Soft Comput
DOI 10.1007/s00500-011-0718-z

FOCUS

EASEA: specification and execution of evolutionary algorithms on GPGPU

Ogier Maitre · Frédéric Krüger · Stéphane Querry · Nicolas Lachiche · Pierre Collet

© Springer-Verlag 2011

Abstract EASEA is a framework designed to help non-expert programmers to optimize their problems by evolutionary computation. It allows to generate code targeted for standard CPU architectures, GPGPU-equipped machines as well as distributed memory clusters. In this paper, EASEA is presented by its underlying algorithms and by some example problems. Achievable speedups are also shown onto different NVIDIA GPGPUs cards for different optimization algorithm families.

Keywords Evolutionary computation · Evolution strategy · Genetic algorithm · Genetic programming · Memetic algorithm · CUDA · GPGPUs

1 Introduction

Artificial evolution algorithms raise great interest in their ability to find solutions that are not necessarily optimal, but adequate for complex problems. The power of these algorithms depends on various factors, including the available computing power. Increasing it is interesting, as it would allow to explore the usually huge search spaces more widely and deeply, for better results.

Recently, GPGPUs (general-purpose graphical processing units) appeared on the processor market. These computing units are former 3D rendering processors, now able to handle generic computation. They are massively parallel, containing hundreds of cores and optimized memory for texture processing.

Owing to their instrinsic parallelism, loop structure and predictability, evolutionary algorithms are good candidates to be ported onto such an architecture. Indeed, lots of works take this direction. However, the user has to be both expert in evolutionary algorithms and GPGPU programming to enjoy these benefits.

EASEA (EAsy Specification of Evolutionary Algorithm) is a software platform dedicated to the implementation of evolutionary algorithms, that can now port different types of evolutionary algorithms on GPGPUs and clusters of (potentially heterogeneous) machines using an island model. Typically, EASEA can be used to implement: Genetic Algorithm and Evolution Strategy, Memetic and Genetic Programming algorithms.

In this paper, some implementations of standard evolutionary algorithms on GPGPUs are presented, as well as how the EASEA platform allows to specify and run those evolutionary algorithms on one or several GPGPU cards without worrying about GPGPU architectural specificities. These approaches were tested and presented in different papers (Krüger et al. 2010; Maitre et al. 2009a, b, 2010a). The current paper summarizes these works and for the first time presents the complete scheme.

Novelties include the algorithm that is used to distribute the population over the hundreds of cores of GPGPUs, evaluations on new GPGPU cards comparisons with older cards and a short presentation of the island model

O. Maitre (✉) · F. Krüger · S. Querry · N. Lachiche · P. Collet
Pôle API, Bd Sébastien Brant BP 10413,
67412 Illkirch Cedex, France
e-mail: ogier.maitre@unistra.fr

F. Krüger
e-mail: frederic.kruger@unistra.fr

S. Querry
e-mail: stephane.querry@unistra.fr

N. Lachiche
e-mail: nicolas.lachiche@unistra.fr

P. Collet
e-mail: pierre.collet@unistra.fr

Published online: 27 May 2011

implemented into EASEA. This software is available on SourceForge or on the dedicated EASEA platform website.[1]

The paper starts with the presentation of CUDA concepts that will be used throughout the paper. Then, the underlying principles that are presented have been used to port evolutionary algorithms on GPGPU cards. Aftewards, we present how EASEA allows the end user to specify and execute any of those evolutionary algorithms on one or several machines, using GPGPUs or not. Finally, a set of problems is presented that contains artificial and real-world problems.

2 Some hardware considerations

CUDA (Compute Unified Device Architecture) was the first framework created by NVIDIA to perform generic computations onto NVIDIA Graphic Processing Units. Hence, CUDA can be considered as a low-level framework, which guarantees satisfactory performance, due to a medium level of abstraction, which is why it was chosen over other solutions to implement EASEA. The main drawback of this approach is probably the lack of portability between graphic card brands, imposed by the use of this proprietary framework.

To start with, here are some useful notions about NVIDIA GPGPUs and CUDA, which will be used later in the paper. A GPGPU is a processor containing a large number of cores (typically several hundreds). These computing units are grouped into Single Instruction Multiple Data (SIMD) core bundles, which execute the same instruction at the same time, applied to different data. In CUDA, these units are called Multi-Processors (MP).

The number of cores of an MP depends on the GPGPU version, but an MP still performs a single instruction the same number of times. On older cards (before *Fermi* architecture), an MP embeds eight cores, each of which performs the same instructions four times in different threads, before it receives the next one. Recent Fermi cards actually have 32 cores that execute each instruction only once. In both cases, an instruction is executed by 32 different threads, on possibly different data and such a group of threads, which are executed at the same time, is called a warp.

The GPGPU processor accesses its own memory, called global memory. This memory space can be accessed by the host processor by the means of the host memory, due to direct memory access (DMA) transfers. In addition, each MP embeds a few kilobytes of memory (16 or 48 KB, depending on the card), that are only accessible by the MP cores. This extremely fast memory is called shared memory, and replaces what should be a cache to access the global memory.

Without cache, a memory access is very slow (in the order of several hundreds of cycles) and due to its limited size, shared memory cannot always compensate for the lack of cache. A second mechanism complementing the shared memory is a hardware thread scheduling mechanism. It is comparable to the Intel HyperThreading mechanism that can be found on Pentium IV or Core i7, but at a much higher level. These units switch frozen warps, for example on memory reads, with other warps, which are ready for execution. Scheduling units can choose among a set of w warps ($w = 24$ *or* 32), which can come from four different blocks of threads per MP. The blocks are groups of threads that are alloted to one MP, e.g. the minimal pool of threads to schedule. The size of this scheduling pool can explain why GPGPUs are not efficient with only a few threads, as shown in (Maitre et al. 2009a, b).

The structure of the cores and the availability of hardware scheduling units give GPGPUs a SPMD (Single Program Multiple Data)/SIMD structure, as a warp among w runs in parallel on each multi-processor. Indeed, the threads in a warp must necessarily execute the same instruction, because they are executed by the cores of the same MP at the same time. However, each warp can execute a different instruction without causing any divergence among the cores, because only one warp is executed at a time. In addition, the MPs can execute different instructions on their respective cores at the same time.

Therefore, the complex architecture of GPGPUs means that they are not purely SIMD. It is possible to execute different codes at the same time (with restrictions) once the architecture is well understood.

One last detail worth to be noted is the availability of two units that compute trigonometric function approximations per MP. These units are inherited from the original use of GPGPUs, which is 3D rendering. They allow fast approximation of heavy computation functions, such as sine, cosine, etc, and are called special function unit, or SFU.

3 Implementation of evolutionary algorithms on GPGPUs

There are many different evolutionary algorithms. In Collet and Schoenauer (2003), De Jong (2008), the authors propose a unified view of these variants by highlighting the common principles shared by each approach. Differences are sometimes formal, as between evolution strategies (ES) and genetic algorithms (GA), but sometimes deeper, as between GAs and genetic programming (GP). Therefore, it

[1] http://www.lsiit.u-strasbg.fr/easea

was decided to use a common structure for the ES and the GA, but to implement a different algorithm for GP.

In both cases, it was chosen to parallelize the evaluation step only, because this phase is considered to be the most time consuming in the whole algorithm. It means that the parallel and sequential versions of an algorithm can be completely identical by keeping the rest of the algorithm identical.

This idea holds for algorithms that use complex and costly evaluation function. It does not fit in the case of algorithms with a light weight evaluation function, where the evaluation time is shorter than the transfer time of an individual onto the GPGPU memory. However, the need to parallelize this kind of algorithm is also to question.

An counter example could be multi-objective algorithms. Indeed, determining the individuals ranking is an expensive function $O(nm^2)$, where n is the population size, m the number of objectives for the main multi-objective algorithms (Zitzler et al. 2002; Deb et al. 2002). Parallelization of the evaluation function remains attractive, especially if the ranking phase is also executed on GPGPU (Wong 2009) or if one is using an algorithm of lower complexity (Sharma and Collet 2010a). Both ideas can also be combined (Sharma and Collet 2010b).

3.1 Evolution strategies and genetic algorithms

Evolutionary algorithms are greedy in computing power, by nature, as they attempt to find good solutions by evaluating hundreds of individuals over hundreds of generations. However, computing power is mainly consumed by the evaluation function, as the evolutionary part of the algorithm is usually relatively lightweight and executed only once every generation, whereas the evaluation function is run for each individual evaluation, for every generation.

EASEA allows to use GPGPUs for ES and GA, due to the algorithm presented in this section.

3.1.1 Related works

Various attempts at porting evolutionary algorithms onto GPGPU have been made. All these implementations were done early in the maturity cycle of these cards (Li et al. 2007; Fok et al. 2007; Yu et al. 2005). Indeed, these implementations use 3D programming paradigms, by the means of an evolutionary algorithm coded into a shader. These solutions were also trying to port a full algorithm onto the card, which generates a complex code, written using graphical programming languages. These constraints result in disappointing performances, although it has to be noted that the tests were performed on old cards. Their systems still possess the advantage of being portable between different types of cards (from different manufacturers), relying on standard 3D programming (shader programming language). To port the algorithms using these paradigms, the evolutionary algorithms are modified to satisfy programming constraints. The comparison between these algorithms and more standard ones becomes thus difficult. The last point is that these implementations are not publicly available and are understood only by 3D programming specialists.

3.1.2 EASEA implementation

For the EASEA platform, it was decided to use GPUs for population evaluations only. This choice is based on three main considerations:

1. porting the evaluation step only results in a simpler and more comparable code than porting the whole algorithm,
2. the evolutionary engine is considered to be very fast to execute, compared to population evaluation,
3. keeping the evolutionary engine on the host machine allows to access individuals in the GPU in a read-only manner: the new population needs to be transferred to the GPU at each generation, but only fitness values need to be copied back.

The population needs to be distributed into blocks, in order to be assigned to multi-processors on the card. This step is important, as it ensures a good load balancing on each GPGPU multi-processor and an efficient scheduling. However, this distribution needs to be compatible with hardware constraints, that is, every block should not use more registers than are available on an MP. The algorithm implemented in EASEA assigns to a block the maximum number of threads allowed per MP, considering both register and scheduling limitations. This ensures that the size of the thread pool to be scheduled is maximal.

Input: N: PopulationSize, w: WarpSize, M: Number of MPs, s: max number of schedulable tasks, e: max number of tasks
Output: b: number of blocks, t: number of threads per block
b=0;
repeat
  b := b+M;
  t := $\lceil \mathrm{Min}(s, e, n/(b \times M))/w \rceil \times w$;
until $b \times t > N \wedge e > t \wedge s > t$;

Algorithm 1: Distribution method

The method used to distribute the population for evaluation over every core in every MP is described in Algorithm 1.

Ideally, the number of blocks should be greater than or equal to M (number of MPs). Then, the minimum number

of threads per block is w (the minimum number of SIMD threads), or the thread limit (e, s) that is given by the maximum scheduling capacity (s) or the number of tasks that an MP can execute (e). The last limit is related to the task complexity (number of registers used by a thread and the number of available registers per MP). As the number of threads that are really executed is a multiple of w, the first multiple of w which is greater than this minimum is taken.

When a population of children is ready to be evaluated, it is copied in the GPGPU card memory. All the individuals are evaluated with the same call to the GPU evaluation function and results (fitnesses) are sent back to the host CPU, which contains the original individuals and manages the populations.

In the host memory, an EA population is mainly a sparse collection of objects representing individuals. An individual is composed of a genome and other fields, such as a fitness value and an "already evaluated" boolean variable. To evaluate an individual, only the genome is needed. This implementation groups the individuals in a contiguous buffer, which allows to transfer everything needed for evaluation in one single DMA transfer, instead of one transfer per individual. This allows to reduce overhead.

3.1.3 Evaluation step

The evaluation phase of the population on the GPGPU uses one thread per individual. This guarantees the independence of the evaluation and provides a number of tasks equal to the number of children produced per generation. Furthermore, in the general case, the exact same function is applied to every individual, which matches the SIMD model well.

The implementation presented here uses only the GPGPU global memory. This is justified because in the general case, it is not possible to guarantee that all the data necessary to evaluate an individual would fit in the small shared memory (even less all data needed to evaluate a block of individuals). Furthermore, using the shared memory accelerates execution only if the data reuse are important, otherwise the gain provided by the access speed would be wasted by the time spent in transferring the data to the shared memory, as it cannot be directly accessed from the host CPU. A two-step copy is necessary to transfer data from CPU memory to the global GPGPU memory and only then can it be copied to the shared memory. In special cases, this mechanism can be profitable, but as EASEA is designed to do generic evolutionary algorithms, shared memory is left unused.

3.2 Genetic programming

The evaluation of an individual in genetic programming is different. Indeed, where in GAs or ESs, the same evaluation function is used on different data, in GP, different evaluation functions (individuals) are evaluated on the same data (the learning set). If the same mechanism as the one described above was used, many divergences between SIMD cores would be observed, as the nodes of two different individuals are very likely to be different. However, there is a solution to use GPGPUs: it is possible to evaluate one single individual in SIMD parallelism on the different values of the training set.

3.2.1 Related works

Several attempts have already been made to port GP algorithms onto GPGPU cards.

To our knowledge, the first work of this kind was published in 2007 by Chitty (2007). This implementation evaluates a population of GP-compiled individuals. This method is implemented using the cg (C for Graphics) programming language. The author compares the performance of his algorithm onto an NVIDIA 6400GO card versus an Intel 1.7 GHz processor. On a first linear regression problem with 400 training cases, the implementation gets a speedup of 10× and on a second problem, the 11-way multiplexer with about 100,000 cases of fitness, a speedup of 30× .

Harding and Banzhaf published a first implementation of interpreted GP in 2007 (Harding and Banzhaf 2007). The authors use the GPGPU card to evaluate a single individual on every core. Using *.Net* and *MS Accelerator*, they apply their algorithm to a sextic regression problem. They evaluate randomly created trees on different numbers of learning cases. The obtained speedup for this test ranges between 0.04 and more than 7,000× . To achieve this high speedup, a large tree and a high number of fitness cases are needed (10k fitness cases and 65k nodes per individual). This method requires a large number of training cases to observe an interesting speedup and the very complex implementation makes it difficult to reproduce.

Another implementation of interpreted GP is done by Langdon and Banzhaf (2008). The authors use RapidMind to do a complete implementation of a GP population evaluation. Here, the population is dispatched on the cores and the interpretor computes all the operators contained in the function set for every node, picking up the interesting result, and discarding the others: if the function set consists of four operators (+, –, × , /), all individuals will execute the four operators. If individual #1 needs to execute a +, it will pick up the result of the + operator and discard the other results. This is equivalent to assuming the worst divergence case for each node and to using the GPGPU card as a fully SIMD processor (which it is not). For a function set of five operators, the authors evaluate the loss to be a factor 3, which is close to 5. By applying this

algorithm to a real problem, they get a speedup of 12× with a 8800GTX card versus a 2.2 GHz AMD processor.

Robilliard et al. (2008, 2009) present an implementation using the CUDA environment. This latest implementation tends to take into consideration the internal SPMD structure of NVIDIA cards, by conducting the evaluation of several individuals over several fitness cases at the same time. The authors apply their approach to the 11-way multiplexer and the sextic regression with a learning set of 2,500 points. They obtain a speedup of 80× for the population evaluation step. These experiments are performed on an NVIDIA 8800GTX compared with one core of an Intel 2.6 GHz CPU. Using CUDA, the authors manage to use an NVIDIA GPGPU card efficiently, but some hardware tools are left unused, in particular the MP scheduling capability.

3.2.2 EASEA implementation of GP

As with the implementation of Robilliard et al., the one presented here uses individuals represented in a flat RPN (Reverse Polish Notation). This avoids pointers and tree-shaped individuals, allowing them to be transferred onto the GPU card using a single copy. The lack of pointer makes the representation more compact and access to the operators/operands more straigthforward on the GPGPU side.

The EASEA implementation uses an interpreter that allows the evaluation of several individuals over several training cases at the same time on the same MP. This interpreter is described in more details in the next section.

There are different ways to handle a flat representation in the other parts of the algorithm. The simplest is to use trees as in Koza's tree GP (Koza 1992) and to flatten the trees before transferring them on the GPGPU. This solution is used by Robilliard et al. (2008) which proves its feasibility. This first method was chosen here for its simplicity and its resemblance to standard genetic programming algorithms.

However, there are other methods that would save the flattening phase. In Langdon and Banzhaf (2008), Robilliard et al. (2009), the authors directly use an RPN representation in the evolutionary algorithm. Other models of genetic programming algorithms use a flat representation directly, such as PushGP, FIFTH and Linear GP (Spector and Robinson 2002; Holladay et al. 2007; Brameier and Banzhaf 2007).

3.2.3 Evaluation step

In genetic programming, the fitness of an individual is generally a sum of errors obtained when comparing the values produced by the individual with a training set. The executions of individuals on these training cases are independent tasks. Only the sum of errors for all training cases requires a synchronization.

The implementation presented here is inspired from Robilliard's paper (Robilliard et al. 2008), with a difference, allowing to benefit from the hardware scheduling capability to overlap memory latencies.

If one wants to maximise the use of the hardware schedulers using the CUDA card in its full SIMD model, it is necessary to load $k/4$ tasks per MP (because an MP can load 4 different blocks as explained in 2), each task being the evaluation of the same individual on a different fitness case. Depending on the type of card used, k can be 768 or 1,024 on the current model. This is related to the number of warps that can be scheduled (768 tasks = 24 Warps and 1,024 tasks = 32 Warps). Considering the MP as a completly SIMD processor, it would be necessary to have 192 (768/4) or 256 (1,024/4) learning cases to maximize SIMD parallelism using the hardware scheduling.

But the NVIDIA documentation (NVIDIA 2008) asserts that a divergence in the execution path will result in a loss of performance *only if this divergence occurs within one warp*. Taking this into consideration allows to evaluate different individuals into one MP if the different individuals reside into different warps. Knowing that, it is possible to maximize scheduling capability with only 32 fitness cases by loading $k/(32 \times 4)$ individuals per MP, because 32 is the warp size. 32 is the minimal number of threads that need to execute the same instruction at the same moment, without causing any divergence.

Owing to the shared memory size limit, stacks are stored into global memory. Nevertheless, the implementation exhibits great speedups, using this high latency memory instead of the shared memory. For example, on a GT200 hardware, such as a GTX295 card, it is possible to schedule between 24 warps ($k = 768$). Without any register constraints, four blocks can be used on the same MP at the same time.

With a single warp per block implementation and 32 fitness cases, only 128 threads can be scheduled, but with a multi-individual implementation as is implemented in EASEA, using 6 individuals per block, the hardware scheduler is completely busy ($6\ Inds \times 32\ fit_cases \times 4\ blocks = 768\ tasks$).

Small discussion on the number of test cases: the presented approach allows to maximize scheduling ability with as few as 32 fitness cases, which none of the previously presented approaches could do. This is very important for GP, where many real-world problems do not come with many test cases (problems with less than 100 test cases are common when each value comes from an expensive experiment). Moreover, if 100 values are

available, they are not all useable for the learning set knowing that such problems with a small number of test cases are prone to overfitting (GP will learn to match the learning set rather than find a function that generalizes well). Good practice involves dividing the test cases into at least a learning set and a test set (to check for overfitting) and even better, a third evaluation set, to evaluate the obtained individuals on test cases that have not been involved in finding the solution, as described in Gagné et al (2006). Other methods are available, but if this three-set methodology is used, the 100 test cases shrink into only 33 learning cases.

Then, below 32 test cases, one could argue that the problem could be solved with other heuristics using more assumptions than a generic stochastic method like genetic programming.

3.3 Memetic algorithms

Memetic algorithms are also referred to as *hybrid algorithms*. They couple a global search algorithm (e.g. an evolutionary algorithm) with a local search algorithm that rapidly optimizes created children in a possibly deterministic way.

Memetic algorithms can be implemented using two different approaches that change the goal of the local optimization: lamarckism(Ong and Keane 2004) and baldwinism(Ackley and Littman 1992).

Lamarckism is an idea named after the French biologist Jean-Baptiste Lamarck. Also known as *soft inheritance* or *heritability of acquired characteristics*, it stipulates that an organism can pass on characteristics that it acquired during its lifetime to its offsprings.

In a lamarckian EA (which is a memetic algorithm), after a child is created from genetic operators on its parents, it is improved with a local search. The improved child will replace the initial child, hence passing on its improved characteristics/genome to the next generation.

The Baldwin effect refers to a theory presented in a book entitled *A New Factor in Evolution* written by James Mark Baldwin. According to that theory, one should evaluate an individual on its potential, and not only on its genome, allowing for the development of intelligence, for instance.

In a Baldwinian EA, an individual will, therefore, not be evaluated according to its genes, but rather according to its *potential*. This potential can be computed using the local optimization function. The fitness of the individual will be the fitness value found by the local search algorithm, *but the genome of the individual will not be modified* (this also allows to fight against premature convergence of the population towards similar individuals).

3.3.1 Related works

In 2008, Wong et al. use a memetic algorithm on a standard graphics card in Wong and Wong (2006). The authors get a speedup of more than 4×.

Munawar et al. (2009) present an implementation of a memetic algorithm using CUDA. The authors use this implementation to solve the MAXSat problem. They obtain a speedup of 25× on this specific implementation. But none of these implementations is designed to be generic.

3.3.2 Implementation of a generic memetic algorithm in EASEA

The implementation of the memetic algorithm on GPGPU is based on the standard algorithm with parallel evaluation on GPGPU presented in Sect. 1. The standard algorithm has been altered in a way as to include a local optimization function as well as the few necessary parameters, such as the number of search iteration the local optimization function will have to perform.

For the population to be locally optimized on GPGPU, it first needs to be transferred from CPU to GPGPU memory.

3.3.3 Evaluation step

Then, to be able to perform the local optimization without needing to change the genome of the original individuals, the population is uploaded onto the card twice. The local search algorithm works on the second buffer, by optimizing an individual copied from the original population. At the end of the local search on the GPU, the results are brought back to the CPU memory space. For a baldwinian EA, only the fitness values are copied back, as for a standard EA. In the case of a lamarckian EA, the optimized children population also needs to be transferred back to the CPU memory space to create the next generation.

3.4 Island model

The island model is a well known way to parallelize evolutionary algorithms (Alba and Tomassini 2002). It allows to obtain very interesting speedup (sometimes supra-linear), while being simple to implement. For instance, on a cluster of computers, every node maintains a subpopulation, which can be seen as an island. A migration mechanism is added and allows to periodically export some individuals to other nodes.

The island model thus allows to parallelize an evolutionary algorithm on a distributed memory machine. Exchanges between nodes are limited to the migration of individuals during the execution and do not put too much pressure on the communication network.

Because EASEA can automatically parallelize an evolutionary algorithm in an efficient way on GPGPU cards, adding the possibility to run an island model allows to exploit a set of machines containing GPU cards, such as a cluster of GPGPUs.

As described below, the implementation was designed to be totally asynchronous, which allows to use different algorithms on different machines, and yet have them exchange individuals, if they share the same genome. In fact, it is also possible to interconnect different platforms as EASEA can produce code for machines with or without one or several GPGPU cards.

As long as the different machines are using the same representation for the individuals, it is possible to create a cluster of heterogeneous machines, and even link several clusters together for more computing power.

In our University, a classroom with 20 machines each hosting a 1 Teraflop nVidia GTX275 pre-Fermi card is regularly used (during nights and week-ends) as a 5,000 cores 20 Teraflops cluster of machines, to which some machines of the laboratory with Fermi-technology Tesla cards and GTX480 cards are added for even more power, yielding all in all around 30 Teraflops, for around 8,000 cores.

Then, EASEA has been run on a 500,000 cores Petaflop machine with no observed scaling problems.

3.4.1 Implementation

EASEA implements the island model in a very flexible manner, as it uses UDP-based asynchronous communication, which allows a loose coupling between nodes. At the beginning of every generation, a node sends some individuals to other nodes depending on a probability set by the user (for a 24 h run, good results have been obtained by sending a round of individuals roughly every 5 s, i.e. with a probability of 0.01 per generation if it takes 5 s to run 100 generations).

Sent individuals are duplicated from the population. They are selected using usual EA selectors such as: the best individual, an individual selected using an n-ary tournament, a fitness proportionate selection (roulette-wheel), etc. It is possible to send several individuals to one or more nodes selected at random in a list of IP addresses given to the host in a file.

It is possible to not include all the machines in the IP list, meaning that several island models can be implemented within the same run.

Typically, a good setup consists in having the cluster of 20 machines find good individuals, with an algorithm set towards exploration and a migration scheme that does not always send the best individual of the population to preserve diversity. These 20 machines also include in their IP list the IP numbers of three other machines that host respectively one 480GTX card, two 480GTX cards and two Tesla C2050 cards that also periodically receive individuals from the cluster.

However, these three machines only have the two other fast machines in their IP list plus another slow one to save the best individuals, and their migration scheme always includes the best individual of their population for maximal exploitation of potentially good solutions that are sent to them by the cluster. Periodically, when their population has converged, one machine sends to the others an individual with a fitness equal to -1, which causes all three machines to restart with random individuals. The 20 machines of the cluster are not included in their IP list, otherwise their populations would be “polluted” by extremely good individuals found by the three exploitation machines, which would favour premature convergence.

A fourth machine with no GPU is included in the IP list of the three fast machines. This very slow machine also runs a genetic algorithm with however no hope of finding any good results. It does not restart when it receives an individual of fitness -1 (that it simply discards). Its purpose is only to collect the best individuals that it receives from the three fast exploitation machines. This machine has no IP list to send individuals to, so it keeps its good individuals for itself.

The distant Petaflop machine has the IP number of this slow local machine in its list of IPs, so whenever some time is alloted to the algorithm on the Petaflop machine, the best individuals automatically get sent to our laboratory.

All this is possible because communication takes place in non-connected UDP mode, with no synchronizations whatsoever. It is, therefore, also possible to lose messages, but this is a stochastic algorithm after all, so message losses can be considered as being part of the algorithm (and could be seen as a scheme to prevent premature convergence).

4 EASEA

The EASEA platform was initially designed to assist users in the creation of evolutionary algorithms (Collet et al. 2000). It is designed to produce an evolutionary algorithm from a problem description. This description is written in a C-like language that contains code for the genetic operators (crossover, mutation, initialization and evaluation) and the genome structure. From these functions, written into an *.ez* file, EASEA generates a complete evolutionary algorithm with potential parallelization of evaluation over GPGPUs, or over a cluster of heterogeneous machines, due to the embedded island model discussed above.

The generated complete evolutionary algorithm is user-readable. It can be used as-is, or could be used as a primer, to be manually extended by an expert programmer.

4.1 Genome

The first important point in the specification of an evolutionary algorithm is the definition of the genome structure. EASEA dedicates a specific section to this task. The user defines the elements present in its genome as basic C types and EASEA objects, which are defined from these same basic C types. EASEA also handles 1D arrays in genome definition and the use of pointers is possible as well. An example of EA genome using pointer and user defined class is given in code snippet 1 below. Implicitly, the individual has a boolean validity field and a real fitness field.

The underlying EASEA library also defines a type used by genetic programming manipulation functions, the GPNode. A tree GP individual will be composed of such basic elements. An example of basic GP genome definition is given in code snippet 1.

EASEA automatically generates functions for manipulating individuals based on their structure. These functions are a deep copy of an individual, a deep equality test of two individuals, creation, deep deletion, serialization and deserialization. The "deep" copy, equality and deletion operators are intended to manipulate recursive structures, such as trees (for GP) or linked lists. Copy and deletion are applied to all elements of the tree. For algorithms using GPGPUs, a transfer function of an individual is also generated. Yet, pointers are not supported in this particular case, except for genetic programming, where the individual is flattened as explained in Sect. 3.2.2.

Finally, a display function of an individual is also created automatically, but can be overridden by the user.

```
\User classes:
  Element      { int       Value;
                 Element  *pNext; }
  GenomeClass  { Element  *pList;
                 int       Size; }
\end
```

EZ source 1: Example of a complex genome structure with pointers, to implement individuals containing a linked list.

```
\User classes :
GenomeClass { GPNode* root; }
\end
```

EZ source 2: Example of a GP genome.

4.2 Initialization

Once the structure of the genome is defined, it becomes possible to provide the initialization function. This function is obviously called to fill the initial population. It is automatically called for each individual.

```
\GenomeClass::initialiser:
  Element *pElt;
  Genome.Size=0;
  Genome.pList=NULL;
  // creation of a linked list of SIZE elts
  for (int i=0;i<SIZE;i++){
    pElt=new Element;
    pElt->Value=Random(0,i);
    pElt->pNext=Genome.pList;
    Genome.pList=pElt;
    Genome.Size++;
  }
\end
```

EZ source 3: Example of an initializer attached to the previous genome example of EZ code snippet 1

As one can see in code snippet 2, the method "GenomeClass::Initialiser" uses the predefined `Genome` keyword, which is used as an instance of the genome of the individual to be initialised. EASEA defines a set of utility macros, random(n,m) being one of them. The EASEA code generator replaces this macro by a call to a MersenneTwister random generator (Matsumoto and Nishimura 1998).

For genetic programming, the EASEA library implements the conventional construction methods inspired from Koza's (1992), as can be seen in code snippet 2.

John Koza defines two types of construction methods. The first is called "Grow," that selects the operator of a node out of a function set. These functions can have an arity of zero (terminal node) or non-zero (internal node). It recursively builds the tree until its penultimate level. When the maximum depth (max) is reached, the method selects the operator from the terminal function set only. Thus, the tree does not exceed the maximum depth but does not necessarily reach it. The root tree cannot be a terminal node (minimal depth cannot be 1).

A second defined method is called "Full." It ensures that the trees are complete, meaning that all the leaves reach the maximum depth (max). This is ensured by choosing the operators of a node in the function set until the maximum depth is reached, in which case the node is chosen among the terminals.

The method "Ramped Half and Half" builds half of the population using the Grow method, and the other half using the Full method. In each case, the maximum tree depth is

increased from the minimum (INIT_TREE_DEPTH_MIN) to the maximum size (INIT_TREE_DEPTH_MAX).

Using this method, the initial population contains individuals with various shapes and sizes.

4.3 Crossover and mutation

As for the initialization, EASEA provides three predefined keywords: `parent1`, `parent2` and `child`, that refer to the two parents and the children to be created. By default, `child` is a copy of `parent1`.

```
\GenomeClass::initialiser : {
  Genome.root =
   RAMPED_H_H(INIT_TREE_DEPTH_MIN,
              INIT_TREE_DEPTH_MAX,
              GROW_FULL_RATIO, VAR_LEN,
              OPCODE_SIZE, opArity , OP_ERC);
}
\end
```

EZ source 4: Example of GP tree initialization.

The mutation function expects the user to apply mutations to the child's genome that can be accessed through the `Genome` predefined keyword. For statistic reasons, the number of applied mutations should be returned.

The crossover and mutation genetic operators are executed on the host CPU, so no GPGPU limitations apply here.

Examples of a crossover and mutation functions are given in code snippets 3 and 3.

```
\GenomeClass::crossover :
  for (int i=0; i<SIZE; i++) {
    // barycentric crossover
    float alpha = (float)random(0.,1.);
    child.x[i] = alpha*parent1.x[i]
                 + (1.-alpha)*parent2.x[i];
  }
\end
```

EZ source 5: A barycentric crossover.

4.4 Evaluation method

The evaluation function is defined in EASEA as in code snippet 4. This function takes an implicit argument, which is again the `Genome`. The evaluation function must return a real value, the individual fitness.

```
\GenomeClass::mutator : // Must return the number of mutations
  int NbMut=0;
  float pond = 1./sqrt((float)SIZE);
  for (int i=0; i<SIZE; i++)
  if (tossCoin(pMutPerGene)){
    Genome.sigma[i] = Genome.sigma[i] * exp(SIGMA*pond*(float)gauss
        ());
    Genome.sigma[i] = MIN(0.5,Genome.sigma[0]);
    Genome.sigma[i] = MAX(0.,Genome.sigma[0]);
    Genome.x[i] += Genome.sigma[i]*(float)gauss();
  }
  return NbMut;
\end
```

EZ source 6: Implementation of Schwefel's auto-adaptative mutation.

```
\GenomeClass::evaluator :
  return Weierstrass(Genome.x, SIZE);
\end
```

EZ source 7: Evaluation function.

The evaluation function can contain the code of the function, or can call other user-defined functions, as shown in the snippet. No restrictions apply to the function if it is designed to be executed on a CPU. However, if the evaluation function is intended to be run on a GPGPU card, the code of this function must comply with the requirements of such equipment.

These limitations depend on the version of the GPGPU which is used. They were strong for the first nVidia cards but they tend to reduce. The code which is now executable approximately matches what can be used onto a standard CPU.

For example, early versions were unable to call functions. Calls in a code were inlined at compile time, which obviously prevented recursive functions. This limitation is part of those which have been removed on the latest cards (Fermi generation). Similarly, dynamic allocations, which were forbidden, are now allowed.

However, GPU code remains different from CPU code on a few points. The card is an external computing accelerator which cannot access the main memory of the host processor. This point prohibits access to global variables and host machine devices.

Functions that participate to the individual fitness computation will be compiled by a dedicated compiler (nvcc) in order to be executed onto the GPGPU. It is, therefore, essential to prefix these functions by the keywords *__host__* *__device__* in the EASEA source file to indicate that they need to be compiled for both the CPU and the GPU, as shown in snippet 4. The keyword *__host__* is present to allow the same code to run onto a

processor, when the algorithm is used on a machine without GPGPU.

```
\User functions:
__device__ __host__ inline float Weierstrass(float *x, int n){
  float res = .0, b:2.0, h=.25;
  float val[SIZE];

  for (int i = 0 ; i<n ; i++) {
    val[i] = 0.;
    for (int k=0;k<ITER;k++)
      val[i] += pow(b,-k*h) * sin(pow(b,k)*x[i]);
    res += Abs(val[i]);
  }
  return (res);
}
\end
```

EZ source 8: A example of a GPU-compatible evaluation method.

As explained in the previous section, the code executed in the evaluation function is not necessarily suitable for execution on GPGPU (read parallelizable). However, the parallelism of evolutionary algorithms can compensate for this lack if the evaluation function has a high enough ratio *computation/memory access*. The parallel population fitness computation is sufficient in this case to take advantage of the power of GPGPU cards. The population size can allow the scheduler to manage some of the memory problems.

```
\GenomeClass::evaluator header:
\end

\GenomeClass::evaluator for each fc :
  float expected_value = OUTPUT;
ERROR=powf(expected_value-EVOLVED_VALUE,2);
\end

\GenomeClass::evaluator accumulator :
  return sqrtf(ERROR/NO_FITNESS_CASES);
\end
```

EZ source 9: A example of a GP evaluation method

Evaluation in genetic programming is a special case. Indeed, the population evaluation follows in all cases a strict scheme which is the execution of an individual on the whole training set. To allow this algorithm to be ported on GPGPU, following the pattern described in Sect. 1, EASEA defines several sections to make it more convenient. Code snippet 4 shows an example of code using these EASEA sections. Maitre et al (2010a, b) paper describes specifically the use of these sections.

The first part allows the user to define the initialization of the evaluation function. Typically this part can be used to initialize the error variable. The second part is the body of the evaluation loop. It is possible to use different keywords like: "output" which is the expected value for the current training cases and "EVOLVED_VALUE," the value returned by the tree execution. The last part is the end of the loop, where the user processes the result and returns the evaluation value (e.g. the fitness of the individual).

4.5 Utility sections

EASEA being designed to create custom evolutionary algorithms, different utility sections allow to add steps inside the classic pattern of an evolutionary algorithm.

Figure 1 shows the details of the optimization loop that is implemented by EASEA. The black boxes are the utility sections that allow to customize the algorithm. These boxes take an implicit parameter, which is an object including the algorithm itself. For instance, these sections allow to load data, apply post-processing on the results and modify the algorithm during the execution (change the mutation and crossover probabilities, depending on the number of generations for instance, ...).

4.6 Parameters

The end of an *.ez* file contains the classical evolutionary algorithm parameters, as well as options specific to their implementation into EASEA. This section allows the user to set the default settings of the algorithm as in code snippet 6. Many of these parameters can be modified on the command line.

Some parametes are used for the island algorithm, which are presented in code snippet 6. The first parameter enables or disables the island model, the second gives the path and name of a file containing IP addresses to which individuals can be sent. The last parameter sets the probability to send an individual to another node per generation.

5 Experiments/applications

The same *.ez* source file can be compiled for CPU or for parallelization on GPGPU cards, if the `-cuda` parameter is used on the command line, which helps in comparing both implementations. Three benchmarks will be used to show the obtained speedups, before a real-world problem is presented.

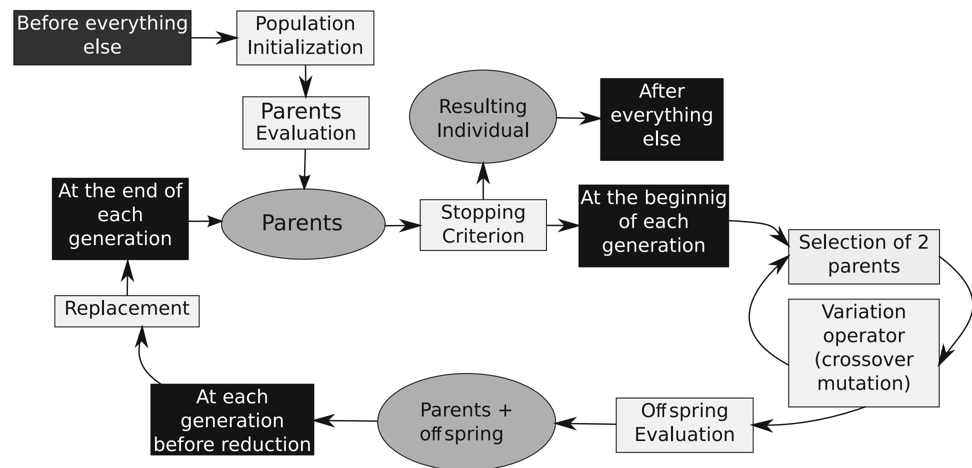

Fig. 1 Overview of the EASEA optimization loop

```
Number of generations : 100
Time limit: 0
Population size : 3840
Offspring size : 3840
Mutation probability : 1
Crossover probability : 1
Evaluator goal : minimise
Selection operator: Tournament 2.0
Surviving parents: 3840
Surviving offspring: 3840
Reduce parents operator: Tournament 2.0
Reduce offspring operator: Tournament 2
Final reduce operator: Tournament 2
Elitism: Strong
Elite: 5
```

EZ source 10: A subset of EASEA parameters

```
Remote island model: true
IP file: ip.txt
Migration Probability: .01
```

EZ source 11: Parameters for the island model.

5.1 Artificial problems

To demonstrate the ability to solve problems and to show the kind of speedups that can be obtained using EASEA, initial experiments were performed on well-known benchmarks and such problems allow to observe the behavior of an algorithm, as well as some specific characteristics, as the settings and solutions are exactly known.

5.1.1 Weierstrass–Mandelbrot

To evaluate the ES/GA implementation, it was chosen to optimise solutions of the Weierstrass–Mandelbrot function. This function is defined as:

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

From the evolutionary algorithm point of view, the function is difficult to optimize, because it has an irregular fitness landscape, but this is not the point of the experiment, which is to analyse the behaviour of the parallelization over a GPU card. A variant is used here that calculates an approximation of the infinite sum, by limiting the number of iterations of the sum. Changing the number of iterations allows to test the impact of having a short or long evaluation function on the overhead induced by the parallelization of the evaluation on a GPU card. A multi-dimensional version is used to test the impact of the genome size on transfer time over the GPU, so the evaluation function finally is

$$W_{b,h}(x) = \sum_{j=0}^{n} \sum_{i=1}^{\text{iteration}} b^{-ih} \sin(b^i x_j)$$
with $b > 1$ and $0 < h < 1$

These experiments were performed on a Linux machine with an EASEA code. The same machine was used for testing CPU and GPU versions. It is a Pentium IV 3.6 GHz machine-based, equipped with a 8800GTX card (hardware of the same generation). Different population sizes are tested with an evaluation function with several complexities (iteration = 10, 70 or 120). For testing the influence of population size, the dimension of the problem remains the same($n = 10$), even if the genome size artificially increases.

This function is also used for its complexity. Indeed, Fig. 2 shows the distribution of the execution time of the algorithm on the CPU, for 10 generations.

Figure 3 shows that the population size must be large enough to fully exploit the parallelism of a GPGPU card. Below 2,000 individuals, the increase in computation time comes from the sequential part of the algorithm on the GPU. This can be seen because for 10 iterations (extremely

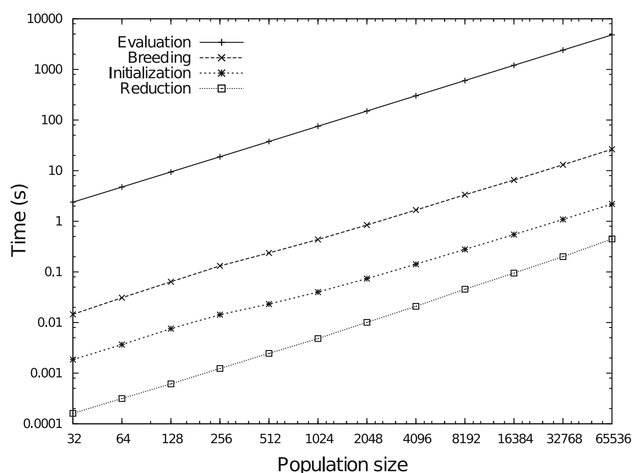

Fig. 2 Time repartition for an ES/GA example taken from the Sect. 6

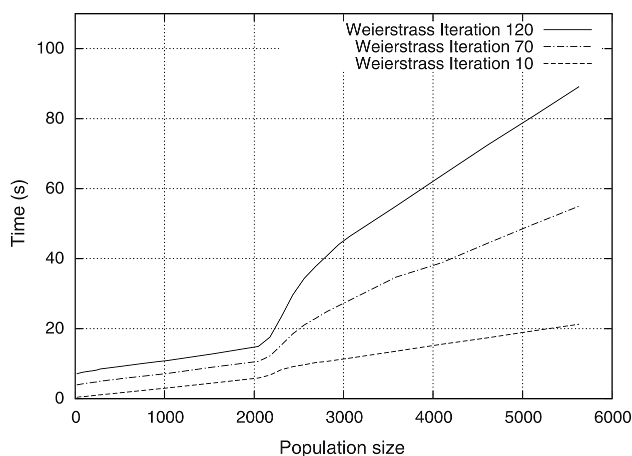

Fig. 3 Impact of population size and evaluation complexity on execution time

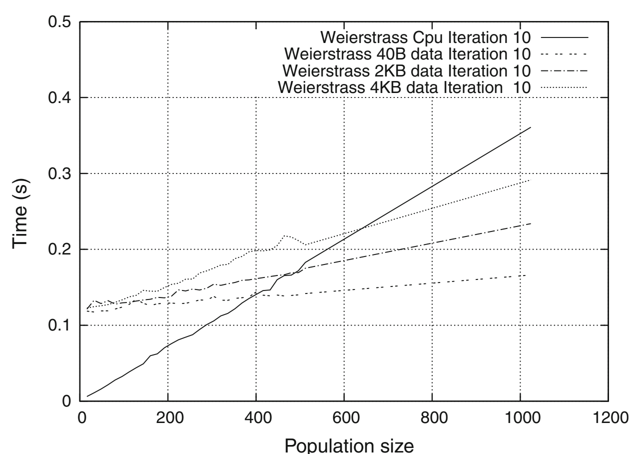

Fig. 4 Impact of individual size on transfer time overhead

fast evaluation function) the slope is identical before and after 2000.

On longer evaluations (70 and 120 iterations), no impact is seen as the population increases up to 2,000, because the card is not fully loaded yet. A larger population means more evaluations, but since they are done in parallel, they are done for free.

Beyond 3,000 individuals, the card is fully loaded. The impact of evaluation time can be seen, for long evaluations (70 and 120 iterations).

Between 2,000 and 2,500, the slope raises a lot, because supposing that the card can deal with 2,048 evaluations in parallel, 2,049 evaluations will take twice the time of 2,048 evaluations. In reality, there is no obvious step because the complex hardware scheduling capacities of the card manage to smoothen the curve until it is fully loaded, which happens beyond 3,000 individuals.

The curves become straight again as hardware scheduling manages to load the parallel card linearly with the number of threads.

In Fig. 4, it is possible to see that even with a very fast evaluation function (10 iterations), the GPU card becomes faster than the CPU when there are more than 400 individuals for a small genome (10 floats = 40 bytes) and 650 individuals for a very large genome (1,000 floats = 4 Kbytes). The influence of the evaluation execution-time on the slopes is negligible for the GPU because for such population sizes, the card is not loaded yet, and the evaluation function is very limited in forms of computation, so execution time is still virtually null. The slopes come from the sequential management of the population on the CPU. Transfer plus initiation of the computation on the GPU takes 0.12 s.

The CPU slope is steeper because to the opposite of GPU computation, one must add evaluation time on top of population management.

Finally, several cards were pitted against a recent Intel core i7 950 CPU, clocked at 3.07 GHz (cf. Fig. 5). Due to

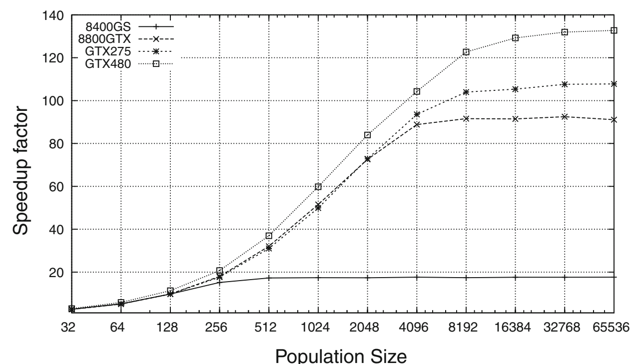

Fig. 5 Speedup on an average of 10 runs for several GPGPUs cards versus an Intel i7 950 on a 10 dimensions 120 iterations Weierstrass benchmark

Algorithm 1 in Sect. 3.1.2, EASEA distributes the population in a transparent way, according to the resources available on the card. As with all experiments below, speedups are compared to a sequential execution done on a single processor of the CPU. At best, one can imagine that a parallel version running on the CPU could yield a linear speedup with respect to the number of processor cores.

The figure shows the work of the hardware scheduler as the population size increases. The 8400GS card is a very low-end card used for display, that is very cheap, for an advertised computation power of 33 GFlops. It is interesting to see that its eight cores yield a speedup of around 20× compared with one core of a latest generation Intel core i7. Because it has eight cores only, the card is fully loaded with a population of 512 individuals.

The next curve comes from the old 8800GTX card released in November 2006 that was used to create Fig. 3. The card hosts 128 cores and is supposed to yield 518 GFlops. The obtained speedup is around 90×, which is great, but disappointing, compared with the very cheap eight cores 8400GS. The fact that it now needs around 4,096 individuals to attain its maximal speedup (rather than slightly more than 2,000) suggests that the old 3.6 GHz Pentium IV that was used in Fig. 3 was too slow to drive the card.

Then comes a GTX275 card, released in April 2009, that was the top of the pre-Fermi generation of single GPU NVIDIA cards. It hosts 240 cores, with an advertised computing power of 1 Teraflop. The maximum obtained speedup factor is "only" about 108×, which is again a bit disappointing because this card was supposed to be nearly twice as fast as the 8800GTX.

Finally, the top curve comes from a GTX480, which is the current top of the line Fermi generation card. It hosts 480 cores, i.e. twice as many as the GTX275 card, but interestingly enough, its computing power is advertised to be 1.344 TFlops, i.e. only 1.3× more powerful than the GTX275. Indeed, the observed speedup is around 132×, which is about 1.22× faster than the GTX275, but for around 64k individuals.

The speedup obtained with EASEA on these cards is clearly not proportional to the number of cores they contains even though the clock speeds are roughly identical between the GTX275 and the GTX480. The difference in speedup is however in line with the advertised difference in TFlops.

5.1.2 Memetic algorithm and Rosenbrock function

The experiments were performed on the Rosenbrock function because it is very fast to compute. The idea was to expose the incurred overheads as much as possible so as to obtain the worst possible results and get a fair idea on the advantages of parallelizing the optimisation on a GPGPU card. Using an evaluation function that was much longer to evaluate would have hidden away the inevitable overhead.

Here again, the purpose of the experiments was not to test the efficiency of the local search algorithm, but rather to measure the speedup that parallelizing the local search would bring.

Rosenbrock's function (Shang and Qiu 2006) can be defined by the following equation :

$$f(x_1,\ldots,x_N) = \sum_{i=1}^{N/2}\left[100(x_{2i-1}^2 - x_{2i})^2 + (x_{2i-1} - 1)^2\right]$$

where N represents the number of dimensions, and therefore the genome size.

Experiments have been performed on a GTX275 nVidia card versus a 3.6 GHz Pentium IV under linux 2.6.27 32 bits.

Because GPGPU cards were not fitted with a random number generator at the time of the experiments, a deterministic local search algorithm was used. If a random generator is necessary, pseudo-random generators have been efficiently implemented on GPGPUs (Langdon 2008) and other pseudo random generators are now available through the recent CURAND library.

The local search algorithm used for the experiments requires a specific step and a specific number of search iterations. Until the maximum number of iterations is reached, the algorithm adds the step value to the first dimension of the individual, then evaluates the individual and compares its fitness to the fitness of the best individual to date. If the fitness improves, the individual replaces the best one and one step is added to the same dimension until the fitness stops improving, in which case the next dimension is explored in the same way. If, after the first attempt on one dimension, the fitness does not improve, the algorithm starts looking in the opposite direction.

Once the algorithm has browsed through all the dimensions, it goes back to the first dimension and repeats the process again until the specified number of iterations has been reached.

This algorithm is very crude, in that the step size is not adaptive, for instance. But the aim of this study is not to find the best local search algorithm that would fit all problems, but to experiment a Lamarkian memetic algorithm on a GPGPU card.

Other local search algorithms were tested during the development process but no impact on speedup has been detected. Therefore, all presented results use the simple algorithm described above.

Finally, the memetic algorithm is a special case for parallelization. Indeed, as one can see in Fig. 6, the time needed to evaluate an individual does not represent the

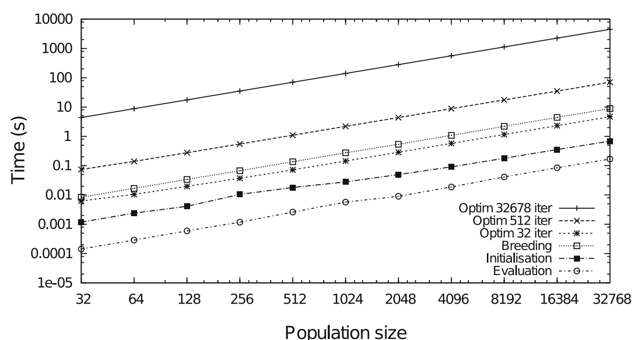

Fig. 6 Memetic algorithm time repartition on the artificial problem of the section 5 for different number of iterations

main part of the algorithm. Rosenbrock's function is lightweight to compute. Nevertheless, the parallelized part being the evaluation and the local optimization, the code ported onto the GPGPU is clearly predominant in terms of computing time. Moreover, as explained in Sect. 3, the optimization includes a call to the evaluation function in each cycle, which can explain why the optimization part is so much time consuming.

Figure 7 shows the obtained speedup for evaluation time and population transfer time on the GPGPU versus only the evaluation time on the Intel CPU, i.e. all this without the evolutionary algorithm.

Maximum speedup reaches a plateau above $\times 120$ for a population size of 32K individuals and as few as 256 iterations of the local search function.

Maximum speedup is attained for 2,048 and more individuals because under this population size, the cores of the GPU card are not fully loaded.

A good speedup of $\times 58$ is obtained for 2,048 individuals and 256 iterations, but it is important to remember that this very fast benchmark function maximises the influence of overhead. The surface seems to rise steeply still afterwards, but this impression is given by the fact that the scales are logarithmic. It requires 16K individuals to obtain $a \times 115$ speedup, i.e. approximately only twice as much speedup (over $\times 58$) for eight times the population size.

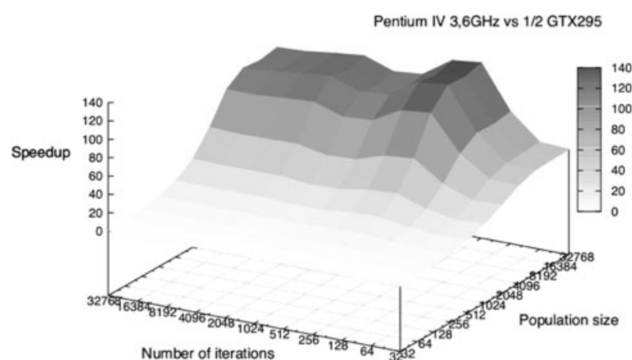

Fig. 7 Speedup for evaluation + transfer time only

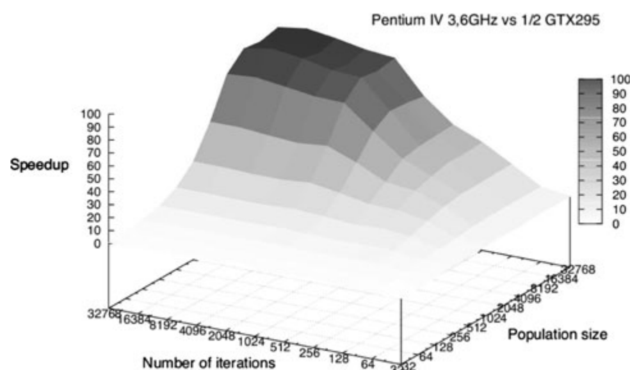

Fig. 8 Speedup for the complete algorithm

No explanation was found for the "pass" observed for 1,024 and 2,048 iterations above 8K individuals.

Because the power of GPU cards comes from their parallel architecture, one must use large populations to benefit from it.

Figure 8 shows the obtained speedup for the complete memetic algorithm (and not evaluation time only) automatically created by the EASEA language.

Maximum speedup reaches a plateau at around $\times 91$ for a population size of 32K individuals and 32K iterations of the local search function.

As above, speedup increases a lot up to 2,048 individuals, because under this population size the cores of the GPU card are not fully loaded.

A good speedup of $\times 47$ is obtained for 2,048 individuals and 2,048 iterations. Much larger numbers are required to overcome the overhead of the evolutionary algorithm that runs on the CPU. Maximum speedup ($\times 95$) is obtained for 32K individuals and 16K iterations.

5.1.3 Symbolic regression

To test the parallel genetic programming implemented by EASEA, it was decided to use a problem from Koza's book (1992) that is quite common in GP: symbolic regression. To analyze the behavior of this implementation, different tree and training set sizes were used.

Figure 9 shows the speedup achievable by the evaluation function only, with the evaluation method described in Sect. 1. The speedup is calculated for randomly generated populations with different sizes. These tests were performed on a Linux machine, equipped with an Intel Quad core Q8200 and a GTX275 GPGPU card. The multi-individuals per MP evaluation method described in section 3.2.3 was used, four individuals being evaluated at the same time on each MP.

As before, the evaluation phase is the predominant part in the algorithm, as long as the number of learning cases is sufficiently high. Figure 10 shows the distribution of

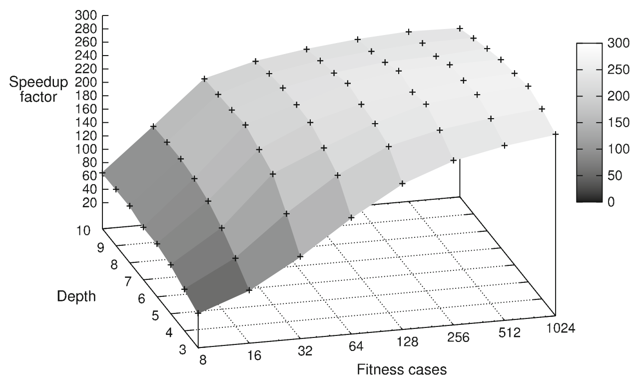

Fig. 9 Speedup factor with respect to the tree depth and the size of the learning set for the evaluation function only

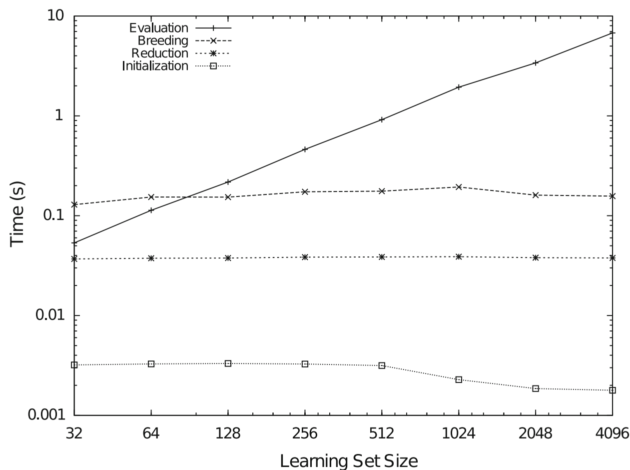

Fig. 10 Time repartition with respect to the number of learning cases

execution time with respect to the number of training cases, in a symbolic regression algorithm using a population of 4,096 individuals over 10 generations.

A first point that can be seen from this figure is that the speedups reach a plateau fairly quickly, with only 32 fitness cases, confirming that scheduling is a real gain. Secondly, in terms of speedup, the influence of tree size is much less important than the fact that the number of fitness cases is greater than 32.

Another interesting point is the influence of the function set on the speedup. Indeed, two learning function sets are listed in Table 1 that have different arithmetic intensities. Using the set containing the trigonometric functions, the computation / memory consumption ratio increases and the speedup improves, as can be seen in Fig. 11. Furthermore, with this set, it is possible to use SFUs, which allows to compute approximate values of trigonometric operators and to accelerate the calculation of the population evaluation with less accuracy. For the low arithmetic intensity function set, the speedup does not exceed 50×, while it reaches 200× using FS2 and 250× using the SFU.

Table 1 Different learning function sets

FS1	FS2
+, -, $\oplus$, ×	+, -, $\oplus$, ×, cos, sin, log, exp

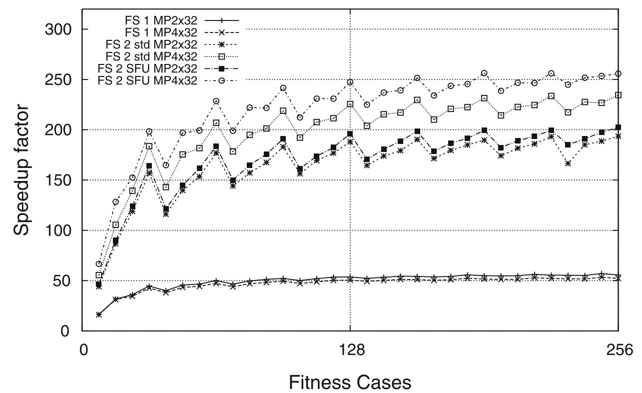

Fig. 11 Influence of function set computational intensity

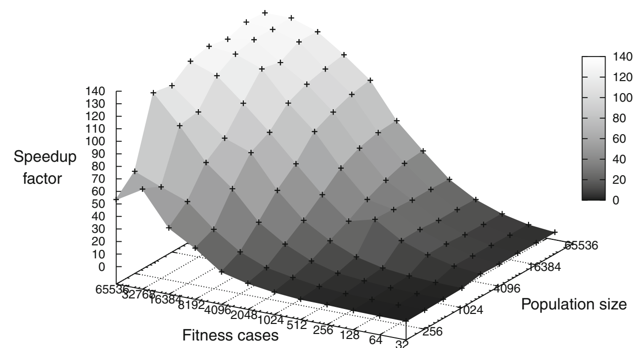

Fig. 12 Speedup for symbolic regression with respect to the number of fitness cases

Figure 12 shows the speedup achieved on the complete algorithm on a symbolic regression problem. The goal function being $cos(2x)$, the learning function set used here is the FS2 of Table 1. Even if the speedup is much greater than 1 on the evaluation function only even with a small number of training cases, it is to be noted that the speedup of the complete algorithm is lower. Indeed, the data volume is quite high here compared to the cost of the population evaluation. For a large number of training cases, the speedup becomes more attractive especially on a large population, such population sizes being common in genetic programming. This negative effect can also be attributed to the cost of the tree flattening step, which is more easily made beneficial in these extreme cases.

It is interesting to see that the obtained speedup is greater for GP than for other EAs. This comes from the fact that GP is very computation intensive, and parallelizing evaluation of the same individual over different test cases in SIMD mode is very efficient.

5.1.4 Conclusion on benchmark problems

The use of GPGPUs into EASEA reaches very interesting speedups with minimal effort (compile with the *-cuda* option on the command line). This achieves a real gain in the quality of the solution when search time is a problem, even for very cheap NVIDIA graphics cards. In addition, by providing speedups in the hundreds, EASEA should allow the exploration of search space still inaccessible to evolutionary algorithms.

Then, EASEA can parallelize over several GPU cards in one machine, and, if even more computing power is needed, EASEA provides an island model that can deal with potentially heterogeneous machines with or without GPUs, assuming they are accessible over the internet through their IP number.

It is still necessary to keep a few considerations in mind. Speedups obtained with the EASEA parallelisation model are interesting when the evaluation time taken by one generation is important. GPGPU is an external host system and communication between the systems must of course be justified. For tasks with a high computation/memory consumption ratio, using GPGPUs for evolutionary computation gives very interesting results.

5.2 Real-world problems: aircraft model regression

One of the main automation science principles is to describe the evolution of any controlled system as a first-order differential equation with respect to physical variables and control inputs. The number of physical variables (called state variables) needed to describe such a system can vary, as well as the number of control inputs.

Usually, the state variables cannot be directly measured and the system outputs correspond to installed sensor measurements. In such cases, state estimators (or observers) are used to estimate the state variable values from the sensor outputs.

A system is called SISO (Single Input Single Output) when its control and output vector contain only one element each. Respectively, a system with multiple control variables and multiple outputs is called MIMO (Multi Input Multi Output). This means that only one differential equation can describe the physical evolution of a SISO system, while several equations are needed in the case of an MIMO system and such a system is called a State Space Representation.

There are two types of State Space Representations: Linear State Space Representation (which is represented by matrix products) and Nonlinear State Space Representation (which is represented by mathematical functions).

Most of the automation tools are designed to be used with a linear representation. However, in the case of a nonlinear system modelling, a linearization is performed around the equilibrium point and the obtained linear form allows the use of classical automation mathematical operations.

Let us consider the state vector X:

$$x^T = [x_1 x_2 x_3 \ldots x_n]$$

and the following control vector U:

$$u^T = [u_1 u_2 u_3 \ldots u_m]$$

the associated nonlinear state space representation is

$$\begin{cases} \dot{x}_1(t) = f_1(t, x_1(t), \ldots, x_n(t), u_1(t), \ldots, u_m(t)) \\ \dot{x}_2(t) = f_2(t, x_1(t), \ldots, x_n(t), u_1(t), \ldots, u_m(t)) \\ \qquad \ldots \\ \dot{x}_n(t) = f_n(t, x_1(t), \ldots, x_n(t), u_1(t), \ldots, u_m(t)) \end{cases}$$

5.2.1 Aircraft nonlinear state space

In aeronautics control science, an aircraft can be described as an MIMO nonlinear system.

The state variables vector was chosen to contain 14 elements and the control input vector to contain 4. All the state variables are supposed to be known, due to the proper application of an Extended Kalman Filter (EKF) dedicated to navigation.

It is important to notice that an autopilot needs a theoretical state–space representation to perform adapted control laws and to determine the optimal actuator orders to be sent to the platform system (the plane). As previously mentioned, for each algorithm step, the nonlinear state–space representation is linearized in real time, and classical automation laws are applied (such as optimal control, robust control).

The choice of the state vector is

$$x^T = [V, \alpha, \beta, p, q, r, q_1, q_2, q_3, q_4, N, E, h, T],$$

where V is the airspeed, α the angle of attack, β the heeling angle, p the x axis rotation rate, q the y axis rotation rate, r the z axis rotation rate, $q_1 q_2 q_3 q_4$ the attitude quaternions, N the latitude, E the longitude, h the altitude, T the real thrust.

The choice of the control vector is $u^T = [T_c \delta_e \delta_a \delta_r]$, where T_c is the commanded throttle, δ_e the commanded elevators, δ_a the commanded ailerons, δ_r the commanded rudders, as in Fig. 13.

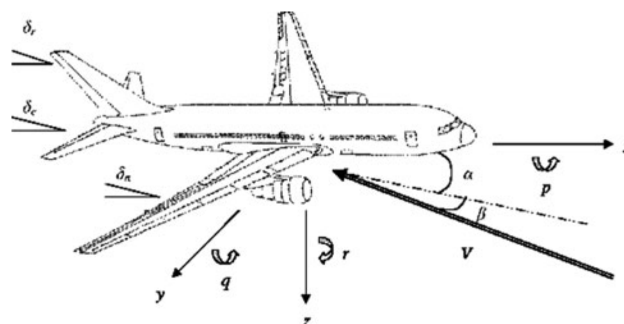

Fig. 13 State and control variables of an airplane

As described earlier, the nonlinear state–space representation is $\dot{x}(t) = f(t, x(t), u(t))$

And more precisely

$$\begin{cases} \dot{V}(t) = f_1(t, V(t), \alpha(t), \ldots, T(t), T_c(t), \ldots, \delta_r(t)) \\ \dot{\alpha}(t) = f_2(t, V(t), \alpha(t), \ldots, T(t), T_c(t), \ldots, \delta_r(t)) \\ \ldots \\ \dot{q}_1 = f_7(t, V(t), \alpha(t), \ldots, T(t), T_c(t), \ldots, \delta_r(t)) \\ \dot{q}_2 = f_8(t, V(t), \alpha(t), \ldots, T(t), T_c(t), \ldots, \delta_r(t)) \\ \dot{q}_3 = f_9(t, V(t), \alpha(t), \ldots, T(t), T_c(t), \ldots, \delta_r(t)) \\ \dot{q}_4 = f_{10}(t, V(t), \alpha(t), \ldots, T(t), T_c(t), \ldots, \delta_r(t)) \\ \ldots \\ \dot{h}(t) = f_{13}(t, V(t), \alpha(t), \ldots, T(t), T_c(t), \ldots, \delta_r(t)) \\ \dot{T}(t) = f_{14}(t, V(t), \alpha(t), \ldots, T(t), T_c(t), \ldots, \delta_r(t)) \end{cases}$$

5.2.2 *Aircraft model regression*

As previously mentioned, an aircraft theoretic State-Space representation is needed by the autopilot system. The determination of the system equations can be quite difficult and specialized experts are often needed to perform this task.

Genetic programming could be used to compute all the equations of the state–space system, by learning from telemetry files containing data of a previously performed flight. A telemetry file is a sequence of recorded points, each of them describing the state space variables and inputs of the plane.

In this paper, a small F3A airplane has been chosen through its nonlinear state space representation. F3A are radio-controlled aerobatic competition airplanes that can fly various trajectories, without major structural constraints.

5.2.3 *Considered functions*

Usually, the attitude of an aircraft can be described in two ways: Euler angles and Quaternions. The Euler angles are intuitively easier to understand, because they represent what is called: Roll (aircraft angle around x axis), Pitch (aircraft angle around z axis) and Yaw (aircraft angle around z axis). Quaternions are four variables which are more often used in the navigation field, because they cover the entire angles domain.

In the state–space system, an equation is dedicated to the evolution of each quaternion. These equations have been chosen to be regressed by genetic programming.

The Quaternion functions in the state–space system are given by the following equations:

$$\begin{cases} \dot{q}_1 = f_7 = 0.5(q_4 p - q_3 q + q_2 r) \\ \dot{q}_2 = f_8 = 0.5(q_3 p + q_4 q - q_1 r) \\ \dot{q}_3 = f_9 = 0.5(-q_2 p + q_1 q + q_4 r) \\ \dot{q}_4 = f_{10} = 0.5(-q_1 p - q_2 q - q_3 r) \end{cases}$$

A telemetry file, containing the necessary state variables as well as the control variables, has been created through a nonlinear state space system of a small F3A airplane performing a simulated flight. The learning set contains 6,000 points, i.e. around 1 min of flight.

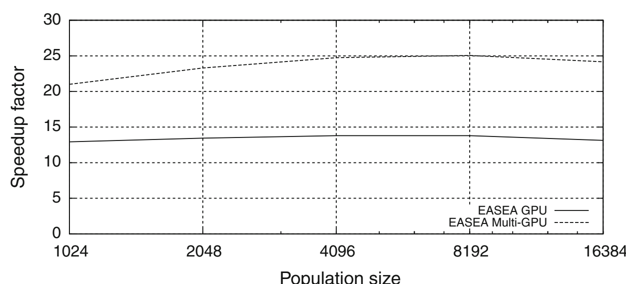

Fig. 14 Speedup obtained on the airplane problem with EASEA tree-based GP implementation

5.2.4 *Experiments*

An EASEA GP run has been done in order to regress each equation on a GNU/Linux machine, equipped with an Intel Core i7 CPU 920 and three NVIDIA GTX480 GPGPU cards. The same machine has been used to compute CPU and GPGPU algorithms, to calculate speedups. Figure 14 shows the obtained speedups using the parameters shown in Table 2, using 1–3 GPGPU cards versus the Intel Core i7 only. EASEA divides the population into bundles that are sent to each GPGPU. A GPGPGU evaluates its part of the population and results are collected by the CPU whenever the evaluation phase is finished onto each GPGPU. In this case, the time spent in the evolution engine is negligible as compared to the evaluation time. Because of the simple function set (which does not contain complex functions such as sine, cosine, exponential, that are approximated by SFUs in a very efficient way), the speedup is lower than shown in previous section 8 for more complex function sets. In the current experiment, the terminal set is larger (ERC, 17 variables), i.e. the GPU interpretor has to perform more memory accesses than the one presented in the original paper, where the only variable can be stored in a register by the optimizer. These drawbacks, as well as the evolutionary process, can be held responsible for the drop in speedup.

Speedup factors still remain satisfactory given the size of the problem. A real run takes hours of computation on

Table 2 Parameter used to regress a part of the airplane model

Population size	40 960
Number of generation	250
Function set	+, −, *, /
Terminal set	×[1]...,×[17], ERC {0,1}
Learning set	6,000 values

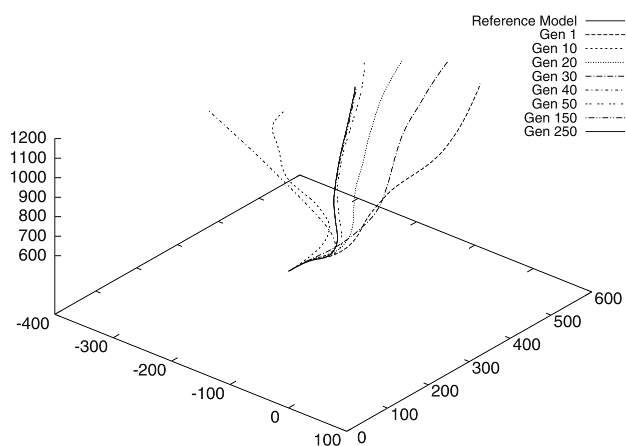

Fig. 15 Trajectories from several models obtained at different generations and from the original model

CPU, but just several minutes on GPU. It is very important, because 14 different functions need to be found to solve the complete problem.

To assess the four regression functions, the same series of orders have been sent to the real state–space representation, originally used to create the telemetry file, and to the state–space representation containing four evolved Quaternion functions, taken at different generations. The plotted curves are shown in Fig. 15.

Quaternion functions impact the plane trajectory in a very strong manner, because they describe the attitude of the aircraft which has important consequences on the navigation equations, as an attitude matrix is used to determine the direction and the length of the absolute aircraft acceleration vector, in the inertial frame.

In this example, the last trajectory is so close, that it is not possible to distinguish it from the trajectory obtained with the original model. Other trajectories are given by the best individuals of different generations showing that bad individuals have a real influence on the model.

6 Conclusion

Multi-core architecture is becoming the new standard, due to hardware (heat emission) constraints which limit CPU frequency. Furthermore, many-core processors, such as GPGPUs, are widespread nowadays. Highly parallel algorithms can directly benefit from the full power of these processors, which is the case of evolutionary computation.

In fact, evolutionary algorithms have an edge over other paradigms, as they are not only intrinsically parallel, but they follow the same flowchart as rendering algorithms, where an identical algorithm is run over thousands of pixels or vertices. This means that the hardware that is developed for graphic video industry can directly be used by evolutionary algorithms to optimize virtually any kind of problems. This is quite unique, as the use of GPGPU cards is currently restricted to small parallel parts of large scientific programs. Further developments consist in porting a complete algorithm on a GPGPU card, for even better speedups.

Once this is done, EAs will be able to completely harness the power of massively parallel systems, which will give them a tremendous edge over other techniques that may not parallelize as well.

Then, it is probable that the current algorithms that have been designed in the late 1970s will need to be revisited, as they were tuned for populations ranging between 100 and 1000 individuals (GP was an exception with John Koza, who regularly used populations in the million).

The advent of massively parallel systems will put forward the necessity to be able to deal with huge populations, even for evolution strategies if users want to benefit from the speedup that such systems offer.

Massively parallel evolutionary systems will allow to tackle problems that are yet unreachable with standard CPUs. Right now, our team can execute runs on a heterogeneous cluster of 30 TeraFlops that has yielded interesting results in chemistry. The time is near where the computing power that was needed by Koza's GP to routinely obtain human-competitive results is widely available, due to massively parallel systems.

The EASEA platform is a first step towards a general use of massively parallel systems for evolutionary computation.

References

Ackley D, Littman M (1992) Interactions between learning and evolution. In: Langton CG, Taylor JDFC, Rasmussen S (eds) Artificial life II, vol 10. Addison-Wesley, pp 487–509

Alba E, Tomassini M (2002) Parallelism and evolutionary algorithms. IEEE Trans Evol Comput 6(5):443–462

Brameier M, Banzhaf W (2007) Linear genetic programming. No. XVI in genetic and evolutionary computation. Springer, Berlin

Chitty DM (2007) A data parallel approach to genetic programming using programmable graphics hardware. In: Proceedings of the 9th annual genetic and evolutionary computation conference (GECCO). ACM, New York, pp 1566–1573

Collet P, Schoenauer M (2003) Guide: unifying evolutionary engines through a graphical user interface. In: Liardet P, Collet P, Fonlupt C, Lutton E, Schoenauer M (eds) Artificial evolution. Lecture notes in computer science, vol 2936. Springer, Berlin, pp 203–215

Collet P, Lutton E, Schoenauer M, Louchet J (2000) Take it EASEA. In: Proceedings of the 6th international conference on parallel problem solving from nature. Springer, London, pp 891–901

De Jong K (2008) Evolutionary computation: a unified approach. In: Proceedings of the 10th annual genetic and evolutionary computation conference (GECCO). ACM, New York, pp 2245–2258

Deb K, Agrawal S, Pratap A, Meyarivan T (2002) A fast and elitist multi-objective genetic algorithm: NSGA-II. IEEE Trans Evol Comput 6(2):182–197

Fok KL, Wong TT, Wong ML (2007) Evolutionary computing on consumer graphics hardware. IEEE Intell Syst 22(2):69–78

Gagné C, Schoenauer M, Parizeau M, Tomassini M (2006) Genetic programming, validation sets, and parsimony pressure. In: Collet P, Tomassini M, Ebner M, Gustafson S, Ekárt A (eds) Proceedings of the 9th European conference on genetic programming. Lecture notes in computer science, vol 3905. Springer, Budapest, pp 109–120

Harding S, Banzhaf W (2007) Fast genetic programming on GPUs. In: Ebner M, O'Neill M, Ekárt A, Vanneschi L, Esparcia-Alcázar A (eds) 10th European conference on genetic programming (EuroGP). Lecture notes in computer science, vol 4445. Springer, Berlin, pp 90–101

Holladay K, Robbins K, Ronne JV (2007) Fifth™ A stack based GP language for vector processing. In: Ebner M, O'Neill M, Ekárt A, Vanneschi L, Esparcia-Alcázar A (eds) 10th European conference on genetic programming (EuroGP). Lecture notes in computer science, vol 4445. Springer, Berlin, pp 102–113

Koza JR (1992) Genetic programming: on the programming of computers by means of natural selection (complex adaptive systems). MIT Press, Cambridge

Krüger F, Maitre O, Jiménez S, Baumes L, Collet P (2010) Speedups between ×70 and ×120 for a generic local search (memetic) algorithm on a single GPGPU chip. Appl Evol Comput 501–511

Langdon WB (2008) A fast high quality pseudo random number generator for graphics processing units. In: Wang J (ed) IEEE World Congress on computational intelligence, Hong Kong, pp 459–465

Langdon WB, Banzhaf W (2008) A SIMD Interpreter for genetic programming on GPU graphics cards. In: O'Neill M, Vanneschi L, Gustafson S, Esparcia Alcazar AI, De Falco I, Della Cioppa A, Tarantino E (eds) 11th European conference on genetic programming (EuroGP), Lecture notes in computer science, vol 4971. Springer, Berlin, pp 73–85

Li JM, Wang XJ, He RS, Chi ZX (2007) An efficient fine-grained parallel genetic algorithm based on GPU-accelerated. In: IFIP International conference on network and parallel computing workshops (NPC). IEEE Computer Society, Los Alamitos, pp 855–862

Maitre O, Baumes LA, Lachiche N, Corma A, Collet P (2009a) Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA. In: Rothlauf F (ed) Proceedings of the 11th annual conference on genetic and evolutionary computation (GECCO). ACM, New York, pp 1403–1410

Maitre O, Lachiche N, Clauss P, Baumes L, Corma A, Collet P (2009b) Efficient parallel implementation of evolutionary algorithms on GPGPU cards. In: Sips H, Epema D, Lin HX (eds) Euro-Par 2009 parallel processing. Lecture notes in computer science, Springer, Berlin, pp 974–985

Maitre O, Lachiche N, Collet P (2010a) Fast evaluation of GP trees on GPGPU by optimizing hardware scheduling. In: Esparcia-Alcázar A, Ekárt A, Silva S, Dignum S, Uyar A (eds) Genetic programming. Lecture notes in computer science, vol 6021. Springer, Heidelberg, pp 301–312

Maitre O, Querry S, Lachiche N, Collet P (2010b) EASEA parallelization of tree-based genetic programming. In: IEEE congress on evolutionary computation (CEC), pp 1–8

Matsumoto M, Nishimura T (1998) Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans Model Comput Simulat 8(1):3–30

Munawar A, Wahib M, Munetomo M, Akama K (2009) Hybrid of genetic algorithm and local search to solve MAX-SAT problem using nVidia CUDA framework. Genet Program Evol Mach 10(4):391–415

NVIDIA (2008) Nvidia cuda programming guide 2.0

Ong YS, Keane AJ (2004) Meta-Lamarckian learning in memetic algorithms.IEEE Trans Evol Comput 8(2):99–110

Robilliard D, Marion-Poty V, Fonlupt C (2008) Population parallel GP on the G80 GPU. In: O'Neill M, Vanneschi L, Gustafson S, Esparcia Alcázar A, De Falco I, Della Cioppa A, Tarantino E (eds) 11th European conference on genetic programming (EuroGP), vol 4971. Springer, Berlin, pp 98–109

Robilliard D, Marion V, Fonlupt C (2009) High performance genetic programming on GPU. In: Workshop on bio-inspired algorithms for distributed systems. ACM, New York, pp 85–94

Shang YW, Qiu YH (2006) A note on the extended Rosenbrock function. Evol Comput 14(1):119–126

Sharma D, Collet P (2010a) An archived-based stochastic ranking evolutionary algorithm (ASREA) for multi-objective optimization. In: GECCO '10: Proceedings of the 12th annual conference on genetic and evolutionary computation. ACM, New York, pp 479–486

Sharma D, Collet P (2010b) Gpgpu-compatible archive based stochastic ranking evolutionary algorithm (g-asrea) for multi-objective optimization. In: Schaefer R, Cotta C, Kolodziej J, Rudolph G (eds) PPSN (2). Lecture notes in computer science, vol 6239. Springer, Berlin, pp 111–120

Spector L, Robinson A (2002) Genetic programming and autoconstructive evolution with the push programming language. Genet Program Evol Mach 3(1):7–40

Wong ML (2009) Parallel multi-objective evolutionary algorithms on graphics processing units. In: GECCO '09: proceedings of the 11th annual conference companion on genetic and evolutionary computation conference. ACM, New York, pp 2515–2522

Wong ML, Wong TT (2006) Parallel hybrid genetic algorithms on consumer-level graphics hardware. In: IEEE congress on evolutionary computation (CEC). pp 2973–2980

Yu Q, Chen C, Pan Z (2005) Parallel genetic algorithms on programmable graphics hardware. In: First international conference on natural computation (ICNC). LNCS, vol 3612. Springer, Heidelberg, pp 1051–1059

Zitzler E, Laumanns M, Thiele L (2002) SPEA2: improving the strength pareto evolutionary algorithm for multiobjective optimization. In: Giannakoglou K et al (eds) Evolutionary methods for design, optimisation and control with application to industrial problems (EUROGEN 2001). International Center for Numerical Methods in Engineering (CIMNE), pp 95–100