Information Sciences 293 (2015) 35–58

Contents lists available at ScienceDirect

Information Sciences

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

A high performance memetic algorithm for extremely high-dimensional problems

Miguel Lastra[a,*], Daniel Molina[b], José M. Benítez[c]

[a] Depto. Lenguajes y Sistemas Informáticos, E.T.S. Ingeniería Informática y Telecomunicación, CITIC-UGR, iMUDS, Universidad de Granada, Spain
[b] Depto. Ingeniería Informática, E.S. Ingeniería, Universidad de Cádiz, Spain
[c] Depto de Ciencias de la Computación e Inteligencia Artificial, E.T.S. Ingeniería Informática y Telecomunicación, CITIC-UGR, iMUDS, Universidad de Granada, Spain

ARTICLE INFO

Article history:
Received 3 August 2013
Received in revised form 3 August 2014
Accepted 12 September 2014
Available online 22 September 2014

Keywords:
Memetic algorithm
Optimization problem
GPU
CUDA

ABSTRACT

Throughout the last years, optimization problems on a large number of variables, sometimes over 1000, are becoming common. Thus, algorithms that can tackle them effectively, both in result quality and run time, are necessary. Among these specific algorithms for high-dimensional problems, memetic algorithms, which are the result of the hybridization of an evolutionary algorithm and a local improvement technique, have arisen as very powerful optimization systems for this type of problems. A very effective algorithm of this kind is the MA-SW-Chains algorithm. On the other hand, general purpose computing using Graphics Processing Units (GPUs) has become a very active field because of the high speed-up ratios that can be obtained when applied to problems that exhibit a high degree of data parallelism.

In this work we present a new design of MA-SW-Chains to adapt it to the GPU-based massively parallel architecture. The experiments with the new GPU memetic technique, compared to the original sequential version, prove that the results are obtained with the same quality but with a reduction of the run time of two orders of magnitude. This great improvement in computing time makes our proposal suitable for future optimization problems with a dimensionality several orders of magnitude greater than current high-dimensionality standards (i.e. problems with millions of variables). The remarkable run time reduction comes at the cost of a higher design and implementation overhead compared to the CPU-based single-threaded counterpart.

1. Introduction

Evolutionary Algorithms (EAs) [3] are meta-heuristic techniques that have arisen as very good algorithms for optimization problems. These algorithms can be applied to a variety of real-world optimization problems because they do not require specific information about the problem at which they are targeted, obtaining very good results in optimization problems with computing and/or run time restrictions. Furthermore, they have shown considerable success in dealing with problems characterized by complex solution spaces.

To improve the effectivity of the search process an EA can be hybridized with a local improvement method. Memetic Algorithms (MAs) [35,36] are extensions of EAs with a separate local search process (LS) that improves the optimization process [29]. MAs are simple but flexible and powerful algorithms [32,41] that find high-quality solutions in many real-word problems [12,25,53].

* Corresponding author. Tel.: +34 958246144.
E-mail addresses: mlastral@ugr.es (M. Lastra), daniel.molina@uca.es (D. Molina), J.M.Benitez@decsai.ugr.es (J.M. Benítez).

http://dx.doi.org/10.1016/j.ins.2014.09.018
0020-0255/© 2014 Elsevier Inc. All rights reserved.

Nowadays, it is very common to face real-world problems which require optimizing a rising number of variables. Typical optimization problems involve tens of variables. However, in research fields that have a growing interest it became obvious that a larger number of variables is required, leading to high-dimensional optimization problems: in data mining problems, by the huge size of available data (such as clustering in text analysis [31,6]); biomedical problems as in DNA and molecular simulation [27,65,4]; networking problems [54,18]. In fact, the requirement of high-dimensional algorithms is increasing and currently it is not unusual to tackle problems represented by a very high number of dimensions (over 10^6) like feature selection techniques [24], molecular simulation [65] or forecasting [38]. With the appearance of big data the number of huge-scale optimization problems (above dimension 10^8) is growing, as complex simulation [13], data mining [1], quantum chemistry [46], spectroscopy analysis [44], geophysical analysis [7], drug discovery [17], genomic studies [47], etc.

Optimization of high-dimensional problems, also called large-scale optimization, implies a higher complexity in EAs, not only because many techniques are not suitable for higher dimensions but because the error increases with dimensionality. EAs try to achieve a compromise between accuracy and invested effort (in terms of number of evaluations or run time), thus, when applied to high-dimensional problems, EAs do not always produce good results in an affordable time.

High-dimensional optimization problems have several characteristics that makes them difficult to solve. First, when the dimensionality increases the search domain size increases exponentially, while the number of evaluations that can be performed usually increases lineally as computational resources are added. Also, techniques that use gradient information (or approximation, such as Quasi-Newton techniques), present problems in high-dimensional systems [55]. Additionally, proximity and neighborhood relationships are difficult to assess because, although it is widely used, the Euclidean distance may not be an appropriate measure in high-dimensional problems [1].

In recent years different EAs specifically designed for large-scale optimization have been proposed to tackle the aforementioned issues. In particular, MAs have proven to be very competitive in this field because they are based on stochastic algorithms that do not require any gradient information and because the use of LS methods may improve the performance of the algorithm by quickly exploring around the best solutions (especially useful in high-dimensions). Unfortunately, applying traditional LS techniques to high-dimensional problems increases the computational intensity because of the high number of variables and the additional fitness function evaluations added by the local search process. MA-SW-Chains [34] is a specially interesting MA because it allows a greater computational intensity to be applied only to the most promising solutions until they do not improve, while maintaining a fixed effort ratio (ratio of number of evaluations) invested in the LS method. With this combination the algorithm exhibits a good trade-off between exploration and exploitation in high-dimensional problems. The fact that MA-SW-Chains was the winner of the large-scale global optimization session in the IEEE Congress on Evolutionary Computation in 2010 proves it is a very competitive algorithm in this field.

However, MA-SW-Chains is a sequential algorithm, where the process is run by one CPU and each step of the algorithm is run after the previous one. On the other hand, parallel algorithms are able to perform several computations at the same time and can greatly reduce the required processing time. For high-dimensional optimization problems this kind of algorithms are specially interesting because they could reduce the high computing time due to the huge amount of variables [2,43,45].

One parallel architecture that has obtained very good results for certain kind of parallel algorithms is the one provided by Graphics Processing Units (GPUs). General Purpose computation on GPUs (GPGPU) allows algorithms to perform parallel computations over different data using the general purpose computing capabilities of modern GPUs. Recently, several parallel EAs for optimization using GPUs have been published, as Particle Swarm Optimizations (PSO) [11] or Differential Evolution (DE) [26,20].

In this work we propose a GPU-based design and implementation of the MA-SW-Chains algorithm. The objective we have sought was to improve the performance of this algorithm to make it adequate for huge-dimensional problems without reducing the quality of the results.

The GPU version of MA-SW-Chains is compared to the original version over several well-known optimization problems which are frequently used in different high-dimensional benchmarks. This empirical study includes both the run time and error measure analysis. Furthermore, the scalability achieved using GPUs is also studied and experiments are not limited to usual high-dimensional values in the order of 1000 variables and reach dimensionality values up to 3,000,000. The goal was to analyze if GPU-based implementations could be ready even for future high-dimensional problems. The results that were obtained show that GPUs allow tackling optimization problems with dimensionality levels not affordable with the original MA-SW-Chains version in a reasonable time.

This paper has the following structure: in Section 2, GPGPU programming is introduced, in Section 3, a brief review of EAs designed for high-dimensionality, paying special attention to memetic algorithms, is shown. In Section 4 the original MA-SW-Chains algorithm is explained. In Section 5, our parallel algorithm is described, remarking the differences with respect to the CPU version. In Section 6, the experiments using the GPU and CPU version are presented. Finally, in Section 7 the results are analyzed and in Section 8 the main conclusions and future work are detailed.

2. GPGPU programming

The GPGPU computation discipline (General Purpose computation on Graphics Processing Units) has been a very active research topic in the last years, especially since computing frameworks such as CUDA [14] or openCL [42] were introduced. These platforms have allowed using the great computing capabilities of modern Graphics Processing Units for general purpose problems by using extensions of high level programming languages. CUDA is a computing platform provided by Nvidia [39] which allows applications to be developed on Nvidia GPUs using a subset of C/C++ (Fortran and other programming languages are also supported) with some extensions that provide access to the many-core processing units which are part of any modern GPU.

The GPU hardware architecture is exposed to developers as a set of multiprocessors composed of many computing cores. As an example, the Nvidia Tesla K20m GPU offers 2496 cores grouped in 13 multiprocessors (SMXs) which yields 192 processors per SMX. These computing cores can run many instances (threads) of the same program (kernel) in parallel where each thread usually accesses a fraction of the input data and produces a part of the results. The main limitation of this computing schema is that threads are run in groups of 32 threads (warp) where code execution divergences should be avoided to prevent serialization. Problems which exhibit a high degree of data parallelism are suitable for this type of architecture. Threads are also grouped into blocks, where each block is always scheduled to run on the same SMX which allows its threads to collaborate using a high speed shared memory area and synchronization barriers. The set of blocks required to perform the whole computation is called grid. Threads, and blocks can be organized in 1D, 2D or 3D in order to efficiently map the problem being resolved. A scheme of the threads, blocks and multiprocessors paradigm is shown in Fig. 1.

Any problem to be solved using a GPU requires to be decomposed into a large set of independent tasks which will run the same code (kernel) but applied to different data subsets (data parallelism). Each of these tasks will be carried out by a thread and by running as many threads as possible in parallel, the GPU produces the result in a fraction of the time required by a sequential implementation. The key is to design the software to use the maximum number of parallel threads the GPU hardware is able to handle at a time but this is not trivial work. GPU-based software design decisions have a very important impact on the performance obtained.

In order to reach the peak performance of this type of computing architectures, both thread level and instruction level parallelism must be maximized. Taking into account the large amount of computing cores available, and how memory access latencies affect computational speed on any platform, a really large amount of threads should be able to run in parallel. For this reason, the GPU hardware provides a large number of registers per SMX to prevent costly context switches produced when memory accesses are required. On the other hand, like in any other computing platform, it is necessary to keep the computing pipelines of each core as full as possible which requires a sufficient workload per thread and also a sufficient instruction independence to allow several instructions to be run in parallel on each core.

Finally, coalesced thread memory accesses is also one of the key aspects to reduce the impact of memory operations on the total computation time. Coalesced memory access patterns usually require a SOA (structure of arrays) memory scheme and not AOS (array of structures) data schemes which are more common when designing programs to be run on CPUs (unless vectorization instructions are used).

3. High-dimensional optimization

High-dimensional optimization problems are becoming a field drawing increasing attention. The dimension of the optimization problems has been growing along last years. In order to boost and direct the research on effective methods for this task, in recent years, several large-scale optimization competitions have been proposed with experimental frameworks for high-dimensional problems (up to dimension 1000). Arguably, the algorithms presented and evaluated at these competitions represent the state-of-the-art of the field. In this section we will review the most representative methods. This includes the best proposals in the competitions of the IEEE Congress on Evolutionary Computation in 2008 [49], 2010 [50] (also used in 2012) and the proposals presented in the Special Issue on Large Scale Continuous Optimization Problems. Additionally, a short review of GPU-based EAs is also presented. We focus just on GPU-based proposals, since the literature in general parallel EAs is rather extensive, but the approach of other kind of parallel architectures —i.e. multicore machines or distributed clusters— is quite different from the GPUs.

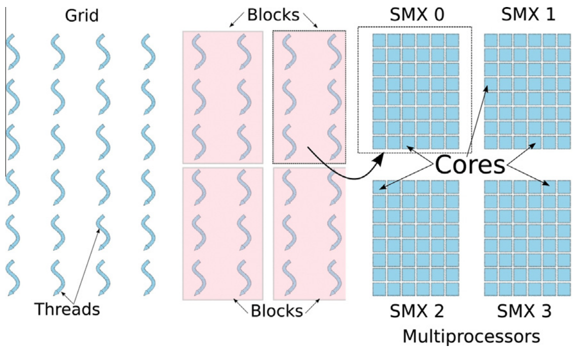

Fig. 1. Threads, blocks and multiprocessors.

3.1. Main algorithms in high-dimensional benchmarks

In general, DEs are specially adequate for high-dimensional problems and many specific versions have been proposed [16]. One interesting DE for high-dimensional problems is jDEdynNP-F [9]. Classic DEs generate new solutions as indicated by Eq. (1), where $i \in \{1, \ldots, popsize\}$, and r_2, r_3 are random values in the range $[0,1]$:

$$\begin{aligned} v_i^g &= x_i^g + F \cdot (x_{r_2}^g - x_{r_3}^g) \\ t_i^g &= \begin{cases} v_i^g, & \text{if } rand(0,1) \leqslant R \\ x_i^g, & \text{otherwise} \end{cases} \\ x_i^{g+1} &= \begin{cases} t_i^g, & \text{if } fitness(t_i^g) \text{ is better than } fitness(x_i^g) \\ x_i^g, & \text{otherwise} \end{cases} \end{aligned} \tag{1}$$

DEs create new individuals by the addition of a fraction (parameter F) of a vector difference between two random individuals. In general DEs are very sensitive to parameter F and also to parameter CR (ratio of variables changed in each crossover operation). This latter parameter, CR, is also a very important and in separable functions a low value is recommended, but in non-separable functions, large values are preferred.

$$\begin{aligned} F_i^{g+1} &= \begin{cases} 0.1 + rand_1 \cdot 0.9, & \text{if } rand_2 < 0.1 \\ F_i^g, & \text{otherwise} \end{cases} \\ CR_i^{g+1} &= \begin{cases} rand_3, & \text{if } rand_4 < 0.1 \\ CR_i^g, & \text{otherwise} \end{cases} \end{aligned} \tag{2}$$

jDEdynNP-F is characterized by two aspects: the self-adaptation of the F and CR parameters, as these values are introduced into the individuals of the population according to Eq. 2; and a decreasing population size that enforces exploration in initial stages and exploitation in latter stages. Although in high-dimensional problems the search domain size increases, generally, a larger population does not improve the results. The reduction scheme that is applied is very simple: the population is halved a fixed number of times, and the selection is made by binary tournament. In 2010 a new version was proposed [10] where the last iterations are invested in improving the best candidate (using the best solution of the crossover operator). This algorithm obtains good results, especially when the best solution is improved to obtain more accuracy and proves EAs can tackle high-dimensional problems with a reduced population size (although the results may be affected by the initial population size). Other several DEs were proposed with very good results, such as SOUPDE [59], GaDE [63], or MOS [28].

Cooperative co-evolutionary (CC) algorithms are algorithms that combine several techniques applied on the same population, obtaining good results in high-dimensional problems, such as MLCC [62] or EOEA [58]. These algorithms show that a good combination of different simple algorithms can improve the results.

Estimation of distribution algorithms (EDAs) are algorithms that use a distribution function to create several individuals, and this function is adapted to create better solutions. The most known EDA is CMA-ES [22]. Although CMA-ES was not initially suitable for high-dimensional problems, its separable version is better than previous CC techniques [40]. There are other EDAs, such as LSEDA-gl [56] or MUEDA [57], but they do not produce better results than CMA-ES.

However, the algorithms that have obtained the best results in the aforementioned competitions are MAs: Points Multi-PSO + Harmony [64] uses a PSO and a harmony algorithm as LS; MOS [28] combines the application of a DE with MTS-LS1 [52] which is an LS suitable for high-dimension. MOS obtains good results because its LS method does not require a high intensity.

One interesting MA is MA-SW-Chains [34], the winner of the 2010 competition, which combines a genetic algorithm with a Solis Wets search method [48] (it is explained in detail in Section 4). This algorithm combines a relative small population, and iteratively improves the best current solution (as jDEdynNP-F). The LS mechanism applies evaluations to the most promising solutions. This feature is interesting because it allows a general local search technique to be used in high-dimensional problems. Other MAs such as MOS [28] use a specific LS method for high-dimensional environments, MTS-LS1 [52], that does not explore all variables at the same time. The Solis Wets method is a general and fast search algorithm with good behavior while other more complex LS methods, such as CMA-ES [34], produce worse results in high-dimensional problems (especially in terms of computing time).

To summarize, the two better algorithms in these competitions, MA-SW-Chains (in 2010) and MOS (2012, 2013) are MAs that alternate an EA (genetic algorithm in the first case, DE in the second one) with an LS method (classical or specific for high-dimensional problems). We have chosen MA-SW-Chains over MOS to develop a GPU-based MA because MA-SW-Chains was designed as a general optimization algorithm which uses all the variables during all its phases and not just a subset of them.

3.2. Parallel EAs using GPU

In the parallel metaheuristics field, GPUs have been mainly used for accelerating the fitness function evaluation, and in population based algorithms, to process data of multiple solutions at the same time. In particular, the use of GPGPU takes advantage of the SIMD parallel model to reduce the time required to compute one generation [23,26]. The computing power supplied by the many-core architecture of GPUs can also be used to implement other metaheuristic models, such as distributed population models. EAs have been the preferred metaheuristic parallelized on GPUs, in combination with a parallel fitness evaluation for EAs [61,66].

Proposals of GPU implementations of other metaheuristics have also been recently presented, such as the fine-grain parallel fitness evaluation in [8], and parallel implementations of different EAs as Ant Colony Optimization (ACO) [5], PSO [11], or DE [26,20]. A more detailed review of parallel algorithms is included in the following work [2]. However, none of those proposals have addressed high-dimensional problems and are not suitable for the kind of problems considered in this paper.

4. MA-SW-Chains

In MAs, local search is used to search around the most promising solutions. As the local region extension increases with the dimensionality, high-dimensional problems require a high number of evaluations during each LS application, called LS intensity. MA-SW-Chains [34] is an MA specifically designed to adapt the LS intensity, exploiting with higher intensity the most promising individuals. This section briefly describes the original MA-SW-Chains, which is explained in detail in [34].

The main idea of MA-SW-Chains is to adapt the LS intensity applying the LS several times over the same individual, with a fixed LS intensity, and storing its final parameters, creating LS chains [33]. Using these LS chains, an individual previously improved by an LS invocation may later become the initial point of a subsequent LS application. The final strategy parameter values achieved by the previous step are used as the initial ones in the following application. Thus, this algorithm alternates the population algorithm, a steady-state genetic algorithm, with the application of the local search. Fig. 2 shows the general scheme and in the following sections each step is described in more detail.

4.1. Steady-state MAs

In steady-state genetic algorithms (SSGAs) [60] one or two offspring are produced in each generation. Parents are selected to produce offspring and then a decision is made to select which individuals in the population will be replaced by the new offspring. SSGAs are overlapping systems because parents and offspring compete for survival. Their replacement strategy is to substitute the worst individual only if the new individual is better. SSGAs are elitist (solutions can only be forced out of the population by better ones) and they allow the results of LS to be maintained in the population.

The SSGA applied was specifically designed to promote high population diversity levels by means of the combination of the $BLX-\alpha$ crossover operator [19] with a high value for its associated parameter ($\alpha = 0.5$) and the *negative assortative mating* strategy [21]. Diversity is favored as well by means of the BGA mutation operator [37].

4.2. Local search method

The Local Search strategy used is the classic *Solis and Wets'* algorithm [48], a randomized hill-climber with an adaptive step size. An increment d is randomly created with a normal distribution defined by a standard deviation ρ. If either $x + d$ or $x - d$ is better than current solution x, a move is made to the better point and a success is recorded. Otherwise, a failure is recorded.

The search step size is defined by its strategy parameter ρ, and it can be adapted very quickly. After 5 consecutive successful movements, ρ is increased to raise the step size of the search process. After 3 consecutive failures, ρ is decreased to focus the search. In our experiments we have applied an initial $\rho = 0.2$.

```
1. Generate the initial population.
2. Perform the steady-state GA throughout $n_{frec}$ evaluations.
3. Build the set $S_{LS}$ with those individuals that potentially may be refined by LS.
4. Pick the best individual $c_{LS}$ in $S_{LS}$.
5. If $c_{LS}$ belongs to an existing LS chain then
6.     Initialize the LS operator with the LS state stored together with $c_{LS}$.
7. Else
8.     Initialize the LS operator with the default LS state.
9. Apply Solis Wets' algorithm to $c_{LS}$ with an LS intensity of $I_{str}$ (Let $c^r_{LS}$ be the resulting individual).
10. Replace $c_{LS}$ by $c^r_{LS}$ in the steady-state GA population.
11. Store the final LS state along with $c^r_{LS}$.
12. If (not termination-condition) go to step 2.
```

Fig. 2. Pseudocode algorithm for the proposed MACO model.

4.3. Local search chains

In steady-state MAs, individuals may reside in the population during a long time. This circumstance allows these individuals to become starting points of subsequent LS invocations. Local Search Chains based strategies allow the final configuration reached by the former step (strategy parameter values, internal variables, etc.) to be used as an initial configuration for the next application. This allows the LS algorithm to continue under the same conditions that were reached when the LS operation was halted, providing an *uninterrupted connection between successive LS invocations*, i.e., forming an *LS chain*.

Two important aspects that were taken into account for the management of LS chains are:

- Every time the LS algorithm is applied to refine a particular chromosome, a fixed LS intensity should be considered for it, which is called *LS intensity stretch* (I_{str}). An LS chain formed throughout n_{app} LS applications which started from solution s_0 will return the same solution as the application of the continuous LS algorithm to s_0 employing $n_{app} \cdot I_{str}$ fitness function evaluations.
- After the LS operation, the parameters that define the current state of the LS process are stored along with the final individual reached (in the steady-state GA population). When this individual is selected to be improved, the initial values for the parameters of the LS algorithm are available. In the case of the *Solis Wets*' algorithm, the following parameters are stored: success and failure numbers, and the *bias* and ρ parameters.

4.4. LS application

The results obtained from an MA depend on the fitness function evaluation ratio assigned to each of the components (EA for exploration and LS method for exploitation). We define $r_{L/G}$ as the ratio of evaluations used by the LS method with respect to the total number of evaluations. Our algorithm computes n_{frec} to ensure (Step 2 in Fig. 2) that this $r_{L/G}$ ratio is always fulfilled. Without this strategy, the application of the LS method in high-dimensional problems could consume most of the maximum number of allowed fitness evaluations. For this work $r_{L/G}$ was set to 0.5.

One goal of the LS application is to favor the enlargement of those LS chains that are showing promising fitness improvements in the best current search areas represented in the SSGA population. In addition, encouraging the activation of innovative LS chains with the aim of refining unexploited zones, whenever the current best ones may not offer profitability is also desirable. The criterion to choose the individuals that should undergo LS is specifically designed to manage the LS chains in this way (Steps 3 and 4 in Fig. 2):

1. Build the set of individuals in the steady-state GA population, S_{LS} that fulfills:
   (a) They have never been optimized by the LS algorithm, or
   (b) They previously underwent LS, obtaining a fitness function value improvement.
2. If $|S_{LS}| = 0$, then apply the Solis Wets algorithm to a randomly selected individual.

With this mechanism, when the steady-state GA finds a new best individual, it is improved by the LS as soon as possible. Furthermore, the best performing individual in the steady-state GA population will always undergo LS whenever it improves the current best fitness value.

5. GPU-based MA-SW-Chains for extremely high-dimensional problems

In general, there is no automatic way to transform a serial algorithm into a truly efficient GPU algorithm. On the contrary, a careful redesign of the algorithm, a well thought design of data structures, memory access patterns and computation threads is required. This is also the case for the MA-SW-Chains algorithm.

The two natural sources of parallelism of this algorithm are:

- the number of variables of each individual (main parallelism source)
- the number of individuals in the population (secondary parallelism source).

The general scheme that has been designed to adapt the MA-SW-Chains algorithm to GPU architectures and to exploit these parallelism possibilities is shown in Fig. 3. All the steps of the original algorithm have been parallelized and although the basic structure of the original algorithm is kept unmodified, the role of the CPU program gets relegated to launching parallel GPU kernels in the right sequence. All the modules that are run on the GPU are prefixed with *gpu* in Fig. 3.

This section explains in detail how each step has been parallelized. It is organized as follows: Section 5.1 presents the GPU-based parallelization of the fitness function evaluation process, Section 5.2 presents the adaptation of the crossover operation to the GPU architecture and Section 5.3 deals with the optimization of the local search process. Finally, Section 5.4 explains how the random number generation process has been implemented and Section 5.5 illustrates the population sorting scheme that has been designed.

The source-code is available at http://dicits.ugr.es/software/GPU-MA-SW-CHAINS/.

```

gpuEvalutePopulation(population);
gpuSortPopulation(population);

for (int i = 0; i < totalEvaluations; i++) {
  //Crossover
  for (c = 0; c < (crossoverEvaluations/parallelCrossoverN); c++) {
    gpuCrossover(poputation,parallelCrossoverN);
    gpuEvaluateGeneratedIndivuals(poputation,parallelCrossoverN);
    gpuSortPopulation(population);
  }

  //Solis Wets local search
  for (s = 0; s < localSearchEvaluations;s++) {
    p=chooseIndividual(population);
    gpuChangeIndividualPlus(p);
    gpuEvaluateIndividual(p);

    if (hasImproved(p)) {
      gpuIncrementBias();
      recordSuccess();
    }
    else {
      gpuChangeIndividualMinus(p);
      gpuEvaluateIndividual(p);

      if (hasImproved(p)) {
        gpuDecrementBias();
        recordSuccess();
      }
      else
        recordFailure();
    }
    if (success)
      gpuReplaceInPopulation(population,p);

    adjustParameters();
  }
}
```

Fig. 3. Computational scheme.

5.1. Fitness function evaluation

In order to compute the fitness function value for a certain individual, the operations associated to each dimension or variable have to be mapped to the threads and thread blocks computational schema explained in Section 2. This mapping should maximize both the thread and instruction level parallelism.

In our system, each thread block is assigned the processing of a set of dimensions of each individual. Within each block, each thread processes several variables (bucket) to provide enough workload per thread. To keep memory accesses coalesced, each thread work bucket processed is composed of interleaved elements to make the thread warps access consecutive elements. This means each block processes

$$threadsPerBlock \times bucketSize$$

variables and that thread i processes the elements with indices:

$$i, (i + bD), (i + 2 \times bD), \ldots, (i + (bucketSize - 1) \times bD)$$

where bD represents the number of threads per block (block dimension). The computation over these elements generates the thread-local contribution to the final result.

In Fig. 4 a scheme of the computation is shown where each block is composed of four threads ($bD = 4$) and each thread processes a bucket of two elements (threads are represented using black rectangles). Fig. 4 also illustrates that to account for the second source of parallelism, a 2D grid of thread blocks is launched where the second coordinate identifies the individual being processed by each block.

Each thread accumulates the partial result of the fitness function value corresponding to the elements included in its work bucket. After all the threads of each block have finished the computation step, a reduction step is performed inside each

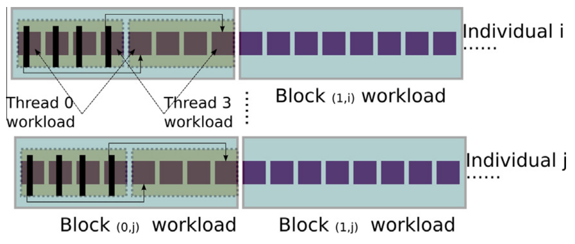

Fig. 4. Fitness function processing.

block using shared memory (all these operations are performed by the same kernel). This reduction step usually consists in adding the partial results produced by each thread using the parallel scheme that is shown in Fig. 5.

The first thread of each block, therefore, computes the block-local result of the summation when the process finishes. In our work we have implemented the completely unrolled version of the aforementioned reduction code.

Finally, to compute the fitness value associated to each individual another reduction operation is performed with the partial results obtained by each block. This final reduction operation is implemented using a dedicated kernel using the **`Thrust`** library [51] provided by Nvidia.

The assignment of several elements of each individual to each thread (using unrolled loops) gears towards increasing the instruction level parallelism and increasing the arithmetic intensity (more computations per memory write operation). The high dimensionality and the processing of several individuals in parallel provides the thread level parallelism.

In addition to these general design considerations, parallelizing and optimizing each fitness function requires considering the intrinsic characteristics of each particular function.

5.2. Crossover

The scheme used to redesign the crossover operation is very similar to the one used in Section 5.1 except for the fact that each thread processes a bucket of pairs of elements (one from each parent) and that each thread needs to write a result to memory. Fig. 6 provides a graphic representation of the GPU-based crossover operation.

The workload is assigned to threads as follows. Each thread crosses *bucketSize* elements of the two parent individuals and generates *bucketSize* elements of the new individual created. This means that each thread block generates *threadsPerBlock* $\times$ *bucketSize* elements of the new individual. The indices of the work bucket associated to each thread are organized to force each thread warp to access consecutive memory positions.

In the proposed design *n* crossing operations are performed in parallel to increase the thread level parallelism. The second dimension of each block represents the index of the individual resulting from the crossover operation. To optimize the addition of new individuals to the population set, the population is created with an additional GPU memory area to store the *n*

```
dim = computeStartDimension();
step = 0;
localResult = 0;
//Local processing performed by each thread
while (step < bucketSize)
{
  localResult += computeLocalResult(dim);
  dim += bD
  step++;
}

//Parallel reduction to obtain the final result
sharedMem[threadIndex] = localResult;
__syncthreads();

for (unsigned int s = threadsPerBlock/2; s > 0; s >>= 1)
{
  if (threadIndex < s)
    sharedMem[threadIndex] += sharedMem[threadIndex+s];
__syncthreads();
}
```

Fig. 5. General scheme of fitness function computation. Processing and reduction steps.

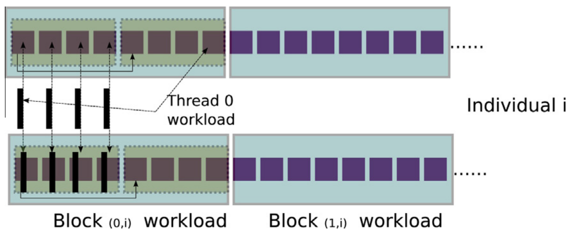

Fig. 6. Crossover processing scheme.

individuals generated in parallel by the crossover operation. These new individuals are always stored in this padding memory area and then the population is sorted by the fitness value. The sorting operation *moves* the *n* worst individuals to the padding area and the generated individuals get naturally inserted in the population into their corresponding position defined by their fitness value. Individuals in the padding area are not considered for any of the other computational steps of the algorithm. The sorting operation is explained in detail in Section 5.5.

Another operation related to the crossover phase that has been parallelized is the computation of the Euclidean distance between two individuals used to select the parents to be crossed. In this case the same design used for fitness functions was applied. Each thread computes a set of partial results of the distance, sums them up and then a reduction operation is performed at block level to sum the contributions of all the threads of each block. Finally, a second kernel is run to sum the contribution of each block.

5.3. Local search

The local search phase is the one less prone to be accelerated by a massively parallel design. This is due to the low arithmetic intensity and the high potential of code divergences of its operations. Nonetheless several kernels were developed to perform this process by assisting the CPU in tasks that could be performed in parallel. This avoided costly memory transfer operations from the GPU to main memory.

The operations that were parallelized are the following:

- individual change operations using the per element bias values
- bias values increment and decrement operations
- individual substitution in the population
- fitness evaluation: this step uses the same functionality and code explained in Section 5.1.

5.4. Random number generation

The random number generation has also been parallelized because it is one of the building blocks for the rest of computation phases. The CURAND library [15] included in the CUDA SDK was used (XORWOW random generator). For this purpose, the maximum number of parallel random number generators is computed by considering the dimensionality, the number of individuals processed in parallel and the number of active threads per individual.

The initialization of these random number generators produces some overhead during the start of the optimization process but this overhead gets rapidly compensated as the computation iterations are carried out.

5.5. Population sorting

Several steps of the computation process require the population to get sorted in terms of the fitness value. In order to avoid any costly GPU memory copy operations an indexing scheme was implemented. Each individual is assigned an index value to represent its order in the population based on its fitness value where the current best individual of the population will have associated an index value of 0. These indices are computed with a parallel sort by key function when necessary avoiding the need to store individuals in a sorted way.

The only drawback of this scheme is the extra memory access to compute at which position the *ith* individual is actually stored in the population container.

6. Empirical study

This section describes the experiments that have been carried out to assess the performance and accuracy of the proposed GPU-based memetic algorithm. It has the following structure: in Section 6.1 the goals that are evaluated through the experiments are described, in Section 6.2 the experimental design that was applied is explained, Section 6.3 presents the description and implementation details of the objective functions, Section 6.4 describes the hardware platforms that were used in the experimentation and Section 6.5 presents the experimental results that were obtained. Finally, Section 6.6 shows the profiling information about the various software modules.

6.1. Research questions evaluated by the experimental study

The experimental study presented in this work is aimed at analysing the GPU-based MA-SW-Chains algorithm with respect to the existing sequential, and purely CPU-based, version of the same algorithm. This analysis is based on the performance comparison in terms of run time required by both pieces of software and the speed-up factor achieved by using GPU hardware platforms.

The speed-up factor (*SU*) shows to what extent the GPU-based algorithm outperforms its CPU counterpart using the following expression:

$$SU = \frac{RT_C}{RT_G}$$

where RT_C is the run time of the sequential CPU implementation and RT_G the run time obtained using a GPU.

To assess the quality of the results obtained with each algorithm, as the optimum objective function value will be known a priori, the actual fitness value reached in each experiment can serve as a quality indicator. Moreover, in this study the fitness function will be the same as the objective function. Nevertheless, as we are dealing with high-dimensional problems, even individuals that are close to the global optimum may have associated fitness values whose absolute difference to the optimum is high. For this reason we have also computed the Euclidean distance of the best individual generated by each experiment with respect to the optimum. This measure is also used to compare CPU and GPU-based results.

6.2. Experimental design

To evaluate the performance of the proposal, six well-known optimization problems with different kinds of objective functions have been considered. For each function, experiments using different high-dimensionalities have been performed. The set of dimensionality values used for the experiments was: {100,000, 500,000, 1,000,000, 1,500,000, 3,000,000}.

For each function, and each dimension, ten experiments were run. From each experiment the best individual of the population was chosen and its fitness value and Euclidean distance to the optimum were recorded. The end of each experiment was set at the point where the number of fitness function evaluations reached 500,000 evaluations.

The parameters that were used for the MA-SW-Chains algorithm were the following:

- The SSGA uses $BLX - \alpha$ with $\alpha = 0.5$
- $I_{str} = 500$
- $r_{L/G} = 0.5$

This means that the SSGA and the LS are alternatively applied every 500 fitness function evaluations and that each phase is assigned 50% of the total number of evaluations. In the GPU-based version, at each crossover step 20 individuals are created in parallel.

6.3. Benchmark

This section describes the optimization problems and their associated objective functions that have been chosen as benchmark. First, a general description of each function will be provided in Section 6.3.1. Subsequently, in Section 6.3.2 the adaptation and implementation of these functions to GPU architectures will be explained.

6.3.1. Optimization problems

This section describes the set of optimization problems based on well-known functions used to test the performance of our proposal. The goal of each problem is to minimize an objective function whose expression is known. Therefore, the fitness function used to evaluate the quality of each solution is the objective function itself.

Regarding the fitness functions, it is important to highlight that:

- All fitness functions have been shifted to move the global optimum from the center of the search domain. This avoids favoring algorithms that have a tendency to generate solutions around the center of the search domain. $z_i = x_i - o_i$ represents the shifted variable of dimension *i*.
- A global bias value (*fBias*) was added to make the optimum fitness and objective function values not equal to 0. *fBias* was set to -330 for all functions which means -330 is the optimum value.
- All the functions can be extended to high-dimensionality without depending on matrices that could limit the maximum dimension.

In our study, to keep the computational cost of the experiments reasonable, we have used a small but representative set of objective functions, all well-known in literature. This set includes multimodal functions and functions where the influence on the final function value is not evenly distributed over the different variables, which is very common in real-world problems. Each of these functions is described justifying why it was chosen.

In addition to the previous benchmark setup, and following the indications by one of the reviewers, we have extended the optimization problems set with some of the functions used in the CEC 2010 competition [50]. Functions F1, F2, F3, F4, F6, F8, F13, F18 and F20 were used. Some of these functions include permutations and rotations applied to some variable sets. The results produced by MA-SW-Chains are not affected by permutations because all variables are treated as *first class citizens* at every step because no variable subsets are used to make any assumptions about the rest of the variables of the function. Only the GPU software has been run on all of these functions to provide evidence of its performance and scalability over a wider set of goal functions.

6.3.1.1. Rastrigin function. This function is defined as:

$$f(x) = \sum_{i=1}^{dim} (z_i)^2 - 10\cos(2\pi z_i) + 10 + fBias$$

This test function is non-convex, multimodal and additively separable. It has several local optima arranged in a regular lattice, but it has only one global optimum. It will show how the algorithm tackles multimodal functions.

6.3.1.2. Griewank function. This function is defined as:

$$f(x) = \sum_{i=1}^{dim} \frac{z_i^2}{4000} - \prod_{i=1}^{dim} \cos\left(\frac{z_i}{\sqrt{i}}\right) + 1 + fBias$$

This is a multimodal and non-separable function, with several local optima. It is similar to the Rastrigin function, but has a larger number of local optima. While this function has an exponentially increasing number of local minima, a simple multistart algorithm is able to detect this global minimum easier as the dimension increases [30].

6.3.1.3. Salomon function. The Salomon function has the following expression:

$$f(x) = 1 - \cos\left(2\pi\sqrt{g(z_i)}\right) + 0.1\sqrt{g(z_i)} + fBias$$

where $g(x) = \sum_{i=1}^{dim} (z_i)^2$. In this multimodal function the optima distribution is more complex than in the previous function, but, as the dimension increases, the complexity scales down, because the influence of the sinusoidal component is limited.

6.3.1.4. Discus function. The discus function is also very similar to the Sphere function, but it gives more importance to the first variable:

$$f(x) = 10^6 z_1^2 + \sum_{i=2}^{dim} i(z_i)^2 + fBias$$

The main interest of this function is that the influence of the first variable in the fitness value is inversely proportional to the dimension. As the dimension increases this first value becomes less important with respect to the rest and the function becomes more similar to the Sphere function.

6.3.1.5. Sum Squares function. This function has the particularity of using the values of the indices of the variables to produce the final result

$$f(x) = \sum_{i=1}^{dim} i(z_i)^2 + fBias$$

This fact increases the complexity because the influence of a variable increases with its dimension index.

6.3.1.6. Schwefel 1.2 function. The expression of this function is:

$$f(x) = \sum_{i=1}^{dim} \left(\sum_{j=1}^{i} z_j\right)^2$$

This is the most complex function of the set that has been used in this work. In this case the lower the index of a variable the higher the influence it has on the fitness function value. Moreover, as the dimensionality increases, the influence of each variable increases. This behavior makes it a very interesting function for evaluating our proposal.

6.3.2. Implementation of objective functions

This section details how each of the objective functions has been implemented as parallel GPU kernels using the general scheme explained in 5.1.

6.3.2.1. Rastrigin function. The expression of this function is the following:

$$f(x) = \sum_{i=1}^{dim} (z_i)^2 - 10\,cos(2\pi z_i) + 10 + fBias$$

The GPU implementation of this function follows the design principles explained in Section 5.1. The whole computation is split over a set of blocks (which can be run in parallel). The workload of each block is split over a set of threads (which can be run in parallel on the same SMX). Finally, each thread processes a set of elements controlled by the bucket size parameter. The sum over the threads of each block is performed in shared memory using intra-block synchronization mechanisms. The final sum over the set of blocks is performed in a second step using a dedicated kernel for this task. The expression of the function is therefore decomposed as shown in the following equation:

$$f(x) = \sum^{blocks} \sum^{threads} \sum^{bucket} (z_i)^2 - 10\,\cos(2\pi z_i) + 10 + fBias$$

It is important to state that the set of indices of the elements assigned to each thread is set up in a way that when the *i*th element of each bucket is processed all threads in the same warp access consecutive memory positions. This ensures fully coalesced memory accesses.

6.3.2.2. Griewank function. Using the same principles as in the previous section, the original function:

$$f(x) = \sum_{i=1}^{dim} \frac{z_i^2}{4000} - \prod_{i=1}^{dim} \cos\left(\frac{z_i}{\sqrt{i}}\right) + 1 + fBias$$

is decomposed as follows:

$$\sum^{blocks} \sum^{threads} \sum^{bucket} \frac{z_i^2}{4000} - \prod^{blocks} \prod^{threads} \prod^{bucket} \cos\left(\frac{z_i}{\sqrt{i}}\right) + 1 + fBias$$

The same mapping scheme was used to map the computation to threads and blocks.

6.3.2.3. Salomon function. The Salomon function has the following expression:

$$f(x) = 1 - \cos\left(2\pi\sqrt{g(z_i)}\right) + 0.1\sqrt{g(z_i)} + fBias$$

This function is based on the computation of the Sphere function:

$$g(x) = \sum_{i=1}^{dim} (z_i)^2$$

which is the part of the computation that can be parallelized using the following schema:

$$f(x) = \sum^{blocks} \sum^{threads} \sum^{bucket} (z_i)^2$$

Following the same design principles introduced above, each thread computes the partial sum of a set of squared variables (work bucket) and the whole computation is spread over a set of threads organized in thread blocks. The indices of the variables associated to each thread force coalesced memory access operations by each thread warp. The work of each thread block finishes with a reduction operation that sums the partial results of each thread. Finally, a global reduction operation sums the results produced by each thread block to produce the final fitness result.

6.3.2.4. Sum Squares function. The sum squares function is very similar to the Sphere function:

$$f(x) = \sum_{i=1}^{dim} i(z_i)^2 + fBias$$

but introduces and additional product operation using the index of each variable. Its adaptation to the GPU follows exactly the same principles:

$$f(x) = \sum^{blocks} \sum^{threads} \sum^{bucket} i(z_i)^2$$

It is important to remark that this function increases the arithmetic cost of the computation associated to each thread with respect to the Sphere function because of the additional product operation.

6.3.2.5. Discus function. The Discus function:

$$f(x) = 10^6 z_1^2 + \sum_{i=2}^{dim}(z_i)^2 + fBias$$

can also be considered a derivative of the Sphere function and introduces a variation in the computation associated to the first variable. The effect of the first variable is larger compared to the contribution of the rest of the variables in the final fitness value. The importance of the first variable becomes smaller as the number of dimensions increases.

The adaptation of this function to the GPU can be done as explained in Sections 6.3.2.3 and 6.3.2.4. The thread warp that processes the first variable will suffer from a code divergence as the computation associated to the rest of the variables is slightly different from the first one. Taking into account the dimensionality values considered in this work the performance penalty of this fact can be neglected.

6.3.2.6. Schwefel 1.2 function. The expression of this function:

$$f(x) = \sum_{i=1}^{dim}\left(\sum_{j=1}^{i} z_j\right)^2$$

shows that a more elaborated GPU implementation is required compared to the functions previously addressed.

- First a kernel computes the shifted variables z_j.
- Then the prefix sum or scan is computed $(\sum_{j=1}^{i} z_j)$ in parallel using the built-in operation in the Thrust library provided by Nvidia.
- A third kernel computes the square of each element.
- Finally, a reduction operation (also using the Thrust library) is performed to obtain the sum of all elements.

This function requires a higher number of GPU kernels with a lower arithmetic intensity and this fact has an impact on the speed-up results.

6.4. Hardware platforms

The CPU-based experiments using the standard sequential implementation were run on a server node equipped with an Intel Core i7-930 @2.8 GHz processor and 24 GB of RAM memory.

On the other hand, the GPU-based optimization software was run on an Nvidia Titan GPU. This GPU is equipped with 2688 CUDA cores @837 MHz and 6 GB of GDDR5 memory and is hosted on a PC equipped with an Intel Core i7-3770 K @3.50 GHz processor and 32 GB of RAM memory. The CPU on the GPU node is mainly used just to launch the parallel kernels and collect the results.

6.5. Empirical results

This section shows the results, in terms of performance and quality, obtained for each of the aforementioned optimization problems. For each objective function a table shows the run time for each dimension and optimization software type along with the speed-up factor of the GPU-based approach.

A second table shows the mean of the best fitness values obtained by the ten runs performed for each dimension and the mean Euclidean distance to the optimum of the ten individuals that exhibited those best fitness values. The optimum value of each objective function is −330 (which equals the *fBias* value) and the mean of the best solutions of each optimization run are the values shown on the tables as objective function values. The difference between the table values and −330 can be

Table 1
Rastrigin function test. Run time expressed in seconds. 500,000 objective function evaluations.

Dimensions	CPU	GPU	Speed-up
100,000	53,543 s (>0.5 days)	851.8 s (<15 min)	62.9
500,000	273,517 s (>3 days)	3188.3 s (<1 h)	85.8
1,000,000	547,424 s (>6 days)	4150.4 s (<1.5 h)	131.9
1,500,000	823,475 s (>9 days)	5404.2 s (≈1.5 h)	152.4
3,000,000	1,632,003 s (>18 days)	8776.9 s (<2.5 h)	185.9

Table 2
Rastrigin function. Quality of results performing 500,000 objective function evaluations. Optimum value -3.3×10^3.

Dimensions	Objective function		Euclidean distance	
	CPU	GPU	CPU	GPU
100,000	9.1×10^5	5.1×10^5	3.2×10^2	3.4×10^2
500,000	5.6×10^6	4.1×10^6	6.6×10^2	7.2×10^2
1,000,000	1.1×10^7	9.0×10^6	9.2×10^2	9.9×10^2
1,500,000	1.6×10^7	1.4×10^7	1.1×10^3	1.2×10^3
3,000,000	3.2×10^7	2.9×10^7	1.6×10^3	1.7×10^3

Table 3
Griewank function. Run time expressed in seconds. 500,000 objective function evaluations.

Dimensions	CPU	GPU	Speed-up
100,000	49,258 s (>13 h)	1086.7 s (≈18 min)	45.3
500,000	240,820 s (>2 days)	3460.8 s (<1 h)	69.6
1,000,000	479,457 s (>5 days)	4332.5 s (<1.5 h)	110.7
1,500,000	727,870 s (>8 days)	5519.2 s (≈1.5 h)	131.9
3,000,000	1,444,441 s (>16 days)	8639.8 s (<2.5 h)	167.2

Table 4
Griewank function. Quality of results performing 500,000 objective function evaluations. Optimum value -3.3×10^3.

Dimensions	Objective function		Euclidean distance	
	CPU	GPU	CPU	GPU
100,000	-3.1×10^2	-3.2×10^2	2.8×10^2	2.1×10^2
500,000	-2.3×10^2	-2.4×10^2	6.4×10^2	5.9×10^2
1,000,000	-1.2×10^2	-1.4×10^2	9.1×10^2	8.8×10^2
1,500,000	-1.7×10^1	-3.2×10^1	1.2×10^3	1.1×10^3
3,000,000	3.0×10^2	2.9×10^2	1.6×10^3	1.6×10^3

Table 5
Salomon function. Run time expressed in seconds. 500,000 objective function evaluations.

Dimensions	CPU	GPU	Speed-up
100,000	26,957 s (>7 h)	826.6 s (≈13 min)	32.6
500,000	136,214 s (>1.5 days)	3250.7 s (<1 h)	41.9
1,000,000	291,457 s (>3 days)	3825.5 s (≈1 h)	76.2
1,500,000	454,624 s (>5 days)	4781.4 s (<1.5 h)	95.1
3,000,000	813,716 s (>9 days)	7637.1 s (<2.5 h)	106.5

Table 6
Salomon function. Quality of results performing 500,000 objective function evaluations. Optimum value -3.3×10^3.

Dimensions	Objective function		Euclidean distance	
	CPU	GPU	CPU	GPU
100,000	-3×10^2	-3×10^2	2.8×10^2	2.8×10^2
500,000	-2.7×10^2	-2.7×10^2	6.5×10^2	6.4×10^2
1,000,000	-2.4×10^2	-2.4×10^2	9.1×10^2	9.1×10^2
1,500,000	-2.2×10^2	-2.2×10^2	1.1×10^3	1.1×10^3
3,000,000	-1.7×10^2	-1.7×10^2	1.6×10^3	1.6×10^3

considered as an error measure. The Euclidean distance to the optimum individual is supplied as an additional error measure.

As an indication of the size and computational cost of the experimentation that has been carried out, the CPU-based optimization tests required over 150 days (≈5 months) of single threaded CPU time (split over several physical CPU cores). The same experiments required over 38 h to complete on the GPU hardware.

Table 7
Sum Squares function. Run time expressed in seconds. 500,000 objective function evaluations.

Dimensions	CPU	GPU	Speed-up
100,000	28,983 s (>8 h)	881.6 s (≈14 min)	32.9
500,000	142,000 s (>1.5 days)	3320.0 s (<1 h)	42.8
1,000,000	312,872 s (>3 days)	3991.9 s (≈1 h)	78.4
1,500,000	454,497 s (>5 days)	4994.7 s (<1.5 h)	91.0
3,000,000	877,941 s (>10 days)	8119.1.1 s (<2.5 h)	108.1

Table 8
Sum Squares function. Quality of results performing 500,000 objective function evaluations. Optimum value -3.3×10^3.

Dimensions	Objective function		Euclidean distance	
	CPU	GPU	CPU	GPU
100,000	3.6×10^9	1.5×10^9	2.8×10^2	2.2×10^2
500,000	1.0×10^{11}	8.4×10^{10}	6.4×10^2	6.1×10^2
1,000,000	4.1×10^{11}	3.7×10^{11}	9.1×10^2	8.9×10^2
1,500,000	9.3×10^{11}	8.7×10^{11}	1.1×10^3	1.1×10^3
3,000,000	3.8×10^{12}	3.6×10^{12}	1.6×10^3	1.6×10^3

Table 9
Discus function. Run time expressed in seconds. 500,000 objective function evaluations.

Dimensions	CPU	GPU	Speed-up
100,000	27,233 s (>7 h)	862.5 s (≈14 min)	31.6
500,000	136,382 s (>1.5 days)	3275.0 s (<1 h)	41.6
1,000,000	282,043 s (>3 days)	3874.8 s (≈1 h)	72.8
1,500,000	406,658 s (>4.5 days)	4850.1 s (<1.5 h)	83.8
3,000,000	805,641 s (>9 days)	7836.7 s (<2.5 h)	102.8

Table 10
Discus function. Quality of results performing 500,000 objective function evaluations. Optimum value -3.3×10^3.

Dimensions	Objective function		Euclidean distance	
	CPU	GPU	CPU	GPU
100,000	7.7×10^4	7.7×10^4	2.8×10^2	2.8×10^2
500,000	4.1×10^5	4.1×10^5	6.4×10^2	6.4×10^2
1,000,000	8.3×10^5	8.3×10^5	9.1×10^2	9.1×10^2
1,500,000	1.2×10^6	1.3×10^6	1.1×10^3	1.1×10^3
3,000,000	2.5×10^6	2.5×10^6	1.6×10^3	1.6×10^3

Table 11
Schwefel function. Run time expressed in seconds. 500,000 objective function evaluations.

Dimensions	CPU	GPU	Speed-up
100,000	32,743 s (>9 h)	1197.7 s (<20 min)	27.3
500,000	165,456 s (>1 day)	4386.6 s (<1.5 h)	37.7
1,000,000	331,086 s (>3 days)	5334.2 s (≈1.5 h)	62.1
1,500,000	467,190 s (>5 days)	6681.7 s (<2 h)	69.9
3,000,000	939,027 s (>10 days)	10849.4 s (≈3 h)	86.6

6.5.1. Rastrigin function

The results for each function are expressed in terms of run time and quality of the results. Table 1 shows the run time, expressed in seconds, for the CPU and the CPU optimization software for a set of dimensionality values up to 3,000,000. Run time is also expressed in more human readable format to better highlight the important performance differences of each piece of software. Finally, the last column of Table 1 shows the speed-up factor achieved by the GPU-based software. These factors range from one to two orders of magnitude.

Table 12
Schwefel function. Quality of results performing 500,000 objective function evaluations. Optimum value -3.3×10^3.

Dimensions	Objective function		Euclidean distance	
	CPU	GPU	CPU	GPU
100,000	1.2×10^7	6.8×10^6	3.8×10^2	3.8×10^2
500,000	3.6×10^8	2.7×10^8	9.5×10^2	9.7×10^2
1,000,000	1.5×10^9	8.7×10^{10}	1.4×10^3	1.4×10^3
1,500,000	9.2×10^{14}	4.9×10^{11}	1.7×10^3	1.8×10^3
3,000,000	4.7×10^{16}	1.0×10^{15}	1.7×10^3	2.4×10^3

Table 13
CPU profiling information.

100,000		500,000	
37%	Fitness evaluation	38%	Fitness evaluation
28%	Random number generation	28%	Random number generation
18%	Crossover operation	20%	Crossover operation
5%	Distance computation	5%	Distance computation

Table 2 shows the mean absolute objective function values of the best individuals and mean Euclidean distance to the optimum value obtained with both the CPU and GPU version. It can be seen that the Euclidean distance obtained with both optimization pieces of software are virtually identical. The absolute fitness values of the best individual show higher differences but within an acceptable range.

6.5.2. Griewank function

Table 3 shows the run times and speed-up factors obtained for this function, which, as the results show, also presents a good adaptation to the GPU architecture and the speed-up values keep in line with the ones described in Section 6.5.1 for the Rastrigin function.

Table 4 shows the mean objective function values and Euclidean distances to the optimum result. In both cases the results of the CPU and GPU-based pieces of software are almost identical.

6.5.3. Salomon function

Table 5 contains the run times and speed-up factors for the Salomon function. In this case the speed-up factors are lower compared to the ones obtained in the previous sections. This is caused by the lower arithmetic intensity of this function with respect to Rastrigin and Griewank functions. For this function each thread does only compute z_i^2 compared to more computationally intensive operations such as the cosine and square root required by other functions.

Table 6 allows to compare the quality of the results obtained with the sequential and the parallel software versions. As in the previous sections, the results are very similar in both cases.

6.5.4. Sum Squares function

Table 7 displays the run time related results of the Sum Squares function. As expected, they are very similar to the ones obtained with the Salomon function.

Table 8 displays the mean objective function values and the Euclidean distances. There are small differences in the fitness values (more noticeable for lower dimensions) but the distances of the solutions to the optimum are almost identical.

6.5.5. Discus function

Table 9 shows the run time values and speed-up factors for this function. They are in line with the ones obtained for the Salomon and Sum Squares functions.

Table 14
GPU profiling information.

100,000		500,000		1,000,000	
18%	Distance	23%	Crossover + rand. gen.	25%	Crossover + rand. gen.
12%	Crossover + rand. gen.	21%	Distance	20%	Distance
15%	GPU to RAM transfers	17%	Solis Wets indiv. change	17%	Solis Wets indiv. change
15%	Solis Wets indiv. change	14%	Fitness eval	14%	Fitness eval.
13%	Fitness eval	10%	Random number init.	13%	Random number init.

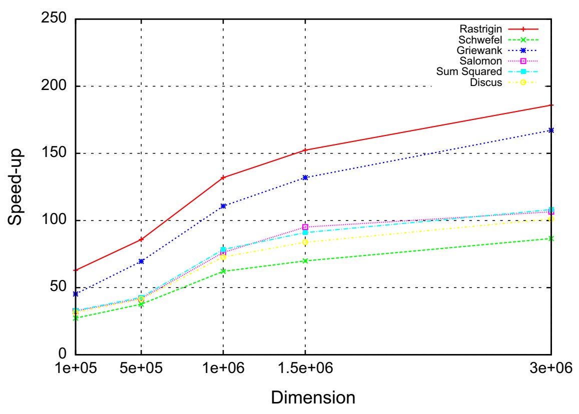

Fig. 7. Speed up factors.

Table 10 shows the mean objective function values and distances to the optimum result for both pieces of software which are again almost identical.

6.5.6. Schwefel function

The results obtained for this function are shown in Table 11. It is important to emphasize the fitness function computations for the Schwefel function have to be split over several kernel functions and that the arithmetic intensity of these kernels is low, which reduces the time the GPU can run tasks autonomously (without the intervention of the host computer), and that this is the reason because of which the speed-up factors obtained are lower than the other presented in this work.

Table 12 shows the mean objective function values of the best individual and the Euclidean distance of these individuals to the optimum.

This is the only function from the ones considered in this work that presents noticeable differences in the mean fitness values and distances to the optimum result on the two pieces of software analyzed. Because of the nature of this function and the high-dimensionalities used in this paper the quality of the best individuals obtained in each optimization run are very different. This is demonstrated by the high standard deviation values obtained for this function and that are shown in the Appendix (Table 20). The quality of the results for this function is further analyzed in the Appendix at the end of this paper.

6.6. Profiling results

This section explores how the computational cost is distributed among the main modules of the two optimization systems that were used. Table 13 shows the result of the CPU sequential code. The profiling results for the CPU have only been performed for dimensions 100,000 and 500,000 and over just one optimization run due to the large overhead required to profile the CPU code.

The most costly computational parts of the base CPU code are the fitness function evaluation, the random number generation, the crossover operation and to a lesser degree, the computation of the distance between individuals (used to choose which individuals are crossed). All these modules have been parallelized and are run on the GPU in the optimization software presented in this work.

Table 14 displays the results of the GPU code. For higher dimensions, the operations that require most of the total computational time are the crossover operation, which includes the random number generation, the distance computation, the change operation of the individuals during the Solis Wets local search process, the fitness function evaluation and the random number generator initialization.

For dimension 100,000 GPU to RAM memory transfers play an important role with respect to the total time which is not the case anymore when the dimension increases.

6.7. CEC 2010 competition functions

The results obtained for the set of functions part of the CEC 2010 competition [50] are not as comprehensive as the ones presented in previous sections. As the computational requirements of many of these functions are higher because of the rotation operations, due to time constraints, it was not possible to compute the CPU-based results.

Nevertheless, the GPU-based software has been run on all the functions and dimensions, just as in the previous group of functions. The result tables for dimension values {100,000, 500,000, 1,000,000, 1,500,000, 3,000,000} are presented in the appendix Section A.1.

Table 15
Rastrigin function. Quality of results (extended information) performing 500,000 objective function evaluations. Optimum value −3.3E+02. The table shows the mean, standard deviation and median of the best solutions after 10 optimization runs.

	CPU					GPU				
	100,000	500,000	1,000,000	1,500,000	3,000,000	100,000	500,000	1,000,000	1,500,000	3,000,000
Objective function										
μ	9.1E+05	5.4E+06	1.1E+07	1.6E+07	3.2E+07	5.1E+05	4.1E+06	9.0E+06	1.4E+07	2.9E+07
σ	5.5E+03	1.5E+04	1.4E+04	1.9E+04	2.9E+04	1.3E+04	3.2E+04	3.4E+04	5.5E+04	7.6E+04
$\tilde{x}$	9.1E+05	5.4E+06	1.1E+07	1.6E+07	3.2E+07	5.1E+05	4.1E+06	9.0E+06	1.4E+07	2.9E+07
Distance										
μ	3.3E+02	6.6E+02	9.2E+02	1.1E+03	1.6E+03	3.4E+02	7.2E+02	1.0E+03	1.2E+03	1.7E+03
σ	4.7E+00	4.5E+00	1.7E+00	1.8E+00	3.2E+00	9.0E+00	3.3E+00	7.9E+00	4.0E+00	5.3E+00
$\tilde{x}$	3.3E+02	6.6E+02	9.2E+02	1.1E+03	1.6E+03	3.4E+02	7.2E+02	1.0E+03	1.2E+03	1.7E+03

Table 16
Griewank function. Quality of results (extended information) performing 500,000 objective function evaluations. Optimum value −3.3E+02. The table shows the mean, standard deviation and median of the best solutions after 10 optimization runs.

	CPU					GPU				
	100,000	500,000	1,000,000	1,500,000	3,000,000	100,000	500,000	1,000,000	1,500,000	3,000,000
Objective function										
μ	−3.1E+02	−2.3E+02	−1.2E+02	−1.7E+01	3.0E+02	−3.2E+02	−2.4E+02	−1.4E+02	−3.2E+01	2.9E+02
σ	8.5E−02	2.3E−01	2.1E−01	5.1E−01	1.0E+00	1.7E−01	6.1E−01	5.7E−01	6.3E−01	9.6E−01
$\tilde{x}$	−3.1E+02	−2.3E+02	−1.2E+02	−1.7E+01	3.0E+02	−3.2E+02	−2.4E+02	−1.4E+02	−3.2E+01	2.9E+02
Distance										
μ	2.8E+02	6.4E+02	9.1E+02	1.1E+03	1.6E+03	2.1E+02	5.9E+02	8.7E+02	1.1E+03	1.6E+03
σ	6.3E−01	7.2E−01	4.6E−01	9.1E−01	1.3E+00	1.5E+00	2.0E+00	1.3E+00	1.1E+00	1.2E+00
$\tilde{x}$	2.8E+02	6.4E+02	9.1E+02	1.1E+03	1.6E+03	2.1E+02	5.9E+02	8.7E+02	1.1E+03	1.6E+03

Table 17
Salomon function. Quality of results (extended information) performing 500,000 objective function evaluations. Optimum value −3.3E+02. The table shows the mean, standard deviation and median of the best solutions after 10 optimization runs.

	CPU					GPU				
	100,000	500,000	1,000,000	1,500,000	3,000,000	100,000	500,000	1,000,000	1,500,000	3,000,000
Objective function										
μ	−3.0E+02	−2.7E+02	−2.4E+02	−2.2E+02	−1.7E+02	−3.0E+02	−2.7E+02	−2.4E+02	−2.2E+02	−1.7E+02
σ	4.9E−02	1.4E−01	6.0E−02	8.9E−02	1.4E−01	7.6E−02	8.5E−02	9.7E−02	5.6E−02	8.7E−02
$\tilde{x}$	−3.0E+02	−2.7E+02	−2.4E+02	−2.2E+02	−1.7E+02	−3.0E+02	−2.7E+02	−2.4E+02	−2.2E+02	−1.7E+02
Distance										
μ	2.8E+02	6.4E+02	9.1E+02	1.1E+03	1.6E+03	2.8E+02	6.4E+02	9.1E+02	1.1E+03	1.6E+03
σ	4.9E−01	1.4E+00	6.1E−01	8.7E−01	1.4E+00	7.8E−01	8.9E−01	1.0E+00	5.4E−01	8.0E−01
$\tilde{x}$	2.8E+02	6.4E+02	9.1E+02	1.1E+03	1.6E+03	2.8E+02	6.4E+02	9.1E+02	1.1E+03	1.6E+03

7. Analysis of results

The goal of the experiments described in Section 6 was to show the speed-up factors obtained with the GPU-based MA-SW-Chains optimization technique and how they increase as the dimensionality grows. This is also shown in Fig. 7 and is due to the fact that the higher the dimensionality of the problem the higher the thread level parallelism and the performance obtained. On the other hand, the run time of the CPU algorithm scales linearly as the dimensionality increases.

The impact of speed-up factors on run times illustrates how the use of GPUs allows to deal with problems that require an unacceptable time to be completed by single-threaded CPU-based implementations. This becomes especially obvious for dimensionality 3,000,000 where the run time is reduced from 18 days to less than 2.5 h. It could be suggested that the CPU-based algorithm could be redesigned for a multi-core CPU, but in that case, only one order of magnitude would be the maximum speed-up that could be expected. This is still an order of magnitude less than the performance shown by the GPU-based version.

The importance of the arithmetic intensity per kernel is also clearly shown. The speed-up factors obtained for the Schwefel function are lower because it requires several kernel launches where each kernel exhibits a very low arithmetic intensity when compared to the required memory access operations. On the other hand, more complex fitness functions achieve higher speed-up ratios even for low dimensions.

Table 18
Sum Squares function. Quality of results (extended information) performing 500,000 objective function evaluations. Optimum value −3.3E+02. The table shows the mean, standard deviation and median of the best solutions after 10 optimization runs.

	CPU					GPU				
	100,000	500,000	1,000,000	1,500,000	3,000,000	100,000	500,000	1,000,000	1,500,000	3,000,000
Objective function										
μ	3.6E+09	1.0E+11	4.1E+11	9.3E+11	3.8E+12	1.5E+09	8.4E+10	3.7E+11	8.7E+11	3.6E+12
σ	3.0E+07	1.7E+08	1.0E+09	1.8E+09	7.5E+09	6.2E+07	6.8E+08	1.2E+09	2.2E+09	6.4E+09
$\tilde{x}$	3.6E+09	1.0E+11	4.1E+11	9.3E+11	3.8E+12	1.5E+09	8.4E+10	3.7E+11	8.7E+11	3.6E+12
Distance										
μ	2.8E+02	6.4E+02	9.1E+02	1.1E+03	1.6E+03	2.2E+02	6.1E+02	8.9E+02	1.1E+03	1.6E+03
σ	6.6E−01	5.5E−01	1.2E+00	1.0E+00	1.6E+00	3.0E+00	1.8E+00	1.2E+00	1.0E+00	9.4E−01
$\tilde{x}$	2.8E+02	6.4E+02	9.1E+02	1.1E+03	1.6E+03	2.2E+02	6.1E+02	8.9E+02	1.1E+03	1.6E+03

Table 19
Discus function. Quality of results (extended information) performing 500,000 objective function evaluations. Optimum value −3.3E+02. The table shows the mean, standard deviation and median of the best solutions after 10 optimization runs.

	CPU					GPU				
	100,000	500,000	1,000,000	1,500,000	3,000,000	100,000	500,000	1,000,000	1,500,000	3,000,000
Objective function										
μ	7.7E+04	4.1E+05	8.3E+05	1.2E+06	2.5 + E06	7.7E+04	4.1E+05	8.3E+05	1.3E+06	2.5E+06
σ	3.2E+02	4.4E+02	1.3E+03	1.8E+03	5.0 + E03	2.4E+02	6.3E+02	2.1E+03	2.3E+03	6.1E+03
$\tilde{x}$	7.7E+04	4.1E+05	8.3E+05	1.2E+06	2.5 + E06	7.7E+04	4.1E+05	8.3E+05	1.3E+06	2.5E+06
Distance										
μ	2.8E+02	6.4E+02	9.1E+02	1.1E+03	1.6E+03	2.8E+02	6.4E+02	9.1E+02	1.1E+03	1.6E+03
σ	5.7E−01	3.4E−01	7.0E−01	8.1E−01	1.6E+00	4.2E−01	4.9E−01	1.2E+00	1.0E+00	1.9E+00
$\tilde{x}$	2.8E+02	6.4E+02	9.1E+02	1.1E+03	1.6E+03	2.8E+02	6.4E+02	9.1E+02	1.1E+03	1.6E+03

In regard to the quality of the solutions, some statistics of the empirical result distribution are detailed in the Appendix. It is easy to check that: (a) there is, in general, a wide diversity in the results; and (b) there are no relevant differences between the solutions obtained by the CPU algorithm and the GPU one.

With respect to (a), this is due to the stochastic nature of the algorithms and the huge size of the search space. And obviously, this effect grows with the dimension. In regard to (b) the quality of the solutions is quite similar: one would expect the same error from each one. The detailed results allow to check that none of the two software versions is systematically better than the other.

8. Conclusions

We have presented a GPU-based redesign of the MA-SW-Chains memetic algorithm suited for high dimensionality. This GPU algorithm offers between one and two orders of magnitude of speed-up compared to the single threaded CPU version while maintaining the quality of the results. This opens this algorithm to completely new usability scenarios because it allows to address problems with a dimensionality that cannot be handled with current CPU implementations. The extremely high dimensionality scenarios that have been considered in the experiments also show how they can be handled by suitable memetic algorithms running on GPU-based hardware platforms.

The empirical study of this work shows that an optimization process that can take several days on a CPU-based architecture can be solved in a few hours using a single GPU. This massive performance increase is the key aspect that makes the difference between having a problem that can be managed by an optimization technique or not. A solution to a problem that requires an unacceptable amount of time is not valid in most use case environments. This is precisely the rationale under heuristics after all. While the concept of *acceptable* run times is dependent on the technology, currently, the only feasible choice to go for over millions of dimensions are high performance architectures, with GPUs playing a leading role.

In our trek for a superefficient GPU algorithm that yields the maximum performance out of this particular architecture, a non-trivial effort has been required. The redesign process not only considered the parallelization of the fitness function, which is problem-dependent, but also several other components of the algorithm have been redeveloped to boost the performance. This includes the crossover operator, distance computation and several steps in the local search phase. For each kernel, an adequate workload packaging level has been found and applied to each thread and an efficient mapping between threads and thread blocks has been designed to allow an efficient use of the GPU hardware. These details are highlighted because initial and more naïve GPU designs of the optimization system did not lead to good enough results and the fine-tuning of all the aforementioned aspects are crucial to obtain peak performance.

Table 20
Schwefel function. Quality of results (extended information) performing 500,000 objective function evaluations. Optimum value −3.3E+02. The table shows the mean, standard deviation and median of the best solutions after 20 optimization runs.

	CPU					GPU				
	100,000	500,000	1,000,000	1,500,000	3,000,000	100,000	500,000	1,000,000	1,500,000	3,000,000
Objective function										
μ	1.2E+07	3.6E+08	1.5E+09	9.2E+14	4.7E+16	6.8E+06	2.8E+08	8.7E+10	4.9E+11	1.0E+15
σ	5.2E+05	2.1E+07	1.1E+08	1.6E+15	5.0E+14	1.4E+06	5.6E+07	8.3E+10	2.5E+11	1.1E+15
$\tilde{x}$	1.2E+07	3.5E+08	1.5E+09	8.9E+09	4.7E+16	6.8E+06	2.7E+08	7.3E+10	4.5E+11	7.7E+14
Distance										
μ	3.9E+02	9.6E+02	1.4E+03	1.6E+03	1.7E+03	3.8E+02	9.8E+02	1.4E+03	1.8E+03	2.4E+03
σ	6.1E+00	5.7E+00	6.9E+00	2.3E+02	3.4E+00	2.0E+01	4.4E+01	1.3E+01	8.4E+00	2.5E+01
$\tilde{x}$	3.9E+02	9.6E+02	1.4E+03	1.7E+03	1.7E+03	3.8E+02	1.0E+03	1.4E+03	1.8E+03	2.4E+03

Table 21
CEC 2010 F1 function. Mean, standard deviation and median of the best solutions obtained performing 500,000 objective function evaluations per optimization run (20 runs). Optimum value 0.0.

	100,000	500,000	1,000,000	1,500,000	3,000,000
GPU					
μ	1.4E+13	1.2E+14	2.6E+14	4.1E+14	8.6E+14
σ	2.3E+11	4.5E+11	1.2E+12	1.4E+12	1.8E+12
$\tilde{x}$	1.4E+13	1.2E+14	2.6E+14	4.1E+14	8.6E+14

Table 22
CEC 2010 F2 function. Mean, standard deviation and median of the best solutions obtained performing 500,000 objective function evaluations per optimization run (20 runs). Optimum value 0.0.

	100,000	500,000	1,000,000	1,500,000	3,000,000
GPU					
μ	1.9E+06	9.9E+06	2.0E+07	3.0E+07	6.1E+07
σ	1.4E+04	4.5E+04	3.6E+04	4.6E+04	4.4E+04
$\tilde{x}$	1.8E+06	9.9E+06	2.0E+07	3.0E+07	6.1E+07

Table 23
CEC 2010 F3 function. Mean, standard deviation and median of the best solutions obtained performing 500,000 objective function evaluations per optimization run (20 runs). Optimum value 0.0.

	100,000	500,000	1,000,000	1,500,000	3,000,000
GPU					
μ	2.1E+01	2.2E+01	2.2E+01	2.2E+01	2.2E+01
σ	3.5E−03	4.6E−03	2.8E−03	2.8E−03	2.8E−03
$\tilde{x}$	2.2E+01	2.2E+01	2.2E+01	2.2E+01	2.2E+01

Table 24
CEC 2010 F4 function. Mean, standard deviation and median of the best solutions obtained performing 500,000 objective function evaluations per optimization run (20 runs). Optimum value 0.0.

	100,000	500,000	1,000,000	1,500,000	3,000,000
GPU					
μ	2.5E+13	1.3E+14	2.8E+14	4.2E+14	8.9E+14
σ	8.3E+11	1.4E+12	9.9E+11	1.2E+12	3.6E+12
$\tilde{x}$	2.5E+13	1.3E+14	2.8E+14	4.3E+14	8.9E+14

As future work we plan extending the presented scheme to multi-GPU platforms to fully exploit the increasing availability of high speed PCI Express slots even on commodity computers. This would allow running several optimization steps in parallel or reducing the run time of each optimization process by allowing several GPUs to collaborate on the same optimization task.

Table 25
CEC 2010 F6 function. Mean, standard deviation and median of the best solutions obtained performing 500,000 objective function evaluations per optimization run (20 runs). Optimum value 0.0.

	100,000	500,000	1,000,000	1,500,000	3,000,000
GPU					
μ	2.2E+01	2.2E+01	2.2E+01	2.2E+01	2.2E+01
σ	5.3E−01	4.3E−01	6.4E−01	5.1E−01	5.6E−01
$\widetilde{x}$	2.1E+01	2.1E+01	2.2E+01	2.2E+01	2.2E+01

Table 26
CEC 2010 F8 function. Mean, standard deviation and median of the best solutions obtained performing 500,000 objective function evaluations per optimization run (20 runs). Optimum value 0.0.

	100,000	500,000	1,000,000	1,500,000	3,000,000
GPU					
μ	4.3E+07	4.2E+07	1.1E+08	1.6E+08	4.4E+07
σ	2.1E+05	9.7E+05	7.8E+07	7.8E+07	1.2E+06
$\widetilde{x}$	4.3E+07	4.E+07	4.3E+07	1.6E+08	4.3E+07

Table 27
CEC 2010 F13 function. Mean, standard deviation and median of the best solutions obtained performing 500,000 objective function evaluations per optimization run (20 runs). Optimum value 0.0.

	100,000	500,000	1,000,000	1,500,000	3,000,000
GPU					
μ	1.3E+14	9.5E+14	2.0E+15	3.1E+15	6.4E+15
σ	6.3E+12	5.7E+12	4.1E+12	1.4E+13	1.1E+13
$\widetilde{x}$	1.3E+14	9.4E+14	2.0E+15	3.1E+15	6.4E+15

Table 28
CEC 2010 F18 function. Mean, standard deviation and median of the best solutions obtained performing 500,000 objective function evaluations per optimization run (20 runs). Optimum value 0.0.

	100,000	500,000	1,000,000	1,500,000	3,000,000
GPU					
μ	3.2E+14	2.0E+15	4.2E+15	6.4E+15	1.3E+16
σ	3.2E+12	8.7E+12	7.5E+12	1.8E+13	6.5E+12
$\widetilde{x}$	3.2E+14	2.0E+15	4.2E+15	6.4E+15	1.37E+16

Table 29
CEC 2010 F20 function. Mean, standard deviation and median of the best solutions obtained performing 500,000 objective function evaluations per optimization run (20 runs). Optimum value 0.0.

	100,000	500,000	1,000,000	1,500,000	3,000,000
GPU					
μ	3.2E+14	2.0E+15	4.2E+15	6.4E+15	1.3E+16
σ	3.7E+12	1.2E+13	1.2E+13	1.2E+13	1.6E+13
$\widetilde{x}$	3.2E+14	2.0E+15	4.2E+15	6.4E+15	1.3E+16

Appendix A

This appendix provides a more detailed view and some summary statistics of the results obtained with both CPU and GPU-based optimization pieces of software. Each table of this section shows the mean value (μ), standard deviation (σ) and median $\widetilde{x}$ of the series of fitness values and Euclidean distance values to the optimum value of the best individuals of each optimization run. Each table is divided into columns corresponding to the different dimensions considered in the experiments.

As it has been explained in Section 6, for each function and dimension, a set of ten optimization runs were performed. Each run produced an objective function value of the best individual and the Euclidean distance of this individual to the optimum. As an example, using the CPU sequential optimization software on the Rastrigin function and dimension 100,000, the

Table 30
CEC 2010 functions. Euclidean distance of the best found solution to the optimum solution for each function and dimension value.

	100,000	500,000	1,000,000	1,500,000	3,000,000
GPU					
F1	1.4E+04	2.8E+04	2.6E+04	3.9E+04	6.7E+04
F2	5.9E+02	1.3E+03	1.9E+03	2.3E+03	3.2E+03
F3	4.5E+03	9.9E+03	1.4E+04	1.7E+04	2.4E+04
F4	1.2E+04	2.7E+04	3.8E+04	4.7E+04	6.7E+04
F6	3.8E+03	8.6E+03	1.2E+04	1.5E+04	2.1E+04
F8	1.2E+04	2.7E+04	3.8E+04	4.6E+04	6.6E+04
F13	1.3E+04	2.7E+04	3.8E+04	4.6E+04	6.4E+04
F18	1.2E+04	2.6E+04	3.7E+04	4.5E+04	6.4E+04
F20	1.2E+04	2.7E+04	3.7E+04	4.5E+04	6.4E+04

mean best objective function value was 9.1E+05, and the standard deviation and median of these ten values were: 5.5E+03 and 9.1E+05, respectively. For the same test case, the mean distance of the best individual of each run was 3.3E+02, whereas the standard deviation and median of the distance values were 4.7E+00 and 3.3E+02. All these values can be found in the first column of Table 15.

Tables 15–19 show that the results obtained by the GPU-based optimization software are almost identical to the ones obtained with the reference CPU solution. This is not the case of the objective function results obtained for the Schwefel function (Table 20) as important differences appear for higher dimensionality values. This can be explained by the high variability of the fitness values obtained in each run, confirmed by the high standard deviation values. This happens due to the very different influence of the variables of each individual in the final fitness value. The lower the index of the variable, the greater its influence on the result. The information about the distance values shows the different approximations obtained are at a very similar Euclidean distance to the optimum but due to the intrinsic nature of this function, with very different associated fitness values.

Taking into account the results associated to the Schwefel function, the authors would like to highlight that both pieces of software where validated using some test individuals to ensure the correct implementation of the fitness function and that twenty optimization runs were performed for the Schwefel function instead of ten as for the rest of the functions.

A.1. Result tables of the CEC 2010 competition functions

This section presents the results obtained for a set of the optimization problems from the CEC 2010 competition. Tables 21–29 contain the mean, standard deviation and median of the best objective function values obtained per optimization run. For each function and dimension, 20 experiments were performed.

Table 30 provides the Euclidean distance of the mean best solution of each optimization run to the optimum solution. The fact that these distance values are larger than the ones obtained for the rest of the functions of this paper is explained by the wider search domains used for each variable in this competition with respect to the testing environment used previously. This fact produces a slower convergence to the solution.

GPU run times are not provided as it was not possible to obtain the CPU counterpart results to compare with. The CPU run times for these functions would exceed the total CPU time used for the rest of the experiments and indicated in Section 6.5 (over 5 months of CPU time). Nevertheless, these GPU run times are of the same order as the ones shown in the result tables presented in previous sections.

References

[1] Deepak Agarwal, Srujana Merugu, Predictive discrete latent factor models for large scale dyadic data, in: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '07, ACM, New York, NY, USA, 2007, pp. 26–35.
[2] Enrique Alba, Gabriel Luque, Sergio Nesmachnow, Parallel metaheuristics: recent advances and new trends, Int. Trans. Oper. Res. 20 (1) (2013) 1–48.
[3] T. Bäck, D.B. Fogel, Z. Michalewicz (Eds.), Handbook of Evolutionary Computation, IOP Publishing.Ltd., Bristol, UK, 1997.
[4] Silvia Bahmann, Jens Kortus, EVO—Evolutionary algorithm for crystal structure prediction, Comput. Phys. Commun. 184 (6) (2013) 1618–1625.
[5] Hongtao Bai, Dantong Ouyang, Ximing Li, Lili He, Haihong Yu, MAX-MIN ant system on GPU with CUDA, in: 2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC), 2009, pp. 801–204.
[6] Liang Bai, Jiye Liang, Chuangyin Dang, Fuyuan Cao, A novel attribute weighting algorithm for clustering high-dimensional categorical data, Pattern Recogn. 44 (12) (2011) 2843–2861.
[7] Jacques Blum, François-Xavier Le Dimet, I Michael Navon, Data assimilation for geophysical fluids, in: P.G. Ciarlet (Ed.), Handbook of Numerical Analysis, Handbook of Numerical Analysis, vol. 14, Elsevier, 2009, pp. 385–441.
[8] Wojciech Boejko, Czesaw Smutnicki, Mariusz Uchroski, Parallel calculating of the goal function in metaheuristics using GPU, in: Gabrielle Allen, Jarosaw Nabrzyski, Edward Seidel, GeertDick Albada, Jack Dongarra, Peter, M.A. Sloot (Eds.), Computational Science ICCS 2009, Lecture Notes in Computer Science, vol. 5544, Springer, Berlin, Heidelberg, 2009, pp. 1014–2023.
[9] J. Brest, A. Zamuda, B. Bovskovi'c, M.S. Mauvcec, V. Zumer, High-dimensional real-parameter optimization using self-adaptive differential evolution algorithm with population size reduction, in: 2008 IEEE Congress on Evolutionary Computation, 2008, pp. 2032–2039.
[10] J. Brest, A. Zamuda, I. Fister, M.S. Maucec, Large scale global optimization using self-adaptive differential evolution algorithm, in: 2010 IEEE Congress on Evolutionary Computation (CEC), 2010, pp. 1–8.

[11] Rogerio M. Calazan, Nadia Nedjah, Luiza De Macedo Mourelle, Parallel GPU-based implementation of high dimension particle swarm optimizations, 2013 IEEE 4th Latin American Symposium on Circuits and Systems (LASCAS), February 2013, pp. 1–4.
[12] A. Caponio, G.L. Cascella, F. Neri, N. Salvatore, M. Sumner, A fast adaptive memetic algorithm for off-line and on-line control design of PMSM drivers, IEEE Trans. Syst. Man Cybern. — Part B, Special Issue Memetic Algorithms 37 (1) (2007) 28–41.
[13] L Chen, I Fujishiro, K Nakajima, Parallel performance optimization of large-scale unstructured data visualization for the earth simulator, In: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization, EGPGV '02, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association, pp. 133–140.
[14] Cuda. <http://www.nvidia.com/object/cuda_home_new.html>.
[15] Curand library. <https://developer.nvidia.com/curand>.
[16] Swagatam Das, Ponnuthurai Nagaratnam Suganthan, Differential evolution: a survey of the state-of-the-art, IEEE Trans. Evol. Comput. 15 (1) (2011) 4–31.
[17] J.T. Dudley, E. Schadt, M. Sirota, A.J. Butte, E. Ashley, Drug discovery in a multidimensional world: systems, patterns, and networks, J. Cardiovasc. Trans. Res. 3 (5) (2010) 438–447.
[18] Hakan Ergun, Dirk Van Hertem, Ronnie Belmans, Transmission system topology optimization for large-scale offshore wind integration, IEEE Trans. Sust. Energy 3 (4) (2012) 908–917.
[19] L.J. Eshelman, J.D. Schaffer, Real-coded genetic algorithms in genetic algorithms by preventing incest, Found. Genetic Algorithms 2 (1993) 187–202.
[20] Fbio Fabris, Renato A. Krohling, A co-evolutionary differential evolution algorithm for solving minmax optimization problems implemented on GPU using C-CUDA, Expert Syst. Appl. 39 (12) (2012) 10324–10333.
[21] C. Fernandes, A. Rosa, A study of non-random matching and varying population size in genetic algorithm using a royal road function, in: Proc. of the 2001 Congress on Evolutionary Computation, 2001, pp. 60–66.
[22] N. Hansen, A. Ostermeier, Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation, in: Proceeding of the IEEE International Conference on Evolutionary Computation (ICEC '96), 1996, pp. 312–317.
[23] Simon L. Harding, Wolfgang Banzhaf, Distributed genetic programming on GPUs using CUDA, WPABA'09: Proceedings of the Second International Workshop on Parallel Architectures and BioInspired Algorithms, 2009, pp. 1–10.
[24] Jin-Hyuk Hong, Sung-Bae Cho, Efficient huge-scale feature selection with speciated genetic algorithm, Pattern Recogn. Lett. 27 (2) (2006) 143–150.
[25] Licheng Jiao, Maoguo Gong, Shuang Wang, Biao Hou, Zhi Zheng, Qiaodi Wu, Natural and remote sensing image segmentation using memetic computing, IEEE Comput. Intell. Mag. 5 (2) (2010) 78–91.
[26] Pavel Krömer, Václav Snåšel, Jan Platoš, Ajith Abraham, Many-threaded implementation of differential evolution for the CUDA platform, in: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, GECCO '11, ACM, New York, NY, USA, 2011. 1595–1602.
[27] Stefan Lang, Panos Drouvelis, Enkelejda Tafaj, Peter Bastian, Bert Sakmann, Fast extraction of neuron morphologies from large-scale SBFSEM image stacks, J. Comput. Neurosci. 31 (3) (2011) 533–545.
[28] Antonio LaTorre, Santiago Muelas, José-María Peña, A MOS-based dynamic memetic differential evolution algorithm for continuous optimization: a scalability test, Soft. Comput. 15 (11) (2011) 2187–2199.
[29] Jih-yiing Lin, Ying-ping Chen, Analysis on the collaboration between global search and local search in memetic computation, October 15(5) (2011) 608–623.
[30] M. Locatelli, A note on the Griewank Test Function, J. Global Optim. 25 (2) (2003) 169–174.
[31] Yanping Lu, Shengrui Wang, Shaozi Li, Changle Zhou, Particle swarm optimizer for variable weighting in clustering high-dimensional data, Mach. Learn. 82 (1) (2009) 43–70.
[32] P. Merz, Bernd Freisleben, Fitness Landscapes and Memetic Algorithm Design, in: D. Corne, M. Dorigo, F. Glower (Eds.), McGraw-Hill, London, 1999.
[33] D. Molina, M. Lozano, C. García-Martínez, F. Herrera, Memetic algorithms for continuous optimization based on local search chains, Evol. Comput. 18 (1) (2010) 27–63.
[34] D. Molina, M. Lozano, F. Herrera, MA-SW-Chains: memetic algorithm based on local search chains for large scale continuous global optimization, in: 2010 IEEE Congress on Evolutionary Computation (CEC), 2010, pp. 1–8.
[35] P. Moscato. On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Toward Memetic Algorithms. Technical Report, Caltech Concurrent Computation Program, California Institute of Technology, Pasaden, 1989.
[36] P.A. Moscato, Memetic Algorithms: a Short Introduction, in: D. Corne, M. Dorigo, F. Glower (Eds.), McGraw-Hill, London, 1999.
[37] H. Mülenbein, D. Schlierkamp-Voosen, Predictive models for the breeding genetic algorithm in continuous parameter optimization, Evolutionary Computation 1 (1993) 25–49.
[38] Dongxiao Niu, Yongli Wang, Desheng Dash Wu, Power load forecasting using support vector machine and ant colony optimization, Expert Syst. Appl. 37 (3) (2010) 2531–2539.
[39] Nvidia. <http://www.nvidia.com/>.
[40] M.N. Omidvar, Xiaodong Li, A comparative study of CMA-ES on large scale global optimisation, in: AI 2010: Advances in Artificial Intelligence, 2011, pp. 303–312.
[41] Y.S. Ong, M.H. Lim, N. Zhu, K.W. Wong, Classification of adaptive memetic algorithms: a comparative study, IEEE Trans. Syst. Man. Cybern. 36 (1) (2006) 141–152.
[42] openCL. <http://www.khronos.org/opencl/>.
[43] Yong Peng, Bao-Liang Lu, A hierarchical particle swarm optimizer with latin sampling based memetic algorithm for numerical optimization, Appl. Soft Comput. 13 (5) (2013) 2823–2836.
[44] Tapta Kanchan Roy, R Benny Gerber, Vibrational self-consistent field calculations for spectroscopy of biological molecules: new algorithmic developments and applications, Phys. Chem. Chem. Phys.: PCCP 15 (24) (2013) 9468–9492.
[45] Carlos Segura, Eduardo Segredo, Coromoto León, Scalability and robustness of parallel hyperheuristics applied to a multiobjectivised frequency assignment problem, Soft. Comput. 17 (6) (2012) 1077–1093.
[46] Yihan Shao, Laszlo Fusti Molnar, Yousung Jung, Jörg Kussmann, Christian Ochsenfeld, Shawn T. Brown, Andrew T.B. Gilbert, Lyudmila V. Slipchenko, Sergey V. Levchenko, Darragh P. O'Neill, Robert a DiStasio, Rohini C. Lochan, Tao Wang, Gregory J.O. Beran, Nicholas a Besley, John M. Herbert, Ching Yeh Lin, Troy Van Voorhis, Siu Hung Chien, Alex Sodt, Ryan P. Steele, Vitaly a Rassolov, Paul E. Maslen, Prakashan P. Korambath, Ross D. Adamson, Brian Austin, Jon Baker, Edward F.C. Byrd, Holger Dachsel, Robert J. Doerksen, Andreas Dreuw, Barry D. Dunietz, Anthony D. Dutoi, Thomas R. Furlani, Steven R. Gwaltney, Andreas Heyden, So Hirata, Chao-Ping Hsu, Gary Kedziora, Rustam Z. Khalliulin, Phil Klunzinger, Aaron M. Lee, Michael S. Lee, Wanzhen Liang, Itay Lotan, Nikhil Nair, Baron Peters, Emil I. Proynov, Piotr a Pieniazek, Young Min Rhee, Jim Ritchie, Edina Rosta, C. David Sherrill, Andrew C. Simmonett, Joseph E. Subotnik, H. Lee Woodcock, Weimin Zhang, Alexis T. Bell, Arup K. Chakraborty, Daniel M. Chipman, Frerich J. Keil, Arieh Warshel, Warren J. Hehre, Henry F. Schaefer, Jing Kong, Anna I. Krylov, Peter M.W. Gill, Martin Head-Gordon, Advances in methods and algorithms in a modern quantum chemistry program package, Physical Chemistry Chemical Physics: PCCP, 8(27), July 2006, pp. 3172–3191.
[47] W. Shi, G. Wahba, R.A. Irizarry, H.C. Bravo, S.J. Wright, The partitioned LASSO-patternsearch algorithm with application to gene expression data, BMC Bioinformatics 13 (1) (2012).
[48] F.J. Solis, R.J. Wets, Minimization by random search techniques, Math. Oper. Res. 6 (1981) 19–30.
[49] K. Tang, Summary of Results on CEC'08 Competition on Large Scale Global Optimization, Technical report, Nature Inspired Computation and Application Lab (NICAL), 2008.
[50] K. Tang, Xiaodong Li, P.N. Suganthan, Z. Yang, T. Weise, Benchmark Functions for the CEC'2010 Special Session and Competition on Large Scale Global Optimization, Technical report, Nature Inspired Computation and Applications Laboratory, USTC, China, 2009.
[51] Thrust Library. <https://developer.nvidia.com/thrust>.

[52] L.Y. Tseng, C. Chen, Multiple trajectory search for large scale global optimization, in: 2008 IEEE Congress on Evolutionary Computation, 2008, pp. 3057–3064.
[53] M. Urselmann, S. Barkmann, G. Sand, S. Engell, A memetic algorithm for global optimization in chemical process synthesis problems, IEEE Trans. Evol. Comput. 15 (5) (2011). 659–283.
[54] Kush R. Varshney, Alan S. Willsky, Linear dimensionality reduction for margin-based classification: High-dimensional data and sensor networks, IEEE Trans. Signal Process. 59 (6) (2011) 2496–2512.
[55] S. Walton, O. Hassan, K. Morgan, M.R. Brown, Modified cuckoo search: a new gradient free optimisation algorithm, Chaos Solitons Fract. 44 (9) (2011) 710–718.
[56] Y. Wang, B. Li, A restart univariate estimation of distribution algorithm: Sampling under mixed gaussian and lévy probability distribution, in: 2008 IEEE Congress on Evolutionary Computation, 2008, pp. 3918–3925.
[57] Yu Wang, Bin Li, A self-adaptive mixed distribution based uni-variate estimation of distribution algorithm for large scale global optimization, Nature-Inspired Algorithms Optim. 193 (2009) 171–198.
[58] Yu Wang, Bin Li, Two-stage based ensemble optimization for large-scale global optimization, in: 2010 IEEE Congress on Evolutionary Computation (CEC), 2010, pp. 1–2.
[59] Matthieu Weber, Ferrante Neri, Ville Tirronen, Shuffle or update parallel differential evolution for large-scale optimization, Soft Comput. 15 (11) (2011) 2089–2107.
[60] D. Whitley, The GENITOR algorithm and selection pressure: why rank-based allocation of reproductive trials is best, in: Proc. of the Third Int. Conf. on Genetic Algorithms, 1988, pp. 116–121.
[61] ManLeung Wong, TienTsin Wong, Implementation of parallel genetic algorithms on graphics processing units, in: Mitsuo Gen, David Green, Osamu Katai, Bob McKay, Akira Namatame, Ruhul A. Sarker, Byoung-Tak Zhang (Eds.), Intelligent and Evolutionary Systems, Studies in Computational Intelligence, vol. 187, Springer, Berlin Heidelberg, 2009, pp. 197–216.
[62] Z. Yang, K. Tang, X. Yao, Multilevel cooperative coevolution for large scale optimization, in: 2008 IEEE Congress on Evolutionary Computation, 2008, pp. 1663–1670.
[63] Zhenyu Yang, Ke Tang, Xin Yao, Scalability of generalized adaptive differential evolution for large-scale continuous optimization, Soft Comput. 15 (11) (2011) 2141–2155.
[64] Shi-Zheng Zhao, P.N. Suganthan, S. Das, Dynamic multi-swarm particle swarm optimizer with sub-regional harmony search, in: 2010 IEEE Congress on Evolutionary Computation (CEC), 2010, pp. 1–2.
[65] Yutong Zhao, Fu Kit Sheong, Jian Sun, Pedro Sander, Xuhui Huang, A fast parallel clustering algorithm for molecular simulation trajectories, J. Comput. Chem. 34 (2) (2013) 95–104.
[66] Weihang Zhu, A study of parallel evolution strategy: pattern search on a GPU computing platform, in: Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation, GEC '09, ACM, New York, NY, USA, 2009, pp. 765–772.