Parallel Computation of Evolutionary Algorithms on GPUs. A review.

Journal Title
XX(X):1–31

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

anonymous[1]

Abstract

Metaheuristics and bioinspired methods have been recently adapted to massively parallel computation devices such as the General Purpose Graphics Processing Units (GPGPUs). One of the most prolific techniques taking advantage of this approach are the Evolutionary Algorithms (EAs), since they have an inherently parallel structure, based on a population of independent agents. This paper presents an exhaustive survey on all the literature related to parallelization of EAs on GPUs. We have reviewed more than 200 publications, extracting their features, such as the type of EA implemented, its properties, or if it is applied to solve real world problems. Thus, we have defined a taxonomy of the different approaches used in the parallelization process. Moreover, a deeper analysis has been conducted on a big selection, also describing the most representative in each category. The analysis shows that this is a very prolific research area, with an increasing number of published papers every year. Genetic Algorithms are so far the most widely used approach in EAs, and they are mainly applied to solve real world problems due to the advantages that GPGPU computation offers. The paper also presents a review of advances in hardware and software along with trends in the field.

Keywords

Survey, Review, GPUs, General Purpose GPUs, Parallel Computation, Evolutionary Algorithms, metaheuristics.

[1]My University

Corresponding author:
anonymous

Introduction

The application of Evolutionary Algorithms to obtain the solution of many real world problems or large instances of some benchmark problems requires a high amount of computational power. In order to deal with them, parallelization of this kind of algorithms has been conducted with the principal aim of reducing the computational time whereas, eventually, the quality of the solutions is improved. These algorithms were usually executed in one or more central processing units (CPUs) each one with one or more cores. However the advances in the video game industry have led to the production of low cost and high-performance graphics processing units (GPUs) included in almost every PC nowadays and also in most of the mobile devices.

GPUs are specialized stream processors, initially designed for rendering graphics in all kind of applications, but specially games. Typically, a GPU is able to perform graphics manipulations at a much higher speed than a general purpose CPU, since the graphics processor is specifically designed to handle certain primitive operations which occur frequently in graphics applications. Internally, the GPU contains a number of small processors that are used to perform calculations.

Depending on the power of a GPU, the number of threads that can be executed in parallel on such devices is currently in the order of hundreds to thousands and this number is expected to be increased every year. Nowadays, developers can write their own high-level programs on GPU. Due to the wide availability, programmability, and high-performance of these consumer-level GPUs, they are cost-effective for, not just game playing, but also scientific computing.

Since some years ago, GPUs are exposed to the programmer as a set of general-purpose shared-memory Single Instruction Multiple Data (SIMD) multi-core processors. This makes these architectures well suited to run large or complex computational problems in parallel. Thus, nowadays, these units are used as general-purpose processors, and leveraged as powerful substitutes or complements for CPUs in many problem solving frameworks.

General-purpose computing on graphics processing units (GPGPU) consists in the use of a Graphics Processing Unit (GPU), which typically handles computer graphics related calculus, to perform any kind of computation traditionally carried out by a central processing unit (CPU). Recently there has been a growing interest in GPU computation, due to the fact that this kind of processors has the ability to perform massive and restricted parallel processing, which is very attractive to researchers with applications that require intensive parallel computation.

EAs[1] are a class of probabilistic search and optimisation algorithms inspired on the model of Darwinistic evolution. There are several subtypes, depending on the data structure that is preferentially used for representing solutions, but Genetic Algorithms (GAs)[2] and Genetic Programming (GP)[3] are the most extended. The main features are common to all of them: a population of individuals or potential

solutions of the target problem, a selection method that favours better solutions, and a set of operators that act upon the selected solutions. Thus, after an initial population is created (usually random), the selection and operators - crossover and mutation - are applied to the individuals, and the new population then replaces the older one. This is repeated for a number of generations or until another stop condition is met. If there is defined a correct *fitness function*, which assigns a reliable value to every individual, this process guarantees that the average quality of the population increases with the number of generations.

EAs present a structure which is inherently parallel, because the individuals of the population are relatively independent (they work independently, but all of them cooperate to find the solution). Thus, their distribution on different computation nodes is quite straight and not very difficult to implement, being normally a node devoted to work on subpopulations (island model)[4] and even, in the finest-grained models (or fully distributed), every node is assigned to one individual[5].

The aim of the parallelization is usually the improvement of the running time for yielding an objective solution (if known), or for obtaining a certain degree of quality in the solution, if it is not the optimal. Sometimes this distribution could also imply a different searching scheme, which can lead to a different searching area of the space of solutions and thus, to a different solution (or set of solutions) for an optimization problem, which, in turn, could be better than the one obtained in a sequential approach. For these reasons, EAs are so far the most prolific metaheuristic in terms of existing parallel approaches, including the (recent) distribution on GPUs, by means of the new parallelization paradigm which we focus on here.

Thus, the paper presents a survey of the most extended evolutionary algorithms and the different proposals of GPUs-parallelization approaches made in a number of research works. They are organized according to a parallelization taxonomy, which is also introduced. In addition, this work describes the features and advantages of GPGPUs, then it explains the general approaches to use and profit a GPGPU model, and gives an overview of currently available programming tools and software systems.

The rest of the paper is structured as follows: Next section presents GPUs as highly parallel devices architectures. Then, a background on the different higher level programming languages used to take advantage on GPUs is described. The following section proposes a taxonomy based on the different parallel structures implemented in the literature, in order to categorize all the papers. Then, a set of representative works describing different approaches using GPUs are reviewed and divided following the proposed taxonomy. Finally, some conclusions are remarked in the last section.

Throughput, parallelism and GPUs

Moore's Law describes a long-term trend in the history of computing hardware: the number of transistors that can be placed inexpensively on an integrated circuit has doubled approximately every two years.

The trend has continued for more than half a century and is not expected to stop, theoretically until not too many years above 2016. On 2005 Gordon Moore stated in an interview that his law cannot be sustained indefinitely because transistors would eventually reach the limits of miniaturization at atomic levels. Maybe it is time for Koomey's Law[6] to replace Moore's Law. Koomey says that energy efficiency is doubled every 18 months. For fixed computing load, the amount of battery you need will fall by a factor of two every year and a half.

Parallel computation has recently become necessary to take full advantage of the gains allowed by Moore's law. For years, processor makers consistently delivered increases in clock rates and instruction-level parallelism, so that single-threaded code is executed faster on newer processors with no modification. Now, to manage CPU power dissipation, processor makers favor multi-core chip designs, and software has to be written in a multi-threaded or multi-process manner to take full advantage of the new hardware[7].

Graphics processors have rapidly matured over the last years, leaving behind their roots as fixed function accelerators, and growing into almost general purpose computational devices for highly parallel workloads. Some of the earliest academic work about GPUs as computational devices date back to University of Washington in 2002[8] and Stanford in 2004[9].

GPUs are similar to multi-core CPUs but with two main differences, as can be seen in Figure 1. Traditionally, CPU main interest has been fast execution of single threaded code instead GPUs are made for throughput. CPUs try to improve the execution of a single instruction stream while GPUs take the opposite route obtaining benefits from massively threaded streams of instructions and/or data. The second difference is how threads are scheduled. The operating system schedule threads over different cores of a CPU in a preemptive fashion. GPUs have dedicated hardware for the cooperative scheduling of threads.

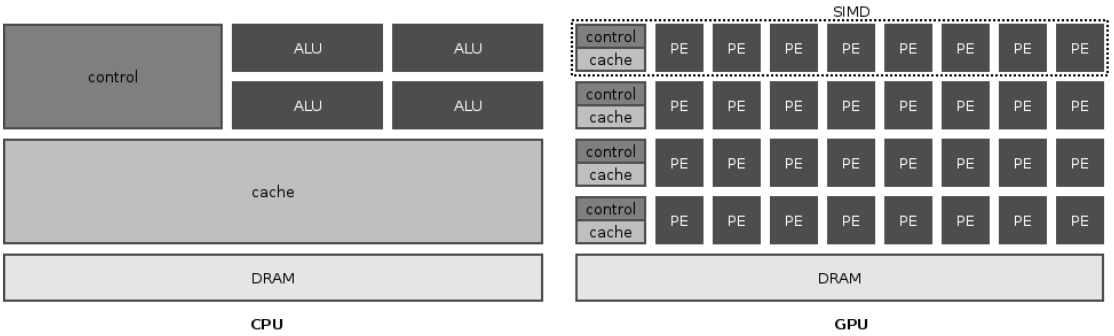

Figure 1. CPU and GPU block diagram comparison.

Physically GPUs are huge in comparison with CPUs as shown in Table 1. Latest microprocessors from the two main vendors, AMD and Intel, are over 1 billion transistors while latest GPUs from AMD

and NVIDIA[10] are over 7 billion transistors. CPUs draw 100W at most, a limit established by the cost of commodity heat sinks and fans. GPUs have bigger power consumption and currently can be reach 400 to 500W. This is made possible with the use of exotic cooling solutions. CPUs are built with the finest technology, read best lithography available, while GPUs are made with budget in mind in cheaper processes.

No standard terminology can be found in the world of graphics hardware. Every company have a different set of terms to refer to the same underlying objects and principles. Many authors try to alleviate this lack of standard terms calling 'shader core' to an Intel Execution Unit (EU), an AMD Single Instruction Multiple Data (SIMD) or an NVIDIA Streaming Multiprocessors (SM). Any of these refer to a single processor core inside of the GPU that can fetch, decode, issue and execute instructions. The shader core is composed of several execution units (EU) that can execute an individual vector operation equivalent to an Advanced Vector Extensions (AVX) or Streaming SIMD Extensions (SSE) instruction. AMD call this kind of EU streaming processor (SP), while NVIDIA uses the term CUDA core. Table 1 compares several existing CPUs, GPUs and Accelerated Processing Units (APUs), taking into account these terms and their meaning.

manufacturer & model	estimated transistors (billions)	die area (mm^2)	(shader) cores CPU/GPU	clock rate (GHz) CPU/GPU	memory bandwidth (GB/s)	GFLOPS (single precision)	TDP (W)	approx. price ($)
AMD Opteron 6386 SE	2.4	630	16	3.5	75	332.8	140	1392
Intel Xeon E7 8890 V4	5.69	662	24	2.2	102	3500	165	7174
AMD A10-7890K	2.41	245	4/512	4.1/0.866	15.76	1018	95	150
Intel Core i7 6950X	3.2	122	10	3.0	34	900	140	1723
AMD Radeon RX 480	6.2	438	2304	1.266	256	5100	150	200
NVIDIA GeForce GTX 1080	7.2	314	2560	1.607	320	8228	180	700
AMD Radeon R9 290X	6.2	438	2816	1	320	5632	250	550
AMD Radeon R9 295X2	2×6.2	2×438	2×2816	1.018	2×320	2×5733	500	1500
AMD Radeon R9 Fury X	8.9	596	4096	1.050	512	8601.6	275	650
NVIDIA GeForce GTX 980M	5.2	398	2048	1.126	224.3	3189	165	550
NVIDIA GeForce Titan Z	2×7.08	2×561	2×2880	0.705	2×336.5	8122	475	1500
AMD FirePro S10000	2×4.3	2×352	2×1792	0.825	2×240	2×2956	375	3000
NVIDIA QUADRO K6000	7.1	561	2880	0.9	288	3950	225	5000
NVIDIA Tesla K40	7.1	561	2886	0.745	288	5364	235	4000
Intel Xeon Phi 5100P	5	600	60	1.053	320	2020	225	2200

Table 1. CPUs, APUs and GPUs comparison of the best professional and commodity desktop hardware available nowadays. A slash is used in APUs to separate CPU/GPU parts.

Nowadays NVIDIA has one of the best performance for GPU computing[11] and in the second step is AMD. Both makers take different compromises in the design of theirs GPUs. AMD has more execution units but its memory hierarchy is weaker. This way software bounded by memory bandwidth or with

strong ordering inter-dependencies prefers NVIDIA hardware. On the other side, loads capped by pure ALU execution power use to be faster on AMD hardware.

Traditionally CPUs and GPUs have been separated. In 2016 the trend is to integrate both into a single chip for cost efficiency reasons. Only top models remain as discrete devices. Some of this are top models from Intel's Core i7 or AMD's Opteron 6300 families. Also best discrete GPUs from NVIDIA and AMD, like consumer GeForce GTX 1080 and Radeon RX 480 and professional line QUADRO K6000 and Radeon FirePro S10000.

From 2006 CPUs with an integrated GPU have became popular. AMD and Intel started selling this kind of combined processor and graphics card. The term Accelerated Processing Unit has been coined for this kind of chips. The architectural names are Llano for AMD and Sandy Bridge for Intel. Many reviews state that AMD's CPU cores are slower than Intel's ones but their GPU is faster. Latest Intel APUs, based on Skylake microarchitecture, integrate a more powerful GPU unit with up to 72 cores, Iris Pro Graphics 580. Which combination is better is not an easy question to answer. It must be backed by specific benchmarks, or better, the real application that we want it to run.

Over time the use of GPUs has passed from odd to common in our systems. Actually, several time consuming processes have been parallelized inside our operating systems such as web page rendering. However, the speedup that graphics hardware can bring to us is not free, as each application to be accelerated must be rewritten to take full advantage of the GPU.

As CPU makers did some years ago, passing from single core to symmetric multiprocessing system (SMP), and more recently to multi-cores, GPU makers are following the same trend. We can connect more than one graphic card to our computer to improve its GPU capacities or buy a card with 2 graphic chips inside. GPUs are so much powerful than CPUs that even a small cluster of a few GPUs can be faster than classic, and much more expensive, big cluster of computers. First cluster of this kind appear in the scientific literature in 2004 [12] with big success. Nowadays, more and more people build small GPU clusters with a couple of mighty graphic cards just to game. Connecting 2, 3 or 4 graphics card is called CrossFire by AMD and Scalable Link Interface (SLI) by NVIDIA.

Most information shown in this and next sections is available from the websites of the respective manufacturer: AMD [13], Intel [14] and NVIDIA [10].

GPUs Programming

Programming Model

The way GPUs can be exploited is deeply rooted on its hardware. There exists several APIs: every company has a proprietary one tied to their respective products. This way AMD started with Close to Metal and NVIDIA with CUDA. Over time another standard appear, OpenCL [15].

With respect to the programming tools available for developers, most the Application Program Interfaces (APIs) are based on C-like languages, but having some restrictions to improve the parallel throughput, such as no recursion at all or limited pointers usage. A popular option is the open source compiler LLVM [16] from the University of Illinois.

From 2003 the two main GPU developers, ATI an NVIDIA, started selling hardware solutions that need to be programmed with proprietary APIs. Despite previous work, the first widely supported GPUs were DX10 generation GeForce 8 series from NVIDIA, using the more mature CUDA API. On the other hand, the Radeon series from ATI, were programmed with the Close To Metal API.

Paying attention to operating system vendors, there were efforts in the same direction. Some people at Apple betted on the potential of GPUs and started developing an open API, latter known as OpenCL. In the same time, Microsoft created the DirectCompute API for Windows.

OpenCL aimed to became the OpenGL of heterogeneous computing for parallel applications. It is a cross-platform API with a broad and inclusive approach to parallelism, both in software and in hardware. While explicitly targeting GPUs, it also considers multi-core CPUs and FPGAs. The applications are portable across different hardware platforms, varying performance while keeping functionality and correctness. The first software implementations date back to 2009.

Most companies support OpenCL across their products. Apart from AMD and NVIDIA we can use it on graphic hardware from S3 and VIA. Also IBM has a version of OpenCL for PowerPC and CELL processors. Intel started to offer support from the APU architecture Ivy Bridge and GPUs. Embedded world is also interested in OpenCL. Imagination Technologies offer support for the SGX545 graphics core. As does Samsung with their ARM based microprocessors.

To make life easier for potential users many libraries are appearing. One significant example are STL like extensions for the C++ language. In [17] we can read a comparison of several modern C++ libraries providing high-level interfaces for programming multi- and many-core architectures on top of CUDA and OpenCL.

Execution Model

OpenCL, DirectCompute and CUDA are APIs designed for heterogeneous computing with both a host CPU and an optional GPU device. The applications have serial portions, that are executed on the host CPU, and parallel portions, known as *kernels*. The parallel kernels may execute on an OpenCL compatible device, CPU or GPU, but synchronization is enforced between kernels and serial code. OpenCL is distinctly intended to handle both task and data parallel workloads, while CUDA and DirectCompute are primarily focused on data parallelism.

A kernel applies a single stream of instructions to vast quantities of data that are organized as a 1-3 dimensional array (see Figures 2 and 3). Each piece of data is known as a work-item in OpenCL terminology, and kernels may have hundreds or thousands of work-items. The kernel itself is organized into many work-groups that are relatively limited in size; for example a kernel could have 32K work-items, but 64 work-groups of 512 items each.

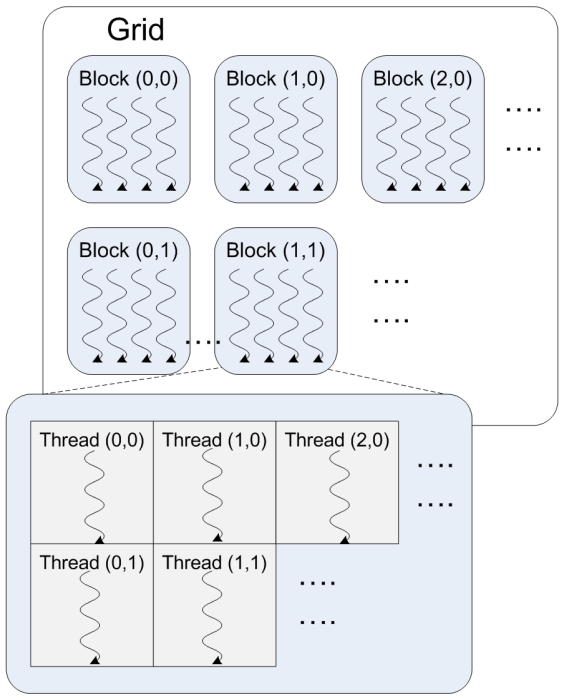

Figure 2. Hierarchy of computing structure in a GPU.

Unlike traditional computation, arbitrary communication within a kernel is strongly limited. However, communication and synchronization is generally allowed locally within a *work-group*. So work-groups serve two purposes: first, they break up a kernel into manageable chunks, and second, they define a limited scope for communication.

Memory Model

The memory model defines how data is stored and communicated within a device and between the device and the CPU. The memory model can be seen in Figure 4. DirectCompute, CUDA and OpenCL share the same four memory types but employ different terminology:

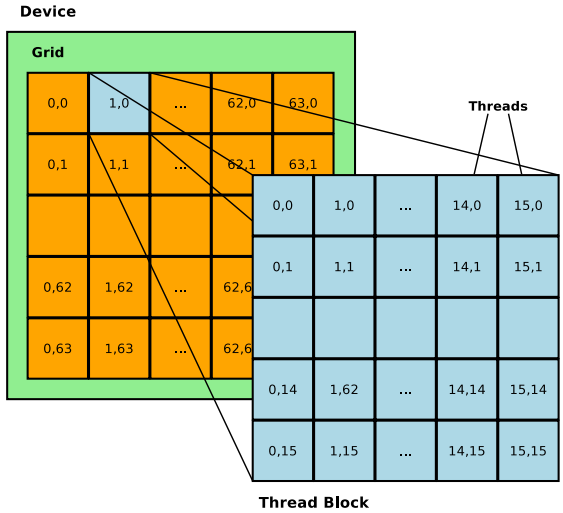

Figure 3. Execution model: Each piece of data is a work-item (thread); a kernel has thousands of work-items and is organized into many work-groups (thread blocks); each work-group process many work-items.

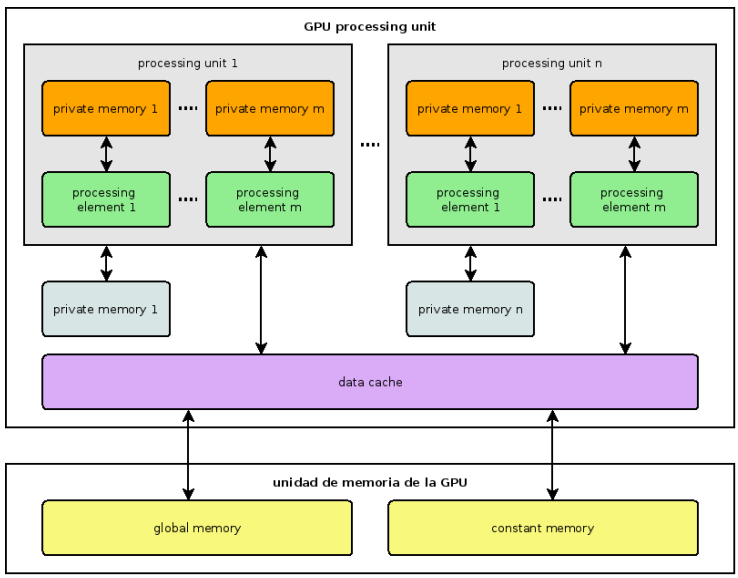

Figure 4. Memory model defines how the data is stored and communicated between CPU and GPU. Global memory is RW for both CPU and work-items; constant memory is RW for CPU and RO for work-items; private memory is RW for a single work-item; local memory is RW for a work-group.

- Global memory: it is available for both read and write access to any work-item and the host CPU.
- Constant memory: is a read-only region for work-items on the GPU device, but the host CPU has full read and write access. Since the region is read-only, it is freely accessible to any work-item.
- Private memory: is accessible to a single work-item for reads and writes and inaccessible for the CPU host. The vast majority of computation is done using private memory, thus in many ways it is the most critical term of performance.
- Local memory: is accessible to a single work-group for reads and writes and is inaccessible for the CPU host. It is intended for shared variables and communication between work-items and is shared between a limited number of work-items.

Unfortunately many times we have to pass data from CPU to GPU before a serious calculation can be done. Also after that, CPU must gather results from the GPU. Too much data passing can ruin speedups so they should be minimized if it can't be avoided at all.

These problem have been with us because current CPUs and GPUs have been designed as separate processing elements and do not work together efficiently. At least until last APUs from AMD. New AMD A7 series uses what they call Heterogeneous System Architecture (HSA). HSA seamlessly shares data between CPU and GPU, without memory copies or cache flushes with minimal dispatch overhead. Both components can share a common memory architecture.

Different EAs adapt to the limitations of the memory model with different degrees of success. Talking about population size, the smaller the the better. The variability is huge between different kinds. Also when possible, many small islands is preferable to a big unique population. The same is valid for any aspect of the algorithm which demands a big quantity of memory.

Frameworks used with EAs

In this section we cite some relevant material not int the EA field but involved with the programming frameworks used to implement EAs in a more convenient way. Most EC metaheuristics can be implemented without effort by using any of the existing frameworks.

The most complete and comprehensive review of these frameworks is from Parejo [18]. It is a comparative study of metaheuristic optimization frameworks. As criteria for comparison, a set of 271 features grouped in 30 characteristics and 6 areas has been selected. These features include the different metaheuristic techniques covered, mechanisms for solution encoding, constraint handling, neighborhood specification, hybridization, parallel and distributed computation, software engineering best practices, documentation and user interface. The difference is that this survey just includes frameworks with GPGPU support only for evolutionary computation metaheuristics.

PUGACE[19] is a framework in CUDA for cellular GAs (cGAs), but only for linear neighborhood structures. The architecture of this framework is presented and experiments with the QAP problem are performed. Fitness evaluation uses a thread for each chromosome. Speedups for several problem instances varies from $\times 15$ to $\times 18$. The execution platform was a Pentium Dual Core at 2.5Ghz and an nVidia GeForce 98000 GTX+.

Proposed Taxonomy

This section proposes a taxonomy to distinguish between the different distribution methods applied in the literature. It is inspired by the classical models used in the distribution of EAs. There are many different bioinspired approaches included in the bibliography, but we are mainly focused on Parallel Evolutionary Computation (EC). That is the most extended metaheuristic to address many types of large or complex problems, which require some kind of parallelization to be solved with enough quality and on a reasonable quantity of time.

Alba[20] reviewed and surveyed parallel metaheuristics on EC (Evolutionary Computation). They identified the majority of paradigms with respect to parallel/distributed Evolutionary Algorithms (EAs), according to Flynn's taxonomy, under the Multiple Instruction Multiple Data (MIMD) category. This argument has been fairly valid during the last two decades because the most dominant platform for running parallel/distributed EAs were distributed memory architectures, like clusters. However fine-grained EAs deployed on massive parallel processors (MPPs) are resurfacing, due to GPUs architecture gives low cost support for them.

The parallel EAs community has a wide legacy with MIMD architectures and a very little contribution for SIMD systems. This comes in part due to the dominance of MIMD architectures as compared to SIMD ones.

Alba classifies the main parallel metaheuristics models as follows:

- Parallel Genetic Algorithms[21]
- Parallel Genetic Programming[22].
- Parallel Evolution Strategies[23].
- Parallel Ant Colony Algorithms S[24].
- Parallel Estimation of Distribution Algorithms[25].
- Parallel Scatter Search[26].
- Parallel Variable Neighborhood Search[27].
- Parallel Simulated Annealing[28] .
- Parallel Tabu Search[29].
- Parallel Greedy Randomized Adaptive Search Procedures[30].

- Parallel Hybrid Metaheuristics[31].
- Parallel MultiObjective Optimization[32].
- Parallel Heterogeneous Metaheuristics[33].

Nevertheless, when the research community uses GPGPU, all the EC approaches are parallel, so we revise the bibliography taking in mind how an algorithm has been parallelized using that paradigm, so, for this survey, the specific metaheuristic used in every paper is being placed in a secondary position. For that reason, we focus the rest of the survey on the different ways for implementation of Parallel Evolutionary Algorithms (PEAs) as Zhang and Zhenming propose in[34]: master-slave model[35], fine-grained model[36], coarse-grained model[37 38] or hierarchical models[39] that use two or more of the previous parallel approaches in an hierarchical way.

Taxonomy

The proposed taxonomy is based on the classical approach followed by other authors, but it adds two new groups that we consider important: the hierarchical and the non-standard models.

Master-slave approaches Master-slave Evolutionary Algorithms traditionally solve problems involving computationally expensive fitness functions, where the master node runs the entire algorithm while the slaves execute the fitness evaluation of the individuals. Using this approach, the evaluation of the fitness function is not sequential, but in parallel, so the master-slave version is more efficient because the evaluations usually are the more expensive portion of the total run time.

Fine-grained approaches Cellular Evolutionary Algorithms (CEAs) or Fine-grained Evolutionary Algorithms have not had as much impact as other types of PEAs. The main reason is that they adapt themselves very well to the especial hardware architecture, with shared memory, but very bad to other distributed memory architectures. On the other hand, parallel distributed memory architectures supply all loosely coupled algorithms and the fine-grained EAs are not included into this category. The main reason behind that is the high cost of building massively parallel architectures which normally attracted fewer researchers to work on fine-grained EAs. However, a review of the trends in parallel architectures foresees a strong increase in fine-grained EAs face to face other EA categories. This is due to three reasons:

- Growing trend of massive number of processors on chip or card.
- The high inter-processors speed which is a major factor affecting the efficiency of fine-grained EAs.
- The huge low cost of these architectures which attract a wide base of researchers and developers.

For this category of algorithms, each individual is a parent at the beginning of the algorithm and it looks for the second parent by selection in a neighborhood. As a result CEAs provide automatic niching effect, avoiding an early convergence.

Coarse-grained approaches These approaches work with a set of sub-populations (normally a part of the whole population) distributed in different nodes, being every node devoted to work with just one of them, so they can work in parallel independently. Each of these sub-populations is known as 'island'. In the standard model, after a number of generations (migration rate) one or more individuals, normally the best, are interchanged (migrated) between islands, which are connected following a specific neighborhood topology.

Hierarchical models This hybrid model just utilizes two or more of the master-slave, coarse-grained and fine-grained approaches in an hierarchical method. At the higher level, an island model algorithm runs while at the lower level, the demes, or sub-populations, are running in parallel, fine-grained, master-slave or even another island model with high migration rates. Hybrid models are not the most common in traditional implementation of EAs due to:

- The need for additional parameters related to the more complex topology structure.
- The need for 'quite rare' hierarchical parallel architectures to host hybrid algorithms.
- The high complexity of programming such models.

Nevertheless, for the GPUs, hybrid EAs are a perfect candidate to exploit the hierarchical memory and flexible block sizing, by means of well structured populations.

Non-standard approaches By the end of the work we have included a subsection for non-standard approaches[39] including papers which may not be included in any of the previous ones because they mapped the algorithms and the problem they solve in a special way that cannot be considered as fine grained, coarse grained or master-slave ones.

Literature analysis and review

In this section we present a quantitative analysis of published papers on EAs on GPUs. Then, the most relevant ones are classified following the previously described taxonomy.

General analysis

Firstly, a deep analysis of the existing related literature has been conducted. In order to conduct this analysis we performed the query "*(GPGPU OR GPU) AND (GENETIC OR EVOLUTIONARY)*" in

the Web Of Science (WoS)[40] database, referred to any part of the publications, and just enclosed into *Computer Science* research area. We got 237 references, and we eliminated those references unrelated to the scope of this paper (for example, those whose topic is related with genetic engineering and not with genetic algorithms). After that removal, we kept 162 results, which we read in order to extract its features and useful information for this survey.

Looking at the publication rate per year (inside this scope), it can be noticed in Figure 5, the publication figures has been increased exponentially every year, starting from 2004 (according to the results), being nowadays a field very attractive for researchers and very prolific.

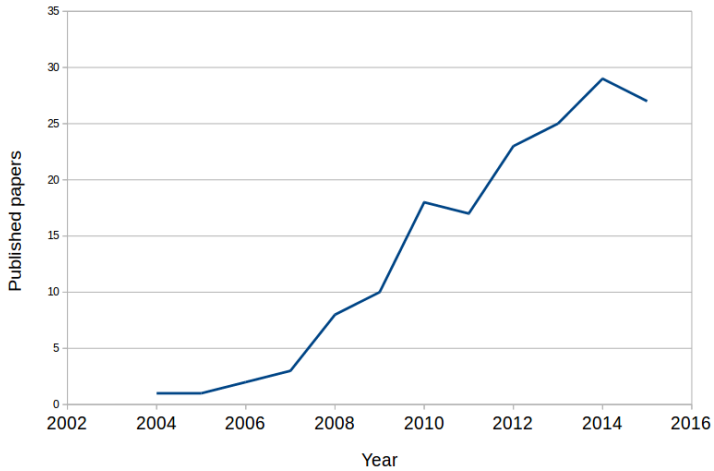

Figure 5. Papers published per year inside the scope analyzed here (EAs on GPGPUs) according to Web of Science database.

The decrease in 2015-2016 is due to recent papers have not been indexed yet on WoS.

With respect to the type of publication, as expected, it can be seen in Figure 6 that conferences and journals share almost all the cake, being slightly higher the figures for conference papers, as it is obviously easier to get published there. However, this factor should be taken just as a reference, since not all the conferences and journals are indexed in WoS, but mainly relevant ones.

Regarding the type of EA implemented or studied in those papers, Figure 7 shows the percentages of the different approaches. The most prolific ones are GAs, since it is the classic optimization method inside EAs. It is easy to implement and normally yields very good results. GP and DE (Differential Evolution) are more specific and thus, less extended or used. 'Others' refers to EAs such as EDA (Estimation of Distribution Algorithm), SGS (Systolic Genetic Search) or MA (Memetic Algorithms), to cite a few.

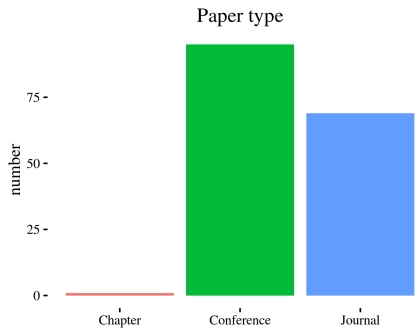

Figure 6. Type of publication, being Conference paper, Journal Paper, or Book Chapter.

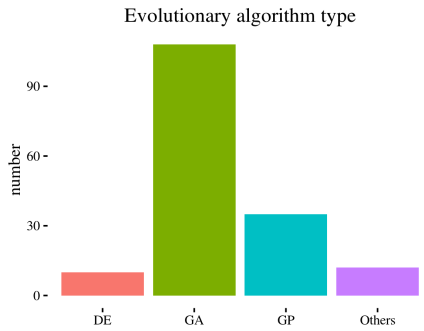

Figure 7. Type of Evolutionary Algorithm implemented in the analyzed publications.

Finally, the majority of the analysed publications deal with solving real problems as it is presented in Figure 8. This means that the parallelization using GPUs is very effective and can be used in relevant and very hard problems, as real world ones normally are.

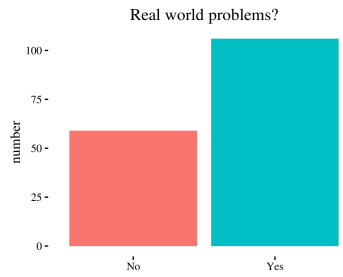

Figure 8. Portion of the analyzed publications which address real problems.

Once we have analyzed the whole literature, in the following sections we will go through each of the different groups of the proposed taxonomy. Then, for every one the most representative or relevant works (mainly according to their number of citations) are reviewed and commented.

Master-slave Approaches

As a GPGPU implementation, the GPU is responsible for fitness evaluation but it does not get involved with other phases of the algorithm such as crossover, selection or even population ranking for the next generation. In this approach, every generation the individuals are copied to the global memory of the GPU, so every single processor gets one or more individuals from global memory and puts them into the shared memory, in order to decrease global memory overhead. After that, each thread evaluates it or them, and stores the result into the global memory again. Usually each single core evaluates one individual. The programmer has to minimize the number of global memory accesses, as well as to minimize data transfers between CPU and GPU. Some additional problems might be faced, like for example, individuals that require additional data for being evaluated, such as a distance matrix or a data set. This additional information might be copied from CPU to GPU once at the beginning of the process, so the threads might access this data for evaluation minimizing CPU/GPU traffic.

One of the first proposals for parallel Evolutionary Computation using a GPU was done by Wong et al.[35]. They presented an Evolutionary Programming (EP) algorithm for solving five simple test functions, called Fast Evolutionary Programming (FEP). In this master-slave approach, some actions such as the main loop of the algorithm were executed in the CPU, while evaluation and mutation were run within the GPU device because both steps do not need of external information exchange. In this case, the reproduction process implies interaction among, at least, two individuals so the authors eliminated this step in the algorithm. A maximum speedup of $\times 5.02$ was obtained when the population size increases. This is the most common organization in GPU implementations, since no interaction among individuals is required during the evaluation, so this process can be fully parallelized. In that paper the individuals was real-coded and the genomes were mapped into the texture memory. Each GPU thread evaluated one individual and returned its fitness at the end of the parallel process to the CPU, for the next evolution cycle.

A Genetic Programming (GP) method proposed by Harding and Banzhaf[41] uses the GPU only for performing the evaluation, while the rest of the steps of the algorithm are run on the CPU. The authors tested real-coded expressions in up to 10000 nodes, boolean expressions in up to 1500 nodes, and some real world problem where they evaluate expressions in up to 10000 nodes. In some cases, the results yielded speedup of thousand times.

Zhang et al.[34] adapt different EAs to a GPU using CUDA. The authors implemented an hierarchical parallel genetic algorithm using a deme model at high level, and a master-slave schema at low level. In this implementation, the CPU initializes the populations and distributes them to thread blocks in shared memory. Then, GPU threads within each block run a GA independently, selection, crossover, mutation and evaluation, and migrates individuals to other thread blocks in its neighborhood. In this case, no speedup results were reported.

Van Luong et al.[42] published in 2010 a methodology for mapping the search space onto the GPU memory hierarchy in three levels: the distribution of the search process among the CPU and GPU (1), the mapping of the neighborhood in the GPU threads (2), and the effective usage of the texture memory in the context of hybrid EAs (3). Experiments to solve the QAP problem with CUDA were presented, where the evolutionary process was performed in the CPU and the generation of the Local Search neighborhood are performed on parallel in the GPU, achieving an acceleration of $\times 14.6$ times faster. Hardware used was Core 2 Duo 2,67 Ghz and Nvidia GTX 280. As it can be seen this work also showed the benefits of using a conjunction of CPU and GPU, being the GPU used only for intensive calculations.

There are other papers related with the previous approach, like the one by Tsutsui et al.[43]. This work uses a master-slave approach with an Ant Colony Optimization (ACO) algorithm[44] and Tabu Search[45]. Tsutsui uses an Intel Core i7 965 (3.2 GHz) processor and a single NVIDIA GeForce GTX480 GPU. They compare CPU and GPU implementations with and the results showed that GPU computation with MATA, an efficient method for thread assignment cost which they call Move-Cost Adjusted Thread Assignment, showed a promising speedup compared to computation with CPU.

There are papers related with other subjects like[46] which studies the impact of using different representations for binary problems in GPUs to evaluate GAs: boolean data type versus packing multiple bits in a non-boolean data type. The execution platform was a PC with Quad Core Intel Xeon E5530 processor at 2,4 Ghz and a Tesla C1060 with 240 CUDA cores. The CPU only calculates the initial population and a matrix of random number to be used in crossover and mutation by the GPU. The authors review the problem of the data types when GPGPU is used. Several data types of 8, 16, 32, and 64 bits are compared in CPU and GPU versions. Results shows that packing in 32 bits data types achieve speedup values of up to $\times 100$ compared with boolean data types, specially when the size of the instances increases. This make sense because the Tesla C1060 are equipped with 32-bit integer ALUs.

Cano et al.[47] described a massively parallel evaluation model using a Genetic Programming algorithm for evolving rules for dataset classification. They copied the dataset to the GPU global memory, and evaluated the rules (individuals) for each instance of the dataset in parallel (match kernel). At the end, each successful match was reduced getting each individual fitness (reduction kernel). The implementation, based on CUDA, speedups the fitness calculation phase and greatly reduce the computation time. Results were compared using one, two and four CPU threads and the combination of

one or two GPUs of different features (285 cores for GPUa -NVIDIA GeForce 285 GTX- or 480 cores for GPUb -NVIDIA GeForce 480 GTX-). They test three classification algorithms and the results were not very significant for small datasets, but they increased the speedup with large datasets until $\times 820$ using two GPUs with 480 cores each one.

The work by Chitty[48] uses a Master-Slave architecture for evaluating at time the population for four GP problems. The results were achieved using an NVidia GeForce Kepler 670 GTX graphics card and compared with a previous one-dimensional stack approach for the same problems. The author fit the algorithm until to observe a peak computational speed of over 55 billion Genetic Programming Operations per second a twofold improvement over the previous one-dimensional stack approach.

Recent works are currently being applied to real-world problems. Jaros et al.[49] use GPUs to lower the run time and improve the quality of the solutions in the problem of wormhole switching in collective communications. The start from an evolutionary tool capable of finding optimal communication schedules for various communication patterns on wide range of interconnection networks topologies of up to 256 nodes. The tool consumes tens of hours so they decide to improve it. As the fitness evaluation was responsible for the 93% of the execution time this was the only part moved to the GPU. In a first step only one GPU was used with speedups up to $\times 5$. After that they made a new implementation capable of using 8 GPUs and 30 times faster than the original.

In[50] the authors use a CPU-GPU architecture for stock market trading. The proposed architecture offers the benefices of GPU distribution for stock market researchers, rather for computer architecture experts. The authors used Jacket[51], a software platform for the rapid development of GPGPU computing applications within the MATLAB computing environment, C, and C++. However, this framework is no longer available *. Steps and guidelines to migrate from CPU code to GPU code are explained, what is a great contribution, because usually, the authors do not include this information in the papers. The algorithm run in the CPU, but all *for* loops in selection, evaluation, crossover and mutation are translated to Jacket's *gfor* to be parallelized in the GPU. Three different CPU configurations are used for the experiments: Pentium 4, Pentium SU41000 and Intel Core i7-860. The former is also combined with an nVidia 460GTX for GPU experiments. Different population size are also tested. The time is reduced to 65% in comparison with the CPU version. Other conclusions are obtained: speedups with high number of individuals, rather than increasing the number of evaluations; and time reduction for tournament selection over roulette-wheel selection.

Stock market trading analysis haas also been recently addressed by using GP in GPUs. The algorithm described in the work by Sungjoo et al.[52] is used to solve the problem of knowledge discovery in stock market time series, i.e. finding the so-called precursor patterns, which model events occurred in the time

*http://blog.accelereyes.com/blog/2012/12/12/exciting-updates-from-accelereyes/

series. Thus, every individual (tree) in the GP algorithm is a pattern, which must be contrasted against the whole dataset to find out if it is a precursor. This is what the evaluation function does. The time series is divided into different subsets, and stored in several GPUs. The data is then copied once and just the new individuals to evaluate and the partial results of the evaluation must be transferred and updated. The run is conducted for 50 generations and with 500 individuals of a maximum depth of 3 levels, using a i7-3820 CPU and 8 CUDA GPUs (GeForce GTX 690). The authors considered 512 threads per block, 260 blocks per grid and just one grid per GPU. The comparison in performance yields a reduction of 56 times for just one GPU to 277 using all of them. This improvement is got using local memory allocation, using the shared memory of blocks instead yields a smaller improvement of around 188 times faster in the best case. Regarding the influence of individuals' tree size (number of nodes) in the performance, the authors show that the running time grows for just one GPU, but when more than four are used, this growing is minimum and could be assumed in order to improve the quality of the results.

Fine-grained Approaches

Researchers implement fine-grained algorithms using GPGPU, making every scalar processor (SP) to evolve an individual. This individual interacts with other SPs that belongs to the same Streaming Multiprocessor (SM) to perform the basic operations of the algorithm like crossover or mutation. In this approach a big part of the algorithm runs within the GPU and not only the evaluation phase like in previous section (). The GPU architecture assists the neighborhoods emerge since each group of SMs have a shared memory, where SPs access without penalty. The exchange of information between individuals through this shared memory is inexpensive. However, programmers have to be careful with the information exchange between individuals in different SMs.

The second problem of this approach is the random number generator. The CPU can easily generate random numbers, but the GPU is not designed for this task. So researchers have to think about how to solve this problem. Usually the algorithm's implementation generate at the beginning a big set of random numbers and they are copied to the GPU as a list. After that, the GPU uses the random list when needed and when it need. This approach saves time for random generation but the list of random numbers has to be limited. Moreover, all SPs could use the random numbers, so the list must be available for every thread.

Yu et al.[53] implemented the first real cellular EA using GPU, for solving the Colville problem[54] in 2005. They place the population in a toroidal 2D grid and use the classical Von Newmann neighborhood structure with five cells. They store chromosomes and their fitness values in texture memory on the graphic card, and both, fitness evaluation and genetic operations, are implemented entirely with fragment programs executed in parallel on GPU. Real-coded individuals was represented as a set of 2D texture

maps. $BLX-\alpha$ crossover and non-uniform mutation was developed as tiny programs on every pixel at each step in a SIMD-like fashion. They solved some optimization problems and reached a speedup of $\times 15$ with a population of 512x512 individuals. They store a set of random numbers at the beginning of the evolution process to solve the random number generation problem when using GPU processors.

In 2006,[55] proposed a parallel hybrid GA (HGA) where the whole evolutionary process run on the GPU, and only the random number generation is done in CPU. Every individual is assigned to a GPU thread, and each one selects probabilistically an individual in its neighborhood to mate with it. Just one offspring individual is generated each time, and it replaces the old one in that GPU thread. The authors compare their implementation with a standard GA run in a CPU and the FEP[35] algorithm. Using a new pseudo-deterministic selection method, the amount of random numbers transferred from the CPU is reduced. HGA reaches speedup of $\times 5.30$ when compared against the sequential version. In 2009 Wong et al.[56] provide implementation details for fine-grained evolutionary algorithms.

Liu and Luo[57] implemented a cellular algorithm on GPU to solve three different satisfiability problems (SAT) using a greedy local search (GSAT)[58] and a cellular GA (cGA). They saved local minimums using a random walk strategy, jumping to other search space location. The cellular GA adopts a 2D toroidal grid, using the Moore neighborhood, stored on texture GPU memory. This implementation generates the random numbers in the GPU (using a generated seed on the CPU at the beginning of the process). They carried out the experiments using two GSAT implementations, CGSAT (with crossover and without mutation) and PGSAT (without crossover and with mutation) running them on CPU and GPU with different population sizes. A great time reduction was reached using the GPU parallelization approach (from 95ms to 18 ms for CGSAT and from 464ms to 77 ms for PGSAT).

Li et al.[36] proposed a cellular algorithm on GPU for solving some common approximation functions. The authors reported experiments using big populations (up to 10000 individuals) reaching speedups of $\times 73.6$ for some implementations. The novelty of this paper is the bit dataset because it is not easy to deal with bit datasets using a GPU due to the memory size restrictions. The GPUs does not support binary operations, but in this paper the authors dealt with binary individuals of 24 bits and simulated bit-operator by judging each bit of a binary value is 1 or 0. They faced a general problem for all GPGPU community with bit operations. They proposed to use random-textures for random numbers generation, but the random numbers were generated on CPU previously.

In[59] the authors proposed an Fine-grained parallel immune algorithm (FGIA) which is an Artificial Immune System combined with evolutionary algorithm to generate several optimization algorithms. They solve three medium-size instances of the Travelling Salesman problem (TSP) using a fine-grained parallel algorithm with CUDA C. The paper enlarges the population size of FGIA and maintains better population diversity. In conclusion, they included a sub-linear relation between the population size and the execution

time, which made the proposal very useful for solving difficult problems that require huge population sizes.

González et al.[60] use CUDA and store individuals and their fitness values in the GPU global memory. Both, fitness evaluation and genetic operators, run on GPU (no CPU is used). They use a pseudo random number generator provided by the SDK of CUDA named Merseinne Twister. Their experiments include some general discrete and continuous optimization problems, and they compare physical efficiency and numerical efficacy with respect to CPU implementation.

Depending on the problem size, fine-grained approaches may improve the efficiency with respect to the coarse-grained version. For example, in the work of Franco et al.[61], two different approaches for strategies for GPU implementations of the evaluation stage of evolutionary rule learning are analyzed and the types of problems where each method performs best have been identified. The coarse-grained implementation is more conservative and only parallelizes the evaluation of instances, while the fine-grained implementation expands the parallelism to the attribute dimension and is faster on data sets with 10 to 50 discrete attributes.

Coarse-grained Approaches (island model)

Coarse grained algorithms are the most common among parallel EAs. Generally, these algorithms require less tightly coupled parallel architectures, as compared to fine-grained. Coarse-grained EAs divide the main population into sub-populations (also known as islands) which evolve concurrently. This basic feature of coarse-grained EAs hits a physical limit of GPUs.

In order to run a coarse-grained EA using a GPU, several kernels are run simultaneously, where each kernel handles a sub-population. GPU architecture is not designed to work this way. This limitation of GPU would mean the conventional mechanics of coarse-grained EAs would need to be changed if GPU would be used.

With regard to this topology, one of the first island models on GPU approaches was published on the GPU competition of GECCO 2009[62]. It presented some technical details of an island model entirely hard-coded on GPU, with a ring-neighborhood topology. Nevertheless, the evolutionary operators implemented on GPU were only specific to the GECCO competition, and the validity of the experiments is not clear, since the approach just worked on a small number of problems.

Tsutsui and Noriyuki[63] propose run a coarse-grained GA on GPU to solve the QAP problem using CUDA. This is one of the hardest optimization problems in permutation domains. Their model generates the initial population on CPU and copied it to the GPU VRAM; then, each subpopulation in a GPU (NVIDIA GeForce GTX285) is evolved. At some generations, individuals in subpopulations are shuffled

through the GPU VRAM. Results show a speedup from $\times 3$ to $\times 12$ (using eight QAP instances), in the comparison with an Intel i7 965 processor.

The model by [64] is based on a re-design of the island model. Three different schemes are proposed: The first one implements a coarse-grained EA using a master-slave model to run the evaluation step on GPU. The second one distributes the EA population on GPUs, while the third proposal extends the second one using fast on-chip memory. Second and third approaches reduce the CPU-GPU memory latency, although their parameters (number of islands, migration topology, frequency, and number of migrants) must be adapted to the GPU features. Sequential and parallel implementations are compared, obtaining a speedup of $\times 1757$ using the third approach.

Pospichal et al. [38;65] propose a parallel GA with island model running on GPU. The authors map threads to individuals, thus, threads-individuals can be synchronized easily in order to maintain data consistency, and on-chip hardware scheduler can swiftly swap existing islands between multiprocessors to hide memory latency. Fast, shared memory within the multiprocessor is used to maintain populations. Since the population size is limited to 16KB per island on most GPUs, if the population is larger, slower main memory has to be used. The migration process is based on an asynchronous unidirectional ring that requires an inter-island communication (slower main memory has to be used). The authors report speedups up to $\times 7000$ times higher on GPU compared to CPU sequential version of the algorithm.

Hierarchical Models

The design and implementation of a hybrid EA with local search to solve MAX-SAT over GPUs was thoroughly discussed in [66]. Manuwar et al. uses a hierarchical algorithm of 2D structured sub-populations arranged as islands in a 2D grid. Therefore, every individual has 4 neighboring individuals (north, south, east and west) and each sub-population has 4 neighboring sub-populations (north, south, east and west). Instead of using a conventional algorithm for migration between the sub-populations, they introduced a new technique called diffusion. Diffusion is more suitable for implementation of cGAs based pGA over a GPU. In the proposed implementation, the host processor (CPU) acts as a controller while an NVIDIA Tesla C1060 GPU provides the required computational resources. Configuration, memory allocation and initialization are performed over the host processor. After the initialization stage, data is transferred to the device and the code enters a loop. The loop keeps on repeating until the maximum number of generation criteria is satisfied. Results were collected over a system with NVIDIA Tesla C1060 GPU mounted on a motherboard with Intel Core i7 920@2.67GHz as the host CPU. C1060 have 4GB of device memory, 30 streaming multiprocessors, and the total number of processing cores is 240. The maximum amount of shared memory per block is 16KB and clock rate is 1.30GHz. They compare the results of the algorithms over NVIDIA with optimized for local search, mutation, recombination,

selection and diffusion (migration) with different implementations using serial implementation, OpenMP implementation over Intel and over Ultra Spark architectures. The found that the maximum speedup is for larger problems, and it is up to $\times 25$ if compared the serial implementation over Intel Core 2 Duo 3.3GHz with the NVIDIA implementation.

Diego et al.[67] proposed a parallel strategy for solving the Capacitated Vehicle Routing Problem (CVRP) by means of an ACO algorithm. They combined the CPU processing, random number generation and centralized pheromone information dealing, with the GPU parallel capabilities: initialization of trails, build solutions, choosing of the best solution, and pheromone evaporation. The experiments were conducted on a GeForce GTX460 in a Pentium Dual Core 2.7GHz with 2GB RAM, using CUDA. The results showed a speedup of $\times 12$ in the best case.

Delevacq et al.[68] tests different parallel strategies for the ACO metaheuristic on a NVIDIA Fermi C2050 GPU, on a four-core Xeon E5640 @2.6 GHz with 24GB RAM. The Max-Min Ant System is considered for solving the Travelling Salesman Problem (TSP) for problem sizes of up to 2100 cities. Authors proposed multi-ant (one colony) and multi-colony approaches implemented with CUDA. They distributed the solutions on the GPU processing elements: an ant per thread, and an ant per block respectively. The common structures, pheromone matrix, distances and candidate lists, are stored in shared memory, meanwhile the CPU is used for a preliminary set of random numbers generation and centralized control tasks. For the multi-colony algorithms, a set of ants is assigned to every processing element. This time, almost the whole algorithm is run in the GPU. They obtained speedups of up to $\times 19$ in the multi-ant, and up to 8 in the multi-colony approaches, having similar solution quality than the original sequential approach.

Cecilia et al.[69] proposed three different techniques for improving the classical ACO algorithms parallelization on GPUs: a data parallelism scheme for tour construction on GPU, a GPU-based pheromone updating stage, and a roulette wheel implementation on GPU, called I-Roulette. They introduced the queen ants, associated with CUDA blocks, and the worker ants, associated to CUDA threads. The experiments were performed on a four-core Intel Xeon E5620 running at 2.4 GHz, with 16GB RAM, and a NVIDIA Tesla C2050 Fermi. The proposed techniques lead to obtain a speedup factor of up to $\times 20$ in comparison with the sequential implementation.

In[70] three cellular GA versions are compared: a CPU, a mono-GPU and a multi-GPU version, being the first work in use multi-GPU for cEAs. A 2D grid population is used and divided in two GPUs in the multi version. Selection, recombination, mutation and evaluation are performed in parallel. In the multi GPU case, a thread for each GPU is controlled by a CPU thread and borderline individuals of the sub-populations are exchanged. The speedup ranges from $\times 8$ to $\times 711$. However, there is not significantly difference between mono and multi GPU versions, probably due the overhead of the CPU. Hardware used is Intel Quad processor 2.67GHz and Nvidia GeForce GTX 285.

During 2012, Luong[71] implemented a GPGPU algorithm from the metaheuristic point of view. In the paper a guideline to exploit heterogeneous computing resources, including GPU and CPUs, for effective hybrid metaheuristics was proposed. The goal of this paper is not the evaluation of the fitness of some individuals, but the parallelization of metaheuristics which were not inherently parallel, like local search algorithms. Task distribution is clearly defined into the paper, the CPU manages the whole sequential Local Search process and the GPU is dedicated to the costly part i.e. the parallel generation and evaluation of the neighbor solutions.

Authors mapped the population to the GPU and applied the parallel metaheuristic to each individual using its neighborhood structure taking advantage of the blocks of threads present in the GPU architecture. The parallelization of these metaheuristics generated several different individuals that the GPU evaluated at the same time. The critical issue was finding efficient mappings between a GPU thread and a particular neighbor. Indeed, this step is crucial in the design of new large neighborhood local search algorithms for binary problems since it is clearly identified as the gateway between a GPU process and a candidate neighbor. They reviewed neighborhoods based on a Hamming distance of one, two and three, and applied their suggestions to Permuted Perceptron problem[72]. They considered a Tabu Search algorithm[73] using CUDA for each neigborhood with a configuration of Intel Core 2 Duo 2.67GHz with a NVIDIA GTX 280 card. The CUDA implementation increased the number of successful solutions drastically every instance of the problem. Regarding execution time, acceleration factors using GPU are very significant (from $\times 24.2$ to $\times 25.8$).

In 2014,[74] used a supercomputer platform to run a CPU+GPU high performance EA. 16 islands of 8 individuals were set as parameters. The experiments were carried in a combination of Intel Xeon X5650 @2.66GHz and Intel Xeon X7550 @2.0GHz processors, although the paper do not includes information about the graphic cards used. Several instances of heterogeneous scheduling problems were used to compared with other metaheuristics, obtaining better makespans.

Non-standard Approaches

Pospichal et al. in 2011 has presented several papers related with GPU devices. One of the last is[39] where they propose to use a GPU device for Grammatical Evolution, evolving complete programs in an arbitrary language using variable length binary strings. For this problem, every individual is a program, so it must be compiled and sent to the GPU for being evaluated every generation. However the most time-consuming part of this approach is to sent the individuals to the GPU and not the evaluation itself, so, the authors propose to evolve the whole Grammatical Evolution algorithm on the GPU with a special mapping function. They mapped one individual per block of threads and one thread is used to manage each individual. They compare the execution time of an Intel Core i7 with NVIDIA GTX 480

using CUDA (CPU implementation with C) an a Java library called GEVA[75], which is an interpreted language (CPU implementation using GEVA). The authors include results for both implementations (C and GEVA) and one GPU implementation, and they compare them with and without overhead for CPU-GPU communication. We include only the results with overhead, because the parallel version needs the CPU-GPU communication, so the authors might take care about it. The results of GPU implementation are in average 5.3 speedup than CPU with C implementation which is an expected result. The speedup grows until $\times 102.8$ when GPU implementation and CPU implementation with GEVA are analysed, but the authors do not include any standard deviation results, so we do not consider the last result as the most important of the paper. Thus, this paper proposes two levels of parallelism: 1) individuals are evaluated in parallel (by threads in the same block) and 2) data within individuals (genes, crossover points, mutations, fitness points, etc.) are maintained by parallel access as well (by block of threads).

Conclusions

Evolutionary Algorithms have been widely implemented and executed in traditional computers. They have also been usually parallelized in clusters or distributed-processor systems, in the past decades. However with the current technological enhancement of computer graphics, and thus, with the arise of powerful GPUs, a new way to execute this kind of algorithms has been opened.

This paper presents an introduction to the different existing GPU architectures, programming languages, and frameworks, used nowadays in the industry and in this scientific area.

Focusing in the latter, this work is also a survey of the usage of General-Purpose GPUs for running Evolutionary Computation approaches, implemented to solve large-scale or complex optimization problems. Thus, a number of research works available in the literature have been reviewed, including a description of a taxonomy to classify them.

In the majority of cases, speedups are attained in the comparison with traditional CPU sequential and parallel versions. Most authors agree that bottlenecks appear copying data from main memory to GPU device, so new hardware and software solutions are emerging trying to minimize the communication cost between GPUs and CPUs.

Acknowledgements

deliberately empty...

References

1. Bäck T. *Evolutionary algorithms in theory and practice*. Oxford University Press, 1996.

2. Goldberg DE. *Genetic Algorithms in search, optimization and machine learning*. Addison Wesley, 1989.
3. Koza JR. *Genetic Programming: On the programming of computers by means of natural selection*. Cambridge, MA: MIT Press, 1992.
4. Cantú-Paz E. *Efficient and Accurate Parallel Genetic Algorithms*. Norwell, MA, USA: Kluwer Academic Publishers, 2000. ISBN 0792372212.
5. Tomassini M. *Spatially Structured Evolutionary Algorithms: Artificial Evolution in Space and Time (Natural Computing Series)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. ISBN 3540241930.
6. Koomey JG, Berard S, Sanchez M et al. Implications of historical trends in the electrical efficiency of computing. *IEEE Annals of the History of Computing* 2011; 33: 46–54. DOI:http://doi.ieeecomputersociety.org/10.1109/MAHC.2010.28.
7. Esmaeilzadeh H, Blem E, StAmant R et al. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. pp. 365–376.
8. Thompson CJ, Hahn S and Oskin M. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*. MICRO 35, Los Alamitos, CA, USA: IEEE Computer Society Press. ISBN 0-7695-1859-1, pp. 306–317. URL http://portal.acm.org/citation.cfm?id=774861.774894.
9. Buck I, Foley T, Horn D et al. Brook for gpus: stream computing on graphics hardware. *ACM Trans Graph* 2004; 23: 777–786. DOI:http://doi.acm.org/10.1145/1015706.1015800. URL http://doi.acm.org/10.1145/1015706.1015800.
10. NVidia Corporation. NVIDIA Homepage. http://www.nvidia.com, 2014. Accessed: 20/01/2014.
11. AnandTech. GPU 2015 Benchmarks, 2015. URL http://www.anandtech.com/bench/GPU15/1248.
12. Fan Z, Qiu F, Kaufman A et al. Gpu cluster for high performance computing. *SC Conference* 2004; 0: 47. DOI:http://doi.ieeecomputersociety.org/10.1109/SC.2004.26.
13. AMD Corporation. AMD Homepage. http://www.amd.com, 2014. Accessed: 20/01/2014.
14. Intel Corporation. Intel Homepage. http://www.intel.com, 2014. Accessed: 20/01/2014.
15. Munshi A. The OpenCL Specification. http://www.khronos.org/opencl, 2014. Accessed: 20/01/2014.
16. Illinois U. The LLVM Compiler Infrastructure, 2011.
17. Demidov D, Ahnert K, Rupp K et al. Programming cuda and opencl: A case study using modern c++ libraries. *SIAM Journal on Scientific Computing* 2013; 35(5): C453–C472. DOI:10.1137/120903683. URL http://epubs.siam.org/doi/abs/10.1137/120903683. http://epubs.siam.org/doi/pdf/10.1137/120903683.
18. Parejo JA, Ruiz-Cortés A, Lozano S et al. Metaheuristic optimization frameworks: a survey and benchmarking. *Soft Computing* 2012; 16: 527–561. DOI:10.1007/s00500-011-0754-8. URL http://dx.doi.org/10.1007/s00500-011-0754-8.

19. Soca N, Blengio J, Pedemonte M et al. Pugace, a cellular evolutionary algorithm framework on gpus. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*. New York, NY, USA: IEEE, pp. 1 –8. DOI: 10.1109/CEC.2010.5586286.
20. Alba E. *Parallel Metaheuristics: A New Class of Algorithms*. The Atrium Southern Gate, Chichester West Sussex PO19 8SQ England: Wiley-Interscience, 2005. ISBN 0471678066.
21. Cantú-Paz E. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis* 1998; 10(2): 141–171.
22. Fernández F, Tomassini M and Vanneschi L. An empirical study of multipopulation genetic programming. *Genetic Programming and Evolvable Machines* 2003; 4: 21–51. URL http://dx.doi.org/10.1023/A:1021873026259. 10.1023/A:1021873026259.
23. Rudolph G. Parallel approaches to stochastic global optimization. In *In Parallel Computing: From Theory to Sound Practice, W. Joosen and E. Milgrom, Eds., IOS*. Dortmund: IOS Press, pp. 256–267.
24. Janson S, Merkle D and Middendorf M. Parallel ant colony algorithms. Technical report, Parallel Metaheuristics, Wiley Book Series on Parallel and Distributed Computing, -, 2005.
25. Madera J, Alba E and Ochoa A. A parallel island model for estimation of distribution algorithms. In Lozano J, Larrañaga P, Inza I et al. (eds.) *Towards a New Evolutionary Computation*, *Studies in Fuzziness and Soft Computing*, volume 192. -: Springer Berlin / Heidelberg. ISBN 978-3-540-29006-3, 2006. pp. 159–186.
26. García-López F, Melián-Batista B, Moreno-Pérez JA et al. Parallelization of the scatter search for the p-median problem. *Parallel Computing* 2003; 29(5): 575 – 589. DOI:10.1016/S0167-8191(03)00043-7. URL http://www.sciencedirect.com/science/article/pii/S0167819103000437.
27. García-lópez F, Melián-batista B, Moreno-pérez JA et al. The parallel variable neighborhood search for the p-median problem. *Journal of Heuristics* 2004; 8: 200–2.
28. Genetic DB, Miki M, Hiroyasu T et al. Parallel simulated annealing with adaptive temperature. In *Proceedings of IEEE International Conference on Systems, Man and Cybernetics 2002*. -: IEEE Press, pp. 1–6.
29. Crainic TG and Gendreau M. Towards a taxonomy of parallel tabu search heuristics, 1997.
30. Resende MGC and Ribeiro CC. Parallel gredy ranomezed adaptive search procedures, 2004.
31. Cotta C, g Talbi E and Alba E. E.: Parallel hybrid metaheuristics. In *Parallel Metaheuristics, a New Class of Algorithms*. -: John Wiley, pp. 347–370.
32. Nebro AJ, Durillo JJ, Luna F et al. Mocell: A cellular genetic algorithm for multiobjective optimization. *International Journal of Intelligent Systems* 2009; 24(7): 726–746. DOI:10.1002/int.20358. URL http://dx.doi.org/10.1002/int.20358.
33. Alba E, Nebro AJ and Luna F. Advances in parallel heterogeneous genetic algorithms for continuous optimization. *International Journal of Applied Mathematics and Computer Science* 2004; 14(3): 101–117.
34. Zhang S and He Z. Implementation of parallel genetic algorithm based on cuda. In Cai Z, Li Z, Kang Z et al. (eds.) *Advances in Computation and Intelligence*, *Lecture Notes in Computer Science*, volume 5821. -: Springer

Berlin / Heidelberg. ISBN 978-3-642-04842-5, 2009. pp. 24–30.

35. Wong ML, Wong TT and Fok KL. Parallel evolutionary algorithms on graphics processing unit. In *Congress on Evolutionary Computation*, volume 3. Edinburgh, Scotland, UK: IEEE Press, pp. 2286–2293 Vol. 3. DOI: 10.1109/CEC.2005.1554979.
36. Li JM, Wang XJ, He RS et al. An efficient fine-grained parallel genetic algorithm based on gpu-accelerated. In *Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference on*. -: IEEE Press, pp. 855–862. DOI:10.1109/NPC.2007.108.
37. Maitre O, Baumes LA, Lachiche N et al. Coarse grain parallelization of evolutionary algorithms on gpgpu cards with easea. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. GECCO '09, New York, NY, USA: ACM. ISBN 978-1-60558-325-9, pp. 1403–1410. DOI:http://doi.acm.org/10.1145/1569901.1570089. URL http://doi.acm.org/10.1145/1569901.1570089.
38. Pospichal P, Jaros J and Schwarz J. Parallel genetic algorithm on the cuda architecture. In Di Chio C, Cagnoni S, Cotta C et al. (eds.) *Applications of Evolutionary Computation*, *Lecture Notes in Computer Science*, volume 6024. -: Springer Berlin / Heidelberg. ISBN 978-3-642-12238-5, 2010. pp. 442–451.
39. Pospichal P, Murphy E, O'Neill M et al. Acceleration of grammatical evolution using graphics processing units: computational intelligence on consumer games and graphics hardware. In Krasnogor N and Lanzi PL (eds.) *GECCO (Companion)*. Dublin,Ireland: ACM. ISBN 978-1-4503-0690-4, pp. 431–438.
40. Thomson Reuters. Web of science, 2016. Available at: https://apps.webofknowledge.com.
41. Harding S and Banzhaf W. Fast genetic programming and artificial developmental systems on gpus. In *High Performance Computing Systems and Applications*. Saskatoon, SK: IEEE Press, pp. 2–2. DOI:10.1109/HPCS.2007.17.
42. Van Luong T, Melab N and Talbi EG. Parallel hybrid evolutionary algorithms on gpu. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*. New York, NY, USA: IEEE, pp. 1 –8. DOI:10.1109/CEC.2010.5586403.
43. Tsutsui S and Fujimoto N. Aco with tabu search on a gpu for solving qaps using move-cost adjusted thread assignment. In et al NK (ed.) *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*. Dublin, Ireland: ACM. ISBN 978-1-4503-0557-0, pp. 1547–1554.
44. Dorigo M and Di Caro G. New ideas in optimization. In Corne D, Dorigo M, Glover F et al. (eds.) *New Ideas in Optimization*, chapter The Ant Colony Optimization Meta-heuristic. Maidenhead, UK, England: McGraw-Hill Ltd., UK. ISBN 0-07-709506-5, 1999. pp. 11–32. URL http://dl.acm.org/citation.cfm?id=329055.329062.
45. Glover F and Laguna M. *Tabu Search*. Norwell, MA, USA: Kluwer Academic Publishers, 1997. ISBN 079239965X.
46. Pedemonte M, Alba E and Luna F. Bitwise operations for gpu implementation of genetic algorithms. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*. GECCO '11,

New York, NY, USA: ACM. ISBN 978-1-4503-0690-4, pp. 439–446. DOI:10.1145/2001858.2002031. URL http://doi.acm.org/10.1145/2001858.2002031.

47. Cano A, Zafra A and Ventura S. Speeding up the evaluation phase of gp classification algorithms on gpus. *Soft Comput* 2012; 16(2): 187–202. DOI:10.1007/s00500-011-0713-4. URL http://dx.doi.org/10.1007/s00500-011-0713-4.
48. Chitty DM. Fast parallel genetic programming: multi-core CPU versus many-core GPU. *Soft Comput* 2012; 16(10): 1795–1814. DOI:10.1007/s00500-012-0862-0. URL http://dx.doi.org/10.1007/s00500-012-0862-0.
49. Jaros J and Tyrala R. Gpu-accelerated evolutionary design of the complete exchange communication on wormhole networks. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. GECCO '14, New York, NY, USA: ACM. ISBN 978-1-4503-2662-9, pp. 1023–1030. DOI:10.1145/2576768.2598315. URL http://doi.acm.org/10.1145/2576768.2598315.
50. Contreras I, Jiang Y, Hidalgo J et al. Using a gpu-cpu architecture to speed up a ga-based real-time system for trading the stock market. *Soft Computing* 2012; 16: 203–215. DOI:10.1007/s00500-011-0714-3. URL http://dx.doi.org/10.1007/s00500-011-0714-3.
51. Pryor G, Lucey B, Maddipatla S et al. High-level gpu computing with jacket for matlab and c/c++. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conferences*, volume 8060. -: Matlab, p. 4.
52. Ha S and Moon B. Fast knowledge discovery in time series with GPGPU on genetic programming. In Silva S and Esparcia-Alcázar AI (eds.) *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015*. ACM. ISBN 978-1-4503-3472-3, pp. 1159–1166. URL http://dl.acm.org/citation.cfm?id=2739480.
53. Yu Q, Chen C and Pan Z. Parallel genetic algorithms on programmable graphics hardware. In Wang L, Chen K and Ong Y (eds.) *Advances in Natural Computation*, *Lecture Notes in Computer Science*, volume 3612. Changsha, China: Springer Berlin Heidelberg. ISBN 978-3-540-28320-1, 2005. pp. 1051–1059. URL http://dx.doi.org/10.1007/11539902_134.
54. Ng CK, Zhang LS, Li D et al. Discrete filled function method for discrete global optimization. *Comput Optim Appl* 2005; 31(1): 87–115. DOI:10.1007/s10589-005-0985-7. URL http://dx.doi.org/10.1007/s10589-005-0985-7.
55. Wong ML and Wong TT. Parallel hybrid genetic algorithms on consumer-level graphics hardware. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*. Vancouver, Canada: IEEE Press, pp. 2973–2980. DOI:10.1109/CEC.2006.1688683.
56. Wong ML and Wong TT. Implementation of parallel genetic algorithms on graphics processing units. In Gen M, Green D, Katai O et al. (eds.) *Intelligent and Evolutionary Systems*, *Studies in Computational Intelligence*, volume 187. Berlin, Heidelberg: Springer. ISBN 978-3-540-95977-9, 2009. pp. 197–216. URL http://dx.doi.org/10.1007/978-3-540-95978-6_14.

57. Luo Z and Liu H. Cellular genetic algorithms and local search for 3-sat problem on graphic hardware. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*. Vancouver, Canada: IEEE Press, pp. 2988–2992. DOI:10.1109/CEC.2006.1688685.
58. Selman B and Kautz H. Domain-independent extensions to gsat: Solving large structured satisfiability problems. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, volume 93. Chambèry, France: IJCAI, pp. 290–295.
59. Li J, Zhang L and Liu L. A parallel immune algorithm based on fine-grained model with gpu-acceleration. In *Proceedings of the 2009 Fourth International Conference on Innovative Computing, Information and Control*. ICICIC '09, Washington, DC, USA: IEEE Computer Society. ISBN 978-0-7695-3873-0, pp. 683–686. DOI: http://dx.doi.org/10.1109/ICICIC.2009.44. URL http://dx.doi.org/10.1109/ICICIC.2009.44.
60. Vidal P and Alba E. Cellular genetic algorithm on graphic processing units. In González JR, Pelta DA, Cruz C et al. (eds.) *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*, *Studies in Computational Intelligence*, volume 284. Berlin, Heidelberg: Springer. ISBN 978-3-642-12537-9, 2010. pp. 223–232. URL http://dx.doi.org/10.1007/978-3-642-12538-6_19.
61. Franco MA and Bacardit J. Large-scale experimental evaluation of GPU strategies for evolutionary machine learning. *Inf Sci* 2016; 330: 385–402. DOI:10.1016/j.ins.2015.10.025. URL http://dx.doi.org/10.1016/j.ins.2015.10.025.
62. Pospichal P and Jaros J. Gpu-based acceleration of the genetic algorithm. Technical report, GECCO competition, Montreal, Canada, 2009.
63. Tsutsui S and Fujimoto N. Solving quadratic assignment problems by genetic algorithms with gpu computation: a case study. In *GECCO09*. NY, USA: ACM. ISBN 978-1-60558-505-5, pp. 2523–2530.
64. Luong TV, Melab N and Talbi EG. Gpu-based island model for evolutionary algorithms. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*. GECCO '10, New York, USA: ACM. ISBN 978-1-4503-0072-8, pp. 1089–1096. DOI:10.1145/1830483.1830685. URL http://doi.acm.org/10.1145/1830483.1830685.
65. Pospíchal P, Schwarz J and Jaroš J. Parallel genetic algorithm solving 0/1 knapsack problem running on the gpu. In *16th International Conference on Soft Computing MENDEL 2010*. Brno, Czech Republic: Brno University of Technology. ISBN 978-80-214-4120-0, pp. 64–70.
66. Munawar A, Wahib M, Munetomo M et al. Hybrid of genetic algorithm and local search to solve max-sat problem using nvidia cuda framework. *Genetic Programming and Evolvable Machines* 2009; 10: 391–415.
67. Diego FJ, Gómez EM, Ortega-Mier M et al. Parallel cuda architecture for solving de vrp with aco. In Sethi SP, Bogataj M and Ros-McDonnell L (eds.) *Industrial Engineering: Innovative Networks*. London: Springer. ISBN 978-1-4471-2320-0, 2012. pp. 385–393. URL http://dx.doi.org/10.1007/978-1-4471-2321-7_43.

68. Delevacq A, Delisle P, Gravel M et al. Parallel ant colony optimization on graphics processing units. *J Parallel Distrib Comput* 2013; 73(1): 52–61.
69. Cecilia JM, García JM, Nisbet A et al. Enhancing data parallelism for ant colony optimization on gpus. *Journal of Parallel and Distributed Computing* 2013; 73(1): 42 – 51. DOI:10.1016/j.jpdc.2012.01.002. URL http://www.sciencedirect.com/science/article/pii/S0743731512000032. Metaheuristics on GPUs.
70. Vidal P and Alba E. A multi-gpu implementation of a cellular genetic algorithm. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*. New York, NY, USA: IEEE, pp. 1 –7. DOI:10.1109/CEC.2010.5586530.
71. Luong T, Taillard E, Melab N et al. Parallelization strategies for hybrid metaheuristics using a single gpu and multi-core resources. In Coello CA, Cutello V, Deb K et al. (eds.) *Parallel Problem Solving from Nature - PPSN XII*, *Lecture Notes in Computer Science*, volume 7492. Berlin, Heidelberg: Springer. ISBN 978-3-642-32963-0, 2012. pp. 368–377. URL http://dx.doi.org/10.1007/978-3-642-32964-7_37.
72. Knudsen L and Meier W. Cryptanalysis of an identification scheme based on the permuted perceptron problem. In Stern J (ed.) *Advances in Cryptology (EUROCRYPT'99)*, *Lecture Notes in Computer Science*, volume 1592. Berlin: Springer Berlin Heidelberg. ISBN 978-3-540-65889-4, 1999. pp. 363–374. URL http://dx.doi.org/10.1007/3-540-48910-X_25.
73. Taillard E. Robust taboo search for the quadratic assignment problem. *Parallel Computing* 1991; 17(4-5): 443 – 455. DOI:http://dx.doi.org/10.1016/S0167-8191(05)80147-4. URL http://www.sciencedirect.com/science/article/pii/S0167819105801474.
74. Wang J, Gong B, LIU H et al. Heterogeneous computing and grid scheduling with hierarchically parallel evolutionary algorithms. *Journal of Computational Information Systems* 2014; 10(8): 3291–3298.
75. O'Neill M, Hemberg E, Gilligan C et al. Geva: grammatical evolution in java. *SIGEVOlution* 2008; 3(2): 17–22. DOI:10.1145/1527063.1527066. URL http://doi.acm.org/10.1145/1527063.1527066.