Soft Comput
DOI 10.1007/s00500-013-1041-7

METHODOLOGIES AND APPLICATION

Systolic neighborhood search on graphics processing units

Pablo Vidal · Francisco Luna · Enrique Alba

Abstract In this paper, we propose a parallel processing model based on systolic computing merged with concepts of evolutionary algorithms. The proposed model works over a Graphics Processing Unit using the structure of threads as cells that form a systolic mesh. Data passes through those cells, each one performing a simple computing operation. The systolic algorithm is implemented using NVIDIA's compute unified device architecture. To investigate the behavior and performance of the proposed model we test it over a NP-complete problem. The study of systolic algorithms on GPU and the different versions of the proposal show that our canonical model is a competitive solver with efficacy and presents a good scalability behavior across different instance sizes.

Keywords Optimization · Parallel computing · Systolic architecture · Evolutionary algorithm · CUDA

Communicated by G . Acampora.

P. Vidal (✉) · E. Alba
Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, E.T.S. Ingeniera Informática, Campus de Teatinos, 29071 Malaga, Spain
e-mail: pablo.vidal.20@gmail.com

E. Alba
e-mail: eat@lcc.uma.es

F. Luna
Departamento de Informática, Universidad Carlos III de Madrid, Avda. de la Universidad 30, 28911 Leganés, Madrid, Spain
e-mail: fluna@inf.uc3m.es

1 Introduction

The interest in parallel computation has progressively increased since the computers first appeared. The need for faster problem solvers has led researchers to create new hardware and to develop new software tools to address the demands of quick execution in domains such as simulation, bioinformatics, communications, and other fields of research and business Grama et al. (2003).

Over the years, various proposals on parallelism have been presented in the literature but have not fully worked for various reasons. One of these known as Systolic Computation. This concept was initially proposed at Carnegie-Mellon University by Kung Kung (1979). This architecture has a number of hardware components connected in a highly regular fashion within which data flow. Each component performs similar or identical task computations and the data flow through the network as blood through the heart, hence the name *systolic*. A systolic algorithm must focus on three basic points: the topology of the systolic structure, the computation in the cells and the data flow. First, each systolic algorithm must define the exact nature of the network, which indicates the propagation of information through the entire structure allowing a regular communication and control to enable efficient implementation. Second, the computation is performed by each component over different data in a repetitive, regular and simple manner. Finally, each cell can share the information with its neighbours immediately after processing in a pipelined fashion, exploiting concurrency and greatly reducing the overall execution time of the algorithm. To implement a systolic computing algorithm, one ideally needs access to a kind of hardware realization of the SIMD (Single Instruction, Multiple Data) type of parallelism. In the past, these kinds of architectures were not very popular

Published online: 07 April 2013

because of the difficulties of building systolic computers and, in particular, for programming high level algorithms on such low-level architecture. Nowadays, the advances in technology have allowed the emergence of new, powerful and cheap computing platforms such as Graphics Processing Units (GPUs), that are devices especially well suited for massively parallelism.

Over the last few years, as the complexity and computational efforts of computer graphics algorithms have grown, GPUs have progressively become more flexible and have incorporated fully programmable processing units. They have evolved from fixed function hardware for the support of primitive graphical operations to programmable processors that outperform conventional CPUs. This architecture presents an opportunity to execute thousands of concurrent threads onto hundreds of processing units, which makes GPUs ideal accelerators for many arithmetic intensive data-parallel workloads found in science and engineering applications. Moreover, other features such their low cost prices and programming model make them particularly well suited for scientific numerical computations. They present a tremendous opportunity for developing new lines of research in optimization algorithms especially targeted for GPUs. This may lead to improved efficiency and effectiveness by merging the inherent parallelism of the architecture with existing or new optimization techniques, that have been proven effective in various fields of research and real life Alba (2005).

This paper proposes a novel algorithm approach called systolic neighborhood search (SNS). The proposal uses existing concepts in systolic computation and develops new techniques to exploit the resources offered by the architecture of a GPU using a systolic approach. Furthemore, if we consider existing optimization methods in our approximation, we allow for the addition of a powerful tool for solving multiple complex problems. Our intention is to ignore hardware cells (mandatory in the past) and consider a GPU as a container of structured components, where each one performs an easy and homogeneous optimization task. This kind of model has proven effective in other optimization problems as shown in Alba and Vidal (2011). This previous work introduces a first approximation of the model and demonstrates the effectiveness of the design based on a very small group of instances. In this work presents an in-depth analysis of the proposed model and several extensions that clearly improve the effectiveness of the original algorithm. In order to evaluate the behavior and performance of SNS and several modified versions we will use a large testbed composed of instances of increasing size of the Multidimensional Knapsack Problem, and compare their results with different versions of a Genetic Algorithm and a Random Search implemented both in CPU and GPU.

The paper is structured as follows. Section 2 explains what systolic computation means here. Section 3 briefly introduces the concept of the GPU. The implementation details of SNS are presented in Sect. 4. Section 5 explains the problem used, the parameter settings of the algorithms, the statistical tests used, and the studies performed. Section 6 discusses the results obtained. Finally, Sect. 7 is devoted to conclusions, as well as some comments about future work.

2 Systolic computation

Systolic computation represents a realistic model of computation which captures the concept of pipelining, parallelism, and interconnection structures. This kind of concept was originally created for VLSI implementation for some matrix operations Kung (1979). The name "systolic" derives from the bioinspired analogy with the regular pumping of blood in heart and the pumping of data through an array of simple processing units (cells).

Thus, a systolic system works over a large array of connected data processing units called cells, each capable of performing simple operations. A systolic cell is the basic workhorse of the system and can be a small processing element, hardware or software, depending of the environment of development. The work of each cell is usually the same as others, and it performs simple tasks like computing simple functions like multiplications or summations. Cells are typically interconnected to form a systolic array. The function of the array is to keep a constant flow of data throughout the entire network in a pipelined fashion. This disposition allows a very simple regular communication between cells. With the idea of pipelining, processing may proceed concurrently between input and output. Consequently, the overall execution time is minimized. Usually, the communication with the outside world occurs in the boundary cells transferring data to other systems, or used as feedback for the system itself. The use of systolic arrays enables the creation of various topologies, such as rectangular, triangular and hexagonal networks. This flexibility in the architecture allows the use of a higher degree of parallelism. Moreover, data flow in a systolic mesh may have multiple spread speeds in multiple directions and is able to receive data from different directions arriving at cells in the array at regular intervals where they are combined. In general, this kind of system is very easy to implement and reconfigure because of its modularity. All of these basic concepts are illustrated in Fig. 1 where each cell o produces a change over data a and b and then moves the result to the next cell.

Now, we try to explain on the one hand, some approaches in the optimization field using as basis the systolic

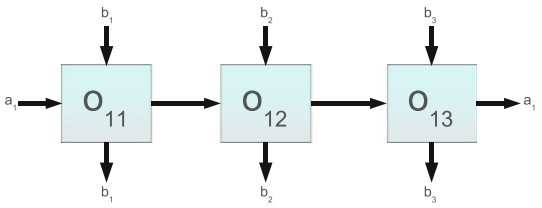


Fig. 1 Systolic array, the basis of systolic computation

architecture. On the other hand we discuss a similar approach to the one presented here, showing the features and main differences between them.

In general, systolic computation has been absent in the field of parallel computing and optimization in general. Some trials in this regard have been carried out in the past Chan and Mazumder (1995) Megson and Bland (1998). Both trials consisted in implementing the operators of a GA on VLSI and FPGA in a systolic fashion, respectively. The difficulties for expressing the operations of a GA in terms of the recurrent equations needed in order to later use such hardware building blocks (multiplications, summations) led to an early abandonment of this line of research without even a simple evaluation of the performance or a comparison to existing algorithms. This was simply an (interesting) intelectual exercise, with no further implications.

Other approaches have been proposed in the literature with the aim of improving performance of algorithms in parallel computing platforms. There are several of these approaches that use a similar disposition of the algorithmic components as the systolic model. Among them, the parallel Evolutionary Algorithms models that use structured populations Alba (2005) are rather similar and we want to make clear the difference between them and our approach. Cellular EAs (cEAs) Alba and Dorronsoro (2008) are one of the main types of EAs with structured populations in which the individuals are usually arranged in toroidal grid. The cEA introduces the concept of "neighborhood", this mean that an individual may only cooperate with its nearby neighbors. The overlap between small neighborhoods in the cEAs helps in exploring the search space because the induced slow diffusion of solutions through the population provides a kind of exploration (diversification), while exploitation (intensification) takes place inside each neighborhood by stochastic operations. The only similarity relies on the topology since each systolic array can be a row of the grid of the cEA. The differences, however, are relevant and strongly determine the search engine of the algorithms. Firstly, the information flow within the two topologies. The cEA topology works on solutions that remain static in the same position on the grid over time and the communication of information is given by the overlap between neighborhoods. While the systolic system performs a constant movement of all the information through the array, allowing communication between cells in a fast and fluid manner. Secondly, the basic operation of the cells within each algorithm is also very different. In other words, each cell in the cEA needs its neighborhood to generate the new individuals. In turn, each systolic cell performs isolated operations, which means that the cell's decisions do not affect the contiguous ones.

3 Graphics processing units

Recently, GPUs have emerged as a powerful computational platform. The model for GPU computing uses a CPU and GPU together in a heterogeneous co-processing computing model. The sequential part of the application runs on the CPU and the computationally-intensive part is accelerated by the GPU. GPUs may contain several hundred simple processors, and typically run a single program on many different sections of data in parallel. The set of processors are usually considered a single instruction multiple data (SIMD) computer.

Compute unified device architecture (CUDA) is an extension of the C programming language and was created by nVidia from the G80 GPU generation. CUDA is a C language environment that provides services to programmers and developers ranging from common GPU operations in the CUDA library to traditional C memory management semantics in the CUDA run-time.

Two important core concepts in programming with CUDA are thread batching and memory model CUD (2007). When working with CUDA, the GPU is viewed as a computing device capable of executing a very high number of threads in parallel. Each thread executes a function called kernel, which is launched from the host and is executed by a batch of threads on the GPU. The batch of threads can be organized as a grid of thread blocks. A thread block is a batch of threads that can cooperate together by efficiently sharing data through diverse levels of memory and synchronizing their execution to coordinate memory accesses. The thread batching mechanism is illustrated in Fig. 2.

As mentioned above, the memory hierarchy is an important feature of modern GPUs. Nowadays, current GPUs that we work have four levels of memory: registers, local memory, shared block memory, and global memory. Registers are the fastest memory in the multiprocessor and are only accessible by each thread. The local memory exists because its scope is local to the thread, not because of its physical location. In fact, local memory is off-chip. Hence, access to local memory is as expensive as access to global memory. Shared memory is almost as fast as registers and can be accessed by any thread in the same block. Finally, the global memory is the slowest (but largest)

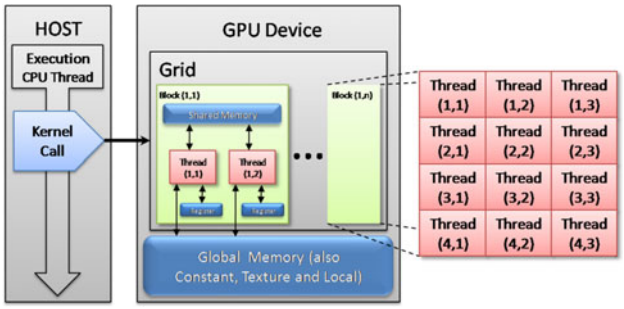


Fig. 2 Architecture of the CUDA model and the distribution of different memories in GPU

memory in the multiprocessor (it is more than 100 times slower than shared memory), and while the local memory is only accessible to each thread, the global memory is accessible to all the threads on the GPU. Also, allocated in the global memory there is space for constant and texture memory. The constant memory is optimized for broadcast and is where the constants and kernel arguments are stored. Texture memory is a special type of read-only cache optimized for 2D spatial access pattern.

The two factors described (thread batching and memodory distribution) are of great importance for the performance of a given GPU implementation. As a consequence, GPU algorithms have to be carefully engineered so as to profit from the full parallel capabilities of the GPU cards.

4 Systolic neighborhood search

The goal of this section is to present our algorithmic proposal, called SNS. The essential idea of this algorithm is to provide a mesh where the tentative solutions are located in each cell, then perform some simple operation and then the result is moved to the next cell to start the same process. In this sense, SNS follows the idea of moving the solutions through the entire mesh with the objective of performing different movements on the solution values.In particular, solutions are conceptually set in a toroidal mesh defined in the GPU. The potential of the algorithm relies on exploring the search space efficiently by performing small modifications in the tentative solutions passing different stages across the mesh. As explained in the introduction section, a systolic algorithm is defined providing three main components: topology, cell computation and data flow. They will be explained in the following sections.

4.1 Topology

Systolic neighborhood search uses the systolic concept to provide a new computing organization. The solutions are arranged in a toroidal grid (especially well suited for GPU architecture). The core idea is that each solution in the grid is a piece of information that flows through the units of computing (cells). Each cell performs small modifications in the incoming solutions with the aim of improving them. This model seeks to take advantage of two effects: the modifications of the solutions within each cell and the improvement of the solutions moving across the grid. Fig. 3 shows the disposition of the elements of SNS, in which each row is a representation of systolic array. Each element can be located through the axis (x, y) within the matrix. The cell $T_{x,y}$ has a solution $S_{x,y}$ which undergoes simple modifications at each step i of the evaluation. However, the dimension of this mesh is defined by the k number of components in the problem tested. In this way, we create an $m \times n$ matrix, with m and n equal to k.

SNS is described in Algorithm 1. It starts by defining the size of the grid, that depends of the problem size. Each solution is usually encoded as an array of values (depending on the given optimization problem). Then SNS generates these solutions at random and evaluates them. Next, several operations within each cell are performed aiming to improve on the current solution. So, each cell on the grid works over a single solution located in a position (x, y).

Algorithm 1 Pseudocode of a Canonical SNS

```
1: Define variable m and n depending the problem size
2: Define mesh M of size m × n
3: for all sol_(x,y) ∈ M do in parallel
4:    (x, y) ← compute position of the cell in the mesh
5:    sol_(x,y) ← generateSolution(sol_(x,y));
6:    sol_(x,y) ← evaluateSolution(sol_(x,y));
7: end for
8: while (not stop_criterion) do
9:    For each solution sol_(x,y) ∈ M do in parallel:
10:      cellComputation(sol_(x,y)) //Algorithm 2
11: end while
12: getBest(M).
```

4.2 Cell computation

The basic idea is that every cell in SNS performs a simple operation to change part of the components in the solution. Algorithm 2 presents the operations performed by each cell in the SNS algorithm. Likewise, Fig. 4 provides a graphical guide to better understand the behavior within each cell.

Algorithm 2 starts creating a copy $sol'_{(x,y)}$ of $sol_{(x,y)}$, the newly incoming solution that reaches the cell (line 2). Then, $sol'_{(x,y)}$ undergoes a perturbation process that depends on x and y, i.e., its location in the mesh (line 3). In part of $sol'_{(x,y)}$ we use the values of position x and y to identify each cell. Parameter x defines the number of values to be changed in the vector solution, while parameter y indicates the left most position in the array where changes start. Thus, if we have a solution $s_{(2,1)}$, we modify two values starting the position

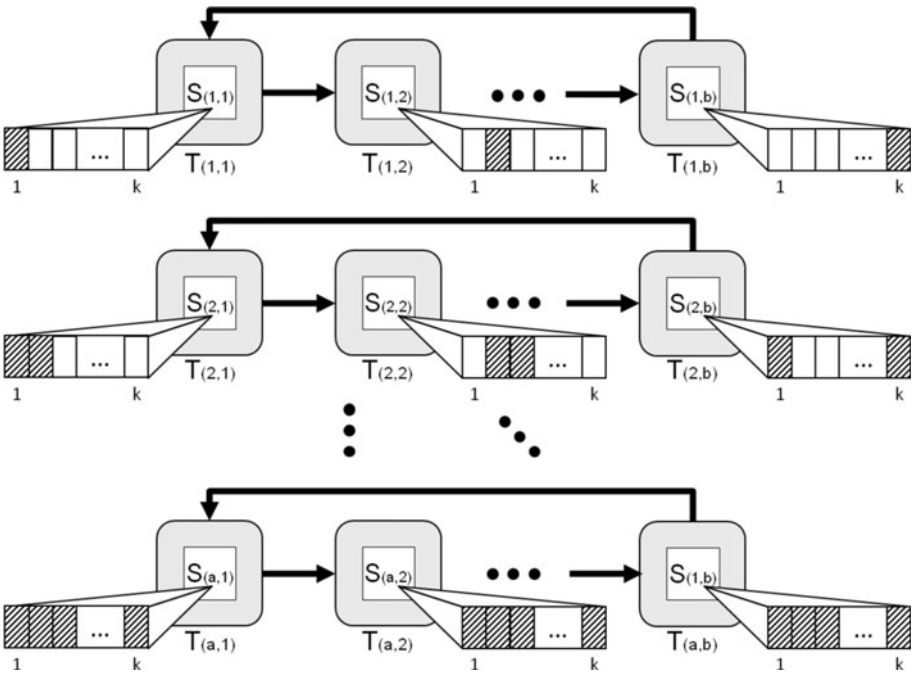


Fig. 3 Systolic neighborhood search

one on, in the vector solution s. In this way, the row where the solution is located defines the number of variables to be changed, and the column position, the location in the vector from which to begin the changes. Once all the changes have been made, the total re-evaluation of the new solution can be very time consuming. So, we decided to use a partial evaluation, thus re-evaluating only the changed section in the vector solution copy (line 4). So, SNS evaluates the copy solution sol'_i. After the changes, if the $sol'_{(x,y)}$ is better than $sol_{(x,y)}$, replace it with the copy to the contrary, the old solution stays without changes (line 5–7).

Algorithm 2 Computation in each cell of SNS

1: Compute x and y position of sol inside the mesh
2: $sol'_{(x,y)} \leftarrow copy(sol_{(x,y)})$
3: Update the solution components from y to $(y+x)$ position
4: $partial\_evaluation(sol'_{(x,y)})$
5: **if** $sol'_{(x,y)}$ $BETTER\ THAN$ $sol_{(x,y)}$ **then**
6: $\quad sol_{(x,y)} \leftarrow sol'_{(x,y)}$
7: **end if**
8: Wait for all the cell to complete their operations (from line 2–7)
9: Move $sol_{(x,y)}$ to the next cell in the mesh

4.3 Data flow

When all the cells of the mesh have performed the changes, SNS moves all solutions in each cell to the next one in the row, synchronously (line 8–9 in Algorithm 2). In this way, we verify that each cell finish all the operations. Also, the reason that m and n are equal to the instance problem size is justified in the flow of the solution through the rows. This flow means that all the solutions can visit the cells in a row and they can suffer changes following a simple pattern of modifications in all the parts of the vector solution.

4.4 Extensions to SNS

Up to now, we only given a rough outline containing the basic elements. In our quest for new algorithms capable of solving different problems and improving the actual performance of SNS, in this section we are going to further

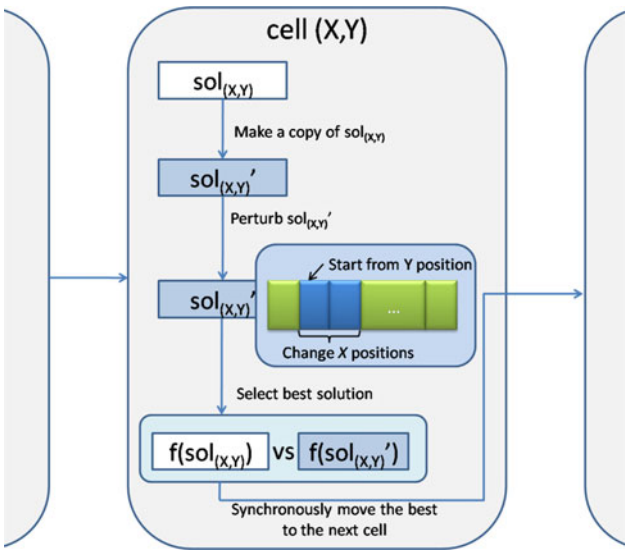


Fig. 4 Graphical operation of the $cell_{(x,\ y)}$

consider several extensions to the basic algorithmic scheme presented before.

4.4.1 Exponential SNS (SNS^{exp})

In a canonical SNS, we use a mesh of $m \times n$, where m and n equal to the value of the problem size k, in order to explore all possible combinations in the array of values of each solution. The idea is that each row modifies a certain number of values within the array, for example, in row 1 changes only 1 bit in all the solutions of that row, row 2 changes two values, and so on, until row m changes k values within each solution. However, all these modifications do not ensure improvement in the quality of solutions and are too numerous, maybe precluding scalability. SNS^{exp} intends to reduce the number rows to evaluate with respect to the original SNS. By reducing the number of rows we can generate a greater flow of solutions, reducing the number of evaluations. However, we cannot use the same mechanism of incremental changes in the solutions per row, so, we need to define two things: the number of rows and how many components change per row. The first factor is computed using the problem size: SNS^{exp} will have $\log_2 k$ rows. Second, as SNS^{exp} cannot use incremental changes in the solutions per row, simply because there are not enough rows to use this mechanism. We use operation 2^x according to the previous computation of row numbers. The exponent x indicates the number of the row. With this approach, the number of values to be changed increases in multiples of two.

4.4.2 Random SNS [(r)SNS]

Until now, *SNS* and SNS^{exp} algorithms have been described to work over solutions changing a group of contiguous components. When the algorithm examines contiguous modifications, the search space is reduced by the group of components to change. Also, in some cases, changes can affect part of the solution that provides initial benefit and later was lost. So, we decided that the choice of those values to be changed could be random.

Therefore, we propose a new extension called (*r*)*SNS* will work like *SNS* but it will focus on making a simple random selection over the components in the solution with the same probability.

This approach attempts to focus the effort in exploring the search space, looking to improve the solutions again with changes in the vector solutions but now adding the feature of exploring a solution through random changes.

The changed performed by (*r*)*SNS* can be seen in line six of Algorithm 1, where instead of changing the bits sequentially $(y + start)$, (*r*) *SNS* changes x bits of the solution randomly.

4.4.3 SNS with movement of rows (mr)SNS

The SNS information flow is based on the movement of solutions through the cells in the same row. With the canonical SNS it is possible to reach a point where the changes will not substantially improve upon the solution. Also, in the lower area of the structure of the SNS, the process of changing an enormous number of bits in each solution can result in the loss of information or not contribute significantly, thereby provoking a premature convergence to the final fitness in these rows. Our focus is to move a group of solutions located in a row to the next one located below it. With this movement the grid is transformed in a toroidal grid for down to up. At first, the mechanism is the same as with the canonical SNS, but when the solutions are completed the movements through all the cells in the same row, all their solutions are moved to the next row below to continue the process of changing the bits. This method is defined as SNS with movement of rows [(*mr*)*SNS*].

4.5 Implementation details

So, we define many families of SNS algorithms, and some implementation details are important to run these in the GPU.

The entire execution of these algorithms, creation, evaluation, and operation of changes are performed on the GPU as kernels. We try to fully exploit the performance of the device and minimize communications to the CPU.

During the execution, the SNS family generates the initial solutions randomly. In order to avoid transferring any data between CPU and GPU, the SNS family passes a global seed as a parameter to the kernel only once at the beginning. Then, each time that each SNS version initializes a new iteration, other global seed is passed to the algorithm and each thread keeps a copy of this seed. The local seed is used to generate random numbers with a fast Linear Congruential Generator (LCG). This approach was used also by Lagae et al. (2009) and Thomas et al. (2009). This approach of the LCG allows working with independent seed between threads and provide independent spatial values.

The group of solutions is stored in the device's memory, more precisely in the global memory. All the solutions are created on the GPU at the beginning of the execution of the algorithm, and are transferred back to the CPU when the algorithm finishes. To improve the algorithmic performance, we could have exploited the different GPU memory levels. But given the different sizes of the instances, the amount of information used and the fact that we are working with instances of thousands of items, we decided to use the global memory instead of the shared memory. By

doing this, we can provide a canonical implementation for all the problem sizes.

5 Experimental settings

This section is devoted to describing the methodology that we have used in the experiments carried out in this work. First, we introduce the definition of the Multidimensional Knapsack Problem and the features of the selected instances. Next, a description of each algorithm used to compare with the SNS versions is given. The methodology used, parameterization and a brief explanation of the representation used for this kind of the problem are also presented.

5.1 Multidimensional knapsack problem (MKP)

5.1.1 Definition

The MKP is one of the most well-known integer programming problems and has received a great deal of attention from the operational research community over the last four decades. The MKP is a NP-hard problem which represents the core of a large class of important real problems such as cutting stock and loading problems, resource allocation in distributed systems, delivery of goods in vehicles with multiple compartments and approval voting Gavish and Pirkul (1986); Gilmore and Gomory (1966); Shih (1979). We have chosen MKP not only because of the actual real-world optimization problems it represents, but also because of the availability of increasingly sized instances that shall enable us to evaluate the scalability capabilities of SNS. Formally, the MKP problem is defined as follows:

$$\text{maximize} \quad \sum_{j=1}^{n} p_j x_j \tag{1}$$

$$\text{subject to} \sum_{j=1}^{n} w_{ij} x_j \leq C_i,\ i \in I, I = \{1, \ldots, m\} \tag{2}$$

$$x_j \epsilon \{0, 1\},\ j \in J,\ J = \{1, \ldots, n\} \tag{3}$$

where n is a number of items, m is the number of knapsacks, where each one has a maximum capacity C_i ($i \in I$). Each item j has a p_j profit and consumes a given amount of resource w_{ij} for each knapsack i. The objective consists in selecting a subset of given items in such a way that the total profit of the selected objects is maximized while a set of knapsack constraints are satisfied. This kind of problem has been widely studied in the literature and different algorithms have been developed for obtaining optimal or very near-optimal solutions Hanafi and Wilbaut (2011); Hwang et al. (2011); Ke et al. (2010).

In some cases, some solutions may violate the weight restriction for C_i, leading to infeasible solutions. In the literature it is common to see the use of a penalty function able to modify this infeasible solutions and provide feasible solutions. We decided to use the function penalty presented by Gottlieb (2001).

$$penaly_x = \frac{p_{max} + 1}{w_{min}} . max\{CV_{(x,i)} | \, i \epsilon I\} \tag{4}$$

where $CV_{(x,i)} = max(0, \sum_{j \epsilon J} w_{ij} * x_j - C_i)$ is the amount of constraint violation for constraint $i \in I$, $w_{min} = min\{w_{ij} | i \in I, j \in J\}$ indicates the minimum resource consumption and $p_{max} = max\{p_j | j \in J\}$. $(p_{max} + 1)/w_{min}$ is used to correlate the objective function based on profits with the resource demands that exceed the capacity constraints. This factor is based on a pessimistic estimation of the profit that would get lost if items were removed from the knapsack in order to obtain feasibility. For more details, the reader is referred to Gottlieb (2001).

5.1.2 Instances

The reason for choosing these problems was driven more by the need to compare our computational results to results already published, than by their effective hardness. The MKP instances addressed here have been divided into three groups, that we have called *small, medium* and *big instances*. The first and second groups correspond to the well-known set of MKP instances proposed by Chu and Beasley (1998). From the first group of instances (*small instances*), we have selected 18 representative instances that vary from $n = 6$–105 items and the number of knapsacks between 2 and 30. The second set of instances (*medium instances*) is composed of the biggest ones in the OR-Library. We have selected from files *mknapcb5.txt, mknapcb6.txt, mknapcb8.txt* and *mknapcb9.txt* the first five instances with $n = \{250, 500\}$ and $m = \{10, 30\}$. The final group (*big instances*) has been selected from the instances defined by Glover and Kochenberger (1996), more specifically the instances GK08 $\{n = 500, m = 50\}$, GK09 $\{n = 1500, m = 25\}$, GK10 $\{n = 1500, m = 50\}$ and GK11 $\{n = 3500, m = 100\}$. This set is very large and varied, so as to capable of analyzing our algorithms in a fair and representative manner.

5.2 Algorithms used in the comparison

Two algorithms have been used to evaluate the search capabilities of SNS: Random (RS) and Genetic Algorithm (GA) algorithms. An important usage of RS is to provide a sanity check with respect to the results of SNS. GA is a

well-known classical algorithm that has been chosen as an advanced intelligent search which will serve to put the results of SNS in the context of optimization.

5.2.1 Random search

- *Random Search on CPU*: RS on CPU initializes a solution x randomly and evaluate it. Then, until a termination criterion is met, RS repeats the process of creating and evaluating a new solution y. If y is better than x, replace x with y otherwise x is left unchanged. Algorithm 3 presents the pseudocode of the canonical RS.

Algorithm 3 Pseudocode of a Canonical Random Search on CPU

```
1: sol ← generateSolution();
2: sol ← evaluateSolution();
3: while (not stop_criterion) do
4:    sol' ← generateSolution();
5:    sol' ← evaluateSolution();
6:    sol ←getBest(sol, sol');
7: end while
```

- *Random Search in GPU*: RS on GPU works on a population of size n. Basically, each solution is managed by an independent thread of GPU simultaneously in parallel. Each thread applies the same process on each single solution as a Canonical RS. When the stop condition is reached, the algorithm gets the best solution from the entire population. This behavior is explained in Algorithm 4.

Algorithm 4 Pseudocode of a Canonical Random Search on GPU

```
1: for all sol_i ∈ pop do in parallel
2:    sol_i ← generateSolution();
3:    sol_i ← evaluateSolution();
4: end for
5: while (not stop_criterion) do
6:    for all sol_i ∈ pop do in parallel
7:       sol'_i ← generateSolution();
8:       sol'_i ← evaluateSolution();
9:       sol_i ←getBest(sol_i, sol'_i);
10:   end for
11: end while
12: getBest(pop);
```

5.2.2 Genetic algorithm

Genetic Algorithms (GAs) have been chosen as the base method in many optimization problems. Also, GAs have been widely used in the literature Zhang et al. (2012); Zhou and Luo (2010). We have considered two implementations of GA both on CPU and GPU, respectively. Each GA starts by generating solutions randomly. Then, the algorithm enters an iterative process, where it randomly selects two parents from the population. After selecting two parents, we cross the two parents, what results in a single individual: the one having the larger part of the best parent. A mutation at bit flip level is immediately applied; next, an evaluation of the offspring solution is performed. Finally the offspring is stored in an auxiliary population *aux_pop*. When *aux_pop* is full, the GA replaces the original population with *aux_pop*. This process is repeated until the stop condition is reached. The two GA implementations thus present a generational replacement policy.

- *Genetic Algorithm on CPU* This version is a canonical version of the algorithm and it is used as base line to measure the efficiency of our algorithms.
- *Genetic Algorithm on GPU* This algorithm aims to test the efficiency of the canonical GA algorithm on the same hardware used by our SNS approach. The main difference with the GA on CPU is that we create as many threads as individuals exist in the population, and the entire process is done in parallel. Each of these threads is responsible for the genetic operations. When all the threads have completed the operations, each one replaces in the population (located in the device global memory) the old solution with the new offspring created. Algorithm 5 shows the behavior of the GA on GPU.

Algorithm 5 Pseudocode of the Canonical GA on GPU

```
1: for all sol_i ∈ pop do in parallel
2:    sol_i ← generateSolution();
3:    sol_i ← evaluateSolution();
4: end for
5: while (not stop_criterion) do
6:    for all sol_i ∈ pop do in parallel
7:       parents ← sol_i, selectParents(pop);
8:       offspring_i ← crossover(parents);
9:       offspring_i ← mutation();
10:      offspring_i ← evaluateSolution();
11:      sol_i ← replacement(offspring_i, i);
12:   end for
13: end while
14: findBetter(pop);
```

5.3 Solution encoding and operators

The problem encoding that we use here is a regular binary encoding. The size of all instances can stay inside the GPU in one batch, and thus we improve the compactness of data. The implementation works at the bit-level in order to save memory space.

The two GA implementations use the next configuration: for the recombination operator we use two point crossover (with a probability $p_c = 0.9$) yielding one single solution: the one keeping the largest part of the best parent in it. The bit mutation probability is set to $p_m = 0.01$, and is applied over each value of the vector solution. This parameterization is presented in Table 1.

5.4 Methodology

We have chosen a stopping condition so that we can guarantee a similar exploration of the search space for all the instance problem sizes. So, we decided to use a variable number of evaluations taking into account previous works over the MKP (Chu and Beasley 1998; Vasquez and Vimont 2005; Hanafi 2009). First, we compute a number $evals = n$*5,000. Then, according the value of *evals*, we decide what is the most appropriate number of evaluations. The parameters are defined in the following equation:

$$maxEvals = \begin{cases} n * 5000 & \text{if } evals \geq 400000 \\ & \& \; evals \leq 3000000 \\ 400000 & \text{if } evals \leq 400000 \\ 3000000 & \text{if } evals \geq 3000000 \end{cases}$$

We executed 30 independent runs for each algorithm in each instance problem size. Once the results of the executions are obtained, we perform statistical tests in order to ensure the confidence and check whether the results are statistically significant. Firstly, a Kolmogorov Smirnov test is performed in order to check whether the values of the results follow a normal distribution (Gaussian) or not. If the distribution is normal, then we apply the Levene test to check the homogeneity of the variances. If samples have equal variances (positive Levene test), an ANOVA test is performed, otherwise a Welch test is performed. For non Gaussian distributions, the non-parametric Kruskal–Wallis test is used to compare the medians of the algorithms. In this work we always consider a confidence level of 95 % in the statistical tests. This means we can guarantee that the differences of the compared algorithms are significant or not with a probability of 95 % and the observed algorithm differences are unlikely to have occurred by chance with a probability of 95 %.

Table 1 Parameter settings

Operator	Value
Selection of parents	Tournament Selection ($k = 2$)
Recombination	Two-point crossover with $pc = 1.0$
Mutation	Bit-flip with $pm = 0.01$
Replacement	Replace if better

5.5 Parameterization

The population used by the RS on CPU is the usual (1 solution) while the RS on GPU use a population size equal to the k value. On the other hand, the GAs work with a population of fixed size, of 100 solutions. However, to make a fair comparison between the GA and SNS versions we have decided that the GA also works with a population equal to the *SNS* and SNS^{exp} version.

The experiments are run on a host with a CPU Intel (R) i7 CPU 920, with a total physical memory of 4096 MB. The operating system is Ubuntu Lucid 10.04.2. In the case of the GPU, we have an NVIDIA GeForce GTX 285 with 1024 MB of DRAM on device. We used CUDA version 3.1 and the driver for the graphic card is version 257.21.

6 Experimental results

This section is devoted to the presentation and analysis of the results of the 30 independent executions for each algorithm over the three groups of MKP instances. The presentation of our results is structured according to the instance size (small, medium and big). We analyze the solution quality and wall-clock time obtained in each group of instances. In order to make this experimentation fully reproducible, both the source code of the SNS algorithms and the instances used are publicly available for interested readers in http://neo.lcc.uma.es/staff/pablo/sns.

6.1 Small size instances

Tables 2 and 3 includes, respectively, the fitness value and the execution time of the small size instances averaged over 30 independent runs.

Table 2 clearly indicates that the (r, mr) SNS^{exp} algorithm gains the largest (best) values in 11 out of 16 instances. Our approach and the extensions used (random changes and movements between rows) have created an algorithm that obtains the best results for this group of instances. The advantage of (r, mr) SNS^{exp} is basically based on three factors: SNS^{exp} manages a lower number of solutions than the canonical *SNS*, hence, the number of allowable maximum iterations increases significantly and therefore more changes are tested in the solutions. The second and the third factors are related to the two variations added to the canonical behavior of the SNS and SNS^{exp} versions: random changes and movements of solutions between rows. The ability to change i values randomly increases the diversity in the solutions and avoids the premature convergence of the population. Also, the flow of solutions between rows makes it possible to modify a more diverse group of components inside the solutions through

Table 2 Average fitness value of 30 runs, for small instances of MKP problem

Algorithm	SENTO02	WEISH01	WEISH02	WEISH06	WEISH08	WEISH10	WEISH12	WEISH16
R_{CPU}	7,968.233	4,301.400	4,319.600	5,025.367	5,065.533	4,677.233	4,547.667	5,136.367
R_{GPU}	7,923.467	4,183.955	4,101.000	5,047.767	5,081.033	4,707.767	4,694.233	5,184.400
GA_{CPU}	**8,722.000**	4,528.167	**4,536.000**	**5,557.000**	5,603.800	6,338.933	**6,339.000**	7,283.633
GA_{GPU}	5,722.800	3,866.900	3,994.500	1,028.800	1,102.900	3,693.733	4,656.567	6,426.267
GA_{SNS}	8,721.400	**4,554.000**	4,531.866	**5,557.000**	5,592.867	6,338.233	6,338.500	7,282.167
$GA_{SNS^{exp}}$	**8,722.000**	**4,554.000**	4,532.000	**5,557.000**	5,603.000	6,338.900	6,336.633	7,281.833
SNS	7,786.967	4,322.787	4,283.167	4,692.700	4,736.100	2,324.267	5,745.467	5,718.267
(*r*) *SNS*	8,335.000	3,886.067	3,749.886	5,281.400	5,352.100	5,522.367	5,441.133	6,307.433
(*mr*) *SNS*	8,485.567	**4,554.000**	**4,536.000**	5,504.033	5,568.167	6,156.800	6,115.933	6,899.900
(*r*, *mr*) *SNS*	8,500.533	4,486.300	4,493.367	5,499.833	5,564.300	6,177.300	6,121.567	6,874.133
SNS^{exp}	6,864.833	**4,554.000**	3,805.000	4,581.133	4,556.700	4,885.433	4,826.033	7,239.933
(*mr*) SNS^{exp}	8,709.133	4,538.066	**4,536.000**	5,550.967	5,602.100	6,339.000	**6,339.000**	7,275.867
(*r*) SNS^{exp}	8,326.367	4,331.000	4,281.081	5,282.267	5,351.100	5,507.500	5,422.600	6,282.333
(*r*, *mr*) SNS^{exp}	8,712.000	4,531.000	**4,536.000**	5,555.633	**5,604.867**	**6,339.000**	**6,339.000**	**7,284.233**
Optimal	8,722.000	4,554.000	4,536.000	5,557.000	5,605.000	6,339.000	6,339.000	7,289.000
Algorithm	WEISH18	WEISH22	WEISH30	PB5	PB6	HP1	HP2	WEING7
R_{CPU}	7,771.067	5,490.433	8,292.733	2,117.933	585.167	3,348.967	3,036.667	933,266.800
R_{GPU}	7,855.267	5,437.767	8,261.567	2,117.267	567.533	3,354.533	3,027.833	931,947.967
GA_{CPU}	9,547.667	8,795.767	1,1035.567	2,124.900	758.000	3,338.267	3,156.000	1,089,106.000
GA_{GPU}	5,075.033	7,947.533	6,886.233	1,726.300	694.767	2,893.367	2,526.367	786,536.333
GA_{SNS}	9,554.033	8,884.467	11,113.533	2,117.100	**776.000**	3,286.133	3,119.767	1,090,633.333
$GA_{SNS^{exp}}$	**9,554.767**	8,841.800	11,093.333	2,105.167	**776.000**	3,331.000	3,171.000	1,091,188.333
SNS	7,731.500	5,532.567	7,539.900	1,965.400	631.367	3,173.900	3,008.833	1,089,370.833
(*r*) *SNS*	8,783.800	6,918.800	9,267.800	2,072.633	639.333	3,320.667	3,024.433	1,082,577.333
(*mr*) *SNS*	8,944.700	7,459.233	9,835.967	2,137.300	729.967	3,400.067	3,140.867	1,042,101.667
(*r*, *mr*) *SNS*	8,959.100	7,462.367	9,842.833	2,138.433	737.100	3,400.867	3,141.633	1,040,627.667
SNS^{exp}	6,989.200	7,248.100	7,659.200	1,903.600	477.033	3,124.367	2,784.067	830,926.200
(*mr*) SNS^{exp}	9,542.967	8,829.800	11,117.767	2,136.733	775.533	**3,412.600**	3,171.333	1,093,839.667
(*r*) SNS^{exp}	8,811.367	7,379.067	9,890.200	2,054.400	656.533	3,313.467	3,021.900	1,083,721.000
(*r*, *mr*) SNS^{exp}	9,542.667	**8,831.633**	**11,132.200**	**2,138.433**	775.633	3,412.167	**3,176.500**	**1,093,953.000**
Optimal	9,580.000	8,947.000	11,191.000	2,139.000	776.000	3,418.000	3,186.000	1,095,445.000

Bold values indicate the best values

iterations, reducing the chance of falling into a local optimum (larger if the solutions remains in the same row forever). The combination of these two factors has enabled us to enhance the canonical *SNS* model considerably.

Table 3 shows that the canonical SNS algorithm obtains the lower times in 13 out of 16 instances. These good times are obtained because the SNS provides a simple structure that can maximize the efficiency of GPU architecture. SNS fills the pipe completely while the rest of versions are numerically better but do not fully exploit the GPU's power. Concerning the amount of speedup obtained, we compute this metric by dividing the time of the other algorithms with the best. The SNS speedup ranges from 1.025 to 33. In general, the greatest speedup differences are related with the GA versions.

Among the results in the *SNS* and SNS^{exp} groups, the SNS^{exp} versions obtain in general numerical results closest to the overall and even in some instances reach it. In general, the canonical versions of the two group obtain the lowest (worst) results compared with the extension versions.

To clarify the results of the last paragraph, Figs. 5 and 6 show a general picture of the similarities between the *SNS* and SNS^{exp} algorithm. These figures present the evolution of these versions for the instance *WEISH*16. In each figure, x-axis shows the number of rows in the mesh, y-axis corresponds to the percentage of the number of evaluations

Table 3 Average time of 30 runs, for small instances of MKP problem

Algorithm	SENTO02	WEISH01	WEISH02	WEISH06	WEISH08	WEISH10	WEISH12	WEISH16
R_{CPU}	1.895	0.224	0.233	0.332	0.374	0.405	0.395	0.482
R_{GPU}	7.001	1.188	1.233	1.718	1.718	1.709	1.709	1.702
GA_{CPU}	3.473	0.899	0.966	1.097	1.236	1.065	1.063	1.051
GA_{GPU}	9.627	2.108	1.988	2.814	2.928	2.755	2.748	2.697
GA_{SNS}	2.044	0.317	0.317	0.428	0.429	0.528	0.527	0.635
$GA_{SNS^{exp}}$	2.008	0.315	0.312	0.416	0.416	0.515	0.514	0.613
SNS	0.424	**0.098**	**0.106**	**0.149**	**0.148**	**0.118**	**0.118**	**0.104**
(*r*) *SNS*	**0.423**	0.108	0.115	0.152	0.155	0.121	0.120	**0.104**
(*mr*) *SNS*	0.467	0.113	0.129	0.160	0.163	0.129	0.129	0.122
(*r*, *mr*) *SNS*	0.465	0.112	0.131	0.161	0.159	0.130	0.130	0.122
SNS^{exp}	0.850	0.103	0.127	0.324	0.303	0.274	0.276	0.253
(*r*) SNS^{exp}	0.896	0.106	0.139	0.372	0.341	0.312	0.314	0.291
(*mr*) SNS^{exp}	0.851	0.105	0.125	0.330	0.301	0.274	0.277	0.254
(*r*, *mr*) SNS^{exp}	0.897	0.120	0.148	0.370	0.340	0.311	0.315	0.291

Algorithm	WEISH18	WEISH22	WEISH30	PB5	PB6	HP1	HP2	WEING7
R_{CPU}	0.551	0.611	0.764	0.252	1.325	0.180	0.250	0.668
R_{GPU}	1.707	1.702	1.916	2.848	7.067	1.517	1.510	1.406
GA_{CPU}	1.031	1.020	1.133	1.706	3.565	1.058	1.044	0.892
GA_{GPU}	2.742	2.748	3.097	4.166	9.565	2.527	2.548	2.640
GA_{SNS}	0.739	0.827	1.060	0.317	1.382	0.275	0.345	0.969
$GA_{SNS^{exp}}$	0.721	0.802	1.028	0.312	1.370	0.270	0.339	0.924
SNS	**0.096**	**0.090**	**0.101**	0.468	**0.583**	0.181	**0.149**	**0.080**
(*r*) *SNS*	**0.096**	0.091	0.103	0.476	0.593	0.183	0.150	0.082
(*mr*) *SNS*	0.122	0.126	0.154	0.490	0.599	0.194	0.158	0.144
(*r*, *mr*) *SNS*	0.122	0.127	0.154	0.496	0.607	0.194	0.158	0.143
SNS^{exp}	0.182	0.169	0.184	0.761	1.018	0.418	0.290	0.153
(*r*) SNS^{exp}	0.208	0.197	0.218	0.833	1.063	0.485	0.338	0.186
(*mr*) SNS^{exp}	0.176	0.165	0.181	0.765	1.013	0.420	0.290	0.147
(*r*, *mr*) SNS^{exp}	0.203	0.190	0.212	0.830	1.059	0.489	0.338	0.179

Bold values indicate the best values

completed, and z-axis shows the fitness value. Figure 5 presents the average fitness of all the solutions (by row) through the evolution of the *SNS* algorithm, while Fig. 6 presents the same data for SNS^{exp}. Both figures show the worst solutions marked in gray and as the fitness improves, the colors become lighter until the best solutions are reached(marked in white). We can clearly see the differences between the number of rows in the two algorithms and therefore how the evolution of the solutions is very different. The *SNS* version will improve gradually in the upper area of the grid, and as the algorithm progresses the solutions are converging prematurely or get stuck at a local optimum. On the contrary, SNS^{exp} can execute a greater number of cycles (for the same number of visited points in the search space) and have fewer solutions to evaluate so, each solution is modified extensively (higher exploitation), until it reaches the optimal or very close to it.

Additionally, in order to better compare the two main versions developed in this work, namely *SNS* and

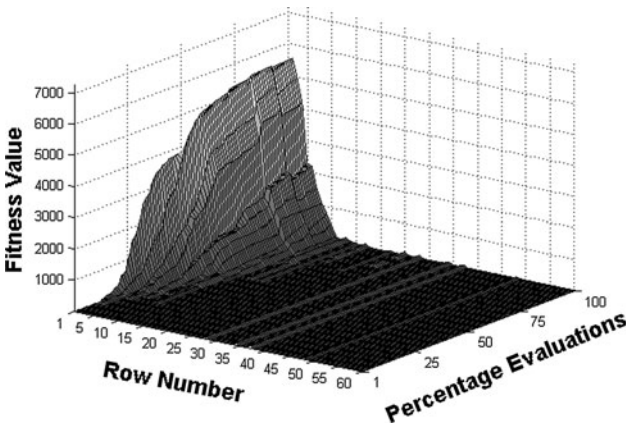


Fig. 5 Canonical SNS

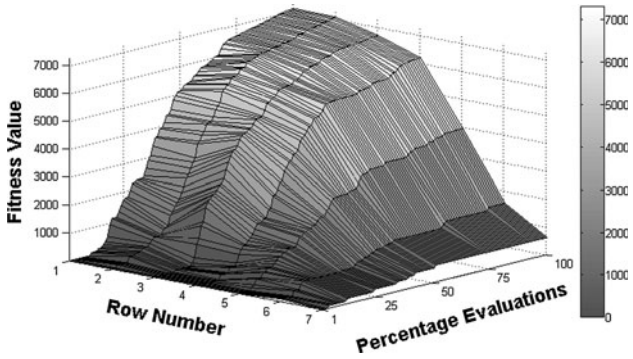
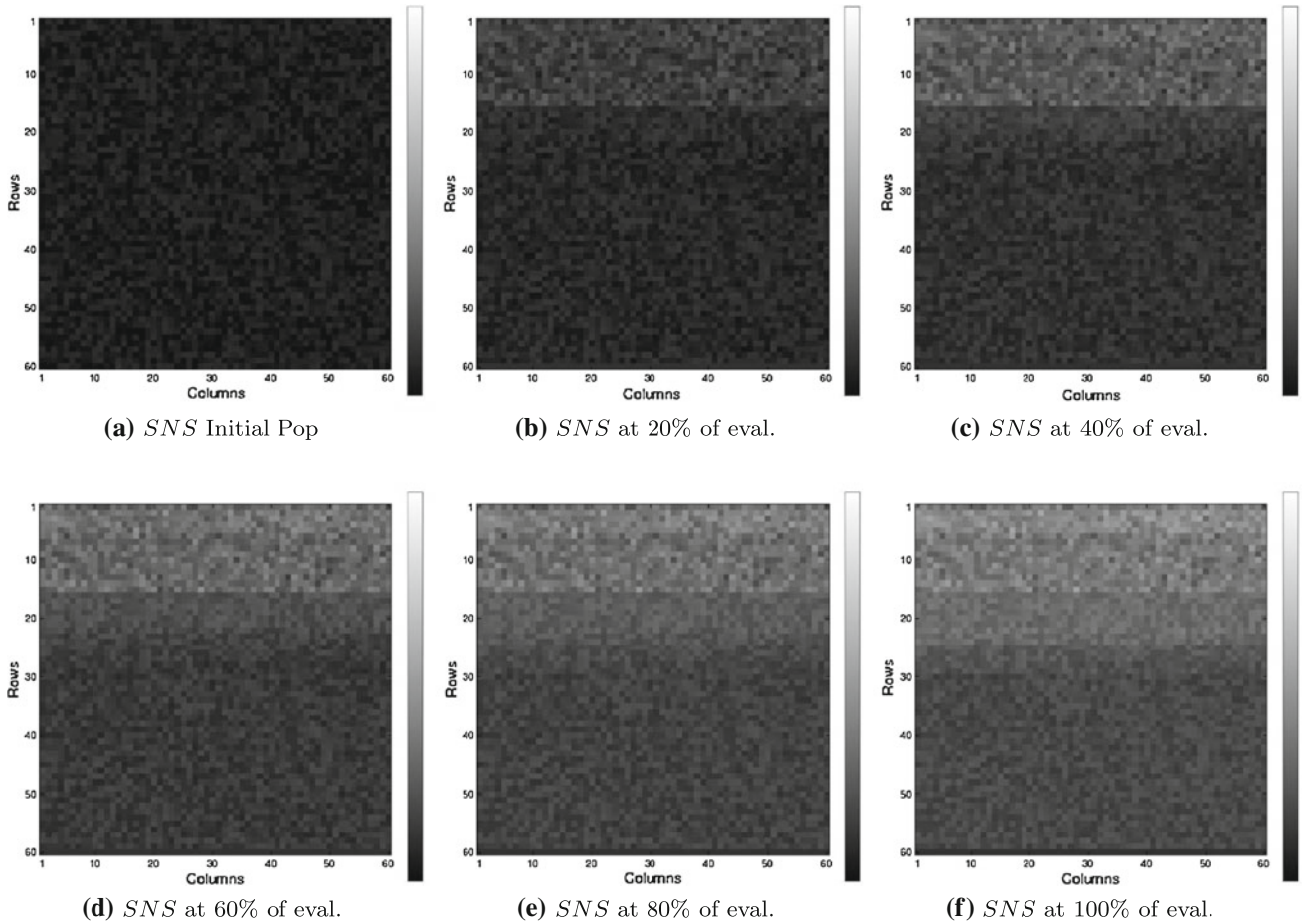


Fig. 6 Exponential SNS

SNS^{exp}, Figs. 7 and 8 displays a sequence of subfigures with the evolution of the population at six different iterations of the algorithm for the *WEISH*16 instance. Each snapshot shows mesh in which the population is arranged so that each single solution is represented by a square. The main difference between the two algorithms is the population size and how it is arranged: *SNS* uses a $k \times k$ square mesh whereas SNS^{exp} uses a $\log_2 k \times n$ layout. The color scale indicates that, the darker a solution (a square), the worse its fitness. As fitness values improve the color of the square becomes gray and the best solution is displayed as a white square. The aim of the figures is to show that solutions are getting lighter, i.e., higher (better) fitness, with the evolution of the algorithm. Comparatively, it can be seen that SNS^{exp} improves solutions very fast on the top rows, leading to near optimal solutions (almost withe squares in the meshes). In *SNS*, however, the improvements occur in a slower pace, especially in the intermediate and lower rows of the mesh.

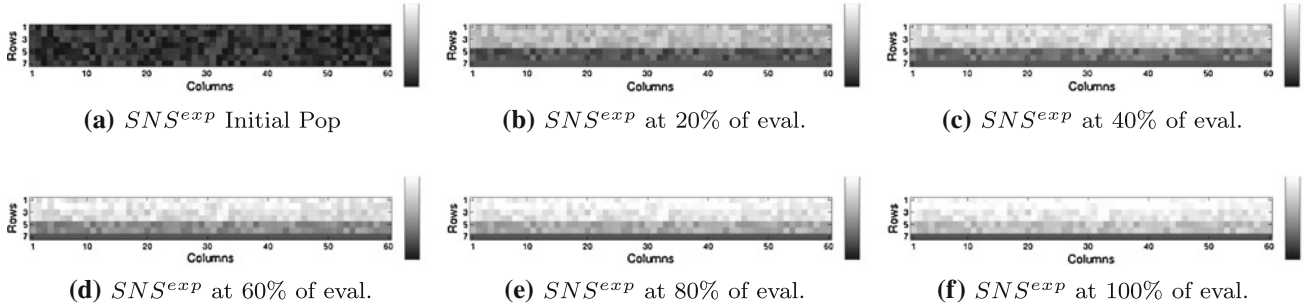
Among the GA versions, GA_{CPU} is the better algorithm with fitness values more closest to the optimum fitness. Comparing GA_{CPU} with respect the values of the *SNS* and SNS^{exp}, we can see that GA_{CPU} obtains the best values in 3 of 16 instances. For the rest of the instances, the GA_{CPU} obtains higher values than the *SNS* versions, which are far from the best optimum value. However, SNS^{exp} versions obtain similar fitness values to those of GA_{CPU} and even outperform GA_{CPU} in some cases, especially with (r, mr)



(a) SNS Initial Pop **(b)** SNS at 20% of eval. **(c)** SNS at 40% of eval.

(d) SNS at 60% of eval. **(e)** SNS at 80% of eval. **(f)** SNS at 100% of eval.

Fig. 7 Evolution of fitness value of the entire population in a canonical SNS for instance WEISH01. In the scale, the lighter the color the better (higher) the fitness



(a) SNS^{exp} Initial Pop **(b)** SNS^{exp} at 20% of eval. **(c)** SNS^{exp} at 40% of eval.

(d) SNS^{exp} at 60% of eval. **(e)** SNS^{exp} at 80% of eval. **(f)** SNS^{exp} at 100% of eval.

Fig. 8 Evolution of fitness value of the entire population in a canonical SNS^{exp} for instance WEISH01. In the scale, the lighter the color the better (higher) the fitness

SNS^{exp}. We must remember that all GA versions are based on a canonical version applied over a GPU architecture without improvements or tuning. We can say that this simple version does not produce any drastic improvement in helping to reach optimum fitness. Table 3 clearly indicates that executing any version of GA algorithm is more expensive when compared to the *SNS* and SNS^{exp} versions.

Among the results of RS implementations, we can clearly observe that these algorithms obtain an average fitness value lower (worse) than those obtained by *SNS* and SNS^{exp} versions (except in the case of canonical SNS, where the difference is minimal) and far below the optimum fitness. These results indicate that the *SNS* approach can perform an intelligent exploration of the search space. Moreover, Table 3 shows a clear difference between the times of RS implementations and *SNS* and SNS^{exp} versions that obtaining the lower times in all instances.

The results of statistical analysis have not been included by space restriction. However, the results obtained (*r*, *mr*) SNS^{exp} are significantly better with respect to GA_{GPU}, both versions of *RS*, (*mr*) *SNS*, SNS^{exp} and (*mr*) SNS^{exp}.

6.2 Medium size instances

In this section, we divide the analysis into two parts, each one focused on a particular set of instances. The best solution for each instance is not known so, we used as a reference value those found in previous related publications Chu and Beasley (1998); Vasquez and Vimont (2005); Hanafi (2009).

The first group of instances consist of 10 knapsacks with 250 and 500 elements respectively. Tables 4 and 5 presents, respectively, the average solution quality and execution time for this group of instances.

Table 4 Average fitness value of 30 runs, for medium instances of MKP problem

Algorithm	10.250.0	10.250.1	10.250.2	10.250.3	10.250.4	10.500.0	10.500.1	10.500.2	10.500.3	10.500.4
R_{CPU}	35,505.000	31,745.000	31,590.000	34,566.667	30,148.000	90,033.333	87,566.667	85,620.000	84,966.667	89,533.333
R_{GPU}	38,710.000	34,423.333	36,030.000	33,970.000	37,033.333	91,933.333	92,060.000	861,660.000	87,340.000	82,490.000
GA_{CPU}	**56,670.700**	**55,528.533**	**57,138.600**	56,169.567	**55,053.433**	96,793.933	97,281.167	97,306.900	97,062.033	95,584.433
GA_{GPU}	53,305.333	52,667.600	51,083.100	54,200.300	51,731.000	97,462.867	94,731.367	92,768.367	98,765.433	93,401.100
GA_{SNS}	49,864.267	49,698.533	48,761.300	51,302.333	49,220.633	95,538.667	96,390.367	96,059.533	95,839.700	95,839.700
$GA_{SNS^{exp}}$	50,804.333	50,421.700	49,399.967	52,315.433	50,012.367	96,364.167	96,164.833	95,965.867	97,725.400	96,725.400
SNS	35,316.667	36,836.667	41,696.667	36,853.333	38,663.333	89,780.000	87,466.667	89,833.333	86,720.000	81,822.000
(*r*) *SNS*	40,116.667	41,846.667	44,846.667	39,736.667	39,296.667	91,110.000	87,566.667	86,860.000	93,566.667	92,466.667
(*mr*) *SNS*	45,343.000	46,344.333	40,253.333	43,443.333	42,415.000	90,966.667	92,466.667	93,233.333	94,033.333	92,833.333
(*r*, *mr*) *SNS*	46,485.000	43,030.667	43,693.333	47,577.667	46,639.333	97,766.667	96,220.000	95,029.000	97,730.000	92,800.000
SNS^{exp}	47,213.000	46,953.133	46,750.200	47,759.967	45,569.867	94,301.233	98,732.900	94,621.800	95,090.867	93,399.667
(*r*) SNS^{exp}	55,190.767	54,544.800	53,724.367	56,980.467	54,309.833	103,744.500	104,950.733	104,923.700	**104,310.200**	102,697.033
(*mr*) SNS^{exp}	52,324.767	51,418.000	50,748.333	**53,592.167**	51,607.367	102,831.300	103,514.100	104,116.167	103,239.233	101,480.367
(*r*, *mr*) SNS^{exp}	55,231.400	54,547.667	53,806.133	56,886.000	54,287.400	**104,018.733**	**105,008.567**	104,796.433	104,279.967	102,792.300
Optimum	59,187.000	58,781.000	58,097.000	61,000.000	58,092.000	117,821.000	119,249.000	119,215.000	118,829.000	116,530.000

Bold values indicate the best values

Table 5 Average time (seconds) of 30 runs, for medium instances of MKP problem

Algorithm	10.250.0	10.250.1	10.250.2	10.250.3	10.250.4	10.500.0	10.500.1	10.500.2	10.500.3	10.500.4
R_{CPU}	9.307	9.148	9.307	9.146	8.974	35.466	35.465	35.542	35.442	35.482
R_{GPU}	10.098	10.097	10.098	10.098	10.097	20.181	20.180	20.181	20.181	20.192
GA_{CPU}	4.256	4.254	4.253	4.400	4.426	8.372	8.375	8.388	8.383	8.371
GA_{GPU}	16.172	16.154	16.117	16.101	16.103	30.050	30.030	30.079	30.064	30.048
GA_{SNS}	11.458	11.437	11.422	11.440	11.431	46.974	46.985	46.982	47.100	47.100
$GA_{SNS^{exp}}$	10.870	10.860	10.852	10.863	10.857	42.571	42.568	42.582	42.650	42.650
SNS	1.084	1.083	1.082	1.086	1.081	4.402	4.400	4.399	4.397	4.398
(*r*) *SNS*	1.088	1.089	1.089	1.088	1.087	4.381	4.388	4.387	4.383	4.389
(*mr*) *SNS*	2.301	2.302	2.301	2.301	2.301	9.402	9.404	9.403	9.403	9.405
(*r*, *mr*) *SNS*	2.300	2.302	2.301	2.301	2.302	9.401	9.412	9.412	9.401	9.408
SNS^{exp}	*0.388*	0.387	0.387	0.389	0.388	0.551	0.553	0.554	0.552	0.553
(*r*) SNS^{exp}	0.482	0.480	0.480	0.482	0.481	0.886	0.888	0.889	0.889	0.888
(*mr*) SNS^{exp}	**0.388**	**0.385**	**0.385**	**0.387**	**0.387**	**0.545**	**0.545**	**0.547**	**0.546**	**0.546**
(*r*, *mr*) SNS^{exp}	0.473	0.470	0.469	0.471	0.471	0.788	0.789	0.790	0.790	0.789

Bold values indicate the best values

Table 4 shows that all the algorithms not reached the best known optimum. In the instances with 250 elements, the GA_{CPU} obtain competitive results and have been overcome by very little difference in the fitness values to the family of SNS^{exp}. It is interesting to note that *SNS* and SNS^{exp} versions have obtained values below the GA_{CPU} except in the case of instance 10.250.3 where (*r*)SNS^{exp} obtains the highest value. The instances with 500 elements show that the SNS^{exp} versions reach values near to the optimal with respect to the rest of algorithms. In particular, the (*r*)SNS^{exp} and (*r*, *mr*)SNS^{exp}. According to these results, no algorithm is the clear winner. Table 5 clearly indicates that the set of SNS^{exp} are the fastest algorithms for this instance group. In particular, the (*mr*)SNS^{exp} obtain lower times than the rest of the algorithms. These results indicate that is better to work on fewer solutions with a greater number of iterations and not an other way on a GPU architecture. We see that as the number of elements increases, the SNS^{exp} group times do not increase significantly, in contrast with the times of the *SNS* versions which are four times or more larger as the number of elements increases.

Analyzing the *SNS* and SNS^{exp} versions, we can clearly see that the family of SNS^{exp} are the algorithms with

Table 6 Average fitness value of 30 runs, for medium instances of MKP problem

	30.250.0	30.250.1	30.250.2	30.250.3	30.250.4	30.500.0	30.500.1	30.500.2	30.500.3	30.500.4
R_{CPU}	32,715.000	33,173.000	31,912.033	36312.007	33,730.166	88,912.933	88,172.667	85,620.000	84,966.667	89,533.333
R_{GPU}	31,010.337	30,153.000	32,800.333	31587.500	31,215.000	82,303.000	82,009.233	83,850.100	82,899.267	80,488.900
GA_{CPU}	46,187.333	48,070.467	45,941.367	46340.300	46,621.400	96,663.133	95,872.100	91,832.500	92,045.233	92,561.133
GA_{GPU}	49,254.833	48,158.111	47,211.333	46998.661	46,868.000	93,154.847	90,745.007	88,248.100	91,702.400	92,019.900
GA_{SNS}	47,265.100	48,831.967	47,115.700	47395.233	47,479.767	95,113.913	94,881.667	92,566.133	91,223.933	91,887.600
$GA_{SNS^{exp}}$	45,851.467	49,617.900	45,560.800	45244.200	47,203.967	92,221.991	91,047.100	87,811.333	90,983.766	91,748.167
SNS	30,156.100	30,812.233	30,656.133	31453.000	32,122.833	87,800.300	87,411.113	81,811.521	81,136.100	79,152.233
(*r*) *SNS*	32,556.033	32,181.367	31,652.500	31384.033	31,115.467	82,035.167	81,347.833	82,988.900	83,251.167	92,466.267
(*mr*) *SNS*	32,455.400	33,300.333	31,248.133	31348.667	32,466.333	85,018.400	82,862.611	83,112.400	81,485.533	83,050.633
(*r*, *mr*) *SNS*	38,455.233	35,315.467	35,167.600	38154.933	36,415.100	91,414.000	92,000.100	90,588.933	91,545.800	89,333.500
SNS^{exp}	40,355.050	43,486.100	42,546.120	40154.100	41,886.441	90,152.133	95,135.100	91,311.000	91,133.011	89,152.000
(*r*) SNS^{exp}	50,458.717	**53,521.300**	51,200.167	51258.117	51,009.183	101,911.011	101,450.133	104,323.300	101,210.900	99,719.053
(*mr*) SNS^{exp}	50,113.767	50,125.010	49,868.991	47998.127	50,611.117	100,587.100	101,384.100	102,156.107	100,111.241	100,447.173
(*r*, *mr*) SNS^{exp}	**53,455.000**	51,247.067	**53,806.133**	**56886.000**	**54,287.400**	**104,018.733**	**105,008.567**	**104,796.433**	**101,243.113**	**101,032.550**
Optimum	58,842.000	58,418.000	56,614.000	56930.000	56,629.000	115,950.000	114,810.000	116,683.000	115,301.000	116,435.000

Bold values indicate the best values

Table 7 Average time (seconds) of 30 runs, for medium instances of MKP problem

	30.250.0	30.250.1	30.250.2	30.250.3	30.250.4	30.500.0	30.500.1	30.500.2	30.500.3	30.500.4
R_{CPU}	23.641	23.479	23.430	23.459	23.415	92.481	92.491	92.685	92.486	92.470
R_{GPU}	26.474	26.470	26.463	26.470	26.474	52.895	52.894	52.891	52.893	52.896
GA_{CPU}	9.838	9.850	10.068	9.848	9.836	19.777	19.782	19.834	19.815	19.829
GA_{GPU}	32.695	32.690	32.706	32.660	32.677	64.685	64.714	64.702	64.686	64.687
GA_{SNS}	26.187	26.169	26.170	26.182	26.168	108.605	108.556	108.652	108.683	108.685
$GA_{SNS^{exp}}$	24.986	24.977	24.970	24.986	24.975	98.773	98.712	98.788	99.114	98.847
SNS	2.710	2.713	2.709	2.713	2.710	11.362	11.350	11.354	11.366	11.307
(*r*) *SNS*	2.729	2.736	2.732	2.735	2.731	11.381	11.401	11.400	11.393	11.396
(*mr*) *SNS*	5.471	5.473	5.469	5.472	5.471	20.633	20.647	20.636	20.629	20.629
(*r*, *mr*) *SNS*	5.473	5.469	5.472	5.471	5.473	20.633	20.646	20.641	20.635	20.633
SNS^{exp}	0.844	0.846	0.844	0.844	0.844	1.153	1.156	1.157	1.152	1.155
(*r*) SNS^{exp}	1.017	1.018	1.017	1.017	1.016	1.917	1.934	1.934	1.925	1.906
(*mr*) SNS^{exp}	**0.843**	**0.844**	**0.843**	**0.843**	**0.843**	**1.137**	**1.139**	**1.138**	**1.135**	**1.124**
(*r*, *mr*) SNS^{exp}	0.981	0.982	0.981	0.981	0.981	1.618	1.626	1.621	1.617	1.598

Bold values indicate the best values

solutions more close to the best known optimum. The extensions help the canonical SNS and SNS^{exp} algorithms to improve the results. In general, SNS^{exp} versions have shown a stable behavior as the number of elements increases. Regarding the RS implementations, these algorithms do not outperform the solutions values of the SNS^{exp} family. Among the GA versions, only the GA_{CPU} shows competitive results against SNS^{exp} versions. These results show that the algorithmic approach fits with the architecture of the GPU. Furthermore, this indicates that increasing the number of elements does not lead to a substantial degradation in the solution's quality.

In general, we can see a statistical significance in (*r*) SNS^{exp} and (*mr*, *r*) SNS^{exp} with respect to RS_{CPU}, RS_{GPU}, and GA_{GPU} and the different SNS versions.

The second group has instances of 30 knapsacks with 250 and 500 elements respectively. Table 6 shows the average fitness values in 30 independent runs and Table 7 indicates the time execution for this group. Table 6 clearly shows that (*r*, *mr*)SNS^{exp} outperforms all the other algorithms in 9 out of 10 instances, but it does not reach the known maximum fitness value. With respect to the time, the lower times are obtained by the (*mr*)SNS^{exp} in all the instances. Furthermore, the times obtained by the algorithm (*r*, *mr*)SNS^{exp} is very similar to (*mr*)SNS^{exp}, which indicates that the (*r*, *mr*)SNS^{exp} reaches solutions very near the best optimum in a little time.

In the *SNS* and SNS^{exp} group, only the canonical version obtains fitness values that are worse than the rest of the group. Clearly, we can see that the extensions proposed (random changes, and movement of solutions between rows) have generated a significant improvement over the canonical *SNS* and SNS^{exp} base for these groups of instances. In particular, using both extensions with SNS^{exp} algorithm has shown an adaptation of the algorithm to diverse instance sizes with results very close to the best known optimum.

Regarding the RS implementations, the behavior of the results is the same as observed in the first group of instances, where the RS does not exceed the numerical and time results obtained by our approaches in any instance. Among the GA versions, (*r*, *mr*)SNS^{exp} improves all the values obtained by the GA implementations.

As a final comment of these complete groups of instances, we observe a robust behavior of the SNS^{exp} versions, in particular from the extension versions that they have obtained better results very close to best known optimum in a very short execution time.

For this group of instances, we can see statistical significance with (*r*) *SNSexp* and (*mr*, *r*) *SNSexp* with respect to the RS and GA in both versions.

6.3 Big size instances

Finally, Table 8 shows the average fitness values of each algorithm for the group of the biggest MKP instances. Table 9 presents the average execution times for this group of instances. These values are the results of 30 independent runs.

For Tables 8 and 9, we need to remark that some algorithms in the instances *GK*09, *GK*10 and *GK*11 are marked with *NA*. This mark shows that these instance sizes could not be tested by this class of algorithms on GPU. They need more memory space than that available in the graphic card. Nevertheless, this could be a line of future work, which would focus on the analysis of these

Table 8 Average fitness of 30 runs, for large instances of MKP problem

	GK09	GK10	GK10	GK11
R_{CPU}	16,748.667	55,445.100	54,007.600	85,775.900
R_{GPU}	16,788.467	55,450.967	54,016.067	85,716.133
GA_{CPU}	15,967.100	53,002.733	52,036.767	81,326.333
GA_{GPU}	15,418.500	51,589.800	51,056.533	65,364.900
GA_{SNS}	16,528.333	54,512.133	52,080.933	82,964.533
$GA_{SNS^{exp}}$	16,332.067	54,995.333	53,478.133	83,932.667
SNS	16,987.267	NA	NA	NA
(*r*) *SNS*	16,670.533	NA	NA	NA
(*mr*) *SNS*	16,907.500	NA	NA	NA
(*r*, *mr*) *SNS*	16,915.667	NA	NA	NA
SNS^{exp}	17,091.667	55,346.067	53,915.967	87,255.400
(*r*) SNS^{exp}	17,288.333	57,952.467	**56,383.300**	88,604.400
(*mr*)SNS^{exp}	**18,485.900**	57,610.967	56,235.000	87,510.033
(*r*, *mr*) SNS^{exp}	18,293.833	**57,974.230**	56,380.133	**88,666.833**
Optimum	18,806.000	58,087.000	57,297.000	95,237.000

Bold values indicate the best values

Table 9 Average time (seconds) of 30 runs, for large instances of MKP problem

	GK09	GK10	GK10	GK11
R_{CPU}	146.640	278.974	526.842	1,725.676
R_{GPU}	85.378	53.932	102.792	200.818
GA_{CPU}	30.865	20.442	36.869	71.005
GA_{GPU}	99.997	72.559	129.484	287.455
GA_{SNS}	174.156	442.671	818.155	3,575.626
$GA_{SNS^{exp}}$	158.309	314.956	569.796	1,812.432
SNS	18.629	NA	NA	NA
(*r*) *SNS*	18.744	NA	NA	NA
(*mr*) *SNS*	31.443	NA	NA	NA
(*r*, *mr*) *SNS*	31.446	NA	NA	NA
SNS^{exp}	1.771	1.970	3.073	8.327
(*r*) SNS^{exp}	2.834	3.322	5.283	12.812
(*mr*) SNS^{exp}	**1.737**	**1.942**	**2.970**	**8.160**
(*r*, *mr*) SNS^{exp}	2.384	2.762	4.374	11.146

Bold values indicate the best values

algorithms on on instance sizes never addressed in the literature.

With respect to the results, in Table 8 the best solution for each instance is not known so, we used as a reference value those found in previous related publication Vasquez and Hao (2001). This table indicates that no algorithm has been able to achieve the best optimum. According to this table, the SNS^{exp} versions obtain the highest fitness values (more closest to the optimum) in this set of experiments. In particular, (*r*, *mr*)SNS^{exp} is better in the instance GK010 and GK11, while the algorithms (*mr*)SNS^{exp} and (*r*)SNS^{exp} are better in GK09 and GK10 respectively. In addition, Table 9 clearly indicates that the (*mr*)SNS^{exp} is the faster algorithm in all the instances. With respect to the speedup, this ranges 1.01–439.

Among the *SNS* and SNS^{exp} versions, we can only conclude with the data of instance GK09, in which the results of the SNS^{exp} family has fitness values closer to the optimum than the *SNS* group. Once again, we see the repetition of results seen in previous instances groups where extension versions have better numerical behavior. Regarding the times, the SNS^{exp} group obtains lower times compared with the *SNS* family.

Among the GA versions, the difference is clear, any GA algorithm outperforms the results of an SNS^{exp} group and in the case of instance GK09 it exhibits the same behavior as the *SNS* and SNS^{exp} algorithms. Table 9 shows a clearly difference between the SNS^{exp} implementations and the GAs.

The results of RS implementations follow the same behavior with the GA versions. The RS results do not outperform any results of our approach numerically nor in time efficiency.

Concerning the results of the statistical analysis, it shows that for all instances, there is statistical significance of the two versions: (*r*) SNS^{exp} and (*mr*, *r*) SNS^{exp} with respect to both implementations of GA and RS.

6.4 Final remarks

In this section, we have presented the results of the *SNS* and SNS^{exp} versions tested over different instances size of MKP. Also, we have compared these results with those at the different implementation of RS and GA. These preliminary results show that, regarding our approaches, (*r*, *mr*) SNS^{exp} is the best algorithm in general. This algorithm generates a systematic and intelligent search through the search space, outperforming the rest of the algorithms in most of the instances.

The efficiency of the canonical *SNS* and SNS^{exp} algorithms have shown that the model is perfectly suited to the GPU architecture. However, the numerical performance has not been as good as expected. In general, all *SNS* and SNS^{exp} versions have outperformed the fitness values of the RS versions. This means that our algorithms are smart and competent when compared to these greedy ones. Regarding the GA versions, the *SNS* versions in general do not outperform the values obtained by GA_{CPU}. The results of the SNS^{exp} and GA versions were similar; however, but particularly in medium-big instances, the SNS^{exp} has improved on the results obtained by the GA versions. Moreover it is important to note that we are testing our algorithm against canonical basis of RS and GA, because our algorithm does not seek to focus on a specific problem but to solve the

problem of designing an algorithm for a GPU architecture without introducing operators or specific methods for the MKP.

In addition, we observe that adding random changes and parallel movements of solutions can provide a significant improvement in our algorithms. These changes improve the time consumed and the quality of the solution.

Concerning our approach with respect to the number of elements and knapsacks, we find that the best results are obtained by (r, mr) SNS^{exp}. This shows that our approximation supports working with different instance sizes without a huge loss in time and solution quality. Furthermore, one of the most relevant issues is the balance of communication and computation that exists in our model.

Furthermore, parallelism provides an environment where these changes are quite easy and allow a fast performance. These features combined with the two extensions have provided a greater improvement in our approach. The addition of solution movements through the rows and the random changes has outperformed the rest of the tested algorithms without achieving the best known optimum in any instance.

7 Conclusions

The work presented here is based on a new algorithm called SNS. The algorithm is executed in a GPU, a platform for highly parallel arithmetic computation. The new algorithm is inspired by systolic computing, aimed at exploiting both the search space using the systolic architecture and the intrinsic parallel nature of GPUs. We are evaluated the performance of this algorithm and their different versions over the classical NP-Problem Multidimensional Knapsack Problem.

The experimental analysis solving benchmark MKP instances from the famous OR-Library and for large instances the provided for Glover and Kochenberger (1996). We divided the instances in three groups depending the number of elements in each instance, small, medium and large. The results demonstrate that for the three groups, the exponential SNS versions and in concrete [(mr, r) SNS^{exp}] obtains good numerical solutions with a reduced execution times respect the other algorithms. In these scenarios, we observe that the canonical SNS do not obtain good results, possibly, because the number of iterations with respect to the exponential versions is less.

The scalability analysis shown indicates that the [(mr, r) SNS^{exp}] is the algorithm that generates best mean solutions in general taking into account the results in the different instance sizes.

The main lines for future work include exploring the role of each interacting element of the SNS version and in those versions that obtain better results, and also to improve the algorithmic proposal in order to analyze larger or different scenarios. Regarding the first issue, it would be interesting to evaluate the specific contribution of each cell, especially when the algorithm provokes the local changes, in order to get useful information about the changes and the contribution to improving, both globally and locally, solutions. On the other hand, using other heuristics by group or defining sub-population is also an interesting extension of the proposed SNS algorithms. Future work may also be centered on studying the applicability of SNS for solving other problems by using the strategies analyzed in this work.

Acknowledgments Pablo Vidal acknowledges continuous support from the University of Patagonia Austral. The work of Francisco Luna and Enrique Alba has been partially funded by the Spanish Ministry of Science and Innovation and FEDER under contract TIN2011-28194 (the roadME project). Francisco Luna also acknowledges support from TIN2011-28336.

References

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

Alba E, Dorronsoro B (2008) Cellular genetic algorithms, operations research/computer science interfaces, vol 42. Springer, Heidelberg

Alba E, Vidal P (2011) Systolic optimization on gpu platforms. In: EUROCAST (1), pp 375–383

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, lecture notes in computer science, vol 956. Springer, Berlin, pp 109–126

Chu P, Beasley J (1998) A genetic algorithm for the multidimensional knapsack problem. J Heuristics 4:63–86

CUDA (2007) NVIDIA CUDA Compute Unified Device Architecture—Programming Guide

Gavish B, Pirkul H (1986) Computer and database location in distributed computer systems. IEEE Trans Comput 35(7): 583–590

Gilmore PC, Gomory RE (1966) The theory and computation of knapsack functions. Oper Res 14(6):1045–1074

Glover F, Kochenberger GA (1996) Critical event tabu search for multidimensional knapsack problems. metaheuristics: the theory and applications. Kluwer Academic Publishers, Boston, pp 407–427

Gottlieb J (2001) On the feasibility problem of penalty-based evolutionary algorithms for knapsack problems. In: Applications of evolutionary computing, lecture notes in computer science, pp 50–59

Grama A, Karypis G, Kumar V, Gupta A (2003) Introduction to parallel computing, 2nd edn. Addison Wesley, Pearson

Hanafi CWS (2009) New convergent heuristics for 0-1 mixed integer programming. Eur J Oper Res 195(1):62–74

Hanafi S, Wilbaut C (2011) Improved convergent heuristics for the 0-1 multidimensional knapsack problem. Ann Oper Res 183:125–142

Hwang J, Park S, Kong IY (2011) An integer programming-based local search for large-scale multidimensional knapsack problems. Int J Comput Sci Eng 3(6):2257–2264

Ke L, Feng Z, Ren Z, Wei X (2010) An ant colony optimization approach for the multidimensional knapsack problem. J Heuristics 16:65–83

Kung HT (1979) Let's design algorithms for vlsi systems. In: Proceedings of the conference on very large scale integration: architecture, design, fabrication, pp 65–90

Lagae A, Lefebvre S, Drettakis G, Dutré P (2009) Procedural noise using sparse Gabor convolution. ACM Trans Graph (Proceedings of ACM SIGGRAPH 2009) 28(3):54:1–54:10

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

Shih W (1979) A branch and bound method for the multiconstraint zero one knapsack problem. J Oper Res Soc 30(4):369–378

Thomas DB, Howes L, Luk W (2009) A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In: Symposium on Field Programmable Gate Arrays, pp 63–72

Vasquez M, Hao JK (2001) A hybrid approach for the 01 multidimensional knapsack problem. In: In Proceedings of the international joint conference on artificial intelligence 2001, pp 328–333

Vasquez M, Vimont Y (2005) Improved results on the 0-1 multidimensional knapsack problem. Eur J Oper Res 165:70–81

Zhang X, Liu Z, Bai Q (2012) A new hybrid algorithm for the multidimensional knapsack problem. In: Bio-inspired computing and applications, lecture notes in computer science, pp 191–198

Zhou Q, Luo W (2010) A novel multi-population genetic algorithm for multiple-choice multidimensional knapsack problems. In: Cai Z, Hu C, Kang Z, Liu Y (eds) Advances in computation and intelligence, lecture notes in computer science, vol 6382. Springer, Berlin, pp 148–157