Chapter 5

Parallel Genetic Programming on Graphics Processing Units

Douglas A. Augusto, Heder S. Bernardino and Helio J.C. Barbosa

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/48364

1. Introduction

In program inference, the evaluation of how well a candidate solution solves a certain task is usually a computationally intensive procedure. Most of the time, the evaluation involves either submitting the program to a simulation process or testing its behavior on many input arguments; both situations may turn out to be very time-consuming. Things get worse when the optimization algorithm needs to evaluate a population of programs for several iterations, which is the case of genetic programming.

Genetic programming (GP) is well-known for being a computationally demanding technique, which is a consequence of its ambitious goal: to automatically generate computer programs—in an arbitrary language—using virtually no domain knowledge. For instance, evolving a classifier, a program that takes a set of attributes and predicts the class they belong to, may be significantly costly depending on the size of the training dataset, that is, the amount of data needed to estimate the prediction accuracy of a single candidate classifier.

Fortunately, GP is an inherently parallel paradigm, making it possible to easily exploit any amount of available computational units, no matter whether they are just a few or many thousands. Also, it usually does not matter whether the underlying hardware architecture can process simultaneously instructions and data ("MIMD") or only data ("SIMD").[1] Basically, GP exhibits three levels of parallelism: (i) *population-level* parallelism, when many populations evolve simultaneously; (ii) *program-level* parallelism, when programs are evaluated in parallel; and finally (iii) *data-level* parallelism, in which individual training points for a single program are evaluated simultaneously.

Until recently, the only way to leverage the parallelism of GP in order to tackle complex problems was to run it on large high-performance computational installations, which are normally a privilege of a select group of researchers. Although the multi-core era has emerged and popularized the parallel machines, the architectural change that is probably going to

[1] MIMD stands for *Multiple Instructions Multiple Data* whereas SIMD means *Single Instruction Multiple Data*.

INTECH
open science | open minds

©2012 Augusto et al., licensee InTech. This is an open access chapter distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

revolutionize the applicability of GP started about a decade ago when the GPUs began to acquire general-purpose programmability. Modern GPUs have an astonishing theoretical computational power, and are capable of behaving much like a conventional multi-core CPU processor in terms of programmability. However, there are some intrinsic limitations and patterns of workload that may cause huge negative impact on the resulting performance if not properly addressed. Hence, this paper aims at presenting and discussing efficient ways of implementing GP's evaluation phase, at the program- and data-level, so as to achieve the maximum throughput on a GPU.

The remaining of this chapter is organized as follows. The next Section, 2, will give an overview of the GPU architecture followed by a brief description of the open computing language, which is the open standard framework for heterogeneous programming, including CPUs and GPUs. Section 3 presents the development history of GP in the pursuit of getting the most out of the GPU architecture. Then, in Section 4, three fundamental parallelization strategies at the program- and data-level will be detailed and their algorithms presented in a pseudo-OpenCL form. Finally, Section 5 concludes the chapter and points out some interesting directions of future work.

2. GPU programming

The origin of graphics processing units dates back to a long time ago, when they were built exclusively to execute graphics operations, mainly to process images' pixels, such as calculating each individual pixel color, applying filters, and the like. In video or gaming processing, for instance, the task is to process batches of pixels within a short time-frame—such operation is also known as *frame rendering*—in order to display smooth and fluid images to the spectator or player.

Pixel operations tend to be very independent among them, in other words, each individual pixel can be processed at the same time as another one, leading to what is known as *data parallelism* or SIMD. Although making the hardware less general, designing an architecture targeted at some specific type of workload, like data parallelism, may result in a very efficient processor. This is one main reason why GPUs have an excellent performance with respect to power consumption, price, and density. Another major reason behind such a performance is attributed to the remarkable growing of the game industry in the last years and the fact that computer games have become more and more complex, pressing forward the development of GPUs while making them ubiquitous.

It turned out that at some point the development of GPUs was advancing so well and the architecture was progressively getting more ability to execute a wider range of sophisticated instructions, that eventually it earned the status of a general-purpose processor—although still an essentially data parallel architecture. That point was the beginning of the exploitation of the graphics processing unit as a parallel accelerator for a much broader range of applications besides video and gaming processing.

2.1. GPU architecture

The key design philosophy responsible for the great GPU's efficiency is the maximization of the number of transistors dedicated to actual computing—i.e., arithmetic and logic units (ALU)—which are packed as many small and relatively simple processors [26]. This is

rather different from the modern multi-core CPU architecture, which has large and complex cores, reserving a considerable area of the processor die for other functional units, such as control units (out-of-order execution, branch prediction, speculative execution, etc.) and cache memory [21].

This design difference reflects the different purpose of those architectures. While the GPU is optimized to handle data-parallel workloads with regular memory accesses, the CPU is designed to be more generic and thus must manage with reasonable performance a larger variety of workloads, including MIMD parallelism, divergent branches and irregular memory accesses. There is also another important conceptual difference between them. Much of the extra CPU complexity is devoted to reduce the latency in executing a single task, which classifies the architecture as *latency-oriented* [14]. Conversely, instead of executing single tasks as fast as possible, GPUs are *throughput-oriented* architectures, which means that they are designed to optimize the throughput, that is, the amount of completed tasks per unit of time.

2.2. Open Computing Language – OpenCL

The Open Computing Language, or simply OpenCL, is an open specification for heterogeneous computing released by the Khronos Group[2] in 2008 [25]. It resembles the NVIDIA CUDA[3] platform [31], but can be considered as a superset of the latter; they basically differ in the following points. OpenCL (i) is an open specification that is managed by a set of distinct representatives from industry, software development, academia and so forth; (ii) is meant to be implemented by any compute device vendor, whether they produce CPUs, GPUs, hybrid processors, or other accelerators such as digital signal processors (DSP) and field-programmable gate arrays (FPGA); and (iii) is portable across architectures, meaning that a parallel code written in OpenCL is guaranteed to correctly run on every other supported device.[4]

2.2.1. Hardware model

In order to achieve code portability, OpenCL employs an abstracted device architecture that standardizes a device's processing units and memory scopes. All supported OpenCL devices must expose this minimum set of capabilities, although they may have different capacities and internal hardware implementation. Illustrated in Figure 1 is an OpenCL general device abstraction. The terms SPMD, SIMD and PC are mostly GPU-specific, though; they could be safely ignored on behalf of code portability, but understanding them is important to write efficient code for this architecture, as will become clear later on.

An OpenCL device has one or more *compute units* (CU), and there is at least one *processing element* (PE) per compute unit, which actually performs the computation. Such layers are meant (i) to encourage better partitioning of the problem towards fine-grained granularity and low communication, hence increasing the scalability to fully leverage a large number of CUs when available; and (ii) to potentially support more restricted compute architectures, by

[2] `http://www.khronos.org/opencl`

[3] CUDA is an acronym for *Compute Unified Device Architecture*, the NVIDIA's toolkit for GP-GPU programming.

[4] It is worthy to note that OpenCL only guarantees *functional portability*, i.e., there is no guarantee that the same code will perform equally well across different architectures (performance portability), since some low-level optimizations might fit a particular architecture better than others.

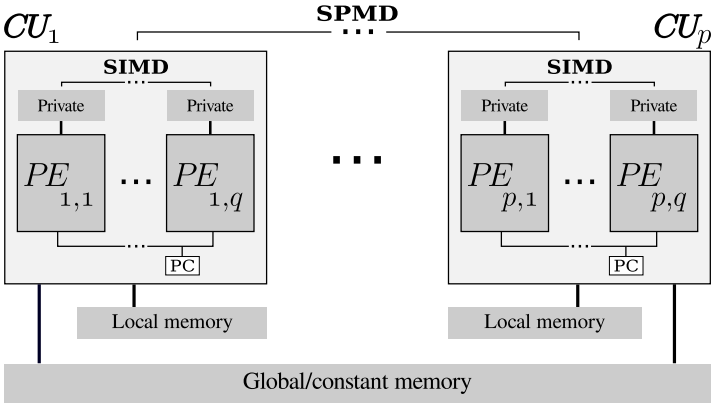

Figure 1. Abstraction of a modern GPU architecture

not strictly enforcing parallelism among CUs while still ensuring that the device is capable of doing synchronism, which can occur among PEs within each CU [15].

Figure 1 shows four scopes of memory, namely, *global*, *constant*, *local*, and *private* memories. The global memory is the device's main memory, the biggest but also the slowest of the four in terms of bandwidth and latency, specially for irregular accesses. The constant memory is a small and slightly optimized memory for read-only accesses. OpenCL provides two really fast memories: local and private. Both are very small; the main difference between them is the fact that the former is shared among all the PEs within a CU—thus very useful for communication—and the latter is even smaller and reserved for each PE.

Most of modern GPUs are capable of performing not only SIMD parallelism, but also what is referred to as SPMD parallelism (literally *Single Program Multiple Data*), which is the ability to simultaneously execute *different* instructions of the *same* program on many data. This feature is closely related to the capability of the architecture in maintaining a record of multiple different instructions within a program being executed which is done by *program counter* (PC) registers. Nowadays GPUs can usually guarantee that at least among compute units there exists SPMD parallelism, in other words, different CUs can execute different instructions in parallel. There may exist SPMD parallelism within CUs also, but they occur among blocks of PEs.[5] For the sake of simplicity, the remaining of this chapter will ignore this possibility and assume that all PEs within a CU can only execute one instruction at a time (SIMD parallelism), sharing a single PC register. A strategy of parallelization described in Section 4.4 will show how the SPMD parallelism can be exploited in order to produce one of the most efficient parallel algorithms for genetic programming on GPUs.

2.2.2. Software model

OpenCL specifies two code spaces: the *host* and *kernel* code. The former holds any user-defined code, and is also responsible for initializing the OpenCL platform, managing the device's memory (buffer allocation and data transfer), defining the problem's parallel

[5] Those blocks are known as *warps* [32] or *wavefronts* [1].

partitioning, submitting commands, and coordinating executions. The latter, the kernel code, is the actual parallel code that is executed by a compute device.

An OpenCL kernel is similar to a C function[6]. Due to architectural differences across devices, it has some restrictions, such as prohibiting recursion, but also adds some extensions, like vector data types and operators, and is intended to be executed in parallel by each processing element, usually with each instance working on a separate subset of the problem. A kernel instance is known as *work-item* whereas a group of work-items is called a *work-group*.

Work-items within a work-group are executed on a unique compute unit, therefore, according to the OpenCL specification, they can share information and synchronize. Determining how work-items are divided into work-groups is a critical phase when decomposing a problem; a bad division may lead to inefficient use of the compute device. Hence, an important part of the parallel modeling concerns defining what is known as *n-dimensional computation domain*. This turns out to be the definition of the *global size*, which is the total amount of work-items, and the *local size*, the number of work-items within a work-group, or simply the work-groups' size.

In summary, when parallelizing the GP's evaluation phase, the two most important modeling aspects are the *kernel* code and the n-dimensional computation domain. Section 4 will present these definitions for each parallelization strategy.

3. Genetic programming on GPU: A bit of history

It is natural to begin the history of GP on GPUs referring to the first improvements obtained by parallelization of a GA on programmable graphics hardware. The first work along this line seems to be [41], which has proposed a genetic algorithm in which crossover, mutation, and fitness evaluation were performed on graphic cards achieving speedups up to 17.1 for large population sizes.

Other GA parallelization on GPUs was proposed in [39] which followed their own ideas explored in [40] for an evolutionary programming technique (called FEP). The proposal, called Hybrid GA, or shortly HGA, was evaluated using 5 test-functions, and CPU-GPU as well as HGA-FEP comparisons were made. It was observed that their GA on GPU was more effective and efficient than their previous parallel FEP.

Similarly to [41], [24] performed crossover, mutation, and fitness evaluation on GPU to solve the problem of packing many granular textures into a large one, which helps modelers in freely building virtual scenes without caring for efficient usage of texture memory. Although the implementation on CPU performed faster in the cases where the number of textures was very small (compact search space), the performance of the implementation on GPU is almost two times faster when compared to execution on CPU.

The well-known satisfiability problem, or shortly SAT, is solved on graphic hardware in [30], where a cellular genetic algorithm was adopted. The algorithm was developed using NVIDIA's C for Graphics (Cg) programming toolkit and achieved a speedup of approximately 5. However, the author reports some problems in the implementation process, like the nonexistence of a pseudo-random number generator and limitations in the texture's size.

[6] The OpenCL kernel's language is derived from the C language.

Due the growing use of graphics cards in the scientific community, in general, and particularly in the evolutionary computation field, as described earlier, the exploration of this high-performing solution in genetic programming was inevitable. Ebner et al. published the first work exploring the GPU capacity in GP [11]. Although a high level language was used in that case (Cg), the GPU was only used to generate the images from the candidate programs (vertex and pixel shaders). Then the created images are presented to the user for his evaluation.

However, it was in 2007 that the extension of the technique of general purpose computing using graphics cards in GP was more extensively explored [2, 9, 17, 18]. Two general purpose computation toolkits for GPUs were preferred in these works: while [2, 9] implemented their GP using Cg, Harding and Banzhaf [17, 18] chose Microsoft's Accelerator, a .Net's library which provides access to the GPU via DirectX's interface.

The automatic construction of tree-structural image transformation on GPU was proposed in [2], where the speedup of GP was explored in different parallel architectures (master-slave and island), as well as on single and multiple GPUs (up to 4). When compared with its sequential version, the proposed approach obtained a speedup of 94.5 with one GPU and its performance increased almost linearly by adding GPUs.

Symbolic regression, fisher iris dataset classification, and 11-way multiplexer problems composed the computational experiments in [9]. The results demonstrated that although there was little improvement for small numbers of fitness cases, considerable gains could be obtained (up to around 10 times) when this number becomes much larger.

The classification between two spirals, the classification of proteins, and a symbolic regression problem were used in [17, 18] to evaluate their Cartesian GP on GPU. In both works, each GP individual is compiled, transferred to GPU, and executed. Some benchmarks were also performed in [18] to evaluate floating point as well as binary operations. The rules of a cellular automaton with the von Neumann neighborhood and used to simulate the diffusion of chemicals were generated by means of Cartesian GP in [17]. The best obtained speedup in these works was 34.63.

Following the same idea of compiling the candidate solutions, [16] uses a Cartesian GP on GPU to remove noise in images. Different types of noise were artificially introduced into a set of figures and performance analyses concluded that this sort of parallelism is indicated for larger images.

A simple instruction multiple data interpreter was developed using RapidMind and presented in [28], where a performance of one Giga GP operations per second was observed in the computational experiments. In contrast to [16–18] where the candidate programs were compiled to execute on GPUs, [28] showed a way of interpreting the trees. While the previous presented approach requires that programs are large and run many times to compensate the cost of compilation and transference to the GPU, the interpretable proposal of [28] seems to be more consistent because it achieved speed ups of more than an order of magnitude in the Mackey-Glass time series and protein prediction problems, even for small programs and few test cases.

The same solution of interpreting the candidate programs was used in [27], but a predictor was evolved in this case. Only the objective function evaluation was performed on GPU, but this step represents, in that study, about 85% of the total run time.

Another study exploring the GPU capacity in GP is presented in [29]. RapidMind is used to implement a GP solution to solve a cancer prediction problem from a dataset containing a million inputs. A population of 5 million programs evolves executing about 500 million GP operations per second. The author found a 7.6 speed up during the computational experiments, but their discussion indicates that the increment in the performance was limited by the access to the 768 Mb of the training data (the device used had 512Mb).

Since these first works were published, improving GP performance by using GP-GPU becomes a new research field. Even the performance of GP on graphic devices of video game consoles was analyzed [36–38], but PC implementations of GP have demonstrated to be faster and more robust. However, it was with the current high level programming languages [4, 34], namely NVIDIA's CUDA and OpenCL, that GP implementations using GP becomes popular, specially in much larger/real world applications. Also, TidePowerd's GPU.NET was studied for speed up Cartesian GP [19].

Genetic programming is used in [10] to search, guided by user interaction, in the space of possible computer vision programs, where a real-time performance is obtained by using GPU for image processing operations. The objective was evolving detectors capable of extracting sub-images indicated by the user in multiple frames of a video sequence.

An implementation of GP to be executed in a cluster composed by PCs equipped with GPUs was presented in [20]. In that work, program compilation, data, and fitness execution are spread over the cluster, improving the efficiency of GP when the problem contains a very large dataset. The strategy used is to compile (C code into NVIDIA CUDA programs) and to execute the population of candidate individuals in parallel. The GP, developed in GASS's CUDA.NET, was executed in Microsoft Windows (during the tests) and Gentoo Linux (final deployment), demonstrating the flexibility of that solution. That parallel GP was capable of executing up to 3.44 (classification problem of network intrusion) and 12.74 (image processing problem) Giga GP operations per second.

The computational time of the fitness calculation phase was reduced in [7, 8] by using CUDA. The computational experiments included ten datasets, which were selected from well-known repositories in the literature, and three GP variants for classification problems, in which the main difference between them is the criterion of evaluation. Their proposed approach demonstrated good performance, achieving a speedup of up to 820.18 when compared with their own Java implementation, as well as a speedup of up to 34.02 when compared with BioHEL [13].

Although with much less articles published in the GP field, OpenCL deserves to be highlighted because, in addition of being non-proprietary, it allows for heterogeneous computing. In fact, up to now only [4] presents the development of GP using OpenCL, where the performance of both types of devices (CPU and GPU) was evaluated over the same implementation. Moreover, [4] discusses different parallelism strategies and GPU was up to 126 times faster than CPU in the computational experiments.

The parallelism of GP techniques on GPU is not restricted only to linear, tree, and graph (Cartesian) representations. The improvement in performance of other kinds of GP, such as Grammatical Evolution [33], is just beginning to be explored. However, notice that no papers were found concerning the application of Gene Expression Programming [12] on GPUs. Some complementary information is available in [3, 6].

4. Parallelization strategies

As mentioned in Section 2.2, there are two distinct code spaces in OpenCL, the host and kernel. The steps of the host code necessary to create the environment for the parallel evaluation phase are summarized as follows [4]:[7]

1. **OpenCL initialization**. This step concerns identifying which OpenCL implementation (platform) and compute devices are available. There may exist multiple devices on the system. In this case one may opt to use a single device or, alternatively, all of them, where then a further partitioning of the problem will be required. Training data points, programs or even whole populations could be distributed among the devices.
2. **Calculating the n-dimensional computation domain**. How the workload is decomposed for parallel processing is of fundamental importance. Strictly speaking, this phase only determines the *global* and *local* sizes in a one-dimensional space, which is enough to represent the domain of training data points or programs. However, in conjunction with a kernel, which implements a certain strategy of parallelization, the type of parallelism (at data and/or program level) and workload distribution are precisely defined.
3. **Memory allocation and transfer**. In order to speedup data accesses, some content are allocated/transferred directly to the compute device's memory and kept there, thus avoiding as much as possible the relatively narrow bandwidth between the GPU and the computer's main memory. Three memory buffers are required to be allocated on the device's global memory in order to hold the training data points, population of programs, and error vector. Usually, the training data points are transferred only once, just before the beginning of the execution, remaining then unchanged until the end. The population of programs and error vector, however, are dynamic entities and so they need to be transferred at each generation.
4. **Kernel building**. This phase selects the kernel with respect to a strategy of parallelization and builds it. Since the exact specification of the target device is usually not known in advance, the default OpenCL behavior is to compile the kernel just-in-time. Although this procedure introduces some overhead, the benefit of having more information about the device—and therefore being able to generate better optimized kernel object—usually outweighs the compilation overhead.
5. **GP's evolutionary loop**. Since this chapter focuses on accelerating the evaluation phase of genetic programming by parallelizing it, the iterative evolutionary cycle itself is assumed to be performed sequentially, being so defined in the host space instead of as an OpenCL kernel. [8] The main iterative evolutionary steps are:
   (a) **Population transfer**. Changes are introduced to programs by the evolutionary process via genetic operators, e.g. crossover and mutation, creating a new set of derived programs. As a result, a population transfer needs to be performed from host to device at each generation.

[7] This chapter will not detail the host code, since it is not relevant to the understanding of the parallel strategies. Given that, and considering that the algorithms are presented in a pseudo-OpenCL form, the reader is advised to consult the appropriate OpenCL literature in order to learn about its peculiarities and fill the implementation gaps.

[8] However, bear in mind that a full parallelization, i.e. both evaluation and evolution, is feasible under OpenCL. That could be implemented, for instance, in such a way that a multi-core CPU device would perform the evolution in parallel while one or more GPUs would evaluate programs.

(b) **Kernel execution**. Whenever a new population arrives on the compute device, a kernel is launched in order to evaluate (in parallel) the new programs with respect to the training data points. For any non-trivial problem, this step is the most computationally intensive one.

(c) **Error retrieval**. Finally, after all programs' errors have been accumulated, this vector is transferred back to the host in order to guide the evolutionary process in selecting the set of parents that will breed the next generation.

Regarding the kernel code, it can be designed to evaluate programs in different parallel ways: (i) training points are processed in parallel but programs sequentially; or (ii) the converse, programs are executed in parallel but training points are processed sequentially; or finally (iii) a mixture of these two, where both programs and training points are processed in parallel.

Which way is the best will depend essentially on a combination of the characteristics of the problem and some parameters of the GP algorithm. These strategies are described and discussed in Sections 4.2, 4.3 and 4.4.

4.1. Program interpreter

The standard manner to estimate the fitness of a GP candidate program is to execute it, commonly on varying input arguments, and observe how well it solves the task at hand by comparing its behavior with the expected one. To this end, the program can be *compiled* just before the execution, generating an intermediate object code, or be directly *interpreted* without generating intermediate objects. Both variations have pros and cons. Compiling introduces overhead, however, it may be advantageous when the evaluation of a program is highly demanding. On the other hand, interpretation is usually slower, but avoids the compilation cost for each program. Moreover, interpretation is easy to accomplish and, more importantly, is much more flexible. Such flexibility allows, for example, to emulate a MIMD execution model on a SIMD or SPMD architecture [23]. This is possible because what a data-parallel device actually executes are many instances of the *same* interpreter. Programs, as has always been the case with training points, become data or, in other words, arguments for the interpreter.

A program interpreter is presented in Algorithm Interpreter. It is assumed that the program to be executed is represented as a *prefix linear tree* [5], since a linear representation is very efficient to be operated on, specially on the GPU architecture. An example of such program is:

$$\boxed{+}\boxed{\sin}\boxed{x}\boxed{3.14}$$

which denotes the infix expression $\sin(x) + 3.14$.

The program interpretation operates on a single training data point at a time. The current point is given by the argument n, and $X_n \in \Re^d$ is a d-dimensional array representing the n-th variables (training point) of the problem.

The command `INDEX` extracts the class of the current operator (*op*), which can be a function, constant or variable. The value of a constant is obtained by the `VALUE` command; for variables, this command returns the variable's index in order to get its corresponding value in X_n.

```
Function Interpreter( program, n )
for op ← program_size − 1 to 0 do
    switch INDEX ( program[op] ) do
        case ADD:
            PUSH ( POP + POP );
        case SUB:
            PUSH ( POP − POP );
        case MUL:
            PUSH ( POP × POP );
        case DIV:
            PUSH ( POP ÷ POP );
        case IF-THEN-ELSE:
            if POP then
                PUSH ( POP );
            else
                POP; PUSH ( POP );
        ⋮
        case CONSTANT:
            PUSH ( VALUE ( program[op] ) );
        otherwise
            PUSH ( X_n[VALUE ( program[op] )] );
return POP;
```

The interpreter is stack-based; whenever an operand shows up, like a constant or variable, its value is pushed onto the stack via the `PUSH` command. Conversely, an operator obtains its operands' values on the stack by means of the `POP` command, which removes the most recently stacked values. Then, the value of the resulting operation on its operands is pushed back onto the stack so as to make it available to a parent operator.

As will be seen in the subsequent sections, whatever the parallel strategy, the interpreter will act as a central component of the kernels, doing the hard work. The kernels will basically set up how the interpreter will be distributed among processing elements and which program and training point it will operate on at a given time.

4.2. Data-level Parallelism – DP

The idea behind the data-level parallelism (DP) strategy is to distribute the training data points among the processing elements of a compute device. This is probably the simplest and most natural way of parallelizing GP's evaluation phase when the execution of a program on many independent training points is required.[9] Despite its obviousness, DP is an efficient

[9] However, sometimes it is not possible to trivially decompose the evaluation phase. For instance, an evaluation may involve submitting the program through a simulator. In this case one can try to parallelize the simulator itself or, alternatively, opt to use a program- or population-level kind of parallelism.

strategy, specially when there are a large number of training data points—which is very common in complex problems. Moreover, given that this strategy leads to a data-parallel SIMD execution model, it fits well on a wide range of parallel architectures. Figure 2 shows graphically how the training data points are distributed among the PEs.[10]

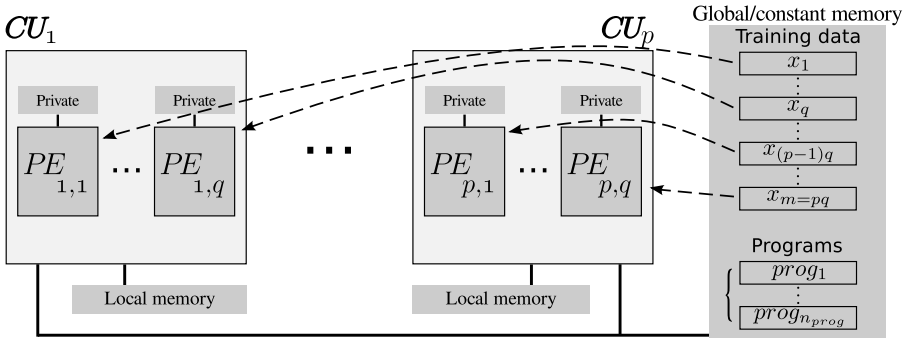

Figure 2. Illustration of the data-level parallelism (DP).

As already mentioned, to precisely define a parallelization strategy in OpenCL, two things must be set up: the n-dimensional domain, more specifically the global and local sizes, and the kernel itself. For the data-level parallelism, it is natural to assign the global computation domain to the training data points domain as a one-to-one correspondence; that is, simply

$$global_{size} = dataset_{size}, \tag{1}$$

where *dataset size* is the number of the training data points. OpenCL lets the programmer to choose whether he or she wants to explicitly define the local size, i.e. how many work-items will be put in a work-group. The exact definition of the local size is only really needed when the corresponding kernel assumes a particular work-group division, which is not the case for DP. Therefore, no local size is explicitly defined for DP, letting then the OpenCL runtime to decide on any configuration it thinks is the best.

Algorithm 1 presents in a pseudo-OpenCL language the DP's kernel. As with any OpenCL kernel, there will be launched $global_{size}$ instances of it on the compute device.[11] Hence, there is one work-item per domain element, with each one identified by its global or local position through the OpenCL commands `get_global_id` and `get_local_id`, respectively. This enables a work-item to select what portion of the compute domain it will operate on, based on its absolute or relative position.

For the DP's kernel, the $global_{id}$ index is used to choose which training data point will be processed, in other words, each work-item will be in charge of a specific point. The *for loop* iterates sequentially over each program of the population (the function `NthProgram` returns the p-th program), that is, every work-item will execute the same program at a given time. Then, the interpreter (Section 4.1) is called to execute the current program, but each work-item will provide a different index, which corresponds to the training data point it took

[10] To simplify, in Figures 2, 4 and 5 it is presumed that the number of PEs (or CUs) coincides with the number of training data points (or programs), but in practice this is rarely the case.

[11] It is worthy to notice that the actual amount of work-items executed in parallel by the OpenCL runtime will depend on the device's capabilities, mainly on the number of processing elements.

Algorithm 1: GPU DP's OpenCL kernel

$global_{id} \leftarrow$ `get_global_id();`

for $p \leftarrow 0$ **to** $population_{size} - 1$ **do**

  $program \leftarrow$ `NthProgram(`p`);`

  $error \leftarrow |$`Interpreter(`$program, global_{id}$`)` $- Y[global_{id}]|$;

  $E[p] \leftarrow$ `ErrorReduction(`$0, \ldots, global_{size} - 1$`);`

responsibility for. Once interpreted, the output returned by the program is then compared with the expected one for that point, whose value is stored in array Y. This results in a prediction error; however, the overall error is what is meaningful to estimate the fitness of a program.

Note however that the errors are spread among the work-items, because each work-item has processed a single point and has computed its own error independently. This calls for what is known in the parallel computing literature as the *reduction* operation [22]. The naive way of doing that is to sequentially cycle over each element and accumulate their values; in our case it would iterate from work-item indexed by 0 to $global_{size} - 1$ and put the total value in $E[p]$, the final error relative to the p-th program. There is however a clever and parallel way of doing reduction, as exemplified in Figure 3, which decreases the complexity of this step from $O(N)$ to just $O(log_2 N)$ and still assures a nice coalesced memory access suited for the GPU architecture [1, 32].[12]

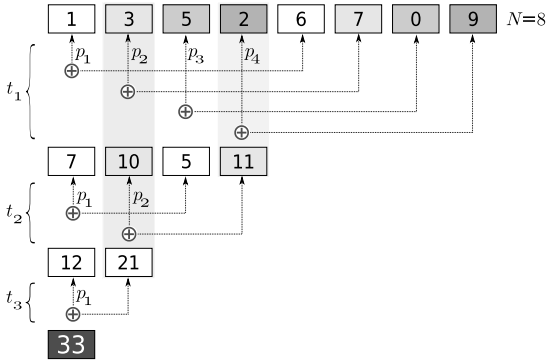

Figure 3. $O(log_2 N)$ parallel reduction with sequential addressing.

4.3. Program-level Parallelism – PP

One serious drawback of the data-level parallelism strategy is that when there are few training data points the compute device may probably be underutilized. Today's high-end GPUs have thousands of processing elements, and this number has increased at each new hardware generation. In addition, to achieve optimal performance on GPUs, multiple work-items should be launched for each processing element. This helps, for instance, to hide memory

[12] This chapter aims at just conveying the idea of the parallel reduction, and so it will not get into the algorithmic details on how reduction is actually implemented. The reader is referred to the given references for details.

access latencies while reading from or writing to the device's global memory [1, 32]. Therefore, to optimally utilize a high-end GPU under the DP strategy, one should prefer those problems having tens of thousands of training data points. Unfortunately, there are many real-world problems out there for which no such amount of data is available.

Another limitation of the DP strategy is that sometimes there is no easy way to decompose the evaluation of a program into independent entities, like data points. Many program evaluations that need a simulator, for example, fall into this category, where a parallel implementation of the simulator is not feasible to accomplish.

An attempt to overcome the DP limitations, particularly what concerns the desire of a substantially large amount of training data points, is schematically shown in Figure 4. This parallelization strategy is here referred to as program-level parallelism (PP), meaning that programs are executed in parallel, each program per PE [4, 35]. Assuming that there are enough programs to be evaluated, even a few training data points should keep the GPU fully occupied.

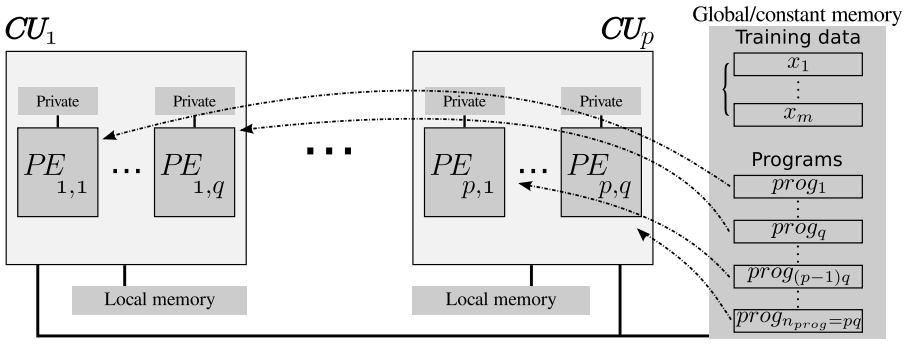

Figure 4. Illustration of the program-level parallelism (PP).

In PP, while programs are interpreted in parallel, the training data points within each PE are processed sequentially. This suggests a computation domain based on the number of programs, in other words, the global size can be defined as:

$$global_{size} = population_{size} \tag{2}$$

As with DP, PP does not need to have control of the number of work-items within a work-group, thus the local size can be left untouched.

A pseudo-OpenCL code for the PP kernel is given in Algorithm 2. It resembles the DP's algorithm, but in PP what is being parallelized are the programs instead of the training data points. Hence, each work-item takes a different program and interpret it iteratively over all points. A positive side effect of this inverse logic is that, since the whole evaluation of a program is now done in a single work-item, all the partial prediction errors are promptly available locally. Put differently, in PP a final reduction step is not required.

4.4. Program- and Data-level Parallelism – PDP

Unfortunately, PP solves the DP's necessity of large training datasets but introduces two other problems: (i) to avoid underutilization of the GPU a large population of programs should now

Algorithm 2: GPU PP's OpenCL kernel

```
global_id ← get_global_id();
program ← NthProgram(global_id);
error ← 0.0;
for n ← 0 to dataset_size − 1 do
    error ← error + |Interpreter( program, n ) − Y[n]|;
E[global_id] ← error;
```

be employed; and, more critically, (ii) the PP's execution model is not suited for an inherently data-parallel architecture like GPUs.

While (i) can be dealt with by simply specifying a large population as a parameter choice of a genetic programming algorithm, the issue pointed out in (ii) cannot be solved for the PP strategy.

The problem lies on the fact that, as mentioned in Section 2, GPUs are mostly a SIMD architecture, specially among processing elements within a compute unit. Roughly speaking, whenever two (or more) different instructions try to be executed at the same time, a hardware conflict occurs and then these instructions are performed sequentially, one at a time. In the related literature, this phenomenon is often referred to as *divergence*. Since in PP each PE interprets a different program, the degree of divergence is the highest possible: at a given moment each work-item's interpreter is potentially interpreting a different primitive. Therefore, in practice, the programs within a CU will most of the time be evaluated sequentially, seriously degrading the performance.

However, observing the fact that modern GPUs are capable of simultaneously executing different instructions at the level of compute units, i.e. the SPMD execution model, one could devise a parallelization strategy that would take advantage of this fact. Such strategy exists, and it is known here as program- and data-level parallelism, or simply PDP [4, 35]. Its general idea is illustrated in Figure 5. In PDP, a single program is evaluated per compute unit—this

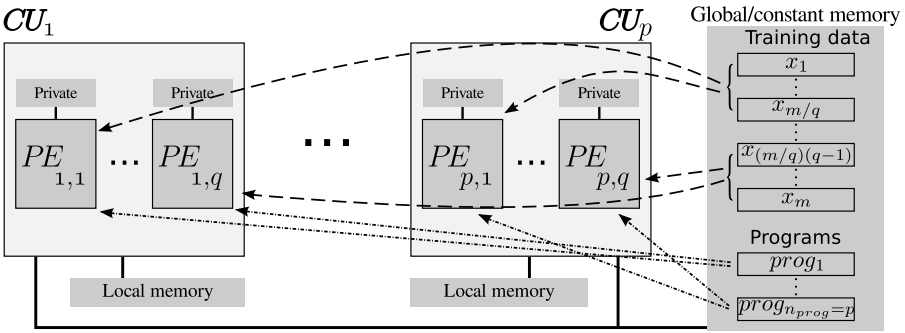

Figure 5. Illustration of the program- and data-level parallelism (PDP).

prevents the just mentioned problem of divergence—but within each CU all the training data points are processed in parallel. Therefore, there are two levels of parallelism: a program-level parallelism among the compute units, and a data-level parallelism on the processing elements.

Indeed, PDP can be seen as a mixture of the DP and PP strategies. But curiously, PDP avoids all the drawbacks associated with the other two strategies: (i) once there are enough data to saturate just a single CU, smaller datasets can be used at no performance loss; (ii) large populations are not required either, since the number of CUs on current high-end GPUs is in the order of tens; and (iii) there is no divergence with respect to program interpretation.[13]

In order to achieve both levels of parallelism, a fine-tuned control over the computation domain is required; more precisely, both local and global sizes must be properly defined.

Since a work-group should process all training data points for a single program and there is a population of programs to be evaluated, one would imagine that setting $local_{size}$ as $dataset_{size}$ and $global_{size}$ as $population_{size} \times dataset_{size}$ would suffice. This is conceptually correct, but an important detail makes the implementation not as straightforward as one would expect. The OpenCL specification allows any compute device to declare an upper bound regarding the number of work-items within a work-group. This is not arbitrary. The existence of a limit on the number of work-items per work-group is justified by the fact that there exists a relation between the maximum number of work-items and the device's capabilities, with the latter restricting the former. Put differently, an unlimited number of work-items per work-group would not be viable, therefore a limit, which is provided by the hardware vendor, must be taken into account.

With the aforementioned in mind, the local size can finally be set to

$$local_{size} = \begin{cases} dataset_{size} & \text{if } dataset_{size} < local_{max\_size} \\ local_{max\_size} & \text{otherwise} \end{cases}, \tag{3}$$

which limits the number of work-items per work-group to the maximum supported, given by the variable $local_{max\_size}$, when the number of training data points exceeds it. This implies that when such a limit takes place, a single work-item will be in charge of more than one training data point, that is, the work granularity is increased. As for the global size, it can be easily defined as

$$global_{size} = population_{size} \times local_{size}, \tag{4}$$

meaning that the set of work-items defined above should be replicated as many times as the number of programs to be evaluated.

Finally, algorithm 3 shows the OpenCL kernel for the PDP strategy. Compared to the other two kernels (Algorithms 1 and 2), it comes as no surprise its greater complexity, as this kernel is a combination of the other two and still has to cope with the fact that a single instance, i.e. a work-item, can process an arbitrary number of training data points. The command `get_group_id`, which returns the work-group's index of the current work-item, has the purpose of indexing the program that is going to be evaluated by the entire group. The *for loop* is closely related to the local size (Equation 3), and acts as a way of iterating over multiple training data points if the work-item (indexed locally by $local_{id}$) is in charge of many of them; when the dataset size is less or equal to the local size, only one iteration will be performed. Then, an index calculation is done in order to get the index (n) of the current training data

[13] Notice, though, that divergence might still occur if two (or more) training data points can cause the interpreter to take different paths for the same program. For instance, if the conditional *if-then-else* primitive is used, a data point could cause an interpreter's instance to take the *then* path while other data could make another instance to take the *else* path.

Algorithm 3: GPU PDP's OpenCL kernel

```
local_id ← get_local_id();
group_id ← get_group_id();
program ← NthProgram(group_id);

error ← 0.0;
for i ← 0 to ⌈dataset_size / local_size⌉ − 1 do
    n ← i × local_size + local_id;
    if n < dataset_size then
        error ← error + |Interpreter( program, n ) − Y[n]|;

E[group_id] ← ErrorReduction(0, ..., local_size − 1);
```

point to be processed.[14] Due to the fact that the dataset size may not be evenly divisible by the local size, a range check is performed to guarantee that no out-of-range access will occur. Finally, since the prediction errors for a given program will be spread among the local work-items at the end of the execution, an error reduction operation takes place.

5. Conclusions

This chapter has presented different strategies to accelerate the execution of a genetic programming algorithm by parallelizing its costly evaluation phase on the GPU architecture, a high-performance processor which is also energy efficient and affordable.

Out of the three studied strategies, two of them are particularly well-suited to be implemented on the GPU architecture, namely: (i) data-level parallelism (DP), which is very simple and remarkably efficient for large datasets; and (ii) program- and data-level parallelism (PDP), which is not as simple as DP, but exhibits the same degree of efficiency for large datasets and has the advantage of being efficient for small datasets as well.

Up to date, only a few large and real-world problems have been solved by GP with the help of the massive parallelism of GPUs. This suggests that the potential of GP is yet under-explored, indicating that the next big step concerning GP on GPUs may be its application to those challenging problems. In several domains, such as in bio-informatics, the amount of data is growing quickly, making it progressively difficult for specialists to manually infer models and the like. Heterogeneous computing, combining the computational power of different devices, as well as the possibility of programming uniformly for any architecture and vendor, is also an interesting research direction to boost the performance of GP. Although offering both advantages, OpenCL is still fairly unexplored in the field of evolutionary computation.

Finally, although optimization techniques have not been thoroughly discussed in this chapter, this is certainly an important subject. Thus, the reader is invited to consult the related material found in [4], and also general GPU optimizations techniques from the respective literature.

Acknowledgements

The authors would like to thank the support provided by CNPq (grants 308317/2009-2 and 300192/2012-6) and FAPERJ (grant E-26/102.025/2009).

[14] The careful reader will note that this index calculation leads to an efficient coalesced memory access pattern [1, 32].

Author details

Douglas A. Augusto and Heder S. Bernardino
Laboratório Nacional de Computação Científica (LNCC/MCTI), Rio de Janeiro, Brazil

Helio J. C. Barbosa
Laboratório Nacional de Computação Científica (LNCC/MCTI), Rio de Janeiro, Brazil
Federal University of Juiz de Fora (UFJF), Computer Science Dept., Minas Gerais, Brazil

6. References

[1] Advanced Micro Devices [2010]. *AMD Accelerated Parallel Processing Programming Guide - OpenCL*.

[2] Ando, J. & Nagao, T. [2007]. Fast evolutionary image processing using multi-gpus, *Proc. of the International Conference on Systems, Man and Cybernetics*, pp. 2927 –2932.

[3] Arenas, M. G., Mora, A. M., Romero, G. & Castillo, P. A. [2011]. Gpu computation in bioinspired algorithms: a review, *Proc. of the international conference on Artificial neural networks conference on Advances in computational intelligence*, Springer-Verlag, pp. 433–440.

[4] Augusto, D. A. & Barbosa, H. J. [2012]. Accelerated parallel genetic programming tree evaluation with opencl, *Journal of Parallel and Distributed Computing* (0): –.
URL: *http://www.sciencedirect.com/science/article/pii/S074373151200024X*

[5] Augusto, D. A. & Barbosa, H. J. C. [2000]. Symbolic regression via genetic programming, *Proceedings of the VI Brazilian Symposium on Neural Networks*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 173–178.

[6] Banzhaf, W., Harding, S., Langdon, W. B. & Wilson, G. [2009]. Accelerating genetic programming through graphics processing units., *Genetic Programming Theory and Practice VI*, pp. 1–19.
URL: *http://dx.doi.org/10.1007/978-0-387-87623-8_15*

[7] Cano, A., Zafra, A. & Ventura, S. [2010]. Solving classification problems using genetic programming algorithms on gpus, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6077 LNAI(PART 2): 17–26.

[8] Cano, A., Zafra, A. & Ventura, S. [2012]. Speeding up the evaluation phase of gp classification algorithms on gpus, *Soft Computing* 16(2): 187–202.

[9] Chitty, D. M. [2007]. A data parallel approach to genetic programming using programmable graphics hardware, *in* D. Thierens, H.-G. Beyer, J. Bongard, J. Branke, J. A. Clark, D. Cliff, C. B. Congdon, K. Deb, B. Doerr, T. Kovacs, S. Kumar, J. F. Miller, J. Moore, F. Neumann, M. Pelikan, R. Poli, K. Sastry, K. O. Stanley, T. Stutzle, R. A. Watson & I. Wegener (eds), *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, Vol. 2, ACM Press, London, pp. 1566–1573.
URL: *http://www.cs.bham.ac.uk/ wbl/biblio/gecco2007/docs/p1566.pdf*

[10] Ebner, M. [2009]. A real-time evolutionary object recognition system, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5481 LNCS: 268–279. cited By (since 1996) 1.
URL: *http://www.scopus.com/inward/record.url?eid=2-s2.0-67650697120&partnerID=40&md5=1a9de902eb5649a01e3e87c222a79ee3*

[11] Ebner, M., Reinhardt, M. & Albert, J. [2005]. Evolution of vertex and pixel shaders, *Proceedings of the European Conference on Genetic Programming Genetic Programming – EuroGP*, Vol. 3447 of *LNCS*, Springer Berlin / Heidelberg, pp. 142–142.

[12] Ferreira, C. [2006]. *Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence*, 2 edn, Springer.

[13] Franco, M. A., Krasnogor, N. & Bacardit, J. [2010]. Speeding up the evaluation of evolutionary learning systems using gpgpus, *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO '10, ACM, New York, NY, USA, pp. 1039–1046.
URL: *http://doi.acm.org/10.1145/1830483.1830672*

[14] Garland, M. & Kirk, D. B. [2010]. Understanding throughput-oriented architectures, *Commun. ACM* 53: 58–66.

[15] Gaster, B., Kaeli, D., Howes, L., Mistry, P. & Schaa, D. [2011]. *Heterogeneous Computing With OpenCL*, Elsevier Science.
URL: *http://books.google.com.br/books?id=qUJVU8RH3jEC*

[16] Harding, S. [2008]. Evolution of image filters on graphics processor units using cartesian genetic programming, pp. 1921–1928. cited By (since 1996) 3.
URL: *http://www.scopus.com/inward/record.url?eid=2-s2.0-55749093400&partnerID=40&md5=fddf39574ff1025ad80adf204ccb451f*

[17] Harding, S. & Banzhaf, W. [2007a]. Fast genetic programming and artificial developmental systems on gpus, *Proc. of the International Symposium on High Performance Computing Systems and Applications*, p. 2.

[18] Harding, S. & Banzhaf, W. [2007b]. Fast genetic programming on GPUs, *Proc. of the European Conference on Genetic Programming – EuroGP*, Vol. 4445 of *LNCS*, Springer, Valencia, Spain, pp. 90–101.

[19] Harding, S. & Banzhaf, W. [2011]. Implementing cartesian genetic programming classifiers on graphics processing units using gpu.net, *Proceedings of the Conference Companion on Genetic and evolutionary computation – GECCO*, ACM, pp. 463–470.

[20] Harding, S. L. & Banzhaf, W. [2009]. Distributed genetic programming on GPUs using CUDA, *Workshop on Parallel Architectures and Bioinspired Algorithms*, Universidad Complutense de Madrid, Raleigh, NC, USA, pp. 1–10.

[21] Hennessy, J. & Patterson, D. [2011]. *Computer Architecture: A Quantitative Approach*, The Morgan Kaufmann Series in Computer Architecture and Design, Elsevier Science.
URL: *http://books.google.com.br/books?id=v3-1hVwHnHwC*

[22] Hillis, W. D. & Steele, Jr., G. L. [1986]. Data parallel algorithms, *Commun. ACM* 29: 1170–1183.

[23] Juille, H. & Pollack, J. B. [1996]. Massively parallel genetic programming, *in* P. J. Angeline & K. E. Kinnear, Jr. (eds), *Advances in Genetic Programming 2*, MIT Press, Cambridge, MA, USA, chapter 17, pp. 339–358.

[24] Kaul, K. & Bohn, C.-A. [2006]. A genetic texture packing algorithm on a graphical processing unit, *Proceedings of the International Conference on Computer Graphics and Artificial Intelligence*.

[25] Khronos OpenCL Working Group [2011]. *The OpenCL Specification, version 1.2*.
URL: *http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf*

[26] Kirk, D. & Hwu, W. [2010]. *Programming Massively Parallel Processors: A Hands-On Approach*, Applications of GPU Computing Series, Morgan Kaufmann Publishers.
URL: *http://books.google.com.br/books?id=qW1mncii_6EC*

[27] Langdon, W. [2008]. Evolving genechip correlation predictors on parallel graphics hardware, pp. 4151–4156. cited By (since 1996) 0.
URL: *http://www.scopus.com/inward/record.url?eid=2-s2.0-55749103342&partnerID=40&md5=028c81cb3bb1b8380f2f816b8e50b1f4*

[28] Langdon, W. & Banzhaf, W. [2008]. A SIMD interpreter for genetic programming on GPU graphics cards, *Genetic Programming*, pp. 73–85.

[29] Langdon, W. & Harrison, A. [2008]. Gp on spmd parallel graphics hardware for mega bioinformatics data mining, *Soft Computing* 12(12): 1169–1183. cited By (since 1996) 13.
URL: *http://www.scopus.com/inward/record.url?eid=2-s2.0-49049115131&partnerID=40&md5=4d332814a77dc0233bed7ff3184a6ccb*

[30] Luo, Z. & Liu, H. [2006]. Cellular genetic algorithms and local search for 3-sat problem on graphic hardware, *Proc. of the Congress on Evolutionary Computation – CEC*, pp. 2988 –2992.

[31] NVIDIA Corporation [2007]. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*.
URL: *http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf*

[32] NVIDIA Corporation [2010]. *OpenCL Best Practices Guide*.

[33] Pospichal, P., Murphy, E., O'Neill, M., Schwarz, J. & Jaros, J. [2011]. Acceleration of grammatical evolution using graphics processing units: Computational intelligence on consumer games and graphics hardware, pp. 431–438. cited By (since 1996) 0.
URL: *http://www.scopus.com/inward/record.url?eid=2-s2.0-80051950282&partnerID=40&md5=46bb1910d0121a948a804f8aa62308eb*

[34] Robilliard, D., Marion-Poty, V. & Fonlupt, C. [2008]. Population parallel gp on the g80 gpu, *Artificial Intelligence and Lecture Notes in Bioinformatics* 4971: 98–109.

[35] Robilliard, D., Marion, V. & Fonlupt, C. [2009]. High performance genetic programming on gpu, *Proceedings of the 2009 workshop on Bio-inspired algorithms for distributed systems*, BADS '09, ACM, New York, NY, USA, pp. 85–94.

[36] Wilson, G. & Banzhaf, W. [2008]. Linear genetic programming gpgpu on microsoft's xbox 360, *2008 IEEE Congress on Evolutionary Computation, CEC 2008*, pp. 378–385. cited By (since 1996) 4.
URL: *http://www.scopus.com/inward/record.url?eid=2-s2.0-55749108355&partnerID=40&md5=304d6784cd00eac6e253229092ba7788*

[37] Wilson, G. & Banzhaf, W. [2009]. Deployment of cpu and gpu-based genetic programming on heterogeneous devices, *Proceedings of the Conference Companion on Genetic and Evolutionary Computation Conference*, Late Breaking Papers, ACM, pp. 2531–2538.

[38] Wilson, G. & Banzhaf, W. [2010]. Deployment of parallel linear genetic programming using gpus on pc and video game console platforms, *Genetic Programming and Evolvable Machines* 11(2): 147–184. cited By (since 1996) 2.
URL: *http://www.scopus.com/inward/record.url?eid=2-s2.0-77954814128&partnerID=40&md5=8e8091dedc7d49dfccc20e0f569af0ce*

[39] Wong, M.-L. & Wong, T.-T. [2006]. Parallel hybrid genetic algorithms on consumer-level graphics hardware, *Proc. of the Congress on Evolutionary Computation*, pp. 2973 –2980.

[40] Wong, M., Wong, T. & Fok, K. [2005]. Parallel evolutionary algorithms on graphics processing unit, *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, Vol. 3, pp. 2286–2293 Vol. 3.

[41] Yu, Q., Chen, C. & Pan, Z. [2005]. Parallel genetic algorithms on programmable graphics hardware, *Proc. of the international conference on Advances in Natural Computation*, Springer-Verlag Berlin, pp. 1051–1059.