**Noname manuscript No.**
(will be inserted by the editor)

# Speeding up the evaluation phase of GP classification algorithms on GPUs

**Alberto Cano · Amelia Zafra · Sebastián Ventura**

Received: date / Accepted: date

**Abstract** The efficiency of evolutionary algorithms has become a studied problem since it is one of the major weaknesses in these algorithms. Specifically, when these algorithms are employed for the classification task, the computational time required by them grows excessively as the problem complexity increases. This paper proposes an efficient scalable and massively parallel evaluation model using the NVIDIA CUDA GPU programming model to speedup the fitness calculation phase and greatly reduce the computational time. Experimental results show that our model significantly reduces the computational time compared to the sequential approach, reaching a speedup of up to 820X. Moreover, the model is able to scale to multiple GPU devices and can be easily extended to any evolutionary algorithm.

**Keywords** Evolutionary Algorithms · Genetic Programming · Classification · Parallel Computing · GPU.

## 1 Introduction

Evolutionary algorithms (EAs) are search methods inspired by natural evolution to find a reasonable solution for data mining and knowledge discovery [Fre02]. Genetic programming (GP) is a specialization of EAs, where each individual represents a computer program. It is a machine learning technique used to optimize a population of computer programs according to a fitness function that determines the program's ability to perform a task. Recently, GP has been applied to different common data mining tasks such as classification [EVH10], feature selection [LKB06], and clustering [DDFT04]. However, they perform slowly with complex and high dimensional problems. Specifically, in the case of classification, this slowness is due to the fact that the model must be evaluated according to a fitness function and training data. Many studies, using different approaches [Har10, SHM+03], have focused on solving this problem by improving the execution time of these algorithms. Recently, the use of GPUs has increased for solving high dimensional and parallelizable problems and in fact, there are already EA models that take advantage of this technology [FKB10]. The main shortcoming of these models is that they do not provide a general purpose model: they are too specific to the problem domain or their efficiency could be significantly improved.

In this paper, we present an efficient and scalable GPU-based parallel evaluation model to speedup the evaluation phase of GP classification algorithms, that overcomes the shortcomings of the previous models. In this way, our proposal is presented as a general model applicable to any domain within the classification task regardless of its complexity and whose philosophy is easily adaptable to any other paradigm.

The proposed model parallelizes the evaluation of the individuals, which is the phase that requires the most computational time in the evolutionary process of EAs. Specifically, the efficient and scalable evaluator model designed uses GPUs to speed up the performance, receiving a classifier and returning the confusion matrix of that classifier on a database. To show the generality of the model proposed, it is applied to some of the most popular GP classification algorithms and several datasets with distinct complexity. Thus,

A. Cano · A. Zafra · S. Ventura
Department of Computing and Numerical Analysis
University of Córdoba, 14071 Córdoba, Spain
Tel.: +34-957212218
E-mail: i52caroa@uco.es, azafra@uco.es, sventura@uco.es

among the datasets used in experiments, there are some widely used as benchmark datasets on the classification task characterized by its simplicity and with others that have not been commonly addressed to date because of their extremely high complexity when applied to previous models. The use of these datasets of varied complexity allows us to demonstrate the performance of our proposal on any problem complexity and domain. Experimental results show the efficiency and generality of our model, which can deal with a variety of algorithms and application domains, providing a great speedup of the algorithm's performance, of up to 820 times, compared to the non-parallel version executed sequentially. Moreover, the speedup obtained is higher when the problem complexity increases. The proposal is compared with other different GPU computing evolutionary learning system called BioHEL [FKB10]. The comparison results show the efficiency far better obtained by our proposal.

The remainder of this paper is organized as follows. Section 2 provides an overview of previous works related to evolutionary classification algorithms and GPU implementations. Section 3 discusses the GP model and analyzes the computational cost associated with its different phases. Section 4 describes the GPU architecture and the CUDA programming model. Section 5 explains our proposal and its advantages as a scalable and efficient evaluation model. Section 6 describes the experimental study. In Section 7, the results will be announced and finally the last section presents the final remarks of our investigation and outlines future research work.

## 2 Related works

GP has been parallelized in multiple ways to take advantage both of different types of parallel hardware and of different features of particular problem domains. Most of the parallel approaches during the last decades deal with the implementation over CPU machine clusters. More recently, works about parallelization have been focusing on using graphics processing units (GPUs) which provide fast parallel hardware for a fraction of the cost of a traditional parallel system. GPUs are devices with multicore architectures and parallel processor units. The GPU consists of a large number of processors and recent devices operate as multiple instruction multiple data (MIMD) architectures. Today, GPUs can be programmed by any user to perform general purpose computation (GPGPU) [GPG]. The use of GPUs has been already studied for speeding up algorithms within the framework of evolutionary computation. Concretely, we can cite some studies about the evaluation process in genetic algorithms and GP on GPUs [Har10].

Previous investigations have focused on two evaluation approaches [BNKF98]: population parallel or fitness parallel and both methods can exploit the parallel architecture of the GPU. In the fitness parallel method, all the fitness cases are executed in parallel with only one individual being evaluated at a time. This can be considered an SIMD approach. In the population parallel method, multiple individuals are evaluated simultaneously. These investigations have proved that for smaller datasets or population sizes, the overhead introduced by uploading individuals to evaluate is larger than the increase in computational speed [CM07]. In these cases there is no benefit in executing the evaluation on a GPU. Therefore, the larger the population size or the number of instances are, the better the GPU implementation will perform. Specifically, the performance of the population parallel approaches is influenced by the size of the population and the fitness case parallel approaches are influenced by the number of fitness cases, i.e., the number of training patterns from the dataset.

Next the more relevant proposals presented to date are discussed. D. Chitty et al. [CM07] describes the technique of general purpose computing using graphics cards and how to extend this technique to GP. The improvement in the performance of GP on single processor architectures is also demonstrated. S. Harding [HB07] goes on to report on how exactly the evaluation of individuals on GP could be accelerated; both proposals are focused on population parallel.

D. Robilliard et al. [RMPF09] proposes a parallelization scheme to exploit the performance of the GPU on small training sets. To optimize with a modest-sized training set, instead of sequentially evaluating the GP solutions parallelizing the training cases, the parallel capacity of the GPU is shared by the GP programs and data. Thus, different GP programs are evaluated in parallel, and a cluster of elementary processors are assigned to each of them to treat the training cases in parallel. A similar technique, but using an implementation based on the single program multiple data (SPMD) model, is proposed by W. Langdon and A. Harrison [LH08]. They implement the evaluation process of GP trees for bioinformatics purposes using GPGPUs, achieving a speedup of around 8X. The use of SPMD instead of SIMD affords the opportunity to achieve increased speedups since, for example, one cluster can interpret the $if$ branch of a test while another cluster treats the $else$ branch independently. On the other hand, performing the same computation inside a cluster is also possible, but the two branches are processed sequentially

in order to respect the SIMD constraint: this is called divergence and is, of course, less efficient. Moreover, Maitre et al. [MBL+09] presented an implementation of a genetic algorithms which performs the evaluation function using a GPU. However, they have a training function instead of a traning set, which they run in parallel over different individuals. Classification fitness computation is based on learning from a training set within the GPU device which implies memory occupancy, while other proposals use a mathematical representation function as the fitness function.

M. Franco et al. [FKB10] introduces a fitness parallel method for computing fitness in evolutionary learning systems using the GPU. Their proposal achieves speedups of up to 52X in certain datasets, performing a reduction function [W. 09] over the results to reduce the memory occupancy. However, this proposal does not scale to multiple devices and its efficiency and its spread to other algorithms or to more complex problems could be improved.

These works are focused on parallellizing the evaluation of multiple individuals or training cases and many of these proposals are limited to small datasets due to memory constraints where exactly GPU are not optimal. By contrast, our proposal is an efficient hybrid population and fitness parallel model, that can be easily adapted to other algorithms, designed to achieve maximum performance solving classification problems using datasets with different dimensions and population sizes.

## 3 Genetic programming algorithms

This section introduces the benefits and the structure of GP algorithms and describes a GP evolutionary system for discovering classification rules in order to understand the execution process and the time required.

GP, the paradigm on which this paper focuses, is a learning methodology belonging to the family of evolutionary algorithms [BFM97] introduced by Koza [Koz92]. GP is defined as an automated method for creating a working computer program from a high-level formulation of a problem. GP performs automatic program synthesis using Darwinian natural selection and biologically inspired operations such as recombination, mutation, inversion, gene duplication, and gene deletion. It is an automated learning methodology used to optimize a population of computer programs according to a fitness function that determines their ability to perform a certain task. Among successful evolutionary algorithm implementations, GP retains a significant position due to such valuable characteristics as: its flexible variable length solution representation, the fact that a priori knowledge is not needed about the statistical distribution of the data (data distribution free), data in their original form can be used to operate directly on them, unknown relationships that exist among data can be detected and expressed as mathematical expressions, and, finally, the most important discriminative features of a class can be discovered. These characteristics suit these algorithms to be a paradigm of growing interest both for obtaining classification rules [DDT01, Fre02, TTLH02] and for other tasks related to prediction, such as feature selection [LKB06] and the generation of discriminant functions [EVH10].

### 3.1 GP classification algorithms

In this section we will detail the general structure of GP algorithms before proceeding to the analysis of its computational cost.

**Individual representation**

GP can be employed to construct classifiers using different kinds of representations, e.g., decision trees, classification rules, discriminant functions, and many more. In our case, the individuals in the GP algorithm are classification rules whose expression tree is composed by terminal and non-terminal nodes. A classifier can be expressed as a set of IF-antecedent-THEN-consequent rules, in which the antecedent of the rule consists of a series of conditions to be met by an instance in order to consider that it belongs to the class specified by the consequent.

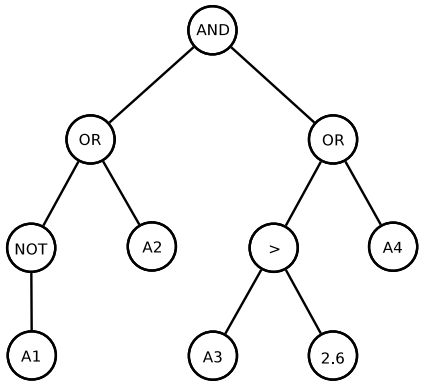


Fig. 1: Example of an individual expression tree.

The rule consequent specifies the class to be predicted for an instance that satisfies all the conditions of the rule antecedent. The terminal set consists of the attribute names and attribute values of the dataset being mined. The function set consists of logical operators

(AND, OR, NOT), relational operators (<, ≤, =, <>, ≥, >) or interval range operators (IN, OUT). These operators are constrained to certain data type restrictions: categorical, real or boolean. Fig. 1 shows an example of the expression tree of an individual.

**Generational model**

The evolution process of GP algorithms [Deb05], similar to other evolutionary algorithms, consists of the following steps:

1. An initial population of individuals is generated using an initialization criterion.
2. Each individual is evaluated based on the fitness function to obtain a fitness value that represents its ability to solve the problem.
3. In each generation, the algorithm selects a subset of the population to be parents of offspring. The selection criterion usually picks the best individuals to be parents, to ensure the survival of the best genes.
4. This subset of individuals is crossed using different crossover operators, obtaining the offspring.
5. These individuals may be mutated, applying different mutation genetic operators.
6. These new individuals must be evaluated using the fitness function to obtain their fitness values.
7. Different strategies can be employed for the replacement of individuals within the population and the offspring to ensure that the population size in the next generation is constant and the best individuals are kept.
8. The algorithm performs a control stage that determines whether to finish the execution by finding acceptable solutions or by having reached a maximum number of generations, if not the algorithm goes back to step 3 and performs a new iteration (generation).

The pseudo-code of a simple generational algorithm is shown in Algorithm 1.

**Algorithm 1** Generational Algorithm

**Require:** max_generations

```
Initialize (P)
Evaluate (P)
num_generations ← 0
while num_generations < max_generations do
    P ← Parent selection (Population)
    C ← Crossover (P)
    M ← Mutation (C)
    Evaluate (M)
    Population ← Replacement (M ∪ Population)
    num_generations++
end while
```

**Evaluation: fitness function**

The evaluation stage is the evaluation of the fitness function over the individuals. When a rule or individual is used to classify a given training instance from the dataset, one of these four possible values can be obtained: true positive ($t_p$), false positive ($f_p$), true negative ($t_n$) and false negative ($f_n$). The true positive and true negative are correct classifications, while the false positive and false negative are incorrect classifications.

- **true positive:** the rule predicts the class and the class of the given instance is indeed that class.
- **false positive:** the rule predicts a class but the class of the given instance is not that class.
- **true negative:** the rule does not predict the class and the class of the given instance is indeed not that class.
- **false negative:** the rule does not predict the class but the class of the given instance is in fact that class.

The results of the individual's evaluations over all the patterns from a dataset are used to build the confusion matrix which allows us to apply different quality indexes to get the individual's fitness value and its calculation is usually the one that requires more computing time. Therefore, our model will also perform this calculation so that each algorithm can apply the most convenient fitness function.

The main problem of the evaluation is the computational time required for the match process because it involves comparing all the rules with all the instances of the dataset. The number of evaluations is huge when the population size or the number of instances increases, thus the algorithm must perform up to millions of evaluations in each generation.

**Evaluation: computational study**

Several previous experiments have been conducted to evaluate the computational time of the different stages of the generational algorithm. These experiments execute the different algorithms described in Section 6.1 over the problem domains proposed in Section 6.3. The population size was set to 50, 100 and 200 individuals, whereas the number of generations was set to 100 iterations. The results of the average execution time of the different stages of the algorithms among all the configurations are shown in Table 1.

The experience using these GP algorithms proves that on average around 94% of the time is taken by the evaluation stage. This percentage is mainly linked

to the algorithm, the population size and the number of patterns, increasing up to 99% on large problems. Anyway, the evaluation phase is always more expensive regardless of the algorithm or its parameters. We can conclude that evaluation takes most of the execution time so the most significant improvement would be obtained by accelerating this phase. Therefore, we propose a parallel GPU model detailed in Section 5 to speed up the evaluation phase.

Table 1: GP classification execution time.

| Phase | Percentage |
|---|---|
| Initialization | 8.96% |
| Creation | 0.39% |
| Evaluation | 8.57% |
| Generation | 91.04% |
| Selection | 0.01% |
| Crossover | 0.01% |
| Mutation | 0.03% |
| Evaluation | 85.32% |
| Replacement | 0.03% |
| Control | 5.64% |
| Total | 100 % |

## 4 CUDA programming model

Computer unified device architecture (CUDA) [CUD] is a parallel computing architecture developed by NVIDIA that allows programmers to take advantage of the computing capacity of NVIDIA GPUs in a general purpose manner. The CUDA programming model executes kernels as batches of parallel threads in a SIMD programming style. These kernels comprise thousands to millions of lightweight GPU threads per each kernel invocation.

CUDA's threads are organized into a two-level hierarchy represented in Fig. 2: at the higher one, all the threads in a data-parallel execution phase form a grid. Each call to a kernel execution initiates a grid composed of many thread groupings, called thread blocks. All the blocks in a grid have the same number of threads, with a maximum of 512. The maximum number of thread blocks is 65535 x 65535, so each device can run up to 65535 x 65535 x 512 = $2 \cdot 10^{12}$ threads per kernel call.

To properly identify threads within the grid, each thread in a thread block has a unique ID in the form of a three-dimensional coordinate, and each block in a grid also has a unique two-dimensional coordinate.

Thread blocks are executed in streaming multiprocessors. A stream multiprocessor can perform zero overhead scheduling to interleave warps and hide the overhead of long-latency arithmetic and memory operations.

There are four different main memory spaces: global, constant, shared and local. These GPU memories are specialized and have different access times, lifetimes and output limitations.

- **Global memory:** is a large, long-latency memory that exists physically as an off-chip dynamic device memory. Threads can read and write global memory to share data and must write the kernel's output to be readable after the kernel terminates. However, a better way to share data and improve performance is to take advantage of shared memory.
- **Shared memory:** is a small, low-latency memory that exists physically as on-chip registers and its contents are only maintained during thread block execution and are discarded when the thread block completes. Kernels that read or write a known range of global memory with spatial or temporal locality can employ shared memory as a software-managed cache. Such caching potentially reduces global memory bandwidth demands and improves overall performance.
- **Local memory:** each thread also has its own local memory space as registers, so the number of registers a thread uses determines the number of concurrent threads executed in the multiprocesor, which is called multiprocessor occupancy. To avoid wasting hundreds of cycles while a thread waits for a long-latency global-memory load or store to complete, a common technique is to execute batches of global accesses, one per thread, exploiting the hardware's warp scheduling to overlap the threads' access latencies.
- **Constant memory:** is specialized for situations in which many threads will read the same data simultaneously. This type of memory stores data written by the host thread, is accessed constantly and does not change during the execution of the kernel. A value read from the constant cache is broadcast to all threads in a warp, effectively serving 32 loads from memory with a single-cache access. This enables a fast, single-ported cache to feed multiple simultaneous memory accesses. The amount of constant memory is 64 KB.

For maximum performance, these memory accesses must be coalesced as with accesses to global memory. Global memory resides in device memory and is accessed via 32, 64, or 128-byte segment memory transactions. When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of

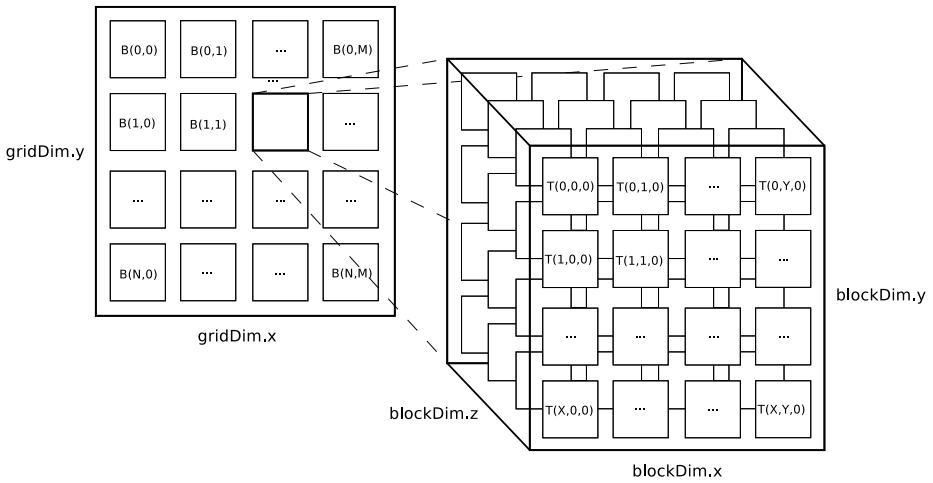


Fig. 2: CUDA threads and blocks model.

the memory addresses across the threads. In general, the more transactions are necessary, the more unused words are transferred in addition to the words accessed by the threads, reducing the instruction throughput accordingly.

To maximize global memory throughput, it is therefore important to maximize coalescing by following the most optimal access patterns, using data types that meet the size and alignment requirement or padding data in some cases, for example, when accessing a two-dimensional array. For these accesses to be fully coalesced, both the width of the thread block and the width of the array must be multiple of the warp size.

## 5 Model description

This section details an efficient GPU-based evaluation model for fitness computation. Once it has been proved that the evaluation phase is the one that requires the most of the computational time, this section discusses the procedure of the fitness function to understand its cost in terms of runtime and memory occupancy. We then employ this knowledge to propose an efficient GPU-based evaluation model in order to maximize the performance based on optimization principles [RRB+08] and the recommendations of the NVIDIA CUDA programming model guide [CUD].

### 5.1 Evaluation complexity

The most computationally expensive phase is evaluation since it involves the match of all the individuals generated over all the patterns. Algorithm 2 shows the pseudo-code of the fitness function. For each individual its genotype must be interpreted or translated into an executable format and then it is evaluated over the training set. The evaluation process of the individuals is usually implemented in two loops, where each individual iterates each pattern and checks if the rule covers that pattern. Considering that the population size is $P$ and the training set size is $T$, the number of iterations is $O(P \times T)$. These two loops make the algorithm really slow when the population size or the pattern count increases because the total number of iterations is the product of these two parameters. This *one by one* iterative model is slow but it only requires $4 \times populationSize \times sizeof(int)$ bytes from memory, i.e., the four integer counters for $t_p$, $t_n$, $f_p$ and $f_n$ values for each individual, this is $O(P)$ complex.

**Algorithm 2** Evaluation: fitness function

**Require:** population_size, number_instances

```
1: for each individual within the population do
2:     tp ← 0, fp ← 0, tn ← 0, fn ← 0
3:     for each instance from the dataset do
4:         if individual's rule covers actual instance then
5:             if the consequent matches predicted class then
6:                 tp++
7:             else
8:                 fp++
9:             end if
10:        else
11:            if the consequent matches predicted class then
12:                fn++
13:            else
14:                tn++
15:            end if
16:        end if
17:    end for
18:    fitnessValue ← computeFitness(tp,tn,fp,fn)
19: end for
```

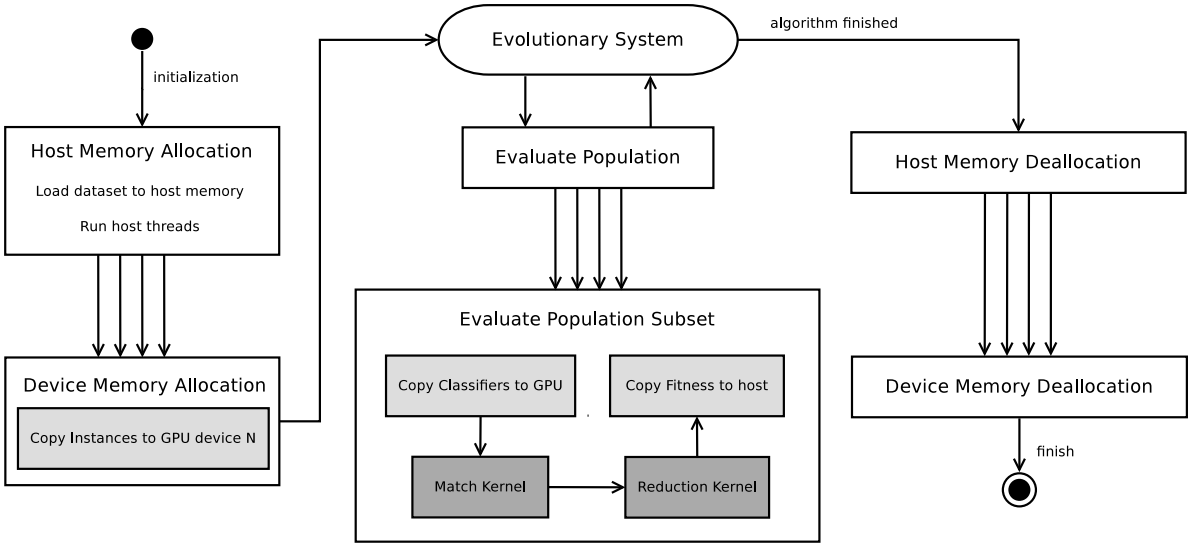


Fig. 3: Model schema.

### 5.2 Efficient GPU-based evaluation

The execution of the fitness function over the individuals is completely independent from one individual to another. Hence, the parallelization of the individuals is feasible. A naive way to do this is to perform the evaluations in parallel using several CPU threads, one per individual. The main problem is that affordable PC systems today only run CPUs with 4 or 8 cores, thus larger populations will need to serialize its execution so the speedup would be limited up to the number of cores that there is. This is where GPGPU systems can exploit its massively parallel model.

Using the GPU, the fitness function can be executed over all individuals concurrently. Furthermore, the simple match of a rule over an instance is a self-dependent operation: there is no interference with any other evaluation. Hence, the matches of all the individuals over all the instances can be performed in parallel in the GPU. This means that one thread represents the single match of a pattern over an instance. The total number of GPU threads required would be equal to the number of iterations from the loop of the sequential version. Once each thread has obtained the result of its match in the GPU device, these results have to be copied back to the host memory and summed up to get the fitness values. This approach would be very slow because in every generation it will be necessary to copy a structure of size $O(P \times T)$, specifically $populationSize \times numberInstances \times sizeof(int)$ bytes from device memory to host memory, i.e, copying the match results obtained from the coverage of every individual over every pattern. This is completely inefficient because the copy transactions time would be larger than the speedup obtained. Therefore, to reduce the copy structure size it is necessary to calculate the final result for each individual inside the GPU and then only copy a structure size $O(P)$ containing the fitness values. Hence, our fitness calculation model involves two steps: matching process and reducing the results for fitness values computation. These two tasks correspond to two different kernels detailed in 5.2.2.

The source code of our model can be compiled into a shared library to provide the user the functions to perform evaluations in GPU devices for any evolutionary system. The schema of the model is shown in Fig. 3.

At first, the user must call a function to perform the dynamic memory allocation. This function allocates the memory space and loads the dataset instances to the GPU global memory. Moreover, it runs one host thread per GPU device because the thread context is mandatorily associated to only one GPU device. Each host thread runs over one GPU and performs the evaluation of a subset of the individuals from the population. The execution of the host threads stops once the instances are loaded to the GPU awaiting a trigger. The evaluate function call is the trigger that wakes the threads to perform the evaluations over their population's subset. Evaluating the present individuals require the copy of their phenotypes to a GPU memory space. The GPU constant memory is the best location for storing the individual phenotypes because it provides broadcast to all the device threads in a warp. The host threads execute the kernels as batches of parallel threads, first the match kernel obtains the results of the match process and then the reduction kernel calculates the fitness values from these results. The fitness values must be copied back to host memory and associated to the indi-

| A(0,0) | A(1,0) | ... | A(N,0) | A(0,1) | A(1,1) | ... | A(N,1) | ... | A(0,I) | ... | A(N,I) |
|---|---|---|---|---|---|---|---|---|---|---|---|

Fig. 4: Uncoalesced instances data array structure.

Fig. 5: Uncoalesced attributes request.

| A(0,0) | A(0,1) | ... | A(0,I) | A(1,0) | A(1,1) | ... | A(1,I) | ... | A(N,0) | ... | A(N,I) |
|---|---|---|---|---|---|---|---|---|---|---|---|

Fig. 6: Coalesced instances data array structure.

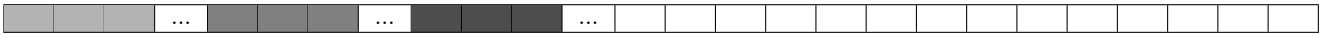

Fig. 7: Coalesced attributes request.

| A(0,0) | ... | A(0,I) | X | A(1,0) | ... | A(1,I) | X | ... | A(N,0) | ... | A(N,I) | X |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Fig. 8: Fully coalesced intra-array padding instances data array structure.

Fig. 9: Fully coalesced attributes request.

| R(0,0) | ... | R(0,I) | X | R(1,0) | ... | R(1,I) | X | ... | R(N,0) | ... | R(N,I) | X |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Fig. 10: Results data array structure.

viduals. Once all the individuals from the thread have been evaluated, the host thread sends a signal to the main thread telling it its job has finished and the algorithm process can continue once all the host threads have sent the go-ahead. This stop and go model continues while more generations are performed. At the end, a free memory function must be called to deallocate the dynamic memory previously allocated.

### *5.2.1 Data structures*

The scheme proposed attempts to make full use of global and constant memory. The purpose is to optimize memory usage to achieve maximum memory throughput. Global memory is employed to store the instances from the dataset, the results of the match process, and the fitness values.

The most common dataset structure is a 2D matrix where each row is an instance and each column is an attribute. Loading the dataset to the GPU is simple, allocate a 2D array of width $numberInstances \times numberAttributes$ and copy the instances to the GPU. This approach, represented in Fig. 4, is simple-minded and it works but for maximum performance, the memory accesses must be coalesced. When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more 32, 64, or 128-byte memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. In general, the more transactions are necessary, the more unused words are transferred in addition to the words accessed by the threads, reducing the instruction throughput accordingly.

The threads in the match kernel perform the match process of the classifiers over the instances. The $i$ thread performs the match process over the $i$ instance. Therefore, consecutive threads are executed over consecutive instances. The threads execute the phenotype of the individuals represented in reverse polish notation using a stack and an interpreter. The phenotype follows the individual representation shown in 3.1. Thus, when attribute nodes have to obtain the attribute values for each instance covered by the threads within the warp, the attributes addresses are spaced *numberAttributes* memory addresses (stride is $numberAttributes \times sizeof(datatype)$). Depending on the number of attributes, a memory transaction would transfer more or less useful values. Anyway, this memory access pattern shown in Fig. 5 is altogether inefficient because the memory transfer engine must split the memory request into many memory transactions that are issued independently.

The second approach shown in Fig. 6 for storing the instances is derived from the first one. The problem

is the stride of the memory requests which is *numberAttributes*. The solution is to lower the stride to one transposing the 2D array that stores the instances. The length of the array remains constant but instead of storing all the attributes of an instance first, it stores the first attributes from all the instances. Now, the memory access pattern shown in Fig. 7 demands attributes which are stored in consecutive memory addresses. Therefore, a single 128-byte memory transaction would transfer the 32 integer or float attributes requested by the threads in the warp.

For these accesses to be fully coalesced, both the width of the thread block and the number of the instances must be a multiple of the warp size. The third approach for achieving fully coalesced accesses is shown in Figs. 8 and 9. Intra-array padding is necessary to align the addresses requested to the memory transfer segment sizes. Thus, the array must be expanded to $multiple(numberInstances, 32) \times numberAttributes$ values.

The individuals to be evaluated must be uploaded in each generation to the GPU constant memory. The GPU has a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory. All the threads in a warp perform the match process of the same individual over different instances. Thus, memory requests point to the same node and memory address at a given time. Servicing one memory read request to several threads simultaneously is called broadcast. The resulting requests are serviced at the throughput of the constant cache in case of a cache hit, or at the throughput of device memory otherwise.

The results of the match process for each individual and instance must be stored in global memory for counting. Again, the memory accesses must be coalesced to device global memory. The best data structure is a 2D array $numberInstances \times populationSize$ shown in Fig. 10. Hence, the results write operations and the subsequent read operations for counting are both fully coalesced.

The fitness values calculated by the reduction kernel must be stored in global memory, then copied back to host memory and set to the individuals. A simple structure to store the fitness values of the individuals is a 1D array of length *populationSize*.

### *5.2.2 Evaluation process on GPU*

The evaluation process on the GPU is performed using two kernel functions. The first kernel performs the match operations between the individuals and the instances storing a certain result. Each thread is in charge of a single match. The second kernel counts the results of an individual by a reduction operation. This 2-kernel model allows the user to perform the match processes and the fitness values calculations completely independently. Once the results of the match process are obtained, any fitness function can be employed to calculate the fitness values. This requires copying back to global memory a large amount of data at the end of the first kernel. M. Franco [FKB10] proposes to minimise the volume of data by performing a reduction in the first kernel. However, the experiments carried out indicate to us that the impact in the run-time of reducing data in the first kernel is larger than that of storing the whole data array because our approach allows the kernels to avoid synchronization between threads and unnecessary delays. Furthermore, the threads block dimensions can be ideally configured for each kernel independently.

#### Match kernel

The first kernel performs in parallel the match operations between the classifiers and the instances. Algorithm 3 shows the pseudo-code for this kernel.

**Algorithm 3** Match kernel

```
 1: instance ← blockDim.y * blockIdx.y + threadIdx.y;
 2: if instance < numberInstances then
 3:     resultMemPosition ← blockIdx.x * numberInstancesAligned + instance;
 4:     if covers(classifier[blockIdx.x],instance) then
 5:         if classifiedClass == instancesClass[instance] then
 6:             result[resultMemPosition] ← tp
 7:         else
 8:             result[resultMemPosition] ← fp
 9:         end if
10:     else
11:         if classifiedClass != instancesClass[instance] then
12:             result[resultMemPosition] ← tn
13:         else
14:             result[resultMemPosition] ← fn
15:         end if
16:     end if
17: end if
```

The number of matches and hence the total number of threads is $populationSize \times numberInstances$. The maximum amount of threads per block is 512 or 1024 depending on the device's computing capability. However, optimal values are multiples of the warp size. A GPU multiprocessor relies on thread-level parallelism to maximize utilization of its functional units. Utilization is therefore directly linked to the number of resident warps. At every instruction issue time, a warp scheduler selects a warp that is ready to execute, if any, and issues the next instruction to the active threads of

the warp. The number of clock cycles it takes for a warp to be ready to execute its next instruction is called latency, and full utilization is achieved when the warp scheduler always has some instruction to issue for some warp at every clock cycle during that latency period, i.e., when the latency of each warp is completely hidden by other warps.

The CUDA occupancy calculator spreadsheet allows of computing the multiprocessor occupancy of a GPU by a given CUDA kernel. The multiprocessor occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU. The optimal number of threads per block obtained from the experiments carried out for this kernel is 128 for devices of compute capability 1.x, distributed in 4 warps of 32 threads. The active thread blocks per multiprocessor is 8. Thus, the active warps per multiprocessor is 32. This means a 100% occupancy of each multiprocessor for devices of compute capability 1.x. Recent devices of compute capability 2.x requires 192 threads per block to achieve 48 active warps per multiprocesor and a 100% occupancy. The number of threads per block does not matter, since the model is adapted to achieve maximum performance in any case.

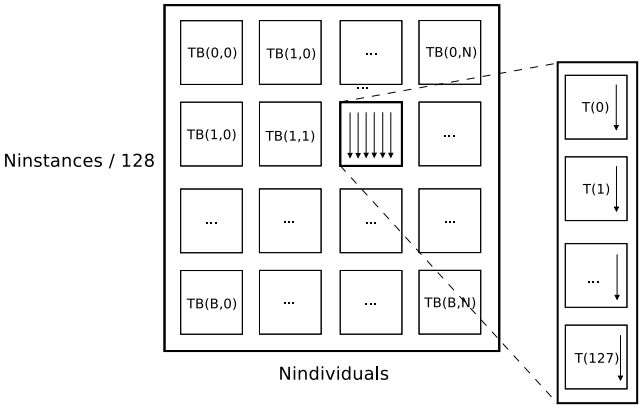


Fig. 11: Match kernel 2D grid of thread blocks.

The kernel is executed using a 2D grid of thread blocks as shown in Fig. 11. The first dimension length is *populationSize*. Using $N$ threads per block, the number of thread blocks to cover all the instances is $ceil(numberInstances/N)$ in the second dimension of the grid. Thus, the total number of thread blocks is $populationSize \times ceil(numberInstances/N)$. This number is important as it concerns the scalability of the model in future devices. NVIDIA recommends that one run at least twice as many thread blocks as the number of multiprocessors.

**Reduction kernel**

The second kernel reduces the results previously calculated in the first kernel and obtains the fitness value for each individual. The naive reduction operation shown in Fig. 13 sums in parallel the values of an array reducing iteratively the information. Our approach does not need to sum the values, but counting the number of $t_p$, $f_p$, $t_n$ and $f_n$ resulted for each individual from the match kernel. These four values are employed to build the confusion matrix. The confusion matrix allows us to apply different quality indexes defined by the authors to get the individual's fitness value.

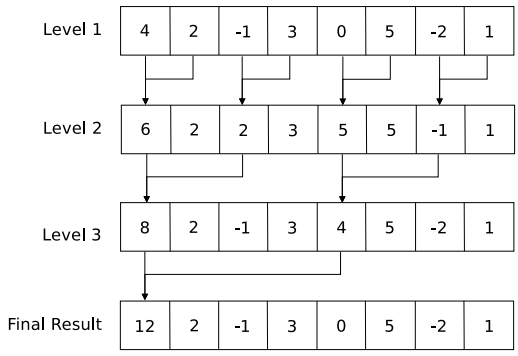


Fig. 13: Parallel reduction algorithm.

Designing an efficient reduction kernel is not simple because it is the parallelization of a natively sequential task. In fact, NVIDIA propose six different approaches [CUD]. Some of the proposals take advantage of the device shared memory. Shared memory provides a small but fast memory shared by all the threads in a block. It is quite desirable when the threads require synchronization and work together to accomplish a task like reduction.

A first approach to count the number of $t_p$, $f_p$, $t_n$ and $f_n$ in the results array using $N$ threads is immediate. Each thread counts for the Nth part of the array, specifically $numberInstances/numberThreads$ items, and then the values for each thread are summed. This 2-level reduction is not optimal because the best would be the $N$/2-level reduction, but reducing each level requires the synchronization of the threads. Barrier synchronization can impact performance by forcing the multiprocessor to idle. Therefore, a 2 or 3-level reduction has been proved to perform the best.

To achieve a 100% occupancy, the reduction kernel must employ 128 or 192 threads, for devices of compute capability 1.x or 2.x, respectively. However, it is not trivial to organize the threads to count the

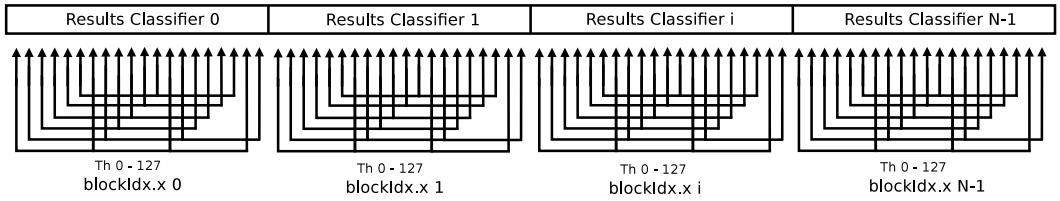


Fig. 12: Coalesced reduction kernel.

items. The first approach involves the thread $i$ counting $numberInstances/numberThreads$ items from the $threadIdx \times numberInstances/numberThreads$ item. The threads in a thread-warp would request the items spaced $numberInstances$ memory addresses. Therefore, once again one has a coalescing and undesirable problem. Solving the memory requests pattern is naive. The threads would count again $numberInstances/numberThreads$ items but for coalescing purposes the memory access pattern would be $iteration \times numberThreads + threadIdx$. This way, the threads in a warp request consecutive memory addresses that can be serviced in fewer memory transactions. This second approach is shown in Fig. 12. The reduction kernel is executed using a 1D grid of thread blocks whose length is *populationSize*. Using 128 or 192 threads per block, each thread block performs the reduction of the results for an individual. A shared memory array of length $4 \times numberThreads$ keeps the temporary counts for all the threads. Once all the items have been counted, a synchronization barrier is called and the threads wait until all the threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to the synchronization point are visible to all threads in the block. Finally, only one thread per block performs the last sum, calculates the fitness value and writes it to global memory.

## 6 Experimental study

This section describes the details of the experiments, discusses the application domains, the algorithms used, and the settings of the tests.

### 6.1 Parallelized methods with our proposal

To show the flexibility and applicability of our model, three different GP classification algorithms proposed in the literature are tested using our proposal in the same way as could be applied to other algorithms or paradigms. Next, the major specifications of each of the proposals that have been considered in the study are detailed.

1) Falco, Cioppa and Tarantino [DDT01] propose a method to get the fitness of the classifier by evaluating the antecedent over all the patterns within the dataset. Falco et al. uses the logical operators AND, OR, NOT, the relational operators $=, <, >, \leq, \geq$ and two numerical interval comparator operators IN and OUT. The evolution process is repeated as many times as classes holds the dataset. In each iteration the algorithm focuses on a class and keeps the best rule obtained to build the classifier.

The crossover operator selects two parent nodes and swaps the two subtrees. The crossover is contrained by two restrictions: the nodes must be compatible and the depth of the tree generated must not exceed a preset depth.

The mutation operator can be applied either to a terminal node or a non terminal node. This operator selects a node randomly, if it is a terminal node, it is replaced by another randomly selected compatible terminal node, otherwise, the non terminal node is replaced by another randomly selected compatible terminal node with the same arity and compatibility.

The fitness function calculates the difference between the number of instances where the rule correctly predicts the membership or not of the class and number of examples where the opposite occurs and the prediction is wrong. Finally, the fitness function is defined as:

$$fitness = nI - ((t_p + t_n) - (f_p + f_n)) + \alpha * N$$

where $nI$ is the number of instances, $\alpha$ is a value between 0 and 1 and $N$ is the number of nodes in the rule. The closer is $\alpha$ to 1, the more importance is given to simplicity.

2) Tan, Tay, Lee and Heng [TTLH02] proposes a modified version of the steady-state algorithm [BNKF98] which uses an external population and elitism to ensure that some of the best individuals of the current generation survive in the next generation.

The fitness function combines two indicators that are commonplace in the domain, namely the sensitivity ($Se$) and the specifity ($Sp$), defined as follows:

$$Se = \frac{t_p}{t_p + w1 * f_n} \qquad Sp = \frac{t_n}{t_n + w2 * f_p}$$

The parameters $w1$ and $w2$ are used to weight the influence of the false negatives and false positives cases in the fitness calculation. This is very important because these values are critical in problems such as diagnosis. Decreasing $w1$ or increasing $w2$ generally improves the results but also increases the number of rules. The range [0.2 to 1] for $w1$ y [1-20] for $w2$ is usually reasonable for the most cases. Therefore, the fitness function is defined by the product of these two parameters.

$$fitness = Se * Sp$$

The proposal of Tan et al. is similar to that of Falco et al. but the OR operator because combinations of AND and NOT operators which can generate all the necessary rules. Therefore, the simplicity of the rules is affected. Tan et al. also introduces the token competition technique proposed by Wond and Leung [WL00] and it is employed as an alternative niche approach to promote diversity. Most of the time, only a few rules are useful and cover most of the instances while most others are redundant. The token competition is an effective way to eliminate redundant rules.

3) Bojarczuk, Lopes and Freitas [BLFM04] presents a method in which each rule is evaluated for all of the classes simultaneously for a pattern. The classifier is formed by taking the best individual for each class generated during the evolutionary process. The Bojarczuk et al. algorithm does not have a mutation operator.

This proposal uses the logical operators AND, OR and NOT, although AND and NOT would be sufficient, this way the size of the generated rules is reduced. GP does not produce simple solutions. The comprehensibility of a rule is inversely proportional to its size. Therefore Bojarczuk et al. define the simplicity ($Sy$) of a rule:

$$Sy = \frac{maxnodes - 0.5 * numnodes - 0.5}{maxnodes - 1}$$

where *maxnodes* is the maximum depth of the syntax-tree, *numnodes* is the number of nodes of the current rule, and $Se$ and $Sp$ are the sensitivity and the specifity parameters described in the Tan et al., with $w1$ and $w2$ equal to 1. The fitness value is the product of these three parameters.

$$fitness = Se * Sp * Sy$$

These three methods implement the match kernel and the reduction kernel. The match kernel obtains the results from the match processes of the prediction of the examples with their actual class. The reduction kernel counts the $t_p$, $t_n$, $f_p$ and $f_n$ values and computes the fitness values.

### 6.2 Comparison with other proposal

One of the most recent works and similar to our proposal is the one by M. Franco et al. [FKB10]. This work speeds up the evaluation of the BioHEL system using GPGPUs. BioHEL is an evolutionary learning system designed to cope with large-scale datasets. They provide the results, the profiler information, the CUDA and the serial version of the software in the website `http://www.cs.nott.ac.uk/~mxf/biohel`. The experiments carried out compare our model and its speedup to the speedup obtained from the CUDA version of BioHEL system over several problem domains in order to demonstrate the improvements provided by the parallelization model proposed. The configuration settings of the BioHEL system were the provided by the authors in the configuration files.

### 6.3 Problem domains used in the experiments

To evaluate the performance of the proposed GP evaluation model, some datasets selected from the UCI machine learning repository [NA07] and the KEEL website [AFSG+09] are benchmarked using the algorithms previously described. These datasets are very varied considering different degrees of complexity. Thus, the number of instances ranges from the simplest containing 150 instances to the most complex containing one million instances. Also, the number of attributes and classes are different in different datasets. This information is summarized in Table 2. The wide variety of datasets considered allows us to evaluate the model performance in both low and high problem complexity. It is interesting to note that some of these datasets such as KDDcup or Poker have not been commonly addressed to date because they are not memory and CPU manageable by traditional models.

Table 2: Complexity of the datasets tested.

| Dataset | #Instances | #Attributes | #Classes |
|---|---|---|---|
| Iris | 150 | 4 | 3 |
| New-thyroid | 215 | 5 | 3 |
| Ecoli | 336 | 7 | 8 |
| Contraceptive | 1473 | 9 | 3 |
| Thyroid | 7200 | 21 | 3 |
| Penbased | 10992 | 16 | 10 |
| Shuttle | 58000 | 9 | 7 |
| Connect-4 | 67557 | 42 | 3 |
| KDDcup | 494020 | 41 | 23 |
| Poker | 1025010 | 10 | 10 |

### 6.4 General experimental settings

The GPU evaluation code is compiled into a shared library and loaded into the JCLEC [VRZ+07] framework using JNI. JCLEC is a software system for evolutionary computation research developed in the Java programming language. Using the library, our model can be easily employed in any evolutionary learning system.

Experiments were run on two PCs both equipped with an Intel Core i7 quad-core processor running at 2.66GHz and 12 GB of DDR3 host memory. One PC features two NVIDIA GeForce 285 GTX video cards equipped with 2GB of GDDR3 video RAM and the other one features two NVIDIA GeForce 480 GTX video cards equipped with 1.5GB of GDDR5 video RAM. No overclock was made to any of the hardware. The operating system was GNU/Linux Ubuntu 10.4 64 bit.

The purpose of the experiments is to analyze the effect of the dataset complexity on the performance of the GPU evaluation model and the scalability of the proposal. Each algorithm is executed over all the datasets using a sequential approach, a threaded CPU approach, and a massively parallel GPU approach.

## 7 Results

This section discusses the experimental results. The first section compares the performance of our proposal over different algorithms. The second section provides the results of the BioHEL system and compares them with the obtained by our proposal.

### 7.1 Results obtained using our proposal

In this section we discuss the performance achieved by our proposal using three different GP algorithms. The execution time and the speedups of the three classification algorithms solving the various problems considered are shown in Tables 3, 4 and 5 where each column is labeled with the execution configuration indicated from left to right as follows: the dataset, the execution time of the native sequential version coded in Java expressed in seconds, the speedup of the model proposed using JNI and one CPU thread, two CPU threads, four CPU threads, one GTX 285 GPU, two GTX 285 GPUs, and with one and two GTX 480 GPUs. The results correspond to the executions of the algorithms with a population of 200 individuals and 100 generations.

The results in the tables provide useful information that in some cases, the external CPU evaluation is inefficient for certain datasets such as Iris, New-thyroid or Ecoli. This is because the time taken to transfer the data from the Java virtual machine memory to the native memory is higher than just doing the evaluation in the Java virtual machine. However, in all the cases, regardless of the size of the dataset, the native GPU evaluation is always considerably faster. If we look at the results of the smallests datasets such as Iris, Newthyroid and Ecoli, it can be seen that its speedup is acceptable and specifically Ecoli performs up to 25X faster. Speeding up these small datasets would not be too useful because of the short run time required, but it is worthwhile for larger data sets. On the other hand, if we focus on complex datasets, the speedup is greater because the model can take full advantage of the GPU multiprocessors' offering them many instances to parallelize. Notice that KDDcup and Poker datasets perform up to 653X and 820X faster, respectively. We can also appreciate that the scalability of the proposal is almost perfect, since doubling the number of threads or the graphics devices almost halves the execution time. Fig. 14 summarizes the average speedup, depending on the number of instances.

The fact of obtaining significant enhacements in all problem domains (both small and complex datasets) as has been seen is because our proposal is a hybrid model that takes advantage of both the parallelization of the individuals and the instances. A great speedup is not only achieved by classifying a large number of instances but by a large enough population. The classification of small datasets does not require many individuals but high dimensional problems usually require a large population to provide diversity in the population genetics. Therefore, a great speedup is achieved by maximizing both parameters. These results allow us to determine that the proposed model achieves a high speedup in the algorithms employed. Specifically, the best speedup is 820X when using the Falco et al. algorithm and the poker dataset, hence the execution time can be impressively reduced from 30 hours to only two minutes.

### 7.2 Results of other proposal

This section discusses the results obtained by BioHEL. The results for the BioHEL system are shown in Table 6 where the first column indicates the execution time of the serial version expressed in seconds, the second column shows the speedup of the CUDA version using a NVIDIA GTX 285 GPU, and the third using a NVIDIA GTX 480 GPU. These results show that for a dataset with a low number of instances the CUDA version of BioHEL performs slower than the serial version. However, the speedup obtained is higher when the number of instances increases, achieving a speedup of up to 34 times compared to the serial version.

Table 3: Falco et al. algorithm execution time and speedups.

| Execution Time (s) | | Speedup | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Dataset | Java | 1 CPU | 2 CPU | 4 CPU | 1 285 | 2 285 | 1 480 | 2 480 |
| Iris | 2.0 | 0.48 | 0.94 | 1.49 | 2.96 | 4.68 | 2.91 | 8.02 |
| New-thyroid | 4.0 | 0.54 | 1.03 | 1.99 | 4.61 | 9.46 | 5.18 | 16.06 |
| Ecoli | 13.7 | 0.49 | 0.94 | 1.38 | 6.36 | 10.92 | 9.05 | 17.56 |
| Contraceptive | 26.6 | 1.29 | 2.52 | 3.47 | 31.43 | 55.29 | 50.18 | 93.64 |
| Thyroid | 103.0 | 0.60 | 1.15 | 2.31 | 37.88 | 69.66 | 75.70 | 155.86 |
| Penbased | 1434.1 | 1.15 | 2.26 | 4.37 | 111.85 | 207.99 | 191.67 | 391.61 |
| Shuttle | 1889.5 | 1.03 | 2.02 | 3.87 | 86.01 | 162.62 | 182.19 | 356.17 |
| Connect-4 | 1778.5 | 1.09 | 2.14 | 3.87 | 116.46 | 223.82 | 201.57 | 392.86 |
| KDDcup | 154183.0 | 0.91 | 1.77 | 3.30 | 136.82 | 251.71 | 335.78 | 653.60 |
| Poker | 108831.6 | 1.25 | 2.46 | 4.69 | 209.61 | 401.77 | 416.30 | 820.18 |

Table 4: Bojarczuk et al. algorithm execution time and speedups.

| Execution Time (s) | | Speedup | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Dataset | Java | 1 CPU | 2 CPU | 4 CPU | 1 285 | 2 285 | 1 480 | 2 480 |
| Iris | 0.5 | 0.50 | 0.96 | 1.75 | 2.03 | 4.21 | 2.39 | 5.84 |
| New-thyroid | 0.8 | 0.51 | 1.02 | 1.43 | 2.99 | 5.85 | 3.19 | 9.45 |
| Ecoli | 1.3 | 0.52 | 1.02 | 1.15 | 3.80 | 8.05 | 5.59 | 11.39 |
| Contraceptive | 5.4 | 1.20 | 2.40 | 2.47 | 14.58 | 31.81 | 26.41 | 53.86 |
| Thyroid | 27.0 | 0.56 | 1.11 | 2.19 | 25.93 | 49.53 | 56.23 | 120.50 |
| Penbased | 42.6 | 0.96 | 1.92 | 3.81 | 18.55 | 36.51 | 68.01 | 147.55 |
| Shuttle | 222.5 | 1.18 | 2.35 | 4.66 | 34.08 | 67.84 | 117.85 | 253.13 |
| Connect-4 | 298.9 | 0.69 | 1.35 | 2.65 | 42.60 | 84.92 | 106.14 | 214.73 |
| KDDcup | 3325.8 | 0.79 | 1.55 | 2.98 | 30.89 | 61.80 | 135.28 | 306.22 |
| Poker | 6527.2 | 1.18 | 2.32 | 4.45 | 39.02 | 77.87 | 185.81 | 399.85 |

Table 5: Tan et al. algorithm execution time and speedups.

| Execution Time (s) | | Speedup | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Dataset | Java | 1 CPU | 2 CPU | 4 CPU | 1 285 | 2 285 | 1 480 | 2 480 |
| Iris | 2.6 | 0.44 | 0.80 | 1.01 | 2.94 | 5.44 | 4.90 | 9.73 |
| New-thyroid | 6.0 | 0.77 | 1.43 | 1.78 | 7.13 | 12.03 | 9.15 | 21.74 |
| Ecoli | 22.5 | 0.60 | 1.16 | 2.09 | 9.33 | 16.26 | 14.18 | 25.92 |
| Contraceptive | 39.9 | 1.28 | 2.44 | 3.89 | 40.00 | 64.52 | 60.60 | 126.99 |
| Thyroid | 208.5 | 1.10 | 2.11 | 2.66 | 64.06 | 103.77 | 147.74 | 279.44 |
| Penbased | 917.1 | 1.15 | 2.23 | 4.25 | 86.58 | 148.24 | 177.78 | 343.24 |
| Shuttle | 3558.0 | 1.09 | 2.09 | 3.92 | 95.18 | 161.84 | 222.96 | 431.80 |
| Connect-4 | 1920.6 | 1.35 | 2.62 | 4.91 | 123.56 | 213.59 | 249.81 | 478.83 |
| KDDcup | 185826.6 | 0.87 | 1.69 | 3.20 | 83.82 | 138.53 | 253.83 | 493.14 |
| Poker | 119070.4 | 1.27 | 2.46 | 4.66 | 158.76 | 268.69 | 374.66 | 701.41 |

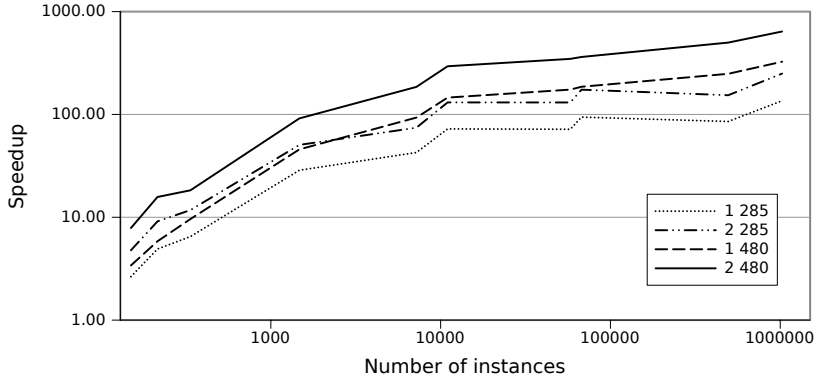


Fig. 14: Average speedups.

Table 6: BioHEL execution time and speedups.

| Execution Time (s) | | Speedup | |
|---|---|---|---|
| Dataset | Serial | CUDA 285 | CUDA 480 |
| Iris | 0.5 | 0.64 | 0.66 |
| New-thyroid | 0.9 | 0.93 | 1.32 |
| Ecoli | 3.7 | 1.14 | 6.82 |
| Contraceptive | 3.3 | 3.48 | 3.94 |
| Thyroid | 26.4 | 2.76 | 8.70 |
| Penbased | 147.9 | 5.22 | 20.26 |
| Shuttle | 418.4 | 11.54 | 27.84 |
| Connect-4 | 340.4 | 10.18 | 12.24 |
| KDDcup | 503.4 | 14.95 | 28.97 |
| Poker | 3290.9 | 11.93 | 34.02 |

The speedup results for the BioHEL system shown in Table 6 compared with the results obtained by our proposal shown in Tables 3, 4 and 5 demonstrate the better performance of our model. One of the best advantages of our proposal is that it scales to multiple GPU devices whereas BioHEL does not. Both BioHEL and our proposal employ a 2 kernel model. However, we do not to perform a one-level parallel reduction in the match kernel, in order to avoid synchronization between threads and unnecessary delays even if it means storing the whole data array. Thus, the memory requirements are larger but the reduction performs faster as the memory accesses are fully coalesced and synchronized. Moreover, our proposal improves the instruction throughput upto 1.45, i.e., the number of instructions that can be executed in a unit of time. Therefore, our proposal achieves 1 Teraflops performance using two GPUs NVIDIA GTX 480 with 480 cores running at 700 MHz. This information is provided in the CUDA profiler available in the respective websites.

Additional information of the paper such as the details of the kernels, the datasets employed, the experimental results and the CUDA profiler information are published in the website:
`http://www.uco.es/grupos/kdis/kdiswiki/SOCOGPU`

## 8 Conclusions and future work

The classification of large datasets using EAs is a time consuming computation as the problem complexity increases. To solve this problem, many studies have aimed at optimizing the computational time of EAs. In recent years, these studies have focused on the use of GPU devices whose main advantage over previous proposals are their massively parallel MIMD execution model that allows researchers to perform parallel computing where million threads can run concurrently using affordable hardware.

In this paper there has been proposed a GPU evaluation model to speed up the evaluation phase of GP classification algorithms. The parallel execution model proposed along with the computational requirements of the evaluation of individuals, creates an ideal execution environment where GPUs are powerful. Experimental results show that our GPU-based proposal greatly reduces the execution time using a massively parallel model that takes advantage of fine-grained and coarse-grained parallelization to achieve a good scalability as the number of GPUs increases. Specifically, its performance is better in high dimensional problems and databases with a large number of patterns where our proposal has achieved a speedup of up to 820X compared to the non-parallel version.

The results obtained are very promising. However, more work can be done in this area. Specifically, the development of hybrid models is interesting from the perspective of evolving in parallel a population of individuals. The classical approach of genetic algorithms is not completely parallelizable because of the serialization of the execution path of certain pieces of code. There have been several proposals to overcome these limitations achieving excellent results. The different models used to perform distributed computing and parallelization approaches focus on two approaches [DM93]: The islands model, where several isolated subpopulations evolve in parallel and periodically swap their best individuals from neighboring islands, and the neighborhood model that evolves a single population and each individual is placed in a cell of a matrix.

These two models are available for use in MIMD parallel architectures in the case of islands and SIMD models for the neighborhood. Therefore, both perspectives can be combined to develop multiple models of parallel and distributed algorithms [HB09], which take advantage of the parallel threads in the GPU, the use of multiple GPUs, and the distribution of computation across multiple machines networked with these GPUs.

**Acknowledgements** This work has been financed in part by the TIN2008-06681-C06-03 project of the Spanish Inter-Ministerial Commission of Science and Technology (CICYT), the P08-TIC-3720 project of the Andalusian Science and Technology Department, and FEDER funds.

## References

[AFSG+09] J. Alcalá-Fdez, L. Sánchez, S. García, M. del Jesus, S. Ventura, J. Garrell, J. Otero, C. Romero, J. Bacardit, V. Rivas, J. Fernández, and F. Herrera. KEEL: A Software Tool to Assess Evolutionary Algorithms for Data Mining Problems. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 13:307–318, 2009.

[BFM97] T. Bäck, D. Fogel, and Z. Michalewicz. *Handbook of evolutionary computation*. Oxford Univ. Press, 1997.

[BLFM04] C. C. Bojarczuk, H. S. Lopes, A. A. Freitas, and E. L. Michalkiewicz. A constrained-syntax genetic programming system for discovering classification rules: application to medical data sets. *Artificial Intelligence in Medicine*, 30(1):27–48, 2004.

[BNKF98] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA, January 1998.

[CM07] D. M. Chitty and Q. Malvern. A data parallel approach to genetic programming using programmable graphics hardware. In *GECCO 07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1566–1573. ACM Press, 2007.

[CUD] NVIDIA Programming and Best Practices Guide, http://www.nvidia.com/cuda, November 2010.

[DDFT04] I. De Falco, A. Della Cioppa, F. Fontanella, and E. Tarantino. An Innovative Approach to Genetic Programming-based Clustering. In *9th Online World Conference on Soft Computing in Industrial Applications*, 2004.

[DDT01] I. De Falco, A. Della Cioppa, and E. Tarantino. Discovering interesting classification rules with genetic programming. *Applied Soft Computing*, 1(4):257–269, 2001.

[Deb05] K. Deb. A population-based algorithm-generator for real-parameter optimization. *Soft Computing*, 9:236–253, 2005.

[DM93] M. Dorigo and V. Maniezzo. Parallel genetic algorithms: Introduction and overview of current research. In *Parallel Genetic Algorithms: Theory and Applications*, pages 5–42. IOS Press, Amsterdam, The Netherlands, 1993.

[EVH10] P. G. Espejo, S. Ventura, and F. Herrera. A Survey on the Application of Genetic Programming to Classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 40(2):121–144, 2010.

[FKB10] M. A. Franco, N. Krasnogor, and J. Bacardit. Speeding up the evaluation of evolutionary learning systems using GPGPUs. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO '10, pages 1039–1046, New York, NY, USA, 2010. ACM.

[Fre02] A. A. Freitas. *Data Mining and Knowledge Discovery with Evolutionary Algorithms*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[GPG] General-Purpose Computation on Graphics Hardware, http://www.gpgpu.org, November 2010.

[Har10] S. Harding. Genetic Programming on Graphics Processing Units Bibliography, http://www.gpgpgpu.com, November 2010.

[HB07] S. Harding and W. Banzhaf. Fast genetic programming and artificial developmental systems on gpus. In *High Performance Computing Systems and Applications, 2007. HPCS 2007*, pages 2–2, 2007.

[HB09] S. Harding and W. Banzhaf. Distributed genetic programming on GPUs using CUDA. In *Workshop on Parallel Architectures and Bioinspired Algorithms*, Raleigh, USA, 2009.

[Koz92] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. The MIT Press, 1992.

[LH08] W. B. Langdon and A. P. Harrison. GP on SPMD parallel graphics hardware for mega bioinformatics data mining. *Soft Computing*, 12(12):1169–1183, 2008.

[LKB06] J. Landry, L. D. Kosta, and T. Bernier. Discriminant feature selection by genetic programming: Towards a domain independent multi-class object detection system. *Journal of Systemics, Cybernetics and Informatics*, 3(1), 2006.

[MBL+09] O. Maitre, L. A. Baumes, N. Lachiche, A. Corma, and P. Collet. Coarse grain parallelization of evolutionary algorithms on gpgpu cards with easea. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, pages 1403–1410, New York, NY, USA, 2009. ACM.

[NA07] D. J. Newman and A. Asuncion. UCI machine learning repository, 2007.

[RMPF09] D. Robilliard, V. Marion-Poty, and C. Fonlupt. Genetic programming on graphics processing units. *Genetic Programming and Evolvable Machines*, 10:447–471, 2009.

[RRB+08] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[SHM+03] T. Schmitz, S. Hohmann, K. Meier, J. Schemmel, and F. Schürmann. Speeding up hardware evolution: a coprocessor for evolutionary algorithms. In *Proceedings of the 5th international conference on Evolvable systems: from biology to hardware*, ICES'03, pages 274–285. Springer-Verlag, 2003.

[TTLH02] K. C. Tan, A. Tay, T. H. Lee, and C. M. Heng. Mining multiple comprehensible classification rules using genetic programming. In *Proceedings of the Evolutionary Computation on 2002. CEC '02. Proceedings of the 2002 Congress*, volume 2 of *CEC '02*, pages 1302–1307, Washington, DC, USA, 2002. IEEE Computer Society.

[VRZ+07] S. Ventura, C. Romero, A. Zafra, J. A. Delgado, and C. Hervás. JCLEC: a Java framework for evolutionary computation. *Soft Computing*, 12:381–392, 2007.

[W. 09] W. W. Hwu. Illinois ECE 498AL: Programming Massively Parallel Processors, Lecture 13: Reductions and their Implementation, http://nanohub.org/resources/7376, 2009.

[WL00] M. Leung Wong and K. Sak Leung. *Data Mining Using Grammar-Based Genetic Programming and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.