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.

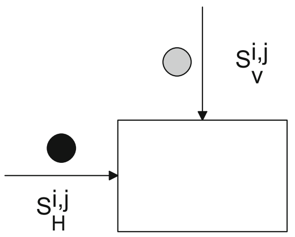
The rest of this section is structured as follows. First, in the next subsection the SGS algorithm is described. Then, the four different instantiations or flavors of SGS that are used in the experimental evaluation are introduced.

2.1 Systolic genetic search algorithm

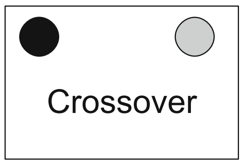
In a SGS algorithm, the solutions are synchronously pumped through a bidimensional grid of cells. At each step of SGS, two solutions enter each cell, one from the horizontal data stream ($S_H^{i,j}$) and one from the vertical data stream ($S_V^{i,j}$), as it is shown in Fig. 1a. Then, adapted evolutionary/genetic 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.

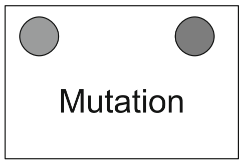
We want to remark that the idea of the SGS algorithm can be adapted to any solution representation and any particular operator. In this work, we address binary problems, for this reason we encoded the solutions as binary strings, and use bit-flip mutation and two-point crossover as evolutionary search 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.



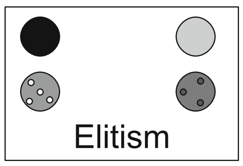
(a) Stage 1: receiving inputs.



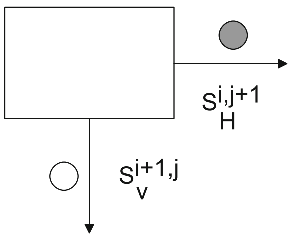
(b) Stage 2: internal operation.



(c) Stage 3: internal operation.



(d) Stage 4: internal operation.



(e) Stage 5: pumping outputs.

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

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
```

2.1.1 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).

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 div$(l, 2\tau)$, while cells of successive rows in the same column differ by div(l, τ). 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 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.

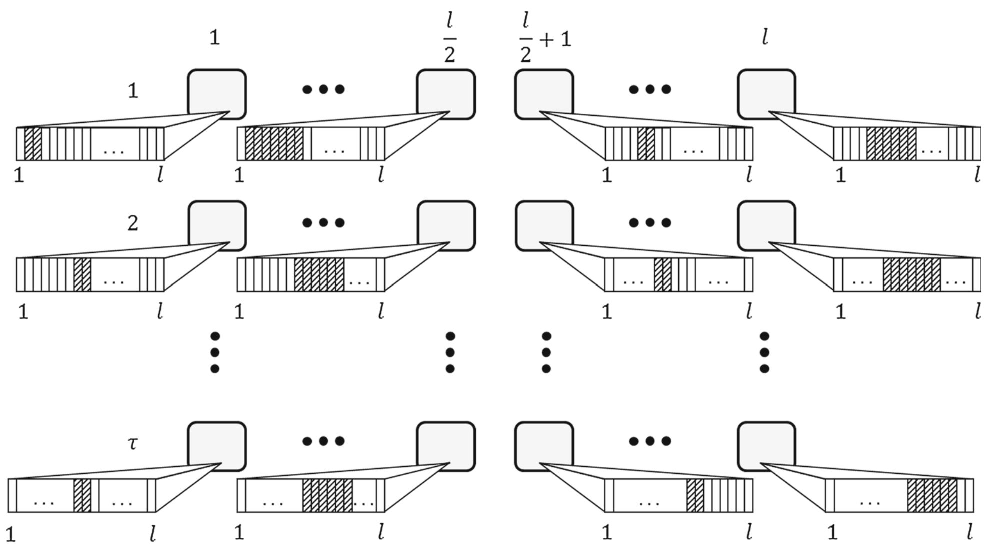


Fig. 2 Distribution of crossover points across the grid

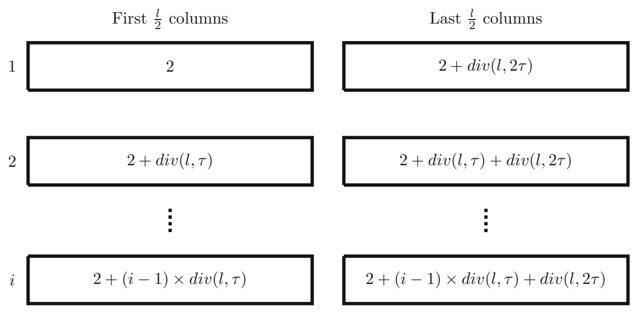

First $\frac{l}{2}$ columns
Last $\frac{l}{2}$ columns
1
2
$2 + div(l, 2\tau)$
2
$2 + div(l, \tau)$
$2 + div(l, \tau) + div(l, 2\tau)$
i
$2 + (i-1) \times div(l, \tau)$
$2 + (i-1) \times div(l, \tau) + div(l, 2\tau)$

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.

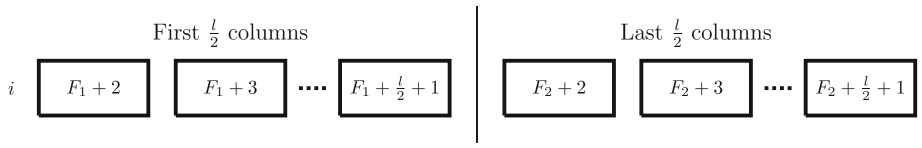
$$1 + \left(3 + \left\lfloor \frac{l}{\tau} \right\rfloor (i-1) + \left\lfloor \frac{j-1}{\left\lfloor \frac{l}{2} \right\rfloor} \right\rfloor \left\lfloor \frac{l}{2\tau} \right\rfloor + \left((j-1) \mod \left\lfloor \frac{l}{2} \right\rfloor\right)\right) \mod l \tag{2}$$

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



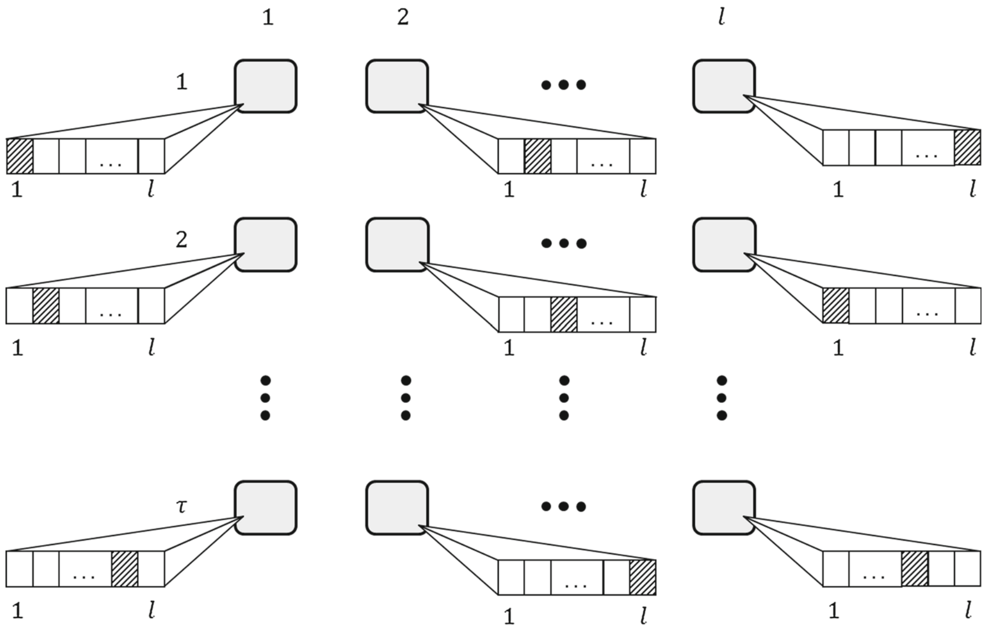


Fig. 5 Distribution of mutation points across the grid

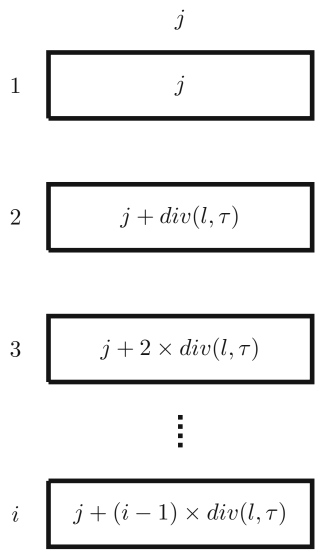


Fig. 6 Mutation points for column j

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

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

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-

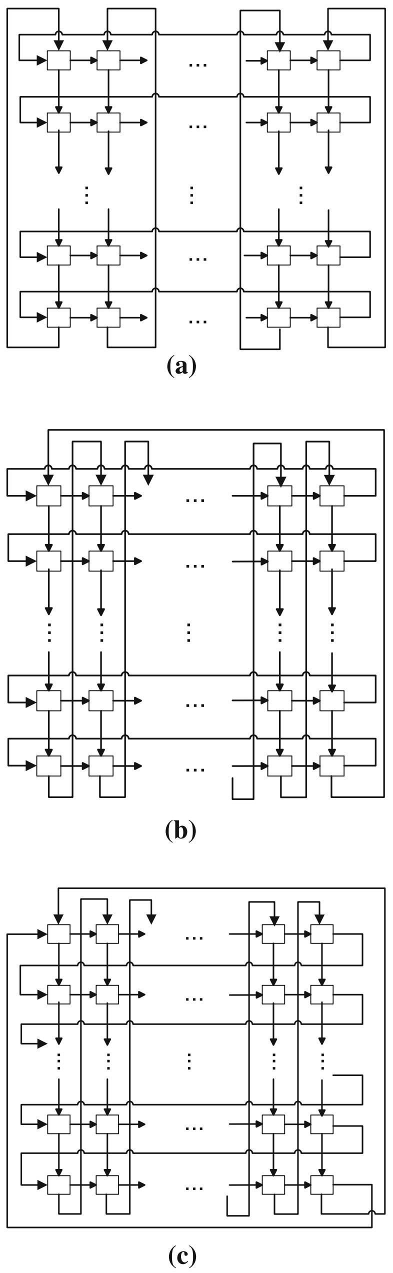


Fig. 7 Interconnection topology for the different flavors. **a** Toroidal grid. **b** Grid with horizontal toroidal flow and vertical flow to the next column. **c** Grid with vertical flow to the next column and horizontal flow to the next row

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 $\mathrm{SGS_B}$, together with the modification of the vertical flow that happens in $\mathrm{SGS_V}$ with respect to $\mathrm{SGS_T}$, 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 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.

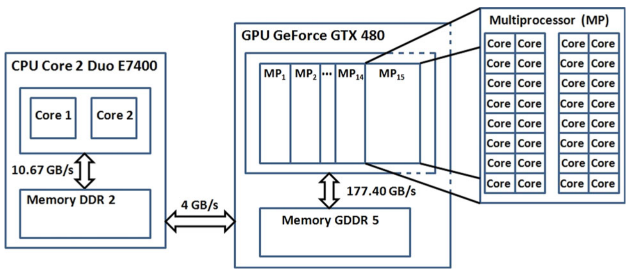


Fig. 8 CPU–GPU system used in this work

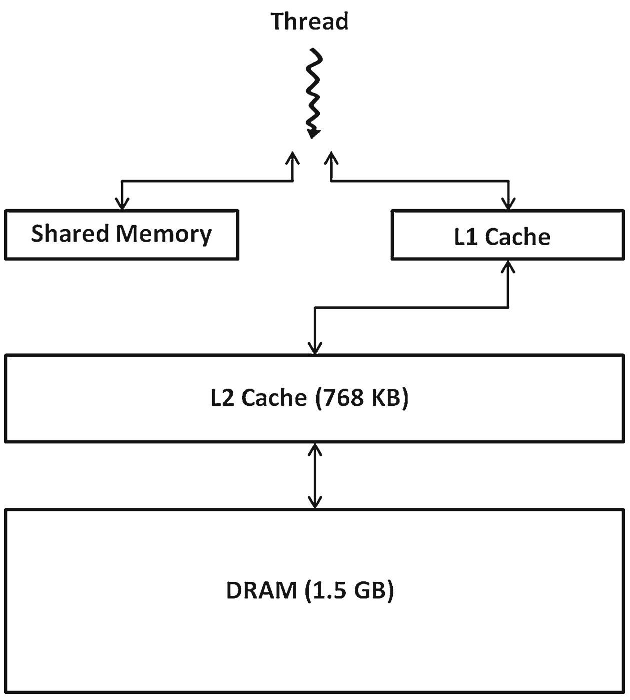


Fig. 9 GeForce GTX 480 (Fermi architecture) memory hierarchy

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.

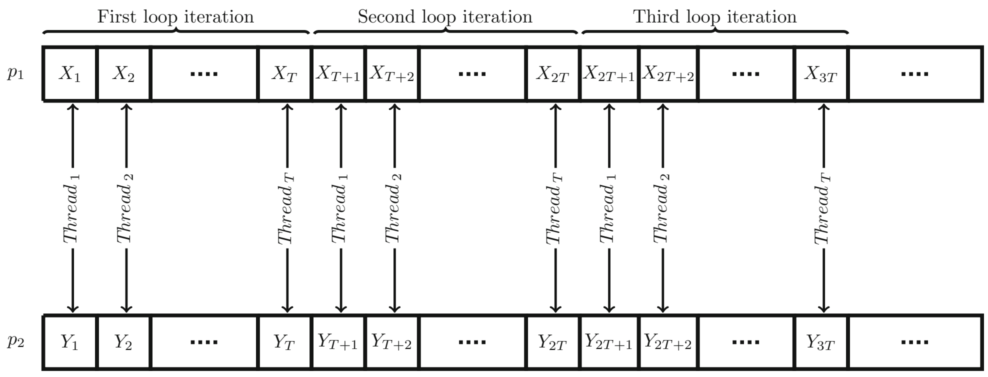

First loop iteration
Second loop iteration
Third loop iteration
p_1
p_2
X_1 X_2 .... X_T X_{T+1} X_{T+2} .... X_{2T} X_{2T+1} X_{2T+2} .... X_{3T} ....
Y_1 Y_2 .... Y_T Y_{T+1} Y_{T+2} .... Y_{2T} Y_{2T+1} Y_{2T+2} .... Y_{3T} ....
$Thread_1$
$Thread_2$
$Thread_T$

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 $\mathrm{SGS_B}$ 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 $\mathrm{SGS_E}$, $\mathrm{SGS_V}$ and $\mathrm{SGS_B}$ are the best performing algorithms for the KP, as they are far superior than RS, SGA and $\mathrm{SGS_T}$ 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 $\mathrm{SGS_V}$ and $\mathrm{SGS_B}$ find the optimal solution on every run for all the instances, while $\mathrm{SGS_E}$ 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 $\mathrm{SGS_T}$ in all the instances studied. Although the results obtained by $\mathrm{SGS_T}$ are not satisfactory due to the rather high mean error, $\mathrm{SGS_T}$ 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 $\mathrm{SGS_V}$ and $\mathrm{SGS_B}$ 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, $\mathrm{SGS_E}$, $\mathrm{SGS_V}$ and $\mathrm{SGS_B}$ are the best performing algorithms for MMDP. They all reach the optimal solution in every independent run for all the considered instances. EGA and $\mathrm{SGS_T}$ have a similar performance, having a small mean error and only being superior to RS. It is interesting that EGA performs worse than $\mathrm{SGS_E}$, $\mathrm{SGS_V}$ and $\mathrm{SGS_B}$. 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

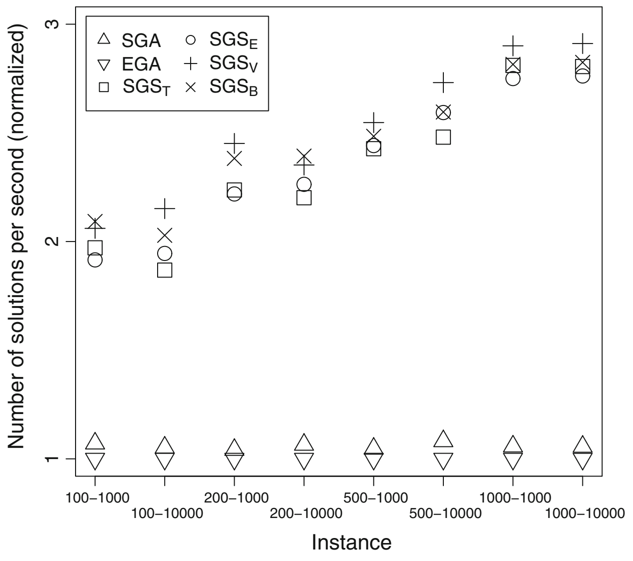


Fig. 11 Performance of CPU implementations for KP

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

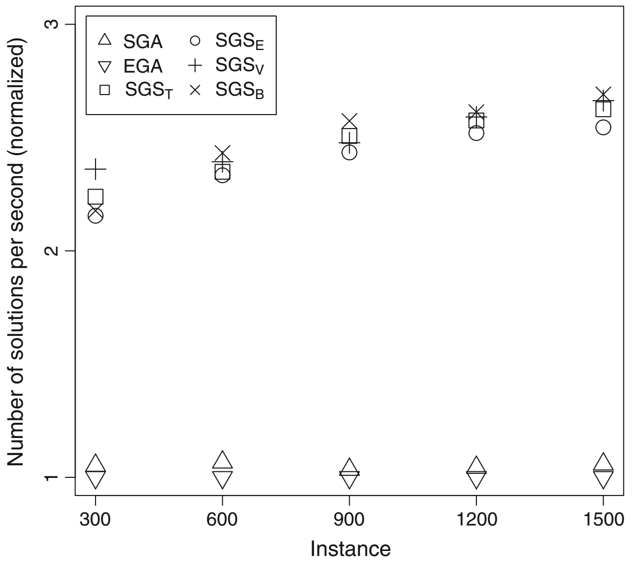


Fig. 12 Performance of CPU implementations for MMDP

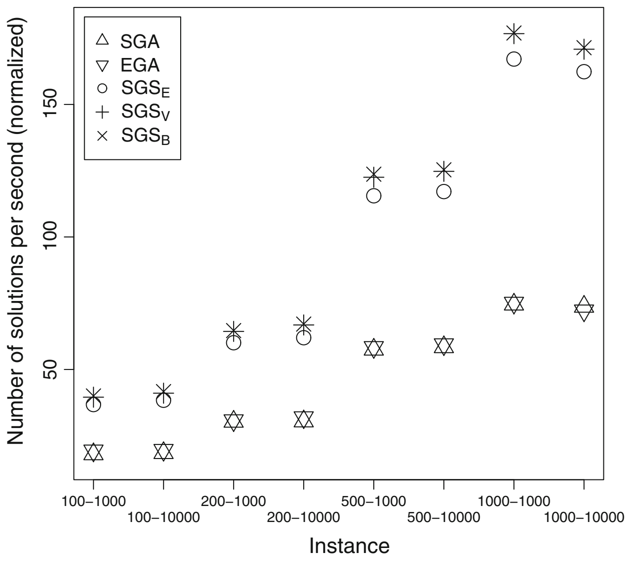


Fig. 13 Performance of GPU implementations for KP

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-

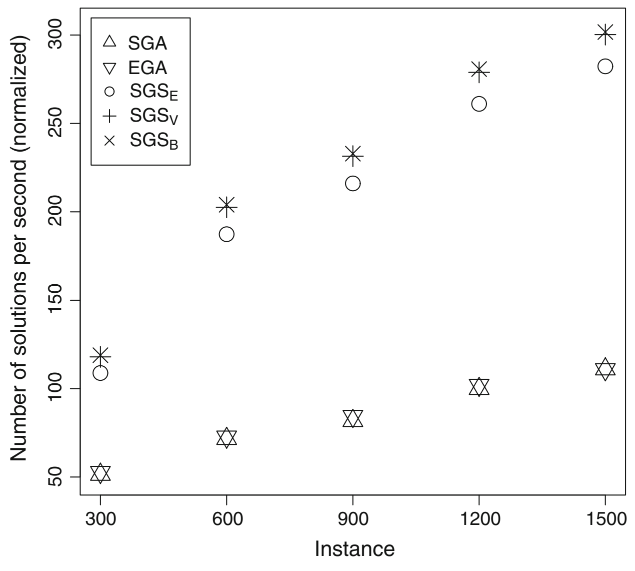


Fig. 14 Performance of GPU implementations for MMDP

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.

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

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 SGS_E, SGS_V and 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 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 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 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 SGS_E, SGS_V and 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, Dorronsoro 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