GPU-based Island Model for Evolutionary Algorithms

Thé Van Luong, Nouredine Melab, El-Ghazali Talbi

GPU-based Island Model for Evolutionary Algorithms

Thé Van Luong
INRIA Dolphin Project
Opac LIFL CNRS
40 avenue Halley
Villeneuve d'Ascq, France
The-Van.Luong@inria.fr

Nouredine Melab
INRIA Dolphin Project
Opac LIFL CNRS
40 avenue Halley
Villeneuve d'Ascq, France
Nouredine.Melab@lifl.fr

El-Ghazali Talbi
INRIA Dolphin Project
Opac LIFL CNRS
40 avenue Halley
Villeneuve d'Ascq, France
El-Ghazali.Talbi@lifl.fr

ABSTRACT

The island model for evolutionary algorithms allows to delay the global convergence of the evolution process and encourage diversity. However, solving large size and time-intensive combinatorial optimization problems with the island model requires a large amount of computational resources. GPU computing is recently revealed as a powerful way to harness these resources. In this paper, we focus on the parallel island model on GPU. We address its re-design, implementation, and associated issues related to the GPU execution context. The preliminary results demonstrate the effectiveness of the proposed approaches and their capabilities to fully exploit the GPU architecture.

Categories and Subject Descriptors

G.1.6 [**Mathematics of Computing**]: Numerical Analysis—*Optimization*

General Terms

Algorithms, Design

Keywords

Island model, parallel, GPU

1. INTRODUCTION

In the last decades, evolutionary algorithms (EAs) have been successfully applied to solve optimization problems. Different models have been proposed in the literature for the design and implementation of EAs [11]. The island model (IM) allows to provide more effective, diversified and robust solutions by delaying the global convergence. Nevertheless, as the mechanism may be CPU time-consuming, it is not often fully exploited in practice. Indeed, for a significant number of islands, experiments with the IM for EAs are often stopped before the convergence is reached. That is the reason why, in designing IM for EAs, there is often a compromise between the number of islands to use and the computational complexity to explore it. As a consequence, only the use of parallelism allows to deal with such highly computationally expensive process.

Nowadays, GPU computing is recognized as a powerful way to achieve high-performance on long-running scientific applications [10]. With the arrival of the general-purpose computation on graphics processings units (GPGPU), EAs on GPU have generated a growing interest. Many works on GPU have been proposed: genetic algorithm [12], genetic programming [2] and evolutionary programming [13].

Designing IMs for EAs for solving real-world optimization problems are good challenges for GPU computing. Regarding previous works on IMs on GPU, in [3, 5], CPU and GPU simultaneously evaluate one separate local population with basic two-directional exchange mechanisms. This approach is used to reduce the overall run time beyond what is achieved by tackling the evaluation stage alone. However, due to many data transfers between the CPU and the GPU and a non-optimal task distribution, the performance of such approach might be limited. On the other hand, during the GPU competition of GECCO 2009, one submission entry [9] presents some technical details of an island model entirely hard-coded on GPU with a ring-like topology. Nevertheless, the evolutionary operators implemented on GPU are only specific to a particular class of problems, and the validity of the experiments works only on an assumption of a small size of problems.

To the best of our knowledge, GPU-based IMs have never been deeply studied and no general models can be outlined from previous works. Indeed, the design of GPU-based EAs often involves the cost of a sometimes painful apprenticeship of parallelization techniques and GPU computing technologies. In order to free from such burden those who are unfamiliar with those advanced features, general models must be provided for replicability.

In this paper, we contribute with the entire re-design of the IM on GPU by specifying its different parameters taking into account the particular features related to both the EA process and the GPU computing. More exactly, we propose three different general schemes for building efficient IMs for EAs on GPU. The first scheme combines the IM with the parallel evaluation of the population on GPU which is the simplest way to broach an EA on GPU and has been conducted in many related works of EAs on GPU. In the second scheme, the evolutionary process of each island is fully distributed on GPU. The third scheme is an extension of the second one using fast on-chip shared memory.

GECCO'10, July 7–11, 2010, Portland, Oregon, USA.

In comparison with the first scheme, the advantage of the full distribution of the evolutionary process on GPU is to reduce the CPU/GPU memory copy latency. However, its achievement is particularly challenging since many relative issues must be faced such as the threads synchronization, the optimization of data transfer between the different memories, the capacity constraints of these memories, etc. Therefore, in the fully distributed schemes, we propose to re-visit the existing parameters of the IM (number of islands, migration topology and frequency, number of migrants, emigration/immigration policy) according to the GPU characteristics.

The remainder of the paper is organized as follows: Section 2 highlights the parallel IM for EAs and the GPU architecture. In Section 3, the three proposed schemes for the design of the IM on GPU are presented. The issues related to the full distribution of the evolutionary process on GPU are discussed in Section 4. To validate the approaches presented in this paper, Section 5 reports the performance results obtained for a continuous optimization problem. Finally, a discussion and some conclusions of this work are drawn in Section 6.

2. PARALLEL ISLAND MODEL AND GPU COMPUTING

2.1 Parallel Island Model

For non-trivial problems, executing the reproductive cycle of a simple EA on long individuals and/or large populations requires high computational resources. Consequently, a variety of algorithmic issues are being studied to design efficient EAs. These issues usually consist of defining new operators, hybrid algorithms, parallel models, and so on. Parallelism arises naturally when dealing with populations, since each of the individuals belonging to it is an independent unit. Due to this, the performance of population-based algorithms is specially improved when running in parallel.

Basically, three major parallel models for EAs can be distinguished: the parallel evaluation of the population, the distributed evaluation of a single solution and the island (a)synchronous cooperative model. A review of the parallel paradigms is proposed in [1, 11] for EAs.

In this paper, we focus on the Island (a)synchronous cooperative model (see Fig. 1). In this latter, different EAs are simultaneously deployed to cooperate for computing better and robust solutions. They exchange in a(n) (a)synchronous way genetic stuff to diversify the search. The objective is to allow the delay of the global convergence, especially when the EAs are heterogeneous regarding the variation operators. The migration of individuals follows a policy defined by few parameters:

- **Exchange topology**: The topology specifies for each island its neighbors with respect to the migration process. In other words, it indicates for each island the other islands to which it may send its emigrants, and the ones from which it may receive immigrants). Different well-known topologies are proposed such as ring, mesh, torus, hypercube, etc.
- **Number of emigrants**: This parameter is often defined either as a percentage of the population size or as a fixed number of individuals.

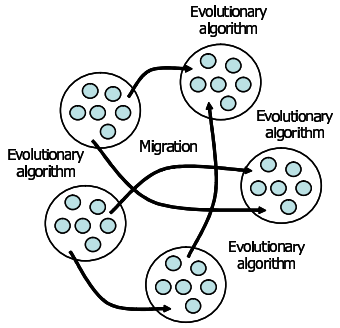

Figure 1: The cooperative island model of evolutionary algorithms.

- **Emigrants selection policy**: The selection policy indicates in a deterministic or stochastic way how to select emigrant individuals from the source island. Different selection policies are defined in the literature: roulette wheel, ranking, stochastic or deterministic tournaments, uniform sampling, etc.
- **Replacement/integration policy**: Symmetrically, the replacement policy defines how to integrate the immigrant individuals in the population. Different replacement strategies may be used including EP-like stochastic replacement, tournaments, elitist and pure random replacements.
- **Migration decision criterion**: Migration can be decided either periodically or according to a given criterion. Periodic decision making consists in performing the migration by each EA at a fixed or user defined frequency.

2.2 GPU Computing

With the recent advances in parallel computing particularly based on GPU computing, the IM has to be re-visited from the design and implementation points of view. Driven by the demand for high-definition 3D graphics on personal computers, GPUs have evolved into a highly parallel, multithreaded and many-core environment. Indeed, this architecture provides tremendous computational horsepower and very high memory bandwidth compared to traditional CPUs. Since more transistors are devoted to data processing rather than data caching and flow control, GPU is specialized for compute-intensive and highly parallel computation. A complete review of GPU architecture can be found in [10].

In GPGPU, the CPU is considered as a host and the GPU is used as a device coprocessor. This way, each GPU has its own memory and processing elements that are separate from the host computer. Data must be transferred between the memory space of the host and the memory of GPU during the execution of programs. In EAs, the types of data which are manipulated are the population representation.

Memory transfer from CPU to GPU device memory is a synchronous operation which is time consuming. In the case of EAs, memory copying operations from CPU to GPU are essentially the duplication operations of the population of solutions. Bus bandwidth and latency between CPU and GPU can significantly decrease the performance of the search, so data transfers must be minimized.

Each processor device on GPU supports the single program multiple data (SPMD) model, i.e. multiple processors simultaneously execute the same program on different data. For achieving this, the concept of kernel is defined. The kernel is a function callable from the host and executed on the specified device by several processors in parallel.

This kernel handling is dependent of the general-purpose language. For instance, CUDA or OpenCL are parallel computing environments which provide an application programming interface. These toolkits introduce a model of threads which provides an easy abstraction for single-instruction and multiple-data (SIMD) architecture. A thread on GPU can be seen as an element of the data to be processed. Compared to CPU threads, GPU threads are lightweight. That means that changing the context between two threads is not a costly operation.

Regarding their spatial organization, threads are organized within so called thread blocks. A kernel is executed by multiple equally threaded blocks. Blocks can be organized into a one-dimensional or two-dimensional grid of thread blocks, and threads inside a block are grouped in a similar way. All the threads belonging to the same thread block will be assigned as a group to a single multiprocessor, while different thread blocks can be assigned to different multiprocessors. As a consequence, regarding EAs, a natural mapping is to associate one GPU thread with an individual of the population.

3. SCHEMES FOR THE ISLAND MODEL ON GPU ARCHITECTURES

In this section, the focus is on the re-design of the IM presented in Section 2.1. To accomplish this, we propose three different schemes allowing a clear separation of the GPU memory hierarchical management concepts.

3.1 Scheme of the Parallel Evaluation of the Population on GPU

A first natural scheme for the design and the implementation of the IM is based on a combination with the parallel evaluation of the population model on GPU. Indeed, in general, evaluating a fitness function for every individual is frequently the most costly operation of the EA. Therefore, in this scheme, task distribution is clearly defined: the CPU manages the whole sequential EA process for each island and the GPU is dedicated only to the parallel evaluation of solutions. Fig. 2 shows the principle of the parallel evaluation of one island on GPU.

First, the CPU sends a certain number of individuals to be evaluated to the GPU via the global memory and then these solutions are processed on GPU. Regarding the kernel thread organization, as quoted above, a GPU is organized following the SPMD model, meaning that multiple autonomous processors simultaneously execute the same program at independent points. Therefore, each GPU thread associated with one individual executes the same evaluation function kernel. Finally, results of the evaluation function are returned back to the host via the global memory.

This way, the GPU is used as a coprocessor in a synchronous manner. The time-consuming part i.e. the evaluation kernel is calculated by the GPU and the rest is handled by the CPU. However, depending on the size of the population, the main drawback of this scheme is that copying

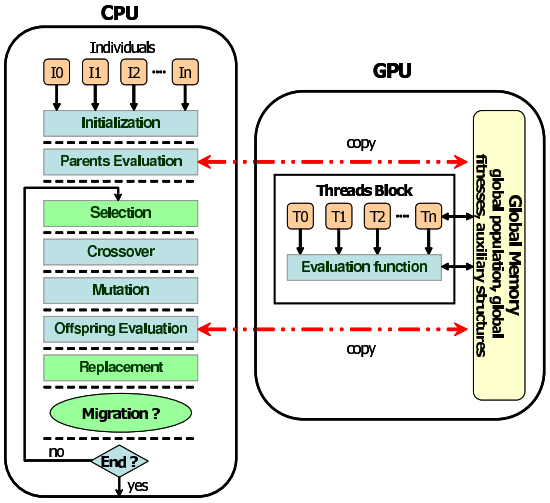

Figure 2: Scheme of the parallel evaluation of the population on GPU.

operations from CPU to GPU (i.e. population and fitnesses structures) can become frequent and thus can lead to a significant performance decrease.

The main goal of this scheme is to accelerate the search process and it does not alter the semantics of the algorithm. As a result, the migration policy between the islands remains unchanged in comparison with a traditional pure design on CPU. However, since the GPU is used as a cooperative coprocessor to evaluate all individuals in parallel, this scheme is intrinsically dedicated to the synchronous IM.

3.2 Schemes of the Fully Distributed Island Model on GPU

3.2.1 Fully Distributed Island Model on GPU

A second natural scheme is to parallelize the whole IM on GPU. This way, the main advantage of this approach is to minimize the data transfers between the host memory and the GPU. Fig. 3 gives an illustration of one particular island on GPU.

In this scheme, a natural representation is to associate one island with one threads block. One individual is represented by one thread and each standard genetic operator e.g. selection, crossover or mutation is separated by block barriers to ensure the synchronization between the threads.

Regarding the migration policy, communications are performed via the global memory which stores the global population. This way, each local island can communicate with any others according to the given topology.

One of the limitations to move the entire algorithm on GPU is the fact that a heterogeneous strategy cannot be easily applied in comparison with the previous scheme. Indeed, since multiprocessors on GPU are used according to the SPMD model, the same parameter configurations and the same different search components (e.g. mutation or crossover) between the islands must be used. Another drawback of this scheme concerns the maximal number of individuals per island since this latter is limited to the maximal number of threads per block (up to 512 or 1024 according to the GPU architecture). A natural idea to solve this limitation would be to associate one island with many

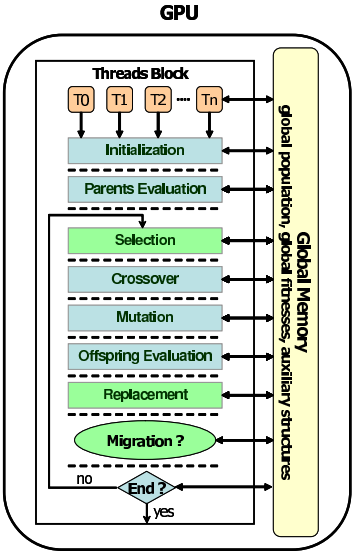

Figure 3: Scheme of the fully distributed island model on GPU.

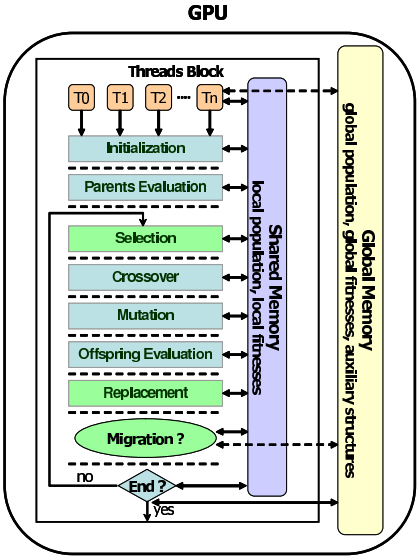

Figure 4: Scheme of the fully distributed island model on GPU using shared memory.

threads blocks. However, it cannot be easily achieved in practice since (1) threads work in an asynchronous manner; (2) threads synchronizations are local to a same block. Indeed, one can imagine a scenario in which a selection is made on two individuals of a different block where one of the two threads has not yet updated its associated fitness value (i.e. one of the two individuals has not yet been evaluated).

3.2.2 *Fully Distributed Island Model on GPU Using Shared Memory*

Regarding the kernel memory management, from a hardware point of view, graphics cards consist of multiprocessors, each with processing units, registers and on-chip memory. Accessing global memory incurs an additional 400 to 600 clock cycles of memory latency. As a consequence, since this memory is not cached and its access is slow, one needs to minimize accesses to global memory (read/write operations) and reuse data within the local multiprocessor memories.

To accomplish this, the shared memory is a fast on-chip memory located on the multiprocessors and shared by threads of each thread block. This memory can be considered as a user-managed cache which can provide great speedups by conserving bandwidth to main memory [7]. Furthermore, since the shared memory is local to each threads block, it provides a way for threads to communicate within the same block.

Therefore, a last natural scheme is to associate each island to a threads block on GPU with the use of the fast shared memory. An illustration for one particular island of this scheme is shown in Fig. 4. This scheme is similar to the previous one except the fact that local island populations and their associated fitnesses are stored in the on-chip shared memory. This way, each individual (thread) on each island (block) performs the process evolution (initialization, evaluation, etc.) via the shared memory.

Regarding migration between islands, since migration requires an inter-island communication, copying operations from each local population (shared memory) to the global population (global memory) have to be considered.

Even if this scheme can improve the efficiency of the IM, it presents a major limitation: since each multiprocessor has a limited capacity of shared memory (16KB), only small problem instances can be dealt with. Indeed, the amount of allocated shared memory per block depends on both the local population size and the problem instance size. Therefore, a trade-off must be considered between the number of threads per block and the size of the handled problem.

4. ISSUES RELATED TO THE FULLY DISTRIBUTED SCHEMES

We have presented three different schemes of the IM on GPU. The first scheme introduces the parallel evaluation of the global population and does not present any difficult issue. However, the two other schemes which involve the full parallelization of the IM on GPU presents many challenging problems related to the GPU execution model. In this section, we propose to re-visit the parameters of IMs on GPU.

- **Exchange topology**: Threads block can be identified using a one-dimensional or two-dimensional index. Therefore, two natural island topologies can be defined on GPU: ring (one-dimensional threads block) and 2D toroidal grid (two-dimensional threads block). Finding an efficient mapping between the GPU threads spatial organization and sophisticate island topologies is a challenging issue not addressed in this paper.

- **Number of emigrants**: This parameter does not present any important issue except the fact that if the number of emigrants is too important, accesses to the global memory will be more frequent leading to a small performance decrease.

- **Emigrants selection policy**: The selection policy is similar to the traditional selection in EAs. Basically, most of selection operators (e.g. tournament or roulette wheel) operate on unsorted data structures. Therefore, their implementation on GPU is similar to

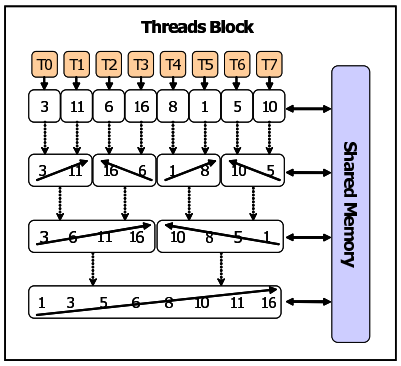

Figure 5: Principle of the bitonic sort.

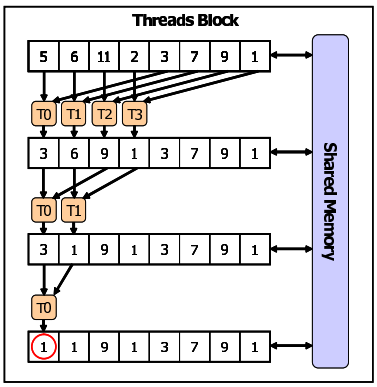

Figure 6: Reduction for finding the minimum value.

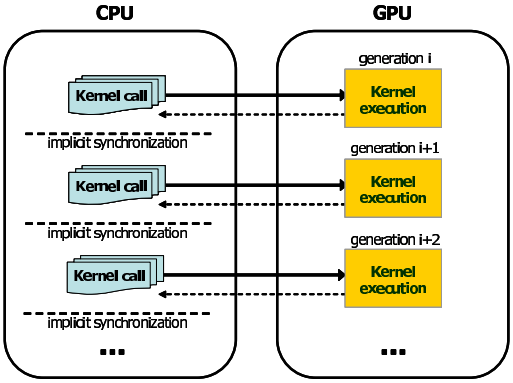

Figure 7: Implicit synchronization of threads to perform the synchronous island model.

a CPU one. However, some selection operators such as rank-based selections operate on sorted data structures. As a consequence, a significant issue occurs since sorting on the GPU is not as straightforward as on the CPU because the GPU is effectively a highly parallel SIMD architecture.

To deal with this issue, one efficient sort on GPU is the bitonic sort (see Fig. 5). Basically, this sort is based on (1) the concept of the bitonic sequence i.e. the composition of two subsequences, one monotonically non-decreasing and the other monotonically non-increasing; (2) merging procedures to create a new bitonic sequence. Despite its complexity of $O(log_2^2 n)$, this sort has a high degree of parallelism. Thereby, it has been stated as one of the fastest sort on GPU for a relatively small number of elements [4]. As a consequence, this sort is particularly well-adapted for the IM since the size of each island is also relatively small.

- **Replacement/integration policy**: Symmetrically, the previous emigrants selection strategies on GPU can be similarly applied for the replacement/integration policy. However, in practice, pure elitist replacements can be also considered in which the worst local individuals are replaced. Since read/write operations on memory are performed in an asynchronous manner, finding the appropriate minimal/maximal fitnesses of each island is not easy. Indeed, traditional parallel techniques such as semaphores which imply the global synchronization (via atomic operations) of thousand of threads can drastically lead to a performance decrease. To deal with this issue, adaptation of parallel reduction techniques [7] for each thread block must be considered. This way, by using local synchronizations between threads in a same block, one can find the minimum/maximum of a given array since threads operate at different memory addresses (see Fig. 6).

- **Migration decision criterion**: Whatever the migration decision criterion (whether periodically or according to a particular criterion), since the order of the execution of threads is undefined, the evolution process of each island can be in different state (e.g. different generation). As a result, the migration on GPU between the different islands is intrinsically done in an asynchronous manner. Performing a synchronous IM on GPU represents a significant issue. Indeed, performing a global synchronization on threads (e.g. by implementing semaphores) would lead to a great loss of performance. To deal with this, implicit synchronizations between the CPU and the GPU must be considered by associating one kernel execution with one generation of the evolutionary process. This way, when the execution of one generation in all islands on GPU is finished, the hand is returned back to the CPU ensuring an implicit global synchronization (see Fig. 7). However, performing such synchronous mechanisms leads to some unavoidable slight decrease of the performance due to the implicit synchronization and the kernel calls overhead.

Whether for a synchronous or an asynchronous model, regarding the scheme using shared memory, emigrants must be copied into the global memory to ensure their visibility between the different islands. Fig. 8 summarizes the main concepts presented above through an example of an IM on GPU using the shared memory.

Regarding the generic components of the traditional evolutionary process, the previous techniques can be applied to the selection and the replacement in a similar way. Regarding the components which are dependent of the problem (i.e. initialization, evaluation function and variation operators), they do not present specific issues related to the GPU kernel execution. From an implementation point of view, the only focus is the management of random numbers. To do this, efficient techniques are provided in many books such as [8]

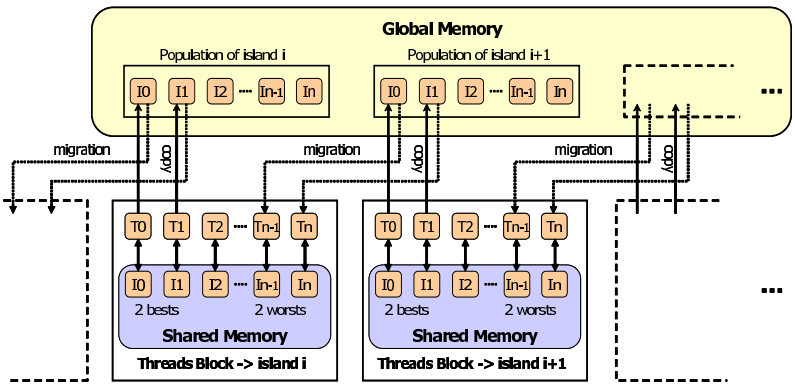

Figure 8: Migration between islands on GPU using a ring topology.

to implement random generators on GPU (e.g. Gaussian or Mersenne Twister generators).

5. EXPERIMENTATION

To validate the different presented approaches of this paper, the Weierstrass-Mandelbrot functions have been considered on GPU. These functions belong to the class of continuous optimization problems. According to [6], Weierstrass-Mandelbrot functions are defined as follows:

$$W_{b,h}(x) = \sum_{i=1}^{\infty} b^{-ih} sin(b^i x) \quad with\, b > 1\, and\, 0 < h < 1 \quad (1)$$

The parameter h has an impact on the irregularity ("noisy" local perturbation of limited amplitude) and these functions possess many local optima. Since the calculation of an infinite sum of sines is unpractical, a number of iterations is used to compute an approximation of the function (instead of ∞). The higher this value is, the more the evaluation of an individual takes time. Another parameter that can be tuned for these functions is the dimension of the problem.

For this problem, different versions of the IM have been implemented on GPU using CUDA. The used configuration for the experiments is an Intel Xeon 8 cores 2.4 Ghz with a NVIDIA GTX 280 card (30 multiprocessors). For each experiment, different implementations are given: a standalone mono-core IM on CPU (CPU), the synchronous IM using the parallel evaluation of the population on GPU (CPU+GPU), the asynchronous fully distributed IM on GPU (GPU), the synchronous version (SGPU) and their associated version using shared memory (GPUShared and SGPUShared).

Regarding the evaluation function of the problem (i.e. the Weierstrass function (1)), the domain definition has been set to $-1 \leq x_k \leq 1$, h has been fixed to 0.25 and the number of iterations to compute the approximation to 100 (instead of ∞). The complexity of the evaluation function is quadratic. The operators used in the evolutionary process for the implementations are the following: the crossover is a standard two-point crossover (crossover rate fixed to 80%) which generates two offspring, the mutation consists in changing randomly one gene with a real value taken in $[-1; 1]$ (with a mutation rate of 30%), the selection is a deterministic tournament (tournament size fixed to the block size divided by four), the replacement is a $(\mu+\lambda)$ replacement and the number of generations has been fixed to 100. Regarding the migration policy, a ring topology has been chosen, a deterministic tournament has been performed for both emigrants selection and migration replacement (tournament size fixed to the block size divided by four), the migration rate is equal to the number of local individuals divided by four and the migration frequency is set to 10 generations.

5.1 Measures in Terms of Efficiency

The objective of the following experiments is to evaluate the impact of the GPU-based implementations in terms of efficiency. Only execution times (in seconds) and acceleration factors (compared to a mono-core CPU without GPU) are reported. The average time has been measured in seconds for 50 runs and the associated standard deviation is not represented since its value is small for every measured values.

The first experiment consists in varying the dimension of the Weierstrass function. Results are reported in Table 1(a). As long as the size of the dimension increases, each GPU version gives some outstanding accelerations compared to a CPU version (up to $\times$ 1757 for the GPUShared version). The use of the shared memory provides a way to accelerate efficiently the search process even if the GPU version is already impressive. However, due to its limited capacity, bigger instances such as a dimension of 11 cannot be handled in any shared memory versions.

Regarding the fully distributed synchronous versions, since implicit synchronizations are performed, a certain decrease of the speed-up (from $\times$63 to $\times$293 for the SGPU version) can be observed in comparison with their associated asynchronous versions. Nevertheless, the acceleration factors are still impressive. For the scheme of the parallel evaluation of the population (CPU+GPU version), the speed-ups are less important even if they remain significant (from $\times$25 to $\times$170). This may be explained by the important number of data transfers between the CPU and the GPU.

A conclusion of the first experiment indicates that changing the dimension of the problem leads to better speed-ups for each GPU version. It seems obvious that the increase of the number of iterations to compute an approximation of the Weierstrass function would lead to similar results. A second experiment consists in varying the number of islands in order to measure the efficiency and the scalability of our approaches. Results of this experiment are reported in Table 1(b). Regarding each fully distributed version, for a small number of islands (i.e. one or two islands), the acceleration factor is significant but not impressive (from $\times$7

Table 1: Table (a) consists in varying the instance size of the problem: the number of individuals per island is fixed to 128 and the global population to 8192 (64 islands). Table (b) consists in varying the number of islands: the dimension of the problem is fixed to 2 and the number of individuals per island to 128.

(a)	CPU	CPU+GPU		GPU		GPUShared		SGPU		SGPUShared	
dimension	**time**	**time**	**speed-up**	**time**	**speed-up**	**time**	**speed-up**	**time**	**speed-up**	**time**	**speed-up**
1	23	0.92	×25	0.16	×143	0.04	×845	0.36	×63	0.06	×375
2	43	0.94	×46	0.16	×268	0.03	×1150	0.36	×119	0.08	×511
3	64	0.95	×67	0.17	×375	0.05	×1365	0.38	×167	0.11	×607
4	85	0.97	×87	0.21	×403	0.06	×1442	0.47	×179	0.13	×641
5	105	1.00	×105	0.22	×479	0.07	×1579	0.50	×213	0.15	×702
6	127	1.02	×125	0.24	×519	0.08	×1639	0.55	×231	0.17	×728
7	148	1.04	×142	0.28	×529	0.09	×1659	0.63	×235	0.20	×737
8	168	1.09	×154	0.30	×554	0.10	×1684	0.68	×246	0.23	×748
9	190	1.19	×159	0.31	×610	0.11	×1736	0.70	×271	0.25	×772
10	211	1.28	×165	0.33	×639	0.12	×1757	0.74	×284	0.27	×781
11	231	1.36	×170	0.35	×666	–	–	0.79	×293	–	–

(b)	CPU	CPU+GPU		GPU		GPUShared		SGPU		SGPUShared	
islands	**time**	**time**	**speed-up**	**time**	**speed-up**	**time**	**speed-up**	**time**	**speed-up**	**time**	**speed-up**
1	3	0.10	×33	0.20	×17	0.12	×27	0.45	×7	0.27	×12
2	7	0.12	×55	0.20	×33	0.13	×51	0.45	×15	0.29	×23
4	13	0.15	×89	0.20	×65	0.13	×104	0.45	×29	0.29	×46
8	26	0.19	×139	0.20	×132	0.13	×207	0.45	×59	0.29	×92
16	53	0.34	×154	0.21	×256	0.13	×403	0.46	×114	0.29	×179
32	106	0.66	×160	0.26	×406	0.13	×828	0.59	×180	0.29	×368
64	211	1.28	×165	0.33	×644	0.14	×1560	0.74	×286	0.30	×693
128	422	2.68	×158	0.45	×939	0.26	×1596	1.01	×417	0.60	×709
256	845	5.61	×151	0.69	×1222	0.50	×1677	1.56	×543	1.13	×746
512	1692	11.81	×143	1.24	×1365	1.00	×1691	2.79	×607	2.25	×752
1024	3382	25.72	×132	–	–	1.70	×1990	–	–	3.82	×885
2048	6781	53.23	×127	–	–	3.27	×2074	–	–	7.36	×922
4096	13585	143.71	×95	–	–	–	–	–	–	–	–

to ×51). This can be explained by the fact that since the global population is relatively small (less than 1024 threads), the number of threads per block is not enough to fully cover the memory access latency. However, the speed-up grows accordingly with the increase of the number of islands and remains impressive (up to ×2074 for GPUShared).

Regarding the CPU+GPU version, speed-ups for one or two islands are more important than for the fully distributed versions. Indeed, since only the evaluation of the population is distributed on GPU, fewer registers are allocated for each thread. As a result, this version benefits from a better occupancy of the multiprocessors for a small number of islands. GPU keeps accelerating the process with the islands increase until reaching a peak performance of ×165 for 64 islands. After, the acceleration factor decreases with the augmentation of the number of islands. Indeed, for each parallel evaluation of the population, the amount of data transfers is proportional to the number of individuals (e.g. 524288 threads for 4096 islands). Thus, from a certain number of islands, the elapsed time for copying operations becomes significant leading certainly to a decrease of the performance.

Regarding the scalability of the fully distributed versions, from a certain number of islands, the GPU failed to execute the program because of the hardware register limitation. For instance, for a number of 1024 islands (131072 threads), the SGPU implementation could not be executed. In the GPUShared and the SGPUShared versions, since the shared memory is used conducting to fewer registers, this limit is reached for a larger number of 4096 islands. For the CPU+GPU version, it provides a higher scalability since far fewer registers are allocated (only the evaluation kernel is executed on GPU).

5.2 Measures in Terms of Effectiveness

After measuring the efficiency of our approaches, the last experiment consists in evaluating the quality of the obtained solutions. To accomplish this in a fair manner, only GPU synchronous versions must be considered since it ensures that all of the synchronous models produce similar results. The parameters of this experiment are the same used before. Fig. 9 reports the evolution of the average of best fitness values during the first minute (50 executions). In agreement with the previous obtained results, the quality of the solutions differs accordingly to the employed scheme. Indeed, for instance, the best results are obtained with the fully distributed IM on GPU using shared memory. Precise measurement of the first seconds is not reported on the figure, but the evolution of the fitness for each GPU version decreases drastically during the first second. The evolution of the fitness keeps decreasing with time but with a rather slow convergence. It is possible that due to a high migration rate (25%), the diversity in the populations is lost too fast and therefore it becomes hard to find better results.

6. CONCLUSION AND DISCUSSION

Parallel metaheuristics such as the IM allow to improve the effectiveness and robustness in optimization problems. Their exploitation for solving real-world problems is possible only by using a great computational power. High-performance computing based on GPU accelerators is recently revealed as an efficient way to use the huge amount

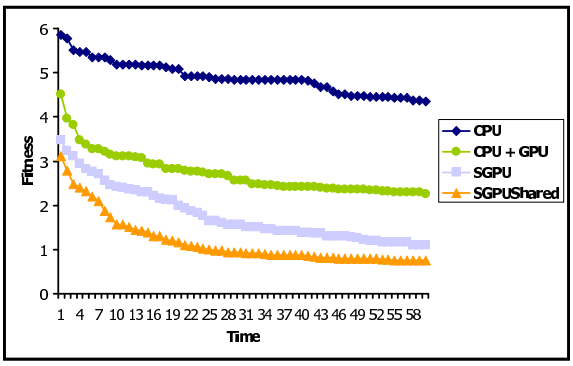

Figure 9: Measures in terms of effectiveness of the different island models. The dimension of the problem is fixed to 10, the global population is set to 8192 and the number of individuals per island to 128.

of resources at disposal. However, the exploitation of the IM is not trivial and many issues related to the context execution of this architecture have to be considered.

In this paper, we have particularly focused on the design of three efficient IM schemes to the hierarchical GPU. The designed and implemented approaches have been experimentally validated on a continuous optimization problem. The experiments indicate that GPU computing allows not only to speed up the search process, but also to exploit parallelism to improve the quality of the obtained solutions. In the first scheme, the IM is combined with the parallel evaluation of the population on GPU. From an implementation point of view, this approach is the most generic since only the evaluation kernel is considered but the performance of this scheme is limited due to the data transfers between the CPU and GPU. To deal with this issue, the two other schemes operate on the full distribution of the search process on GPU. Applying such mechanism allows to improve drastically the performance. However, these schemes could present some limitations due to memory constraints with some problems that could be more demanding in terms of resources.

Nevertheless, we strongly believe that the two last schemes could be easily extended to large scale continuous optimization problems. Indeed, it would be interesting to test our approaches on the seven-function benchmark test suite for the CEC-2008 special session and competition on large scale global optimization.

Another perspective is to test the different schemes to combinatorial optimization problems such as the quadratic assignment problem. However, unlike the continuous functions, data inputs of the combinatorial optimization problem must be considered. Indeed, by working with such structures, non-aligned memory accesses imply many memory transactions leading to a global loss of performance. As a result, dealing with such optimization problems is not straightforward since it involves optimizing data accesses which includes the appropriate use of the various GPU memory spaces.

7. REFERENCES

[1] E. Alba and M. Tomassini. Parallelism and evolutionary algorithms. *IEEE Trans. Evolutionary Computation*, 6(5):443–462, 2002.

[2] W. Banzhaf, S. Harding, W. B. Langdon, and G. Wilson. Accelerating genetic programming through graphics processing units. In *Genetic Programming Theory and Practice VI*, pages 1–19. Springer, 2009.

[3] O. Garnica, J. Risco-Martin, J. Hidalgo, and J. Lanchares. Speeding-up resolution of deceptive problems on a parallel gpu-cpu architecture. In *Parallel Architectures and Bioinspired Algorithms*, 2008.

[4] A. Greß and G. Zachmann. Gpu-abisort: optimal parallel sorting on stream architectures. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, 2006.

[5] T. E. Lewis and G. D. Magoulas. Strategies to minimise the total run time of cyclic graph based genetic programming with GPUs. In *GECCO '09*, pages 1379–1386, Montreal, 8-12 July 2009. ACM.

[6] E. Lutton and J. L. Véhel. Holder functions and deception of genetic algorithms. *IEEE Trans. Evolutionary Computation*, 2(2):56–71, 1998.

[7] NVIDIA. *CUDA Programming Guide Version 2.3*, 2010.

[8] NVIDIA. *GPU Gems 3. Chapter 37: Efficient Random Number Generation and Application Using CUDA*, 2010.

[9] P. Pospichal and J. Jaros. Gpu-based acceleration of the genetic algorithm. GECCO competition, 2009.

[10] S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S.-Z. Ueng, S. S. Baghsorkhi, and W. W. Hwu. Program optimization carving for gpu computing. *J. Parallel Distribributed Computing*, 68(10):1389–1401, 2008.

[11] E.-G. Talbi. *Metaheuristics: From design to implementation*. Wiley, 2009.

[12] S. Tsutsui and N. Fujimoto. Solving quadratic assignment problems by genetic algorithms with gpu computation: a case study. In *GECCO '09*, pages 2523–2530, New York, NY, USA, 2009. ACM.

[13] T.-T. Wong and M. L. Wong. Parallel evolutionary algorithms on consumer-level graphics processing unit. In *Parallel Evolutionary Computations*, pages 133–155. Springer, 2006.