2

Running Particle Swarm Optimization on Graphic Processing Units

Carmelo Bastos-Filho, Marcos Oliveira Junior and Débora Nascimento
University of Pernambuco
Brazil

1. Introduction

Particle swarm optimization (PSO) is a computational intelligence technique widely used to solve optimization problems. It was inspired on the social behavior of flocks of birds. In the PSO algorithm, the search space is bounded and each possible position within the search space represents a solution for the optimization problem.
During the algorithm execution, the particles adjust their velocities and positions based on the best position reached by itself and on the best position reached by its neighborhood during the search process. Some issues are quite important for the convergence velocity and the quality of the solutions, among them we can cite: the communication scheme to exchange information among the particles (topology) (Bratton & Kennedy (2007); Kennedy & Mendes (2002)), the equation used to update the velocity of the particles (Eberhart & Shi (2000)), the mechanisms to avoid explosion states (Clerc & Kennedy (2002)) and the quality of the Random Number Generator (RNGs) (Bastos-Filho et al. (2009)).
PSO is inherently parallel since the fitness can be evaluated for each particle individually. Hence, PSO is naturally suitable for parallel implementations.
In the recent years, the use of Graphic Processing Units (GPUs) have been proposed for some general purpose computing applications. GPUs have shown great advantages on applications requiring intensive parallel computing. Despite GPU based architectures require an additional CPU time to assign the tasks for the GPUs, the speed up obtained by GPU based architectures in relation to simple CPU architectures is higher for application where the processing is much more focused on floating point and matrix operations.
The major benefit to implement the PSO in GPU based architectures is the possibility to reduce the execution time. It is quite possible since the fitness evaluation and the update processes of the particles can be parallelized through different threads. Nevertheless, some issues regarding GPU-based PSOs should be analyzed.
In this chapter, we analyze and discuss some advantages and drawbacks of PSO algorithms implemented on GPU-based architectures. We describe the steps needed to implement different PSO variations in these architectures. These variations include different topologies such as Global, Local, Focal, Four Cluster and von Neumann. Two different approaches used to update the particles are analyzed as well.
We also consider some other relevant aspects such as: (1) how to determine the number of particle for a specific GPU architecture; (2) in which memory the variables should be allocated; (3) which RNG should be used to accelerate the execution; and (4) when and where

is necessary to state synchronization barriers. The analysis of these aspects is crucial to provide high performance for GPU-based PSOs. In order to show this, we performed simulation using a parallel processing platform developed by NVIDIA, called CUDA (NVIDIA, 2010).

2. Particle Swarm optimization

Particle Swarm Optimization (PSO) is a stochastic, bio-inspired, population-based global optimization technique. James Kennedy and Russel C. Eberhart first described the algorithm in 1995 (Kennedy & Eberhart (1995)). It is based on the social behaviour of biological organisms, specifically the ability of groups of animals to work as a whole in searching desirable positions in a given area.

Each particle in the swarm represents a point in the fitness function domain. Each particle i has four attributes: its current position in the D-dimensional space $\vec{x}_i = (x_{i1}, x_{i2}, ..., x_{id})$, its best position found so far during the search $\vec{p}_i = (p_{i1}, p_{i2}, ..., p_{id})$, the best position found by its neighborhood so far $\vec{n}_i = (n_{i1}, n_{i2}, ..., n_{id})$ and its current velocity $\vec{v}_i = (v_{i1}, v_{i2}, ..., v_{id})$ (Bratton & Kennedy (2007)). The position and the velocity of every particle are updated iteratively according to its current best position $\vec{p}_i$ and the best position of its neighborhood $\vec{n}_i$ by applying the following update equations for each particle in each dimension d:

$$v_{id} = v_{id} + c_1 \cdot r_1 \cdot (p_{id} - x_{id}) + c_2 \cdot r_2 \cdot (n_{id} - x_{id}), \tag{1}$$

$$x_{id} = x_{id} + v_{id}, \tag{2}$$

where c_1 and c_2 in the Equation (1) are non negative constants. They are the cognitive and the social acceleration constants, respectively, and they weight the contribution of the cognitive and social components. The values r_1 and r_2 are two different random variables generated at each iteration for each dimension from a uniform distribution in the interval [0,1].

Particles velocities can be clamped to a maximum value in order to prevent an explosion state. Their positions are also bounded by search space limits in order to avoid inutile exploration of space (Bratton & Kennedy (2007)).

The original PSO updates the velocities fully considering the previous velocity of the particles. The first and most famous variation of this updating process uses an inertia factor (ω) (Eberhart & Shi (2000)). It was designed to adjust the influence of the previous velocity of the particles. It helps to switch the search mode of the swarm from exploration to exploitation along the search process. The resulting velocity update equation is stated as follows:

$$v_{id} = \omega \cdot v_{id} + c_1 \cdot r_1 \cdot (p_{id} - x_{id}) + c_2 \cdot r_2 \cdot (n_{id} - x_{id}). \tag{3}$$

Similar to the parameter ω, a parameter χ, known as the constriction factor, was proposed by Clerc (Clerc & Kennedy (2002)). The factor χ is defined according to the following equation:

$$\chi = \frac{2}{|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}|}, \ \varphi = c_1 + c_2. \tag{4}$$

This constriction factor is applied to the entire velocity update equation as shown in the following equation:

$$v_{id} = \chi \cdot [v_{id} + c_1 \cdot r_1 \cdot (p_{id} - x_{id}) + c_2 \cdot r_2 \cdot (n_{id} - x_{id})]. \tag{5}$$

The effects are similar to those obtained by inertia weight approach. The χ parameter controls the exploration-exploitation abilities of the swarm.

2.1 Population topologies

The uniqueness of the PSO algorithm lies in the dynamic interaction of the particles (Kennedy & Mendes (2002)). Even when it is using different types of update equations, the performance depends on the information exchange mechanism inside the swarm.

The information flow scheme through the particles is determined by the communication method used by the swarm (Ferreira de Carvalho & Bastos-Filho (2009)). The topology of the swarm defines the neighborhood of each particle, that is the subset of particles which it is able to communicate with (Bratton & Kennedy (2007)).

Previous investigation performed by Watts (Watts (1999); Watts & Strogatz (1998)) showed that some aspects can affect the flow of information through social networks. These aspects included the degree of connectivity among the nodes; the average number of neighbors in common per node; and the average shortest distance between nodes.

Kennedy and Mendes analyzed these factors on the particle swarm optimization algorithm (Kennedy & Mendes (2002)) and showed that the presence of intermediaries slows down the information. On the other hand, it moves faster between connected pairs of individuals. Thus, when a distance between nodes are too short, the population tends to rapidly toward the best solution found in earlier iterations. On simple unimodal problems, it usually result in a faster convergence on the global optimum.

However, the fast convergence might lead to a premature suboptimal point on multi-modal problems (Bratton & Kennedy (2007)). In this case, a structure with intermediaries could help to reach better results.

The first PSO model used a dynamic topology with a Euclidean distance based neighborhood, where the distance between particles determines which particles were close enough to communicate with (Heppner & Grenander (1990)). It was abandoned due to the high computational cost, albeit the similarity to the natural behavior of bird flocks (Bratton & Kennedy (2007)).

The global topology is the structure proposed in the first PSO approach and is still used by researchers. It uses a global neighborhood mechanism known as *gbest*. Particles can share information globally through a fully-connected structure, as shown in Figure 1. This arrangement leads to a fast convergence, since the information spreads quickly.

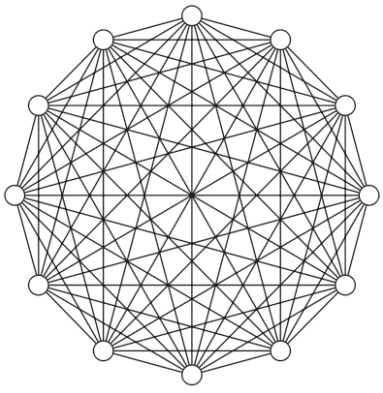

Fig. 1. The Global Topology.

The usual alternative to the global topology is the ring topology, depicted in Figure 2. In this structure, each particle has one neighbor on each side. The information spreads slowly

along the graph, thus different regions of the search space is explored, but it slows down the convergence.

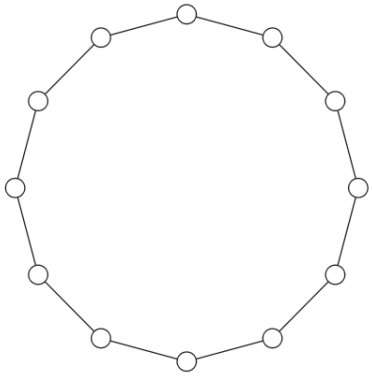

Fig. 2. The Ring Topology.

The dichotomic behaviour of the global and ring topologies suggests to consider structures to balance the previous approaches. Some other topologies were recently proposed.
The four clusters topology consists in clusters of particles connected among themselves by several gateways (Mendes et al. (2003)), as shown in Figure 3. Each gateway particle act as an informant that disseminates the information acquired by its own cluster.

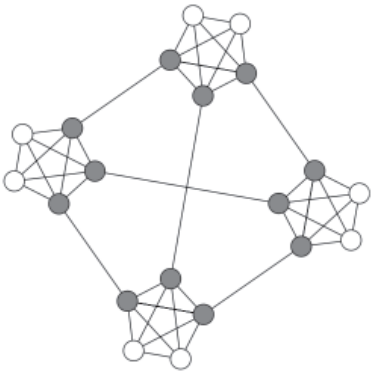

Fig. 3. The Four Clusters Topology.

The topology depicted in Figure 4 is known as focal topology. One single particle is the focus of the swarm, it updates its position based on the performance of the other particles (Ferreira de Carvalho & Bastos-Filho (2009)). If the focus improves its position, this information will be transmitted to the entire neighborhood.
In the von Neumann topology, the particles are connected as a grid. Each particle has neighbors above, below, and on each side (Kennedy & Mendes (2002)), as shown in Figure 5. This structure is commonly used to represent neighborhoods in the Evolutionary Computation and Cellular Automata communities (Mendes et al. (2004)). Kennedy and Mendes pointed this topology as the most consistent in their experiments in (Kennedy & Mendes (2002)).

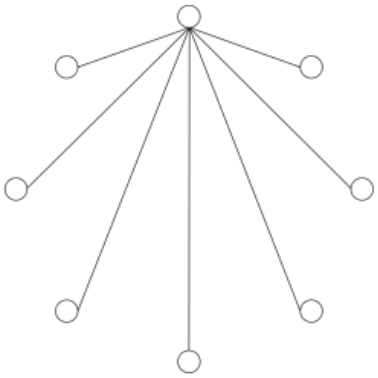

Fig. 4. The Focal Topology.

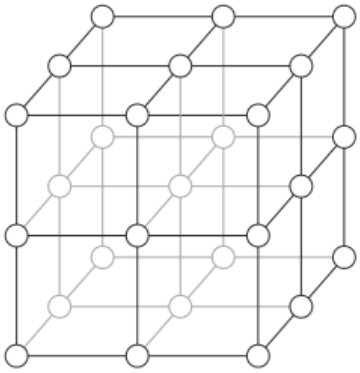

Fig. 5. The von Neumann Topology.

3. GPU computing

GPU parallel computing follows an architecture called SIMT (Single Instruction – Multiple Threads) (NVIDIA (2010)). In the SIMT paradigm, the same instruction set is executed on different data processors at the same time. In our case, the data processors are on the GPUs. The GPU SIMT architecture presents a lower overhead for parallel computing, which is suitable for intensive and repetitive computations.

The GPUs are specially well-suited to tackle problems that can be expressed as data-parallel computations, where the same set of commands can be executed in parallel on many data processing elements. Data-parallel processing maps these data elements to parallel processing threads.

Traditionally, GPUs were designed for image and graphic processing. However, The GPUs have became popular in recent years through the CUDA (Compute Unified Device Architecture), which is a technology that enables programmers and developers to write software to solve complex computational problems by automatically distributing these threads to many-core graphic processors.

The CUDA parallel programming model allows the programmer to divide the main problem in many sub-problems that can be executed independently in parallel. Each sub-problem can

be decomposed in many modules where each module can be executed cooperatively and in parallel. In CUDA terminology, each sub-problem corresponds to a block of threads, where each thread is a module. The function that is performed to solve the sub-problem is called kernel. When a kernel function is invoked by the CPU, it runs on each thread within the corresponding block.
Each thread that executes a kernel function is identified by its thread identifier that is accessible within the kernel with two built-in variables *threadIdx* and *blockIdx*.
Blocks of threads are organized into a one-dimensional or two-dimensional grid of thread blocks. The number of thread blocks in a grid is defined by the size of the data being processed or the number of processors in the system. The maximum number of threads inside a block is defined by the number of processor inside the GPU and its architecture. On the current GPUs, a thread block may contain up to 1024 threads. However, the simulations in this chapter were made with GPUs that supports up to 512 threads.
Threads within a block can cooperate among themselves by sharing data through some shared memory and synchronizing their execution to coordinate memory accesses. Each thread has a private local memory. Each block of threads has a shared memory visible to all threads of the block and with the same lifetime of the block. All threads can access the same global memory. There is a memory hierarchy in the architecture, the memory with fastest access is the local memory and the one with the slowest is the global memory. On the other hand, the largest memory is the global memory and the smallest is the local memory.

3.1 Previous GPU-based particle Swarm optimization approaches

Few parallel PSO implementations on GPU have been developed. They have all shown great speed-up performances when compared to CPU versions. Nevertheless, there is a absence of analysis of the performance impact on the parallel model when using different topologies.
Zhou & Tan (2009) have presented a model of the Standard Particle Swarm Optimization (SPSO) (Bratton & Kennedy (2007)) on CUDA. In their analysis, there is no other topology, but the local ring structure. Furthermore, the implementation is not CPU-free since it needs to generate random numbers on the *host*. Zhu & Curry (2009) presented a SPSO model that combines CPU and GPU codes on the particles best solution updates. Likewise, they carried out the analysis only on the ring topology. On the other hand, Mussi et al. (2009) did not only focus their studies on the local ring topology, they implemented the global topology and also used their PSO model to detect signs on the road.
Moreover, none of these studies presented any analysis of the impact of the location of the synchronization barriers on PSO performance. For instance, the different location of the synchronization barriers is what allows to generate a synchronous PSO, where the best position is obtained after all particles have been totally updated, or a asynchronous PSO, which updates the best position based on the current status of the particles during the optimization process.

4. Our GPU-based Particle Swarm optimization proposals

4.1 First Considerations

There are some first considerations to be analyzed when modeling the particle swarm optimization technique on the CUDA platform. The algorithm correctness must be guaranteed, once race conditions on a parallel implementation may lead to wrong results. Furthermore, since we want to run the algorithm as fast as possible, it is worth to discuss the main issues that may slow it down.

4.1.1 Memory bottleneck

There is a huge bottleneck on data transferring between the host and the device. Any transfer of this kind can harm the performance. Thus, this operation should be avoided when it is possible. One way to get rid of it is to move more code from the host to the device.

On the same way, as shown in Section 3, there is a memory hierarchy present on CUDA platform. The access to the shared memory is faster than the access to the global memory, despite both memory are placed in the GPU. Then, if there is data that must be accessed by all threads in a block, the shared memory is the best choice. However, there are two issues on using it: it is only shared by threads inside one block, if there are more blocks, it will not work right; and as it follows a memory hierarchy, the memory space is not very large, then there will be problems that are not suitable for the shared memory.

4.1.2 Synchronization barriers

Another issue to be pointed out relies on the synchronization barriers. A barrier placed on the code allows a thread to wait until all the other threads have reached the same barrier. They guarantee the correctness of the algorithm running on the GPU, but can mitigate the performance.

On the original PSO algorithm, each particle updates its neighborhood $\vec{n}_i$ after all particles have been totally updated. This first approach is known as synchronous PSO. On the other hand, the asynchronous PSO updates the $\vec{n}_i$ based on the current status of the other particles. It means that a particle can adjust its $\vec{n}_i$ before some other particle j updates its $\vec{n}_j$.

Hence, the synchronous PSO must be implemented carefully with barriers to prevent any race condition that could generate mistaken results. These barriers guarantee the correctness but it comes with a caveat. Since the particles need to wait for all others, all these barriers harm the performance. As a result, the execution of the asynchronous approach is faster than the synchronous one due its absence of synchronization barriers. However, the results will be probably worse, since the information acquired is not necessarily updated.

4.1.3 Random number generation

Bastos-Filho et al. (2009) showed that the random number generator (RNG) used in the PSO technique needs a minimum quality for the algorithm to perform well. However, the classic RNGs can not be executed parallel, since they need a current state and a seed to run.

Most of GPU-based PSO proposed get rid of it pre-generating all random numbers needed to run the algorithm on the CPU. Some other PSO implementations use a GPU version of the Mersenne Twister generator (Matsumoto & Nishimura (1998); Podlozhnyuk (2007)). Both approaches are not CPU-free and the numbers are not generated on demand.

GPU-based random numbers generators are discussed by Nguyen (2007) and Thomas et al. (2009). Bastos-Filho et al. (2010) presented a CPU-free approach for generating random numbers on demand based on the Xorshift generator (Marsaglia (2003)). They also analyzed the quality of the random numbers generated with the PSO algorithm and showed that the quality of the RNG is similar to the frequently RNGs used by researchers.

4.1.4 Hardware limitation

The GPUs present in the current video cards have a huge parallel computation capacity. However, they have also some limitation that may reduce their performance.

Although the GPUs are famous by their parallel high precision operations, there are GPUs with only single precision capacity. Since many computational problems need double

precision computation, this limitation may lead to wrong results and then turn these GPUs inappropriate to solve these kind of problems.
Another issue on the GPU is the maximum number of threads running in parallel in a block. It is defined by the number of processors and the architecture. Therefore, each GPU has its own limitation. This way, application that needs to overpass this limitation have to be executed sequentially with more blocks what might result in wrong computations.
The global memory space is not a big deal for the majority of the applications. However, the other levels of memory – such as the shared memory and the local memory – are not as huge as the global memory. This limitation may harm the performance, since many applications need to store data on a low speed memory – the global memory.
The NVIDIA CUDA platform classify the NVIDIA GPUs with what they call Compute Capability NVIDIA (2010). The cards with double-precision floating-point numbers has Compute Capability 1.3 or 2.x. The cards with 2.x Capability can run 1024 threads in a block and has 48 KB of shared memory space, the other ones only can execute 512 threads and have 16 KB of shared memory space.

4.2 Data structures, kernel functions and GPU-operations

The swarm is modeled with four unidimensional NxD-elements arrays, where N is the number of particles and D is the number of dimensions. The four arrays represent the velocity $\vec{v}_i$, the position $\vec{x}_i$, the best position reached $\vec{p}_i$ and the best position in the neighborhood $\vec{n}_i$ – respectively vx, xx, px and nb – for all the particles of the swarm. For example, the first element of the xx array corresponds to the first dimension of the first particle position vector; the Nth element is the first dimension of the second particle, and so far.
The GPU used to run the algorithm only executes 512 thread simultaneously. Due to this, more than one block is needed to run the algorithm. Once the threads need to communicate through blocks, the vectors used by the algorithm must be stored in the global memory. It will harm the performance but it can be overcomed by using more advanced GPUs cards.
The modulus operation and the integer division are widely used through the pseudocodes. The first one is expressed by the % character and it returns the division rest of the operators. The integer division is actually a simple division operation attributed to a integer variable, it means that only the integer value is stored.
In order to know which particle belongs the Kth element of an array, one can use the modulus operation and the integer division operation. By instance, the particle has index $p = K/D$ and it represents the dimension $d = K\%D$.
The synchronization barrier shown on Section 4.1.2 is expressed on the pseudocodes by the command *synchronization-barrier()*. In the CUDA platform, it is expressed as *__syncthreads()*.
The kernel function calls are written in the code with the < < a, b > > parameters. It means that the function will be executed by a threads in b blocks. It is very similar to the CUDA platform, but, in this case, it is used built-in variable types.
The *particle* variable presented on the codes is actually what defines a thread on the GPU. In the CUDA platform, it is obtained using built-in variables shown on Section 3.
The *first-thread-of-the-particle* is also seen on the codes, it is a boolean that indicates if the current thread is the thread with the duty to manipulate the first dimension of the vectors. It is used to avoid many threads executing the same operation on the same data.

4.3 The synchronous PSO

The synchronous version of the PSO needs a parallel model with barriers to assure that the particles will use the best information available from their neighborhood. For this, we split the code in kernel functions. After each call of a kernel function, there is an implicit synchronization barrier, which means that every thread has terminated its execution before the next kernel function be executed. The main code of the synchronous PSO with the kernel function calls is shown on Pseudocode 1.

Pseudocode 1 The Main Part of the Synchronous PSO on GPU.

```
1  for 1 to number-of-runs
2    initialize-particles << 512, 2 >>;
3    fitness-evaluation << 30, 1 >>;
4    initialize-pbest << 30, 1 >>;
5    do
6        fitness << 30, 1 >>;
7        update-pbestx << 512, 2 >>;
8        update-pbest << 30, 1 >>;
9        find-nbest << 30, 1 >>;
10       update-particles << 512, 2 >>;
11   while (iteration < number-of-iterations)
```

The implicit barrier avoids the particles to be updated before the best information has been found. However, searching for the best particle in the neighborhood may be computationally expensive, once it is a sequential search. The worst case is in the global topology where the order is $O(n)$.

Since the particles are actually arrays, one way to get rid of it is to use a method called Reduction Method. It reduces the searching order to $O(\log n)$. First, it compares in parallel the odd index elements of an array with the even index ones. Then, each thread copies the best one to the odd index element. Subsequently, the threads compare the elements with index being multiple of 2 and copy in parallel the best one to the element with the smallest index. And so forth with the elements multiples of 4, 8, $2^n < N/2$, where N is the number of elements in the array. In the end, the best value is in the first element of the array.

This method is described on Pseudocode 2 and it is used by the Global topology, the Four Clusters topology and the Focal topology.

Pseudocode 2 The Reduction Method with a 32-elements Array.

```
1  for s = 1 to 16 with increment s = s*2
2    if (thread-idx % 2*s == 0)
3        if (array[thread-idx] > array[thread-idx])
4            array[thread-idx] = array[thread-idx];
5    synchronization-barrier().
```

The *find-nbest* kernel has the duty to search the best neighbour of each particle. Each thread acts as a particle and search on its own neighborhood. Thus, the difference between the topologies lies in this kernel. They are shown in the following subsections.

4.3.1 Synchronous GPU-based PSO using the Global Topology

In the global topology, all particles are able to communicate with the other particles. Then, the best particle in a neighborhood is actually the best particle in the swarm. Thus, basically the global topology is the reduction method being used.
The kernel function is shown on the Pseudocode 3. Differently to the Pseudocode 2, there is an auxiliary variable *index* in the code. It is used to not override the *pbest* values.

Pseudocode 3 The *find-nbest* Kernel for the Global Topology.

```
1 for (int s = 1; s <=16; s = s*2)
2   if (particle % 2*s == 0)
3       if (pbest[index[particle]] > pbest[index[particle+s]])
4           index[particle] = index[particle+s];
5   synchronization-barrier();
6 best-neighbour[particle] = index[0];
```

4.3.2 Synchronous GPU-based PSO using the Ring topology

A particle in a swarm with the ring topology has only two neighbours. They are modeled here with the contiguous elements of an array. Thus, a thread knows its neighbours using the modulus operator, as shown on Pseudocode 4.

Pseudocode 4 The *find-nbest* Kernel for the Ring Topology.

```
1 best-neighbour[particle] = particle;
2 neighbour = (particle - 1) % number-of-agents;
3 if (pbest[best-neighbour[particle]] > pbest[neighbour])
4   best-neighbour[particle] = neighbour;
5 neighbour = (particle + 1) % number-of-agents;
6 if (pbest[best-neighbour[particle]] > pbest[neighbour])
7   best-neighbour[particle] = neighbour;
```

4.3.3 Synchronous GPU-based PSO using the von Neumann topology

The von Neumann topology is similar to the Ring topology. A particle has neighbors above, below, and on each side. This structure is a grid that can be defined by the number of columns and the number of rows.
The modulus operator and the integer division are used in the code. They are used with the variables *columns* and *rows* – respectively, the number of columns and the number of rows of the grid – to determine the neighbours. The code is described on Pseudocode 5.

4.3.4 Synchronous GPU-based PSO using the Four clusters topology

The Four Clusters topology is a composition of the Ring topology and the Global topology. First, each cluster find its best particle in the sub-swarm, then each gateway particle in the cluster communicates with others gateway to disseminate information.
The Reduction Method is used in the code but with a lower number of iterations. Thus, it finds the best particle in the current cluster. Then, using the modulus operator and the

Pseudocode 5 The *find-nbest* Kernel for the von Neumann Topology.

```
1  best-neighbour[particle] = particle;
2  x = particle % columns;
3  y = particle / columns;
4  neighbour = y * columns + (x - 1) % columns;
5  if (pbest[best-neighbour[particle]] > pbest[neighbour])
6     best-neighbour[particle] = neighbour;
7  neighbour = y * columns + (x + 1) % columns;
8  if (pbest[best-neighbour[particle]] > pbest[neighbour])
9     best-neighbour[particle] = neighbour;
10 neighbour = ((y - 1) % rows) * columns + x;
11 if (pbest[best-neighbour[particle]] > pbest[neighbour])
12    best-neighbour[particle] = neighbour;
13 neighbour = ((y + 1) % rows) * columns + x;
14 if (pbest[best-neighbour[particle]] > pbest[neighbour])
15    best-neighbour[particle] = neighbour;
```

integer division, the gateways communicate with the other clusters. The code is presented on Pseudocode 6.

Pseudocode 6 The *find-nbest* Kernel for the Four Clusters Topology.

```
1  for (int s = 1; s <=4; s = s*2)
2     if (particle % 2*s == 0)
3        if (pbest[index[particle]] > pbest[index[particle+s]])
4           index[particle] = index[particle+s];
5     synchronization-barrier();
6  gbest-cluster = particle / clusters;
7  gbest-cluster = gbest-cluster * clusters;
8  best-neighbour[particle] = gbest-cluster;
9  y = particle / agents-per-cluster;
10 x = particle % agents-per-cluster;
11 neighbour = particle + agents-per-cluster * (y - x + 1);
12 if (neighbour > number-of-agents)
13    neighbour = particle;
14 if (pbest[best-neighbour[particle]] > pbest[neighbour])
15    best-neighbour[particle] = neighbour;
```

4.3.5 Synchronous GPU-based PSO using the Focal topology

The Focal topology is very similar to the Global topology. The focus find the best particle in the swarm but this information flows slowly through the swarm.
The kernel function is described on Pseudocode 7. The focus is the particle with the index "0" in the array. First all particles are compared with the focus, then the Reduction Method is used to find the best particle in the swarm.

Pseudocode 7 The *find-nbest* Kernel for the Focal Topology.

```
1 best-neighbour[particle] = particle
2 if (pbest[particle] > pbest[0])
3    best-neighbour[particle] = 0;
4 for (int s = 1; s <=16; s = s*2)
5    if (particle % 2*s == 0)
6        if (pbest[index[particle]] > pbest[index[particle+s]])
7            index[particle] = index[particle+s];
8    synchronization-barrier();
9 best-neighbour[0] = index[0];
```

4.4 The Asynchronous PSO

In the asynchronous PSO, all barriers used to assure no race condition on the synchronous PSO are removed. Thus, it allows the best particle variable value to be changed at any time if other particle finds a better position. Once it runs in parallel and it presents race conditions, there is no guarantee that the best information will be considered. The main code is presented on Pseudocode 8.

Pseudocode 8 The Main Part of the Asynchronous PSO on GPU.

```
1 for 1 to number-of-runs
2    initialize-particles << 512, 2 >>;
3    do
4        iteration << 512, 2 >>;
5    while (iteration < number-of-iterations)
```

The whole PSO algorithm lies in the kernel function *iteration*. The function is shown on Pseudocode 9. The difference between the topologies is presented in the line 4. In the following subsections, these differences are presented.

4.4.1 Asynchronous GPU-based PSO using the Global topology

In the global topology, the best particle in the neighborhood is actually the best one in the swarm. In fact, each particle claims to be the best after have compared with the current best one. In the asynchronous PSO, many particles claim to be the best at the same time and try to change the best value found so far by the swarm. Then, there is a race condition here what means that it is not guaranteed that the best value will actually have the best particle value. The asynchronous PSO with global topology is shown in the Pseudocode 10.

4.4.2 Asynchronous GPU-based PSO using the Ring topology

Each particle in a ring topology has two neighbours. The neighborhood of a particle is composed by the elements located after and before its index in the array. The code of this topology is presented in Pseudocode 11 and it is similar to the one in Pseudocode 4. The differences between them are only noticed at runtime, since the functions may be executed asynchronously by threads in different multi-processors in the asynchronous version.

Pseudocode 9 The *iteration* Kernel Function of the Asynchronous PSO on GPU.

```
1  if (the-first-thread-of-particle)
2     minval[particle] = fitness-evaluation();
3  synchronization-barrier();
4  ...
5  synchronization-barrier();
6  dimension = index % number-of-dimensions;
7  pbest-index = dimension + best-neighbour[particle]*number
8             -of-dimensions;
9  vx[index] = factor*(vx[index] + c*random*(pbestx[index]-xx[index])
10             + c*random*(pbestx[pbest-index] - xx[index]));
11 if (vx[index] > maxv)
12    vx[index] = maxv;
13 else if (vx[index] < -maxv)
14    vx[index] = -maxv;
15 xx[index] = xx[index] + vx[index];
16 if (xx[index] > maxx)
17    xx[index] = maxx;
18    vx[index] = -vx[index];
19 if (xx[index] < minx)
20    xx[index] = minx;
21    vx[index] = -vx[index];
```

Pseudocode 10 The Global Topology for the GPU-based Asynchronous PSO.

```
1  if (minval[particle] <= pbest[particle])
2     if (the-first-thread-of-particle)
3        pbest[particle] = minval[particle];
4     pbestx[thread] = xx[index];
5     if (the-first-thread-of-particle)
6        if (pbest[particle] < pbest[gbest])
7           gbest = particle;
```

Pseudocode 11 The Focal Topology for the GPU-based Asynchronous PSO.

```
1  if (minval[particle] <= pbest[particle])
2      if (the-first-thread-of-particle)
3          pbest[particle] = minval[particle];
4      pbestx[thread] = xx[index];
5  best-neighbour[particle] = particle;
6  if (the-first-thread-of-particle)
7      neighbour = (particle - 1)%number-of-agents;
8      if (pbest[best-neighbour[particle]] > pbest[neighbour])
9          best-neighbour[particle] = neighbour;
10     neighbour = (particle + 1)%number-of-agents;
11     if (pbest[best-neighbour[particle]] > pbest[neighbour])
12         best-neighbour[particle] = neighbour;
```

4.4.3 Asynchronous GPU-based PSO using the von Neumann Topology

The von Neumann topology has similar behaviour when compared to the Ring topology. A particle has neighbours above, below, and on each side. This structure is a grid that can be defined by the number of columns and the number of rows.

The modulus operator and the integer division are used for this purpose. They are used with the variables *columns* and *rows* – respectively, the number of columns and the number of rows of the grid – to determine the neighbours. The code is described in Pseudocode 12 and it is very similar to the one presented in Pseudocode 5. Once the functions may be executed asynchronously by threads in different multi-processors, the differences between the synchronous and asynchronous versions are only noticed at runtime.

Pseudocode 12 The von Neumann Topology for the GPU-based Asynchronous PSO.

```
1  if (minval[particle] <= pbest[particle])
2      if (the-first-thread-of-particle)
3          pbest[particle] = minval[particle];
4      pbestx[thread] = xx[index];
5  best-neighbour = particle;
6  x = particle % columns;
7  y = particle / columns;
8  neighbour = y * columns + (x-1)%columns;
9  if (pbest[best-neighbour[particle]] > pbest[neighbour])
10     best-neighbour[particle] = neighbour;
11 neighbour = y * columns + (x+1)%columns;
12 if (pbest[best-neighbour[particle]] > pbest[neighbour])
13     best-neighbour[particle] = neighbour;
14 neighbour = ((y - 1)%rows)*columns + x;
15 if (pbest[best-neighbour[particle]] > pbest[neighbour])
16     best-neighbour[particle] = neighbour;
17 neighbour = ((y + 1)%rows)*columns + x;
18 if (pbest[best-neighbour[particle]] > pbest[neighbour])
19     best-neighbour[particle] = neighbour;
```

4.4.4 Asynchronous GPU-based PSO using the Four clusters topology

In the Four Clusters topology, the particles first find the best particle in each cluster, likewise the Global topology. Then, the gateways present in each sub-swarm communicate with other clusters.

In the asynchronous PSO, the Global topology presented inside each cluster is affected in the same way as seen in Section 4.4.1. The race condition in the code means that it is not guaranteed that the best particle is the last one to set the best value found so far by the swarm. The code is described in Pseudocode 13. It uses the modulus operator and the integer division with the variable *agents-per-clusters* that is the number of particles inside each cluster.

Pseudocode 13 The Four Clusters Topology for the GPU-based Asynchronous PSO.

```
1  if (minval[particle] <= pbest[particle])
2     if (the-first-thread-of-particle)
3         pbest[particle] = minval[particle];
4     pbestx[thread] = xx[index];
5  if (the-first-thread-of-particle)
6     first-of-cluster = particle/clusters;
7     first-of-cluster = first-of-cluster * clusters;
8     best-neighbour[particle] = particle;
9     if (pbest[best-neighbour[first-of-cluster]] < pbest[particle])
10        best-neighbour[first-of-cluster] = particle ;
11    best-neighbour[particle] = best-neighbour[first-of-cluster]
12    y = particle / agents-per-cluster;
13    x = particle % agents-per-cluster;
14    neighbour-cluster = particle + agents-per-cluster*(x + y - 1);
15    if (neighbour-cluster > number-of-agents)
16        neighbour-cluster = particle;
17    if (pbest[particle] > pbest[neighbour-cluster])
18        best-neighbour[particle] = neighbour-cluster;
```

4.4.5 Asynchronous GPU-based PSO using the Focal topology

Since the Focal topology is very similar to the Global topology, once the focus first find the best particle in the swarm, the asynchronous PSO using the Focal topology is affected similarly as shown in Section 4.4.1. There is no guarantee that the best value found so far by the swarm will be set in the end by the best particle.

The code is described in Pseudocode 14 and the focus is the particle with index "0".

5. Experiments

In this section we present the performed experiments to compare the performance of synchronous and asynchronous PSO running on CUDA for each of the topologies analysed previously. Well known benchmark functions were chosen to perform the experiments.

5.1 Benchmark functions

Four benchmark functions were used to employ the simulations and are described in equations (6) to (9). All the functions are used for minimization problems. Two of these

Pseudocode 14 The Focal Topology for the GPU-based Asynchronous PSO.

```
1  if (minval[particle] <= pbest[particle])
2      if (the-first-thread-of-particle)
3          pbest[particle] = minval[particle];
4      pbestx[thread] = xx[index];
5  if (the-first-thread-of-particle)
6      best-neighbour[particle] = particle;
7      if ((pbest[particle] < pbest[0]) && particle != 0)
8          best-neighbour[0] = particle;
9      else
10         best-neighbour[particle] = 0;
```

functions (Rosenbrock and Schwefel) are uni-modal problems, and the others two (Rastrigin and Griewank) are multimodal functions that contains many local optima.
The first one is Rosenbrock function. It has a global minimum located in a banana-shaped valley. The region where the minimum point is located is very easy to reach, but the convergence to the global minimum point is hard to achieve. The function is defined as follows:

$$F_{Rosenbrock}(\vec{x}) = \sum_{i=1}^{n} x^2 \left[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2 \right]. \tag{6}$$

The second is the generalized Rastrigin, a multi-modal function that induces the search to a deep local minima arranged as sinusoidal bumps:

$$F_{Rastrigin}(\vec{x}) = 10n + \sum_{i=1}^{n} \left[x_i^2 - 10cos(2\pi x_i) \right]. \tag{7}$$

The third and the fourth ones are Griewank and Schwefel functions:

$$F_{Griewank}(\vec{x}) = 1 + \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} cos\left(\frac{x_i}{\sqrt{i}} \right), \tag{8}$$

$$F_{Schwefel}(\vec{x}) = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j \right). \tag{9}$$

5.2 Simulations setup

The PSO using the constriction factor, as shown on Section 2, was used to perform all the simulations in this chapter. To compose the constriction factor (χ) we used $\kappa = 1$, $c_1 = 2.05$ and $c_2 = 2.05$ ($\phi \geq 4$) values. All the simulations were performed using 32 particles for all five PSO topologies (Four Clusters, Focal, Global, Local and von Neumann). We run 50 trial to evaluate the average fitness.
Each simulation of the Four Clusters topology uses 4 clusters, each one with 8 particles. The focus of the Focal topology is the first particle of the swarm. All swarms were randomly initialized in an area far from the optimal solution in every dimension. This allows a fair convergence analysis between the topologies. All the random numbers needed by the PSO algorithm were generated as proposed by Bastos-Filho et al. (2010).

6. Results

The results of the experiments involving all the four benchmark functions are described in this section. In the Section 6.1 the results of the comparison between the synchronous and asynchronous algorithm for each one of the considered topologies are presented. Section 6.2 presents a performance analysis in terms of elapsed time to run the algorithm.

6.1 Fitness Analysis

The average value of the best fitness achieved for the Rosenbrock function for five different topologies using the synchronous and the asynchronous algorithm is shown in Figure 6.
Figures 6 (a) and 6 (b) show the convergence curve for Rosenbrock in the Four Clusters and the Focal topologies, respectively. When we compare the results for synchronous and asynchronous algorithm in each oh these topologies, we observe a worse performance for the last one. It is what we expected, since some data were not considered during the velocity update processes due to lack of synchronization barriers. On the other hand, for the topologies Global, Local and von Neumann, presented in Figures 6 (c), 6 (d) and 6 (e), respectively, the results are quite similar, although the loss of information by the asynchronous versions.
Figure 7 show the average value of the best fitness achieved in the Griewank function. The analysis for this function is quite different. In this case, despite it needs more iterations to reach the convergence, the lack of synchronicity helps to avoid the swarm to be trapped on local minima.
The comparison of the average and the (standard deviation of the fitness) for Rastrigin and Schwefel functions for each topology are presented in Table 1 and Table 2. Table 1 shows the results for the synchronous algorithm performance, while Table 2 shows the results for the asynchronous version.
A similar behaviour observed for Griewank can be observed for the Rastrigin function. It is quite expected since both are highly multimodal functions.

Function	Topology	Mean	SD
Rastrigin	Focal	52.31488	13.55017
	Four Clusters	73.07843	39.9018
	Global	53.09091	14.55321
	Local	67.16104	32.97102
	von Neumann	55.72031	32.49462
Schwefel	Focal	$2 \cdot 10^{-12}$	$1.41 \cdot 10^{-11}$
	Four Clusters	$2 \cdot 10^{-12}$	$1.41 \cdot 10^{-11}$
	Global	$4 \cdot 10^{-11}$	$2.82 \cdot 10^{-11}$
	Local	$2.4 \cdot 10^{-11}$	$1.69 \cdot 10^{-10}$
	von Neumann	$4 \cdot 10^{-12}$	$2.82 \cdot 10^{-11}$

Table 1. The Average Value and Standard Deviation of the Fitness After 50 Trials of 10,000 Evaluations for the Synchronous Version.

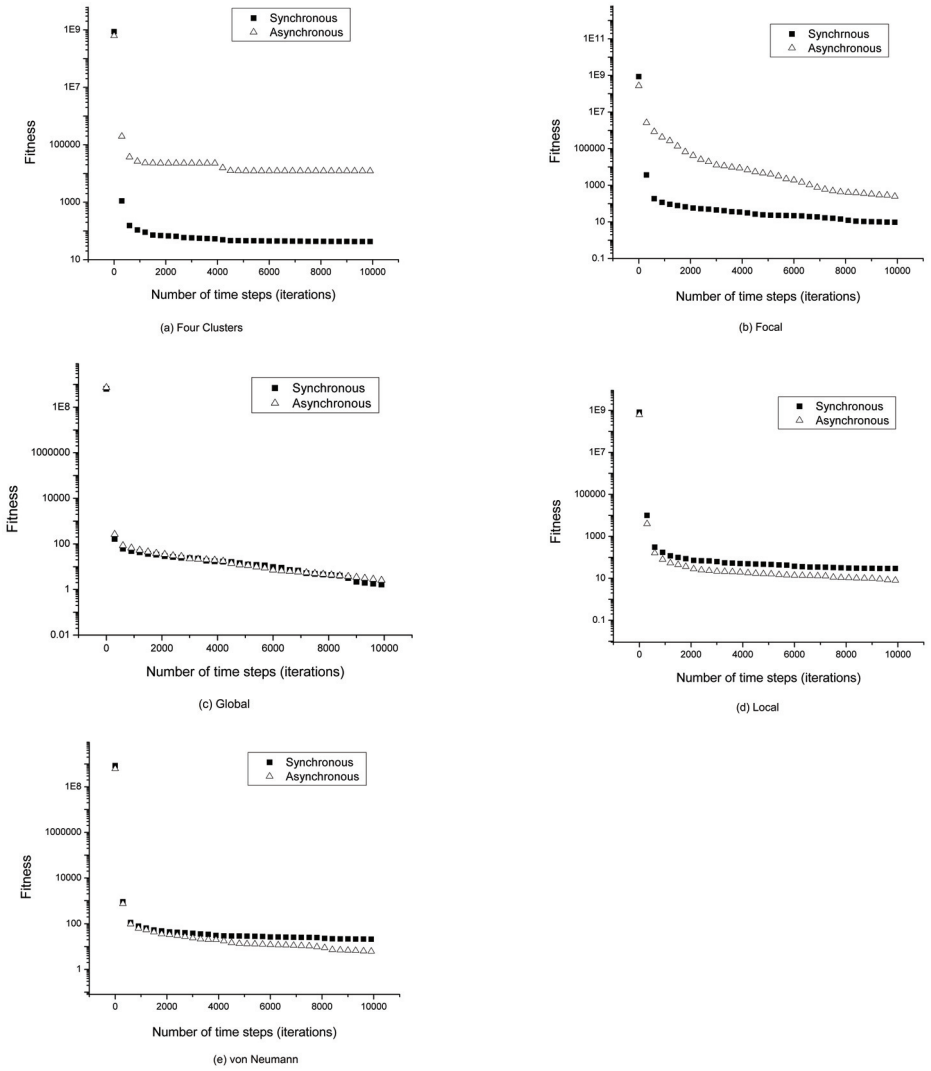

Fig. 6. The Best Fitness Comparison of the Five PSO Topologies for Rosenbrock Function.

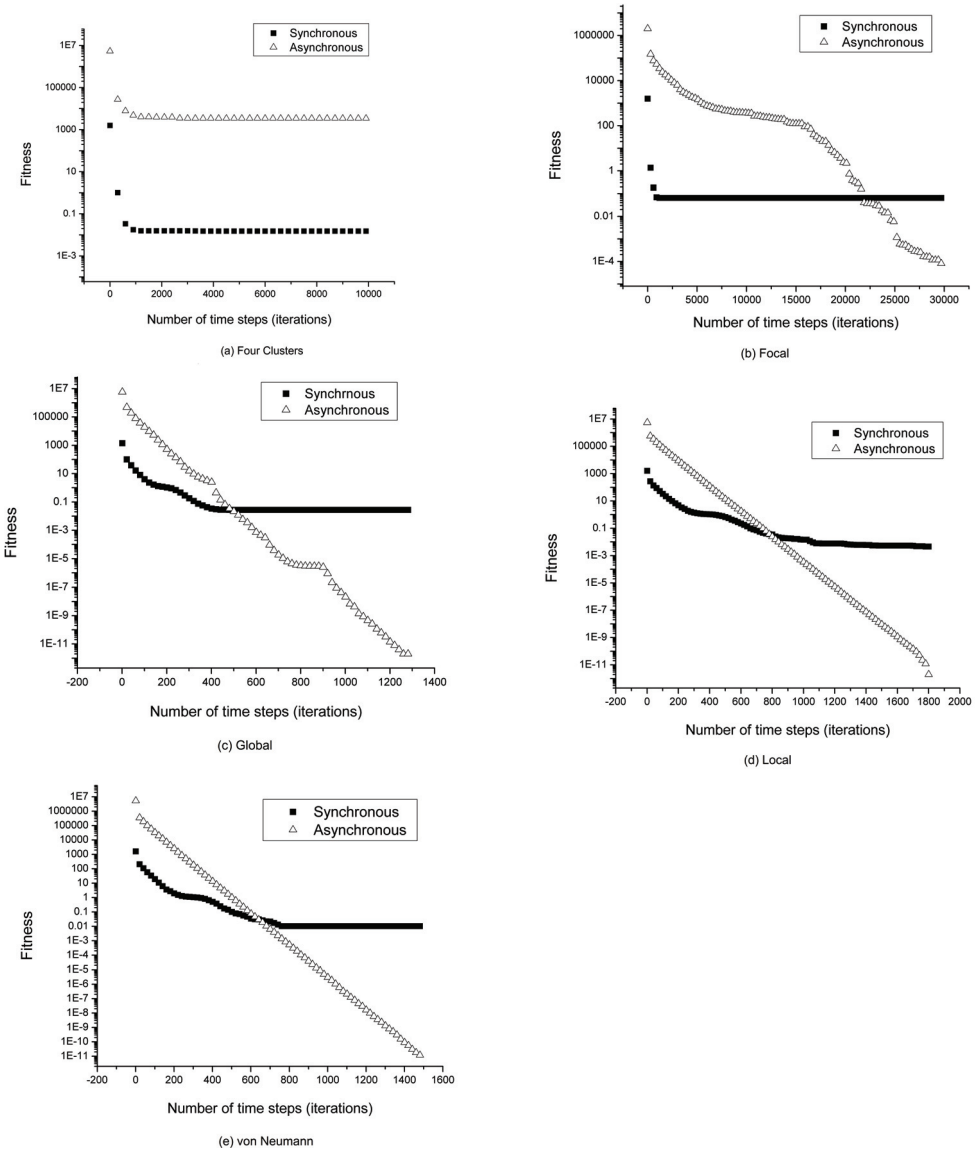

Fig. 7. The Best Fitness Comparison of the Five PSO Topologies of the Griewank Function.

Function	Topology	Mean	SD
Rastrigin	Focal	155.23031	26.16508
	Four Clusters	55.79153	15.49237
	Global	50.96177	13.36732
	Local	41.8803	10.2887
	von Neumann	33.26468	7.79463
Schwefel	Focal	$2 \cdot 10^{-12}$	$1.41 \cdot 10^{-11}$
	Four Clusters	19.6835	61.97515
	Global	$2 \cdot 10^{-11}$	$1.41 \cdot 10^{-11}$
	Local	$4 \cdot 10^{-12}$	$1.9794 \cdot 10^{-10}$
	von Neumann	$1.6 \cdot 10^{-11}$	$3.70328 \cdot 10^{-11}$

Table 2. The Average Value and Standard Deviation of the Fitness After 50 Trials of 10,000 Evaluations for the Asynchronous Version.

6.2 Performance analysis

The Figure 8(a) and Figure 8(b) show the elapsed time for the synchronous and asynchronous PSO for Rosenbrock and Griewank functions in Four Clusters, Focal, Global, Local and von Neumann topologies. One can note that the elapsed time for the asynchronous algorithm is lower than the synchronous algorithm for both Rosenbrock and Griewank functions. It is also expected due to lack of synchronization barriers. However, when we compare the performance between the five topologies, we can note that the synchronous algorithm results are quite similar for both Griewank and Rosenbrock functions. In other hand, when we compare the asynchronous algorithm results, a minor difference in terms of performance between the topologies can be observed.

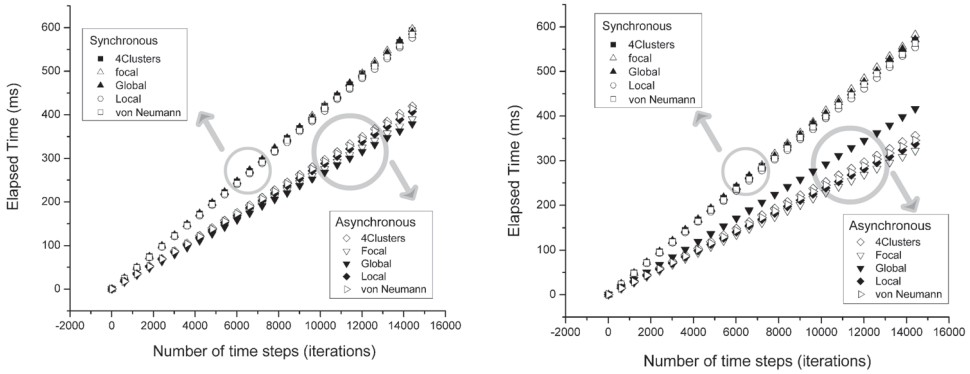

Fig. 8. The Elapsed Time for the GPU-based PSO Execution for Rosenbrock and Griewank Functions Asynchronous and Synchronous.

7. Conclusion

In this chapter, we have presented some concerns that should be carried out to implement a GPU-based PSO algorithm. We also have analysed the performance of the PSO with different topologies in terms of best value achieved and the time needed to execute the algorithm.
We have shown in Section 4.1 that there is a memory bottleneck when transferring data between the GPU and the CPU. The random number generator should run in the GPU in order to avoid data transfer.
We also have presented results for two approaches to update the particles. The synchronization barriers placed in the code influence the algorithm runtime and performance. As shown in Section 6, the behaviour of the algorithm running on the GPU with different functions depends on the approach used to update the particles. For instance, when optimizing the Rosenbrock function with the GPU-based PSO using topology Local, Global or von Neumann, the asynchronous PSO presented similar results to the synchronous one. As the asynchronous is faster, thus it is better use it in this case. On the other hand, by using the Focal topology or the Four Clusters topology, the asynchronous version shows a bad performance and the speed-up reached by using the asynchronous PSO is not worth, then the best choice is the synchronous version.
We also showed that, in some cases, one can remove the synchronization barriers, specially for multimodal search spaces.

8. Acknowledgments

The authors would like to thank FACEPE, CNPq, UPE and POLI (Escola Politécnica de Pernambuco).

9. References

Bastos-Filho, C., Andrade, J., Pita, M. & Ramos, A. (2009). Impact of the quality of random numbers generators on the performance of particle swarm optimization, *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on* pp. 4988 –4993.

Bastos-Filho, C., Oliveira Junior, M., Nascimento, D. & Ramos, A. (2010). Impact of the random number generator quality on particle swarm optimization algorithm running on graphic processor units, *Hybrid Intelligent Systems, 2010. HIS '10. Tenth International Conference on* pp. 85–90.

Bratton, D. & Kennedy, J. (2007). Defining a standard for particle swarm optimization, *Swarm Intelligence Symposium, 2007. SIS 2007. IEEE* pp. 120 –127.

Clerc, M. & Kennedy, J. (2002). The particle swarm - explosion, stability, and convergence in a multidimensional complex space, *Evolutionary Computation, IEEE Transactions on* 6(1): 58 –73.

Eberhart, R. & Shi, Y. (2000). Comparing inertia weights and constriction factors in particle swarm optimization, *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on* 1: 84 –88 vol.1.

Ferreira de Carvalho, D. & Bastos-Filho, C. J. A. (2009). Clan particle swarm optimization, *International Journal of Intelligent Computing and Cybernetics* 2(2): 197–227.
URL: *http://dx.doi.org/10.1108/17563780910959875*

Heppner, F. & Grenander, U. (1990). A stochastic nonlinear model for coordinated bird flocks, *in* E. Krasner (ed.), *The ubiquity of chaos*, AAAS Publications, pp. 233–238.

Kennedy, J. & Eberhart, R. (1995). Particle swarm optimization, Vol. 4, pp. 1942–1948 vol.4.
URL: *http://dx.doi.org/10.1109/ICNN.1995.488968*

Kennedy, J. & Mendes, R. (2002). Population structure and particle swarm performance, Vol. 2, pp. 1671 –1676.

Marsaglia, G. (2003). Xorshift rngs, *Journal of Statistical Software* 8(14): 1–6.
URL: *http://www.jstatsoft.org/v08/i14*

Matsumoto, M. & Nishimura, T. (1998). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator, *ACM Trans. Model. Comput. Simul.* 8: 3–30.
URL: *http://doi.acm.org/10.1145/272991.272995*

Mendes, R., Kennedy, J. & Neves, J. (2003). Watch thy neighbor or how the swarm can learn from its environment, *Swarm Intelligence Symposium*, 2003. SIS 2003. IEEE, pp. 88 – 94.

Mendes, R., Kennedy, J. & Neves, J. (2004). The fully informed particle swarm: simpler, maybe better, *Evolutionary Computation, IEEE Transactions on* 8(3): 204 – 210.

Mussi, L., Cagnoni, S. & Daolio, F. (2009). Gpu-based road sign detection using particle swarm optimization, *Intelligent Systems Design and Applications, 2009. ISDA '09. Ninth International Conference on* pp. 152 –157.

Nguyen, H. (2007). *GPU Gems 3*, Addison-Wesley Professional.

NVIDIA (2010). *NVIDIA CUDA Programming Guide 3.1*.

Podlozhnyuk, V. (2007). Parallel Mersenne Twister, *Technical report*, nvidia Corp.

Thomas, D. B., Howes, L. & Luk, W. (2009). A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation.
URL: *http://doi.acm.org/10.1145/1508128.1508139*

Watts, D. J. (1999). *Small worlds: The dynamics of networks between order and randomness*, Princeton University Press, Princeton, NJ.

Watts, D. J. & Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks., *Nature* 393(6684): 440–442.
URL: *http://dx.doi.org/10.1038/30918*

Zhou, Y. & Tan, Y. (2009). Gpu-based parallel particle swarm optimization, *Evolutionary Computation, 2009. CEC '09. IEEE Congress on* pp. 1493 –1500.

Zhu, W. & Curry, J. (2009). Particle swarm with graphics hardware acceleration and local pattern search on bound constrained problems, *Swarm Intelligence Symposium, 2009. SIS '09. IEEE* pp. 1 –8.