International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

Special Issue Paper

Fitness evaluation reuse for accelerating GPU-based evolutionary induction of decision trees

The International Journal of High Performance Computing Applications
2021, Vol. 35(1) 20–32

DOI: 10.1177/1094342020957393
journals.sagepub.com/home/hpc
SAGE

Krzysztof Jurczuk, Marcin Czajkowski and Marek Kretowski

Abstract

Decision trees (DTs) are one of the most popular white-box machine-learning techniques. Traditionally, DTs are induced using a top-down greedy search that may lead to sub-optimal solutions. One of the emerging alternatives is an evolutionary induction inspired by the biological evolution. It searches for the tree structure and tests simultaneously, which results in less complex DTs with at least comparable prediction performance. However, the evolutionary search is computationally expensive, and its effective application to big data mining needs algorithmic and technological progress. In this paper, noting that many trees or their parts reappear during the evolution, we propose a reuse strategy. A fixed number of recently processed individuals (DTs) is stored in a so-called repository. A part of the repository entry (related to fitness calculations) is maintained on a CPU side to limit CPU/GPU memory transactions. The rest of the repository entry (tree structures) is located on a GPU side to speed up searching for similar DTs. As the most time-demanding task of the induction is the DTs' evaluation, the GPU first searches similar DTs in the repository for reuse. If it fails, the GPU has to evaluate DT from the ground up. Large artificial and real-life datasets and various repository strategies are tested. Results show that the concept of reusing information from previous generations can accelerate the original GPU-based solution further. It is especially visible for large-scale data. To give an idea of the overall acceleration scale, the proposed solution can process even billions of objects in a few hours on a single GPU workstation.

Keywords

Big data mining, CUDA, decision trees, evolutionary algorithms, graphics processing unit, parallel computing

1. Introduction

Although the era of big data and data mining (Zhou et al., 2017) have been going on for some time, the problem of adapting many interesting ideas and algorithms to the new reality is still very relevant. Even the most popular data mining techniques such as decision trees (DTs) (Loh, 2014) struggle when it comes to large-scale data despite so many available parallelization techniques. Unfortunately, not all problems are embarrassingly parallel and often there is a strong need to incorporate knowledge about the specificity of a problem to achieve high efficiency and to exploit the full potential of parallelization.

The problem addressed in this paper is no different. Evolutionary induced DTs (Barros et al., 2012) are an important and relatively new alternative to popular greedy solutions that offer trees only with local, sub-optimal tests. Evolutionary algorithms (EA) (Michalewicz, 1996) that are inspired by the biological evolution make possible a global DT induction in which the tests and the tree structure are searched at the same time. As a result, the generated trees are significantly smaller and often more accurate as of the induced by top-down alternatives. However, the main inconvenience of the evolutionary search is its high computational requirements (Barros et al., 2012), making it hard or even impossible to apply it directly to big data.

In this paper, we extend a GPU-based global induction of classification trees (Jurczuk et al., 2017) to accelerate the evolutionary search for the large-scale data. Noting that a lot of trees or their parts reappear during the evolutionary search, we examine if and when it is worth to archive the most popular individuals (DTs) and reuse them. We introduce a concept of a repository of previously evaluated individuals to limit the fitness recalculation of the new ones founded by EA. The search of the same (or similar) DTs is performed fully on the GPU side where they are stored as a

Faculty of Computer Science, Bialystok University of Technology, Bialystok, Poland

Corresponding author:
Krzysztof Jurczuk, Faculty of Computer Science, Bialystok University of Technology, Wiejska 45a, 15-351 Bialystok, Poland.
Email: k.jurczuk@pb.edu.pl

part of the repository. The second part of the repository, which gathers the corresponding fitness results, is located on the CPU side to limit CPU/GPU memory transfers. We consider different levels of DT similarity. If the same tree is archived in the repository, then the fitness-related data is simply associated with the evaluated tree. If similar (partially the same) trees are found, fitness information from the matched trees' parts is reused and the GPU is called to fill the missing information. Otherwise, the GPU has to evaluate the tree (all nodes) from the ground up. The reuse strategy is not new in EA, however, it was studied in the context of improving the genetic diversity (Acan and Tekol, 2003) or encouraging chromosome revisits by replacing by the most similar (and already evaluated) solution (Charalampakis, 2012). According to our knowledge, it has not been applied to evolutionary data mining and especially DT induction.

Our research was performed on a Global Decision Tree (GDT) system (Kretowski, 2019) which allows evolving different types of classification and regression trees and may be applied in real-life applications (Czajkowski and Kretowski, 2019). In the GPU-supported version of the GDT (Jurczuk et al., 2017), the main evolutionary loop (selection, genetic operators, etc.) is performed sequentially on a CPU, while the most time-consuming operations like fitness calculation are delegated to a GPU. The proposed reuse strategy extends this approach by introducing the repository of DTs to decrease the induction time as well as to observe the population behaviour. Our preliminary studies on the fitness evaluation reuse are described in the conference paper (Jurczuk et al., 2019). In this paper, we improve the reuse strategy and perform its thorough analysis. In particular, we:

- consider more levels of DT similarity, starting from the identical tree, then, a root and its left or right subtree and so on;
- propose various strategies of updating the repository, FIFO and modified FIFO;
- evaluate thoroughly the improvement on both real-life and artificially generated datasets;
- analyse performance across various GPUs cards (with Keppler-, Maxwell- and Pascal-based architectures), different repository settings, dataset size/dimension.

This paper is organized as follows. Section 2 provides a brief background on DTs, the GDT system, and the related works. Section 3 describes in detail the proposed fitness evaluation reuse for accelerating GPU-based evolutionary induction of decision trees. Section 4 presents the experimental validation of our approach on artificial and real-life datasets. In the last section, the paper is concluded and possible future works are outlined.

2. Background

2.1. Decision trees

Decision trees (DTs) (Loh, 2014) play a key role as effective non-parametric machine learning techniques for classification and regression problems. They are especially useful as 'white box' approaches because their prediction models can be easily visualized and interpreted. From a structural point of view, DT is a directed acyclic graph. It starts at a single node called the root which may have at least two outgoing edges (called branches) that lead to other nodes. If a node has an outgoing edge, then we call it an internal node, otherwise, it is called a terminal node (or a leaf). To find a prognosis that is a class label or a value, the DT makes a sequence of hierarchical tests (splits).

We can distinguish many types of decision trees and their division may depend on:

- the type of a problem to which they are applied. If the target feature is discrete, we deal with classification trees. Each leaf has assigned a class label which is the majority class of training instances that reach the leaf. Regression trees are induced when a target is continuous;
- the type of splits in the internal nodes. Most of DTs use inequality tests and splits with a single feature. Such trees are often called univariate ones as they partition the feature space with axis-parallel decision borders. Multivariate decision trees use multiple features tests. The test often has a form of an oblique split which is based on a linear combination of features.
- the way the DTs are induced. Hunt's algorithm is one of the earliest approaches of building DT (Rokach and Maimon, 2005) and it serves as a basis for the most popular solutions like C4.5 (Quinlan, 1993) or CART (Breiman et al., 1984). It uses a top-down approach where a DT is created recursively until stopping criteria are met (e.g., a node contains only the training instances from the same class). Each split in the internal node uses a local search and makes a decision according to the given optimality measure. One of emerging alternatives to such a greedy approach is a global DT induction which may be realized by using various meta-heuristics such as evolutionary algorithms (Barros et al., 2012).

Despite over 50 years of research (Loh, 2014), there are still problems that DTs need to be faced to, especially in the context of new challenges like big data mining. The rest of the background focuses on evolutionary induced univariate classification trees and their speedup in the context of large-scale data analysis.

2.2. Evolutionary induction of decision trees

Evolutionary algorithms (EA)s are the meta-search heuristics that mimic the process of the biological evolution (Michalewicz, 1996). Like in nature, the individuals, which represents a candidate solution to the target problem, constitute a population which evolves and adapts to the environment. The individuals can be represented in various ways, e.g. as an encoded fixed-length linear string (GA approach) or by tree-encoding schema (GP approach). In

general, the initial population should be created at random, but for efficiency reasons, some greedy heuristics are often applied. However, it is important to preserve a balance between exploitation and exploration.

In each evolutionary iteration, individuals are evaluated with the fitness function, which measures their performance, and ones with higher quality are more likely to be selected for reproduction. At least two objectives need to be minimized in the context of DTs: prediction error and the number of nodes. The role of the second objective is essential as it mitigates the over-fitting problem due to overgrown trees. In greedy top-down inducers, this issue is partially addressed by a stopping criterion and additional post-pruning (Esposito et al., 1997).

Genetic operators such as crossover (also called recombination) and mutation produce new offspring which replace individuals (parents) in the next generation. The recombination mechanism is inspired by the sexual reproduction of living organisms. It allows combining the information from two individuals and creates two novel solutions with mixed genotypes of their parents. The mutation operator causes small random changes of the selected individual. The process is iteratively repeated until some stopping criteria are met and, finally, a single DT with the best fitness is returned.

The main drawback of classical top-down inducers is a greedy search that may lead to sub-optimal solutions and over-fitting the training data. One of the remedies for this problem is the ensemble of trees (Breiman, 2001), however, the comprehensibility of analysing a single DT is lost. Evolutionary DT induction process allows not only performing a robust global search but also to preserve the simplicity of the prediction model (Barros et al., 2012). The strength of the evolutionary approach lies in the ability to escape from local optima as the searches for the overall tree structure as well as for the splits in the internal nodes are performed simultaneously. As a result, EAs tend to induce smaller and often more accurate trees than greedy solutions. Another advantage of evolutionary induced DT is the ability to perform multi-objective optimization (Czajkowski and Kretowski, 2019), which is often crucial in many real-life problems.

2.3. Global decision tree system

The Global Decision Tree (GDT) system (Kretowski, 2019) is capable of inducing various DTs with the evolutionary approach. The GDT system follows a typical EA schema and here its description is limited to the univariate classification trees to improve the clarity of description and avoid less important details. Individuals in the population are not encoded and processed in their actual form. They are initialized with a semi-random top-down strategy that is applied to a small fraction of the training data. In each non-terminal node, a random test that separates two instances allocated in the considered node but from different classes is selected (mixed dipole strategy) (Kretowski, 2019). The mixed dipole is a pair of objects belonging to the different classes. The first object can be chosen completely random. The second one is chosen from the remaining objects of other classes.

The GDT system offers several specialized variants of mutation and crossover operators that impact not only the tests in the internal nodes but overall tree structure, e.g.:

- change internal node into the leaf (acts like pruning procedure);
- transform leaf into an internal node by using a new test created on a randomly chosen dipole;
- modify the test in the internal node (shift threshold) or replace it with a new one on a different attribute;
- exchange subtrees/branches/tests between two affected individuals (variants of crossover operator – see Figure 1).

The selection of an affected node depends on its location (modification of nodes from the upper levels results in more global changes so it is performed less frequently) and its quality (less accurate nodes are more likely to be modified). The application probability of each operator and its variants may also vary.

Univariate classification version of GDT uses a weighted form of the fitness function which maximizes accuracy ($A(T)$) estimated on the training dataset and minimizes the size ($S(T)$) of the tree T calculated as the number of nodes:

$$F(T) = A(T) - \alpha * (S(T) - 1.0), \tag{1}$$

where α is a user-defined parameter that reflects the relative importance of the complexity term.

The linear ranking selection (Michalewicz, 1996) is used as the selection mechanism together with the elitist strategy that assures that the best individual founded so far will appear in the next population. GDT terminates evolution when it reaches the maximum number of generation or there is no improvement of the fittest individual by a fixed number of iterations.

2.4. Related works

The main drawback of EAs in DT induction is related to high time requirements, which is particularly visible nowadays in the context of big data analysis. Fortunately, the advances in parallel processing have enabled EAs to work in much a shorter time or/and to process much larger datasets (Bacardit and Llorà, 2013; Cano, 2018). We can distinguish three main strategies for EA parallelization/distribution (Chitty, 2012):

- *master − slave* paradigm that parallelizes the most time-consuming operation in each evolutionary iteration, the master spreads tasks or data over the slaves and, finally, it gathers and merges the results;

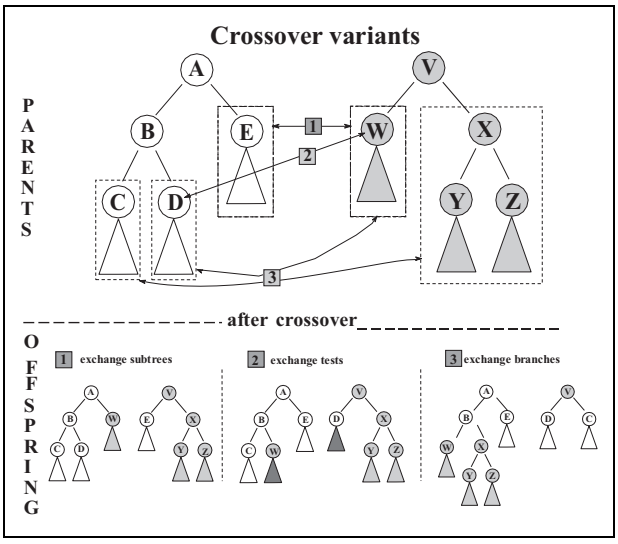

Figure 1. Visualization of crossover variants: (1) exchange subtrees, (2) exchange tests, (3) exchange branches.

- *island* (coarse-grained) model that evolves independently sub-populations distributed between islands;
- *cellular* (fine-grained) algorithm which defines neighbourhood topology that limits the communication (selection and reproduction) between redistributed individuals to the nearest ones.

As the EAs work on a population of independent individuals, the computational load may be distributed among multiple processors through a population decomposition approach (also known as a control approach). Despite its undoubted simplicity, this solution is poorly scalable when analysing large-scale data. An alternative is a data decomposition approach that gradually distributes the dataset among the processors. This is a much more scalable approach since the chunks of the dataset are evaluated in parallel.

The parallelization of various evolutionary computation methods is well-studied (Alba and Tomassini, 2002; Cano, 2018; Tsutsui and Collet, 2013). In recent years, there has been a strong interest in using GPUs as an implementation platform (Franco and Bacardit, 2016; Maitre et al., 2012). However, it seems that the problem of speeding up the evolutionary DT induction is still not adequately explored. To the best of the authors' knowledge, only the GDT system was investigated in the context of MPI/OpenMP (Czajkowski et al., 2015), Spark (Reska et al., 2018) and GPU (Jurczuk et al., 2017) parallelizations. Remaining studies cover either greedy inducers (Lo et al., 2014; Strnad and Nerat, 2016) either random forests (Grahn et al., 2011; Marron et al., 2014) which are beyond the scope and interest of the research presented in this paper.

The level of a speedup, as well as the size of datasets analysed by a parallel version of GDT, strongly depend on the selected framework. CUDA-based acceleration was clearly the fastest with acceleration up to 800×, however, it is limited by the size of the GPU memory (Jurczuk et al., 2018). This is not a problem to the Spark-based solution, which may be slower but can be easily scaled up and processes datasets with billions of objects. The relatively worst results were achieved for the GDT system with the MPI+OpenMP approach in both the speedup and the dataset sizes. All aforementioned methods used the idea of isolating the most time-consuming operations (fitness evaluation, dipoles searching) and run them parallelly, while the rest of EA steps are performed sequentially. This way, the results of the original GDT algorithm and the parallelized one are consistent.

In this work, we investigate acceleration of evolutionary DT induction using an external repository to store the bunch of the previously evaluated trees and reduce redundant fitness computations. Moreover, the GPU-support and CUDA (Storti and Yurtoglu, 2016) programming model are applied to limit the computational repository overhead. The basic idea may resemble the external memory implementations in other evolutionary computation methods such as ant colony (Acan, 2004) and particle swarm optimizations (Acan and Unveren, 2009). The types of information stored in external memory, as well as the ways of using such information, may vary. Some solutions store the unused individuals to be used in further populations and to trace some of the untested search directions (Acan and Tekol, 2003). Others try to increase diversity by storing all the solutions and using them for a non-revisiting strategy (Yuen and Chow, 2009). We are particularly interested in reducing the number of fitness evaluations by reusing past information if a new individual is similar to ones from the repository. A similar idea was explored for genetic algorithms where the author examined the concept of storing all evaluated chromosomes for the future reuse (Charalampakis, 2012). However, the idea of the solution was to encourage chromosome revisiting by replacing by the most similar (and already evaluated) ones from the registry.

3. Repository-supported GPU-based approach

The accelerated version of the GDT is built on the GPU-based predecessor (Jurczuk et al., 2017). As it is illustrated in Figure 2, the core evolution is still run sequentially on a CPU. The most time-consuming operations (evaluation of individuals, dipoles searching, . . .) are isolated and delegated to be performed parallel on a GPU. This way, the parallelization does not affect the behaviour of the original EA.

The algorithm begins by creating an initial population. It is performed only once and on small fractions of the dataset. As a result, it takes an insignificant part of the overall time (less than 1% of the total execution time) and it is not parallelized. Next, the fixed-size repository is created. Adequate amounts of memory are allocated both on the GPU and CPU sides. Initially, the repository is empty. Each element of the repository (an individual) is archived partially on the GPU side and partially on the CPU side. On the

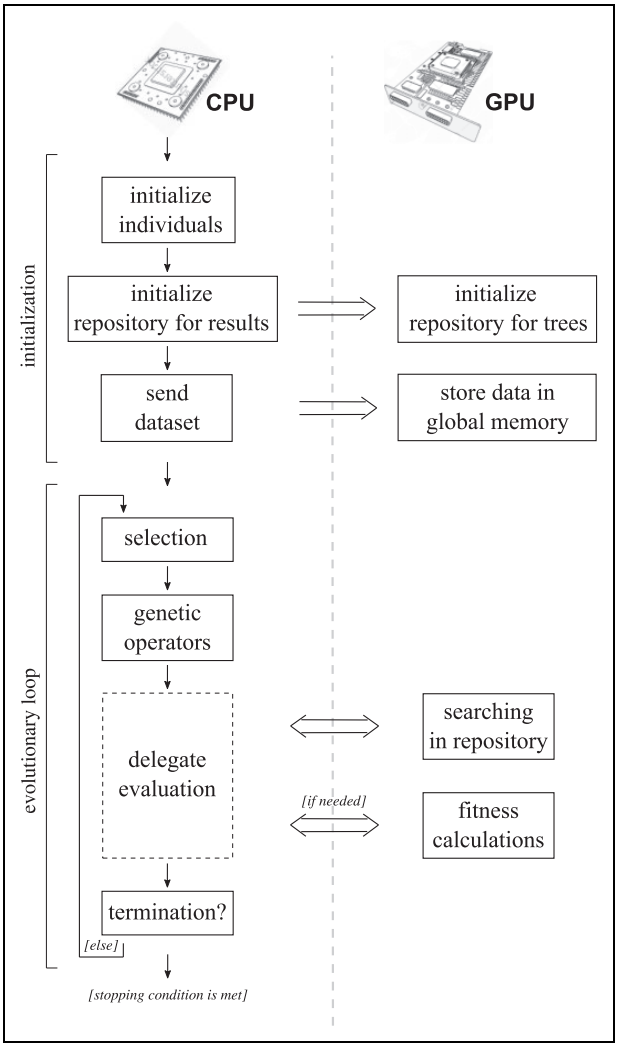

Figure 2. The general flowchart of the repository-supported algorithm for evolutionary induction of decision trees.

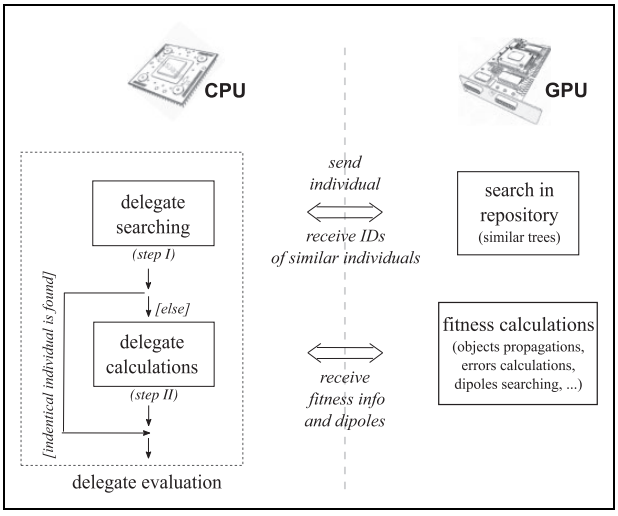

Figure 3. Boosted evaluation of individuals using a GPU. First, similar trees are searched in the repository. If an identical tree is found, there is no need to ask the GPU to perform fitness-related calculations. Otherwise, the CPU delegates the calculation to the GPU. If only similar trees (partially the same) are found, the GPU fills the missing information. If no similarity is found, all fitness-related calculations have to be performed by the GPU.

GPU side, the repository stores the structures of DTs (tree nodes, branches, tests, etc.). Corresponding fitness results and dipoles are stored on the CPU side. Such a repository design allows us to: *(i)* avoid unnecessary CPU/GPU memory transfers and *(ii)* reduce the overhead when searching for similar DTs.

Before the evolutionary loop starts, the whole training dataset is sent to the GPU. This CPU-GPU transfer is done only once and the dataset is kept in the GPU global memory till the DT induction stops. This way, all threads have constant access to all objects.

In each iteration, when a genetic operator (crossover/mutation) is successfully applied, there is a need to evaluate a new individual and then the GPU is called. This is the most time-demanding part of the algorithm since all training objects need to be passed through the tree, starting from the root to an appropriate leaf. Both the dataset size and tree size influence the computational requirements. For large datasets, it takes more than 99% of the total algorithm execution time. Concerning the selection process, it is run sequentially since it takes negligible time.

Figure 3 shows more details of the GPU-accelerated evaluation of newly created individuals. First, the tree is sent to the GPU. Then, the GPU searches in the repository for similar individuals (step I). If such individuals are found, the GPU returns their positions in the repository and the levels of similarity. The CPU uses this information to accelerate the fitness calculations. If an identical tree is found, then the CPU only needs to copy all previously calculated statistics and dipoles from the repository (CPU-CPU transfer). Otherwise, two cases are possible: no similar tree is found or some parts of the evaluated tree are found in the repository (we call such trees as similar trees and the identical trees' parts as matched parts). In both cases, the GPU has to be asked again to fill the missing information needed to finish the evaluation (see Figure 3, step II).

Searching in the repository is an overhead thus the algorithm tries to find as large matching parts of the tree as possible (see Figure 4). The priority is to find the same tree (similarity at level 0). Otherwise, two types of similarity are preferable (similarity at level 1). The first case concerns the situation when the tests in the tree root and in its left subtree are the same (as in a tree from the repository). The second case is analogical but it refers to the right subtree of the root. If any of these similarities is not applicable, then we go at a lower level of the tree and so on (similarity at level 2, 3, etc.).

If no similarity is found, all fitness-related calculations have to be performed (as in the algorithm without the repository-support). In the case of similar (only partially the same) trees, the partial fitness information is copied from

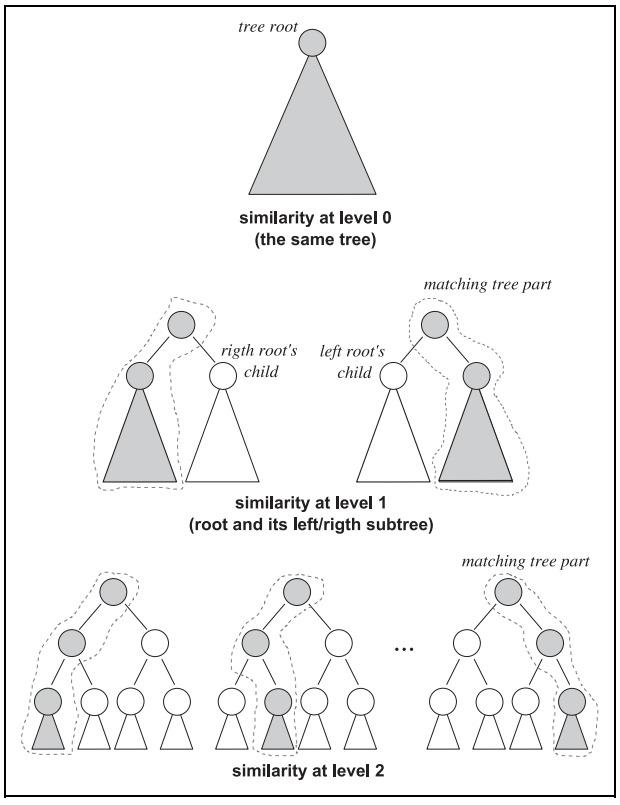

Figure 4. Similarity patterns in the trees' repository: starting from an identical tree down to the successive subtrees. The possible matching parts of a tree at different levels of similarity are wrapped by dashed lines.

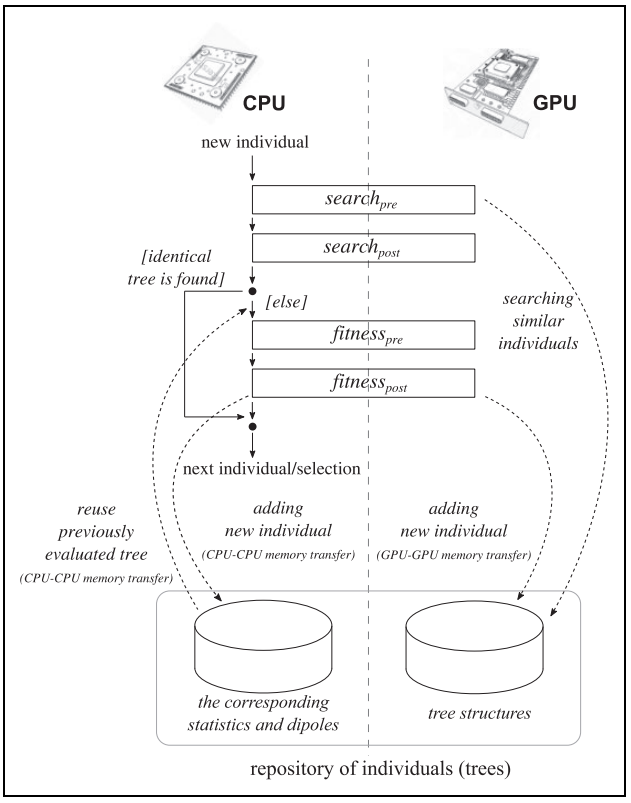

Figure 5. The arrangement of GPU kernels used to evaluate individuals as well as memory transfers when using the repository.

the repository to the evaluated tree. Partial fitness information are the class distribution, errors and dipoles in the tree nodes of the matched trees' parts. To complete the fitness calculations, the GPU is then used to fill the missing information in the unmatched parts of the tree. For all training objects that fall into these parts of the tree, appropriate leaves are searched. The objects are propagated from the tree root towards these parts as well as inside them. In the leaves, the number of objects of each class is accumulated (so-called class distribution) and also some random objects are stored (for dipoles). Based on the class distribution, reclassification errors in the leaves are calculated. Finally, all gathered information: class distribution, errors and random objects, are propagated from the leaves towards the root node and then they are sent back to the CPU. The CPU uses them to update the considered individual and finally calculate its fitness value. The following sections describe in greater detail the fitness reuse strategy as well as the GPU-based evaluation of individuals.

3.1. Searching similar trees

Two kernel functions are used to search for similar trees in the repository (see Figure 5). The first kernel ($search_{pre}$) compares the evaluated individual with all trees from the repository. Two-level data decomposition approach (Grama et al., 2003) is applied. Individuals from the repository are divided between GPU blocks. Inside the blocks, threads are responsible for comparing different parts of the trees. In the simplest case, the tree root is assigned to the first thread, its left and right child to the second and third one, respectively, and so on. Thus, the comparisons with different individuals from the repository as well as at various tree nodes are done in parallel. When a difference is found, the search inside a block is not finished because more similar trees can still be found. However, the found difference is marked in a boolean variable which resides in shared memory. The number of such boolean variables depends on the number of different similarity patterns that are concerned. When synchronization between threads is needed, atomic exchange operations are used.

On the GPU side, trees are represented as one-dimensional arrays where the index of the left and right child of the i-th node equals $(2i + 1)$ and $(2i + 2)$, respectively (Jurczuk et al., 2017). This makes the comparison between individuals easier. Searching a difference consists in checking corresponding elements of two arrays (a tree from the repository and the evaluated individual). Both the attribute and the threshold of the tests in the tree nodes have to be verified. To know which nodes (indices of arrays) have to be checked for different similarity patterns, one-dimensional binary masks are created before the evolution. Figure 6 shows sample patterns and the corresponding masks.

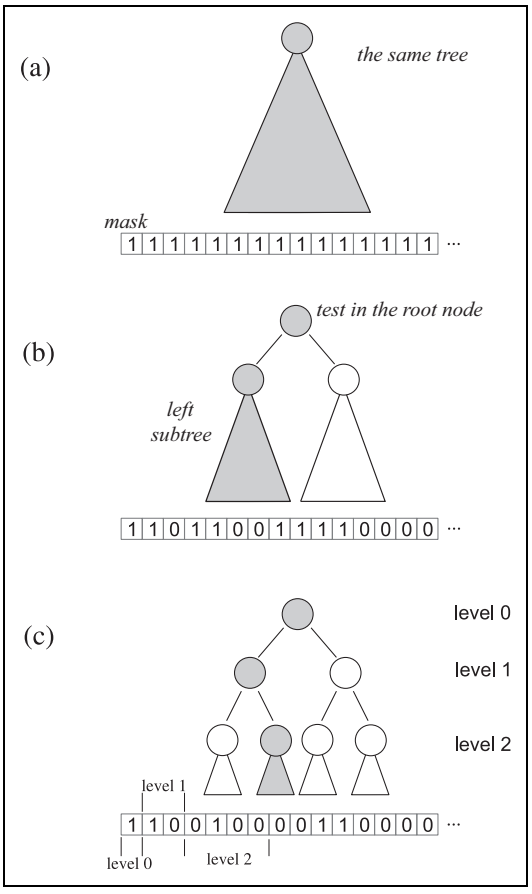

Figure 6. Sample similarity patterns and corresponding masks for: (a) identical tree, (b) the same tests in the tree root and in its left subtree, (c) the same root test, the same tests in its left subtree and in the second subsubtree.

Finally, the first kernel verifies the level of similarity of every tree in the repository. The second kernel ($search_{post}$) reduces the results returned by the first one. It uses only one block. Each GPU thread is responsible for scanning the similarity levels of a bunch of individuals. For each of the similarity patterns, maximum one individual is provided and its identifier is sent back to the CPU. If no individual is found, negative identifier is returned. If more than one individual from the repository fit the same pattern, then the last one that is verified is retained. Synchronization between block threads is provided by atomic exchange operations.

We decided to use the Structure-of-Arrays (SoA) layout which is usually preferable from GPU performance perspective (Mei and Tian, 2016; Strzodka, 2012). In our case, multi-value data of individuals are stored in separated arrays (see Listing 1). An alternative is the Array-of-Structures (AoS) layout that may lead to coalescing issue (Jurczuk, Kretowski and Bezy–Wendling, 2018). The SoA layout generally provides full use of memory bandwidth as well as global memory accesses are always coalesced (Wilt, 2013). One thread may copy data (e.g., node information) to cache for other threads what can decrease the number of memory transactions and, thus, minimize DRAM bandwidth.

Listing 1. AoS vs SoA memory layouts for an invidual.

```
1
2 //M - number of tree nodes
3
4 //Array of Structs (AoS)
5 struct Node {
6       int attributeNumber;
7       float threshold;
8 }
9 Node aosTree[M];
10 //set values for the test in the tree root
11 aosTree[0].attributeNumber = 4;
12 aosTree[0].threshold = 10.1;
13
14 //Structure of Arrays (SoA)
15 struct Tree{
16       int attributeNumber[M];
17       float threshold[M];
18 }
19 Tree soaTree;
20 //set values for the test of the left root child
21 soaTree.attributeNumber[1] = 4;
22 soaTree.threshold[1] = 10.1;
```

3.2. Maintaining the repository of individuals

The repository of individuals is a kind of an external memory (archive) that is used to store a fixed number of previously considered individuals, next to the population. At the beginning of the evolution, the repository is empty. It is filled each time when a new tree appears unless identical tree is already stored (in other words, it is found in the repository).

The fixed size of the repository causes that a replacement of individuals is required when it is full. We consider two strategies. In the first one, a new individual is inserted in the place of the oldest tree – First In First Out (FIFO) algorithm. The advantage of such a mechanism is little overhead because we do not need to search a place for a newly inserted individual (an index is only required to indicate the replacement point).

In the second strategy, an additional criterion is introduced: how often a tree was already reused. For this purpose, each tree in the repository has a reuse counter. When a tree is inserted into the repository, its counter equals 0. Each time when the tree is reused, its counter is incremented. Searching for a place of a new tree still uses the FIFO algorithm but its prioritized version. The oldest tree with a reuse counter equal to 0 is replaced. If the location in the repository has the reuse counter not equal to 0, then its counter value is decremented and the next position is taken into account, and so on. Such a mechanism favours the oldest individuals and, at the same time, rarely used, as candidates to delete from the repository. Nevertheless, the additional criterion causes memory and time overhead that should be unnoticeable when larger datasets are processed.

In order to limit CPU/GPU memory transfers, the repository is divided between CPU and GPU (see

Table 1. Basic specification (processing and memory resources) of three NVIDIA graphics cards used in the experiments.

	Engine		Memory		
NVIDIA graphics card	No. CUDA cores	Clock rate [MHz]	Size [GB]	Bandwidth [GB/s]	Compute capability
Geforce GTX 780	2 304	863	3	288.4	3.5
Geforce GTX Titan X	3 072	1 000	12	336.5	5.2
Tesla P100	3 584	1 328	12	549.0	6.0

Figure 5). The CPU archives the results while the GPU keeps the structures of trees. Thanks to this, most of the repository operations (searching, inserting, reuse) need only CPU-CPU or GPU-GPU memory transfers. Each time a new tree is inserted, both parts of the repository are updated. On the GPU side, the tree structures are copied to the repository (GPU-GPU memory transfer). As regards the CPU side, the corresponding tree statistics and the dipoles are added to the repository (CPU-CPU memory transfer). When an identical or similar tree is found, the tree statistics and the dipoles are copied from the repository to the evaluated tree (CPU-CPU memory transfer).

3.3. GPU-supported evaluation

If no similarity is found, the GPU has to calculate the information for all nodes of the tree. When similar (partially the same) trees are found, the GPU is used to complete the missing information. In both cases, two kernels are used (Jurczuk et al., 2017). The first kernel ($fitness_{pre}$) (see Figure 5) is called to propagate all objects in the training dataset from the tree root to appropriate leaves. The data decomposition approach is applied. The whole dataset is spread into smaller parts over GPU blocks. Next, in each block, the assigned objects are further spread over the threads. In each GPU block, a copy of the evaluated individual is created and loaded into the shared memory space. This way, all the threads process the same individual in parallel but handle different chunks of the dataset.

If a training object reaches a leaf, the counter of its class in this leaf is incremented. When $fitness_{pre}$ kernel finishes, in each tree leaf, the number of objects of each class that reach that particular leaf is stored. Moreover, randomly chosen objects of each class are saved in each tree leaf. However, the results are spread over the GPU blocks. The second kernel function ($fitness_{post}$) reduces the results. The counters of objects from copies of the individual are summed up, and the total number of objects of each class in each leaf is obtained (class distribution) and reclassification errors are calculated. Then, all gathered information (class distribution, errors) as well as objects for dipoles are propagated from the leaves towards the root. Finally, they are sent back to the CPU to finish the tree evaluation and the repository is updated.

Table 2. Characteristics of the datasets: Name, number of instances, number of attributes and number of classes.

Dataset	No. Inst.	No. Attr.	No. Class.
Chess	100 000	2	2
	1 000 000	2	2
	10 000 000	2	2
	100 000 000	2	2
	1 000 000 000	2	2
Suzy	5 000 000	18	2
Higgs	11 000 000	28	2

4. Experimental validation

4.1. Setup

All experiments were performed on a server equipped with 2 eight-core processors Intel Xeon E5-2620 v4 (20 MB Cache, 2.10 GHz), 256 GB RAM and a single graphics card. We tested three different NVIDIA GPUs described in Table 1 where we gather basic specifications that cover the number of CUDA cores, a clock rate, available memory, bandwidth and compute capability. The first GPU card is the consumer line GeForce GPU based on Kepler architecture launched in 2013 but it still quite powerful. The second GPU card is based on Maxwell architecture. The last one is the professional-level GPU accelerator (based on Pascal architecture) that currently costs about $ 5 000 (almost 10 times more than the first one).

The server was running 64-bit Ubuntu Linux 16.04.6 LTS. The algorithm was implemented in C++, the GPU-supported parts in CUDA-C and compiled by nvcc CUDA 10.0 (NVIDIA, 2019) (single-precision arithmetic was applied). The sequential and OpenMP versions (Czajkowski et al., 2015) were compiled by gcc version 5.4.0.

Experimental analysis was performed on artificially generated and real-life datasets described in Table 2. Concerning real-life datasets, two large datasets from the UCI Machine Learning Repository (MLR) (Dua and Karra Taniskidou, 2019) were tested:

- *Higgs* – concerns a classification problem to distinguish between a signal process that produces Higgs bosons and a background process that does not;
- *Suzy* – covers the problem of distinguishing between a signal process that produces super-symmetric particles and a background process that does not.

The artificially generated dataset called *chess3x3* was analysed. This dataset represents a classification problem with two classes, two continuous-valued attributes and objects arranged on a 3×3 chessboard (Kretowski, 2019). We used the synthetic dataset to scale it freely, unlike real-life datasets. We examined a various number of training objects (from hundreds of thousands to a billion).

All presented results correspond to averages of 5–10 runs and were obtained with a default set of parameters from the original GDT system, which are briefly listed in Table 3. For in-depth description and settings of additional parameters like probabilities of different mutation/crossover variants and GPU configurations, please refer to (Jurczuk et al., 2017; Kretowski, 2019). As we are focused in this paper only on time performance, results for the classification accuracy are not included, as they are not changed.

4.2. Results

Table 4 presents the mean execution times of the repository-supported GDT as well as its previous versions (GPU-based, OpenMP-based and the sequential one). The strongest GPU card (Tesla P100) as well as the optimal repository settings (size, similarity levels, ...) were applied. We see that the fitness reuse strategy accelerates the solution further. As it can be expected, the time decrease is more important when large-scale data is processed. For the smallest *Chess* dataset size (100 000 objects), the reuse strategy only slightly accelerates the solution (about 3%). What is important that this improvement was obtained with the simpler repository mechanism (Jurczuk et al., 2019). For the extended strategy (modified FIFO and more levels of similarity), the overall time was not improved. However, for the rest of *Chess* dataset sizes, the extended version gave clearly better results.

For 1 billion objects, the proposed solution decreases the induction time by ≈9 h (it is about 60% faster). We estimated that the sequential GDT version would need over a year to process such a huge dataset and the OpenMP parallelization would decrease this time to a few months.

Concerning real-life datasets (see Table 4), the reuse strategy also accelerates the induction. For the *Suzy* dataset, the improvement is about 22% and it was obtained using the simpler reuse mechanism (Jurczuk et al., 2019). For the extended one, the calculation times were comparable with the version without the repository support. The detailed profiling (not included) showed that the generated trees and dataset were not big enough to compensate the repository overhead. In the case of the *Higgs* dataset, the improvement is about 24% and it was obtained with the extended reuse strategy. The obtained accelerations are lower than for the *Chess* dataset with a comparable number of objects. The probable reason is the higher number of attributes that increases the search space. As a result, it is harder to find similar trees (as it will be discussed and shown later in Figure 9). For both datasets, the induction time decreases from days/hours to a few minutes compared to the CPU implementations.

Table 5 shows the mean induction times for three tested GPU cards. The results suggest that even a cheap GPU card is enough to accelerate the induction of DTs significantly. As it is expected, stronger GPUs manage to achieve much better acceleration. The cheapest tested GPU card (GeForce GTX 780) in comparison to the most expensive

Table 3. Default parameters of GDT.

Parameter	Value
Population size	64 individuals
Crossover rate	20% assigned to the tree
Mutation rate	80% assigned to the tree
Elitism rate	1 individual per generation
Max generations	1 000
Block/Thread numbers	256×256

Table 5. Mean execution times of the repository-supported version on three different NVIDIA GPU cards, for artificial and real-life datasets (in seconds) (* – not enough memory).

Dataset		GTX 780	GTX Titan X	Tesla P100
Chess	100 000	57.8	27.7	21.1
Chess	1 000 000	116.1	56.9	49.4
Chess	10 000 000	774.7	659.4	439.5
Chess	100 000 000	7 177.4	7 015.4	5 047.9
Chess	1 000 000 000	*	75 343.2	52 691.5
Suzy		346.2	306.1	218.2
Higgs		787.3	695.4	498.1

Table 4. Comparison of the mean execution times of the repository-supported version (GPU with REPO) with the previous GPU-based one (using NVIDIA Tesla P100 GPU card). Executions times of the OpenMP and sequential versions are also included (in seconds, for larger datasets also time in hours).

Dataset		GPU with REPO		GPU		OpenMP	Sequential
Chess	100 000	21.1		21.8		100.2	685
Chess	1 000 000	49.4		62.7		3 605.7	23 536
Chess	10 000 000	439.5		666.7		47 600.4	324 000
Chess	100 000 000	5 047.9	(1.4 h)	7 471.3	(2 h)	weeks	months
Chess	1 000 000 000	52 691.5	(14.5 h)	84 245.2	(23.5 h)	months	over a year
Suzy		218.2		266.6		7 h	2 days
Higgs		498.1		632.5		18 h	4 days

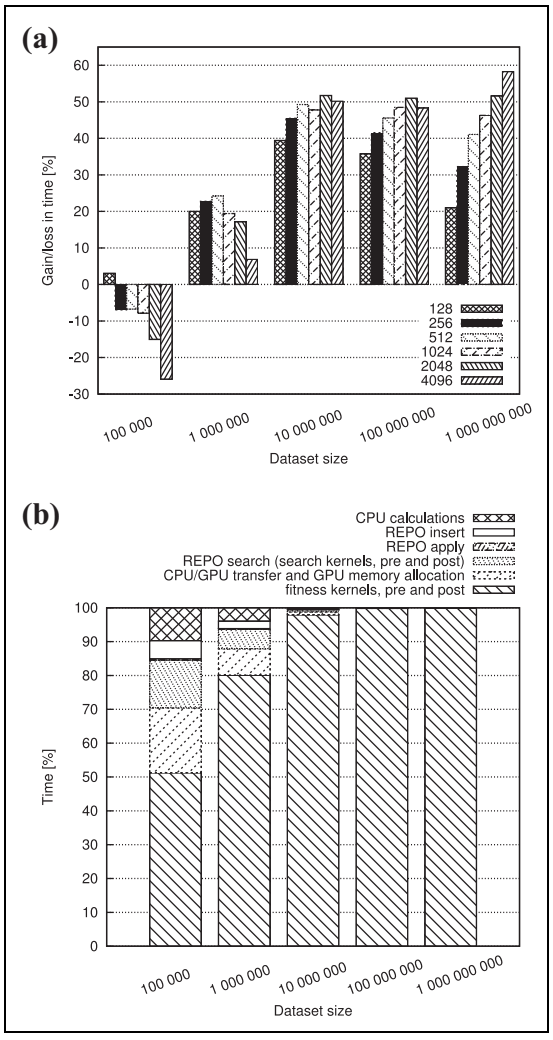

Figure 7. The influence of the reuse strategy on evolution time for the increasing *Chess* dataset size: (a) gain or loss in evolution time for the different repository size (from 128 to 4096 trees), (b) detailed time-sharing information (mean time as a percentage) for the repository size of 256 individuals.

one (Tesla P100) is about 1.5 times slower. We would expect a more prominent difference if double-precision arithmetic operations were applied. It should be noted that the memory size of the GeForce GTX 780 card is not big enough to store the biggest dataset.

Concerning the speedup over the corresponding sequential version of the algorithm, we clearly see that it is very satisfactory for all tested GPUs and datasets (see Tables 4 and 5). In the case of Tesla P100 for Chess 1 000 000, Suzy and Higgs datasets it is ≈ 476, ≈ 396 and ≈ 694, respectively.

Figure 7(a) shows the influence of the repository size on the evolution time when the size of dataset increases (using Tesla P100). The bars represent the gain or loss in time (in percentage) in comparison to the GPU-based GDT without repository support. It is clearly visible that the gain is more prominent when the dataset size increases. For more than 10 millions of objects, the repository-supported version is at least 50% faster than the original GPU-based version. The optimal size of the repository grows when larger data is processed (for 100 000 objects 128 trees, for 1 000 000 objects 256 trees, etc.). For the smallest dataset (100 000 objects), the induction time can even increase if too many trees (more than 256) are archived in the repository.

The observed slowdown of the induction may be explained by the repository overhead that is illustrated in Figure 7(b). We see that the time efficiency drop is mainly blamed by searching in the repository (REPO search). The time spent on other repository operations is substantially smaller (REPO insert – inserting new individuals) than the searching or negligible (REPO apply – reusing stored data). For the smallest dataset (100 000 objects), the repository search overhead is not compensated by the time profit of fitness reuse. When the size of the dataset grows, the evaluation of individuals is more time consuming and the repository searching overhead becomes negligible.

For better understanding, we have also studied the searching success ratio, i.e. how often the search in the

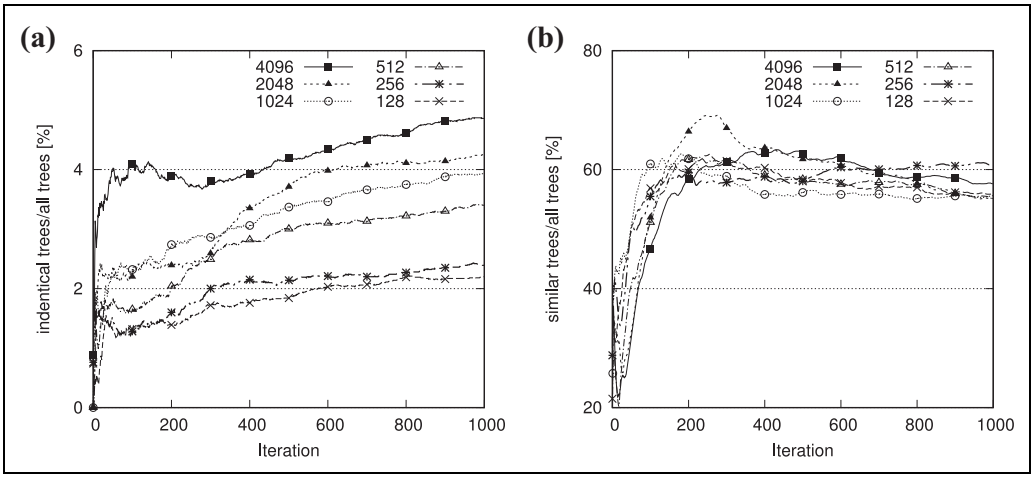

Figure 8. Success ratio of searching in the repository for the *Chess* dataset (10 000 000 objects). The repository size increases from 128 to 4096 trees. On the left, for identical trees. On the right, when similar trees are found.

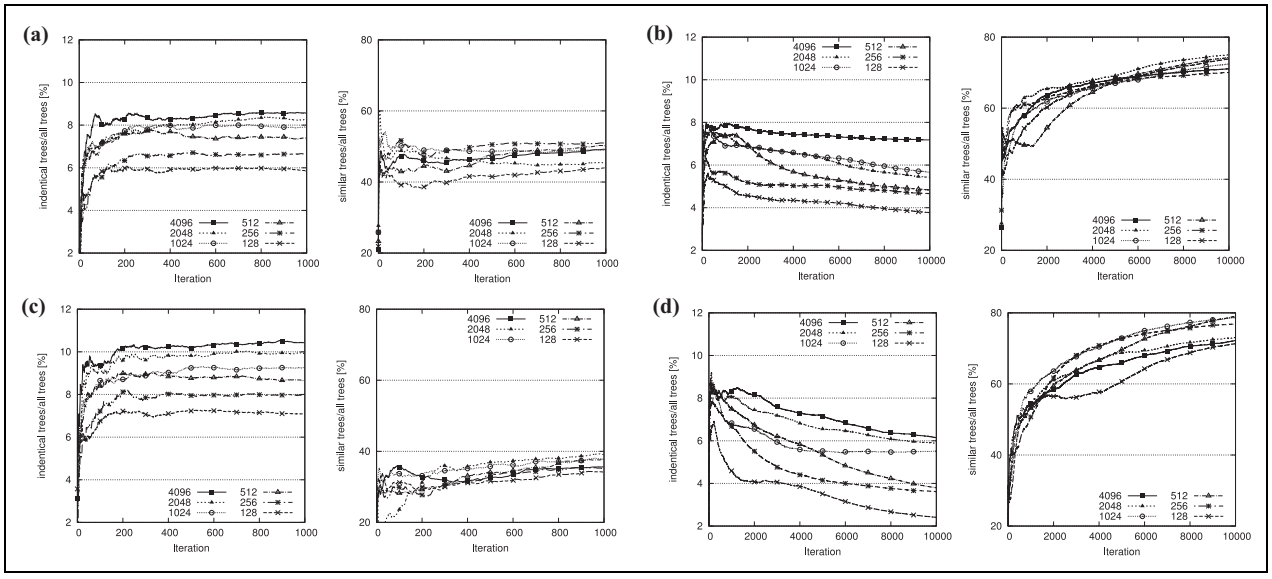

Figure 9. Success ratio of searching in the repository for the *Suzy* and *Higgs* datasets. The repository size increases from 128 to 4096 trees. On the left, for identical trees. On the right, when similar trees are found. (a) *Suzy*, $\alpha = 0.001$, (b) *Suzy*, $\alpha = 0.0001$, (c) *Higgs*, $\alpha = 0.001$, (d) *Higgs*, $\alpha = 0.0001$.

repository ends with a success. Concerning the *Chess* dataset, Figure 8 shows that an identical tree was found in a few percent of cases, from 1% to 5%. This ratio increases when the repository size grows. However, we cannot forget that, in this case, the repository overhead also grows what may diminish the benefits of fitness reuse, especially in the case of smaller datasets. Much better success ratio is obtained for similar trees (about 60%) and the influence of dataset size is not so obvious. It was about 50% in the preliminary repository-supported version (Jurczuk et al., 2019).

Figure 9 gives success ratio results for the real-life datasets. We have examined two cases: *(i)* $\alpha = 0.001$ (default value, as in other experiments), *(ii)* $\alpha = 0.0001$ (for inducing bigger trees). We see that identical trees were found more often than for the *Chess* dataset (about $\times 2$). It can be attributed to the size of the generated trees which were smaller than the ones for the *Chess* dataset. On the other hand, it is also visible that the number of similar trees is about two times smaller (completely inverse pattern). It can be probably explained by more attributes as well as not symmetrical DTs. When α equals 0.0001, the size of the generated DTs grows and the success ratios change. There are much more similar trees, while the number of identical trees decreases slightly. More detailed analysis of the influence of the attribute number as well as tree structures is left for future investigations.

5. Conclusion and future work

Big data mining brings new challenges, especially in the context of high computation demanding solutions. In this paper, we tackle the problem of speeding up the GPU-based evolutionary induction of DTs on large-scale data. With an additional repository, we store some of previously evaluated individuals (the most popular) and try to reuse them (or their parts) in order to skip the fitness evaluation phase which is the most consuming task of the evolutionary loop. To limit the computational overhead of the reuse strategy, GPU-support and CUDA programming model are applied. Extensive experimental validation shows that the proposed strategy successfully accelerate tree induction. The improvement is particularly noticeable with the large-scale data.

There are many promising directions that we currently are working on. The first one considers a multi-GPU approach to scale the solution even further. The repository could be spread over GPUs that will search in parallel. Using OpenCL (instead of CUDA) may provide portability to GPUs of other vendors as well as further time improvement (McIntosh-Smith et al., 2015). On the other side, this research is also the first step in applying 'multi-tree' representation. It assumes that similar individuals can be represented by partially sharing fragments/structures in memory. Such a solution may allow us to observe and better understand the evolutionary induction dynamics in detail, e.g. to follow diversity at each level of a decision tree. It would also be an interesting challenge to formalize and generalize the proposed approach (in a similar way as in Fakhi et al. (2017)), concerning both the fitness evaluation and reuse strategy. Then, the model could be easily applied in other population-based machine learning tools.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of

this article: This work was supported by Bialystok University of Technology under the Grant WZ/WI-IIT/3/2020 founded by Ministry of Science and Higher Education.

ORCID iD

Krzysztof Jurczuk https://orcid.org/0000-0001-6469-1769

References

Acan A (2004) An external memory implementation in ant colony optimization. In: Marco D, Birattari M, Blum C, et al. (eds), *Ant Colony Optimization and Swarm Intelligence*. Berlin: Springer, pp. 73–82.

Acan A and Tekol Y (2003) Chromosome reuse in genetic algorithms. In: Cantú-Paz E, Foster JA, Deb K, et al. (eds), *Genetic and Evolutionary Computation – GECCO 2003*. Berlin: Springer, pp. 695–705.

Acan A and Unveren A (2009) A memory-based colonization scheme for particle swarm optimization. In: *2009 IEEE Congress on Evolutionary Computation*, Trondheim, Norway, 18–21 May 2009, pp. 1965–1972.

Alba E and Tomassini M (2002) Parallelism and evolutionary algorithms. *IEEE Transactions on Evolutionary Computation* 6(5): 443–462.

Bacardit J and Llorà X (2013) Large-scale data mining using genetics-based machine learning. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 3(1): 37–61.

Barros RC, Basgalupp MP, De Carvalho AC, et al. (2012) A survey of evolutionary algorithms for decision-tree induction. *IEEE Transactions on SMC, Part C* 42(3): 291–312.

Breiman L (2001) Random forests. *Machine Learning* 45(1): 5–32.

Breiman L, Friedman JH, Olshen RA, et al. (1984) *Classification and Regression Trees*. Wadsworth Publishing.

Cano A (2018) A survey on graphic processing unit computing for large-scale data mining. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8(1): e1232.

Charalampakis AE (2012) Registrar: a complete-memory operator to enhance performance of genetic algorithms. *Journal of Global Optimization* 54: 449–483.

Chitty DM (2012) Fast parallel genetic programming: multi-core CPU versus many-core GPU. *Soft Computing* 16(10): 1795–1814.

Czajkowski M, Jurczuk K and Kretowski M (2015) A parallel approach for evolutionary induced decision trees. MPI+OpenMP implementation. In: Rutkowski L, Korytkowski M, Scherer R, et al. (eds), *Artificial Intelligence and Soft Computing*, *LNCS*, vol. 9119. Berlin: Springer, pp. 340–349.

Czajkowski M and Kretowski M (2019) Decision tree underfitting in mining of gene expression data. An evolutionary multi-test tree approach. *Expert Systems with Applications* 137: 392–404.

Dua D and Graff C (2019) *UCI Machine Learning Repository*. Irvine, CA: University of California, School of Information and Computer Science. Available at: http://archive.ics.uci.edu/ml.

Esposito F, Malerba D and Semeraro G (1997) A comparative analysis of methods for pruning decision trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 19(5): 476–491.

Fakhi H, Bouattane O, Youssfi M, et al. (2017) A multi-agent model for general-purpose computing on graphics processing units. *Multiagent and Grid Systems* 13(3): 237–252.

Franco MA and Bacardit J (2016) Large-scale experimental evaluation of GPU strategies for evolutionary machine learning. *Information Sciences* 330: 385–402.

Grahn H, Lavesson N, Lapajne MH, et al. (2011) CudaRF: a CUDA-based implementation of random forests. In: *2011 9th IEEE/ACS international conference on computer systems and applications (AICCSA)*, Sharm El-Sheikh, Egypt, 27–30 December 2011, pp. 95–101.

Grama A, Karypis G, Kumar V, et al. (2003) *Introduction to Parallel Computing*. Boston: Addison-Wesley.

Jurczuk K, Czajkowski M and Kretowski M (2017) Evolutionary induction of a decision tree for large-scale data: a GPU-based approach. *Soft Computing* 21(24): 7363–7379.

Jurczuk K, Czajkowski M and Kretowski M (2019) Accelerating GPU-based evolutionary induction of decision trees – fitness evaluation reuse. In: Wyrzykowski R, Dongarra J, Paprzycki M, et al. (eds), *Parallel processing and applied mathematics, PPAM'19*, *LNCS*, vol. 12043. Berlin: Springer, pp. 421–431.

Jurczuk K, Kretowski M and Bezy–Wendling J (2018) GPU-based computational modeling of magnetic resonance imaging of vascular structures. *The International Journal of High Performance Computing Applications* 32(4): 496–511.

Jurczuk K, Reska D and Kretowski M (2018) What are the limits of evolutionary induction of decision trees? In: Auger A, Fonseca CM, Lourenço N, et al. (eds), *Parallel problem solving from nature – PPSN XV*, *LNCS*. Berlin: Springer, pp. 461–473.

Kretowski M (2019) *Evolutionary Decision Trees in Large-Scale Data Mining*. Berlin: Springer.

Lo WT, Chang YS, Sheu RK, et al. (2014) CUDT: a CUDA based decision tree algorithm. *Scientific World Journal* 2014: 745640.

Loh WY (2014) Fifty years of classification and regression trees. *International Statistical Review* 82(3): 329–348.

Maitre O, Kruger F, Querry S, et al. (2012) EASEA: specification and execution of evolutionary algorithms on GPGPU. *Soft Computing* 16(2): 261–279.

Marron D, Bifet A and Morales GDF (2014) Random forests of very fast decision trees on GPU for mining evolving big data streams. In: *21st European Conference on Artificial Conference (ECAI'14)*, Prague, Czech Republic, 18–22 August 2014, pp. 615–620. Amsterdam: IOS Press.

McIntosh-Smith S, Price J, Sessions RB, et al. (2015) High performance in silico virtual drug screening on many-core processors. *The International Journal of High Performance Computing Applications* 29(2): 119–134.

Mei G and Tian H (2016) Impact of data layouts on the efficiency of GPU-accelerated IDW interpolation. *SpringerPlus* 5(104): 1–18.

Michalewicz Z (1996) *Genetic Algorithms + Data Structures = Evolution Programs*. Berlin: Springer.

NVIDIA (2020) NVIDIA Developer Zone - CUDA Toolkit Documentation. Available at: https://docs.nvidia.com/cuda/.

Quinlan JR (1993) *C4.5: Programs for Machine Learning*. Burlington: Morgan Kaufmann.

Reska D, Jurczuk K and Kretowski M (2018) Evolutionary induction of classification trees on spark. In: Rutkowski L, Rafa S, Korytkowski M, et al. (eds), *Artificial Intelligence and Soft Computing*, *LNCS*, vol. 10841. Berlin: Springer, pp. 514–523.

Rokach L and Maimon O (2005) Top-down induction of decision trees classifiers – a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews* 35(4): 476–487.

Storti D and Yurtoglu M (2016) *CUDA for Engineers: An Introduction to High-Performance Parallel Computing*. New York: Addison-Wesley.

Strnad D and Nerat A (2016) Parallel construction of classification trees on a GPU. *Concurrency and Computation: Practice and Experience* 28(5): 1417–1436.

Strzodka R (2012) Abstraction for AoS and SoA layout in C++. In: Hwu WW (ed), *GPU Computing Gems Jade Edition*. Burlington: Morgan Kaufmann, pp. 429–441.

Tsutsui S and Collet P (eds) (2013) *Massively Parallel Evolutionary Computation on GPGPUs*. Natural Computing Series. Berlin: Springer.

Wilt N (2013) *CUDA Handbook: A Comprehensive Guide to GPU Programming*. Upper Saddle River, NJ: Addison-Wesley.

Yuen SY and Chow CK (2009) A genetic algorithm that adaptively mutates and never revisits. *IEEE Transactions on Evolutionary Computation* 13(2): 454–472.

Zhou L, Pan S, Wang J, et al. (2017) Machine learning on big data: opportunities and challenges. *Neurocomputing* 237: 350–361.

Author biographies

Krzysztof Jurczuk received the joined PhD in 2013 from the University of Rennes 1 (France) and the Faculty of Computer Science, Bialystok University of Technology (Poland). He is currently Assistant Professor at the Faculty of Computer Science, Bialystok University of Technology (Poland). His research interests focus on biomedical informatics (CFD, MRI simulations), parallel computing, and data mining.

Marcin Czajkowski received the Masters degree (2007) and the PhD degree with honours (2015) from Computer Science, Bialystok University of Technology (Poland). He is currently Assistant Professor at the Faculty of Computer Science, Bialystok University of Technology (Poland). His research activity mainly concerns bioinformatics, machine learning and data mining, in particular, classification and regression trees and evolutionary algorithms.

Marek Kretowski received the joined PhD in 2002 from the University of Rennes 1 (France) and the Faculty of Computer Science, Bialystok University of Technology (Poland). He is currently Professor at the Faculty of Computer Science, Bialystok University of Technology (Poland). His research interests focus on biomedical applications of computer science (modelling for image understanding, image analysis), bioinformatics and data mining.