G a medium difficulty problem, and Whitley a hard problem.

Table 5 presents the p-values adjusted with Holm's procedure. These tests show that RwoC and Rect are significantly better than both SGA and EGA, and RwoC is also significantly better than Sq.

Now, we analyze the relative error to the optimal solution. For each execution of the algorithms, the relative error is computed as it is shown in Eq. 32, where f_{opt} is the optimal solution to the problem, f_{best} is the best found solution by the execution of the algorithm, and $f_{\min}$ is the worse (minimal) function value for the problem. It should be noted that $f_{\min}$ is 0 in MMDP and Whitley, but in D&G the $f_{\min}$ of each independent subfunction is 0.25.

$$e_r = \frac{|f_{opt} - f_{best}|}{|f_{opt} - f_{min}|}. \tag{32}$$

Since the three problems are composed of the concatenation of subfunctions, there are many repeated values in the relative error obtained by the different algorithms. For this reason, we have chosen to report both median and mean in order to picture a better description of the distribution of the error. Table 6 presents the experimental results regarding the relative error to the optimal solution obtained by each algorithm. The table includes the best (minimum) relative error, the median and the mean of the distribution of the relative error, as well as the interquartile range (IQR) and the standard deviation (SD).

The results show that RwoC is the algorithm with the best distribution of the relative error in 15 out 18 instances, while Rect is the best performing algorithm in the rest of the cases (the smallest instance of MMDP and the two smallest instances of D&G). Both SGA and EGA are the worst performing algorithms, achieving the worse relative error distribution in all instances considered. The results of the pairwise comparisons using the Holm's method on Wilcoxon-Mann-Whitney test are displayed in tabular form in Table 7. The statistical tests show that all SGS algorithms have a better numerical performance than SGA and EGA. RwoC and Rect have a better numerical performance than Sq in 16 out of 18 and 15 out of 18 instances, respectively. There is also statistical evidence that RwoC outperforms Rect in 9 out of 18 instances.

Taking into account the hit rates globally for all instances and the relative error for each instance, it can be concluded that RwoC consistently outperforms Sq, while there is strong statistical evidence that in most cases Rect is superior to Sq (the Kruskal-Wallis test on 15 out of 18 instances). Although RwoC is in general superior to Rect (i.e., Rect is the second best performing SGS algorithm), the statistical evidence is less strong (the Kruskal-Wallis test on 9 out of 18 instances). In summary, the numerical results show that the numerical efficiency of the SGS algorithms that use grids with large average *meeting time* and mating ratio is far superior to the square grid (the grid with the smallest values of the two metrics). The numerical results also show that all SGS algorithms have a better numerical performance than SGA and EGA.

Table 6
Numerical efficacy in terms of relative error.

Size	Alg.	MMDP					D&G					Whitley				
		Best	Median	IQR	Mean	SD	Best	Median	IQR	Mean	SD	Best	Median	IQR	Mean	SD
300	SGA	2.41e−1	2.75e−1	1.83e−2	2.73e−1	1.16e−2	2.36e−1	2.89e−1	1.87e−2	2.88e−1	1.48e−2	1.01e−1	1.16e−1	5.33e−3	1.15e−1	4.24e−3
	EGA	1.75e−1	2.31e−1	3.03e−3	2.31e−1	2.12e−2	1.17e−1	1.71e−1	4.57e−2	1.73e−1	3.31e−2	5.87e−2	7.91e−2	1.16e−2	7.97e−2	9.11e−3
	Sq	0.00e0	0.00e0	0.00e0	1.37e−3	2.83e−3	2.67e−3	1.07e−2	5.33e−3	1.07e−2	3.73e−3	1.78e−3	3.56e−3	8.89e−4	3.97e−3	9.58e−4
	Rect	**0.00e0**	**0.00e0**	**0.00e0**	**7.19e−4**	**2.17e−3**	**0.00e0**	**2.67e−3**	**2.67e−3**	**1.79e−3**	**2.04e−3**	8.89e−4	1.78e−3	8.89e−4	2.12e−3	8.27e−4
	RwoC	0.00e0	0.00e0	0.00e0	7.91e−4	2.26e−3	0.00e0	2.67e−3	2.67e−3	1.84e−3	2.10e−3	**0.00e0**	**1.78e−3**	**8.89e−4**	**1.77e−3**	**7.74e−4**
600	SGA	1.77e−1	2.04e−1	8.06e−3	2.09e−1	2.22e−1	1.81e−1	2.07e−1	9.50e−3	2.07e−1	8.07e−3	8.62e−2	9.60e−2	3.56e−3	9.55e−2	2.71e−3
	EGA	1.86e−1	2.18e−1	1.64e−2	2.19e−1	1.28e−2	1.05e−1	1.56e−1	3.07e−2	1.55e−1	2.27e−2	6.04e−2	7.31e−2	8.00e−3	7.30e−2	5.48e−3
	Sq	0.00e0	0.00e0	0.00e0	4.67e−4	1.21e−3	0.00e0	6.67e−3	2.67e−3	6.47e−3	2.26e−3	2.22e−3	3.33e−3	8.89e−4	3.40e−3	6.26e−4
	Rect	0.00e0	0.00e0	0.00e0	6.47e−4	1.39e−3	**0.00e0**	**0.00e0**	**1.33e−3**	**6.27e−4**	**8.57e−4**	4.44e−4	1.33e−3	8.89e−4	1.45e−3	5.95e−4
	RwoC	**0.00e0**	**0.00e0**	**0.00e0**	**2.16e−4**	**8.58e−4**	0.00e0	0.00e0	1.33e−3	6.80e−4	1.01e−3	**0.00e0**	**1.33e−3**	**4.44e−4**	**1.23e−3**	**4.93e−4**
900	SGA	1.28e−1	1.51e−1	7.53e−3	1.50e−1	6.14e−3	1.40e−1	1.67e−1	8.00e−3	1.66e−1	7.04e−3	7.56e−2	8.39e−2	2.52e−3	8.38e−2	1.99e−3
	EGA	1.87e−1	2.16e−1	1.55e−2	2.17e−1	1.24e−2	1.11e−1	1.54e−1	2.59e−2	1.53e−1	1.66e−2	5.96e−2	6.92e−2	4.52e−3	6.99e−2	4.00e−3
	Sq	0.00e0	0.00e0	0.00e0	5.27e−4	9.98e−4	0.00e0	4.44e−3	2.67e−3	4.56e−3	1.74e−3	2.37e−3	3.26e−3	5.93e−4	3.32e−3	4.89e−4
	Rect	0.00e0	0.00e0	0.00e0	1.44e−4	5.72e−4	0.00e0	8.89e−4	8.89e−4	6.49e−4	7.24e−4	2.96e−4	1.19e−3	5.93e−4	1.14e−3	3.75e−4
	RwoC	**0.00e0**	**0.00e0**	**0.00e0**	**9.58e−5**	**4.72e−4**	**0.00e0**	**8.89e−4**	**8.89e−4**	**6.04e−4**	**6.67e−4**	**2.96e−4**	**8.89e−4**	**2.96e−4**	**1.05e−3**	**3.78e−4**
1200	SGA	9.60e−2	1.13e−1	7.39e−3	1.12e−1	5.72e−3	1.31e−1	1.44e−1	8.50e−3	1.44e−1	5.76e−3	7.13e−2	7.56e−2	2.00e−3	7.55e−2	1.47e−3
	EGA	1.86e−1	2.16e−1	1.62e−2	2.16e−1	1.08e−2	1.10e−1	1.53e−1	2.45e−2	1.53e−1	1.71e−2	6.02e−2	6.69e−2	4.67e−3	6.74e−2	3.05e−3
	Sq	0.00e0	0.00e0	0.00e0	2.70e−4	6.45e−4	0.00e0	2.67e−3	1.33e−3	2.87e−3	1.26e−3	2.00e−3	2.89e−3	4.44e−4	2.85e−3	3.62e−4
	Rect	0.00e0	0.00e0	0.00e0	5.39e−5	3.08e−4	0.00e0	6.67e−4	6.67e−4	5.00e−4	5.64e−4	2.22e−4	8.89e−4	4.44e−4	9.24e−4	3.50e−4
	RwoC	**0.00e0**	**0.00e0**	**0.00e0**	**3.59e−5**	**2.53e−4**	**0.00e0**	**0.00e0**	**6.67e−4**	**3.07e−4**	**4.39e−4**	**2.22e−4**	**6.67e−4**	**2.22e−4**	**7.62e−4**	**2.33e−4**
1500	SGA	7.57e−2	9.28e−2	6.12e−3	9.20e−2	4.81e−3	1.17e−1	1.30e−1	6.53e−3	1.29e−1	4.40e−3	6.61e−2	6.99e−2	1.78e−3	6.95e−2	1.38e−3
	EGA	1.92e−1	2.15e−1	1.53e−2	2.14e−1	9.90e−3	1.21e−1	1.54e−1	2.21e−2	1.53e−1	1.45e−2	5.92e−2	6.59e−2	3.82e−3	6.58e−2	3.02e−3
	Sq	0.00e0	0.00e0	0.00e0	1.73e−4	4.70e−4	2.86e−3	2.13e−3	1.60e−3	2.06e−3	1.11e−3	1.78e−3	2.67e−3	3.56e−4	2.63e−3	3.32e−4
	Rect	0.00e0	0.00e0	0.00e0	1.01e−4	3.69e−4	0.00e0	5.33e−4	5.33e−4	4.48e−4	3.84e−4	1.78e−4	8.89e−4	3.56e−4	8.50e−4	2.84e−4
	RwoC	**0.00e0**	**0.00e0**	**0.00e0**	**1.44e−5**	**1.44e−4**	**0.00e0**	**0.00e0**	**5.33e−4**	**2.72e−4**	**3.60e−4**	**1.78e−4**	**5.33e−4**	**5.33e−4**	**6.54e−4**	**3.01e−4**
1800	SGA	6.00e−2	7.77e−2	5.10e−3	7.72e−2	4.33e−3	1.06e−1	1.18e−1	5.17e−3	1.18e−1	4.21e−3	5.90e−2	6.52e−2	1.63e−3	6.51e−2	1.33e−3
	EGA	1.90e−1	2.17e−1	1.09e−2	2.17e−1	8.43e−3	1.10e−1	1.50e−1	2.38e−2	1.51e−1	1.84e−2	5.88e−2	6.47e−2	3.41e−3	6.49e−2	2.58e−3
	Sq	0.00e0	0.00e0	0.00e0	1.32e−4	3.77e−4	0.00e0	1.78e−3	1.00e−3	1.63e−3	8.07e−4	1.78e−3	2.52e−3	4.44e−4	2.47e−3	2.96e−4
	Rect	0.00e0	0.00e0	0.00e0	2.40e−5	1.69e−4	0.00e0	2.22e−4	8.89e−4	3.69e−4	4.47e−4	2.96e−4	7.41e−4	2.96e−4	7.60e−4	2.71e−4
	RwoC	**0.00e0**	**0.00e0**	**0.00e0**	**0.00e0**	**0.00e0**	**0.00e0**	**0.00e0**	**4.44e−4**	**1.91e−4**	**3.30e−4**	**1.48e−4**	**4.44e−4**	**1.48e−4**	**5.29e−4**	**2.06e−4**

The best distribution of relative error is in bold.

Table 7
Statistical assessment for instances 300, 600, 900, 1200, 1500, 1800.

Comparison	MMDP						D&G						Whitley					
SGA vs EGA	▷	◁	◁	◁	◁	◁	▷	▷	▷	◁	◁	◁	▷	▷	▷	▷	▷	–
SGA vs Sq	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷
SGA vs Rect	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷
SGA vs RwoC	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷
EGA vs Sq	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷
EGA vs Rect	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷
EGA vs RwoC	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷
Sq vs Rect	-	-	▷	▷	-	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷
Sq vs RwoC	-	-	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷	▷
Rect vs RwoC	-	▷	-	-	-	-	-	-	-	▷	▷	▷	▷	▷	-	▷	▷	▷

'◁' states that the first algorithm is statistically better than the second one.
'▷' states that the second algorithm is statistically better than the first one.
'-' states that no statistically significant differences are found.

Table 8
Median and interquartile range of the runtime of the algorithms in seconds.

Problem	Instance	SGA		EGA		Sq		Rect		RwoC	
		Median	IQR	Median	IQR	Median	IQR	Median	IQR	Median	IQR
MMDP	300	0.524	0.003	0.547	0.004	0.480	0.003	**0.471**	**0.002**	0.474	0.001
	600	3.307	0.019	3.668	0.017	3.183	0.012	**3.112**	**0.005**	3.131	0.007
	900	9.610	0.053	11.039	0.040	9.374	0.040	**9.186**	**0.028**	9.216	0.038
	1200	22.376	0.120	26.432	0.109	22.399	0.100	**21.954**	**0.061**	22.073	0.052
	1500	43.822	0.222	51.897	0.233	44.253	0.254	**43.185**	**0.060**	43.388	0.091
	1800	78.068	0.496	92.785	0.377	78.472	0.175	**77.531**	**0.073**	77.752	0.089
D&G	300	0.667	0.005	0.672	0.006	0.441	0.007	**0.410**	**0.009**	0.420	0.010
	600	4.009	0.039	4.786	0.024	2.702	0.045	**2.469**	**0.031**	2.503	0.032
	900	11.148	0.132	14.864	0.080	7.664	0.110	**6.989**	**0.086**	7.050	0.061
	1200	25.235	0.247	36.536	0.165	17.638	0.284	**16.081**	**0.149**	16.168	0.139
	1500	47.699	0.537	72.069	0.461	34.221	0.392	**31.341**	**0.255**	31.547	0.286
	1800	83.152	0.742	129.430	0.480	59.451	0.354	**55.612**	**0.285**	55.936	0.229
Whitley	300	0.770	0.025	0.888	0.013	0.398	0.006	**0.379**	**0.007**	0.385	0.005
	600	4.813	0.143	6.581	0.063	2.391	0.036	**2.182**	**0.029**	2.214	0.034
	900	13.706	0.366	20.651	0.116	6.699	0.087	**5.984**	**0.064**	6.079	0.056
	1200	31.020	0.701	51.362	0.353	15.273	0.147	**13.484**	**0.136**	13.619	0.157
	1500	57.961	1.237	101.454	0.725	29.970	0.315	**26.302**	**0.298**	26.584	0.251
	1800	99.398	1.398	182.018	0.805	52.140	0.291	**46.509**	**0.291**	47.005	0.228

The best results are in bold.

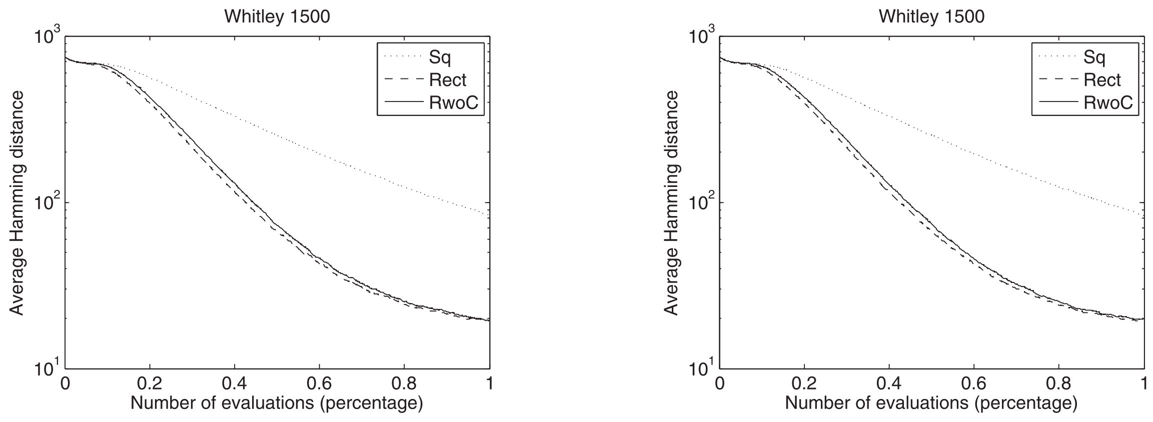
4.4.2. Research question 2 - Computational efficiency

In this subsection we address the following questions: *are SGS algorithms more efficient than the other EAs included in the study?*, and *is the execution time of SGS affected by the use of a grid with large average meeting time and mating ratio?*. To address this question, we analyze the performance of the algorithms using the wall-clock time of execution.

Table 8 shows the median and the interquartile range of the runtime of all the algorithms computed for the one hundred executions for each problem and instance. The Kruskal-Wallis test showed that the results are significant with a confidence level of 95%. The pairwise comparison of all the combinations of pairs of algorithms for each instance, using the Holm's post-hoc method on the Wilcoxon-Mann-Whitney test with the same level of significance, revealed in all the cases that the differences in the median runtime of the algorithms are statistically significant.

In the 18 instances, Rect is the algorithm with the shortest runtime, while RwoC is consistently the second best performing algorithm for all the problems and instances considered, i.e., the SGS algorithms with grids with large average *meeting time* and mating ratio have the shortest execution time. On the other hand, the other EAs have in general a longer execution time than the SGS algorithms, especially in the D&G and Whitley problems. The differences in runtime between Rect and RwoC range between 0.29% and 2.44%, being less than 1% in 10 instances and more than 2% in only one case (D&G with size 300), which means a modest increase in runtime with respect to Rect. The differences in runtime between Rect and Sq are about 2% for the instance of MMDP, while for the other two problems the differences grow to the range of 5.00%-13.95%. The differences between SGS algorithms and the other EAs are larger, and they can be up to 140% in D&G and almost 300% in Whitley.

The main reason for such differences in the runtime of SGS algorithms and the other EAs is the different underlying search engine in which the algorithms are based. On the other hand, the different SGS algorithms use the same underlying search engine and have significant differences in the execution time. It should be noted that although the number of solutions built by each SGS algorithm is the same, the number of generations of each algorithm is not the same due to the



(a) Best horizontal solution vs. vertical subpopulation. (b) Best vertical solution vs. horizontal subpopulation.

Fig. 7. Evolution of the average Hamming distance.

different number of cells of each different grid. The differences in the runtime of the SGS algorithms are partially produced by the overhead of handling the population for some additional iterations (e.g., the population is read from a memory space, while new solutions are stored in a different memory space, allowing concurrent access to the data because of the disjoint storage, etc.). As a matter of fact, the SGS variants that involved a larger number of generations are the ones that required a higher execution time.

4.4.3. Research question 3 - Diffusion of highly fitted genetic material in SGS

In this subsection, we address the following question: *does the use of a grid with large average meeting time and mating ratio really helps the SGS algorithm to diffuse highly fitted genetic material along the grid?*. Since our goal is to understand how highly fitted genetic material is spread between the subpopulations, we begin this study by analyzing how the similarity between the best solution of one subpopulation and the other subpopulation varies during the execution of SGS. For this, in each iteration of the algorithm, we compute the average Hamming distance between the best solution of each of the subpopulations and all the individuals of the other subpopulation. The Hamming distance measures the number of bits that are different in two individuals. Then, the average over the 100 independent executions is calculated for each instance and each SGS algorithm. Since this metric presents almost identical results for each of the problems and instances considered, we only include the results for the instance Whitley 1500. Fig. 7a and Fig. 7b show the evolution of the average Hamming distance for this instance.

In Rect and RwoC, grids with large average *meeting time* and mating ratio, the similarity between the best solution moving in one direction and the other subpopulation is greater than in Sq. A reasonable hypothesis is that this behavior is related to the difference in the number of different solutions from a subpopulation that can interact with the best solution of the other subpopulation in each of the grids. To corroborate this hypothesis, we study how are the differences in the Hamming distance between the best solution and the other subpopulation distributed along the grid.

Fig. 8 presents heat maps that graphically represent the Hamming distance between the best solution of the horizontal subpopulation and each of the solution of the vertical subpopulation for a representative run of Sq, Rect and RwoC for Whitley 1500 at the initial population, and at 25%, 50%, 75% and 100% of the total evaluations of the execution. The darker colors indicate solutions that are more similar. The X or Xs (in black in the initial population and in white in the rest) indicate in which cell is located the best solution of the horizontal population. The black cell at the lower right of the RwoC grid indicates the missing cell.

In Sq, it is clear that the solutions from the vertical subpopulation that are more similar to the best horizontal solution are located in a diagonal band (and its continuation due to the circularity of the grid). In the other two grids, this pattern also appears when the number of solutions evaluated is small even though the diagonal band is much wider and it has lower intensity. This pattern is gradually lost and it has almost disappeared (the differences become more homogeneous among the entire population) when the number of solution evaluated is 50%. These results suggest that in Sq the genetic material of the best found solution of one subpopulation is confined in a region of the grid and it does not circulate among all the solutions of the other subpopulation. This behavior does not occur in the grids that encourage interactions between solutions and their descendants that have not directly interacted in the grid before.

In order to analyze that this is indeed the reason that causes the previous pattern, we study in which cells are located the vertical solutions that are descendants of the interaction of a vertical solution with an ancestor of the best horizontal solution. That is, in which cell is located the descendant of the vertical solution with which an ancestor of the best horizontal solution interacted one step before, two steps before, and so on.

Fig. 9 presents heat maps that graphically represent where are located vertical solution with an ancestor in common with the best horizontal solution for a representative run of Sq, Rect and RwoC for Whitley 1500 at 25% and 50% of the total number of function evaluations (one of the best horizontal solutions is presented when there are more than one). The

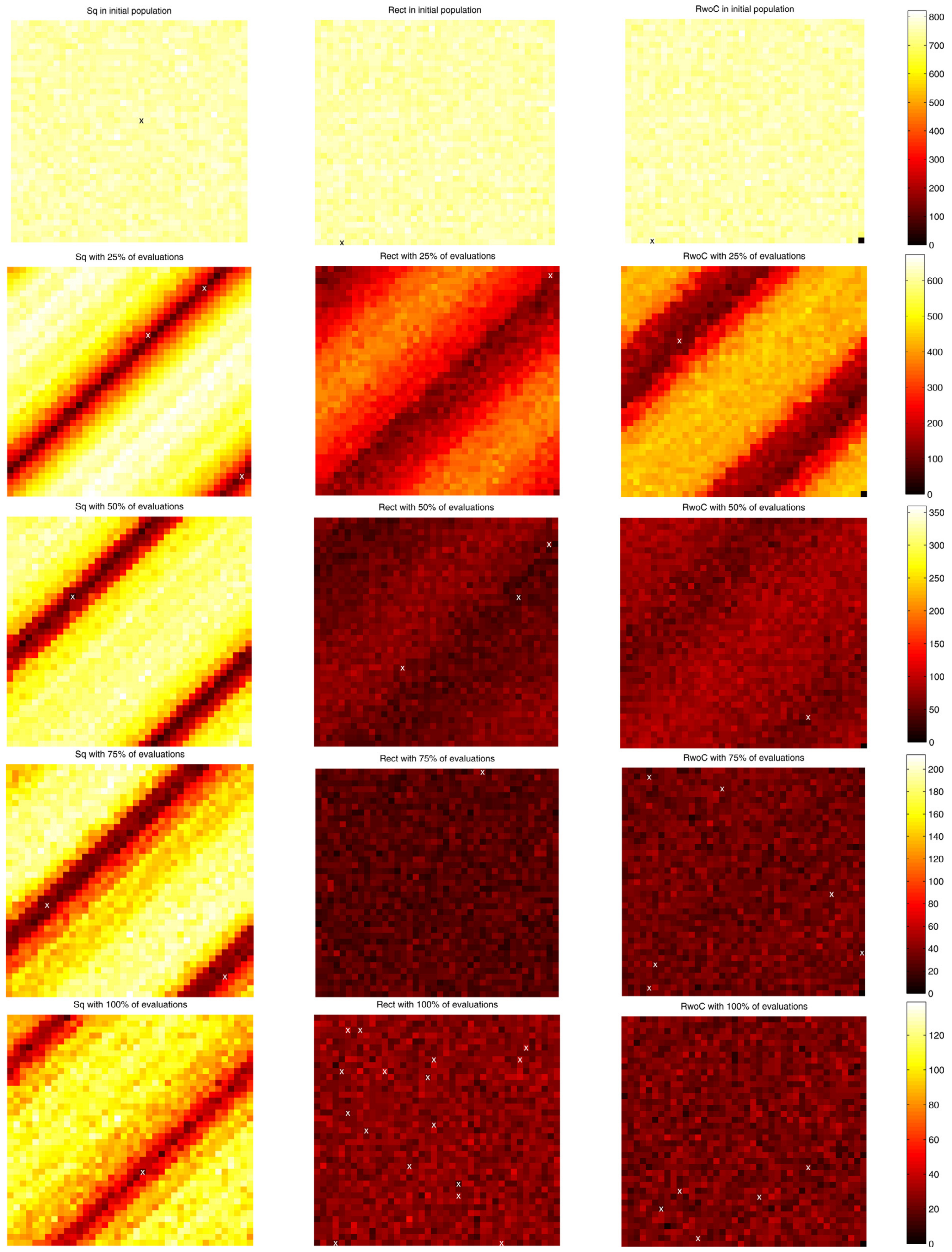


Fig. 8. Hamming distance between best horizontal solution and vertical subpopulation (darker means that solutions are more similar).

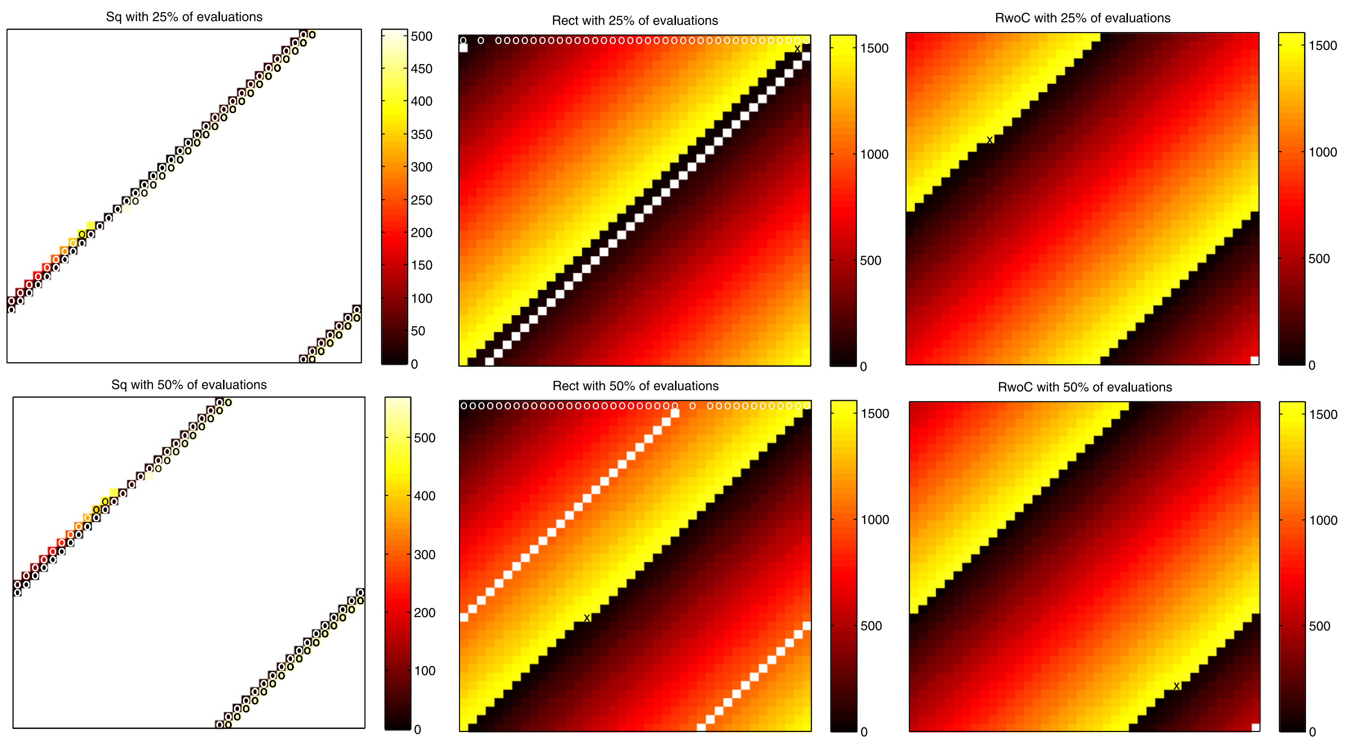


Fig. 9. Number of steps back of the interaction between ancestor of the best horizontal solution and the vertical solution.

cell color indicates the number of steps back that the interaction between the ancestor of the best horizontal solution and the ancestor of the vertical solution has occurred, darker colors indicate that the interaction is more recent. The white cells indicate that the vertical solution in that cell does not have an ancestor in common with the best horizontal solution. The zeros in white indicate that the cell has more than one ancestor in common.

It is clear that in Sq the solutions from the vertical subpopulation that are similar to the best horizontal solution (Fig. 8) are strongly correlated with the solution from the vertical subpopulation with an ancestor in common with the best horizontal solution (Fig. 9). It should be noted that the best horizontal solution only has ancestors in common with a minimal percentage of the vertical subpopulation, which corroborates the theoretical analysis presented in Section 3.1. As a consequence, the genetic material of highly fitted solutions from one subpopulation is barely shared with the solutions of the other subpopulation.

On the other hand, in the other two grids the best horizontal solution has common ancestors with most (or all depending on the grid) of the solutions of the vertical subpopulation, which corroborates that highly fitted genetic material of the best solutions of one of the subpopulations can be shared with a large part of the solutions of the other subpopulation (as it was proven in Section 3.2 and in Section 3.3). This produces that all solutions of the vertical subpopulation are more similar to the best horizontal solution than in Sq.

To complete this study, we analyze how similar are the solutions of both subpopulations. For this purpose, we use two different metrics the population-diversity and the cell-diversity. The population-diversity is computed as the average Hamming distance between solution pairs of the whole population and it allows to understand how the overall diversity of the population is evolving. This metric can be computed efficiently using the approach proposed in [25]. On the other hand, the cell-diversity is computed as the average Hamming distance between all the pair of solutions that are coinciding in a cell in the iteration step, and it provides insight in the diversity of the solutions that are actually being mated by the algorithms.

Fig. 10 presents the average population-diversity and cell-diversity over the 100 independent runs during the execution of Sq, Rect and RwoC for Whitley 1500. The population-diversity decreases during the execution of the SGS algorithms since SGS does not include any explicit mechanism for preserving the diversity. The decrease rate is faster for Rect and RwoC than for Sq, which is consistent with the fact that genetic material is confined in regions of the grid and it does not circulate among the entire subpopulation. The opposite happens with the cell-diversity, the loss on the cell-diversity of Sq is more pronounced than for the grids with large average *meeting time* and mating ratio. Eventually, the curves of population-diversity and cell-diversity of Rect and RwoC converge to similar values after 60% of the total number of solutions are evaluated. This effect can be expected since the solutions interact with all (or most of the) solutions of the other subpopulation in both grids. As a consequence, and due to the effect of the mating of solutions of the two different subpopulations over the iterations of the algorithm, the degree of difference between individuals of each cell tends to be similar to the degree of difference between any pair of individuals of the whole population. On the other hand, the curves of population-diversity and cell-diversity of Sq do not converge to similar values. This result can be explained because solutions in Sq interact with

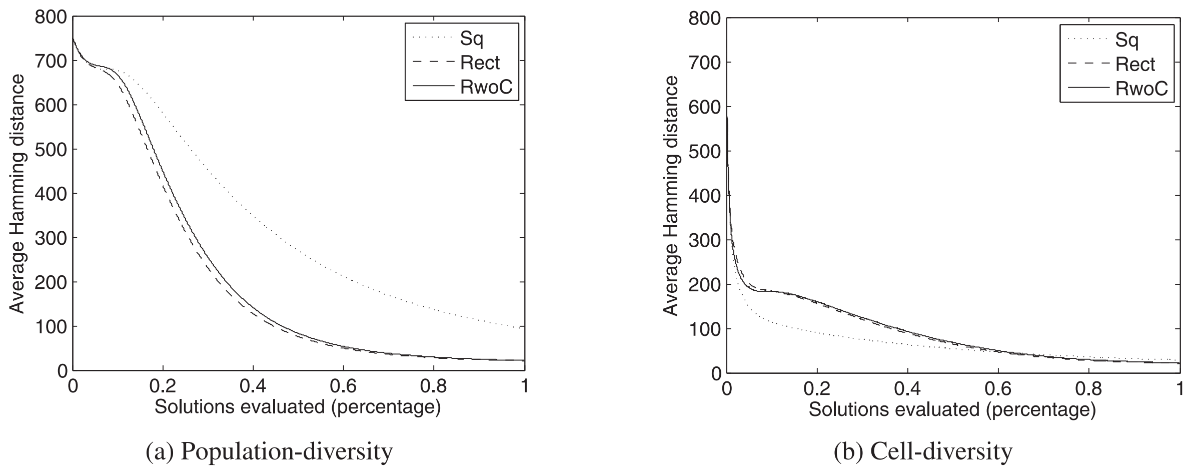


Fig. 10. Evolution of the diversity during the execution of SGS algorithms on Whitley 1500.

few solutions of the other subpopulation. For this reason, and induced by the flows on the grid, clusters of solutions emerge, with solutions that are relatively similar and that are interacting through mating (as it was shown in Fig. 9). In contrast, solutions from different clusters are more diverse but solutions from different clusters have almost no interactions.

These results suggest that SGS with grids with large average *meeting time* and mating ratio, like Rect and RwoC, due to the relatively high diversity in each cell, makes a better use of the genetic material of the highly fitted solutions in the first iterations. This allows the algorithm to identify good regions of the search space that are close to the optimal solution. Instead, Sq presents a much lower cell-diversity in the first iterations and thus makes a poorer use of genetic material of the solutions that limit the algorithm to move in worst regions of the search space. These findings on how the highly fitted genetic material is diffused through the grid are completely consistent with the numerical efficiency of Sq (the worst performing SGS algorithm), Rect (the second best performing SGS algorithm) and RwoC (the best performing SGS algorithm) reported in Section 4.4.1.

5. Conclusions and future work

In this article, we have characterized the population model of SGS_B data flow of the recently proposed optimization algorithm Systolic Genetic Search. This population model is based in the interactions of two individuals that belong to two different subpopulations, the solutions that are moving horizontally and the solutions that are moving vertically. Then, we have theoretically analyzed the trajectories described by the solutions in two different grids that have already been used experimentally. We formalized our analysis through the study of the *meeting time* of the cells. The study measures the minimum number of steps that it takes two solutions to coincide again in a cell (under the assumption of only considering the effect of the movements through the grid). The main conclusions of the analysis are that the number of different pairing of solutions for mating between the subpopulations is quite small in Sq, while, Rect produces a large number of different pairing of solutions between the subpopulations even though several pairs are repeated along the grid. In view of this fact, we have engineered and proposed a new variant of SGS named RwoC with a grid that guarantees that each of the solutions of a subpopulation is paired for mating with all the solutions of the other subpopulation.

An experimental evaluation was conducted using the three different grids of SGS and two EAs for solving three deceptive problems on six instances of increasing size. The most important findings of this experimental evaluation can be summarized as follows. First, the numerical results show that all SGS algorithms systematically outperform SGA and EGA in all the instances considered, which confirms the efficacy of the underlying search engine of SGS. Secondly, the use of grids with large average *meeting time* and mating ratio (i.e., grids that limit the mating of descendants of pairs of solutions that have already been mated), like in Rect and in the newly proposed RwoC, has proven to benefit the search engine of SGS. In particular, the novel variant of SGS is the best performing algorithm, having the best Friedman's ranking according to the hit rate and outperforming Sq in 16 out of 18 instances considered and Rect in 9 out of 18 instances, while Rect has the second best Friedman's ranking and outperforms Sq on 15 out of 18 instances. In the third place, the analysis of the computational performance of the algorithms reveals that the use of grids with large average *meeting time* and mating ratio in SGS does not involve any overhead in the execution time of the algorithm. Finally, we investigated the causes of the best numerical performance of the grids with large average *meeting time* and mating ratio. Their success is explained by a better diffusion of highly fitted genetic material through the subpopulations, which makes SGS achieve higher diversity on the cells of the grid (i.e., the degree of difference of the individuals that are actually being mated in each cell). As a consequence, SGS makes a more suitable use of the information of the search space gathered in the solutions, leading to identify better regions of the search space.

In conclusion, the newly proposed RwoC is not only able to outperform the rest of the algorithms studied, but it also has better theoretical properties that explain the success of this variant of SGS. Moreover, the grid used by RwoC does not increase notably the execution time of SGS. These aspects show the great potential of the proposed variant and contribute to the consolidation of the SGS algorithm.

The main areas that deserve further study are identified next. The first issue is to address the study of other aspects of the SGS algorithms design from theoretical and empirical perspectives. One of such aspects is the use of elitism in the cell of the grid. It is interesting to understand how often elitism is used in each cell. If there is an overuse of elitism, i.e., many new solutions are discarded, a memory mechanism could be incorporated into each cell so the best solutions generated by the cell are still available and the elitism could be reduced or eliminated. This could lead to the incorporation of an explicit selection process in SGS. Another aspect that can also be studied is the use of preprogrammed fixed points (as it was originally conceived SGS) versus random points for the crossover and mutation operators. The second line of interest is to study how to incorporate to the SGS algorithm mechanism to detect the loss of cell-diversity and exploit this information in benefit of the algorithm, e.g., generating completely new solutions for increasing the diversity. The use of this type of mechanism could potentially help to speed up the search of the SGS and even lead to better solutions. The third issue is to extend properties of the grid proposed in this work to the cellular model EAs since in both models the solutions are placed on a structured grid. For instance, a new neighborhood overlapping mechanism could be designed in which neighborhood solutions are shared less frequently, e.g., updating solutions from a neighborhood depending on whether the number of generation is even or odd. Another line of interest is to perform a wider impact analysis by solving problems with different levels of ruggedness in the landscape, like in NK-landscapes. This could help to understand to what extent the proposed grid allows to improve SGS with respect to the existing grids. Finally, since Whitley has represented a hard problem for SGS, it is also interesting to make a deeper analysis on SGS algorithm using this problem for benchmarking.

Acknowledgements

M. Pedemonte acknowledges support from PEDECIBA, SNI and ANII, Uruguay. M. Pedemonte and F. Luna are partially by the Spanish MINECO and FEDER project TIN2016-75097-P. E. Alba is partially funded also by the Spanish MINECO and FEDER project TIN2014-57341-R (http://moveon.lcc.uma.es). F. Luna also acknowledges support from Plan Propio de Investigación of Universidad de Málaga, Campus de Excelencia Internacional Andalucía Tech, under grant PPIT.UMA.B1.2017/15.

The authors would like to thank Prof. Eduardo Fernández for his valuable comments and suggestions in the design of the new grid.

References

[1] E. Alba (Ed.), Parallel Metaheuristics: A New Class of Algorithms, Wiley, 2005.
[2] E. Alba, B. Dorronsorso (Eds.), Cellular Genetic Algorithms, Springer, 2008.
[3] E. Alba, M. Tomassini, Parallelism and evolutionary algorithms, IEEE Trans. Evol. Comput. 6 (5) (2002) 443–462.
[4] E. Alba, P. Vidal, Systolic optimization on GPU platforms, in: 13th International Conference on Computer Aided Systems Theory (EUROCAST 2011), 2011.
[5] E. Cantu-Paz, Efficient and Accurate Parallel Genetic Algorithms, Kluwer Academic Publishers, 2000.
[6] J.M. Cecilia, J.M. García, A. Nisbet, M. Amos, M. Ujaldón, Enhancing data parallelism for ant colony optimization on GPUs, J. Parallel Distrib. Comput. 73 (1) (2013) 42–51. https://doi.org/10.1016/j.jpdc.2012.01.002.
[7] J.M. Cecilia, J.M. García, M. Ujaldón, A. Nisbet, M. Amos, Parallelization strategies for ant colony optimisation on GPUs, in: 25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Workshop Proceedings, 2011, pp. 339–346.
[8] H. Chan, P. Mazumder, A systolic architecture for high speed hypergraph partitioning using a genetic algorithm, in: Progress in Evolutionary Computation, in: Lecture Notes in Computer Science, 956, Springer Berlin / Heidelberg, 1995, pp. 109–126.
[9] K. Deb, D.E. Goldberg, Analyzing Deception in Trap Functions, in: Foundations of Genetic Algorithms, 1992, pp. 93–108.
[10] K. Deb, D.E. Goldberg, Sufficient conditions for deceptive and easy binary functions, Ann. Math. Artif. Intell. 10 (4) (1994) 385–408, doi:10.1007/BF01531277.
[11] J. Derrac, S. García, D. Molina, F. Herrera, A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms, Swarm. Evol. Comput. 1 (1) (2011) 3–18.
[12] M. Giacobini, M. Tomassini, A.G.B. Tettamanzi, E. Alba, Selection intensity in cellular evolutionary algorithms for regular lattices, IEEE Trans. Evol. Comput. 9 (5) (2005) 489–505.
[13] D.E. Goldberg, K. Deb, A comparative analysis of selection schemes used in genetic algorithms, in: Foundations of Genetic Algorithms, Morgan Kaufmann, 1991, pp. 69–93.
[14] D. Goldberg, K. Deb, J. Horn, Massively multimodality, deception and genetic algorithms, in: Proceedings of the International Conference on Parallel Problem Solving from Nature II (PPSNII), 1992, pp. 37–46.
[15] L. Guo, C. Guo, D. Thomas, W. Luk, Pipelined genetic propagation, in: Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on, 2015, pp. 103–110.
[16] A.C. Guyton, J.E. Hall, Textbook of Medical Physiology, 11, Elsevier Saunders, 2006.
[17] P. Krömer, J. Platoš, V. Snášel, Nature-inspired meta-heuristics on modern GPUs: state of the art and brief survey of selected algorithms, Int. J. Parallel Program. 42 (5) (2014) 681–709.
[18] H.T. Kung, Why systolic architectures? Computer (Long Beach Calif) 15 (1) (1982) 37–46.
[19] H.T. Kung, C.E. Leiserson, Systolic arrays (for VLSI), in: Sparse Matrix Proceedings, 1978, pp. 256–282.
[20] W.B. Langdon, Graphics processing units and genetic programming: an overview, Soft Comput. 15 (8) (2011) 1657–1669.
[21] P. Libby, R. Bonow, D. Mann, D. Zipes, Braunwald'S heart disease: A Textbook of cardiovascular medicine, Elsevier Health Sciences, 2007.
[22] G. Luque, E. Alba, Parallel genetic algorithms: Theory and real world applications, Studies in Computational Intelligence, 367, Springer, 2011.
[23] O. Maitre, F. Krüger, S. Querry, N. Lachiche, P. Collet, EASEA: Specification and execution of evolutionary algorithms on GPGPU, Soft Comput. 16 (2) (2012) 261–279.
[24] G. Megson, I. Bland, Synthesis of a systolic array genetic algorithm, in: Parallel Processing Symposium, 1998. IPPS/SPDP 1998, 1998, pp. 316–320.
[25] R.W. Morrison, K.A. De Jong, Measurement of Population Diversity, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 31–41.

[26] M. Pedemonte, F. Luna, E. Alba, New Ideas in Parallel Metaheuristics on GPU: Systolic Genetic Search, in: S. Tsutsui, P. Collet (Eds.), Massively Parallel Evolutionary Computation on GPGPUs, Springer, 2013, pp. 203–225.
[27] M. Pedemonte, E. Alba, F. Luna, Towards the design of systolic genetic search, in: IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IEEE Computer Society, 2012, pp. 1778–1786.
[28] M. Pedemonte, F. Luna, E. Alba, Systolic genetic search for software engineering: The test suite minimization case, in: A.I. Esparcia-Alcázar, A.M. Mora (Eds.), Applications of Evolutionary Computation - 17th European Conference, EvoApplications 2014, Granada, Spain, April 23–25, 2014, Revised Selected Papers, Lecture Notes in Computer Science, 8602, Springer, 2014, pp. 678–689.
[29] M. Pedemonte, F. Luna, E. Alba, Systolic genetic search, a systolic computing-based metaheuristic, Soft Comput. 19 (7) (2015) 1779–1801.
[30] M. Pedemonte, F. Luna, E. Alba, A systolic genetic search for reducing the execution cost of regression testing, Appl. Soft. Comput. 49 (2016) 1145–1161. https://doi.org/10.1016/j.asoc.2016.07.018.
[31] J. Sarma, K. De Jong, An analysis of the effects of neighborhood size and shape on local selection algorithms, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 236–244.
[32] S. Shao, L. Guo, C. Guo, T. Chau, D. Thomas, W. Luk, S. Weston, Recursive pipelined genetic propagation for bilevel optimisation, in: Field Programmable Logic and Applications (FPL), 2015 25th International Conference on, 2015, pp. 1–6.
[33] D.J. Sheskin, Handbook of Parametric and Nonparametric Statistical Procedures, fifth edition, Chapman and Hall/CRC, 2011.
[34] N. Soca, J.L. Blengio, M. Pedemonte, P. Ezzatti, PUGACE, a cellular evolutionary algorithm framework on GPUs, in: IEEE Congress on Evolutionary Computation, 2010, pp. 1–8, doi:10.1109/CEC.2010.5586286.
[35] P. Vidal, E. Alba, Cellular genetic algorithm on graphic processing units, in: Nature Inspired Cooperative Strategies for Optimization (NICSO 2010), 2010, pp. 223–232.
[36] P. Vidal, E. Alba, F. Luna, Solving optimization problems using a hybrid systolic search on GPU plus CPU, Soft Comput. (2016) 1–19.
[37] P. Vidal, F. Luna, E. Alba, Systolic neighborhood search on graphics processing units, Soft. Comput. (2013) 1–18.
[38] L.D. Whitley, Fundamental Principles of Deception in Genetic Search, in: Foundations of Genetic Algorithms, 1990, pp. 221–241.
[39] L. Zheng, Y. Lu, M. Guo, S. Guo, C. Xu, Architecture-based design and optimization of genetic algorithms on multi- and many-core systems, Future Generation Comp. Syst. 38 (2014) 75–91.
[40] Y. Zhou, Y. Tan, GPU-based parallel particle swarm optimization, in: Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2009, 2009, pp. 1493–1500.