Three Alternatives for Parallel GPU-Based Implementations of High Performance Particle Swarm Optimization

Rogério M. Calazan[1], Nadia Nedjah[2], and Luiza de Macedo Mourelle[3]

[1] Department of Telecommunications and Information Technology, Brazilian Navy, Brazil
[2] Department of Electronics Engineering and Telecommunication
[3] Department of System Engineering and Computation, Engineering Faculty, State University of Rio de Janeiro, Brazil
{rogerio,nadia,ldmm}@eng.uerj.br

Abstract. Particle Swarm Optimization (PSO) is heuristics-based method, in which the solution candidates of a problem go through a process that simulates a simplified model of social adaptation. In this paper, we propose three alternative algorithms to massively parallelize the PSO algorithm and implement them using a GPGPU-based architecture. We aim at improving the performance of computationally demanding optimizations of many-dimensional problems. The first algorithm parallelizes the particle's work. The second algorithm subdivides the search space into a grid of smaller domains and distributes the particles among them. The optimization subprocesses are performed in parallel. The third algorithm focuses on the work done with respect to each of the problem dimensions and does it in parallel. Note that in the second and third algorithms, all particles act in parallel too. We analyze and compare the speedups achieved by the GPU-based implementations of the proposed algorithms, showing the highlights and limitations imposed.

1 Introduction

Particle Swarm Optimization (PSO) was introduced by Kennedy and Eberhart [1] and is based on collective behavior, social influence and learning. Many successful applications of PSO have been reported, in which this algorithm has shown many advantages over other algorithms based on swarm intelligence, mainly due to its robustness, efficiency and simplicity. Moreover, it usually requires less computational effort when compared to other stochastic algorithms [2]. The PSO algorithm maintains a swarm of particles, where each of which represents a potential solution. In analogy with evolutionary computation, a *swarm* can be identified as the population, while a *particle* with an individual. In general terms, the particle flows through a multidimensional search space, where the corresponding position is adjusted according to its own experience and that of its neighbors [2].

I. Rojas, G. Joya, and J. Cabestany (Eds.): IWANN 2013, Part I, LNCS 7902, pp. 241–252, 2013.

Several works show that PSO implementation in GPGPU provide a better performance than CPU-based implementations [9] [10] [11]. In contrast, the purpose of this paper is to implement the Global Best version of PSO in GPGPUs. In order to take full advantage of the massively parallel nature of GPGPUs, we explore three different scenarios: *(i)* In the first proposed approach, the work done by the particles of the swarm is performed in parallel until a synchronization is required. Nonetheless, the work done by the particle itself is performed sequentially. Hence, here each thread is associated with a given particle of the swarm. *(ii)* In the second approach, the search space is divided into a grid of smaller subspaces. Then, swarms of particles are formed and assigned to search the subdomains. The swarms act simultaneously. Moreover, within each swarm, particles act in parallel until a synchronization point, during which they exchange knowledge acquired so far, individually. It is worth noting that there is no cooperative work among the swarms. So, there is no exchange of information about best position found by the groups. *(iii)* The third approach explores a fine-grained parallelism, which consists of doing the computational work with respect to each of the problem dimensions in parallel. As in the first approach, this one also handles a single swarm of particles. Nonetheless, here a thread corresponds to a given dimension of the problem and a block of threads to a given particle. This approach should favor optimization problems with high dimensionality.

An analysis is done in order to identify the number of swarms and particles per swarm as well as how to map the swarms into blocks and particles into threads, aiming at maximizing performance. Furthermore, we study the impact of the grid resolution on the convergence time. The grid resolution is defined by the number of cells used. It coincides with the number of swarms invested in the search. Finally, we study the change of the number of dimensions between the implementations.

This paper is organized as follows: First, in Section 2, we sketch briefly the PSO process and the algorithm; After that, in Section 3, we describe the first approach: PPSO; In the sequel, in Section 4, we describe the second approach: SGPSO; Then, in Section 5, we describe the third approach: PDPSO; Subsequently, in Section 6, we present and analyze the obtained results; Finally, in Section 7, we draw some concluding remarks and point out directions for future work.

2 Particle Swarm Optimization

The main steps of the PSO algorithm are described in Algorithm 1. Note that, in this specification, the computations are executed sequentially. In this algorithm, each particle has a *velocity* and an *adaptive direction* [1] that determine its next movement within the search space. The particle is also endowed with a memory that makes it able to remember the best previous position it passed by.

In this variation of the PSO algorithm, the neighborhood of each particle is formed by all the swarm's particles. Using this strategy, the social component of the particle's velocity is influenced by all other particles [2] [3]. The velocity

Algorithm 1. PSO

```
for i = 1 to n do
  randomly initialize position and velocity of particle i
repeat
  for i = 1 to n do
    compute the Fitness_i of particle i
    if Fitness_i ≤ Pbest then
      update Pbest using the position of particle i
    if Fitness_i ≤ Gbest then
      update Gbest using the position of particle i
    update the velocity of particle i; update the position of particle i
until stopping criterion
return Gbest and corresponding position
```

is the element that promotes the capacity of particle locomotion and can be computed as described in (1) [1] [2], wherein w is called *inertia weight*, r_1 and r_2 are random numbers in [0,1], c_1 and c_2 are positive constants, y_{ij} is the best position *Pbest* found by the particle i so far, w.r.t. dimension j, and y_j is the best position *Gbest*, w.r.t. dimension j, found so far, considering all the population's particles. The position of each particle is also updated as described in (1). Note that $x_{i,j}^{(t+1)}$ is the current position and $x_{i,j}^{(t)}$ is the previous position.

$$v_{i,j}^{(t+1)} = wv_{i,j}^{(t)} + c_1 r_1 \left(y_{i,j} - x_{i,j}^{(t)}\right) + c_2 r_2 \left(y_j - x_{i,j}^{(t)}\right), \quad x_{i,j}^{(t+1)} = v_{i,j}^{(t+1)} + x_{i,j}^{(t)} \quad (1)$$

The velocity component drives the optimization process, reflecting both the experience of the particle and the exchange of information between the particles. The particle's experimental knowledge is referred to as the cognitive behavior, which is proportional to the distance between the particle and its best position found, with respect to its first iteration [3]. The maximum velocity $v_{k,max}$ is defined for each dimension k of the search space. It can be expressed as a percentage of this space by $v_{k,max} = \delta(x_{k,max} - x_{k,min})$, wherein $x_{k,max}$ and $x_{k,min}$ are the maximum and minimum limits of the search space explored, with respect to dimension k, respectively and $\delta \in [0, 1]$.

3 First Algorithm: PPSO

The first proposed algorithm, called PPSO, follows from the idea that the work performed by a given particle is independent of that done by the other particles of the swarm, except in terms of *Gbest*, and thus the computation done by the particles could be executed simultaneously. This algorithm has a synchronization point at the election of *Gbest*, wherein $p_1, \ldots p_n$ denote the n particles of the swarm, and $v^{(p_1)}, \ldots v^{(p_n)}$ and $x^{(p_1)}, \ldots x^{(p_n)}$ the respective velocities and positions. Each particle computes the corresponding fitness, velocity and position, independently and in parallel with the other particles, until the election of *Gbest*. In order to synchronize the process and prevent using incorrect values of *Gbest*,

Algorithm 2. CUDA Pseudo-code for PPSO

```
let b = number of blocks; let t = number of threads
kernel⟨b, t⟩ position and velocity random generators
repeat
  kernel⟨b, t⟩ fitness calculator; kernel⟨b, t⟩ velocity and position calculator
until stopping condition
transfer result back to CPU
return Gbest and corresponding position;
```

Algorithm 3. CUDA Pseudo-code of kernel fitness calculator

```
let tid = threadIdx + blockIdx × blockDim
compute fitness of particle tid; update Pbest of particle tid
if (tid = 0) then
  compute Gbest of swarm
```

the velocity and position computations can only commence once *Gbest* has been chosen among the *Pbest* values of all particles of the swarm [4] [5]. Note that the verification of the stopping criterion achievement is also done synchronously by the parallel processes, but it does not hinder the performance of the algorithm.

The CUDA pseudo-code of algorithm PPSO is shown in Algorithm 2. Algorithm 3 shows the code executed by thread *tid* associated with a given particle of the swarm. Note that the processes corresponding to the n threads launched within a kernel are executed in parallel. Recall that, in this first approach, each particle is mapped onto a single thread. The algorithm uses b blocks and t threads per block. Thus, the total number of particles is $b \times t$. In Algorithm 3, a particle thread *tid* identification is done relatively to the associated thread, identified by *threadIdx*, and block, identified by *blockIdx*, and number of of threads per block, identified by *blockDim*.

4 Second Algorithm: SGPSO

The main idea behind the second approach consists of subdividing the search space into a grid of cells, where each cell is searched by an independent swarm of particles. This approach should favor optimization problems with large search space. In [6], we studied the impact of the number and size of the swarms on the optimization process, in terms of the execution time, convergence and quality of the solution found.

The dimension and size of blocks per grid and the dimension and size of threads per block are both important factors. The number of blocks in a grid should be at least the same or larger than the number of streaming multiprocessors (SMs), so that all available SM have at least one block to execute. Furthermore, there should be multiple active blocks per SM, so that blocks that are not waiting, due to a synchronization point, can keep the hardware busy. This recommendation is subject to resource availability. Therefore, it should be

determined in the context of the second execution parameter, which is the number of threads per block, or block size, as well as shared memory usage.

In the proposed parallel implementation, the maximum velocity $v_{i,max}$ with respect to dimension i is formulated as a percentage of the considered search subspace of size D_i for that dimension, as defined in (2), wherein x_{max} and x_{min} are the maximum and minimum values of the whole search space, N_s represents the number of swarms, that work in parallel, and $0 \leq \delta \leq 1$. Moreover, the search space for a given swarm i is delimited by $x_{i,min}$ and $x_{i,max}$. In order to increase the efficiency of the algorithm in high dimensions, we use dynamic update of the inertia weight w.

$$\begin{aligned} D_i &= (x_{max} - x_{min})/N_s, \; v_{i,max} = \delta * D_i \\ x_{i,min} &= i \times D_i + x_{min}, \qquad x_{i,max} = x_{i,min} + D_i \end{aligned} \tag{2}$$

In order to implement the SGPSO approach using CUDA, we opted to exploit two kernels. The first kernel generates random numbers and stores them in the GPU global memory. The second kernel runs all the steps of the parallel PSO. This way, the host CPU, after triggering the PSO process, becomes totally free. Using a single kernel for the whole PSO, excluding the random number generation, allows us to optimize the implementation, as there is no need for host/device communications. Recall that kernel *particle swarm optimizer* updates the inertia weight dynamically.

As introduced earlier, the problem search domain is organized into a grid of swarms, wherein each swarm is implemented as a block and each particle as a thread. The grid size is the number of swarms and block size is the number of particles. So, population size can be defined as the product of the grid size and block size, and this coincides with the total number of threads run by the GPU infrastructure. In this implementation, the position, velocity and *Pbest* of all the particles are kept in the global memory on the GPU chip.

Nonetheless, the *Gbest* obtained for all the grid's swarms are stored in the shared memory of the respective SM. The CUDA pseudo-code for the approach behind SGPSO is shown in Algorithm 4, wherein s denotes the number of segments into which each of the d dimensions of the problem is divided. Note that this subdivision generates s^d voxels which are the search subspaces. The code launches t threads per block, which means that it starts t particles per subspace. The total number of particles is thus $t \times s^d$. Kernel *particle swarm optimizer* proceeds as described in Algorithm 5. Note, that in this approach, the number of blocks coincides with that of swarms and the number of threads coincides with that of particles in each swarm.

The initialization of positions and velocity as well as the maximal velocity allowed for a particle within a swarm is done as described in Algorithm 6.

5 Third Algorithm: PDPSO

The third approach considers the fact that in some computationally demanding optimization problems the objective function is based on a large number of

Algorithm 4. CUDA Pseudo-code for SGPSO

let s = the number of segments; **let** d = the number of dimensions
let $b = s^d$ be the number of blocks ; **let** t = the number of threads
generate the swarm grid according the s and d
transfer data of the grid from CPU to GPU
kernel$\langle b, t \rangle$ random number generator; **kernel**$\langle b, t \rangle$ particle swarm optimizer
transfer result back to CPU
return *Gbest* and corresponding position;

Algorithm 5. CUDA Pseudo-code for particle swarm optimizer

initialize randomly position and velocity of particles according to subspace of respective swarm *blockIdx*
repeat
  compute *fitness* of particle *threadIdx*; **update** *Pbest* of particle *threadIdx*
  if (*threadIdx* = 0) **then**
    update *Gbest* of respective swarm *blockIdx*
  synchronize all threads of swarm *blockIdx*
  update velocity and position of particle *threadIdx*
until stopping condition

dimensions. Here, we are talking about more than thirty different dimensions and can even reach 100. Therefore, in this approach, the parallelism is more fine-grained as it is associated with the problem dimensions. The algorithm is called PDPSO (*Parallel Dimension PSO*). In contrast with SGPSO, this algorithm handles only one swarm and its main characteristic is the parallelism at the dimension level. Thus, the particle is now implemented as a block wherein each dimension is a thread of the block. This should favor optimization problems that exhibit a very high dimensionality. The GPU grid size is the number of particles and block size is the number of dimensions. For example, if the number of dimensions of the problem is 100, the SGPSO needs 100 iterations to compute the fitness values. The PDPSO will do the job using a single iteration to obtain the fitness values with respect to each of the problem dimensions plus an extra 10 iterations to summarize these intermediary results in order to get a single value which is the particle fitness. We call this process the *fitness reduction*. Thus, after 11 steps the result will be ready. Thus, it is possible to distribute the computational load at a lower degree of granularity, which can be up to one thread per problem dimension.

The PDPSO algorithm written in a CUDA-based pseudo-code is given in Algorithm 7. It uses four kernels: The first one launches the random number generators, i.e. one for each particle dimension and initialize the positions and velocities of the particles; The second kernel generates the threads that compute the fitness according to the corresponding dimension, perform the reduction process to get the fitness value of the particle that is represented by the block, and when this is completed, checks whether *Pbest* needs to be updated.

Algorithm 6. CUDA Pseudo-code position and maximum velocity initialization in subspace *blockIdx*

let $k = blockIdx$
for $i = 1$ *to* d **do**
  $x_i := (k \times d + i)(rand(max_k - min_k) + min_k)$; $v_i := 0.0f$
  $vmax_i := \delta(k \times d + i)(max_k - min_k)$

Algorithm 7. CUDA Pseudo-code for PDPSO

let t = number of threads (dimensions); **let** b = number of blocks (particles)
kernel$\langle b, t\rangle$ position and velocity random generators (one for each dimension)
repeat
  kernel$\langle b, t\rangle$ fitness and *Pbest* calculator (one for each dimension)
  kernel$\langle b, t\rangle$ *Gbest* elector
  kernel$\langle b, t\rangle$ velocity and position calculator (one for each dimension)
until stopping condition
transfer result back to CPU
return *Gbest* and corresponding position

Algorithm 8. CUDA Pseudo-code for fitness, *Pbest* calculator

let $j = blockIdx$, $k = threadIdx$ and $b = blockDim$; $tid = k + j * b$
let cache be the shared memory of the GPU where $cache[k] = x[tid]$
compute fitness with respect to dimension k; $i := t/2$
while $(i \neq 0)$ **do**
  if $(k < i)$ **then**
    reduce $fitness[k]$ and $fitness[k + i]$ according to objective function
  $i := i/2$
synchronize all threads
if $(fitness[j] < Pbest[j])$ **then**
  $Pbestx[(j \times d) + k] := cache[(j \times d) + k]$
  if $k = 0$ **then**
    $Pbest[j] := fitness[j]$

Algorithm 9. CUDA Pseudo-code for kernel *Gbest* elector

let $tid = threadIdx + blockIdx * blockDim$; $i := b/2$
while $(i \neq 0)$ **do**
  if $(tid < i)$ **and** $Gbest[tid + i] < Gbest[tid]$ **then**
    $Gbest[tid] := Gbest[tid + i]$
  $i := i/2$

If this is the case, the threads update the coordinates associated with this new *Pbest*. Note that there is a synchronization point of all threads so as to use the fitness value only when all the fitness reduction process has been completed.

6 Performance Results

The three proposed approaches were implemented on a NVDIA GeForce GTX 460 GPU [7]. This GPU contains 7 SMs with 48 CUDA cores each, hence a total of 336 cores. Three classical benchmark functions, as listed in Table 1, were used to evaluate the implementations performance. Function f_1 defines a Sphere, f_2 is Griewank function and f_3 is the Rastrigin function.

In the following, we report on the experiments performed to analyze the impact of each one of the proposed approaches. In all experiments, we always run the PSO algorithms for 2000 iterations.

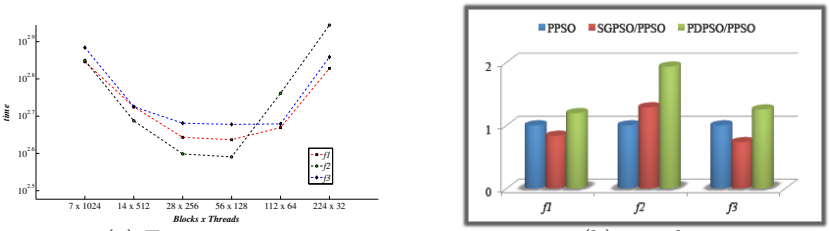
Table 1. Fitness Functions

Function	Domain	f_{min}
$f_1(x) = \sum_{i=1}^{n} (x_i^2)$	$(-100, 100)^n$	0
$f_2(x) = 1 + \frac{1}{4000} \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} cos\left(\frac{x_i}{\sqrt{i}}\right)$	$(-600, 600)^n$	0
$f_3(x) = \sum_{i=1}^{n} (x_i^2 - 10cos(2\pi x_i) + 10)$	$(-10, 10)^n$	0

6.1 Impact of the Swarm Number

Using the CUDA Occupancy calculator [7], the GPU occupancy, which depends on the number of threads per block and that of register as well as the size of the kernel shared memory, amounts to 67%. Note that in all verified cases of different pairs of number and size of blocks per SM, the total number of 7168 threads was kept constant. In the case of SGPSO, this means a total number of particles of 7168 was used, as it is also the case of PPSO. However, in the case of PDPSO, as threads correspond to dimensions in the particles, which is 32 in this experiment, hence the number of particles sums up to 224 only. This explains the poor performance presented by this algorithm. Nonetheless, the disposition of block and thread numbers had significant impact on the performance in the case of SGPSO. Fig. 1(a) shows that despite the fact that the total number of particles is the same in all checked dispositions of number of swarms and particles per swarm, the combination 56×128 leads to the lowest execution time for SGPSO.

The increase in execution time can be explained by the work granularity level that each block of threads is operating at. Parallel computation of position coordinates and subsequently the velocity are performed by all threads within a block, but conditional branches, used to elect *Pbest* and *Gbest*, as well as loops that allow the iteration of the work for each one of the problem dimensions, dominate most part of the thread computation. It is well-known that conditional constructions are not well suited for the Stream Processing model.



(a) Execution time (b) speedup

Fig. 1. Execution times for different configurations of swarms for SGPSO and Impact of the total number of particles

Also, the performance degenerates because more blocks of threads are competing for the resources available to the SMs. A GPU offers a limited amount of shared memory, which limits the number of threads that can be simultaneously executed in the SM for a given application. In general, the more memory each thread requires, the fewer the number of threads that can reside in the processor [8]. Therefore, the choice of pair (block number and block size) has the kind of effect illustrated in Fig. 1(a) on the execution time. This experiment was repeated for different problem dimensions. The observed behavior is confirmed independently of this parameter. The case reported here is for dimension 32. Figure 1(b) shows the speedup achieved. Note that due to the stochastic nature of PSO, we run the same optimization 50 times.

6.2 Impact of the Swarm Size

It is expected that the number of particles influences positively the convergence speed of the optimization process, yet it has a negative impact on the

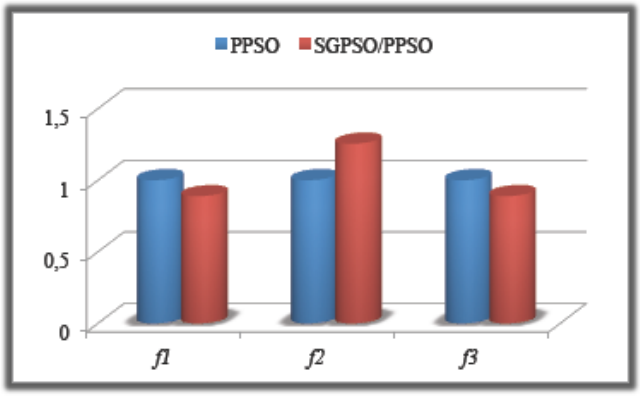
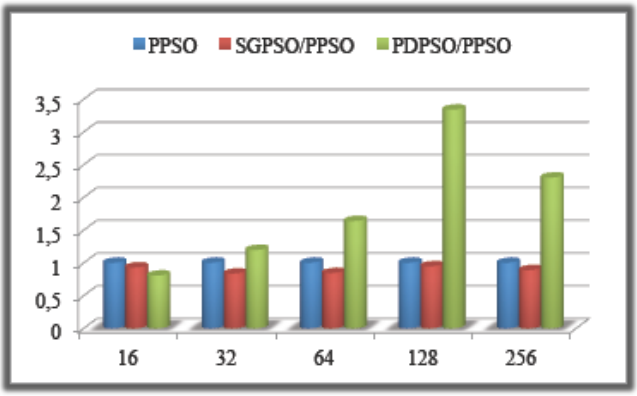
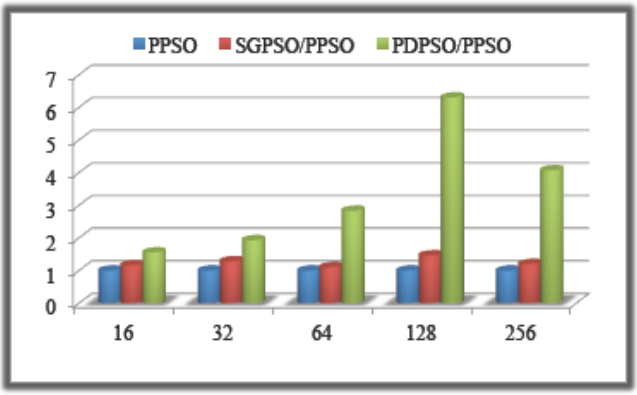


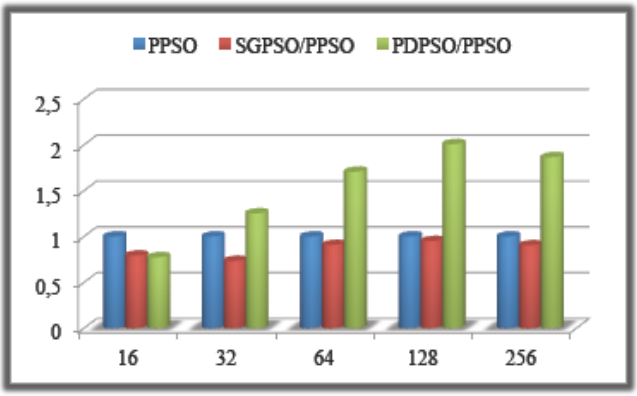
Fig. 2. Impact of the total number of particle



(a) f_1



(b) f_2



(c) f_3

Fig. 3. Impact of the number of dimensions for functions used as benchmarks

corresponding processing time. In SGPSO, increasing the number of particles can be achieved by either increasing the number of swarms and/or the number of particles per swarm. In order to study the impact of this parameter on the performance of SGPSO, we opted to keep the number of particles in a given swarm constant, i.e. 128, and increase the number of swarms. The latter was set as a multiple of the available streaming multiprocessors. Recall that the GPU used here includes 7 SMs.

Considering the optimization time comparison for the different studied configurations, with respect to the three used functions, we could easily observe that, in the case of SGPSO, for at most 56 swarms, which entails $56 \times 128 = 7168$ particles, the increase in terms of processing time is justified as the quality of the best solution is improved proportionally. PPSO presented a similar behavior as SGPSO when varying upwards the number of particles. Nonetheless, one can notice that for function f_2, SGPSO performs better, which in our opinion is due to the large search space and thus SGPSO takes advantage of the topology of distributed swarms. In the case of PDPSO, because of the explosion in terms of number of required threads, even in the first case, wherein a total of 28672 threads are required, the computational work surely ends up being sequentialized. Therefore, we do not show all the results for this approach. Figure 2 shows the speedup achieved.

6.3 Impact of the Number of Dimensions

Surely, the increase in terms of problem dimensions has an impact on the execution time. Recall that, in PPSO and SGPSO approaches, the computation with respect to the many dimensions of the objective function are performed sequentially, while in PDPSO, this is done concurrently. Figures 3(a) – 3(c) show a positive speedup for at most 256 dimensions. Nonetheless, for 512 dimensions the rate of increase of the performance deteriorates for all three implementations. The implementation of PDPSO performed much better than PPSO and SGPSO, even though the latter (PPSO and SGPSO) are handling 7168 threads while PDPSO 14336. This is twice the whole capacity of the GPU.

7 Conclusion

This paper presents three implementations of parallel PSO using GPGPU: PPSO, SGPSO and PDPSO. The first approach explores the parallelism between particles. In the second approach, the algorithm divides the search space into a grid of subspaces and assigns a swarm to each and every one of them. The implementation exploits the parallelism of the particle computation of the corresponding position and velocity as well as the fitness value of the solution associated. This is performed independently of the others particles of the swarm. A swarm of particles was implemented as a block of threads, wherein each thread simulates a single particle. This has a positive impact on the performance of the optimization of problems with large search space. In the third implementation, the

particle is implemented as a block of threads and each dimension as one thread. This allows the distribution of the computational load at a finer degree of granularity, which is up to one thread per problem dimension. This has a positive impact on the performance of the optimization of large dimension problems.

A three-fold analysis was carried out to evaluate the performance of the proposed parallel implementation: first, the impact of the number of invested swarms; second the impact of their size; then the impact of the number of dimensions.

References

1. Kennedy, J., Eberhart, R.: Particle Swarm Optimization. In: Proc. of IEEE International Conference on Neural Network, Australia, pp. 1942–1948. IEEE Press (1995)
2. Engelbrecht, A.P.: Fundamentals of Computational Swarm Intelligence. John Wiley & Sons Ltd., New Jersey (2005)
3. Nedjah, N., Coelho, L.S., Mourelle, L.M.: Multi-Objective Swarm Intelligent Systems − Theory & Experiences. Springer, Berlin (2010)
4. Calazan, R.M., Nedjah, N., Mourelle, L.M.: Parallel co-processor for PSO. Int. J. High Performance Systems Architecture 3(4), 233–240 (2011)
5. Calazan, R.M., Nedjah, N., Mourelle, L.M.: A Massively Parallel Reconfigurable Co-processor for Computationally Demanding Particle Swarm Optimization. In: 3rd International Symposium of IEEE Circuits and Systems in Latin America, LASCAS 2012. IEEE Computer Press, Los Alamitos (2012)
6. Calazan, R.M., Nedjah, N., de Macedo Mourelle, L.: Swarm Grid: A Proposal for High Performance of Parallel Particle Swarm Optimization Using GPGPU. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2012, Part I. LNCS, vol. 7333, pp. 148–160. Springer, Heidelberg (2012)
7. NVIDIA: NVIDIA CUDA C Programming Guide, Version 4.0 NVIDA Corporation (2011)
8. Kirk, D.B., Hwu, W.-M.W.: Programming Massively Parallel Processors. Morgan Kaufmann, San Francisco (2010)
9. Veronese, L., Krohling, R.A.: Swarm's flight: accelerating the particles using C-CUDA. In: 11th IEEE Congress on Evolutionary Computation, pp. 3264–3270. IEEE Press, Trondheim (2009)
10. Zhou, Y., Tan, Y.: GPU-based parallel particle swarm optimization. In: 11th IEEE Congress on Evolutionary Computation (CEC 2009), pp. 1493–1500. IEEE Press, Trondheim (2009)
11. Cádenas-Montes, M., Vega-Rodríguez, M.A., Rodríguez-Vázquez, J.J., Gómez-Iglesias, A.: Accelerating Particle Swarm Algorithm with GPGPU. In: 19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 560–564. IEEE Press, Cyprus (2011)