2

Analysis of the Performance of the Fish School Search Algorithm Running in Graphic Processing Units

Anthony J. C. C. Lins, Carmelo J. A. Bastos-Filho, Débora N. O. Nascimento, Marcos A. C. Oliveira Junior and Fernando B. de Lima-Neto
Polytechnic School of Pernambuco, University of Pernambuco
Brazil

1. Introduction

Fish School Search (FSS) is a computational intelligence technique invented by Bastos-Filho and Lima-Neto in 2007 and first presented in Bastos-Filho et al. (2008). FSS was conceived to solve search problems and it is based on the social behavior of schools of fish. In the FSS algorithm, the search space is bounded and each possible position in the search space represents a possible solution for the problem. During the algorithm execution, each fish has its positions and weights adjusted according to four FSS operators, namely, feeding, individual movement, collective-instinctive movement and collective-volitive movement. FSS is inherently parallel since the fitness can be evaluated for each fish individually. Hence, it is quite suitable for parallel implementations.

In the recent years, the use of Graphic Processing Units (GPUs) have been proposed for various general purpose computing applications. Thus, GPU-based platforms afford great advantages on applications requiring intensive parallel computing. The GPU parallel floating point processing capacity allows one to obtain high speedups. These advantages together with FSS architecture suggest that GPU based FSS may produce marked reduction in execution time, which is very likely because the fitness evaluation and the update processes of the fish can be parallelized in different threads. Nevertheless, there are some aspects that should be considered to adapt an application to be executed in these platforms, such as memory allocation and communication between blocks.

Some computational intelligence algorithms already have been adapted to be executed in GPU-based platforms. Some variations of the Particle Swarm Optimization (PSO) algorithm suitable for GPU were proposed by Zhou & Tan (2009). In that article the authors compared the performance of such implementations to a PSO running in a CPU. Some tests regarding the scalability of the algorithms as a function of the number of dimensions were also presented. Bastos-Filho et al. (2010) presented an analysis of the performance of PSO algorithms when the random number are generated in the GPU and in the CPU. They showed that the XORshift Random Number Generator for GPUs, described by Marsaglia (2003), presents enough quality to be used in the PSO algorithm. They also compared different GPU-based versions of the PSO (synchronous and asynchronous) to the CPU-based algorithm.

Zhu & Curry (2009) adapted an Ant Colony Optimization algorithm to optimize benchmark functions in GPUs. A variation for local search, called SIMT-ACO-PS (*Single Instruction Multiple Threads - ACO - Pattern Search*), was also parallelized. They presented some interesting analysis on the parallelization process regarding the generation of ants in order to minimize the communication overhead between CPU-GPU. The proposals achieved remarkable speedups.

To the best of our knowledge, there is no FSS implementations for GPUs. So, in this paper we present the first parallel approach for the FSS algorithm suitable for GPUs. We discuss some important issues regarding the implementation in order to improve the time performance. We also consider some other relevant aspects, such as when and where it is necessary to set synchronization barriers. The analysis of these aspects is crucial to provide high performance FSS approaches for GPUs. In order to demonstrate this, we carried out simulations using a parallel processing platform developed by NVIDIA, called CUDA.

This paper is organized as follows: in the next Section we present an overview of the FSS algorithm. In Section 3, we introduce some basic aspects of the NVIDIA CUDA Architecture and GPU Computing. Our contribution and the results are presented in Sections 4 and 5, respectively. In the last Section, we present our conclusions, where we also suggest future works.

2. Fish School Search

Fish School Search (FSS) is a stochastic, bio-inspired, population-based global optimization technique. As mentioned by Bastos-Filho et al. (2008), FSS was inspired in the gregarious behavior presented by some fish species, specifically to generate mutual protection and synergy to perform collective tasks, both to improve the survivability of the entire group.

The search process in FSS is carried out by a population of limited-memory individuals - the fish. Each fish in the school represents a point in the fitness function domain, like the particles in the Particle Swarm Optimization (PSO) Kennedy & Eberhart (1995) or the individuals in the Genetic Algorithms (GA) Holland (1992). The search guidance in FSS is driven by the success of the members of the population.

The main feature of the FSS is that all fish contain an innate memory of their success - their weights. The original version of the FSS algorithm has four operators, which can be grouped in two classes: feeding and swimming. The Feeding operator is related to the quality of a solution and the three swimming operators drive the fish movements.

2.1 Individual movement operator

The individual movement operator is applied to each fish in the school in the beginning of each iteration. Each fish chooses a new position in its neighbourhood and then, this new position is evaluated using the fitness function. The candidate position $\vec{n}_i$ of fish i is determined by the Equation (1) proposed by Bastos-Filho et al. (2009).

$$\vec{n}_i(t) = \vec{x}_i(t) + rand[-1, 1].step_{ind}, \quad (1)$$

where $\vec{x}_i$ is the current position of the fish in dimension i, rand[-1,1] is a random number generated by an uniform distribution in the interval [-1,1]. The $step_{ind}$ is a percentage of the search space amplitude and is bounded by two parameters ($step_{ind\_min}$ and $step_{ind\_max}$).

The $step_{ind}$ decreases linearly during the iterations in order to increase the exploitation ability along the iterations. After the calculation of the candidate position, the movement only occurs if the new position presents a better fitness than the previous one.

2.2 Feeding operator

Each fish can grow or diminish in weight, depending on its success or failure in the search for food. Fish weight is updated once in every FSS cycle by the feeding operator, according to equation (2).

$$W_i(t+1) = W_i(t) + \frac{\Delta f_i}{max(\Delta f)}, \tag{2}$$

where $W_i(t)$ is the weight of the fish i, $f[\vec{x}_i(t)]$ is the value for the fitness function (*i.e.* the amount of food) in $\vec{x}_i(t)$, Δf_i is the difference between the fitness value of the new position $f[\vec{x}_i(t+1)]$ and the fitness value of the current position for each fish $f[\vec{x}_i(t)]$, and the $max(\Delta f)$ is the maximum value of these differences in the iteration. A weight scale (W_{scale}) is defined in order to limit the weight of fish and it will be assigned the value for half the total number of iterations in the simulations. The initial weight for each fish is equal to $\frac{W_{scale}}{2}$.

2.3 Collective-instinctive movement operator

After all fish have moved individually, their positions are updated according to the influence of the fish that had successful individual movements. This movement is based on the fitness evaluation of the fish that achieved better results, as shown in equation (3).

$$\vec{x}_i(t+1) = \vec{x}_i(t) + \frac{\sum_{i=1}^{N} \Delta\vec{x}_{ind_i}\{f[\vec{x}_i(t+1)] - f[\vec{x}_i(t)]\}}{\sum_{i=1}^{N} \{f[\vec{x}_i(t+1)] - f[\vec{x}_i(t)]\}}, \tag{3}$$

where $\Delta\vec{x}_{ind_i}$ is the displacement of the fish i due to the individual movement in the FSS cycle. One must observe that $\Delta\vec{x}_{ind_i} = 0$ for fish that did not execute the individual movement.

2.4 Collective-volitive movement operator

The collective-volitive movement occurs after the other two movements. If the fish school search has been successful, the radius of the school should contract; if not, it should dilate. Thus, this operator increases the capacity to auto-regulate the exploration-exploitation granularity. The fish school dilation or contraction is applied to every fish position with regards to the fish school barycenter, which can be evaluated by using the equation (4):

$$\vec{B}(t) = \frac{\sum_{i=1}^{N} \vec{x}_i(t) W_i(t)}{\sum_{i=1}^{N} \vec{x}_i(t)}. \tag{4}$$

We use equation (5) to perform the fish school expansion (in this case we use sign $+$) or contraction (in this case we use sign $-$).

$$\vec{x}_i(t+1) = \vec{x}_i(t) \pm step_{vol} r_1 \frac{\vec{x}_i(t) - \vec{B}(t)}{d(\vec{x}_i(t), \vec{B}(t))}, \tag{5}$$

where r_1 is a number randomly generated in the interval $[0, 1]$ by an uniform probability density function. $d(\vec{x}_i(t), \vec{B}(t))$ evaluates the euclidean distance between the particle i and the barycenter. $step_{vol}$ is called volitive step and controls the step size of the fish. $step_{vol}$ is defined as a percentage of the search space range and is bounded by two parameters ($step_{vol\_min}$ and $step_{vol\_max}$). $step_{vol}$ decreases linearly from $step_{vol\_max}$ to $step_{vol\_min}$ along the iterations of the algorithm. It helps the algorithm to initialize with an exploration behavior and change dynamically to an exploitation behavior.

3. GPU computing and CUDA architecture

In recent years, Graphic Processing Units (GPU) have appeared as a possibility to drastically speed up general-purpose computing applications. Because of its parallel computing mechanism and fast float-point operation, GPUs were applied successfully in many applications. Some examples of GPU applications are physics simulations, financial engineering, and video and audio processing. Despite all successful applications, some algorithms can not be effectively implemented for GPU platforms. In general, numerical problems that present parallel behavior can obtain profits from this technology as can be seen in NVIDIA (2010a).

Even after some efforts to develop Applications Programming Interface (API) in order to facilitate the developer activities, GPU programming is still a hard task. To overcome this, NVIDIA introduced a general purpose parallel computing platform, named Computer Unified Device Architecture (CUDA). CUDA presents a new parallel programming model to automatically distribute and manage the threads in the GPUs.

CUDA allows a direct communication of programs, written in C programming language, with the GPU instructions by using minimal extensions. It has three main abstractions: a hierarchy of groups of threads, shared memories and barriers for synchronization NVIDIA (2010b). These abstractions allow one to divide the problem into coarse sub-problems, which can be solved independently in parallel. Each sub-problem can be further divided in minimal procedures that can be solved cooperatively in parallel by all threads within a block. Thus, each block of threads can be scheduled on any of the available processing cores, regardless of the execution order.

Some issues must be considered when modeling the Fish School Search algorithm for the CUDA platform. In general, the algorithm correctness must be guaranteed, once race conditions on a parallel implementation may imply in outdated results. Furthermore, since we want to execute the algorithm as fast as possible, it is worth to discuss where it is necessary to set synchronization barriers and in which memory we shall store the algorithm information.

The main bottleneck in the CUDA architecture lies in the data transferring between the host (CPU) and the device (GPU). Any transfer of this type may reduce the time execution performance. Thus, this operation should be avoided whenever possible. One alternative is to move some operations from the host to the device. Even when it seems to be unnecessary (not so parallel), the generation of data in the GPU is faster than the time needed to transfer huge volumes of data.

CUDA platforms present a well defined memory hierarchy, which includes distinct types of memory in the GPU platform. Furthermore, the time to access these distinct types of memory vary. Each thread has a private local memory and each block of threads has a shared memory

accessible by all threads inside the block. Moreover, all threads can access the same global memory. All these memory spaces follow a memory hierarchy: the fastest one is the local memory and the slowest is the global memory; accordingly the smallest one is the local memory and the largest is the global memory. Then, if there is data that must be accessed by all threads, the shared memory might be the best choice. However, the shared memory can only be accessed by the threads inside its block and its size is not very large. On the FSS versions, most of the variables are global when used on kernel functions. Shared memory was also used to perform the barycenter calculations. Local memory were used to assign the thread, block and grid dimension indexes on the device and also to compute the specific benchmark function.

Another important aspect is the necessity to set synchronization barriers. A barrier forces a thread to wait until all other threads of the same block reach the barrier. It helps to guarantee the correctness of the algorithm running on the GPU, but it can reduce the time performance. Furthermore, threads within a block can cooperate among themselves by sharing data through some shared memory and must synchronize their execution to coordinate the memory accesses (see Fig. 1). Although the GPUs are famous because of their

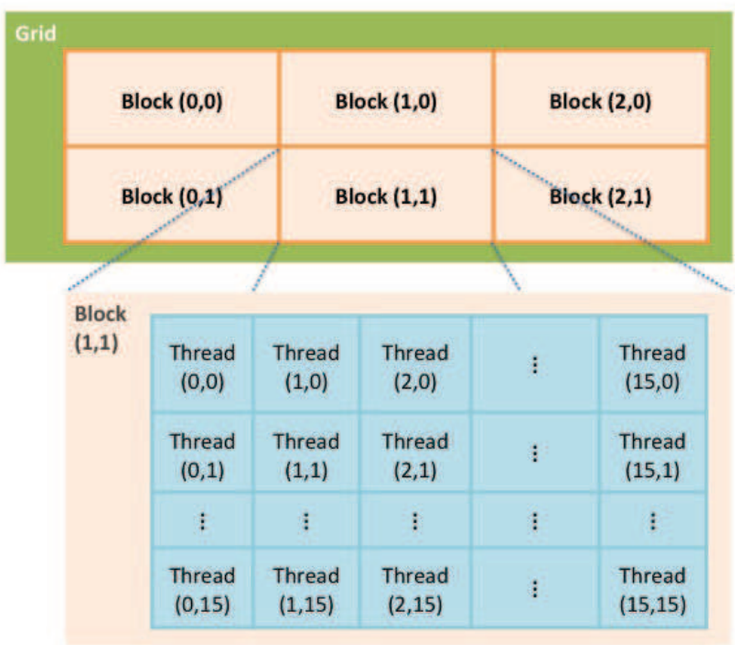


Fig. 1. Illustration of a Grid of Thread Blocks

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 bad results. Therefore, it turns out that these GPUs are inappropriate to solve some types of problems.

The CUDA capacity to execute a high number of threads in parallel is due to the hierarchical organization of these threads as a grid of blocks. A thread block is set of processes which

cooperate in order to share data efficiently using a fast shared memory. Besides, a thread block must synchronize themselves to coordinate the accesses to the memory.

The maximum number of threads running in parallel in a block is defined by its number of processing units and its architecture. Therefore, each GPU has its own limitation. As a consequence, an application that needs to overpass this limitation have to be executed sequentially with more blocks, otherwise it might obtain wrong or, at least, outdated results.

The NVIDIA CUDA platform classify the NVIDIA GPUs using what they call Compute Capability as depicted in NVIDIA (2010b). The cards with double-precision floating-point numbers have Compute Capability 1.3 or 2.x. The cards with 2.x Capability can run up to 1,024 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.

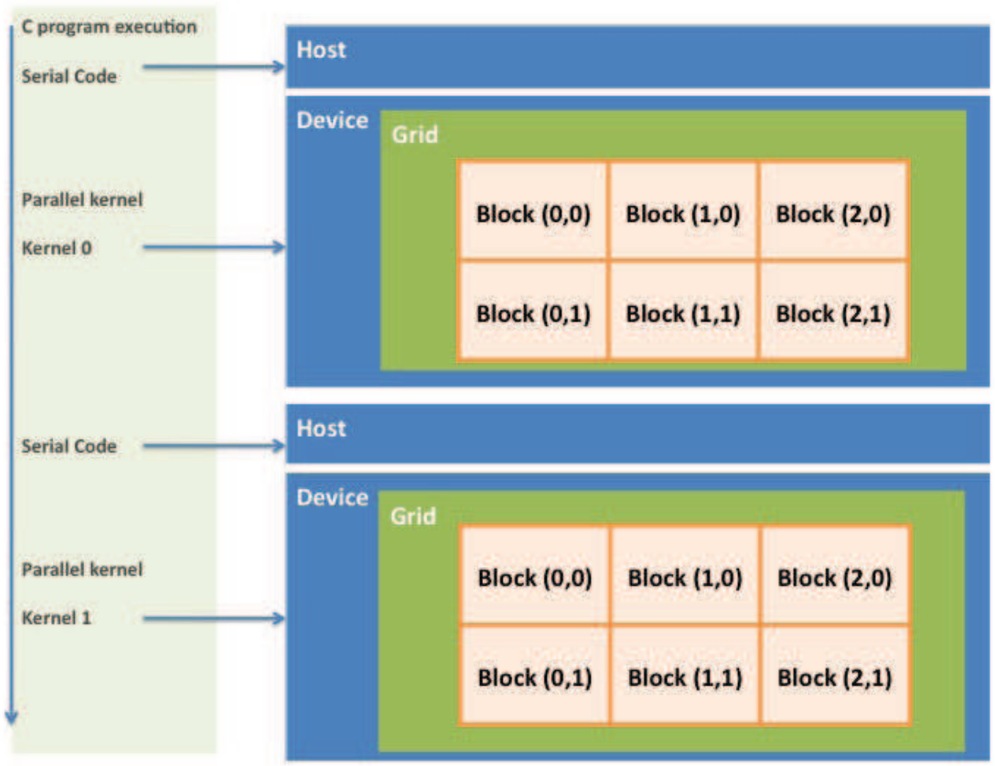


Fig. 2. CUDA C program structure

3.1 Data structures, kernel functions and GPU-operations

In order to process the algorithm in parallel, one must inform the CUDA platform the number of parallel copies of the Kernel functions to be performed. These copies are also known as parallel blocks and are divided into a number of execution threads.

The structures defined by grids can be split into blocks in two dimensions. The blocks are divided in threads that can be structured from 1 to 3 dimensions. As a consequence, the kernel functions can be easily instantiated (see Fig. 2). In case of a kernel function be invoked

by the CPU, it will run in separated threads within the corresponding block. For each thread that executes a kernel function there is a thread identifier that allows one to access the threads within the kernel through two built-in variables *threadIdx* and *blockIdx*. The size of data to be processed or the number processors available in the system are used to define the number of thread blocks in a grid. The GPU architecture and its number of processors will define the maximum number of threads in a block. On the current GPUs, a thread block may contain up to 1024 threads. For this chapter, the simulations were made with GPUs that supports up to 512 threads. Table 1 presents the used configuration for grids, blocks and thread for each kernel function. Another important concept in CUDA architecture is related to Warp, which refers to 32 threads grouped to get executed in lockstep, *i.e.* each thread in a warp executes the same instruction on different data Sanders & Kandrot (2010). In this chapter, as already mentioned, the data processing is performed directly in the memories.

Type of Kernel Functions	Configurations		
	Blocks	Threads	Grids
Setting positions, movement operators	2	512	(512, 2)
Fitness and weights evaluations, feeding operator	1	36	(36, 1)

Table 1. Grids, Blocks and Threads per blocks structures and dimension sizes

CUDA defines different types of functions. A *Host function* can only be called and executed by the CPU. kernel functions are called only by the CPU and executed by the device (GPU). For these functions, the qualifier *__global__* must be used to allow one to access the functions outside the device. The qualifier *__device__* declares which kernel function can be executed in the device and which ones can only be invoked from the device NVIDIA (2010b).

The FSS pseudocode shown in algorithm 1 depicts which functions can be parallelized in GPUs.

4. Synchronous and asynchronous GPU-based Fish School Search

4.1 The synchronous FSS

The synchronous FSS must be implemented carefully with barriers to prevent any race condition that could generate wrong results. These barriers, indicated by *__syncthreads()* function in CUDA, guarantee the correctness but it comes with a caveat. Since the fish need to wait for all others, all these barriers harm the performance.

In the Synchronous version the synchronization barriers were inserted after the following functions (see algorithm 1): fitness evaluations, update new position, calculate fish weights, calculate barycenter and update steps values.

4.2 The asynchronous FSS

In general, an iteration of the asynchronous approach is faster than the synchronous one due to the absence of some synchronization barriers. However, the results will be probably worse, since the information acquired is not necessarily the current best.

Here, we propose two different approaches for Asynchronous FSS. The first one, called *Asynchronous - Version A*, presents some points in the code with synchronization barriers. In

Algorithm 0.1: Pseudocode of the Synchronous FSS

```
begin
    Declaration and allocation of memory variables for the Kernel operations;
    w ⟵ number_of_simulations;
    for i ⟵ 1 to Numberofiterations do
        Start timer event;
        /* calling Kernel functions                                    */
        Initialization_Positions;
        Initialization_Fish_Weights;
        Fitness_evaluation;
        Synchronization_Barrier;
        while number_of_iterations_not_achieved do
            /* Individual operator                                    */
            Calculate_New_Individual_Movement;
            Calculate_Fitness_for_New_Position;
            Update_New_Position;
            Synchronization_Barrier;
            Calculate_Fitness_Diference;
            /* Feeding operator                                       */
            Calculate_Fish_Weights;
            Synchronization_Barrier;
            Calculate_Weights_Difference;
            /* Collective Instinctive operator                        */
            Calculate_Instinctive_Movement;
            Update_New_Position;
            Synchronization_Barrier;
            /* Collective Volitive operator                           */
            Calculate_Barycentre;
            Synchronization_Barrier;
            Calculate_Volitive_Movement;
            Fitness_Evaluation;
            Synchronization_Barrier;
            /* Updating steps                                         */
            Update_Individual_Step;
            Update_Volitive_Step;
            Synchronization_Barrier;
        end
        /* Copy Kernel values to Host                                 */
        Copy_Kernel_Fitness_Value_To_Host;
        Stop timer event;
        Compute_Running_Time;
    end
    Free_memory_variables;
end
```

this case, were have maintained the synchronization barriers only in the functions used to update the positions and evaluate the barycenter. The pseudocode of the *Asynchronous FSS - Version A* is shown in algorithm 2. In the second approach, called *Asynchronous - Version B*, all the synchronization barriers were removed from the code in order to have a full asynchronous version. The pseudocode of the *Asynchronous FSS - Version B* is shown in algorithm 3.

Algorithm 0.2: Pseudocode of the Asynchronous FSS - Version A

```
begin
    Declaration and allocation of memory variables for the Kernel operations;
    w ⟵ number_of_simulations;
    for i ⟵ 1 to Numberofiterations do
        Start timer event;
        /* calling Kernel functions                                    */
        Initialization_Positions;
        Initialization_Fish_Weights;
        Fitness_evaluation;
        while number_of_iterations_not_achieved do
            /* Individual operator                                    */
            Calculate_New_Individual_Movement;
            Calculate_Fitness_for_New_Position;
            Update_New_Position;
            Synchronization_Barrier;
            Calculate_Fitness_Diference;
            /* Feeding operator                                       */
            Calculate_Fish_Weights;
            Calculate_Weights_Difference;
            /* Collective Instinctive operator                        */
            Calculate_Instinctive_Movement;
            Update_New_Position;
            Synchronization_Barrier;
            /* Collective Volitive operator                           */
            Calculate_Barycentre;
            Synchronization_Barrier;
            Calculate_Volitive_Movement;
            Fitness_Evaluation;
            /* Updating steps                                         */
            Update_Individual_Step;
            Update_Volitive_Step;
        end
        /* Copy Kernel values to Host                                 */
        Copy_Kernel_Fitness_Value_To_Host;
        Stop timer event;
        Compute_Running_Time;
    end
    Free_memory_variables;
end
```

Algorithm 0.3: Pseudocode of the Asynchronous FSS - Version B

```
begin
    Declaration and allocation of memory variables for the Kernel operations;
    w ⟵ number_of_simulations;
    for i ⟵ 1 to Numberofiterations do
        Start timer event;
        /* calling Kernel functions                                    */
        Initialization_Positions;
        Initialization_Fish_Weights;
        Fitness_evaluation;
        while number_of_iterations_not_achieved do
            /* Individual operator                                    */
            Calculate_New_Individual_Movement;
            Calculate_Fitness_for_New_Position;
            Update_New_Position;
            Calculate_Fitness_Diference;
            /* Feeding operator                                       */
            Calculate_Fish_Weights;
            Calculate_Weights_Difference;
            /* Collective Instinctive operator                        */
            Calculate_Instinctive_Movement;
            Update_New_Position;
            /* Collective Volitive operator                           */
            Calculate_Barycentre;
            Calculate_Volitive_Movement;
            Fitness_Evaluation;
            /* Updating steps                                         */
            Update_Individual_Step;
            Update_Volitive_Step;
        end
        /* Copy Kernel values to Host                                 */
        Copy_Kernel_Fitness_Value_To_Host;
        Stop timer event;
        Compute_Running_Time;
    end
    Free_memory_variables;
end
```

5. Simulation setup and results

The FSS versions detailed in the previous section were implemented on the CUDA Platform. In this section we present the simulations executed in order to evaluate the fitness performance of these different approaches. We also focused on the analysis of the execution time.

In order to calculate the execution time for each simulation we have used the CUDA event API, which handles the time of creation and destruction events and also records the time of the events with the timestamp format NVIDIA (2010b).

We used a 1296 MHz GeForce GTX 280 with 240 Processing Cores to run the GPU-based FSS algorithms. All simulations were performed using 30 fish and we run 50 trial to evaluate the average fitness. All schools were randomly initialized in an area far from the optimal solution in every dimension. This allows a fair convergence analysis between the algorithms. All the random numbers needed by the FSS algorithm running on GPU were generated by a normal distribution using the proposal depicted in Bastos-Filho et al. (2010).

In all these experiments we have used a combination of individual and volitive steps at both initial and final limits with a percentage of the function search space Bastos-Filho et al. (2009). Table 2 presents the used parameters for the steps (individual and volitive). Three

Operator	Step value	
	Initial	Final
Individual	$10\%(2 * \max(searchspace))$	$1\%(2 * max(searchspace))$
Volitive	$10\%(Step_{ind,initial})$	$10\%(Step_{ind,final})$

Table 2. Initial and Final values for Individual and Volitive steps.

benchmark functions were used to employ the simulations and are described in equations (6) to (8). All the functions are used for minimization problems. The Rosenbrock function is a simple uni-modal problems. The Rastrigin and the Griewank functions are highly complex 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 is hard to achieve. The function is defined as follows:

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

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

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

Equation (8) shows the Griewank function, which is a multimodal function:

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

All simulations were carried out in 30 dimensions. Table 3 presents the search space boundaries, the initialization range in the search space and the optima values. Figures 3, 4 and 5 present the fitness convergence along 10,000 iterations for the Rosenbrock, Rastrigin and

Function	Parameters		
	Search Space	Initialization	Optima
Rosenbrock	$-30 \leq \vec{x}_i \leq 30$	$15 \leq \vec{x}_i \leq 30$	1.0^D
Rastrigin	$-5.12 \leq \vec{x}_i \leq 5.12$	$2.56 \leq \vec{x}_i \leq 5.12$	0.0^D
Griewank	$-600 \leq \vec{x}_i \leq 600$	$300 \leq \vec{x}_i \leq 600$	0.0^D

Table 3. Function used: search space, initialization range and optima.

Griewank, respectively. Tables 4, 5 and 6 present the average value of the fitness and standard deviation at the 10,000 iteration for the Rosenbrock, Rastrigin and Griewank, respectively.

Analyzing the convergence of the fitness values, the results for the parallel FSS versions on the GPU demonstrate that there are no reduction on the quality performance over the original version running on the CPU. Furthermore, there is a slight improvement in the quality of the values found for the Rastrigin function (see Fig. 4), specially for the asynchronous FSS version B. It might occurs because the outdated data generated by the race condition can avoid premature convergence to local minima in multimodal problems.

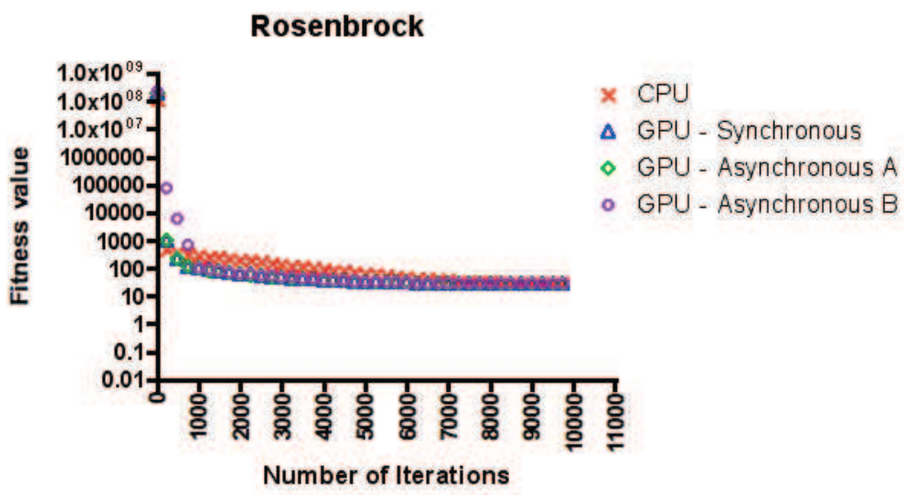


Fig. 3. Rosenbrock's fitness convergence as a function of the number of iterations.

Algorithm Version	Fitness	
	Average	Std Dev
CPU	28.91	0.02
GPU Synchronous	28.91	0.01
GPU Asynchronous A	28.91	0.01
GPU Asynchronous B	28.90	0.02

Table 4. The Average Value and Standard Deviation of the Fitness value at the 10,000 iteration for Rosenbrock function.

Tables 7, 8 and 9 present the average value and the standard deviation of the execution time and the speedup for the Rosenbrock, Rastrigin and Griewank functions, respectively.

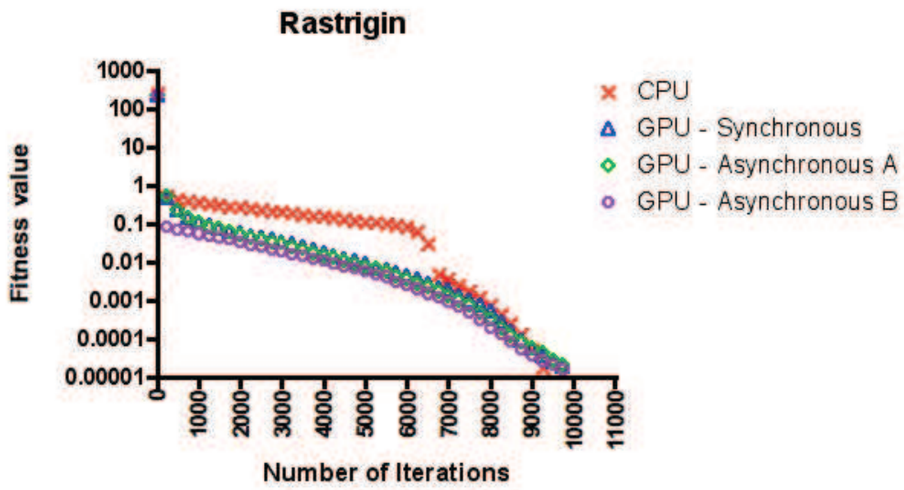


Fig. 4. Rastrigin's fitness convergence as a function of the number of iterations.

Algorithm Version	Fitness	
	Average	Std Dev
CPU	2.88e-07	5.30e-08
GPU Synchronous	1.81e-07	4.66e-08
GPU Asynchronous A	2.00e-07	2.16e-08
GPU Asynchronous B	1.57e-07	1.63e-08

Table 5. The Average Value and Standard Deviation of the Fitness value at the 10,000 iteration for Rastrigin function.

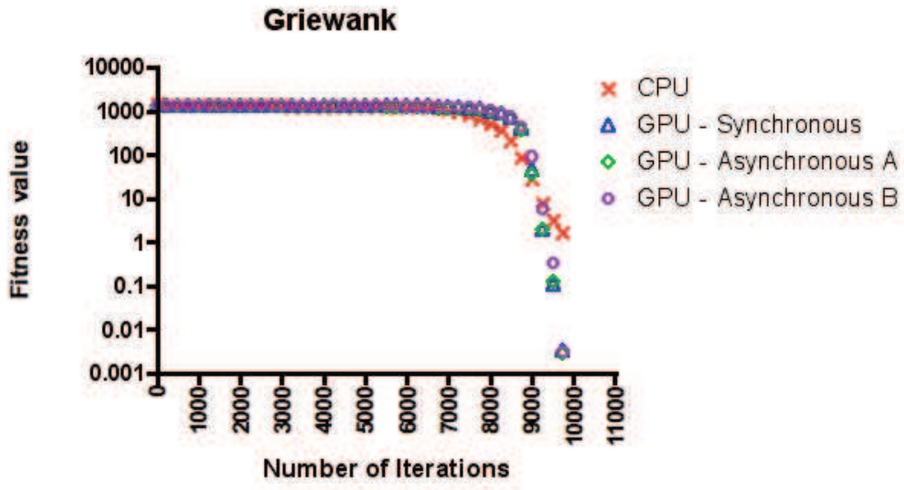


Fig. 5. Griewank's fitness convergence as a function of the number of iterations.

According to these results, all FSS implementations based on the GPU achieved a time performance around 6 times better than the CPU version.

Algorithm Version	Fitness	
	Average	Std Dev
CPU	1.67	0.74
GPU Synchronous	3.27e-05	3.05e-05
GPU Asynchronous A	2.91e-05	1.87e-05
GPU Asynchronous B	3.08e-05	1.54e-05

Table 6. The Average Value and Standard Deviation of the Fitness value at the 10,000 iteration for Griewank function.

Algorithm Version	Time (ms)		
	Average	Std Dev	Speedup
CPU	6691.08	1020.97	–
GPU Synchronous	2046.14	61.53	3.27
GPU Asynchronous A	1569.36	9.29	4.26
GPU Asynchronous B	1566.81	7.13	4.27

Table 7. The Average Value and Standard Deviation of the Execution Time and Speedup Analysis for Rosenbrock function.

Algorithm Version	Time (ms)		
	Average	Std Dev	Speedup
CPU	9603.55	656.48	–
GPU Synchronous	2003.58	2.75	4.79
GPU Asynchronous A	1567.08	2.11	6.13
GPU Asynchronous B	1568.53	4.40	6.13

Table 8. The Average Value and Standard Deviation of the Execution Time and Speedup Analysis for Rastrigin function.

Algorithm Version	Time (ms)		
	Average	Std Dev	Speedup
CPU	10528.43	301.97	–
GPU Synchronous	1796.07	2.77	5.86
GPU Asynchronous A	1792.43	2.88	5.87
GPU Asynchronous B	1569.36	9.30	6.71

Table 9. The Average Value and Standard Deviation of the Execution Time and Speedup Analysis for Griewank function.

6. Conclusion

In this chapter, we presented a parallelized version of the Fish School Search (FSS) algorithm for graphics hardware acceleration platforms. We observed a significant reduction of the computing execution time when compared to the original FSS version running on CPU. This swarm intelligence technique proved to be very well adapted to solving some optimization problems in a parallel manner. The computation time was significantly reduced and better optimization results were obtained more quickly with GPU parallel computing. Since FSS can be easily parallelized, we demonstrated that by implementing FSS in GPU one can benefit from the distributed float point processing capacity. We obtained a speedup around 6 for a cheap GPU-card. We expect to have a higher performance in more sophisticated GPU-based architectures. Since the Asynchronous version achieved the same fitness performance with a lower processing time, we recommend this option. As future work, one can investigate the performance in more complex problems and assess the scalability in more advanced platforms.

7. Acknowledgments

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

8. References

Bastos-Filho, C. J. A., Lima Neto, F. B., Lins, A. J. C. C., Nascimento, A. I. S. & Lima, M. P. (2008). A novel search algorithm based on fish school behavior, *Systems, Man and Cybernetics, 2008. SMC 2008. IEEE International Conference on*, pp. 2646 –2651.

Bastos-Filho, C. J. A., Lima-Neto, F. B., Sousa, M. F. C., Pontes, M. R. & Madeiro, S. S. (2009). On the influence of the swimming operators in the fish school search algorithm, *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, pp. 5012 –5017.

Bastos-Filho, C. J. A., Oliveira Junior, M. A. C., Nascimento, D. N. O. & Ramos, A. D. (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.

Holland, J. H. (1992). *Adaptation in natural and artificial systems*, MIT Press, Cambridge, MA, USA.

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*

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

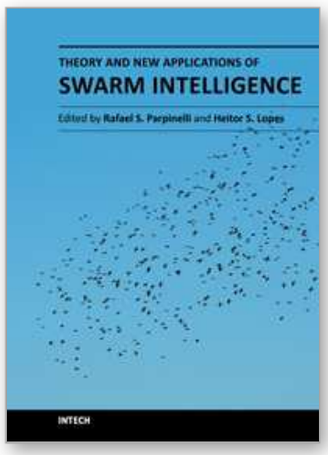
NVIDIA (2010a). *CUDA C Best Practices Guide 3.2*.

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

Sanders, J. & Kandrot, E. (2010). *CUDA By Example: an introduction to general-purpose GPU programming*.

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). Parallel ant colony for nonlinear function optimization with graphics hardware acceleration, *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, pp. 1803 –1808.



Theory and New Applications of Swarm Intelligence
Edited by Dr. Rafael Parpinelli

ISBN 978-953-51-0364-6
Hard cover, 194 pages
Publisher InTech
Published online 16, March, 2012
Published in print edition March, 2012

The field of research that studies the emergent collective intelligence of self-organized and decentralized simple agents is referred to as Swarm Intelligence. It is based on social behavior that can be observed in nature, such as flocks of birds, fish schools and bee hives, where a number of individuals with limited capabilities are able to come to intelligent solutions for complex problems. The computer science community have already learned about the importance of emergent behaviors for complex problem solving. Hence, this book presents some recent advances on Swarm Intelligence, specially on new swarm-based optimization methods and hybrid algorithms for several applications. The content of this book allows the reader to know more both theoretical and technical aspects and applications of Swarm Intelligence.

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen