Information Sciences 181 (2011) 4642–4657

Contents lists available at ScienceDirect

Information Sciences

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

Evaluation of parallel particle swarm optimization algorithms within the CUDA™ architecture

Luca Mussi [a], Fabio Daolio [b], Stefano Cagnoni [a,*]

[a] University of Parma, Department of Information Engineering, Viale G.P. Usberti 181/A, 43124 Parma, Italy
[b] University of Lausanne, HEC – Information Systems Institute, Internef 135, 1015 Lausanne, Switzerland

ARTICLE INFO

Article history:
Available online 8 September 2010

Keywords:
Particle swarm optimization
Parallel computing
GPUs
nVIDIA CUDA™

ABSTRACT

Particle swarm optimization (PSO), like other population-based meta-heuristics, is intrinsically parallel and can be effectively implemented on Graphics Processing Units (GPUs), which are, in fact, massively parallel processing architectures. In this paper we discuss possible approaches to parallelizing PSO on graphics hardware within the Compute Unified Device Architecture (CUDA™), a GPU programming environment by nVIDIA™ which supports the company's latest cards. In particular, two different ways of exploiting GPU parallelism are explored and evaluated. The execution speed of the two parallel algorithms is compared, on functions which are typically used as benchmarks for PSO, with a standard sequential implementation of PSO (SPSO), as well as with recently published results of other parallel implementations. An in-depth study of the computation efficiency of our parallel algorithms is carried out by assessing speed-up and scale-up with respect to SPSO. Also reported are some results about the optimization effectiveness of the parallel implementations with respect to SPSO, in cases when the parallel versions introduce some possibly significant difference with respect to the sequential version.

1. Introduction

Particle swarm optimization (PSO) is a simple but powerful optimization algorithm introduced by Kennedy and Eberhart [15]. PSO searches the optimum of a function, termed *fitness function*, following rules inspired by the behavior of flocks of birds looking for food. As a population-based meta-heuristic, PSO has recently gained more and more popularity due to its robustness, effectiveness, and simplicity.

In PSO, particles 'fly' over the domain of the fitness function in search of an optimum, guided by the results obtained so far. Because of this, the fitness function domain is usually also termed *search space*. A particle's position and velocity at time t can be computed using the following equations, which are the core of PSO:

$$\boldsymbol{V}(t) = w\boldsymbol{V}(t-1) + C_1R_1[\boldsymbol{X}_{best}(t-1) - \boldsymbol{X}(t-1)] + C_2R_2[\boldsymbol{X}_{gbest}(t-1) - \boldsymbol{X}(t-1)] \quad (1)$$

$$\boldsymbol{X}(t) = \boldsymbol{X}(t-1) + \boldsymbol{V}(t) \quad (2)$$

where $\boldsymbol{V}$ is the velocity of the particle, C_1, C_2 are two positive constants, R_1, R_2 are two random numbers uniformly drawn between 0 and 1, w is the so-called 'inertia weight', $\boldsymbol{X}(t)$ is the position of the particle at time t, $\boldsymbol{X}_{best}(t-1)$ is the best-fitness position reached by the particle up to time $t-1$, $\boldsymbol{X}_{gbest}(t-1)$ is the best-fitness point ever found by the whole swarm.

* Corresponding author. Tel.: +39 0521 905731; fax: +39 0521 905723.
E-mail address: cagnoni@ce.unipr.it (S. Cagnoni).

0020-0255/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2010.08.045

Despite its apparent simplicity, PSO is known to be quite sensitive to the choice of its parameters. However, under certain conditions it can be proved that the swarm reaches a state of equilibrium where particles converge onto a weighted average of their personal best and global best positions. A study of the particles' trajectory and guidelines for the choice of the inertia and acceleration coefficients C_1 and C_2, in order to obtain convergence, can be found in [29] and work cited therein.

Many variants of the basic algorithm have been developed [26], some of which have focused on the algorithm behavior when different topologies are defined for the particles' neighborhoods, i.e., swarm subsets within which particles share information [16]. A usual variant of PSO substitutes $\boldsymbol{X}_{gbest}(t-1)$ with $\boldsymbol{X}_{lbest}(t-1)$, the 'local' best position ever found by all particles within a pre-defined neighborhood of the particle under consideration. This formulation, in turn, admits several variants, depending on the topology of the neighborhoods. Among others, Kennedy and coworkers evaluated different kinds of topologies, finding that good performance is achieved using random and Von Neumann neighborhoods [16]. Nevertheless, the authors also indicated that selecting the most efficient neighborhood structure is generally a problem-dependent task.

Another important feature that may affect the search performance of PSO is the strategy according to which $\boldsymbol{X}_{gbest}$ (or $\boldsymbol{X}_{lbest}$) are updated. In 'synchronous' PSO, positions and velocities of all particles are updated one after another in turn during what, using evolutionary jargon somehow improperly, is usually called a 'generation'; this is actually a full algorithm iteration, corresponding to one discrete time unit. Each particle's 'new' position-related fitness is then computed within the same generation. The value of $\boldsymbol{X}_{gbest}$ (or $\boldsymbol{X}_{lbest}$) is only updated at the end of each generation, when the fitness values of all particles in the swarm are known.

The 'asynchronous' version of PSO, instead, allows $\boldsymbol{X}_{gbest}$ (or $\boldsymbol{X}_{lbest}$) to be updated immediately after evaluation of each particle's fitness, leading to a more 'reactive' swarm, attracted more promptly by newly-found optima. In asynchronous PSO, the iterative sequential structure of the update is lost, and the velocity and position update equations can be applied to any particle at any time, in no specific order. Regarding the effect of changing the update order or allowing some particles to be updated more often than others, Oltean and coworkers [4] have published results of an approach by which they evolved the structure of an asynchronous PSO algorithm, designing an update strategy for the particles of the whole swarm using a genetic algorithm (GA). The authors show empirically that the PSO algorithm evolved by the GA performs similarly to, and sometimes even better than standard approaches for several benchmark problems. Regarding the structure of the algorithm, they also indicate that several features, such as particle quality, update frequency, and swarm size, affect the overall performance of PSO [5].

Whatever the choices of the algorithm structure, parameters, etc., and despite good convergence properties, PSO remains an iterative process, which, depending on the problem's complexity, may require several thousands (when not millions) of particle updates and fitness evaluations. Therefore, designing efficient PSO implementations is an issue of great practical relevance. It becomes even more critical if one considers real-time applications to dynamic environments in which, for example, the fast-convergence properties of PSO are used to track moving points of interest (maxima or minima of a specific dynamically-changing fitness function) in real time. This is the case, for example, of computer vision applications in which PSO has been used to track moving objects [20], or to determine location and orientation of objects or posture of people [11,22].

Recently, the use of GPU multi-core architectures for general-purpose, high-performance parallel computing has been attracting researchers' interest more and more, especially after handy programming environments, such as nVIDIA™ CUDA™ [24], were introduced. Such environments or APIs exploit the computing capabilities of the GPUs using parallel versions of high-level languages which require that only the highest-level details of parallel process management be explicitly encoded in the programs. The evolution of both GPUs and the corresponding programming environments has been extremely fast and, up to now, far from any standardization. Because of this, not only is performance of implementations based on different architectures or compilers very hard to compare, so are the same programs run on different releases of software-compatible hardware. Execution time is therefore often the only direct objective quantitative parameter on which comparisons can be based.

In this paper we describe our experience in parallelizing PSO within CUDA™. The next section deals with the problem of PSO parallelization, and is followed by a brief introduction to the main features of CUDA™ (Section 3), based on which the two parallel algorithms studied have been derived. Section 4 is a detailed description of the two parallel PSO implementations, dealing with their advantages and disadvantages. In Section 5 the results obtained on classical benchmark functions are summarized and compared to a sequential implementation of Standard PSO (SPSO) [2] and to other recent parallel PSO implementations based on CUDA™. Some concluding remarks and plans for future extensions and applications of this work are finally reported in Section 6.

2. PSO parallelization

The structure of PSO is very close to being intrinsically parallel. In PSO, the only dependency existing between processes which update the swarm's velocities and positions is related to data which must be shared between particles, within either the whole swarm or within pre-defined neighborhoods. These data are either only $\boldsymbol{X}_{gbest}$, the best position in the search space that any member of the swarm has visited so far, or the corresponding vector $\boldsymbol{X}_{lbest}$ of the best positions found by each member of each particle's neighborhood. PSO can therefore be naturally implemented very efficiently using parallel computing architectures, by either coding synchronous PSO, which directly removes dependencies between updates, or relaxing

synchronicity constraints and allowing particles to rely on information which may not be strictly up-to-date. Among the several approaches to PSO parallelization, many of the most recent ones are GPU implementations [3,32,33,36].

As a matter of fact, there are several similarities between PSO and evolutionary algorithms, even if these are more structural than conceptual.[1] Therefore, in studying the problem of PSO parallelization, it is possible to rely on previously published work about parallelization of evolutionary algorithms, and review previous approaches according to the same taxonomy [1].

The most straightforward technique is to simply distribute fitness evaluations following a *master–slave* paradigm, in which individuals (particles) or partitions of a single population (swarm) are dispatched by a centralized controller to a parallel computing environment for fitness evaluation. Following this approach, which introduces no variations from an algorithmic point of view, in 2003 Gies and Rahmat-Samii [7] and Schutte et al. [27] first implemented and successfully applied parallel PSO to the problems of antenna design and biomechanical system identification, respectively.

A more complex parallel model is given by *coarse-grained* algorithms, which are characterized by multiple independent or loosely connected populations (swarms) occasionally exchanging individuals (particles) according to some fitness-based strategy. This approach corresponds to the so-called 'island model' [6] and normally implies little communication between the distributed groups of individuals. In PSO, different swarms can be placed on different islands, and only the local best of each swarm is occasionally requested to 'migrate' to another island according to some pre-defined connection topology. Focusing on communication strategies, Chang and coworkers [12] proposed a parallel implementation in which three different migration and replacement policies were possible: they showed that the choice of one communication strategy can be problem-dependent but all the variants they proposed appeared to be more effective than standard PSO. Focusing on the enhanced neighborhood topologies induced by different communication strategies in multiprocessor architectures, Waintraub et al. [31] have recently proposed and investigated variants of the coarse-grained model for a nuclear engineering application.

At a lower parallelization level, the *fine-grained* paradigm comprises algorithms in which a single population (swarm) is distributed on a one- or two-dimensional toroidal grid, limiting interactions between individuals (particles). This means that each particle can interact only with a limited subset of its neighbors. This approach, also called 'cellular model', has the advantage of preserving greater diversity, but introduces significant communication overhead: this drawback mostly affects distributed computation environments. On the other hand, the cellular model permits to obtain the highest degree of parallelism. In fact, the first GPU implementation of PSO, proposed in 2007 by Li et al. [18], referred to a fine-grained parallel PSO, although still implemented by mapping the algorithm onto texture-rendering hardware and not directly within a GPU programming environment. An overview of published work according to granularity analysis can be found in [31].

As reported in Section 1, another choice which may affect parallel PSO performance is the best positions update policy, which can be synchronous or asynchronous. Earlier parallel variants belong to the synchronous class. The work by Venter and Sobieszczanski-Sobieski [30] and by Koh et al. [17] opened the way to asynchronous implementations. A classification of approaches from this point of view is reported in [34]. Most applications reported in literature have been run on distributed systems. Asynchronous update is therefore usually claimed to be better because it does not introduce idle waiting times, which are likely to occur in uneven environments when processes are distributed on processors having different performances [17]. A GPU implementation, to be discussed in the following, imposes more constraints on information sharing than on computing, and needs to rely on synchronous parallel execution as much as possible.

One may wonder whether synchronization affects the quality of results from the point of view of convergence to the optimum of the function under consideration. The influence of synchronization can be assessed by estimating the probability of success in finding the global best on a set of test functions as, for instance, those included in the benchmark described in [8,9]. To this purpose, we have preliminarily tested sequential implementations of both policies. On the basis of this preliminary analysis, postponing the update of the particles' best positions to the end of each iteration does seem to affect the effectiveness of PSO as a global optimization task, unless a sparsely connected topology is employed. In particular, results show that synchronizing the personal best updates on a fully-connected swarm communication network causes PSO performance to decrease with the increase of problem dimension, because it negatively affects population diversity and the algorithm's search ability [21,28].

In the first implementations of PSO, particles were organized as a fully- connected social network, best known as global-best topology. In fact, the PSO algorithm which has been used as sequential reference in evaluating results of our parallel GPU implementation [2,14] uses a fixed-ring topology for consistency with our parallel implementation; the Standard PSO actually employs a stochastic star topology in which each particle informs a constant number K of random neighbors. Anyway, the suggested default K value of three results in an algorithm which is very similar to the ones that rely on a classical ring topology for both structure and performance. We will not deal with this problem any further in this paper, as discussion would be lengthy and mostly out of the scope of our work.

3. The CUDA™ architecture

In November 2006 nVIDIA™ introduced a new general-purpose parallel computing architecture called CUDA™, a handy tool to develop scientific programs oriented to massively parallel computation. It is actually sufficient to install a compatible

[1] Incidentally, this has led to PSO being 'adopted' by the evolutionary computing community, within which such an algorithm is mainly studied.

GPU and the CUDA™ SDK, even in a low-end computer, to develop parallel programs using a high-level language (C, in this case).

CUDA™'s programming model requires that the programmer partition the problem under consideration into many independent sub-tasks which can be solved in parallel. Each sub-problem may be further divided into many tasks, which can be solved cooperatively in parallel too. In CUDA™ terms, each sub-problem becomes a *thread block*, each thread block being composed of a certain number of *threads* which cooperate to solve the sub-problem in parallel. The software element that describes the instructions to be executed by each thread is called *kernel*. When a program running on the CPU invokes a kernel, the number of corresponding thread blocks and the number of threads per thread block must be specified. The abstraction on which CUDA™ is based allows a programmer to define a two-dimensional grid of thread blocks; each block is assigned a unique pair of indices that act as its coordinates within the grid. The same mechanism is available within each block: the threads that compose a block can be organized as a two- or three-dimensional grid. Again, a unique set of indices is provided to assign each thread a 'position' within the block. This indexing mechanism allows each thread to personalize its access to data structures and, in the end, achieve effective problem decomposition.

From a hardware viewpoint, a CUDA™-compatible GPU is made up of a scalable array of multithreaded Streaming Multiprocessors (SMs), each of which is able to execute several thread blocks at the same time. When the CPU orders the GPU to run a kernel, thread blocks are distributed to free SMs and all threads of a scheduled block are executed concurrently. If a block cannot be scheduled immediately, its execution is delayed until some other block terminates its execution and frees some resources. This permits a CUDA™ program to be run on any number of SMs, relieving the programmer from managing process allocation explicitly. One key aspect about SMs is their ability to manage hundreds of threads running different code segments: in order to do so they employ an architecture called SIMT (Single Instruction, Multiple Thread) which creates, manages, schedules, and executes groups (warps) of 32 parallel threads. The main difference from a SIMD (Single Instruction, Multiple Data) architecture is that SIMT instructions specify the whole execution and branching behavior of a single thread. This way a programmer is allowed to write parallel code for independent scalar threads, as well as code for parallel data processing which will be executed by coordinated threads.

Each SM embeds eight scalar processing cores and is equipped with a number of fast 32-bit registers, a parallel data cache shared between all cores, a read-only constant cache and a read-only texture cache accessed via a texture unit that provides several different addressing/filtering modes. In addition, SMs can access local and global memory spaces which are (non-cached) read/write regions of device memory: these memories are characterized by latency times about two order of magnitude larger than the registers and the texture cache. Since internal SM resources are distributed among all the threads being executed at the same time, the number of active blocks depends on how many registers per thread and how much shared memory per block are needed for the kernels being executed.

In order to obtain the best from this architecture, a number of specific programming guidelines should be followed, the most important of which are: (a) minimize data transfers between the host and the graphics card; (b) minimize the use of global memory: shared memory should be preferred; (c) ensure global memory accesses are coalesced whenever possible; (d) avoid different execution paths within the same warp.

Moreover, each kernel should reflect the following structure: (i) load data from global/texture memory; (ii) process data; and (iii) store results back to global memory.

An in-depth analysis of the architecture and more detailed programming tips can be found in [23,24].

4. Parallel PSO algorithms within the CUDA™ architecture

The most natural way to remove the dependence between particles' updates which, as described in Section 2, is the main obstacle to PSO parallelization, would be implementing the 'synchronous PSO', in which $\boldsymbol{X}_{gbest}$ or $\boldsymbol{X}_{lbest}$ are updated at the end of each generation only. We used this approach in the first parallel implementation of PSO we considered, which we termed *SyncPSO*, whose most salient feature is the absence of accesses to global memory. However, despite being very 'clean' and possibly the most efficient at the abstract algorithm level (i.e., independently of fitness function complexity), this solution imposes some limitations on the implementation of the fitness function and uses computing resources inefficiently when only one or few swarms are simulated. A second parallel implementation we designed, termed *RingPSO*, tries to overcome these limitations by relaxing the synchronization constraint and allowing the computation load to be distributed over all SMs available.

The two algorithms and their main implementation issues are reported in the following sections.

4.1. Basic parallel PSO design

PSO can be most naturally parallelized in CUDA™ by dividing PSO into as many threads as the number N_p of particles in the swarm; each thread implements the update equations for one particle. In order to avoid using time-consuming global memory accesses, one can keep all data related to one particle's status inside the local registers of the corresponding thread and allow particles to exchange information about the global best position via the shared memory.

A first apparent drawback of this solution is given by the constraint, imposed by CUDA™, by which threads can communicate by means of shared memory only if they all belong to the same thread block. Thus, if the same thread is to update one

particle's status throughout the generations, a single thread block must be used for a whole swarm. Depending on the GPU computing capability, this limits the maximum number of particles in a swarm to the maximum number of threads allowed for each block by the GPU architecture. This limit currently ranges between 256 and 512 threads per block. However, it has little practical impact, since a rule of thumb [14] suggests that the optimal swarm size for optimization of a D-dimensional problem be roughly equal to $10 + 2 \cdot \sqrt{D}$. This implies that, even within a limit of 256 threads per block, swarms could be able to effectively solve problems of dimension equal to a few thousands.

Another problem to be addressed is how to allow particles to update $\boldsymbol{X}_{gbest}$, or $\boldsymbol{X}_{lbest}$, concurrently. The standard conditional sentence which follows a particle's fitness evaluation and checks whether its new fitness value is better than the best found so far unavoidably leads to concurrent accesses to shared memory, if the test succeeds for more than one thread. We solved this problem by allocating a vector of size N_p in shared memory that stores the best-fitness values of all particles, so that each particle acts on a different memory location. The current best fitness of the swarm can then be computed as the minimum of such a vector. Doing so, one can further exploit the parallelism between threads and perform a reduction (see, for example, [13]) of the vector. After that, only the computed minimum needs to be compared to the global or neighborhood best fitness to update $\boldsymbol{X}_{gbest}$ (or $\boldsymbol{X}_{lbest}$). This update is performed by the first thread, while other threads wait for its completion. This way, only $log_2(N_p)$ comparison steps are needed to compute the current minimum fitness value, instead of N_p needed by a sequential implementation. Moreover, concurrent shared memory accesses are completely avoided.

To complete the description of *SyncPSO*, we still need to discuss the implementation of the fitness function and give details about the data structures we have used. As for the first point, fitness computation is nothing more than a sequential piece of code within each thread associated with a particle. As regards data structures, Fig. 2 shows how we organized, in global memory, the vectors containing positions, velocities, fitnesses and random seeds of all particles, as well as $\boldsymbol{X}_{gbest}$ (or $\boldsymbol{X}_{lbest}$) and the corresponding fitness value(s). This organization allows one to transparently manage a variable number of particles, as well as a variable number of swarms. Actually, the organization shown in the figure refers to the parallel PSO implementation presented in the next section. The one used for *SyncPSO* is very similar. Moreover, since *SyncPSO* aims at avoiding global memory accesses at all, this point is of minor importance here.

As regards thread memory allocation, each thread/particle stores its current position, velocity, best position and best-fitness values in local thread registers; the current fitness, the global best fitness and the global best position values are kept in the shared memory of the corresponding thread block, in order to permit inter-particle communications.

The very last issue regards pseudo-random numbers generation: since GPUs do not offer such a primitive function, one must write a pseudo-random generator. Moreover, each thread must use a different seed to prevent all threads from using the same sequence.

In summary, *SyncPSO* implements the PSO algorithm using just one CUDA™ kernel per swarm, within which each particle in the swarm is simulated by a single thread. Only one thread block is needed to simulate a whole swarm, but several swarms can be run at once by scheduling as many identical thread blocks. This way, several PSO runs can be executed on the same problem in parallel. Each thread block must include a number of threads equal to the number of particles in the swarm. The single main kernel implements the algorithm presented in Listing 1. Initialization of particles' position and velocity, as well as the initial fitness evaluation, can be performed by other dedicated kernels before scheduling the main one.

Although we were able to obtain rather impressive results (reported in more detail in the next section), all that glitters is not gold! In fact, as shown in Fig. 1, beyond a certain dimension, the dependence of execution time on problem dimension changes slope. This happens because of the limited internal resources an SM can dedicate to each thread: there exists a point beyond which the number of local registers is no longer sufficient to store all the particles' data. With reference to Fig. 1, from the point in the graph highlighted by the arrow, the compiler starts using local thread memory to store excess data, but this kind of memory is as slow as the global memory. This nullifies the advantages of the strategy followed so far to design a parallel PSO without global memory access, which becomes ineffective with high-dimensional problems.

Therefore, we have examined alternative solutions to the problem of designing a parallel PSO capable of dealing with high-dimensional problems, while allowing one to run at least a small number of swarms in parallel. An immediate observation that can be made about *SyncPSO* is obvious: if only one swarm is to be simulated, one cannot take full advantage of the parallel computation the GPU is capable of. Furthermore, with this design, the parallelism of the GPU cannot be exploited in computing a particle's fitness, since fitness computation must be implemented as sequential code within the thread associated with the particle. This is a very hard limitation, especially in the case of complex fitness functions which process large amounts of data. In fact, in most sequential evolutionary algorithms, as well as in PSO, fitness computation is often the most computation-intensive module, such that the number of fitness evaluations is usually considered to be a significant measure of execution 'time' when different evolutionary algorithms are compared.

Based on these considerations, we designed a new parallel PSO algorithm, termed *RingPSO*, to verify whether distributing the computation over many kernels (one for each of the main processing stages of PSO) could improve performances by exploiting more than one SM at the same time. As a drawback, such a design requires that the global memory be accessed for loading and storing the current status of the swarms every time one switches from one kernel to the next one. However, also *SyncPSO*, in practice, must access global memory beyond a certain problem dimension, which makes this feature less penalizing than could be theoretically expected.

The following section describes the main features of *RingPSO*.

```
if (threadIdx.x == 0){
    <The first thread loads shared global best data from global memory>
}
__syncthreads();
<load particle data from global memory based on thread indexes>
for (int i = 0; i < generationsNumber; i++){
    <Update the position of the particle>
    <Evaluate the fitness value>
    if (currentFitness < currentBestFitness){
        <Update the personal best position>
    }
    __syncthreads();
    <Reduce in parallel the fitness vector to find its minimum>
    __syncthreads();
    if (threadIdx.x == 0)
        if (fitnesses[0] < globalBestFitness){
          <Update the global best data>
        }
    __syncthreads();
}
<Store particle data back to global memory>
if (threadIdx.x == 0){
  <The first thread stores global best data back to global memory>
}
```

Listing 1. SyncPSO kernel.

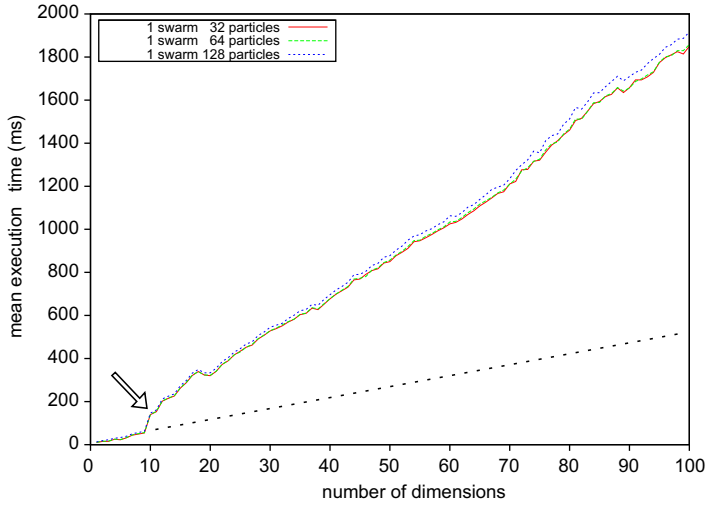


Fig. 1. When the number of dimensions grows, local resources on SMs are not enough to store all the thread's data, and local memory must be used. The arrow indicates the point at which this happens; the heavier dotted line represents the ideal theoretical trend which could be expected when using ideal SMs with unlimited internal resources.

4.2. Multi-kernel parallel PSO algorithm

To take full advantage of the massively parallel computation capabilities offered by CUDA™ in developing a parallel PSO algorithm, one needs to completely revise the design of *SyncPSO* by considering the main stages of the algorithm as separate tasks, each of which can be parallelized in a different manner.

This way, each stage can be implemented as a different kernel and optimization can be achieved by iterating the basic kernels needed to perform one PSO generation. To better clarify this concept we refer to Listing 2, which shows the pseudo-code the CPU must execute to launch a parallel PSO optimization process on the GPU. Actually, this listing is equivalent to the flow chart of a synchronous PSO algorithm, regardless of its sequential or parallel implementation. Since the sequential activation of the kernels must be kept, while each kernel must be executed independently of the others, each kernel must load all the data it needs initially and store the data back at the end of its execution. The only way of sharing data among

```
<Initialize positions/velocities of all particles>
<Perform a first evaluation of the fitness function>
<Set initial personal/global bests>
for(int i = 0; i < generationsNumber; i++){
    <Update the position of all particles>
    <Re−evaluate the fitness of all particles>
    <Update all personal/global bests>
}
<Retrieve global best information to be returned as final result>
```

Listing 2. Synchronous PSO pseudo-code.

different kernels offered by CUDA™ is to keep them in the global memory. The current status of PSO must hence be saved there, introducing a number of very slow memory accesses in each thread of all kernels. Therefore, one of the first details worth considering is how to limit the number of such accesses and organize PSO data in order to exploit the GPU coalescing capability.

For instance, let us consider a particle's coordinates update performed by different threads of the same thread block. Since one thread is used to update each coordinate, in order to obtain coalesced accesses it is necessary to access consecutive locations of $\mathbf{X}(t)$ when the kernel code loads the current position from the global memory. In other words, all threads in a block must retrieve a particle's position using their unique set of indices to determine the index *posID* of the vector which is to be read, according to an expression similar to the following:

$$posID = (swarmID \cdot n + particleID) \cdot D + dimensionID \tag{3}$$

where n represents the number of particles belonging to one swarm, D represents the problem dimension, and *dimensionID*, *swarmID* and *particleID* are the indices to be used to identify one particle's coordinates. If a kernel uses a grid of thread blocks having a number of rows equal to the number of swarms and a number of columns equal to the number of particles per swarm, each thread block is identified by a unique pair (*swarmID*, *particleID*) of coordinates. Then, if one considers a single thread block to be composed of a number of threads equal to D, the thread index can be used as *dimensionID*. This way each thread can load (and possibly update) a different coordinate of a particle. More importantly, each group of 32 threads run in parallel can access adjacent elements of the same vector, performing a coalesced global memory access characterized by the same latency time as a single access. It is easy to see how helpful this pattern can be to optimize parallelization of the particles' position update. This is actually permitted by the position update equation of PSO, which is separable based on coordinates.

Fig. 2 shows how we organized all the status vectors of the swarms to reflect this access pattern and to transparently manage a variable number of particles, as well as a variable number of swarms. Even more significantly, this organization allows to coalesce global memory accesses among threads to increase code efficiency. The D elements used to encode one particle are always packed into a vector whose size is a multiple of 16, so that parallel read operations (for example, once

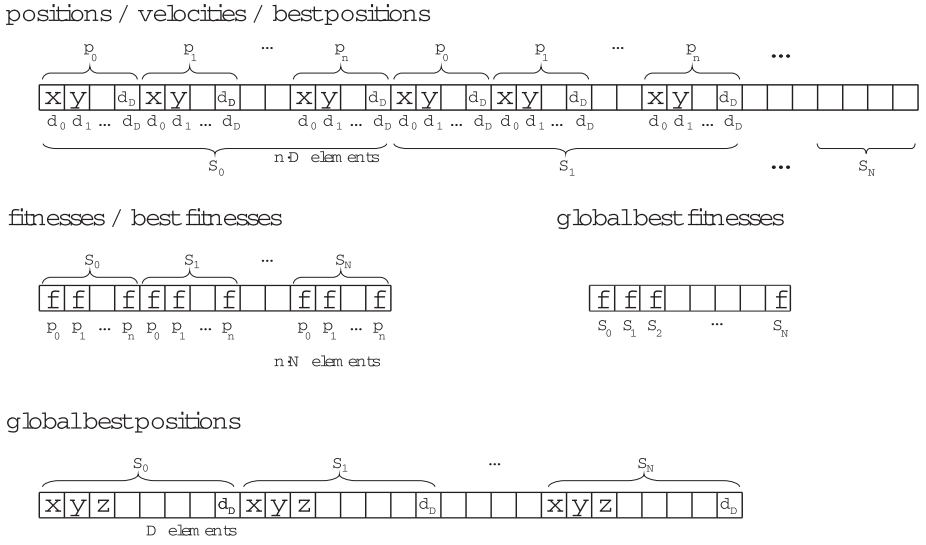


Fig. 2. Global memory data organization.

again, loading the current position of a particle) can perform a unique coalesced read operation which satisfies CUDA™'s requirements for byte alignment.[2] At the time of fitness evaluation, only the coordinates which are actually needed are taken into consideration.

Deriving the design of *RingPSO* from these considerations is now rather straightforward. A first kernel (*PositionUpdateKernel*) updates the particles' positions by scheduling a number of thread blocks equal to the number of particles; each block updates the position of one particle running a number of threads equal to the problem dimension D. Global best, or local best, information must be stored in global memory, ready to be loaded by the thread blocks which compute the basic PSO update equations.

A second kernel (*FitnessKernel*) is used to compute the fitness. Again, each scheduled thread block computes the fitness of one particle, typically using one thread for each position coordinate. This is particularly advantageous for those fitness functions which can be expressed as a sum of functions of the coordinates: after each thread has computed one term of the sum, based on its associated coordinate, the final sum can be determined by means of parallel reduction, taking further advantage of parallel computation.

In order to complete one PSO generation, one last kernel (*BestUpdateKernel*) is needed to update $\boldsymbol{X}_{gbest}$ (or $\boldsymbol{X}_{lbest}$). Since its structure must reflect the swarm topology, the number of thread blocks to be scheduled may vary from one per swarm, in case a global-best topology is used, to many per swarm (to have one thread per particle), in case of ring topology. *BestUpdateKernel*, besides updating the particles' personal bests, stores the index of the global/local best(s) in global memory, since *PositionUpdateKernel* needs this value to load global/local best positions and compute the update equations.

Pseudo-random numbers are directly generated on the GPU by the Mersenne Twister kernel available in the CUDA™ SDK. On the basis of the amount of device memory available, we run this kernel every given number of PSO generations. Pseudo-random numbers are stored in a dedicated array which is 'consumed' by other kernels generation after generation. When there are no numbers left, the Mersenne Twister kernel is run again and the cycle restarts.

Following this design, we implemented different multi-kernel versions of parallel PSO using the global, random and ring topology. Recalling that this second approach was chosen to prevent local memory accesses which hamper efficiency with high-dimensional problems, and that performances of a parallel PSO with global-best topology decrease as problem dimension increases (see Section 2), in the next section we are only going to report results obtained using *RingPSO*, which is a multi-kernel PSO implementation based on a ring topology.

One peculiar aspect of *RingPSO* lies in the synchronization between the updates of the particles' bests. Each particle can determine its local best by accessing the best-fitness values of its two neighbors independently of other particles. If we assume that $\boldsymbol{X}_{lbest}$ is updated just after $\boldsymbol{X}_{best}$ is updated, within the same kernel, and consider that there is no guarantee about both the order of execution of the scheduled thread blocks and the time in which data written onto global memory by one block can be available for other blocks (unless specific slow synchronizing mechanisms are employed), local best information seen by a certain particle may happen not to be strictly up-to-date. In any case this should not be too serious a problem since, on the one hand, it slows down inter-particle communications only slightly while, on the other hand, it preserves diversity between individuals.

This version has additional advantages if one swarm only is to be simulated to optimize simple objective functions: in that case the three kernels can be fused together, avoiding almost all global memory access. Doing so, one must carefully limit the number of particles based on the number of SMs available on the GPU and on the resource usage of the kernel, since thread blocks in excess of those that can be executed in parallel must wait until the full completion of the simulation of some of the others before being scheduled.

In any case, when dealing with very complex and highly parallelizable fitness functions, it might still be better to design separate kernels to make the kernel dedicated to fitness computation independent of the others.

5. Results

We tested the different versions of our parallel PSO implementation on a 'classical' benchmark which comprised a set of functions which are often used to evaluate stochastic optimization algorithms. Since our goal was to compare different parallel PSO implementations with one another and with a sequential implementation, rather than discussing the performances of PSO itself, in all tests we set PSO parameters to the 'standard' values suggested in [14]: $w = 0.729844$ and $C_1 = C_2 = 1.49618$.

Following the results of the theoretical analysis reported in the previous sections, which show that *SyncPSO*, despite being algorithmically more efficient than *RingPSO*, tends to saturate GPU resources much sooner with problem dimensions, we limited our tests of *SyncPSO* to a problem size which did not saturate SM's resources. This allowed to evaluate results that are as independent as possible of CUDA™'s internal scheduling strategies and to highlight mainly the algorithmic aspects and the basic advantages that a parallel implementation offers with respect to the sequential version (SPSO) taken as reference.

Tests of *RingPSO*, instead, were much more extensive, since this version scales much better with problem dimensions and therefore is generally preferable in most difficult real-world situations. To have an idea of the performances achievable by GPUs having different technical features and prices, tests were performed on three graphic cards (see Table 1 for detailed specifications); in all these tests we used the performances of the SPSO run on a top-performing CPU as reference.

[2] Since we use the *float* data type, 16 elements correspond to 64 bytes, which is the required minimum alignment.

Table 1
Major technical features of the GPUs used for the experiments.

Model name	GeForce 8600GT	GeForce EN8800GT	GeForce GTX280	GeForce GTX260AMP^2	Quadro FX5800
Short name used in figures		GF		GT	Qu
GPU	G84	G92	G200	G200	G200
GPU clock (MHz)	540	600	602	650	650
Stream multi processors	4	14	30	27	30
Stream processors	32	112	240	216	240
Bus width (bit)	128	256	512	448	512
Memory (MB)	256	1024	1024	896	4096
Memory clock (MHz)	700	1800	2214	2100	1600
Memory type	GDDR3	GDDR3	GDDR3	GDDR3	GDDR3
Memory bandwidth (GB/s)	22.4	57.6	141.7	117.6	102.0

5.1. SyncPSO

SyncPSO was tested on a rather basic hardware configuration in which an Asus GeForce EN8800GT GPU was installed in a PC based on an Intel Core2 Duo™ CPU, running at 1.86 GHz.

Fig. 3a reports the execution times, averaged over 100 consecutive runs, necessary for a single swarm of 32, 64, and 128 particles to run *SyncPSO* on the generalized 5-dimensional Rastrigin function vs. the number of generations: the execution time is almost insensitive to the number of particles (all threads are in fact executed in parallel in groups of 32), and the time needed to perform about 1.3 million fitness evaluations is about one third of a second for the most computationally-intensive cases (one swarm of size 128 run for 100,000 generations).

Fig. 3b shows how the average execution time needed to run 10,000 generations of one swarm with 32, 64 and 128 particles scales with respect to the dimension of the generalized Rastrigin function (up to nine dimensions). Again, execution time is almost insensitive to the number of particles: about 62 ms to run 10,000 generations of a swarm with 128 particles.

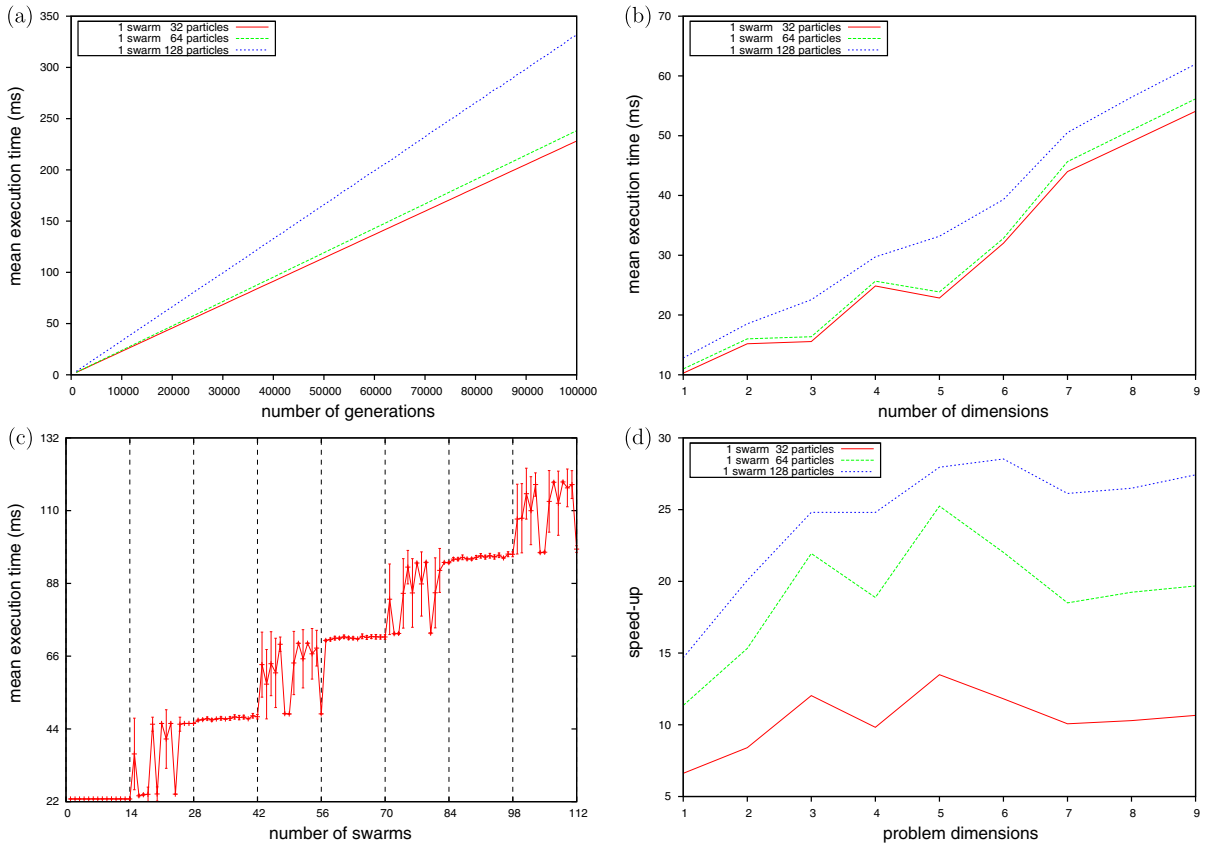


Fig. 3. *SyncPSO* test results in optimizing the Rastrigin Function: (a) average execution times vs. number of generations in five dimensions; (b) average execution times after 10,000 generations vs. problem dimension; (c) average execution times for 10,000 generations in five dimensions vs. number of swarms of size 32; and (d) speed-up results vs. problem dimensions with respect to the sequential synchronous PSO.

In Fig. 3c it is possible to see how execution time depends on the number of swarms: plotted data refer to average times, and corresponding standard deviations, over 100 consecutive runs.

Since the GPU used for these experiments has 14 SMs available and each swarm uses one thread block only, interpretation of these results is straightforward: the first 14 swarms are executed on different SMs in parallel, taking the same time as just one swarm; further raising the number of swarms implies that the GPU starts applying its scheduling strategies to distribute the execution of all thread blocks among SMs available. The profile of the *SyncPSO* kernel obtained using the tool released by nVIDIA™ reveals that one thread block compiled to optimize the generalized Rastrigin function in five dimensions with 32 particles is already rather demanding in terms of SM resources. In order to keep all particles' data inside local registers and avoid global memory accesses, almost half the registers available to one SM must be used. This seems to imply that, for this function, no more than two thread blocks can be executed in parallel on a single SM: this is probably the reason for the weird transitory trend shown for a number of swarms between 14 and 28. Although one could expect to observe, in that interval, a constant execution time (another horizontal segment) located somewhere in between 22 and 44 ms, the GPU seems to have problems in managing two blocks per SM at the same time efficiently. We performed some further tests to understand this behavior, e.g., to see if it could be attributed to some stochastic effect related to the branching instructions inside the PSO algorithm, but we could not find any explanation independent of the internal (and transparent to programmers) scheduling of CUDA™. We could observe that the execution time doubles, on average, every third block to be assigned to a single SM, and that the peculiar transitory behavior discussed above seems to repeat itself regularly.

These observations about the scale-up properties of the algorithms were confirmed by subsequent tests on the most powerful card we had available (an nVIDIA™ Quadro FX5800 equipped with 30 SMs having twice as many internal resources; see Table 1). In particular, we observed that, consistently with a roughly twofold increase in both SM number and internal resources available for each SM, the peculiar transitory behavior occurs with a frequency which is about one quarter with respect to the basic hardware configuration used in our tests. We actually observed one stepwise increase in execution time every 30 swarms instead of every 14, while the peculiar transitory trend occurs between 90 and 120 swarms, i.e. exactly when the scheduler is executing four blocks (instead of two) on each SM.

Apart from these scheduling issues, in Fig. 3c we can see that the execution of 10,000 generations for 112 swarms of 32 particles on the generalized Rastrigin function in five dimensions takes about 120 ms on the cheaper graphic card.

Finally, Fig. 3d reports speed-up results with respect to an equivalent sequential implementation (SPSO compiled with the maximum-optimization option of *gcc 4.2.4*): in optimizing the Rastrigin function in nine dimensions for 10,000 generations with 128 particles, the speed-up is close to 30 times.

5.2. RingPSO

Tests for *RingPSO* were performed on three different graphic cards: an nVIDIA™ Quadro FX 5800, a Zotac GeForce GTX260 AMP^2 edition and an Asus GeForce EN8800GT (see Table 1 for detailed specifications). For the following experiments the sequential SPSO was run on a PC powered by a 64-bit Intel(R) Core(TM) i7 CPU running at 2.67 GHz.

Figs. 4–6 refer to *RingPSO* and compare average execution times and average final fitness values obtained for problem dimension D ranging from 2 to 128 in optimizing the following fitness functions (a typical testbed for the PSO [10]): (a) the simple Sphere function within the domain $[-100, 100]^D$, (b) the Rastrigin function, on which PSO is known to perform well, within the domain $[-5.12, 5.12]^D$ and (c) the Rosenbrock function, which is non-separable and thus difficult to solve by PSO, within the domain $[-30, 30]^D$.

The following implementations of PSO were compared: (1) the sequential SPSO version modified to implement the ring topology; (2) the 'basic' three-kernel version of *RingPSO*; (3) *RingPSO* implemented with two kernels only (one kernel which fuses *BestUpdateKernel* and *PositionUpdateKernel*, and *FitnessKernel*). Values were averaged over the 98 best results out of 100 runs.

We decided to analyze these two different versions of *RingPSO* to emphasize the effect of relaxing the synchronization constraint on both convergence performances and execution times. The difference between the two can be better appreciated recalling that a new kernel never starts before a previous kernel has completed its execution and has written all shared data onto global memory (i.e., caching mechanisms have been terminated). Hence, the three-kernel *RingPSO* is the one which is the most subject to synchronization constraints, of the two parallel versions under consideration. On the opposite end, the two-kernel *RingPSO* imposes less synchronism among thread blocks (i.e. particles). Due to the stochastic nature of the PSO algorithm (personal and local best updates do not occur at every generation), it may be possible for one particle to have its position updated a little earlier, or later, than others.

At the same time, reducing the number of kernels avoids a number of operations of load/store of the swarm status from/onto global memory imposed by the scheduling of every new kernel. This results in shorter execution times and thus in better speed-up performances.

A further analysis of Figs. 4–6 reveals the effect of the relaxation of the synchronicity constraint on convergence performances: the two-kernel version seems to offer the best performances, on average.

Taking speed-up values into consideration, one can notice that the best performances were obtained on the Rastrigin function, probably due to the presence of a trigonometric function in its equation. In fact, GPUs have internal *fast math* functions which can provide good computation speed at the cost of slightly lower accuracy, which causes no problems in this case.

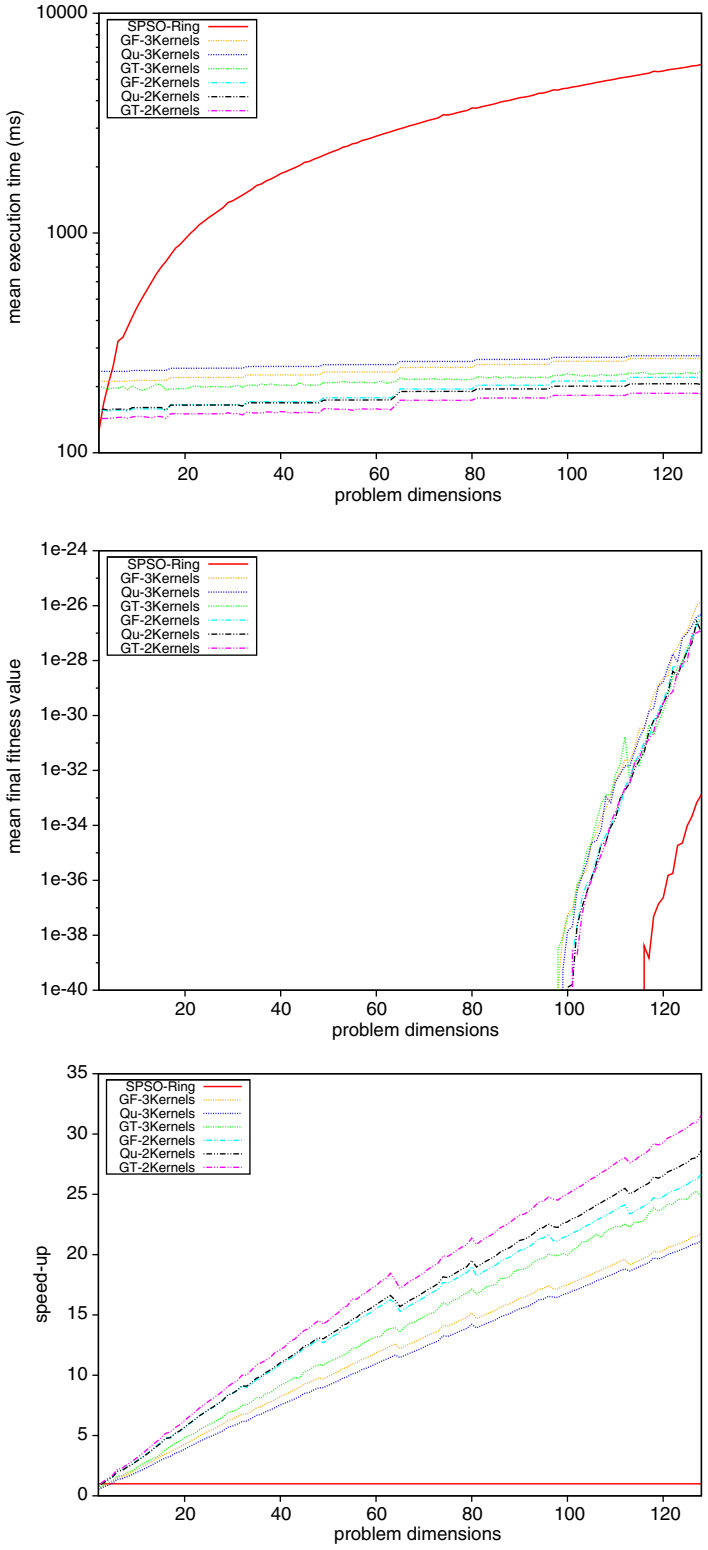


Fig. 4. Average execution times (above), average final fitness values (center) and speed-ups (below) vs. problem dimension for the Sphere function. Experiments were performed running one swarm of 32 particles for 10,000 generations. Values plotted for both *RingPSO* and *SPSO* were averaged over the best 98 results out of 100 runs.

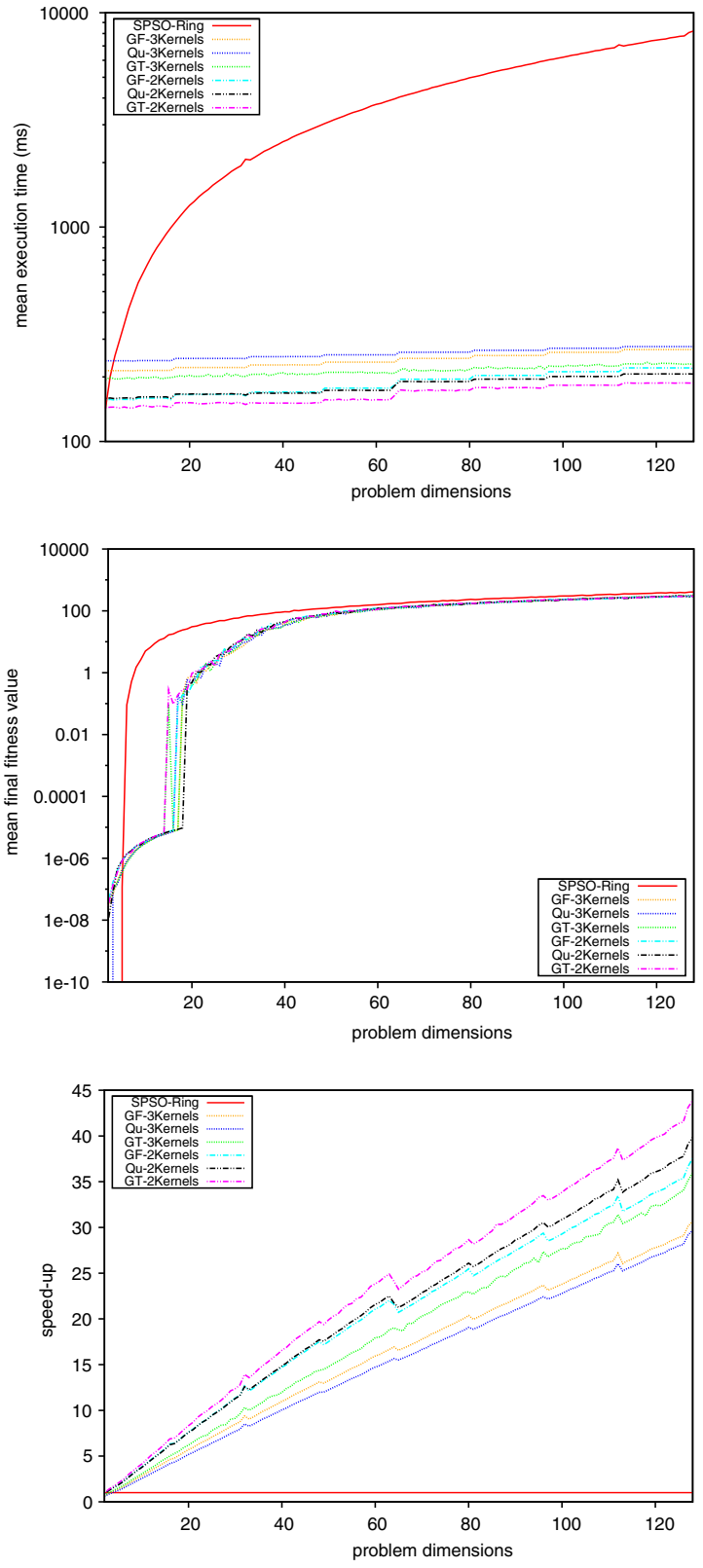


Fig. 5. Average execution times (above), average final fitness values (center) and speed-ups (below) vs. problem dimension for the Rastrigin function. Experiments were performed running one swarm of 32 particles for 10,000 generations. Values plotted for both *RingPSO* and *SPSO* were averaged over the best 98 results out of 100 runs.

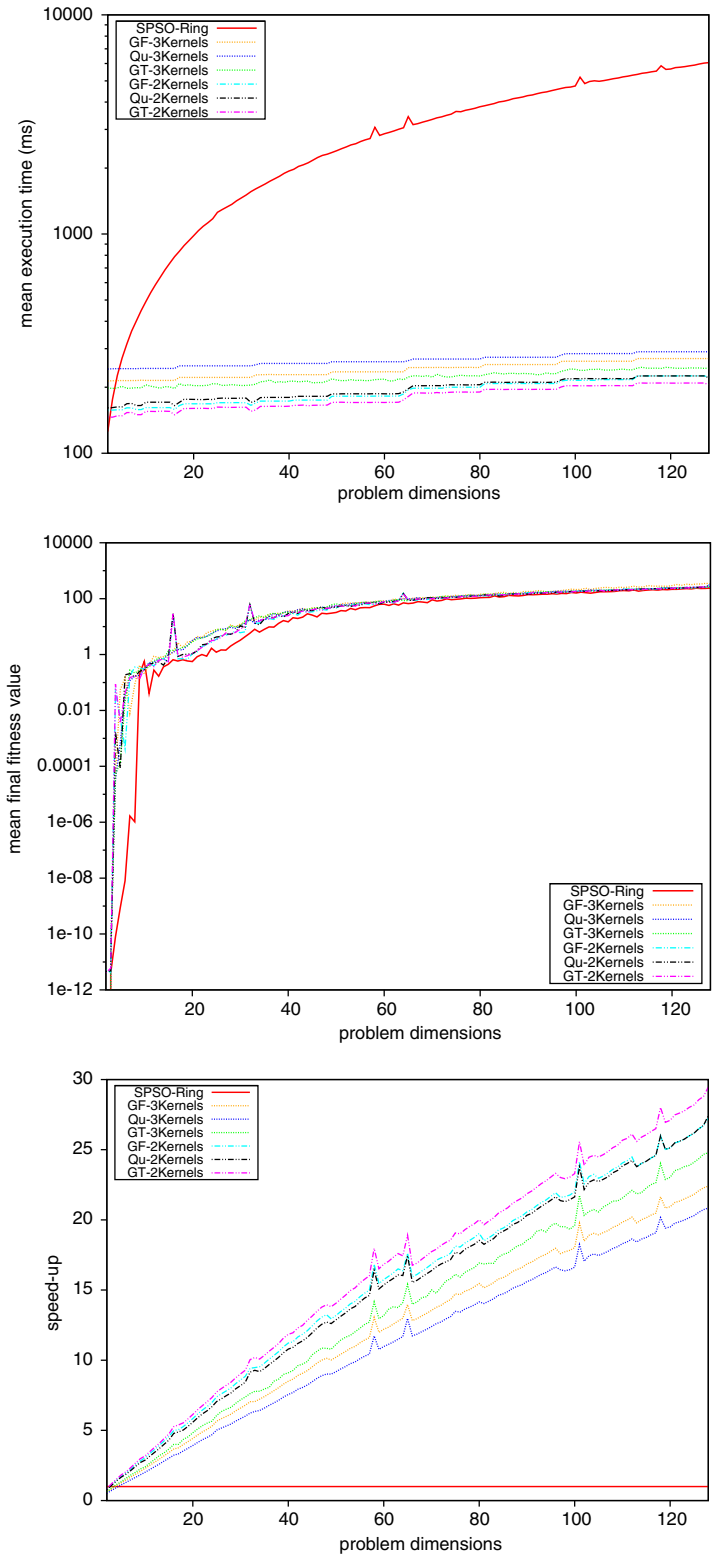


Fig. 6. Average execution times (above), average final fitness values (center) and speed-ups (below) vs. problem dimension for the Rosenbrock function. Experiments were performed running one swarm of 32 particles for 10,000 generations. Values plotted for both *RingPSO* and *SPSO* were averaged over the best 98 results out of 100 runs.

Table 2
Comparison between *RingPSO* and other work found in literature.

Work	Card	N_p	D	G	Time (s)	S_{up}
Zhou	GeForce 8600GT	400	50	2000	14.47	1x
RingPSO	GeForce 8800GT	416			0.140	103x
RingPSO	Quadro FX5800	416			0.114	127x
RingPSO	GeForce GTX260	416			0.105	138x
Veronese	GeForce GTX280	1000	100	100000	178.96	1x
RingPSO	GeForce 8800GT	1024			21.72	8x
RingPSO	Quadro FX5800	1024			17.60	10x
RingPSO	GeForce GTX260	1024			15.68	11x

We want to conclude this section with some comparisons with results published in the two most recent papers, to the best of our knowledge, which describe parallel PSO implementations developed with CUDA™.

Table 2 summarizes the results reported in [3,36] and compares them with those which *RingPSO* obtained with a similar experimental set-up. Here, N_p stands for number of particles, D for number of dimensions, G for number of generations, and S_{up} for speed-up. Since our GPUs are different from those used in the other papers, these results cannot be analyzed in depth, even if our implementations appear to be clearly more efficient. In fact, taking into consideration the different hardware configurations which were used (see Table 1), and the results obtained with the different configuration we tested, we may suppose that lower-performance hardware may have induced a slow-down of performances reported in [36] by a factor of 4–5 with respect to our best configuration, while the configuration used in [3] can be considered roughly equivalent to ours.

6. Final remarks

This work has investigated how much particle swarm optimization can take advantage of a parallel implementation in CUDA™. On the one hand, the design of the two parallel versions of PSO we implemented and tested was strongly influenced by the way CUDA™ is structured, which obviously depends on the internal structure of compatible GPUs. On the other hand, the practical implications also implied two possible solutions. This is what mainly differentiates the use one can make of the two versions. *SyncPSO*, in which each swarm is confined within the same thread block, achieves the goal of minimizing global memory usage, while allowing to simulate with maximum efficiency a number of swarms (equal to the number of SMs available) and a number of particles per swarm (256/512) which are usually more than enough for any practical application. However, its usage of computation resources is very inefficient in cases when only one or few swarms need to be simulated, since only as many SMs as the number of swarms are active. In practice, it also becomes inefficient when the problem size increases above a certain threshold, as SM resources saturate, even if a large number of real-world problems can still be dealt with even using cheap GPU cards, such as the cheapest one used in our experiments.

On the contrary, the multi-kernel version can be distributed over all SMs available, even if it requires a number of slow global memory accesses to allow for a 'centralized management' and synchronization of the swarm(s). In fact, CUDA™ requires that data shared among different SMs be stored in global memory. However, as our experiments have shown, these drawbacks are more than compensated by the advantages of parallelization, with speed-ups of up to more than one order of magnitude. As could be expected, since this was one of the goals the multi-kernel version was designed for, the speed-up increases with problem size.

In any case, both versions obtained results which, in terms of processing time and to the best of our knowledge, are far better than the most recent results published on the same task, even if, as pointed out in a previous section, technical evolution in GPU computing is so fast that new papers reporting new and better results in many fields are being published almost on a daily basis.

Finally, we would like to make some considerations on possible applications of parallel PSO, based on our direct experience in computer vision and pattern recognition, a field for which real-time performances are often as important as difficult to achieve. Several approaches to object detection and tracking which use PSO or other swarm intelligence techniques, of which [19,20,22,25,35] are just a few examples, are being developed more and more frequently. These applications impose strong real-time processing requirements, different from 'traditional' black-box function optimization. Object detection and tracking applications, for example, besides proposing dynamic environments in which the 'optima' to be detected change in time, often require that the swarm get 'close enough' to the target in the shortest possible time.

In this regard, we have recently developed a CUDA™-based traffic-sign detection application [22] in which each of a number of swarms searches, in real time, the values of the offset with respect to the camera and of the rotation matrix which, once applied to a reference sign model, would produce the pattern which is most similar to a corresponding region in the image. In this way, it is possible to both detect the appearance of traffic signs within an image and determine their position and orientation in space. In experiments performed on a synthetic noisy video sequence with two different kinds of traffic signs, we let two swarms run the three-kernel PSO with global-best topology for up to 50 generations per frame. This took 4–6 ms, which means either that we could locate signs in videos running up to 150/160 frames per second, or that, at 25 frames per second, more than 30 ms would still be available to perform sign recognition, after a sign has been detected. The same

task, sequentially implemented on the same PC we used for the experiments reported in this paper took about 80 ms, limiting the maximum acceptable frame rate to about 12 fps. Given these promising results, this sort of application will be central to the development of our future work.

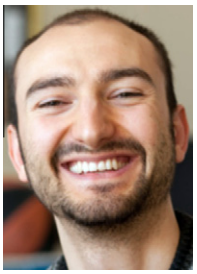
From a more theoretical point of view, but in the same application-oriented perspective, much of our future work will be dedicated to deepening the study of the effects of synchronization on parallel PSO convergence. The results we have reported, while confirming that relaxing synchronization constraints may sometimes be beneficial in this regard, have also confirmed that such effects are problem-dependent. Finding the features of a problem to which this variability is most related is definitely worth careful investigation.

Our CUDA™-based parallel PSO implementations are available at the address ftp://ftp.ce.unipr.it/pub/cagnoni/CUDA-PSO.

References

[1] E. Alba, M. Tomassini, Parallelism and evolutionary algorithms, IEEE Transactions on Evolutionary Computation 6 (5) (2002) 443–462.
[2] D. Bratton, J. Kennedy, Defining a standard for particle swarm optimization, in: Proceedings of IEEE Swarm Intelligence Symposium, pp. 120–127.
[3] L. de P. Veronese, R.A. Krohling, Swarm's flight: accelerating the particles using C-CUDA, in: Proceedings of IEEE Congress on Evolutionary Computation (CEC 2009), 2009, pp. 3264–3270.
[4] L. Dioşan, M. Oltean, Evolving the structure of the particle swarm optimization algorithms, in: European Conference on Evolutionary Computation in Combinatorial Optimization, EvoCOP'06, LNCS, vol. 3906, Springer-Verlag, 2006.
[5] L. Dioşan, M. Oltean, What else is evolution of PSO telling us?, Journal of Artificial Evolution and Applications 1 (2008) 1–12
[6] A. Eiben, J. Smith, Introduction to Evolutionary Computing, Springer-Verlag, 2003.
[7] D. Gies, Y. Rahmat Samii, Reconfigurable array design using parallel particle swarm optimization, in: International Symposium Antennas and Propagation Society, vol. 1, 2003, pp. 177–180.
[8] N. Hansen, A. Auger, S. Finck, R. Ros, Real-parameter Black-Box Optimization Benchmarking 2009: Experimental Setup, Tech. Rep. RR-6828, INRIA, 2009. <http://hal.inria.fr/inria-00362649/en/>.
[9] N. Hansen, S. Finck, R. Ros, A. Auger, Real-parameter Black-Box Optimization Benchmarking 2009: Noiseless Functions Definitions, Tech. Rep. RR-6829, INRIA, 2009. <http://hal.inria.fr/inria-00362633/en/>.
[10] N. Hansen, R. Ros, N. Mauny, M. Schoenauer, A. Auger, PSO facing non-separable and ill-conditioned problems, Research Report RR-6447, INRIA, 2008. <http://hal.inria.fr/inria-00250078/en/>.
[11] Š. Ivekovič, E. Trucco, Y. Petillot, Human body pose estimation with particle swarm optimisation, Evolutionary Computation 16 (4) (2008) 509–528.
[12] Jui-Fang Chang, Shu-Chuan Chu, John F. Roddick, Jeng-Shyang Pan, A parallel particle swarm optimization algorithm with communication strategies, Journal of Information Science and Engineering 21 (4) (2005) 809–818.
[13] G.E. Karniadakis, R.M. Kirby, Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and their Implementation, Cambridge University Press, 2003.
[14] J. Kennedy, M. Clerc, 2006. <http://www.particleswarm.info/Standard_PSO_2006.c>.
[15] J. Kennedy, R. Eberhart, Particle swarm optimization, in: Proceedings of IEEE International Conference on Neural Networks (ICNN 1995), vol. IV, 1995, pp. 1942–1948.
[16] J. Kennedy, R. Mendes, Population structure and particle swarm performance, in: Proceedings of the Congress on Evolutionary Computation (CEC 2002), 2002, pp. 1671–1676.
[17] B.-I. Koh, A.D. George, R.T. Haftka, B.J. Fregly, Parallel asynchronous particle swarm optimization, International Journal for Numerical Methods in Engineering 67 (2006) 578–595.
[18] J. Li, X. Wang, R. He, Z. Chi, An efficient fine-grained parallel genetic algorithm based on GPU-accelerated, in: IFIP International Conference on Network and Parallel Computing Workshops, 2007, pp. 855–862.
[19] S. Mehmood, S. Cagnoni, M. Mordonini, G. Matrella, Hardware-oriented adaptation of a particle swarm optimization algorithm for object detection, in: 11th EUROMICRO Conference on Digital System Design, DSD08, 2008, pp. 904–911.
[20] L. Mussi, S. Cagnoni, Particle swarm for pattern matching in image analysis, in: R. Serra, I. Poli, M. Villani (Eds.), Artificial Life and Evolutionary Computation, World Scientific, Singapore, 2010, pp. 89–98.
[21] L. Mussi, S. Cagnoni, F. Daolio, Empirical assessment of the effects of update synchronization in particle swarm optimization, in: Proceedings of AI*IA Workshop on Complexity, Evolution and Emergent Intelligence, 2009, pp. 1–10 (published on CD).
[22] L. Mussi, F. Daolio, S. Cagnoni, GPU-based road sign detection using particle swarm optimization, in: Proceedings of IEEE Conference on Intelligent System Design and Applications (ISDA09), 2009, pp. 152–157.
[23] nVIDIA, nVIDIA CUDA C Programming – Best Practices Guide v. 2.3., nVIDIA Corporation, 2009.
[24] nVIDIA, nVIDIA CUDA Programming Guide v. 2.3., nVIDIA Corporation, 2009.
[25] Y. Owechko, S. Medasani, A swarm-based volition/attention framework for object recognition, in: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2005), 2005, p. 91.
[26] R. Poli, J. Kennedy, T. Blackwell, Particle swarm optimization: an overview, Swarm Intelligence 1 (1) (2007) 33–57.
[27] J.F. Schutte, J.A. Reinbolt, B.J. Fregly, R.T. Haftka, A.D. George, Parallel global optimization with the particle swarm algorithm, Journal of Numerical Methods in Engineering 61 (2003) 2296–2315.
[28] I. Scriven, D. Ireland, A. Lewis, S. Mostaghim, J. Branke, Asynchronous multiple objective particle swarm optimisation in unreliable distributed environments, in: Proceedings of IEEE Congress on Evolutionary Computation (CEC 2008), 2008, pp. 2481–2486.
[29] F. Van den Bergh, A. Engelbrecht, A study of particle swarm optimization particle trajectories, Information Sciences 176 (8) (2006) 937–971.
[30] G. Venter, J. Sobieszczanski Sobieski, A parallel particle swarm optimization algorithm accelerated by asynchronous evaluations, Journal of Aerospace Computing, Information, and Communication 3 (3) (2006) 123–137.
[31] M. Waintraub, R. Schirru, C. Pereira, Multiprocessor modeling of parallel Particle Swarm Optimization applied to nuclear engineering problems, Progress in Nuclear Energy 51 (2009) 680–688.
[32] W. Wang, Particle Swarm Optimization on GPU, in: Presentation at the First NTU Workshop on GPU Supercomputing, 2009. <http://cqse.ntu.edu.tw/cqse/gpu2009.html>.
[33] W. Wang, Y. Hong, T. Kou, Performance gains in parallel particle swarm optimization via NVIDIA GPU, in: Proceedings of Workshop on Computational Mathematics and Mechanics (CMM 2009), 2009.
[34] S.D. Xue, J.C. Zeng, Parallel asynchronous control strategy for target search with swarm robots, International Journal of Bio-inspired Computation 1 (3) (2009) 151–163.
[35] X. Zhang, W. Hu, S. Maybank, X. Li, M. Zhu, Sequential particle swarm optimization for visual tracking, in: IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2008), 2008, pp. 1–8.
[36] Y. Zhou, Y. Tan, GPU-based parallel particle swarm optimization, in: Proceedings of IEEE Congress on Evolutionary Computation (CEC 2009), 2009, pp. 1493–1500.

Luca Mussi graduated in Computer Engineering at the University of Parma (2005), where he currently holds a research fellowship at the Department of Information Engineering, and was awarded the Ph.D. degree in 'Mathematics and Information Technology for Processing and Representation of Information and Knowledge' at the University of Perugia. His activities focus on the detection and tracking of moving objects for video surveillance and mobile robotics. He is interested in hybrid video sensors, which couple omnidirectional and traditional cameras. He has developed innovative swarm intelligence techniques for object detection and tracking.



Fabio Daolio graduated in Computer Engineering at the University of Parma (2009), where he also participated, as a research fellow, in projects concerning particle swarm optimization. He is currently a Ph.D. student in the Institute of Information Systems, Faculty of Business and Economics, at the University of Lausanne, Switzerland, where he studies the fitness landscapes of combinatorial problems and their associated local optima networks.

Stefano Cagnoni graduated in Electronic Engineering at the University of Florence (1988) where he worked until 1997 in the Bioengineering Lab of the Department of Electronic Engineering. He received the Ph.D. degree in Bioengineering in 1993. In 1994 he was a visiting scientist at the Whitaker College Biomedical Imaging and Computation Laboratory at the Massachusetts Institute of Technology. Since 1997, he has been with the Department of Computer Engineering of the University of Parma, where he is currently Associate Professor. His main research interests are in the fields of Computer vision, Evolutionary Computation and Swarm Intelligence.