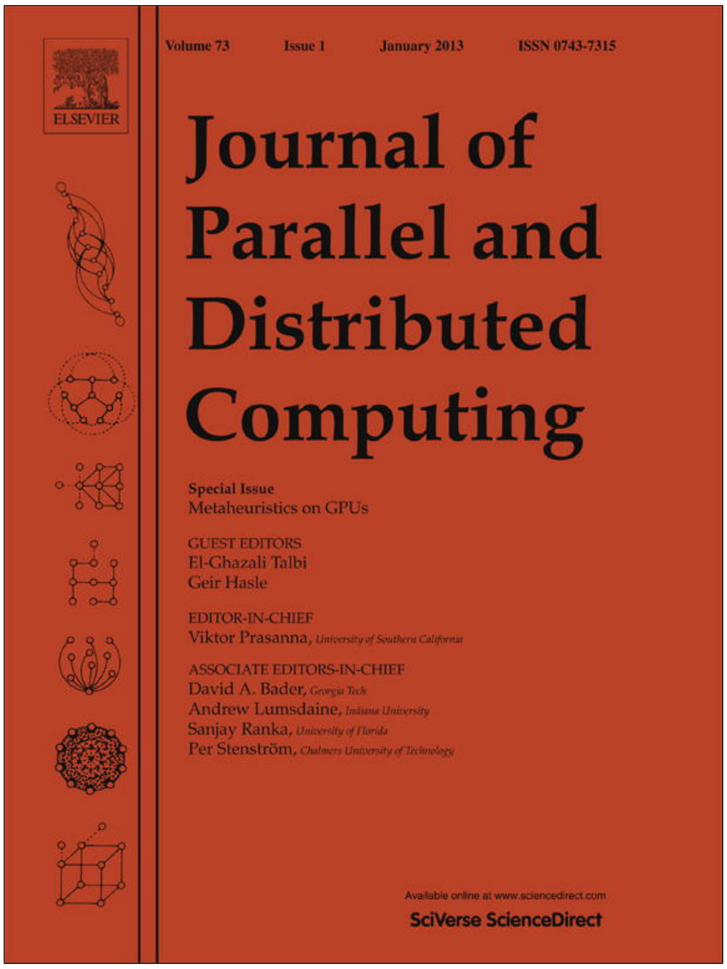

Volume 73 Issue 1 January 2013 ISSN 0743-7315
ELSEVIER
Journal of Parallel and Distributed Computing
Special Issue
Metaheuristics on GPUs
GUEST EDITORS
El-Ghazali Talbi
Geir Hasle
EDITOR-IN-CHIEF
Viktor Prasanna, University of Southern California
ASSOCIATE EDITORS-IN-CHIEF
David A. Bader, Georgia Tech
Andrew Lumsdaine, Indiana University
Sanjay Ranka, University of Florida
Per Stenström, Chalmers University of Technology
Available online at www.sciencedirect.com
SciVerse ScienceDirect

J. Parallel Distrib. Comput. 73 (2013) 86–100

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Accelerated parallel genetic programming tree evaluation with OpenCL

Douglas A. Augusto[a,*], Helio J.C. Barbosa[a,b]

[a] Laboratório Nacional de Computação Científica, Petrópolis, RJ, Brazil
[b] DCC/UFJF, Juiz de Fora, MG, Brazil

ARTICLE INFO

Article history:
Received 19 April 2011
Received in revised form
26 December 2011
Accepted 24 January 2012
Available online 30 January 2012

Keywords:
OpenCL
GP–GPU
Parallel genetic programming
Accelerated tree evaluation

ABSTRACT

Inspired by the process of natural selection, genetic programming (GP) aims at automatically building arbitrarily complex computer programs. Being classified as an "embarrassingly" parallel technique, GP can theoretically scale up to tackle very diverse problems by increasingly adding computational power to its arsenal. With today's availability of many powerful parallel architectures, a challenge is to take advantage of all those heterogeneous compute devices in a portable and uniform way. This work proposes both (i) a transcription of existing GP parallelization strategies into the OpenCL programming platform; and (ii) a freely available implementation to evaluate its suitability for GP, by assessing the performance of parallel strategies on the CPU and GPU processors from different vendors. Benchmarks on the symbolic regression and data classification domains were performed. On the GPU we could achieve 13 billion node evaluations per second, delivering almost 10 times the throughput of a twelve-core CPU.

1. Introduction

Proposed as a technique to automatically build arbitrarily complex computer programs, genetic programming (GP) [19] is beyond doubt a highly ambitious paradigm from the field of evolutionary computation. Strictly speaking, there are only two requirements that need to be satisfied in order to enable GP as a potential solver for a certain problem: that (i) *the problem's solution could be described as a computer program*, and (ii) *there exists a graduated evaluation metric that given two candidate solutions is capable of indicating – with reasonable accuracy – which one is better*. The first condition says that candidate solutions must be represented on a GP system, hence being able to undergo sustained genetic operations, like crossover and mutation, which is the way programs evolve in GP. The latter allows for the implementation of the principle of natural selection, which is an essential component of every evolutionary computation algorithm.

It follows that, at least in theory, GP can be applicable not only to countless problems, but also to a large range of complexities. Although this fact emphasizes how versatile GP can be, there is a consequence: genetic programming is destined to be computationally demanding, whatever the availability of computational resources, because there will always be a more complex problem to be solved. This does not mean, however, that GP is an inefficient "processor-hungry" technique; being capable of tackling increasingly difficult problems as the computational power advances means that GP is able to take full advantage of an arbitrary amount of resources in order to handle a problem.

Most optimization algorithms can somehow take advantage of the availability of computational resources, but what distinguishes GP from most of them is that GP is able to efficiently explore a wider class of hardware architectures at their full extent. The main reason behind that is the high degree of parallelism endowed by GP, both fine- and coarse-grained [2].

1.1. Parallel architectures

The microprocessor industry players have agreed that keeping up with Moore's Law [23] is becoming increasingly difficult as the manufacturing process involved in making microchips approaches the physical limits of the technology [14]. In other words, unless some breakthrough in the field of manufacturing and materials is accomplished, there is not much room for further improvement on the transistor's density per area; moreover, even before the physical limits of the technology are effectively reached, the costs of producing denser and denser microprocessors may not be commercially viable.

An alternative approach to ensure continuous advance in the processor performance is to soften the focus on the transistor's density *per se* and instead employ many independent "not-so-dense" microchips coupled in a unique processor; that is, the parallel multi-core design. However, this approach does not improve

[*] Corresponding author.
E-mail addresses: douglas@lncc.br, daaugusto@gmail.com (D.A. Augusto), hcbm@lncc.br (H.J.C. Barbosa).

0743-7315/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2012.01.012

single-core performance, but rather the *total* performance, which is fully achieved when all processor cores operate, simultaneously, at full capacity.

The multi-core design has become ubiquitous in almost every computing platform and so has displaced the dominance of the serial single-core processors. Not merely that, but the demand for computational power has pressed forward all kinds of parallel architectures and has made them mainstream, the future trend.

According to [8], the variety of parallel processors can be categorized into two classes, namely the *latency*-and the *throughput*–oriented processors. The first class concerns those processors whose design is optimized towards the fast processing of sequential tasks, even for multi-core architectures. The second one refers to those processors in which the goal is to process the greatest amount of tasks per unit of time. A representative of the latency-oriented class is the ordinary CPU processor, be it single- or multi-core. As for the throughput-oriented ones, a noteworthy example is the Graphics Processing Unit (GPU) architecture.

1.2. Related work

One of the early attempts to make use of the GPUs to accelerate the execution of evolutionary algorithms was done by Yu et al. [33]. They implemented a parallel genetic algorithm on a Nvidia GeForce 6800GT GPU using the *Cg* language. Not only the fitness evaluation was implemented on the GPU, but also the genetic operators. Compared with the CPU, the authors have reported peak speedup of about 17 for the fitness evaluation. Harding and Banzhaf [11] implemented genetic programming on the GPU using the *Accelerator* toolkit. They reported excellent results on a Nvidia GeForce 7300 GO on different kinds of benchmarks. Langdon and Banzhaf [20] used the *Rapidmind* framework to implement a SIMD interpreter for linear genetic programming on the GPU. They achieved 895 million GPop/s on the Mackey–Glass chaotic time series using a Nvidia 8800 GTX.[1] Robilliard et al. [27,26] have implemented and evaluated different GP population-parallel approaches using the Nvidia's *Compute Unified Device Architecture* (CUDA). Their peak performance on a Nvidia G80 GPU was an impressive 2.8 billion GPop/s. Also using the CUDA framework, Maitre et al. [21] obtained a peak speedup of 250 when comparing the performance of a dual GPU Nvidia GTX-295 (using half of its cores) with an Intel quad-core Q8200 CPU in sequential mode.

Except for a few exceptions, such as the use of the proprietary GPU.NET [12], it seems that most of the recent studies exploiting the GPU power have been built exclusively around the CUDA technology. While CUDA is a powerful and mature toolkit, it is a closed technology and only works on Nvidia GPUs, precluding the use of all the computational power available from other vendors and architectures.

Although the related works found in the literature have shown the remarkable power of the GPU in speeding up the genetic programming execution using different frameworks, so far none of them have implemented and evaluated a parallel genetic programming using the OpenCL specification, which holds the following properties: (i) open standard; (ii) portable across many parallel architectures and vendors; and (iii) allows low-level access to hardware. One of the consequences is that there is neither references detailing how to map a parallel GP into the OpenCL framework nor studies directly comparing the accelerated parallel genetic programming performance on multi-core CPUs and GPUs from different vendors.

[1] GPop/s stands for *genetic programming operations per second*, which in turn usually means the average number of tree nodes evaluated in a second.

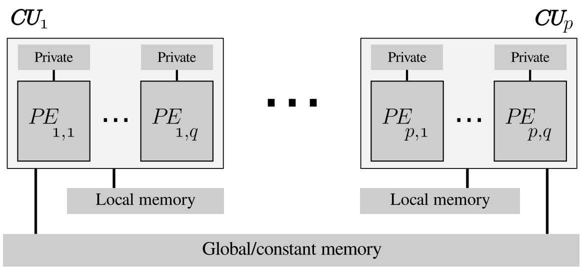


Fig. 1. Conceptual compute device architecture.

1.3. Text organization

This paper is organized as follows. Section 2 introduces OpenCL, a portable multi-vendor language for parallel programming on heterogeneous devices. Different strategies of parallel implementation of a genetic programming system using OpenCL for speeding up tree evaluation on both CPU and GPU devices are described in Section 3. In Section 4, computational experiments comparing raw performance in terms of GPop/s for different class of problems, parallel strategies, hardware architectures, and vendors are presented. Optimization techniques implemented by GPOCL are presented in Section 5. Finally, Section 6 points out conclusions and some directions for future work.

2. Open computing language—OpenCL

OpenCL is an open standard [31] for uniform and portable parallel programming across heterogeneous computing platforms. It aims to provide low and high level access to data- or task-parallel devices, either individually or simultaneously. Besides the conventional multi-core CPUs and GPUs from multiple vendors, OpenCL also supports some other parallel devices, such as Field-Programmable Gate Array (FPGA), digital signal processors (DSP), IBM's Cell Broadband engine [15], and more.

2.1. Programming model

The OpenCL's application program interface (API) exposes the access to the parallel compute devices by querying the *host* system to determine which *platforms* are present. A platform is a particular OpenCL implementation installed on the system. From the platform it is possible to discover which compatible computing *devices* are available and then select the ones according to some criteria defined by the programmer; a criterion, for instance, may be a particular architecture, like CPU or GPU. Given one or more devices, a *memory buffer* can be allocated on them, and a *command queue*, responsible for *kernel* submissions to the devices, can be set up. Roughly speaking, an OpenCL kernel is a C-like function designed to be executed in parallel on each individual processing element (e.g. CPU cores or GPU processors) of a compute device. The kernel has access to many levels of the device memory and can read from, process and write data to it.

2.2. Compute device abstraction

Fig. 1 illustrates a conceptual compute device architecture according to the OpenCL model, which is shared by all supported OpenCL devices. Each device has p compute units (CU), and each compute unit is composed of q processing elements (PE). The conventional CPU has only a single processing element ($q = 1$) per compute unit; p is the number of cores.

The compute device has a *global* memory, which is shared by all the device's processing elements. There is also an optimized global memory for read-only access, the *constant* memory. Each compute unit owns a potentially fast-access memory, termed *local* memory, whose access is restricted to the processing elements that belong to the same compute unit. Finally, each processing element has also a very fast-access *private* memory, which is not shared at all.

Although the GPU is historically a data-parallel SIMD architecture [10], modern GPU devices exhibit some aspects of the SPMD model, that is, they are capable of simultaneously executing *different* instructions from a single stream of instructions ("program") over multiple data. As each compute unit has enough resources to independently manage its own instruction address within the program [26], it is technically possible to execute a different instruction per compute unit. This fact can be exploited to enable very efficient use of the GPU architecture for accelerated GP tree evaluation, as presented in Section 3.4.

The terms *processing element* and *compute unit* are hardware-related names. When dealing with the OpenCL interface, the high-level terms *work-item* and *work-group* are used instead. A work-item is a kernel instance that is usually executed on a processing element, whereas a work-group is a collection of work-items that execute on a single compute unit. *Local size*, sometimes also referred to as *work-group size*, is the number of work-items in a work-group, while *global size* is the total number of work-items.

3. Parallel genetic programming with OpenCL

This section discusses the main aspects of a genetic programming implementation that uses OpenCL to exploit the multi-core CPU and GPU processors. The actual program is called GPOCL, a freely available high-performance implementation written in C++ and OpenCL that implements a canonical GP system [19] using a prefix linear tree representation [3,17].[2]

3.1. General structure

The main routine of GPOCL performs the following high-level procedures in the given order.

1. *OpenCL initialization*: this is the step where the general OpenCL-related tasks are initialized (Section 2). An OpenCL *platform* is discovered, a *device* representation is created and its specifications are extracted, and a *command queue* is created.
2. *Calculating n-dimensional ranges*: defines how much parallel work there will be and how they are distributed among the compute units; in other words, *local size* and *global size* parameters are calculated (Sections 3.3 and 3.4).
3. *Memory buffers creation*: in this phase all global memory regions accessed by the OpenCL kernels are allocated on the *device* and possibly initialized. The fitness cases are transferred and enough space is reserved for the *population* and *error* vectors.
4. *Kernel building*: an OpenCL kernel, relative to a given strategy of parallelization (Sections 3.3 and 3.4), is compiled just-in-time, targeting the compute device. Before the actual compilation, an optimized GP interpreter is assembled (Section 3.2.1) and helper functions and runtime parameters/flags are embodied in the kernel source (Section 5.4).
5. *Evolving*: this iterative routine implements the actual genetic programming dynamics, as described in Section 3.2.

3.2. Main evolutionary algorithm

The steps described in Algorithm 1 belong to a standard generational GP algorithm. Most of these steps are performed on the *host* machine and sequentially, but the costly procedure of *evaluation* of programs (found at line 2 and 15) are done in parallel by the target compute device, be it the CPU or GPU. How

Algorithm 1: GPOCL's evolutionary algorithm

```
   Create (randomly) the initial population $P$;
2  Evaluate($P$);
   for $generation \leftarrow 1$ to $N_G$ do
       Copy the best (elitism) programs of $P$ to the temporary
       population $P_{tmp}$;
       while $|P_{tmp}| < |P|$ do
           Select and copy from $P$ two fit programs, $p_1$ and $p_2$;
           if [probabilistically] crossover then
               Recombine $p_1$ and $p_2$, generating $p'_1$ and $p'_2$;
               $p_1 \leftarrow p'_1$; $p_2 \leftarrow p'_2$;
           if [probabilistically] mutation then
               Apply mutation in $p_1$ and $p_2$, creating $p'_1$ and $p'_2$;
               $p_1 \leftarrow p'_1$; $p_2 \leftarrow p'_2$;
           Insert $p_1$ and $p_2$ into $P_{tmp}$;
       $P \leftarrow P_{tmp}$; then reset $P_{tmp}$;
15     Evaluate($P$);
   return the best program found;
```

the evaluation is parallelized, though, depends on the particular strategy, which will be discussed later on.

The evaluation step itself does not do much—the hard work is done mostly by the OpenCL kernels. Basically, three things happen within `Evaluate`(P):

1. *Population transfer*: all programs of P are transferred to the target compute device. For the GPU, due to the physical separation between the host memory and the device memory, an explicit copy or synchronization is required.
2. *Kernel execution*: for any non-trivial problem, this is the most demanding phase. Here, the entire recently transferred population is evaluated – by interpreting each program over each fitness case – on the compute device. Fortunately, this step can be done both in parallel as well accelerated by GPUs.
3. *Error retrieval*: after being computed and accumulated in the previous step, the prediction errors of the whole population need to be transferred to the host so that this information is available to the evolutionary process. Again, if the compute devices and the host are not the same, an explicit transfer is necessary.

3.2.1. GP tree interpreter

The GP tree interpreter is a function that takes a *genetic program* and its *inputs*, and then executes the instructions stored on the tree's nodes in a flow given by the tree's structure. An interpreter is commonly implemented as a stack-based procedure [17] in which the partial results coming from the children nodes are pushed into a stack and pulled back when a parent operator needs the accumulated result.

The interpreter is also a mechanism through which a task-parallel (MIMD [10]) processor can be emulated on a native data-parallel (SIMD) processor like the GPU [16,20]. The interpreter acts as the single task being executed on the SIMD processor while the population of programs and their inputs are treated as data. Therefore, the interpreter makes possible the parallel processing of fitness cases as well as of multiple programs.

A pseudo-code for a GP tree interpreter is presented by the function `Interpreter`(*program*, *n*). The argument *program* is an array of integers encoding a tree (the structure, operators, operands and variables) via a prefix linear representation [3], whereas n is the index of the training data point currently being evaluated. The function `INDEX()` extracts the operator/operand

[2] GPOCL is Free Software and can be found at http://gpocl.sf.net.

code, VALUE() returns the actual numerical value of a constant or the index if the argument is a variable, X_n is the n-th row of the training dataset, PUSH() puts a number into the stack and, finally, POP pulls back a value from the top of the stack. At the end of the interpretation process, the program's predicted output is left on the top of the stack, so this value is returned by the interpreter and then used to compute the program's prediction error.

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;
```

3.3. CPU strategy

Given the flexibility provided by the true general-purpose CPU architecture, it is easy to implement efficiently any parallelization strategy on this processor. Additionally, since the current CPUs have at most a dozen cores, in practice any genetic programming application, be it evaluating in parallel the fitness cases (*fitness-parallel*) or the population of programs (*population-parallel*), can promptly keep a high processor occupancy.

Because traditional multi-core CPUs do not support in hardware the concept of "processing elements" as GPUs do, for efficiency reasons their $local_{size}$ is usually set to 1, hence assigning to each core a work-item. Moreover, as the multi-core CPU is a native task-parallel processor, it is natural to dedicate each core to evaluate one program at a time, i.e. the population-parallel approach, by setting:

$$\text{global}_{\text{size}} = \text{population}_{\text{size}}, \quad (1)$$

which makes the number of programs equal to the amount of tasks to be done, where each task is a program evaluation routine given by the OpenCL pseudo-kernel shown in Algorithm 2.

The first step is to get an identifier, the global_{id}, for each work-item, which in this case represents the index of the program being evaluated. Then, the corresponding program is assigned to the variable *program* by means of the helper function NthProgram(n), which returns a reference to the n-th program of the population. Next, for each fitness case n, the program is interpreted on it and the resulting prediction is compared with the actual value given by $Y[n]$. The error is accumulated in the course of this iteration and finally, when all fitness cases have been evaluated, the private *error* is stored on the global vector of errors E.

Algorithm 2: CPU'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;
```

3.4. GPU strategies

This section proposes implementations in OpenCL of three main strategies of parallel program tree evaluation on the GPU using an interpreter: *fitness-parallel* (FP), *population-parallel per processing element* (PPPE), and *population-parallel per compute unit* (PPCU). The population-parallel strategies are transcriptions into OpenCL of the schemes described in [28,26], namely *ThreadGP* and *BlockGP*, originally implemented in CUDA.

The *fitness-parallel* strategy is probably the simplest one, both conceptually and in terms of implementation. In this approach, only one program at time is dispatched to the GPU to be evaluated. One instance of the tree interpreter is executed per processing element, each one managing a different fitness case for a unique program. The FP strategy is truly data-parallel, and thus fits perfectly on the GPU architecture. The problem is that, since modern GPUs have a large number of processing elements, and to achieve optimal performance a bunch of data should be assigned to each one of these processing elements at a time, the fitness-parallel approach requires large datasets in order to maximize the throughput.

A way of trying to overcome the efficient applicability of genetic programming when there are not enough fitness cases to fully utilize the processing power of the GPU, is to do the opposite of the FP strategy and evaluate simultaneously the entire population of programs; after all, by using an interpreter both fitness cases and programs can be seen as data. The naive implementation of this idea is to assign one program evaluation per processing element, with the fitness cases being processed serially; this is the *population-parallel per processing element* (PPPE) strategy, and even a few fitness cases are enough to sustain its maximum performance. The implementation is still simple, but this strategy does not fit well the GPU architecture. The problem is that, although modern GPU architectures can technically execute different instructions simultaneously, this SPMD feature is exclusive among compute units, not among processing elements within a compute unit, which remain as SIMD processors. Hence, whenever the instructions being executed within a compute unit diverge, they are executed in batches, each batch sharing the same instruction, one batch at a time. It is easy to notice that interpreting a different program per processing element will likely cause intense instruction divergence and therefore will tend to achieve weak overall performance.

Another proposal to address the issue related to the FP strategy while minimizing instruction divergence is to use a mixed approach, here referred to as *population-parallel per compute unit*, or simply PPCU. As said, the modern GPU architecture is able to act as a SPMD processor among the blocks of compute units. This opens the possibility to process a different program per compute unit without causing much instruction divergence. In the PPCU strategy, both the population of programs and fitness cases are parallelized. While different programs are evaluated simultaneously on different compute units, the processing elements within each compute elements take care, in parallel, of the whole training dataset. Since internally to each compute unit the processing elements will be interpreting the same program, the event of instruction divergence is unlikely, but not impossible. Instruction divergence can still occur when the execution flow depends on the underlying data being processed, as the case when there are conditional structures, such as *if-then*, in the program tree. This is known as *branch divergence* and affects the FP and PPPE strategies as well. Fortunately, in practice the negative effect caused by this weak form of branch divergence is minor and might be tolerated.[3]

The following three sections describe the pseudo-OpenCL kernel implementations for the discussed GPU strategies and how to calculate the n-dimensional range for each strategy.

3.4.1. Fitness-parallel—FP

There are two important things to be taken into account when calculating how to distribute the workloads among the GPU processors upon the definition of the n-dimensional range: proper load balancing and, obviously, respecting the hardware limits.

In the fitness-parallel strategy the goal is to evaluate all the fitness cases at once. Since the GPU architecture is organized as blocks of processors, i.e. compute units, it is necessary to partition the whole dataset among the compute units. In order to provide load balancing, $local_{\text{size}}$ can be defined as:

$$local_{\text{size}} = \min(local_{\text{max\_size}}, \lceil \text{dataset}_{\text{size}}/cu \rceil), \quad (2)$$

where cu is the number of compute units. This ensures that the device's maximum local size is not exceeded and also that even when there are fewer fitness cases than the total of processing elements, each compute unit will handle about the same workload. As for the $\text{global}_{\text{size}}$, it is calculated as

$$\text{global}_{\text{size}} = \lceil \text{dataset}_{\text{size}}/local_{\text{size}} \rceil \times local_{\text{size}}, \quad (3)$$

to guarantee that $\text{global}_{\text{size}}$ is divisible by $local_{\text{size}}$, as required by the OpenCL specification.

The GPU FP's OpenCL kernel is described in Algorithm 3. Each work-item is responsible for executing an instance of the kernel, but each one has a unique identifier, which is stored on global_{id} and determines which fitness case will be processed. Since it may happen that $\text{dataset}_{\text{size}}/cu$ is not evenly divisible, rounding up this expression (as in Eq. (2)) will make $\text{global}_{\text{size}}$ greater than the actual number of fitness cases given by $\text{dataset}_{\text{size}}$, thus the check (at the second line) is required to avoid accessing elements outside the valid range.

In the FP's kernel, the same program p is evaluated on the whole training dataset at once. Then, the prediction error is accumulated from all work-items (indexed from 0 to $\text{global}_{\text{size}} - 1$) via the `ErrorReduction()` procedure and, finally, the program p's error is stored on the global error vector E.

Algorithm 3: GPU FP's OpenCL kernel

```
$global_{id} \leftarrow$ get_global_id();
if $global_{id} < dataset_{size}$ then
    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($0, \ldots, global_{size} - 1$);
```

The error reduction procedure, illustrated in Fig. 2, is a parallel algorithm that computes in $\log_2(N)$ steps the sum of errors in a vector [13]. Besides being parallel, this procedure is also optimized in order to ensure *coalesced* memory access, taking advantage of the fact that on the GPU, accessing sequential memory address in parallel is more efficient than accessing non-contiguous addresses [1,25].

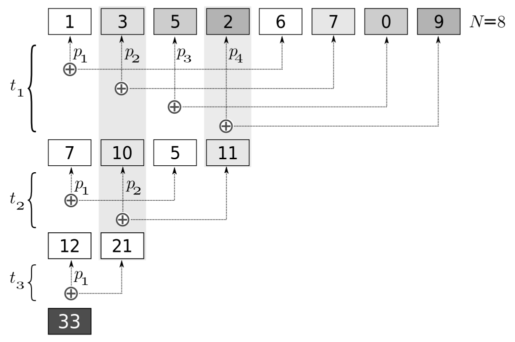


Fig. 2. $O(\log_2 N)$ parallel reduction with sequential addressing.

3.4.2. Population-parallel per processing element—PPPE

Following the same idea from the FP's n-dimensional range calculation, i.e. evenly distributing the workload along the compute units but at the same time avoiding to exceed the hardware's maximum local size, $local_{\text{size}}$ is calculated as:

$$local_{\text{size}} = \min(local_{\text{max\_size}}, \lceil \text{population}_{\text{size}}/cu \rceil), \quad (4)$$

and $\text{global}_{\text{size}}$ as:

$$\text{global}_{\text{size}} = \lceil \text{population}_{\text{size}}/local_{\text{size}} \rceil \times local_{\text{size}}. \quad (5)$$

As seen in Algorithm 4, the PPPE's kernel is very similar to the CPU's kernel from Algorithm 2. The only minor difference is the existence of the range check at the second line that, as in the FP's kernel, is used to avoid out-of-range access due to the rounding up.

One can notice that in the PPPE's kernel there is no need for error reduction. Like in the CPU's kernel, the total error relative to a program is entirely accumulated in each instance of the kernel.

Algorithm 4: GPU PPPE's OpenCL kernel

```
$global_{id} \leftarrow$ get_global_id();
if $global_{id} < population_{size}$ then
    $program \leftarrow$ NthProgram($global_{id}$);
    $error \leftarrow 0.0$;
    for $n \leftarrow 0$ to $dataset_{size} - 1$ do
        $error \leftarrow error +$ |Interpreter( $program, n$ ) $- Y[n]$|;
    $E[global_{id}] \leftarrow error$;
```

3.4.3. Population-parallel per compute unit—PPCU

The GPU PPCU's OpenCL kernel mixes components from both fitness- and population-parallel strategies, and therefore is a bit more complicated. Its $local_{\text{size}}$ is computed as

$$local_{\text{size}} = \begin{cases} \text{dataset}_{\text{size}} & \text{if dataset}_{\text{size}} < local_{\text{max\_size}} \\ local_{\text{max\_size}} & \text{otherwise,} \end{cases} \quad (6)$$

which means that when there are fewer fitness cases than the maximum allowed number of work-items in a given compute unit, then the local size should be shrunk to provide a one-by-one mapping; that is, each work-item would be responsible for each fitness case. Otherwise, when the dataset size does not fit into the

[3] Our performance results on the data classification domain, which is divergent, are almost as good as the results on the non-divergent symbolic regression domain (Section 4).

Table 1
Compute devices' specifications.

Feature	Device	
	Intel Xeon W3680	AMD Phenom II 1090T
Frequency	3.34 GHz	3.2 GHz
Cores (compute units)	12 (6 physical)	6
Power consumption	130 W (Max.)	125 W (Max.)
	Nvidia GTX-285	ATI Radeon HD5750
Compute units	30	9
Processing elements	240	144[a]
Global memory	1 GB	1 GB
Constant memory	64 KB	64 KB
Local memory (per compute unit)	16 KB	32 KB
Maximum local size	512	256
Power consumption	204 W (Max.)	86 W (Max.)

[a] AMD usually refers to those processing elements as *stream cores* [1].

Algorithm 5: GPU PPCU's OpenCL kernel

```
$local_{id} \leftarrow$ get_local_id();
$group_{id} \leftarrow$ get_group_id();
$program \leftarrow$ NthProgram($group_{id}$);
$error \leftarrow 0.0$;
for $i \leftarrow 0$ to $\lceil dataset_{size}/local_{size} \rceil - 1$ do
    $n \leftarrow i \times local_{size} + local_{id}$;
    if $n < dataset_{size}$ then
        $error \leftarrow error + |$Interpreter( $program, n$ ) $- Y[n]|$;
$E[group_{id}] \leftarrow$ ErrorReduction($0, \ldots, local_{size} - 1$);
```

maximum local size, the desired local size is set to the maximum possible value. As for the global size, it is calculated as

$$\text{global}_{\text{size}} = \text{population}_{\text{size}} \times local_{\text{size}}, \tag{7}$$

ensuring that its value is evenly divisible by the local size.

The PPCU's kernel is given in Algorithm 5. Since the kernel will need access to each individual work-item and work-group, their indices are obtained through the helper functions `get_local_id()` and `get_group_id()`, respectively. In this kernel, $local_{id}$ represents the index of a fitness case whereas group_{id} the index of the program to be evaluated.

Since the dataset size may be bigger than the local size (Eq. (6)), more than one step may be necessary in order to cover the whole dataset; this iterative procedure is given by the *for* loop. The variable n denotes the actual index of the fitness case to be evaluated on iteration i—one can note that this access pattern is coalesced, i.e. the memory region accessed by the work-group in each iteration is contiguous. Within the *for* loop a range check is used to only accept valid accesses; then, the error relative to each work-item is accumulated.

Finally, the parallel error reduction procedure (Fig. 2) is used to sum up the partial errors spread among the work-items and then store the final program's error on the global vector E.

4. Computational experiments

The results from two batches of experiments are presented in this section, whose goal is to compare and discuss the performance of the strategies of parallel implementation, as discussed in Section 3, on different processor architectures and models/vendors. What distinguishes these two set of experiments is the workload pattern they present to the compute devices. While the first batch simulates the evaluation of primitives commonly found in symbolic regression problems, i.e., mainly mathematical functions, the second batch presents primitives for data classification problems, usually dominated by conditional, relational, and logical operators.

4.1. Compute devices

Two CPU processors and two GPU devices were used to run the experiments. The top-of-the-line *Intel Xeon W3680* and *AMD Phenom II X6 1090T* processors are the CPU representatives, while the high-end *Nvidia GTX-285* and the mid-range *ATI Radeon HD5750* are the GPU counterparts. Table 1 shows their specifications.

4.2. Methodology

The experiments aim at assessing the performance solely in terms of throughput of node evaluations; that is, *genetic operations per second* (GPop/s). It is measured as the total number of executed tree nodes divided by the elapsed time taken for one generation. While calculating the stated performance, only the *population evaluation time*[4] (see Section 3) was taken into consideration.

In order to avoid biases introduced by the search process, some measures were taken: (i) the training datasets were randomly generated, with their real number points uniformly distributed in $[-1, 1]$; and (ii) only one generation per run was done.

4.2.1. Environment

The experiments were carried out on 64-bit GNU/Linux machines, with the following configuration. As regards the Nvidia GPU, it was used the official Nvidia developer driver version 285.05.23, installed on an Intel Xeon X5550 (@2.67 GHz) machine running Ubuntu 10.04 with Linux version 2.6.32. As for the ATI GPU, plugged in an Intel Xeon W3680 (@3.34 GHz) machine, the official driver Catalyst 11.11 was used. Both CPU compute devices, Intel Xeon W3680 and AMD Phenom II X6 1090T, and the ATI GPU, were operating under Debian GNU/Linux with kernel 3.1.0 and utilizing the AMD Accelerated Parallel Processing (APP) SDK version 2.5.

GP implementation. All the experiments were performed using GPOCL, whose implementation is discussed in Sections 3 and 5.

4.2.2. Parameters

One of the concerns regarding the choice of the parameters, specially the population and dataset size, was to prevent bias towards some architecture, in particular the GPUs. Instead of picking GPU-friendly values, such as "power-of-two" (see Section 5.2), generic ones were chosen so that it was possible

[4] This also includes the time required to gather and accumulate the prediction errors (difference between the predicted and the actual value) at each training data point.

Table 2
Fixed parameters.

Parameter	Value
Program tree size	50
Number of generations	1
Dataset dimensions	10 inputs ($x_1 \cdots x_{10}$), 1 output (y)

Table 3
Varying parameters.

Parameter	Range of values
Population size	{100, 1000, 5000, 10 000, 25 000, 50 000} programs
Dataset size	{100, 1000, 5000, 10 000, 25 000, 50 000} rows

to assess the robustness of the architecture when non-optimum parameters are used. Table 2 shows the parameters that remained constant throughout the experiments. For each combination of the parameters population and dataset size, as enumerated in Table 3, a total of thirty independent runs were performed, each one with a different random seed. The median throughput (GPop/s) was used as the estimated performance per configuration.

4.3. Results

Since there are two varying parameters being considered, the graphics are presented as 3D-plots, where the x-axis denotes the population size, the y-axis the dataset size, and finally the z-axis the resulting median GPop/s over 30 runs. Dark spots represent higher GPop/s while light ones lower GPop/s.

The next two sections present the experimental results, respectively, for the symbolic regression and data classification domains.

4.3.1. Symbolic regression

This batch of experiments, targeted at symbolic regression, tries to simulate the workload pattern typically found in problems from this domain, whose task involves the inference of mathematical expressions that better fit a given set of points. The basic arithmetic operators, two transcendental functions, and constants (fixed integer and real ephemeral constants) are used; more precisely, the function set is defined as $\mathbb{F} = \{+, -, \times, \div, \text{neg}, \sin, \cos\}$ and the terminal set as $\mathbb{T} = \{x_1 \cdots x_{10}, 1, -1, \mathbb{R}\}$.

The first symbolic regression benchmark, depicted in Fig. 3, shows the Intel Xeon's raw performance varying between 168 and 182 million GPop/s when using only one CPU core. The worst performance is found at the smallest dataset size. While the effect of the population size is minor, an increase in the raw performance can be observed as the dataset size approaches its maximum value.

When using all the 12 cores in parallel, the peak performance jumps from about 180 million to about 1.4 billion GPop/s, as shown in Fig. 4. This corresponds to a speedup of a bit less than 8, which is far below the expected linear speedup of 12—in other words, an efficiency E_{12} of approximately 65% ($E_p = S_p/p$, where p is the number of processors and S_p the speedup using p processors [30]). As described in Table 1, only half of the twelve cores in the Intel Xeon are in fact physical ones, the remaining six come via Intel's *Hyper-Threading* technology [22], which adds virtual cores to the processor to improve – usually to a less extent – the overall parallelism. It is easy to notice that the smaller the population and dataset, the worse the throughput.

As seen in Fig. 5, the sequential benchmark on the AMD Phenom presents a very similar pattern when compared to the Intel's counterpart (Fig. 3), but its average raw performance is roughly 75% of Intel's.

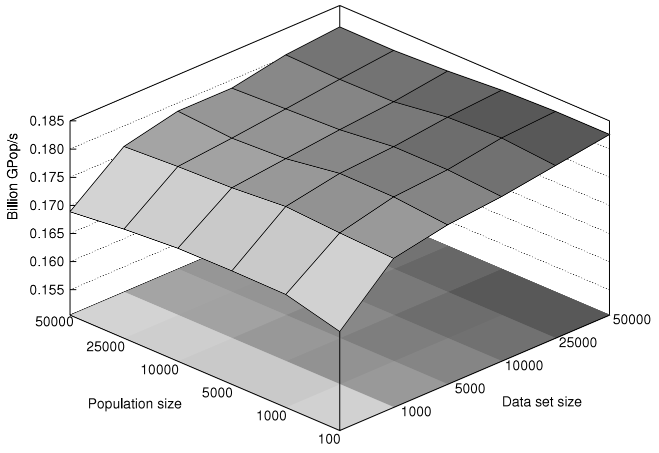


Fig. 3. Regression, CPU, sequential, Intel Xeon W3680.

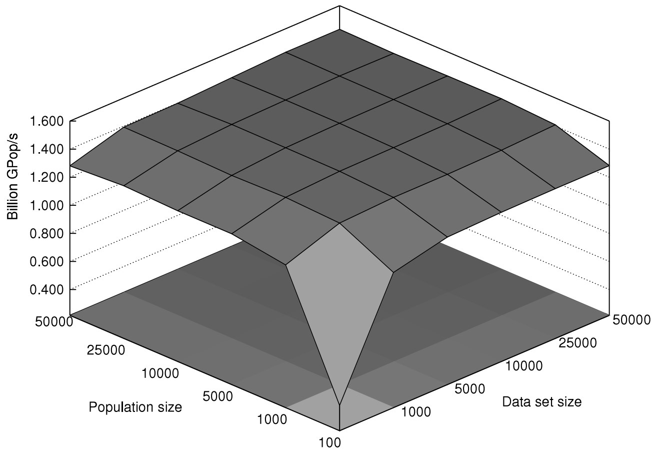


Fig. 4. Regression, CPU, 12-core parallel, Intel Xeon W3680.

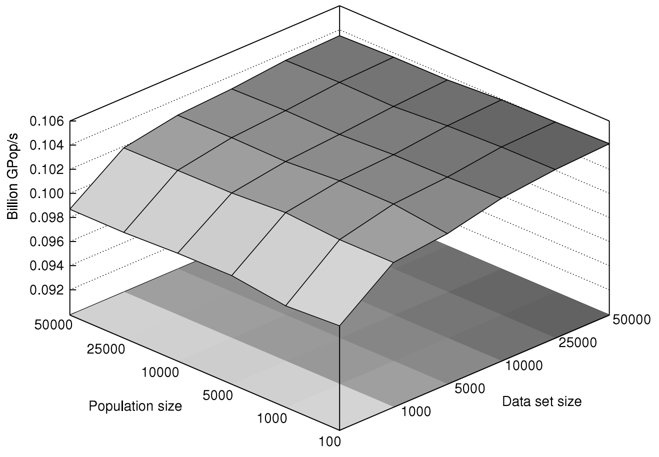


Fig. 5. Regression, CPU, sequential, AMD Phenom II X6 1090T.

Although the 12-core Intel Xeon could evaluate individuals at more than twice the 6-core AMD Phenom's rate, shown in Fig. 6, again their behavior is similar, except for the fact that from 25 000 to 50 000 individuals there is an unexpected slight decrease (of about 7%) in the throughput. Regarding the AMD Phenom's sequential benchmark, the parallel one achieved a speedup of a bit more than 5, which is closer to the ideal speedup of 6 (six cores), meaning that the OpenCL's parallel implementation is highly efficient ($E_6 \approx 85\%$).

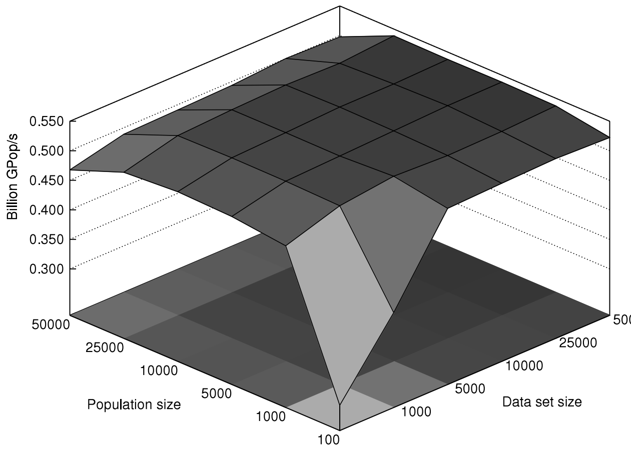


Fig. 6. Regression, CPU, 6-core parallel, AMD Phenom II X6 1090T.

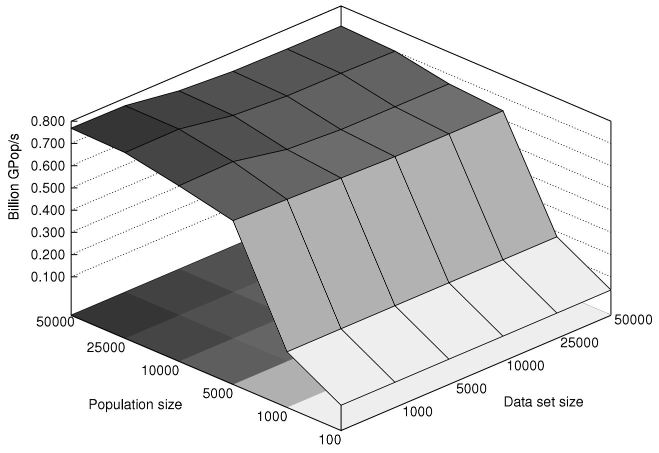


Fig. 8. Regression, GPU, population-parallel per processing element, Nvidia GTX-285.

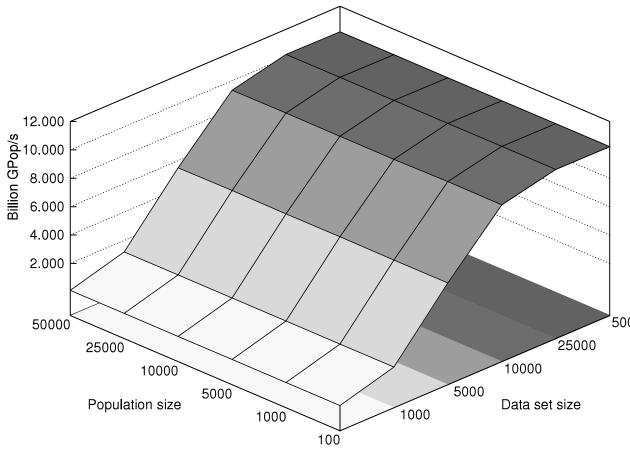


Fig. 7. Regression, GPU, fitness-parallel, Nvidia GTX-285.

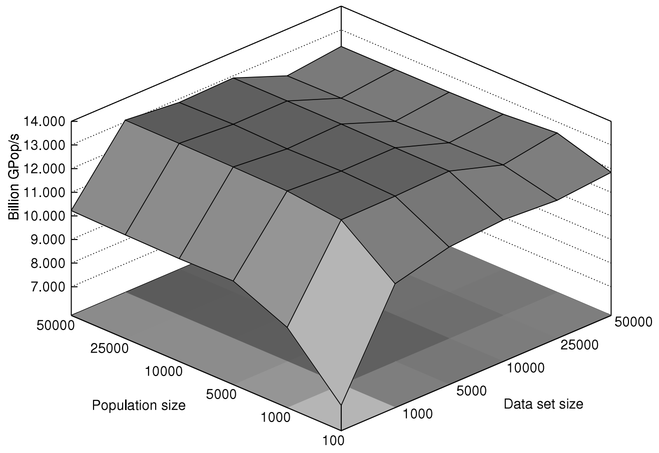


Fig. 9. Regression, GPU, population-parallel per compute unit, Nvidia GTX-285.

The first GPU benchmark is presented in Fig. 7 and demonstrates the power of this architecture over the CPU by reaching more than 10 billion GPop/s at optimum parameter settings. It is clear that for the fitness-parallel OpenCL kernel, the raw performance strictly depends on the size of the dataset: the greater the number of fitness cases to be evaluated, the better the throughput. One important characteristic of this strategy of parallelization is that a large volume of data is required to keep the considerable amount of processing elements busy all the time. Due to instruction pipeline and other optimization techniques found on the GPU [8,1,25], each processing element needs ideally several fitness cases (work-items) at once. As for the population size, since in the fitness-parallel strategy only a single program is evaluated at a time, i.e. the programs are evaluated sequentially, the population size does not affect the throughput at all.

As shown in Fig. 8, evaluating one program per processing element does not seem to be a good strategy of parallelization. The overall raw performance is weak, with its peak throughput corresponding to less than 8% of the fitness-parallel's peak performance.

Unlike the previous benchmark, Fig. 9 shows the impressive throughput achieved by the PPCU OpenCL kernel on the Nvidia GPU. Above 13 billion GPop/s of peak performance, which means a speedup of about 72 when compared with the sequential Intel Xeon (Fig. 3) and 126 times the performance of the sequential AMD Phenom benchmark (Fig. 5). The shape of the surface is similar to those of the parallel CPU benchmarks (Figs. 4 and 6), being rather robust with regard to the parameter variations—the only really bad range of parameters concerns the very small values for population and dataset sizes. There is a noticeable variation of performance (≈5%) on the top of the plot. Since in preliminary benchmarks using an older version of the GPU driver (260.19.21) the throughput for the dataset sizes from 1000 to 10 000 was lower than the current level shown in Fig. 9, it seems that internal driver optimizations have somewhat favored this amount of workload but not others.

The results on the ATI Radeon GPU are depicted in Figs. 10–12. The performance patterns are very similar to those of the Nvidia GPU, but in general the Nvidia GPU is approximately four times faster than the assessed ATI Radeon in this batch of symbolic regression benchmarks.

Table 4 summarizes the performance in terms of billion GPop/s of the studied processors and strategies of parallelization in symbolic regression. For the significance tests with respect to all possible pairwise comparisons, please refer to Appendix A.1.

4.3.2. *Data classification*

To classify means to group things based on similarities, giving them a class label. In data classification, a data sample is described by a set of attributes, so the similarity degree among data samples is taken in some way from their attributes. The goal of the data classification task is to extract useful information from a training dataset so that it is possible to: (i) accurately and automatically classify unseen but related data samples; and (ii) obtain a concise and rich human-readable knowledge about the dataset.

Although classification problems can be seen as a special form of regression problems where the desired output (the dependent

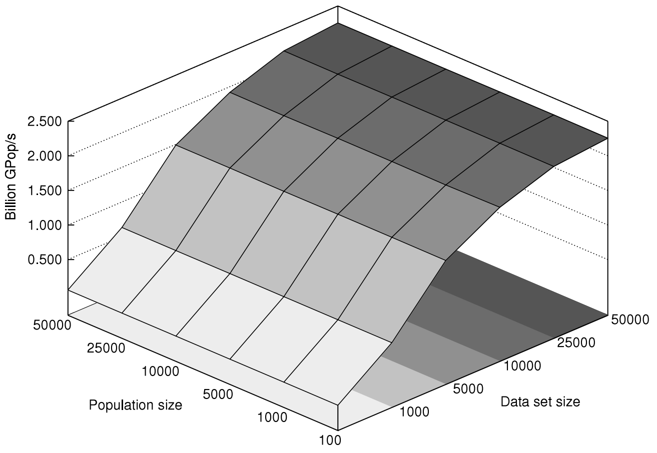


Fig. 10. Regression, GPU, fitness-parallel, ATI Radeon HD5750.

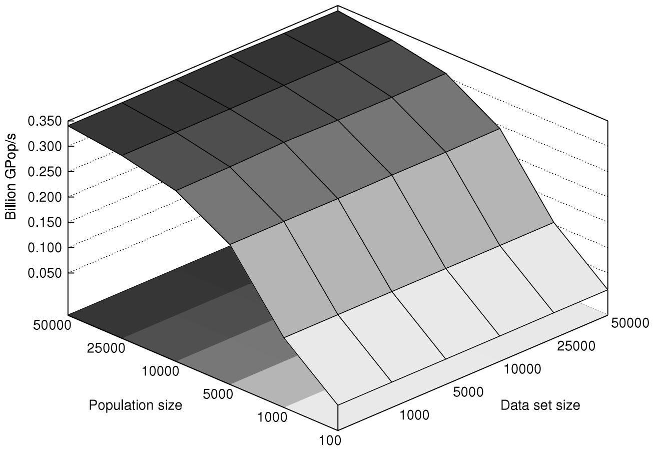


Fig. 11. Regression, GPU, population-parallel per processing element, ATI Radeon HD5750.

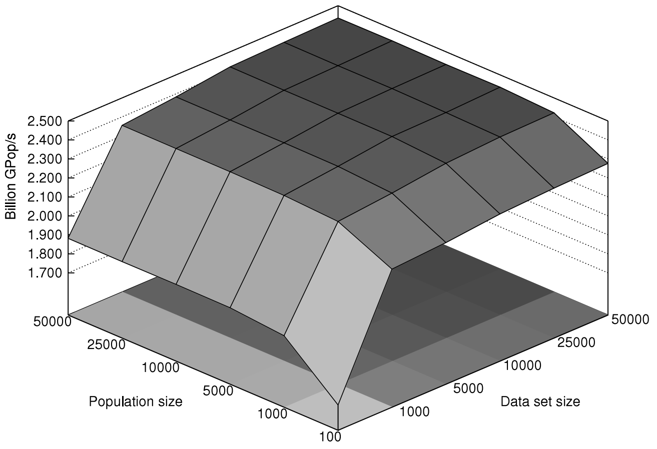


Fig. 12. Regression, GPU, population-parallel per compute unit, ATI Radeon HD5750.

variable) is discrete, this is not a natural way to model them, particularly when no ordering notion among the problem's classes exists. Moreover, it is common sense that classifiers based on rules – i.e. *if-then* conditional structures – in opposition to mathematical expressions, are easier to read and understand by humans. Therefore, the experiments presented in this section will feature those kind of constructions; more specifically, the GP primitive sets used are $\mathbb{F} = \{\text{if-then-else}, =, \langle, \rangle, \wedge, \vee, \neg\}$ and $\mathbb{T} = \{x_1 \cdots x_{10}, 1, -1, \mathbb{R}\}$.

Table 4
Performance summary for symbolic regression experiments (in Billion GPop/s).

Compute device/strategy	Peak	Median
Intel Xeon W3680 CPU/sequential	0.180	0.180
Intel Xeon W3680 CPU/12-core parallel	1.430	1.390
AMD Phenom II X6 1090T CPU/sequential	0.100	0.100
AMD Phenom II X6 1090T CPU/6-core parallel	0.540	0.530
Nvidia GTX-285 GPU/fitness-parallel	10.25	7.425
Nvidia GTX-285 GPU/population-parallel PE	0.770	0.660
Nvidia GTX-285 GPU/population-parallel CU	**13.08**	12.30
ATI Radeon HD5750 GPU/fitness-parallel	2.260	1.700
ATI Radeon HD5750 GPU/population-parallel PE	0.340	0.270
ATI Radeon HD5750 GPU/population-parallel CU	2.440	2.380

The distinction between regression and classification problems when assessing program evaluation throughput is important because conditional primitives (such as the *if-then-else*) causes *branch divergence* on the GPU architecture. As a result, the performance usually decreases by a certain amount, being also influenced by the implementation strategy (Section 3).

The CPU benchmarks in the classification domain, shown in Figs. 13–16, point out an interesting finding. Even though there exist conditional operators in the function set, the overall throughput is significantly higher than that achieved in the symbolic regression benchmarks. This is owed to the fact that (i) conditional structures are managed by the conventional CPU architecture at no significant cost, in other words, there is no *branch divergence*; and (ii) transcendental functions such as *sine* and *cosine* are usually costly, hence they slow down the evaluation of programs that use them, as the case in symbolic regression benchmarks.

Besides the higher performance, the general behavior of these CPU classification benchmarks is quite similar to the one exhibited in the symbolic regression domain. Thus, as long as the population and dataset sizes are sufficiently large, the throughput tends to be optimal.

The pattern presented in Fig. 13, however, does not quite follow the general trend. The throughput on the smallest dataset was slightly better than the remaining ones. Fortunately, the variation is very small, the difference between the minimum and maximum median throughput values is just 3%.[5]

Contrary to the CPU benchmarks, switching from the symbolic regression to the data classification domain has made the overall GPU throughput worse. These results, for both Nvidia and ATI GPUs, are depicted in Figs. 17–22. On average, the Nvidia GPU can sustain approximately 11% more throughput in the symbolic regression domain when compared with the data classification domain, while on the ATI GPU this gap goes down to 8%. The reasons behind this slow down on the GPU architecture are the opposite of the reasons why the CPU architecture turned out to be better in the data classification benchmarks, that is: (i) conditional structures are costly on the GPU, that is, they cause *branch divergence*; and (ii) transcendental functions have native hardware support on the GPUs, with tunable precision, and so can be computed very quickly (Section 5).

Despite the minor slow down, the obtained performance on the GPU architecture in the classification domain is still impressive when compared with the throughput of the CPU processors. The Nvidia GPU's peak performance is 11.8 GPop/s (Fig. 19), a speedup

[5] This behavior may be related to an issue with the adopted OpenCL platform's implementation, provided by the AMD APP SDK 2.5, as a previous version, 2.3, behaves as expected. Version 2.5 was chosen because it is up to date and produces better and stable performance in every other experiment.

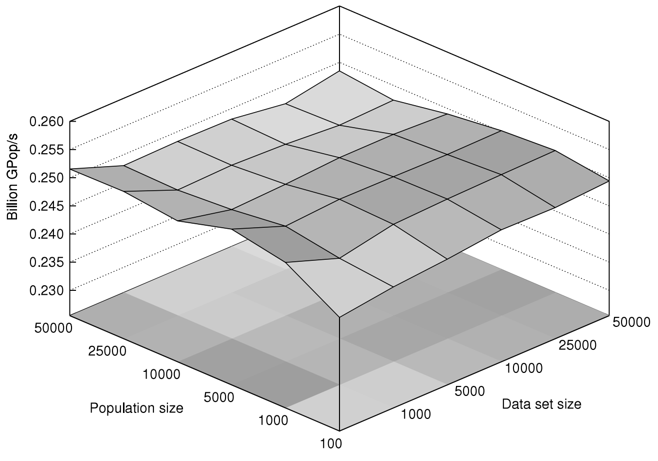


Fig. 13. Classification, CPU, sequential, Intel Xeon W3680.

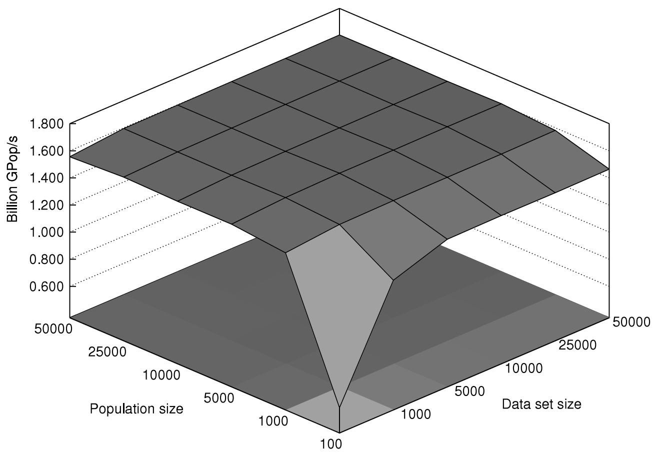


Fig. 14. Classification, CPU, 12-core parallel, Intel Xeon W3680.

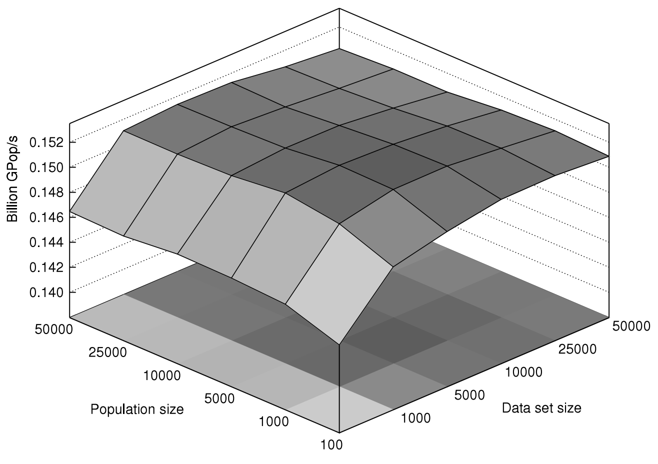


Fig. 15. Classification, CPU, sequential, AMD Phenom II X6 1090T.

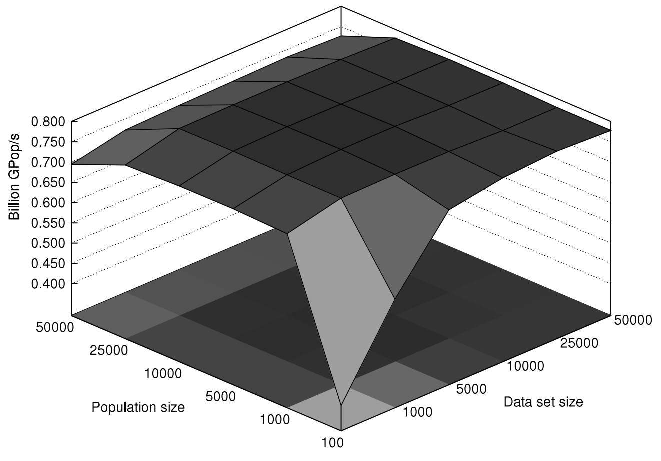


Fig. 16. Classification, CPU, 6-core parallel, AMD Phenom II X6 1090T.

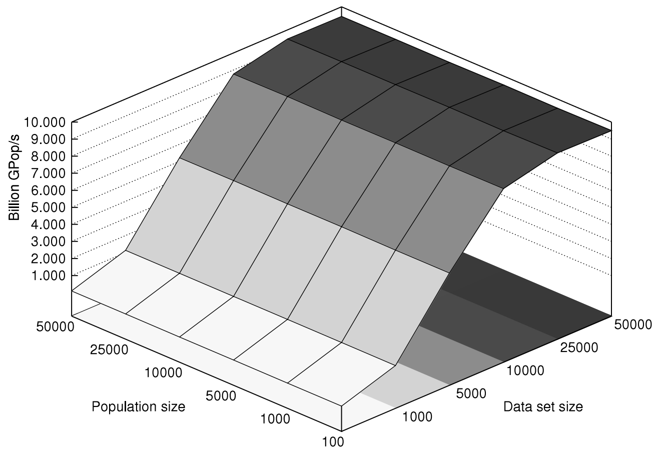


Fig. 17. Classification, GPU, fitness-parallel, Nvidia GTX-285.

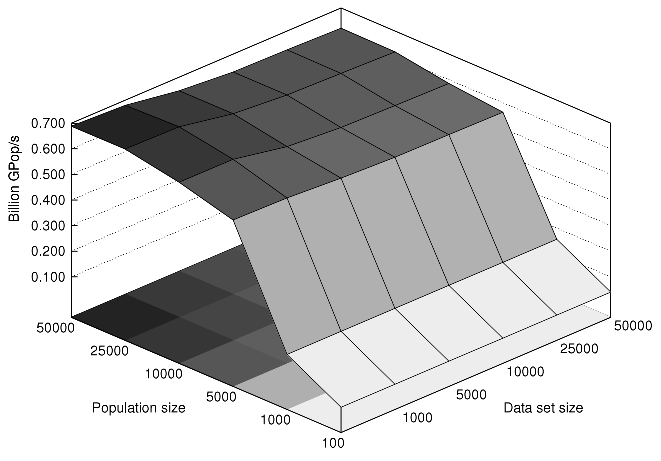


Fig. 18. Classification, GPU, population-parallel per processing element, Nvidia GTX-285.

of 47 and 78 over the sequential performance on the Intel Xeon (Fig. 13) and AMD Phenom (Fig. 15) CPUs, respectively.

Concerning how the raw performance reacts in face of parameter variations, the behavior shown on the GPU data classification benchmarks is similar to the one exhibited on the class of symbolic regression problems.

A summary of the peak and mean performance on the evaluated architectures can be seen in Table 5. The corresponding statistical tests can be found in Appendix A.2.

4.4. Discussion

The benchmarks have shown that the GPU devices hugely outperform the CPU processors on the evaluation of genetic programming trees by means of an interpreter. For instance, in the symbolic regression domain the fastest processor configuration (12-core Intel Xeon W3680 running in parallel) was on average nine times slower than the fastest GPU configuration (Nvidia GTX-285 running the population-parallel CU kernel). Even the

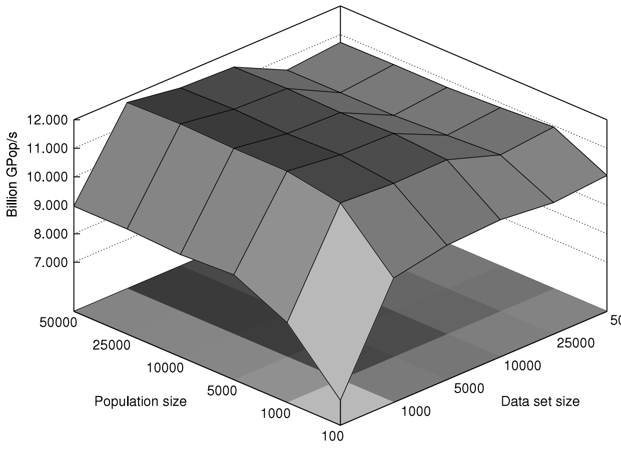


Fig. 19. Classification, GPU, population-parallel per compute unit, Nvidia GTX-285.

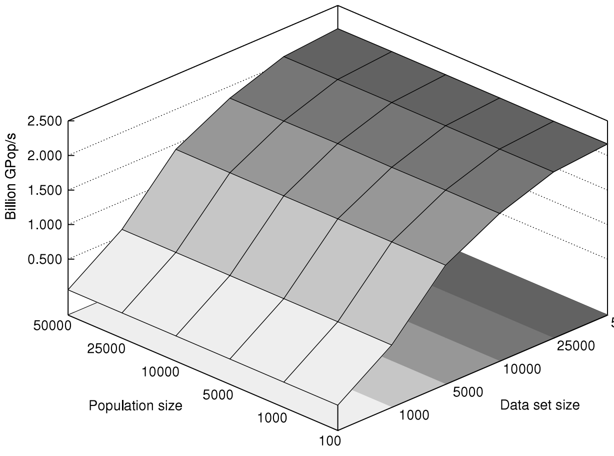


Fig. 20. Classification, GPU, fitness-parallel, ATI Radeon HD5750.

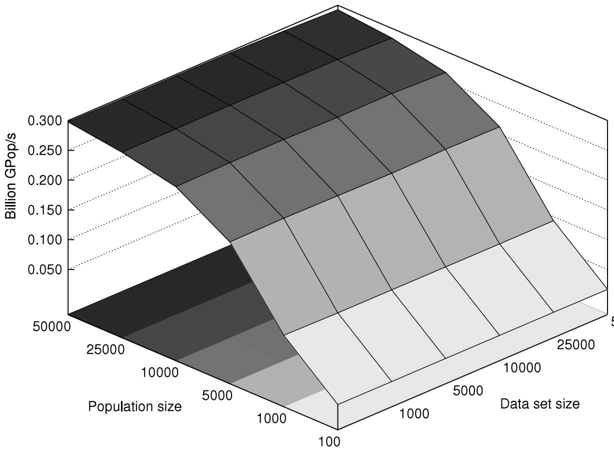


Fig. 21. Classification, GPU, population-parallel per processing element, ATI Radeon HD5750.

inexpensive mid-range ATI Radeon HD5750 was able to deliver almost two times the performance of the top-of-the-range 12-core Intel Xeon W3680.

Following the findings in the related literature, the experiments have confirmed the decisive role played by the underlying strategy of parallelization. Interpreting a different program per processing element, i.e. the *population-parallel per processing element* strategy (PPPE), clearly leads to the worst GPU performance under any parameter configuration and problem domain. Since each program in the population is usually unique, this approach forces each

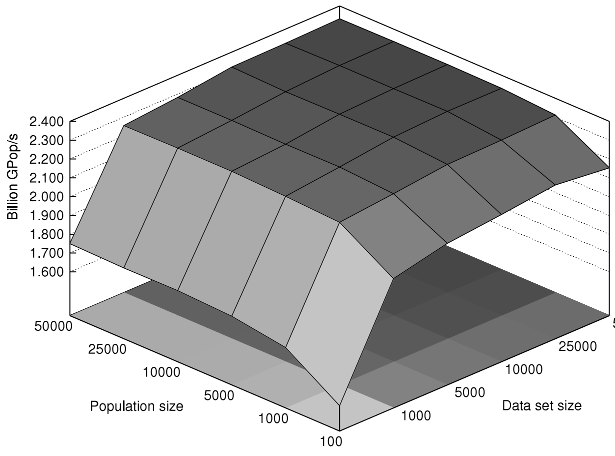


Fig. 22. Classification, GPU, population-parallel per compute unit, ATI Radeon HD5750.

Table 5
Performance summary for data classification experiments (in Billion GPop/s).

Compute device/strategy	Peak	Median
Intel Xeon W3680 CPU/sequential	0.250	0.250
Intel Xeon W3680 CPU/12-core parallel	1.610	1.590
AMD Phenom II X6 1090T CPU/sequential	0.150	0.150
AMD Phenom II X6 1090T CPU/6-core parallel	0.790	0.780
Nvidia GTX-285 GPU/fitness-parallel	9.540	6.990
Nvidia GTX-285 GPU/population-parallel PE	0.690	0.590
Nvidia GTX-285 GPU/population-parallel CU	**11.85**	10.75
ATI Radeon HD5750 GPU/fitness-parallel	2.170	1.625
ATI Radeon HD5750 GPU/population-parallel PE	0.300	0.235
ATI Radeon HD5750 GPU/population-parallel CU	2.330	2.280

processing element to execute a different instruction during the process of evaluation where the entire population is interpreted simultaneously. As the current GPU architecture cannot process in parallel different instructions within the same compute unit, the execution of diverging instructions are serialized, forcing the remaining processing elements to wait and thus degrading the overall throughput.

The simplest and more natural approach on the GPU architecture, which is the evaluation in parallel of the fitness cases (*fitness-parallel* strategy, FP), results in high throughput when there are enough data to keep all the processing elements busy. The downside of the data parallel strategy is that to achieve optimum performance, several fitness cases are required per processing element. Unfortunately, in many real-world problems the amount of available data is not large, leading to a low level of processor occupancy and poor performance.

Under the proposed sets of benchmarks, the clear winner among the studied strategies of parallelization is the *population-parallel per compute unit* approach. This strategy takes advantage of the fact that modern GPU architectures are capable of executing simultaneously a different instruction in each compute unit, and then assigns the evaluation of a different program per compute unit. The PPCU strategy lowers the data requirement of the FP approach while avoids the divergence issues related to the PPPE strategy. As a result, the PPCU strategy is able to achieve very high performance in a wide range of parameter configurations.

4.4.1. *Performance per Watt*

Without a doubt, power consumption has increasingly become a major concern for high-performance computing, due not only to the associated electricity costs, but also to environmental factors [4]. Based on the presented throughput results, we aim at

Table 6
General performance per Watt (in million GPop/s).

Compute device/strategy	Problem domain	
	Regression	Classification
Intel Xeon W3680 CPU/1 core	1.385	1.923
Intel Xeon W3680 CPU/12 cores	10.69	12.23
AMD Phenom II X6 1090T CPU/1 core	0.800	1.200
AMD Phenom II X6 1090T CPU/6 cores	4.240	6.240
Nvidia GTX-285 GPU/FP	36.40	34.26
Nvidia GTX-285 GPU/PPPE	3.235	2.892
Nvidia GTX-285 GPU/PPCU	**60.29**	**52.70**
ATI Radeon HD5750 GPU/FP	19.78	18.89
ATI Radeon HD5750 GPU/PPPE	3.139	2.732
ATI Radeon HD5750 GPU/PPCU	27.67	26.51

measuring the power efficiency of each parallelization strategy on the studied architectures. To simplify our estimation, we assume that the devices work at their full occupancy, that is, at maximum power consumption as listed in Table 1.

Table 6 shows the general performance per Watt for each configuration. The estimates were calculated with regard to the *median* throughput, found in Tables 4 and 5.

As expected, the most power efficient configuration is the population-parallel per CU strategy together with the Nvidia GTX-285 GPU device, being on average 5 times more efficient than the Intel CPU (twelve cores) and 11 times more efficient than the AMD CPU (six cores). On the other hand, excluding the sequential versions, the worst performing strategy was the population-parallel per PE, on both ATI and Nvidia GPU devices.

5. Optimization techniques

This section discusses the main optimization techniques employed in GPOCL, and also addresses the optimal choice of parameter settings. Most of the topics are specific to the GPUs, as this architecture offers a greater extent of optimization opportunities while being more sensitive to certain practices.

5.1. Efficient use of device memory

As mentioned in Section 2.2, OpenCL defines different types of memory space for different purposes.

5.1.1. Global memory

The main GPU memory is the global memory, which has a high storage capacity, but when compared with the other device's memory spaces, it has limited bandwidth and very high latency [1,25]. As discussed earlier, the GPU can take advantage of the coalesced memory access pattern, due to the fact that, by doing so, the memory controller is able to fetch more data in a single memory transaction. In order to explore the coalesced pattern while accessing the training data within the interpreter loop (Section 3.2.1), the original dataset of input variables X are stored in a transposed order as follows: each column representing an input variable is transformed into a row and stored sequentially in memory just after the last linearized column from the previous input variable. This linear layout of input data, let us call it X^L, allows achieving efficient contiguous access. However, the corresponding interpreter's code accessing the training data, that is

$X_n[\texttt{VALUE}(program[op])]$,

must be changed to

$X^L[\texttt{VALUE}(program[op]) \times dataset_{size} + n]$,

in order to be aware of the optimized coalesced access.

5.1.2. Constant memory

The OpenCL constant memory is a small cache-friendly storage capacity memory that can deliver much improved efficiency on many conditions. On the current GPU architecture, high-efficient access is achieved mainly when the GPU is able to detect a uniform access, such as many work-items reading the same address at a time [1,25]. In GPOCL, when the array Y of expect values for the fitness cases fits into the constant memory, it is allocated there. On top of that, as with the global memory, all the implemented OpenCL kernels optimize the access in Y towards a coalesced pattern.

5.1.3. Local memory

Optimizations by means of local memory is very common and can greatly improve the GPU performance [1,25,9,24]. This is due to the fact it is (i) a very low-latency and high-throughput memory; (ii) insensitive to irregular access patterns; and (iii) reusable across work-items because of its shared nature. Additionally, data transfer from global to local memory (and vice versa) is normally done collaboratively, thus the overhead is minimum.[6]

GPOCL takes advantage of the local memory by (i) caching a genetic program prior to its evaluation; and (ii) performing the parallel reduction procedure (Fig. 2) on the partial results stored in this memory scope. Since the use of local memory may not be advantageous on the CPU [5], only the GPU kernels explore this optimization. Moreover, both the caching and reduction optimization can be effectively used only in the FP and PPCU parallel strategies. In the PPPE strategy there is no program data that could be transferred in parallel and reused within the work-group, and there is no partial result to accumulate.

5.2. Optimum parameters for the GPU

On top of what we have seen about parameter configuration in Section 4 concerning optimal population and dataset sizes, some other parameter settings may also harm the performance, specially on the GPU. The precise parameter and range of values that might affect the performance depend on the actual device's architecture and the parallel strategy. The following is a discussion about the interaction between strategies and parameters with respect to performance variation.

Program tree size. Large programs will require a proportionally large stack[7] in order to be fully interpreted (Section 3.2.1), which drains device's resources, such as registers, and hence might degrade the overall performance [1,25]. For kernels using local memory, namely FP and PPCU, when the program size is not multiple of the local size, some work-items from the group will sit idle while the remaining ones finish the memory transfer—that is not a performance penalty itself, but more precisely an underutilization of device's resources.

Dataset size. According to the OpenCL specification, the global size must be evenly divisible by the local size. For the PPCU kernel, because the whole dataset is processed within a work-group, the perfect balance is achieved when the dataset size is multiple of the local size. Otherwise, the local size is rounded up (see Algorithm 5) and a conditional test will be necessary to guarantee that no out-of-range access will occur. Moreover, this means that some work-items will aways sit idle. An analogous problem may arise for the

[6] OpenCL even provides means to perform those copies asynchronously [31].

[7] The required stack size is given by $\max\left(1, \tau - \lfloor\frac{\tau}{\min(\alpha,\tau)}\rfloor\right)$, where τ is the maximum tree size and $\alpha > 0$ is the maximum number of arguments taken by any function in the function set, that is, the maximum arity.

FP kernel, but to a much less extent. Another potential harm related to the dataset size takes place when it is too large, leading to the situation in which the expected output variable (array Y) does not fit in the constant memory; in this case the allocation falls back to the ordinary global memory.

Local size. Due to the GPU architecture's internal organization, the local size should be, whenever possible, (i) a power-of-two and (ii) large enough to maximize the device's occupancy and hide latencies [1,25]. Being power-of-two is also optimum for the parallel reduction, since in each step the number of active work-items is reduced by half.

Population size. The population size affects the PPPE and PPCU kernels, potentially causing load imbalance. Since the PPPE kernel evaluates one program per processing element, the population size should ideally be a multiple of the number of the device's processing elements. Similarly, for the PPCU kernel, setting the population size as a multiple of the number of compute units leads to a balanced workload, because at any given time the device will be fully occupied.[8]

5.3. Acceleration by means of native functions

There are three classes of mathematical built-in functions available in the OpenCL specification [31]: the standard *full* precision, *half* precision, and *native*. The first class represents the default functions, and they comply with the IEEE-754 standard. The functions from the second class halve the precision, are faster, but they still must guarantee half the precision. The native functions, on the other hand, have implementation-defined precision, may violate the IEEE-754 standard, but are usually the fastest ones—because they commonly have direct hardware support. Since GP is highly robust with respect to program evaluation, the use of native functions, as employed by GPOCL, seems to be a good source of optimization.

5.4. Just-in-time kernel optimization

An interesting fact about just-in-time compilation is that it is possible, based on runtime information, to optimize the kernel properly. GPOCL exploits this opportunity by removing (i) unnecessary interpreter's *switch cases*, as the exact function set is known; and (ii) all extra conditionals/operations dedicated to handle non-optimal parameters when optimal ones are detected.

6. Conclusions

We have presented a detailed high-performance GP implementation in OpenCL for accelerated tree evaluation on the CPU and GPU architectures. Three GPU parallelization strategies were assessed in two domains: symbolic regression and data classification. Based on the presented study we can conclude that:

- The GPU is considerably faster and more power efficient than the CPU; even a mid-range GPU can beat a very high-end CPU.
- The *population-parallel per compute unit* strategy is clearly the most efficient GPU parallel model.
- In order to maximize throughput it is better to use larger populations and datasets.
- Branch divergence decreases the GPU performance, although not seriously.
- OpenCL is a compelling replacement for existing non-portable, closed or vendor-controlled GPU programming platforms.

Due to the impressive computational power of the current generation of graphics processing units and the good fitting of genetic programming algorithms to this kind of parallel architecture, we have reached the stage where the GPU is probably going to take over the CPU as the main hardware for genetic programming practice, both for application and development. Despite that, it is unlikely that the CPU will loose its status of an important platform for GP, especially in the multi-core era. Additionally, many other existing and/or upcoming parallel architectures—such as the IBM's Cell Broadband Engine [15], Sun's Niagara [18], Tilera's Tile [32], Intel's Many Integrated Core (MIC) [29], and AMD's Fusion [6] processors—will compete in the promising segment lying between the latency- and the throughput-oriented architectures. To harness the power of those forthcoming multi-vendor and heterogeneous high-performance computing processors in a uniform manner, one will need to resort to programming platforms designed to be portable and device independent; today, the Open Computing Language (OpenCL) seems to be a promising candidate.

Finally, given the fact that the GPU greatly outperforms the CPU, it seems logical to leave the brute-force work of program evaluation to the former and assign to the latter tasks that currently are hard to be carried out by the GPU. A promising topic for future work involves a kind of hybrid CPU and GPU parallelism via OpenCL, in which the GPU would be dedicated to evaluate programs and the CPU would be responsible for performing in parallel the remaining evolutionary algorithm, that is, the breeding phase.

Acknowledgments

The authors would like to thank the reviewers for their valuable suggestions, and the support provided by FAPERJ (grants E-26/102.825/2008 and E-26/102.025/2009), and CNPq (grant 311651/2006-2).

Appendix A. Statistical analyses

In order to test whether the parallel strategies' performances are statistically different or not, and also to provide an indication about how different they are, the Friedman's statistical test, as described in [7], was used in the following analyses. This test is non-parametric, thus no assumption about populations' distribution (e.g. normality) is required, it does not assume equality of variance, and is well-suited for multiple comparisons.

The analyses were done separately for symbolic regression and data classification domains. We aim at testing all pairwise comparisons among every possible combination of *strategy–device*, hereafter referred to as *instance*, based on their achieved median throughput on each problem configuration. A strategy can be the *sequential* and *parallel CPU*, *GPU FP*, *PPPE*, or *PPCU*, whereas the devices are the *Intel* and *AMD* CPUs, or the *Nvidia* and *ATI* GPUs—thus, there is a total of 10 instances. A problem configuration is a particular combination of *population* and *dataset* sizes.

The Friedman's test has two phases. Firstly, the null hypothesis, which states that there is no difference among the instances, is checked. If it is rejected, a post hoc analysis is performed to find out those pairwise comparisons leading to significant difference—to this end, Shaffer's static procedure is used, which is both robust and computationally inexpensive [7]. The significance level adopted was $\alpha = 0.05$, meaning 95% of confidence.

[8] There is no guarantee, however, since the devices are free to schedule the workload at their will.

Table A.7
Symbolic regression: average rankings of the instances.

Instance	Ranking
Intel CPU$_1$	8.0
Intel CPU$_{12}$	4.4
AMD CPU$_1$	9.5
AMD CPU$_6$	6.3
Nvidia FP	3.2
Nvidia PPPE	6.4
Nvidia PPCU	1.0
ATI FP	5.2
ATI PPPE	8.2
ATI PPCU	2.7

Table A.8
Symbolic regression: significantly different pairwise comparisons (typed in bold), with $\alpha = 0.05$.

i	Hypothesis	Adjusted p-value (Shaffer)
1	**AMD CPU$_1$ vs. Nvidia PPCU**	**0.000000**
2	**Nvidia PPCU vs. ATI PPPE**	**0.000000**
3	**Intel CPU$_1$ vs. Nvidia PPCU**	**0.000000**
4	**AMD CPU$_1$ vs. ATI PPCU**	**0.000000**
5	**AMD CPU$_1$ vs. Nvidia FP**	**0.000000**
6	**ATI PPPE vs. ATI PPCU**	**0.000000**
7	**Nvidia PPPE vs. Nvidia PPCU**	**0.000000**
8	**Intel CPU$_1$ vs. ATI PPCU**	**0.000000**
9	**AMD CPU$_6$ vs.Nvidia PPCU**	**0.000000**
10	**Intel CPU$_{12}$ vs. AMD CPU$_1$**	**0.000000**
11	**Nvidia FP vs. ATI PPPE**	**0.000000**
12	**Intel CPU$_1$ vs. Nvidia FP**	**0.000000**
13	**AMD CPU$_1$ vs. ATI FP**	**0.000000**
14	**Nvidia PPCU vs. ATI FP**	**0.000000**
15	**Intel CPU$_{12}$ vs. ATI PPPE**	**0.000001**
16	**Nvidia PPPE vs. ATI PPCU**	**0.000005**
17	**Intel CPU$_1$ vs. Intel CPU$_{12}$**	**0.000006**
18	**AMD CPU$_6$ vs. ATI PPCU**	**0.000007**
19	**Intel CPU$_{12}$ vs. Nvidia PPCU**	**0.000059**
20	**Nvidia FP vs. Nvidia PPPE**	**0.000218**
21	**AMD CPU$_1$ vs. AMD CPU$_6$**	**0.000218**
22	**AMD CPU$_6$vs. Nvidia FP**	**0.000312**
23	**AMD CPU$_1$ vs. Nvidia PPPE**	**0.000312**
24	**ATI FP vs. ATI PPPE**	**0.000485**
25	**Intel CPU$_1$ vs. ATI FP**	**0.001507**
26	**ATI FP vs. ATI PPCU**	**0.006843**
27	**Nvidia FP vs. Nvidia PPCU**	**0.033222**
28	Intel CPU$_{12}$ vs. Nvidia PPPE	0.080817
29	Nvidia FP vs. ATI FP	0.086178
30	Intel CPU$_{12}$ vs. AMD CPU$_6$	0.091448
31	AMD CPU$_6$ vs. ATI PPPE	0.108527
32	Nvidia PPPE vs. ATI PPPE	0.127512
33	IntelCPU$_1$vs. AMDCPU$_6$	0.205488
34	Intel CPU$_{12}$ vs. ATI PPCU	0.210917
35	Nvidia PPCU vs. ATI PPCU	0.214692
36	Intel CPU$_1$vs. Nvidia PPPE	0.214692
37	Intel CPU$_1$ vs. AMDCPU$_1$	0.386641
38	AMD CPU$_1$ vs. ATI PPPE	0.638709
39	Nvidia PPPE vs. ATI FP	0.714566
40	Intel CPU$_{12}$ vs. Nvidia FP	0.714566
41	AMD CPU$_6$ vs. ATI FP	0.714566
42	Intel CPU$_{12}$ vs. ATI FP	0.910231
43	Nvidia FP vs. ATI PPCU	1.308824
44	Intel CPU$_1$ vs. ATI PPPE	1.570514
45	AMD CPU$_6$ vs. Nvidia PPPE	1.570514

Two tables are shown. The first one is the average ranking of the instances, which gives an idea about the underlying ranking distribution, used by the Friedman's test to calculate its statistics. The more the rankings deviate from the expected value (with 10 instances the expected ranking is $(10+1)/2 = 5.5$), the higher the probability of having differences among the instances. The second table, based on the post hoc analysis, lists the actual significantly different pairwise comparisons, from all the possible ones.

Table A.9
Data classification: average rankings of the instances.

Instance	Ranking
Intel CPU$_1$	7.8
Intel CPU$_{12}$	4.2
AMD CPU$_1$	9.2
AMD CPU$_6$	5.5
Nvidia FP	3.4
Nvidia PPPE	7.1
Nvidia PPCU	1.0
ATI FP	5.5
ATI PPPE	8.6
ATI PPCU	2.7

Table A.10
Data classification: significantly different pairwise comparisons (typed in bold), with $\alpha = 0.05$.

i	Hypothesis	Adjusted p-value (Shaffer)
1	**AMD CPU$_1$vs. Nvidia PPCU**	**0.000000**
2	**Nvidia PPCU vs. ATI PPPE**	**0.000000**
3	**IntelCPU$_1$vs. Nvidia PPCU**	**0.000000**
4	**AMDCPU$_1$vs. ATI PPCU**	**0.000000**
5	**Nvidia PPPE vs. Nvidia PPCU**	**0.000000**
6	**ATI PPPE vs. ATI PPCU**	**0.000000**
7	**AMDCPU$_1$vs. Nvidia FP**	**0.000000**
8	**Nvidia FP vs. ATI PPPE**	**0.000000**
9	**IntelCPU$_1$vs. ATI PPCU**	**0.000000**
10	**IntelCPU$_{12}$vs. AMDCPU$_1$**	**0.000000**
11	**AMDCPU$_6$vs. Nvidia PPCU**	**0.000000**
12	**Nvidia PPCU vs. ATI FP**	**0.000000**
13	**Nvidia PPPE vs. ATI PPCU**	**0.000000**
14	**IntelCPU$_1$vs. Nvidia FP**	**0.000000**
15	**IntelCPU$_{12}$vs. ATI PPPE**	**0.000000**
16	**Nvidia FP vs. Nvidia PPPE**	**0.000005**
17	**AMDCPU$_1$vs. ATI FP**	**0.000006**
18	**IntelCPU$_1$vs. IntelCPU$_{12}$**	**0.000007**
19	**AMDCPU$_1$vs.AMDCPU$_6$**	**0.000008**
20	**IntelCPU$_{12}$vs. Nvidia PPCU**	**0.000218**
21	**ATI FP vs. ATI PPPE**	**0.000312**
22	**AMDCPU$_6$$vs.$ATI PPPE**	**0.000445**
23	**IntelCPU$_{12}$vs. Nvidia PPPE**	**0.000685**
24	**AMDCPU$_6$vs. ATI PPCU**	**0.001578**
25	**ATI FP vs. ATI PPCU**	**0.002083**
26	**Nvidia FP vs. Nvidia PPCU**	**0.014159**
27	**IntelCPU$_1$vs. ATI FP**	**0.016876**
28	**IntelCPU$_1$vs. AMDCPU$_6$**	**0.022222**
29	AMDCPU$_6$vs. Nvidia FP	0.052588
30	Nvidia FP vs. ATI FP	0.063538
31	AMDCPU$_1$vs. Nvidia PPPE	0.067348
32	Nvidia PPCU vs. ATI PPCU	0.246070
33	Nvidia PPPE vs. ATI FP	0.253727
34	AMDCPU$_6$vs. Nvidia PPPE	0.287612
35	IntelCPU$_{12}$vs. ATI PPCU	0.430215
36	Nvidia PPPE vs. ATI PPPE	0.430215
37	IntelCPU$_{12}$vs. AMDCPU$_6$	0.508309
38	IntelCPU$_1$vs. AMDCPU$_1$	0.508309
39	IntelCPU$_{12}$vs. ATI FP	0.508309
40	IntelCPU$_{12}$vs. Nvidia FP	1.759624
41	IntelCPU$_1$vs. ATI PPPE	1.759624
42	Nvidia FP vs. ATI PPCU	1.759624
43	IntelCPU$_1$vs. Nvidia PPPE	1.759624
44	AMDCPU$_1$vs. ATI PPPE	1.759624
45	AMDCPU$_6$vs. ATI FP	1.759624

A.1. Symbolic regression

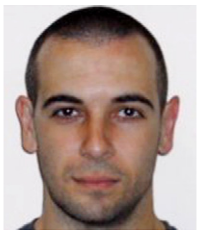
As for Table A.7, Friedman's Chi-Square (9 degrees of freedom) is 260.84, which leads to a p-value < 0.001, that is, the null hypothesis is rejected. The post hoc test (Table A.8) reveals that the pairwise comparisons from 1 to 27 are significantly different, while that cannot be claimed for the remaining ones.

A.2. Data classification

For the data classification domain, regarding Table A.9, Friedman's Chi-Square (9 degrees of freedom) is 256.54 and p—value < 0.001; again, the null hypothesis is rejected. The post hoc test (Table A.10) lists the pairwise comparisons 1–28 as significantly different.

References

[1] Advanced Micro Devices, AMD accelerated parallel processing programming guide-OpenCL, 12 2011.
[2] D. Andre, J.R. Koza, A parallel implementation of genetic programming that achieves super-linear performance, Inform. Sci. 106 (3–4) (1998) 201–218.
[3] D.A. Augusto, H.J.C. Barbosa, Symbolic regression via genetic programming, in: Proceedings of the VI Brazilian Symposium on Neural Networks, IEEE Computer Society, Los Alamitos, CA, USA, 2000, pp. 173–178.
[4] W. Chun Feng, X. Feng, R. Ge, Green supercomputing comes of age, IT Prof. 10 (2008) 17–23. http://doi.ieeecomputersociety.org/10.1109/MITP.2008.8.
[5] I. Corporation, Writing optimal OpenCL code with intel OpenCL SDK, 12 2011.
[6] Advanced Micro Devices, Coming soon: the AMD fusion family of APUs, 2010. URL http://sites.amd.com/us/fusion/APU/Pages/fusion.aspx.
[7] S. García, F. Herrera, An extension on ''statistical comparisons of classifiers over multiple data sets'' for all pairwise comparisons, J. Mach. Learn. Res. 9 (2009) 2677–2694.
[8] M. Garland, D.B. Kirk, Understanding throughput-oriented architectures, Commun. ACM 53 (2010) 58–66.
[9] B. Gaster, D. Kaeli, L. Howes, P. Mistry, Heterogeneous Computing with OpenCL, Morgan Kaufmann Pub., 2011.
[10] A. Grama, G. Karypis, V. Kumar, A. Gupta, Introduction to Parallel Computing (2nd Edition), second ed., Addison Wesley, 2003.
[11] S. Harding, W. Banzhaf, Fast genetic programming and artificial developmental systems on GPUs, in: Proceedings of the 21st International Symposium on High Performance Computing Systems and Applications, IEEE Computer Society, 2007, p. 2.
[12] S. Harding, W. Banzhaf, Implementing cartesian genetic programming classifiers on graphics processing units using GPU.net, in: Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation, GECCO'11, ACM, New York, NY, USA, 2011, pp. 463–470.
[13] W.D. Hillis, G.L. Steele Jr., Data parallel algorithms, Commun. ACM 29 (1986) 1170–1183.
[14] R. Hiremane, From Moore's Law to Intel Innovation - Prediction to Reality. Intel Magazine 1–9 (April 2005).
[15] C.R. Johns, D.A. Brokenshire, Introduction to the cell broadband engine architecture, IBM J. Res. Dev. 51 (2007) 503–519.
[16] H. Juille, J.B. Pollack, Massively parallel genetic programming, in: P.J. Angeline, K.E. Kinnear (Eds.), Advances in Genetic Programming 2, MIT Press, Cambridge, MA, USA, 1996, pp. 339–358 Chapter 17.
[17] M.J. Keith, M.C. Martin, Genetic programming in C++: Implementation issues, 1994.
[18] P. Kongetira, K. Aingaran, K. Olukotun, Niagara: a 32-way multithreaded sparc processor, IEEE Micro 25 (2005) 21–29.
[19] J.R. Koza, Genetic Programming: On the Programming of Computers by Natural Selection, MIT Press, Cambridge, Mass, 1992.
[20] W. Langdon, W. Banzhaf, A SIMD interpreter for genetic programming on GPU graphics cards, in: Genetic Programming, 2008, pp. 73–85.
[21] O. Maitre, P. Collet, N. Lachiche, Fast evaluation of GP trees on GPGPU by optimizing hardware scheduling, in: A.I. Esparcia-Alcazar, A. Ekart, S. Silva, S. Dignum, A.S. Uyar (Eds.), Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010, in: LNCS, vol. 6021, Springer, Istanbul, 2010, pp. 301–312.
[22] D.T. Marr, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, J.A. Miller, M. Upton, Hyper-threading technology architecture and microarchitecture, Intel Technology J. 6 (1) (2002) 1–12.
[23] E. Mollick, Establishing moore's law, IEEE Ann. Hist. Comput. 28 (2006) 62–75.
[24] A. Munshi, B.R. Gaster, T.G. Mattson, J. Fung, D. Ginsburg, OpenCL Programming Guide, Addison Wesley Professional, 2011.
[25] NVIDIA Corporation, OpenCL best practices guide, 2010.
[26] D. Robilliard, V. Marion, C. Fonlupt, High performance genetic programming on GPU, in: Proceedings of the 2009 Workshop on Bio-Inspired Algorithms for Distributed Systems, BADS'09, ACM, New York, NY, USA, 2009, pp. 85–94.
[27] D. Robilliard, V. Marion-Poty, C. Fonlupt, Population parallel GP on the g80 GPU, in: Genetic Programming, 2008, pp. 98–109.
[28] D. Robilliard, V. Marion-Poty, C. Fonlupt, Genetic programming on graphics processing units, Genet. Program. Evol. Mach. 10 (4) (2009) 447–471. (Special issue on parallel and distributed evolutionary algorithms, part I).
[29] K. Skaugen, Petascale to exascale: extending intel's hpc commitment, in: Proceedings of the International Supercomputing Conference, ISC'10, Hamburg, Germany, 2010.
[30] E.-G. Talbi, Metaheuristics: From Design to Implementation, Wiley Publishing, 2009.
[31] Khronos OpenCL Working Group, The OpenCL specification, Version 1.1, 30 September 2010. URL http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.
[32] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J.F. Brown III, A. Agarwal, On-chip interconnection architecture of the tile processor, IEEE Micro 27 (2007) 15–31.
[33] Q. Yu, C. Chen, Z. Pan, Parallel genetic algorithms on programmable graphics hardware, Adv. Nat. Comput. (2005) 1051–1059.



Douglas A. Augusto is a post-doctoral researcher at the Laboratório Nacional de Computação Científica (LNCC), Brazil. He received his M.Sc. and D.Sc. degrees from the Federal University of Rio de Janeiro, Brazil, in 2004 and 2009, respectively. His research interest includes parallel and distributed computing, general purpose GPU programming, and metaheuristics.

Helio J.C. Barbosa is a Senior Technologist at the Laboratório Nacional de Computação Científica, Brazil. He received a Civil Engineering degree (1974) from the Federal University of Juiz de Fora, where he is an Associate Professor in the Computer Science Department, and M.Sc. (1978) and D.Sc. (1986) degrees in Civil Engineering from the Federal University of Rio de Janeiro, Brazil. During 1988–1990 he was a visiting scholar at the Division of Applied Mechanics, Stanford University, USA. His is currently mainly interested in the design and application of nature-inspired metaheuristics in engineering and biology.