Contents lists available at ScienceDirect

Applied Soft Computing

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

GPU-based acceleration of evolutionary induction of model trees

Krzysztof Jurczuk [*], Marcin Czajkowski, Marek Kretowski

Faculty of Computer Science, Bialystok University of Technology, Wiejska 45a, 15-351, Bialystok, Poland

ARTICLE INFO

Article history:
Received 9 February 2021
Received in revised form 17 November 2021
Accepted 19 January 2022
Available online 29 January 2022

Keywords:
Evolutionary data mining
Decision trees
Regression
GPU parallel computing
Large-scale data

ABSTRACT

Evolutionary algorithms (EAs) are naturally prone to parallel processing. However, when they are applied to data mining, the fitness calculations start to dominate and the typical population-based decomposition limits the parallel efficiency. When dealing with large-scale data, the scalable solution may become a real challenge. In this article, we propose a GPU-based parallelization of evolutionary induction of model trees. Such trees are a special case of decision tree (DT) that is designed to solve regression problems. The evolutionary approach allows not only a robust prediction but also to preserve the simplicity of DTs. However, the global approach is much more computationally demanding than state-of-the-art greedy inducers, and thus hard to apply to large-scale data mining directly. A parallelized induction of model trees (with univariate tests in the internal nodes and multiple linear regression models in the leaves) requires a carefully designed decomposition strategy. Six GPU-supported procedures are designed to successively: redistribute, sort and rearrange dataset samples, next, calculate models and fitness, and finally gather the results. Experimental validation is performed on real-life and artificial datasets, using various (low- and high-end) GPU accelerators. Results show that the GPU-supported solution enables time-efficient global induction of model trees on large-scale data, which until now was reserved for greedy methods. The obtained speedup is very satisfactory (even up to hundreds of times). The solution is scalable for datasets of different sizes and dimensions.

1. Introduction

Big data mining [1], which has become very popular in business, industry and science, offers great opportunities but also brings great challenges. Traditional data mining tools and algorithms must either evolve and adapt to handle large-scale data or they will be marginalized. For many solutions, some kind of parallelism is the last resort in order to continue to be relevant in the world of big data.

Among the various tools and algorithms that can effectively identify patterns within data, decision trees (DTs) [2] are one of the most commonly used machine learning techniques. Despite more than 50 years of research on DTs, there are still many open issues [3], such as finding locally/globally optimal splits in internal nodes, right pruning criterion, efficient analysis of cost-sensitive data and multi-objective optimization. There are different ways of induction of DTs but commonly they are built by greedy methods, e.g., in a top-down manner by a recursive-partitioning strategy. Such an induction is usually fast; however, it can offer trees only with local, sub-optimal tests [4]. One of the alternative approaches is the use of evolutionary algorithms (EAs) in tree induction. The strength of the evolutionary induction is global exploration where tree structure, splits in internal nodes and predictions in leaves are searched simultaneously [5]. As a result, the generated trees are much simpler and at least as accurate as the greedy alternatives. Evolutionary induced DTs are less prone to overfitting, instability to changes in training data and attribute-selection bias.

The incorporation of EAs into the DT induction allows for efficient solution-space exploration, leading to better solutions than those induced with traditional methods. However, at the same time, it brings new challenges. Direct application of evolutionary DT induction to big data may be hard or even unachievable [5–7]. Population-based and iterative calculations may simply be too demanding. The motivation of this paper is to show that this barrier can be overcome by smart parallel processing in such a multi-disciplinary approach (Fig. 1). We propose a GPU-supported solution (called cuGMT) and show experimentally that now the global approach can be applied to large-scale data mining.

The acceleration of global DT induction has so far been discussed mainly in the context of classification problems [8,9]. This paper focuses on model trees and proposes to use a graphics processing unit (GPU) to boost their induction and deal with large-scale data. Model trees are one of variants of DTs designed to solve regression problems. Model trees can be seen as an extension of the typical regression tree where the constant value in each leaf is replaced by a linear (or nonlinear) regression function.

[*] Corresponding author.
E-mail address: k.jurczuk@pb.edu.pl (K. Jurczuk).

https://doi.org/10.1016/j.asoc.2022.108503
1568-4946/© 2022 Elsevier B.V. All rights reserved.

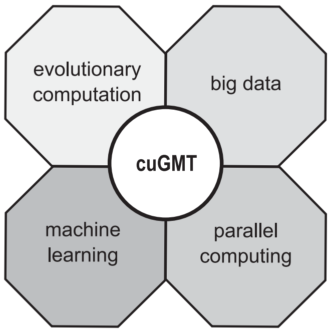

Fig. 1. The proposed cuGMT solution combines different areas. Evolutionary computation is incorporated into a machine learning technique in order to search globally optimal DTs. To be able to apply it to big data mining, smart parallel processing is used. It is powered by a GPU that processes datasets as well as evaluates solutions in subsequent generations.

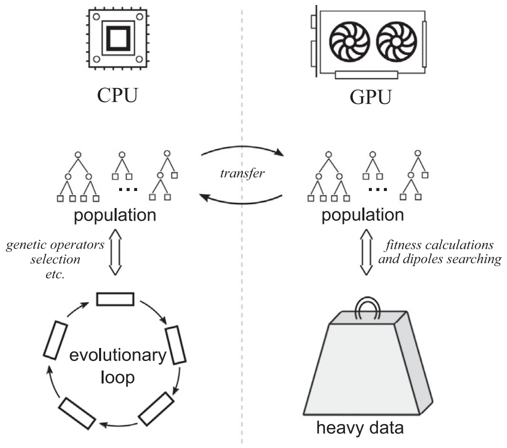
CPU
GPU
transfer
population
population
genetic operators selection etc.
fitness calculations and dipoles searching
evolutionary loop
heavy data

Fig. 2. A sketch of the proposed GPU-accelerated solution for big data mining. On the CPU side, the evolutionary flow control remains. On the GPU side, calculations involving training data are performed. The heavy (training) data is kept on the GPU side, to reduce the bottleneck of CPU/GPU memory transfers.

In contrast to the regression trees, whose leaves and induction may resemble the classification approach, the model trees stand a completely new challenge. They couple tree-based representation with multiple linear regression models in leaves, which (along with the evolutionary approach) requires a new, dedicated parallel algorithm. A straightforward parallelization of models' calculations (the most time-consuming phase) is to enough to provide induction time improvement.

We decided to use GPUs as they provide massive parallel resources and fast memory as well as energy and economic efficiency [10]. The NVIDIA CUDA framework [11] is applied. It supports general-purpose computation on GPUs (GPGPU). In the proposed parallelization, the GPU cores handle computing-intensive jobs, such as samples' redistribution, model generation and fitness calculation. The evolutionary flow control is left to the CPU (see Fig. 2). We decided to keep the training dataset on the GPU side and send it once before the evolution. This was a conscious design decision to reduce the bottleneck of CPU/GPU memory transfers. This forced us to organize most of the dataset-related operations on the GPU side (not only directly related with the regression model calculations but also with searching optimal splits) as well as to design additional functions that were not needed for the CPU implementations. Only the necessary samples and information about updated DT parts are sent to the CPU. The GPU side calculations are divided into six parallelized procedures that boost the induction of model trees. An obtained acceleration is very satisfactory. It concerns the datasets of different sizes and dimensions. To the best of our knowledge, this is the first study on the GPU-based parallelization of global model tree induction. An alternative attempt can be found in the literature [12]; however it was a cluster computing acceleration based on a hybrid MPI+OpenMP approach.

The proposed solution has been deployed in a system called Global Model Tree (GMT) [13,14]. However, the solution is itself independent of any framework. The GMT can be used for evolutionary induction of different types of regression and model trees and was tested in real-life applications, e.g., [15,16]. The main objectives of this work are to accelerate the global induction of model trees and enable efficient evolutionary induction on large-scale data. This way, GMT can be applied to a broader range of problems, and evolutionary induction of DTs could become more competitive in terms of computation time to state-of-the-art greedy inducers.

The next section provides a brief background on DTs, the GPGPU computing model, and most recent related works. Section 3 describes our approach to parallel implementation of evolutionary model tree induction. Section 4 presents the details of experimental setup and algorithm parameters. Further, the results of the proposed solution on real-life and artificially generated datasets are presented and discussed. In the last section, the paper is concluded and possible future work is outlined.

2. Background

This section provides a general background on decision (model) trees and their induction. Distributed/parallel approaches in evolutionary induction and the GPGPU computational model are also presented. In particular, we focus on GPU-supported accelerations of DT inducers.

2.1. Decision trees

Decision trees (DTs) represent one of the main knowledge discovery methods [2]. The hierarchical tree structure, in which appropriate tests in consecutive nodes are sequentially applied, closely resembles a human way of decision-making. The success of tree-based approaches can be explained by their ease of application, fast operation and efficacy. Nevertheless, the run-time of algorithms and memory resources needed to generate DTs still require improvement to meet ever-growing computational demands, especially in the context of big data mining.

In the literature, we may find different variants of DTs [3]. They can be grouped according to the type of:

- the problem they are applied to — classification or regression [17];
- the way they are induced — greedy top-down/bottom-up induction [18], global induction based on evolutionary approach [4,5] or swarm intelligence optimization [19];
- the tree structure — different types of internal nodes (univariate, multivariate, mixed) and leaves (class label, constant value, regression plane: single or multiple linear regression) [14].

In this paper, we target evolutionary induced univariate trees that can be applied to regression problems.

2.1.1. Model trees

Decision trees have a knowledge representation structure. They are built of nodes and branches [3], where each internal node is associated with a test on one or more attributes (features), each branch represents the outcome of a test, and each leaf (terminal node) is designed by a prediction. Most tree-inducing

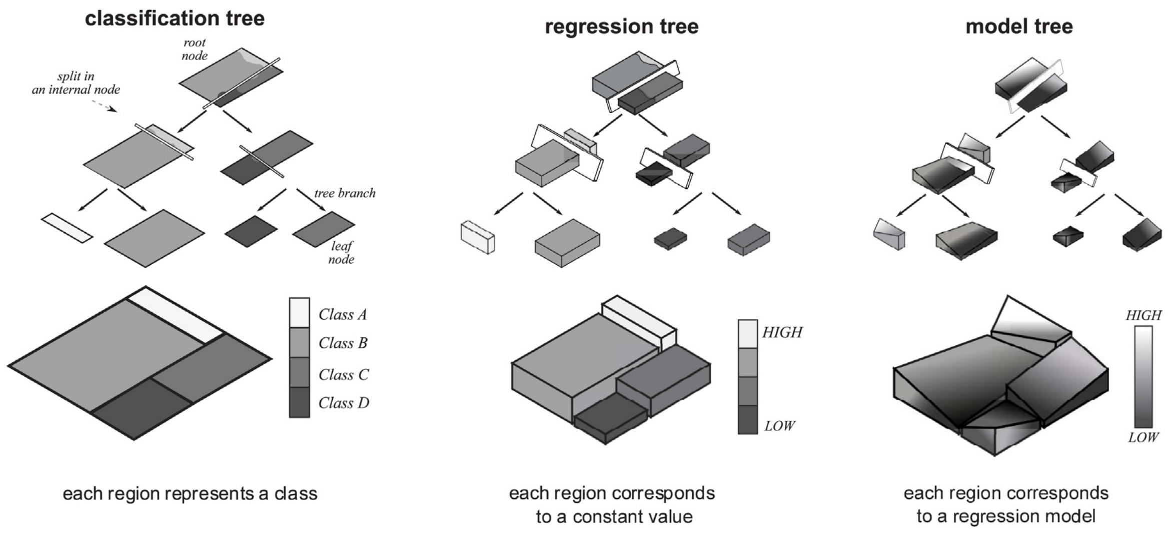
classification tree
root node
split in an internal node
tree branch
leaf node
Class A
Class B
Class C
Class D
each region represents a class
regression tree
HIGH
LOW
each region corresponds to a constant value
model tree
HIGH
LOW
each region corresponds to a regression model

Fig. 3. An illustration of various tree types. At the top, the prediction space is partitioned on successive levels of the tree (starting from the root node). At the bottom, the divided prediction surfaces (corresponding to each type of decision tree) are shown. The black lines with arrows are the tree branches.

algorithms partition the attribute space with axis-parallel hyperplanes. Trees of this type are often called univariate because a test in each non-terminal node usually involves a single attribute.

In classification trees, a class label is assigned to each leaf. Usually, it is the majority class of all training samples (instances/objects) that reach that particular leaf. Regression trees may be considered a variant of DTs designed to approximate real-valued functions. In the case of the simplest regression tree, each leaf contains a constant value, usually an average value of the target attribute. A model tree can be seen as an extension of the typical regression tree [20]. The constant value in each leaf is replaced in the model tree by a linear (or nonlinear) regression function. To predict the target value, the new tested sample is followed down the tree from a root node to a leaf using its attribute values making routing decisions at each internal node. Next, the prediction for the new sample is evaluated based on a regression model in the leaf.

Fig. 3 illustrates an example of classification, regression and model trees as well as their predicted values. The gray level of each region represents a different class label (for a classification tree), while the height corresponds to the value of the prediction function (regression and model trees). Although regression trees are not as popular as classification trees, they are highly competitive with other machine learning algorithms [21] and are often applied to real-life problems [22,23].

2.1.2. Induction of decision trees

Inducing an optimal DT is known as NP-complete [24]. Consequently, practical decision-tree learning systems are based on heuristics such as greedy algorithms where locally best splits are made in each node (a test is selected according to a given goodness of split). The most popular tree induction method is based on the top-down approach [25]. It starts from the root node, where the best split is searched. Next, the training samples are redirected to the newly created subnodes and this process is repeated for each subnode until some stopping-rule is satisfied. Successive subnodes process progressively less samples but, each time, the search is performed over all attributes. Thus, the computational complexity is generally concentrated on the calculation of criterion function used to find locally optimal splits. In addition, post-pruning [26] is usually used after induction to avoid the overfitting to the training data, and improve the generalizing power of the predictive model.

One of the most popular representatives of top-down induced univariate regression and model trees are CART [27] and M5 [20] systems, respectively. The CART system minimizes the sum of squared residuals to find locally optimal tests in the internal node. The M5 algorithm uses a similar splitting strategy, but instead of the mean value from the training samples, it applies a more advanced approach in the terminal nodes. Each leaf in the M5 system contains a multiple linear model which prediction for a particular sample is later on additionally smoothed.

Top-down DT inducers are generally fast but they can trap in local optima and are prone to overfitting [4,19]. Thus, other approaches, like global inducers using various metaheuristics, have been developed to overcome these problems. Evolutionary computation techniques are proven to be effective at escaping local optima and are able to successfully solve a general class of difficult optimization problems [28]. The evolutionary (global) approach to DT induction was initially investigated in a genetic programming (GP) [29] community. One of the first attempts was made by Koza [30], where the author presented GP-method for evolving LISP S-expressions corresponding to simple DTs. EA candidate solutions were represented by the tree structures, instead of the fixed-sized linear chromosome representation. The idea of evolving computer programs by GP has been further developed over the years [31]. However, direct usage of GP to induction of DTs is not an easy task because GP individuals are usually generated by combining limited sets of terminal symbols and functors, which it turns out is not enough for competitive and robust induction. On the other hand, genetic algorithms are rather related with linear chromosomes representation. Thus, EA-based systems have been generally developed for global induction of DTs, in particular for model trees [5,19].

EA-based induction of DTs is able to simultaneously search for the tree structure, tests in internal nodes and regression models in the leaves (for the model trees). Such a global approach can reveal hidden patterns that are often undetectable by greedy methods but it is, of course, much more computationally complex. The dominant operation is the fitness function calculation. Each fitness calculation requires to pass all samples in the training dataset through the tree starting from the root node to an appropriate leaf. The value of fitness function has to be calculated in each iteration for all modified individuals. In addition, the type of DTs can increase the computational requirements. The number of individuals is usually constant (in our case, dozens), while

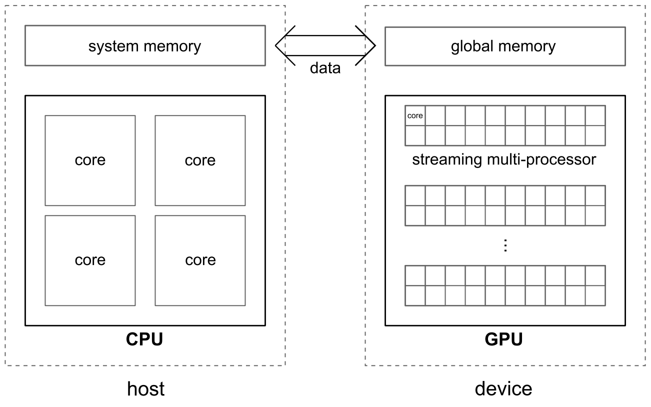

Fig. 4. Schematic overview of a typical GPU (vs. CPU) architecture.

the number of iterations depends on many factors and can be really huge (e.g., from tens of thousands to millions) when the problem is "complicated". Popular representatives of EA-based regression trees are TARGET [32] (evolving a CART-like regression tree with basic genetic operators) and E-Motion [6] (globally inducing model trees that apply a standard 1-point crossover and two mutation strategies — shrinking and expanding).

Currently, alternatives to the greedy top-down approaches include primarily EA-based ones [4,5,33,34]. However, there are also solutions that use other population-based metaheuristics for DT induction, especially based on swarm intelligence (SI) [19]. Different from EAs, it is inspired by the collective behavior of decentralized, simple agents (e.g., animals as ants or bees) which can communicate with each other. The agents follow very simple rules but as a swarm can lead to the emergence of very complicated global behavior (far beyond the capability of a single agent). So far, three SI optimization approaches have been applied for DT induction: ant colony optimization [35], particle swarm optimization [36] and bat algorithms [37].

2.2. GPU computing using CUDA

Modern graphics cards are equipped with a specialized processor (GPU) and high-speed (hierarchical) memory [10]. The GPU is provided with hundreds or even thousands of simple, energy-efficient computing units (GPU cores). Each GPU core is much smaller and slower than a CPU core, but it is especially tuned to be very efficient at basic mathematical operations. High bandwidth memory, coupled with many computational units, creates GPUs that are ideal parallel computing devices. The GPUs often provide unmatched price/performance and energy/performance factors than other parallel hardware. Moreover, they enable scale-up on a single workstation. Thus, GPU accelerators are currently used in general-purpose computations, e.g., in engineering and scientific computing [38].

The computational potential of GPUs was difficult for researchers and IT staff to utilize without a dedicated development environment. The introduction of NVIDIA Compute Unified Device Architecture (CUDA) [11] has revolutionized GPGPU. This is a programming interface and parallel platform that exposes developers to high-level use of GPU resources (without the need for graphics primitives). There are some alternatives (like OpenCL, OpenACC), but CUDA is the most widespread platform. In CUDA, a CPU works together with a GPU (see Fig. 4). The main program is run on the CPU that works as a coordinator. The GPU is a co-processor that carries out a narrower range of more specialized tasks, e.g., complex mathematical computations. Usually, a part of the CPU's tasks is delegated to the GPU to be processed by thousands of threads in parallel, concurrently to the CPU operations.

From the CUDA programmer's perspective [11], some parts of a CPU code are replaced by kernels. In brief, a kernel is a function run on the GPU. When a kernel is called, many threads are created to perform the delegated task faster. The threads are grouped into a grid of thread blocks. A grid is a set of blocks, while a block is a set of concurrent threads. Threads within a block can cooperate using synchronization barriers and shared memory that is not accessible by the threads of other blocks. The number of blocks in a grid and threads in a block must be specified when calling a kernel. The GPU cores are arranged in an array of multithreaded streaming multi-processors (SMs). Each block of threads can be executed on one SM and cannot migrate once assigned. All threads run the same code. The thread's ID allows an assigned part of the data to be processed and/or to make control decisions.

2.3. Related work

Many computational intelligence methods have recently applied the GPGPU [39–41]. As regards evolutionary data mining, the GPUs are especially desired to speed up the time-demanding fitness calculations that are challenging when applied to big data [9,42,43].

2.3.1. Parallelization in EA

Population-based algorithms are naturally prone to parallelism, and the artificial evolution can be implemented in various ways [44]. There are at least three basic parallel approaches to EAs on CPU architectures: master–slave model, island model and cellular one. The EAs operate on a set of independent solutions. Thus, it is relatively easy to distribute the computational load among multiple processors. One of the traditional decomposition techniques is a population approach (also known as a control approach) [44] in which individuals from the population are evaluated parallelly on different processors. The main drawback of this solution is a large population requirement in order to retain good scalability. Besides, distributed-memory systems may have problems with high inter-processor data traffic. In contrast, shared-memory systems can suffer from an insufficient number of available processors and memory access contention [45].

Recent research on parallelization of evolutionary data mining methods has seemed to focus on GPUs as the acceleration platform [46]. The parallel evaluation of samples (data decomposition) is considered much more scalable with respect to the size of a dataset than the population approach. It involves a gradual distribution of the entire dataset among the local memories of processors. However, despite the decrease of the communication overhead, some issues with high inter-processor data traffic can still remain, e.g., during the reduction of results [9]. Some algorithms parallelized EAs with GPGPU using both decomposition techniques [42]. Additional dimensions of parallelization were also studied [40].

The GPGPU has also been successfully used with other strategies for EAs' parallelization. In the island model, the group of individuals in sub-populations distributed between islands were evolved in parallel [47]. The coarse-grained strategy was applied in an evolutionary learning system called BioHEL [48], where the authors proposed a two-dimensional parallelization that processes all rules and samples in the training dataset in parallel. The GPGPU can also be applied to the cellular algorithm which redistributes individuals that can communicate only with the nearest individuals for selection and reproduction based on the defined neighborhood topology. In [49], the authors proposed a control approach parallelization technique, and another study [50] tested the three-dimensional decomposition for the fine-grained parallelization strategy.

2.3.2. Parallelization of DT induction

Evolutionary induction of DTs is computationally complex and thus time-demanding. The strong need for its parallelization has been evident for many years [4,5,8]. However, the topic has not yet been explored enough. There are still open issues, like different variants of DTs, which may require dedicated algorithms to handle big data.

In the literature, several GPU-supported DT inducers can be found. Some of them involve greedy (top-down) inducers [51,52]. Although such inducers seem fast, they encounter computational and memory problems when large-scale data is processed [5,7]. In the CUDT system [51], classification trees were boosted by parallelizing calculations inside each internal node. The attributes (and thresholds) in a node were checked in parallel to find the best test finally. The induction time was reduced up to 55 times (from two to seven times compared to a multi-core implementation). The idea was later extended to search the optimal tests in multiple nodes in parallel [52]. However, very similar speedups were obtained.

Another group of the GPU-supported inducers concerns ensemble classifiers, e.g., Random Forests [53,54] or gradient boosting DTs [55,56]. The simplest approach was used in the CudaRF system [53]. Each CUDA thread was responsible for building one tree in the forest. For a large number of trees (e.g., 256), it reached speedups of up to 30 times. An additional level of parallelism was investigated for mining data streams [54]. The calculations of the majority class in leaves and the splits in internal nodes were GPU-accelerated. The induction time was at least 300 times faster while maintaining similar accuracy. However, these multi-tree (''black-box'') solutions are beyond the direct interest of the research presented in this paper. We focus on a fast construction of the best single tree that can be interpreted by a data analyst (so-called ''white-box'' solutions).

As regards the evolutionary DT inducers, one of the first parallelizations was developed for computing clusters and used a hybrid MPI+OpenMP approach [12]. It concerned model trees and applied a classical master–slave paradigm with the control approach. The individuals from the population were distributed over the slaves that executed the most time-consuming operations (such as recalculation of the regression models in the leaves as well as fitness evaluation and genetic operators) in parallel. The experimental validation showed that such a hybrid solution was able to speed up the induction up to 23 times for 64 CPU cores.

The GPU-supported global DT inducers were also studied; however, only classification [8] and simple regression [57] trees were addressed. In both cases, the experimental validation on artificial and real-life datasets showed that the GPU-supported algorithms were capable of inducing trees two orders of magnitude faster than the original CPU version as well as multi-threaded OpenMP implementation. The approach for classification trees was also extended to multi-GPU platforms [7] as well as boosted using a Spark-based parallelization [58], to process really huge datasets (even up to billions of samples). Although the Spark-accelerated induction was easy to scale up, it was slower than the GPU-supported solutions.

This paper addresses the model trees whose complex tree-based representation with univariate tests in the internal nodes and multiple linear regression models in the leaves stand a new challenge. Their global induction needs a more elaborate and dedicated parallel approach, like a multiphase parallel model or even all-GPU parallel one [59]. In the case of classification trees, two algorithm phases directly related to fitness calculations (dataset samples redistribution and error calculations) are the most time-consuming operations. Thus, it was enough to delegate them onto the GPU to obtain a satisfactory speedup. In the case of model trees, a straightforward parallelization of the most time-consuming operation (calculations of regression models in modified leaves) seems to be not enough to obtain a satisfactory fast solution. There is a need to delegate all dataset-related operations into the GPU, even if their time execution is insignificant on the CPU, leaving fast EA-related tasks (e.g., drawing variants of genetic operators and nodes to be modified) and control decisions for the CPU.

3. Global Model Tree system - GMT

In this paper, we decided to use the Global Model Tree system (GMT) [5] to deploy the proposed solution. The GMT is a part of the Global Decision Tree framework that enables evolutionary induction of different DTs using various hardware (e.g., computer clusters [12,58], GPUs [8,9]) and different engines (e.g., CUDA [8], MPI+OpenMP [12], Spark [58]). The GMT system follows a typical EA framework [60] with an unstructured, fixed-size population and a generational selection.

3.1. Representation

The GMT system allows evolving regression trees with various representations of internal (based on the test type: univariate, oblique, mixed) and terminal (regression, model) nodes [14]. Individuals in the population are not specially encoded and are processed in their actual form (like in GP). The reason for this is that DTs are quite complex structures in which the number of nodes, the tests in internal node and the regression models in the leaves are not known in advance for a given dataset. This is why the direct tree-like representation may be more suitable, especially if the entire tree is searched in one EA run. Such a representation may suggest that we face a GP algorithm; however, the solution can be rather broadly classified into EAs.

We focus on univariate model trees [13], so every test in the internal node is based on a single attribute; however, our study can be easily applied to other representations. Each leaf of the tree contains a multiple linear model that is constructed using the standard regression technique for the samples associated with the node. A dependent (target) attribute y is explained by the linear combination of multiple independent attributes $x_1, x_2, \ldots, x_q$:

$$y = c_0 + c_1x_1 + c_2x_2 + \cdots + c_qx_q, \tag{1}$$

where q is the number of independent attributes and $c_{0..q}$ are fixed coefficients that minimize the sum of the squared residuals of the model. If all c_i $(0 < i \leq q)$ are equal to 0, then the leaf becomes a regression node with a constant target value equal to c_0.

3.2. Initialization and selection

In general, an initial population should be randomly generated in order to ensure sufficient diversity and cover the whole range of possible solutions [61]. Due to the large search space, the use of greedy heuristics in the initialization phase is often considered as a way of reducing the computation time. The disadvantage of this strategy is that EA can trap in the local optima. To maintain a balance between exploration and exploitation, initial individuals in GMT are created by using a simple top-down manner with randomly selected sub-samples of the original training data.

Tests are created according to the dipolar strategy [13]. A dipole is a pair of samples used to find the effective test (returning two different outcomes). At first, a sample that will constitute the dipole is randomly selected from a set of samples located in the considered node. The remaining samples are sorted in decreasing order according to the differences in target attribute

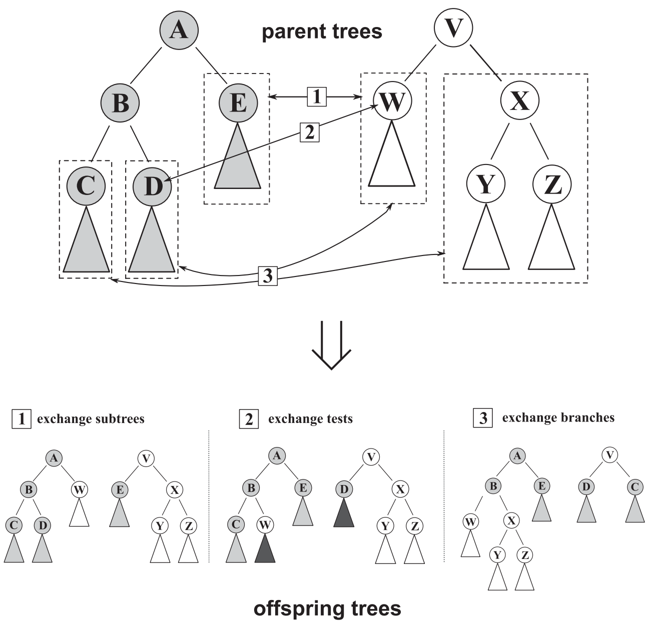

Fig. 5. Different variants of the crossover operator: 1 - exchange subtrees, 2 - exchange tests, 3 - exchange branches.

values in relation to the selected sample. The second sample that constitutes the dipole should be far enough to the first one. To find it, we used a mechanism similar to the ranking linear selection [60]. Finally, the test that splits the dipole is constructed on a randomly selected attribute, and the threshold is set in the midpoint between the pair that forms the dipole.

The linear ranking selection [60] is also applied as the proper evolutionary selection. Additionally, in each iteration, one individual with the highest fitness in the current population is copied to the next one (elitist strategy). The evolution ends when the fitness of the best individual in the population does not improve during the fixed number of generations, or the maximum number of generations is reached.

3.3. Genetic operators

Genetic operators are the main forces that control evolutionary search and provide, at the same time, a necessary diversity and novelty. Application of the operators can modify the tests in internal nodes, tree structure and models in the leaves. The GMT system applies two specialized genetic meta-operators corresponding to the classical mutation and crossover. For both operators, GMT provides several variants [5] that influence the tree structure and the splitting tests in the internal nodes. The crossover operator attempts to combine elements of two existing individuals in order to create new solutions (offspring) that share some characteristics of their "parents". Each crossover starts by selecting the positions (nodes) in two affected individuals. In the most straightforward variant, the subtrees starting in the selected nodes are exchanged (swapped) (see Fig. 5), which corresponds to the classical crossover from genetic programming [30]. When the non-terminal nodes are chosen and the number of outcomes is equal, tests or branches associated with randomly chosen nodes can also be exchanged. An asymmetric crossover is also possible. It duplicates a subtree with a lower prediction error and uses it to replace a subtree with a higher prediction error. Such an operator eliminates weak nodes and promotes ones with a small value of error.

The mutation of an individual starts with selecting a node type (equal probability of selecting a leaf or an internal node). Next, the ranked list of nodes of the selected type is created [62], and a mechanism analogous to the ranking linear selection [60] is applied to decide which node will be affected. Depending on the type of node, the ranking takes into account: (*i*) location (level) of the internal node in the tree — it is evident that modification of the test in the root node affects the entire tree and has a large impact, whereas the mutation of an internal node in the lower parts of the tree has only a local impact; and (*ii*) prediction error of the node (applied to both internal nodes and leaves). This way, internal nodes in the lower parts of the tree and/or those with higher error rates are more likely to be mutated.

The GMT framework offers several specialized variants of crossover and mutations. Here, we include a brief listing of these variations; however, for in-depth description, application and probability of use of each of the variants, please refer to works [13,14]:

- *new test* - reinitializes a test in the internal node using the dipolar strategy (another attribute can be used);
- *shift threshold* - shifts the splitting threshold of the test on the same attribute;
- *change model* - changes linear regression models in the leaves (add, remove, or change attributes);
- *prune* - changes an internal node into a leaf (acts like a standard pruning procedure) with a new multivariate linear regression model;
- *expand* - transforms a leaf into an internal node with a randomly chosen test (allows expansion of the tree and searches for more specific regions).

3.4. Fitness function

The fitness function, which drives the evolutionary search, should reflect the goal of the algorithm. In many data mining tasks, one tries to find the best predictor, but simultaneously, the simplicity of such a predictor is often desired. It is well-known that a predictor that works perfectly on the training data is usually much worse when tested on unseen data due to the overfitting problem [63]. Therefore, a multi-objective optimization is required to minimize the prediction error and the tree complexity at the same time. In the case of model trees, the predictor complexity must include not only the number of nodes but also the size of the linear models in the leaves.

The GMT system [5] provides various multi-objective optimization strategies, including weight formula, lexicographic analysis and Pareto-dominance [64]. As, in this paper, univariate model trees are considered, the simple weighted form of the fitness function, which is maximized, can be used:

$$Fitness(T) = [1 - \frac{1}{(1 + SS_e(T)/n)}] + \alpha \cdot k(T), \tag{2}$$

where $SS_e(T)$ represents the tree error calculated as the sum of squared residuals of the tree T estimated on the learning dataset, n stands for the number of samples, α is a user-defined parameter that reflects the relative importance of the complexity term (default value is 0.001) and $k(T)$ is the tree complexity. The term $k(T)$ can also be viewed as a penalty for over-parametrization. For model trees, it is equal to the sum of both: the number of internal nodes and the number of attributes in the linear models over all the leaves.

4. CUDA-accelerated GMT - cuGMT

The general idea of the GPU-supported solution (called cuGMT) is illustrated in Fig. 6. The evolutionary induction is controlled by a CPU, while the most time-consuming and dataset-related operations (samples' redistribution, models construction,

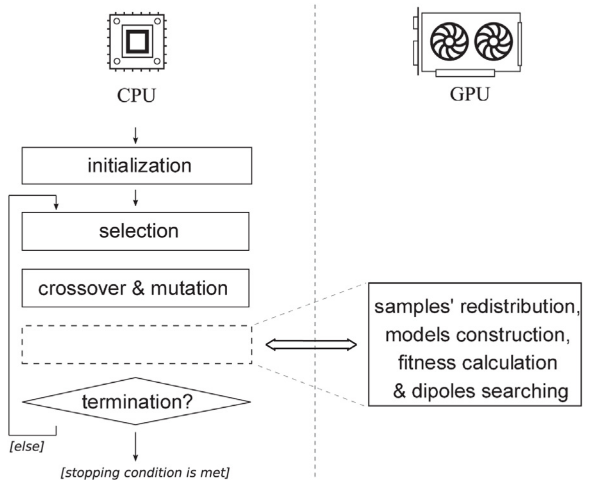
CPU
GPU
initialization
selection
crossover & mutation
termination?
[else]
[stopping condition is met]
samples' redistribution, models construction, fitness calculation & dipoles searching

Fig. 6. General idea of the GPU-accelerated evolutionary induction.

fitness calculation and dipoles searching) are delegated to a GPU. The initialization and selection steps remain unchanged compared to the original GMT system. These steps are not parallelized because they take an insignificant part of the overall time (less than 1% of the total execution time of the algorithm). Moreover, the initial population is created only once on small fractions of the dataset.

Concerning genetic operators, basic activities are run sequentially on the CPU, e.g., changes in the tree structure. However, after successful application of crossover or mutation, there is a need to evaluate the individuals, and then the GPU is called to calculate the fitness. This part of the algorithm is time-consuming operation since all samples in the training dataset need to be passed (redistributed) through to the tree, starting from the root to the appropriate leaves. The model construction itself could also be time-demanding, especially for large datasets. Then, large numbers of samples can be located in some leaves. As all samples that fall into a leaf are used to find the regression model, large size matrix operations may be required.

The construction of the cuGMT system ensures that the parallelization does not affect the behavior of the original EA (see Fig. 6). The evolutionary induction flow is driven by the CPU in a sequential manner, while the time-demanding parts of the algorithm are isolated and delegated to the GPU. The following sections describe the GPU parallelization in greater detail. Successively, decomposition strategy, six GPU-supported procedures (see also Listing 2 in Appendix A) as well as memory and implementation aspects are, among others, shown.

4.1. GPU-based parallelization overview

Before the evolutionary loop, the whole dataset is sent from the CPU side to the GPU side (see Fig. 7). This CPU–GPU data transfer is performed only once, and the data is kept on the GPU side till the evolutionary induction stops. On the GPU side, the dataset is saved in the global memory that has the biggest capacity among GPU's memories. This way, all GPU threads have access to the data.

In the evolutionary loop, each time the genetic operator is successfully applied, the GPU is asked to help the CPU, e.g., to rebuild models, calculate fitness or search dipoles (see Fig. 7). At first, the modified individual is sent to the GPU. Then, six GPU-supported procedures are called in sequence: `redistribute`, `separate`, `reorganize`, `calc_models`, `estimate_errors` and `reduction`. First, they are responsible for the redistribution of samples among the tree leaves. Then, models in the affected leaves are built (or rebuilt). When the models are updated, the GPU estimates a prediction error to find a fitness value. These GPU-supported procedures also search candidate samples for the dipoles from which they are finally constituted. In the end, the obtained tree errors as well as dipoles are sent back to the CPU that uses them to update the affected individual.

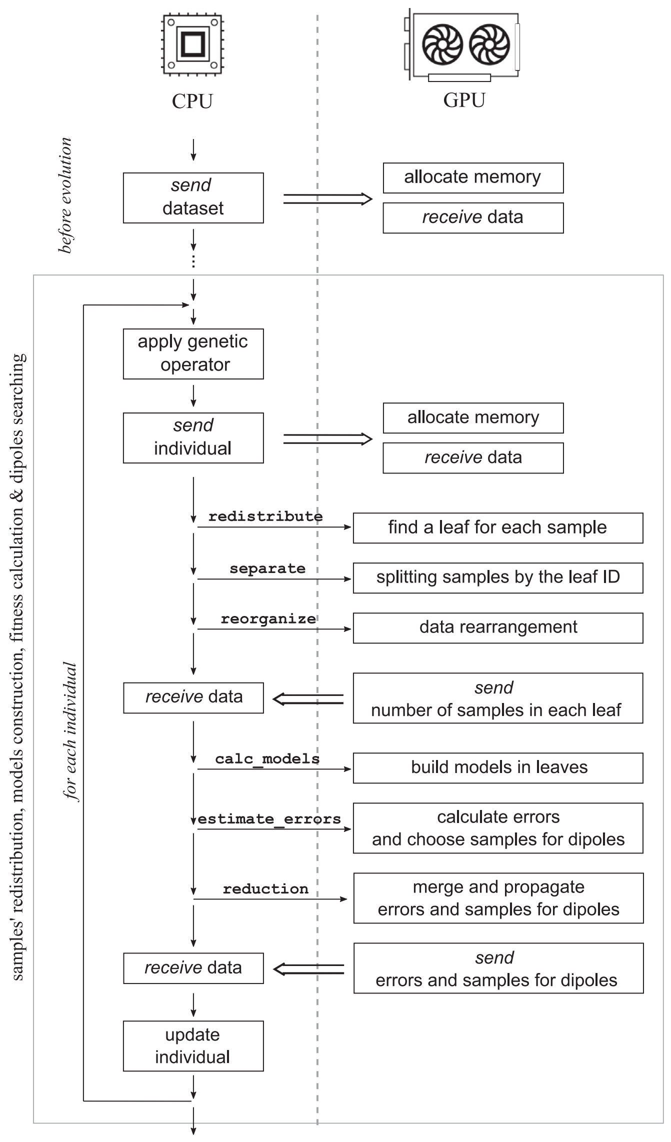
CPU
GPU
before evolution
send dataset
allocate memory
receive data
samples' redistribution, models construction, fitness calculation & dipoles searching
for each individual
apply genetic operator
send individual
allocate memory
receive data
redistribute
find a leaf for each sample
separate
splitting samples by the leaf ID
reorganize
data rearrangement
receive data
send number of samples in each leaf
calc_models
build models in leaves
estimate_errors
calculate errors and choose samples for dipoles
reduction
merge and propagate errors and samples for dipoles
receive data
send errors and samples for dipoles
update individual

Fig. 7. Flow chart of the CPU–GPU communication, memory allocation and kernels' execution.

4.2. Decomposition technique

In all six GPU-supported procedures, calculations are spread over GPU cores. We use the data decomposition strategy illustrated in Fig. 8. In this strategy, each GPU thread does similar jobs but operates on different dataset chunks. We decided to concentrate only on distributing the dataset, as it is a powerful and commonly used strategy for deriving concurrency when operating on large data [8,45].

The dataset is decomposed at two levels (see Fig. 8). At first, the whole dataset is spread into smaller parts that are processed by different GPU blocks. Next, in each block, the assigned samples are spread further over the GPU threads.

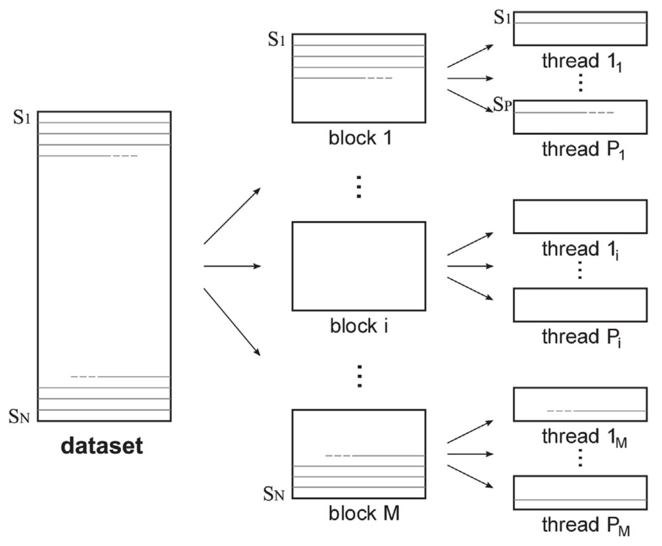

Fig. 8. Dataset decomposition strategy. The dataset is spread into smaller parts, first between different GPU blocks and then further between the threads. Adjacent threads inside blocks process/access neighboring data.

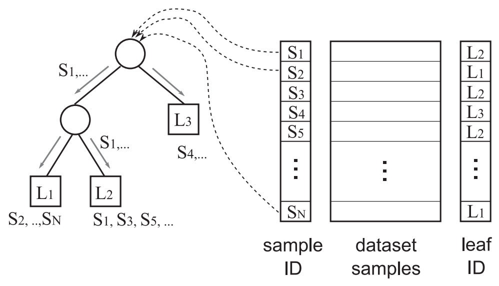

Fig. 9. `redistribute` procedure — all the dataset samples are passed through the tree starting from the root node to leaves. This way, each sample is associated with an appropriate leaf, e.g., 1st sample (S_1) is associated with 2nd leaf (L_2). The dataset samples are processed in parallel by GPU threads using the decomposition strategy from Fig. 8.

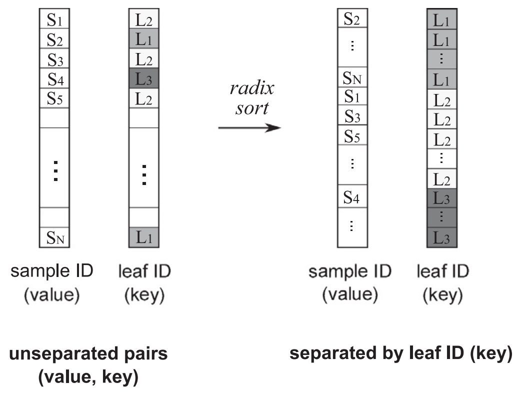

Fig. 10. `separate` procedure — separating the key–value pairs into contiguous buckets (key — the leaf's ID, value — the sample's ID). Tables with the leaves' and samples' IDs are initially created by the `redistribute` kernel.

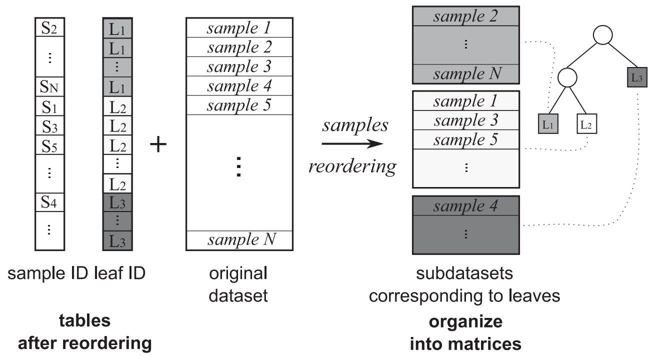

Fig. 11. `reorganize` — reorder of the dataset samples by the leaf ID using two auxiliary tables from the `separate` procedure. The dataset samples are reordered in parallel by GPU threads using the decomposition strategy from Fig. 8.

4.3. Six GPU-supported procedures

4.3.1. `redistribute` procedure

The `redistribute` procedure is a GPU kernel responsible for samples' redistribution. It searches an appropriate leaf for each training sample (see Fig. 9 and Listing 2). To this aim, all samples are propagated through the tree, starting from the root node and moving towards the leaves. The results are stored in two tables. The first table is used to save the identifiers of the samples (samples' IDs), while the second one stores the identifiers of the corresponding leaves (leaves' IDs).

Both tables are allocated in the GPU global memory, thus they are accessible for all threads. There is no need to synchronize threads as they work independently on different parts of the dataset. In the next algorithm steps, the tables are first sorted by the `separate` procedure and then used by the `reorganize` procedure to prepare data for finding multiple linear regression (MLR) models.

4.3.2. `separate` procedure

The role of this procedure is carrying out so-called pre-reorganization. It operates on two auxiliary tables (with samples' and leaves' IDs) created by the `redistribute` kernel. It prepares these tables for the `reorganize` procedure that uses them to reorder dataset samples. As illustrated in Fig. 10, the procedure separates the IDs of the samples by the corresponding IDs of the leaves. In other words, it splits the key–value pairs into contiguous buckets. The key is a leaf ID, the value is a sample ID, while the number of the buckets equals the number of the tree leaves. Each bucket contains IDs of the samples that fall into the same leaf.

In cuGMT, the `separate` procedure is solved by sorting the key–value pairs. We apply the radix sort algorithm that is considered one of the fastest on-GPU sorting algorithms [65,66]. The state-of-the-art GPU-based radix sort implementation from the CUB library [67] is used. The CUB library provides a collection of basic primitives (sort, scan, reduction, ...) at three levels: device, thread block and warp. In our case, a device-wide sort primitive `cub::DeviceRadixSort::SortPairs` is taken.

4.3.3. `reorganize` procedure

This GPU kernel prepares data (in the form of matrices) for building MLR models directly. The kernel reorders (splits) the dataset samples by the leaf ID (see Fig. 11 and Listing 2). To this aim, it uses two auxiliary tables (with samples' and leaves' IDs) that have already been reorganized by the `separate` procedure. In addition, a table of the same size as the dataset is used to store the reordered samples temporarily. The original dataset is not modified.

All auxiliary tables, as well as the original dataset, are allocated in the GPU global memory. Thus, they are accessible for all threads. Different threads operate on different parts of the dataset

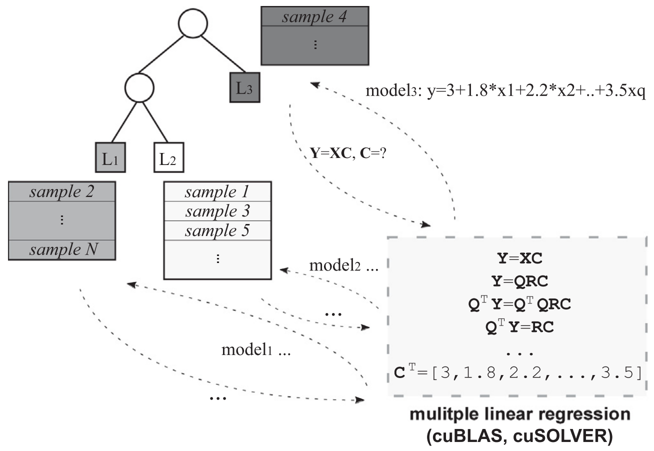

Fig. 12. `calc_models` procedure — building models in the leaves. In each affected leaf, a MLR model is constructed using the QR factorization and GPU linear algebra libraries (cuBLAS, cuSOLVER).

(following the proposed decomposition strategy). Thanks to these three auxiliary tables, threads know in advance how to reorder dataset samples without disturbing each other. Thus, all threads (both inside and outside blocks) reorder samples concurrently without any synchronization.

Finally, the dataset is divided into multiple matrices, one for each leaf (see Fig. 11). In addition, the samples (the values of the samples' attributes) within each matrix are transposed compared to the original dataset. This transposition is done at the same time when the reorganization takes place (during copying the samples from the original dataset to the auxiliary table in appropriate order). The original dataset remains unchanged till the end of the evolution. The created matrices (one for each leaf) are then used in the next GPU-supported procedure to build MLR models.

4.3.4. `calc_models` *procedure*

This procedure is directly responsible for models construction. It aims to build MLR models in all affected leaves (leaves that are directly modified or lie below a modified node). Each model is built using only the samples associated with the considered leaf (see Fig. 12) in accordance with Eq. (1).

To determine q unknown coefficients $\mathbf{C} = [c_0, c_1, \ldots, c_q]^T$ in Eq. (1), multiple linear equations need to be solved. They can be collected into a single matrix equation: $\mathbf{Y} = \mathbf{XC}$, where $\mathbf{X}$ is a matrix with dataset attribute values (assuming that the first column is 1) and $\mathbf{Y}$ provides the dataset target values. For each leaf, $\mathbf{X}$ and $\mathbf{Y}$ are prepared by the previous GPU-supported procedure (`reorganize` kernel). The equation cannot be directly solved. The QR factorization [68,69] is applied to put it into a tractable form. In brief, the matrix $\mathbf{X}$ is decomposed into two matrices $\mathbf{Q}$ and $\mathbf{R}$ ($\mathbf{X} = \mathbf{QR}$) (see Fig. 12), where $\mathbf{Q}$ is orthonormal and $\mathbf{R}$ is upper triangular. Then, both sides of the matrix equation are multiplied by $\mathbf{Q}^T$ and we get $\mathbf{Q}^T\mathbf{Y} = \mathbf{Q}^T\mathbf{QRC}$. The left side simplifies and we end up with $\mathbf{Q}^T\mathbf{Y} = \mathbf{RC}$ that is tractable since $\mathbf{R}$ is upper triangular.

In cuGMT, two GPU-accelerated linear algebra libraries are applied to implement the QR factorization and finally find the MLR coefficients: cuBLAS[1] and cuSOLVER.[2] For each modified leaf, the CPU calls the following functions (see Listing 2). For the QR factorization, the function `cusolverDnQgegrf()` is used. The modified left-hand side $\mathbf{Q}^T\mathbf{Y}$ is computed with `cusolverDnSormqr()`.

[1] NVIDIA cuBLAS, https://docs.nvidia.com/cuda/cublas/.

[2] NVIDIA cuSOLVER, https://docs.nvidia.com/cuda/cusolver/.

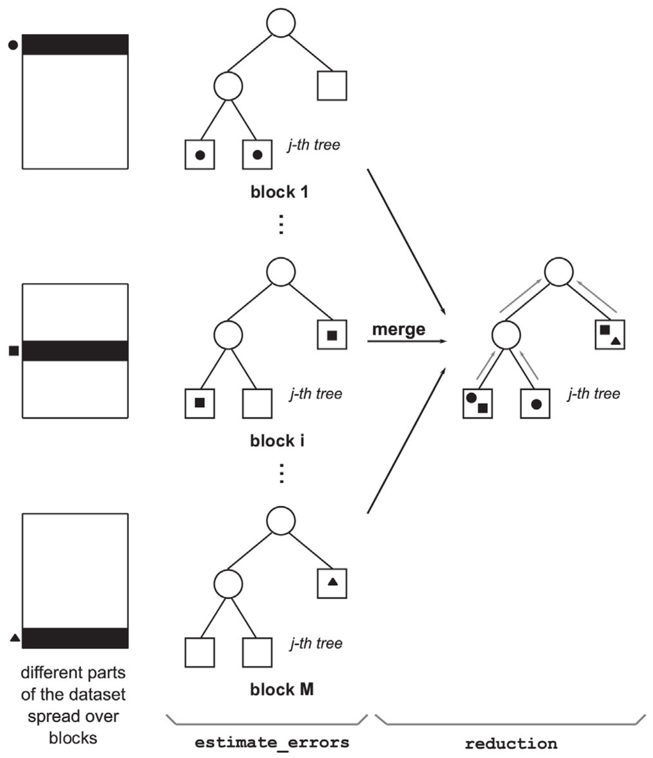

Fig. 13. `estimate_errors` and `reduction` - computing errors in all nodes of the tree to find the fitness value finally. In `estimate_errors` kernel, the same individual (tree) is processed in parallel by successive blocks (of threads). Each block is responsible for a different part of the dataset and stores partial sums of errors in separated memory space ("copy" of the individual). The `reduction` kernel merges the partials results from different blocks and propagates errors and dipole samples from the leaves towards the tree root.

Finally, the function `cublasStrsm()` backsolves the equation $\mathbf{Q}^T\mathbf{Y} = \mathbf{RC}$ providing the MLR coefficients.

4.3.5. `estimate_errors` *and* `reduction` *procedures*

The `estimate_errors` and `reduction` procedures are the GPU kernels. Their role is the error (and finally fitness) calculation. They use the assignment of samples to leaves (provided by `redistribute` kernel) as well as the MLR models (provided by the `calc_models` procedure). Thus, there is no need to propagate the dataset samples through the tree again.

The `estimate_errors` kernel computes errors in all tree leaves (see Fig. 13 and Listing 2). In each leaf, the squared differences between the predictions of the processed samples and the leaf prediction are found and added. The dataset samples are spread over GPU threads. Thus, different threads compute the squared errors for various samples. In each leaf, the sum of the squared errors is stored. Since more than one thread may attempt to add its temporary result in a single leaf, synchronization is needed. As regards threads inside blocks, the CUDA atomic operation `atomicAdd` is applied [11]. Concerning threads in different blocks, there is no native way to synchronize them. Thus, for each GPU block, a separated space in shared memory is created to collect the results for the evaluated individual (it is visible by all threads within the block). It can be seen as separated copies of the individual are created but only the place for the results is multiplied but not the DT structure itself. Threads from different blocks store results in their own space in memory. This way, all threads can process the dataset in parallel.

The `reduction` kernel gathers and merges information about the sum of squared errors from partial results allocated in each

GPU block (reduction with the addition operator) (see Fig. 13 and Listing 2). This operation provides the overall sum of squared errors in all leaves. Finally, the computed sums are propagated from the leaves towards the root node.

4.4. Memory and implementation aspects

In order to suit the computation and memory model of GPGPU [11], the representation of individuals is different on the GPU side than on the CPU one. Before an individual is transferred to the GPU, its flat representation is created [8] (see Listing 1). Based on its references (pointers) CPU representation, a one-dimensional array is built. At the beginning of the array, the root information is stored, following its descendants and so on. The array index of the left child of the *i*th node equals $(2*i+1)$, while for the right child, it is $(2*i+2)$. The globally induced DTs are usually not large; thus, the time of transformation and transfer should be relatively short or even negligible for big datasets.

Another important issue of GPU-parallelized applications is memory access pattern as it can drastically impact computational efficiency [9,70]. In our case, the most frequently used data is the information about the nodes of DTs (attributes and thresholds) as well as the training dataset that is also the most massive data. The dataset samples as well as trees are stored in one-dimensional arrays. Generally, there are two major data layouts for arrays of items: Structure-of-Arrays (SoA) and Array-of-Structures (AoS). The other more sophisticated layouts are hybrid formats [71]. In SoA, multi-value data (e.g., values of subsequent attributes of the dataset) are stored in separated arrays and the arrays are grouped in a structure. In AoS, they are grouped first in a structure (e.g., a structure of sample attributes) and an array of such structures is used. Although for both layouts the same data is represented, they imply completely different memory access patterns. We decided to apply the SoA layout for individuals and the dataset samples. The SoA layout is usually preferred from the GPU perspective because one thread may copy data to cache for other threads providing a coalesced memory transaction [70,71]. In recent works on classification trees [5,7], such memory organization provided the best acceleration.

The training dataset is kept during the whole evolution on the GPU side. It is only transferred to the GPU once before the evolution starts. Thus, the time of sending is negligible even if the dataset is large. We decided not to store the arrangements of samples in each tree node but only in the leaves, in contrast to the sequential version. It helps to save memory space as well as reduce memory transactions at the cost of more arithmetic operations. However, some variants of the mutation operator require a randomly chosen dipole. In order to select the dipole in a node, the information about which samples fall below that node is necessary. As the propagation of the samples is performed on the GPU side, the CPU does not have access to such information. To minimize the GPU–CPU transfer, the assignment of leaves (IDs) to samples (IDs) is not sent to the CPU. Instead, the GPU provides the CPU with the dipole (two samples) for each tree node. These two samples are found by the `estimate_errors` and `reduction` kernels.

In the `estimate_errors` kernel, one of the samples (that reached this leaf) is randomly selected in each tree leaf. The same operation is done independently in each block. When the results from different blocks are merged (`reduction` kernel), two samples are selected from the available set of samples in each leaf according to the differences in the dependent attribute value. Two samples with the biggest (possible) difference are chosen. In the end, the selected samples are propagated to the tree root to provide samples for dipoles also for internal tree nodes. The dipole from the randomly chosen child is copied to the parent node.

Table 1
Characteristics of the real-life and artificial datasets: name, number of samples, number of attributes.

Dataset	No. samples	No. attributes
Pol	15 000	48
Elevators	16 599	18
Cal housing	20 640	8
House 16H	22 784	16
House 8L	22 784	8
2dplanes	40 768	10
Fried	40 768	10
Mv	40 768	7
Layout	66 615	31
Colorhistogram	68 040	31
Colormoments	68 040	8
Cooctexture	68 040	15
Elnino	178 080	9
Year	515 345	90
Suzy'	5 000 000	17
Armchair10K	10 000	2
Armchair50K	50 000	2
Armchair100K	100 000	2 (4, 6, 8, 10)
Armchair500K	500 000	2
Armchair1M	1 000 000	2 (4, 6, 8, 10)
Armchair5M	5 000 000	2
Armchair10M	10 000 000	2

The global memory of the GPU has the largest capacity but, at the same time, the highest latency access among its memories. Thus, in the kernels, data stored in the global memory and frequently used (e.g., tree nodes, sample attributes) is explicitly copied to local kernel variables. At the end of the kernel, the data is transferred back into the global memory containers, if needed. This way, each thread tries to accumulate temporary values into registers (fast on-chip memory but of small capacity).

5. Experimental setup

5.1. Datasets details

Experimental validation has been performed on both real-life and artificial datasets. The details of each one are presented in Table 1. We have chosen the thirteen biggest datasets (half) that were used to verify the prediction performance of GMT [13]. They are provided by Louis Torgo [72] and the UCI Machine Learning Repository [73].

In addition, two large datasets from the UCI repository are included: *Year* and *Suzy*. The *Year* dataset concerns the prediction of the release year of a song based on 90 audio attributes. The *Suzy* originally concerns classification. However, due to the lack of publicly available larger regression datasets, it was slightly transformed and applied. The original class was eliminated and the last attribute was predicted. The only purpose of this operation was to investigate the speedup of the algorithm and not the prediction performance. The modified dataset is called *Suzy'*.

Concerning the artificial datasets, the problem called *Armchair* [13] was analyzed. In Fig. 14 (top), the problem is visualized in 3D, and an example of the model tree generated by GMT is shown. This dataset seems to be relatively easy; however, many traditional approaches (top-down without look-ahead) fail to find the optimal first test. As a result, the obtained trees are overgrown. The GMT induces a model tree built of five models in leaves (see Fig. 14, bottom) and the estimated prediction error measured by root mean square error (RMSE) is less than 0.1. Whereas, two popular greedy inducers, M5 [20] and WEKA REPTree [74], generate the trees with 18 and 87 models with $RMSE = 2.33$ and $RMSE = 1.48$, correspondingly.

The use of the synthetic problem allowed us to scale datasets freely. We investigated a various number of training samples,

Table 2
General specification (processing and memory resources as well as computing capability) of four NVIDIA GPU cards/accelerators used in the experiments.

NVIDIA	Engine		Memory		Computing
GPU	No. CUDA cores	Clock rate [MHz]	Size [GB]	Bandwidth [GB/s]	capability
GeForce GTX 780	2304	863	3	288.4	3.5
GeForce GTX Titan X	3072	1000	12	336.5	5.2
Tesla P100	3584	1328	12	549.0	6.0
GeForce RTX 2080 Ti	8704	1350	11	616.0	7.5

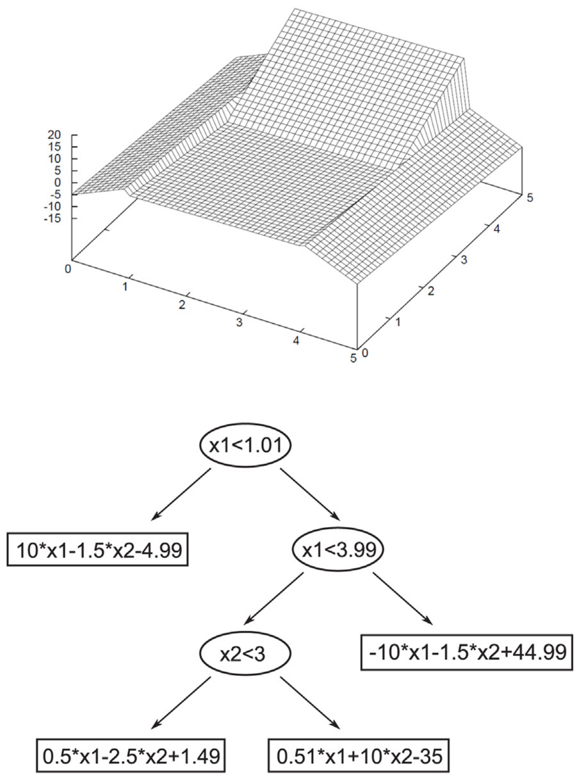

Fig. 14. Visualization of the *Armchair* dataset (top) and the corresponding example of a model tree generated by GMT (bottom).

from 10 thousand to 10 million. We also studied the influence of dataset dimension (number of attributes). The original datasets have two attributes. Into two chosen variants (*Armchair100K* and *Armchair1M*), randomly generated (noisy) attributes from uniform distribution were added, successively 2, 4, 6, and 8.

5.2. Hardware and software details

In the experiments, we tested four different NVIDIA GPU cards. Their basic specification is displayed in Table 2. The used GPUs are a mixture of current and previous generations of NVIDIA architectures. The first GPU card was launched in 2013 but it is still quite powerful. It is based on Kepler architecture and is recognized as one of the slowest aging cards. The Titan X card was launched in 2015 and is based on Maxwell architecture. Tesla P100 is the professional-level GPU accelerator. It is based on Pascal architecture and currently costs about $5000 (at least 10 times more than the first GPU card). The last GPU card is one of the newest NVIDIA products that is based on Turing architecture. It delivers 13.45 TFLOPS of peak single-precision (FP32) performance (theoretical performance), about 1.5 times more than Tesla P100.

The host machine that performed the time performance comparison between CPU and GPU-accelerated algorithms was a server with 2× Intel Xeon E5-2620 v4 (20 MB Cache, 2.10 GHz, 8 physical cores) and 256 GB RAM. It was running the 64-bit Ubuntu Linux 16.04.6 LTS operating system. The algorithm was implemented in C++, while the GPU-supported parts were in NVIDIA CUDA-C and compiled by nvcc CUDA version 10.0.130 [75]. In addition, three CUDA libraries were employed: cuBLAS, cuSOLVER and CUB version 1.8.0 [67]. Single-precision arithmetic was applied. The sequential and CPU-parallelized (OpenMP) versions [15] were compiled by gcc version 5.4.0.

5.3. Parameters details and measurement methodology

In all experiments, a default set of parameters previously applied to the sequential version of GMT was used [5,13]. The main parameters were as follows: population size = 64 individuals; crossover probability = 20% assigned to the tree; mutation probability = 80% assigned to the tree; elitism rate = 1 individual per generation and the number of iterations = 10 000.

Each configuration (dataset and CPU/GPU) was run 10 times. The average value of run-time measurements was presented. For the biggest datasets and CPU-based computing, the number of runs was decreased to five runs and/or the total time was estimated based on fewer iterations than declared (what is marked along with the results). As a speedup [45], we measured the ratio of the time necessary to solve a problem sequentially to the time required by its parallel version. In our case, it was the OpenMP or GPU-supported parallelization.

The main aim of this paper is to assess the time performance of the GPU-accelerated inducer — cuGMT. Thus, the exact results for the prediction error are not included. In previous works (e.g., [5,13]), GMT was thoroughly validated concerning the prediction error, tree size and model size in the leaves. It was shown that GMT outperforms popular, greedy, single-tree counterparts, e.g., M5 [20] (state-of-the-art model tree inducer) and Weka REPTree [10]. However, the greedy inducers were substantially faster, and this is why the boosting of the evolutionary approach is so expected. As regards the more complex solutions, like Gaussian Process Regression or ensembles of trees (Boosting, Bagging, Random Model Trees, and Additive Groves) [76], statistical analysis showed that there was no strong winner [13]. However, GMT appeared to be the most stable solution and gave much easier models to interpret.

Two fitness functions were used. The first choice was the one based on Bayesian information criterion (BIC) [77] as it was shown that it works well as a multi-objective optimization strategy in the induction of regression [32] and model trees [13]. However, it turned out that such a fitness function tends to choose trees that are too complex (too large) when the data size grows (from ≈ 1 million samples). Thus, for such datasets, we decided to apply a simple weighted formula (2) with $\alpha = 0.001$ where the number of samples does not directly influence the complexity penalty. This formula was thoroughly verified in large-scale data mining concerning the evolutionary induction of classification trees [7,8]. For small and medium-size datasets, we left the BIC-based formula to be coherent with the protocol that was applied to validate the sequential GMT version [5,13].

Table 3
Execution time (in seconds) of the GPU-accelerated, OpenMP-accelerated [12] and sequential versions of GMT [13], using real-life datasets. For the fastest GPU and the sequential version, the time is also rounded in minutes/hours/days and presented in brackets.

Dataset	RTX 2080 Ti		Tesla P100	Titan X	GTX 780	OpenMP	Sequential	
Pol	1 226	(20.5 min)	1 933	2 557	3 652	4 124	9 259	(2.5 h)
Elevators	511	(8.5 min)	825	894	1 133	5 992	33 268	(9 h)
Cal housing	431	(7 min)	728	789	895	4 350	18 861	(5 h)
House 16H	624	(10.5 min)	936	957	1 181	9 188	21 831	(6 h)
House 8L	465	(8 min)	801	850	1 092	3 048	8 374	(2.5 h)
2dplanes	662	(11 min)	961	982	1 194	27 100	74 969	(21 h)
Fried	548	(9 min)	883	912	1 138	9 812	23 300	(6.5 h)
Mv	872	(14.5 min)	1 119	1 679	2 338	6 657	21 442	(6 h)
Layout	1 180	(19.5 min)	1 563	1 858	2 550	77 985	328 404	(91 h)
Colorhistogram	2 138	(35.5 min)	4 537	4 851	7 085	27 073	94 755	(26 h)
Colormoments	741	(12 min)	975	1 036	1 205	17 527	62 891	(17.5 h)
Cooctexture	1 183	(20 min)	1 648	1 918	2 617	45 381	153 494	(43 h)
Elnino	1 696	(28 min)	4 166	4 295	6 980	19 475	66 216	(18.5 h)
Year	4 257	(71 min)	9 111	9 222	12 288	427 744	1 758 964[a]	(20 days)
Suzy'	149 565	(42 h)	188 730	194 985	283 357	14 650 219[a]	61 791 636[a]	(2 years)

[a]Note: Execution time estimated based on 1000 first iterations.

Concerning the CUDA kernel execution, the number of thread blocks and threads per block was set at 256 × 256 by default. However, the default configuration was modified in order to keep the workload of each thread when small and medium-size datasets were processed. We decided to provide between four and eight samples for each thread. Thus, for the following real-life datasets (see Table 1), the settings were changed to:

- 64 × 32 for 1st–5th datasets;
- 128 × 64 for 6th–8th;
- 128 × 128 for 9th–12th;
- 256 × 128 for 13th one.

The *Archmair10K*, *Archmair50K* and *Archmair100K* datasets were processed using 64 × 32, 128 × 64 and 128 × 128 configurations, correspondingly.

6. Results and discussion

6.1. Real-life datasets

Table 3 presents the results for all tested real-life datasets. The mean execution times of cuGMT (using four different GPUs), OpenMP-accelerated, and (sequential) GMT versions are shown. In all cases, the GPU-accelerated solution is able to speed up the induction. For the smallest dataset (*Pol*), the scale of the time improvement (a few times) is the least remarkable. The strongest GPU (RTX 2080 Ti) reduces the induction time from ∼2.5 h to ∼20.5 min (7.5×). For datasets with more training samples, the time of evolutionary induction increases. Hopefully, the acceleration provided by cuGMT also grows, allowing an inducer to be generated in minutes instead of hours. For example, in the case of *2dplanes2*, the time is reduced from ∼21 h to ∼11 min (113×), while for *Layout* it is reduced from ∼91 h to ∼19.5 min (278×), using RTX 2080 Ti GPU.

For the biggest datasets, the acceleration is the most prominent (about 410× in both cases). A similar speedup may be explained by a similar dataset memory requirement. Although the *Suzy'* dataset has about 5× more samples than the *Year* dataset, the former has about 5× fewer attributes then the latter. However, the number of samples (5 million) notably influences the induction time. Concerning the *Suzy'* dataset, the sequential evolutionary induction took so long that its time had to be estimated based on fewer iterations. We verified the execution times of the GPU-supported induction for an even larger dataset: *Higgs* (11 million samples, 28 features) that was transformed similarly as *Suzy* dataset. The time of induction was about six days using RTX 2080 Ti GPU. Due to very long computation times, it was not processed using the sequential version of the algorithm.

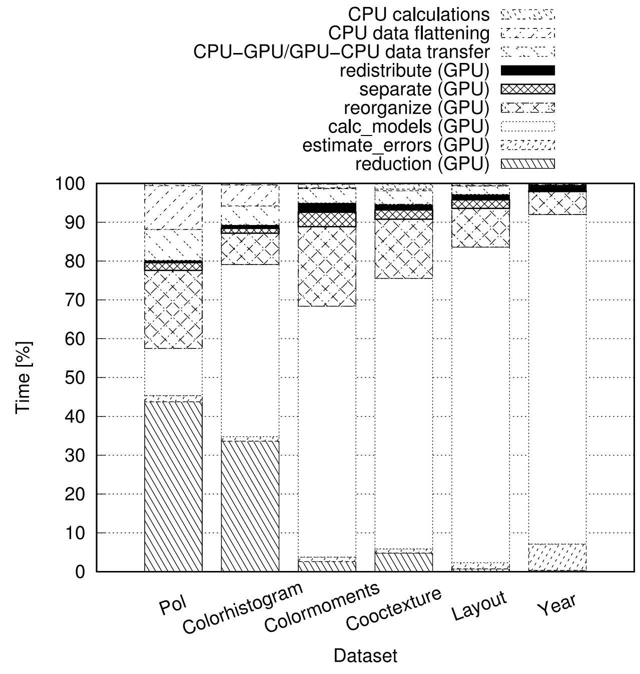

Fig. 15. The run-time breakdown of the GPU-accelerated algorithm using RTX 2080 Ti GPU for chosen real-life datasets. The average time (as a percentage of total run-time) of the most relevant parts is shown, both communication between CPU–GPU/GPU–CPU and GPU/CPU computations are included.

The differences in acceleration between datasets can be investigated with the run-time breakdown presented in Fig. 15. It shows the mean time (in percentage) spent on executing the most relevant parts of the algorithm (e.g., CPU calculations, data transfer between CPU–GPU/GPU–CPU, GPU kernels) for six chosen datasets using RTX 2080 Ti. We clearly see that, for all datasets, the time spent by the CPU on calculations is very short in comparison with the total run-time (from 0.04% for *Year* to 1% for *Cooctexture*). Another observation is that the datasets can be divided into two groups. For the first one (from the left, *Pol* and *Colormoments*), the data flattening (conversion to one-dimensional array representation on the CPU) and reduction (GPU) operations take considerably more time then for other datasets. The time of these two operations strongly depends on the size of model trees (number of nodes + sizes of models in leaves) which are much bigger than for other datasets [13]. Another important issue is the necessity of threads synchronization when merging up the results, which can also diminish the parallelization potential. Moreover, the size of the GPU's shared memory is usually limited to 48 kB per SM [75]. Thus, for larger trees, the algorithm uses the default global memory space during the reduction. All these issues may explain the observed separation.

The data flattening operation is an overhead (excess computation time for parallelization purposes). For *Pol* dataset, it takes more than 10% of the total execution time. Moreover, as it is the smallest dataset, the time of CPU–GPU/GPU–CPU memory transfers (communication overhead) is relevant (8% vs. 0.5% for *Cooctexture* or 0.1% for *Year*). These two overheads affect the time performance and are probably the reasons why the induction time is decreased only a few times in this case. When comparing the *Pol* and *Colorhistogram* datasets, we see that the growth in the number of samples reduces the overhead about two times (in relation to other algorithm parts), providing a higher speedup (44×).

The second group consists of four datasets on the right side of Fig. 15 (*Colormoments*, *Cooctexture*, *Layout* and *Year*). The flattening time is almost unnoticeable for all of them, and the communication overhead contribution decreases when the number of training samples and attributes grows. *Colormoments*, *Cooctexture*, and *Layout* datasets have roughly the same number of samples but a different number of attributes: 8, 15 and 30, correspondingly. Each time that the number of attributes grows, the time needed to build models (`calc_models` procedure) increases in comparison to other algorithm parts. As it is the most time-demanding procedure that is parallelized, the obtained speedup also increases. For example, using RTX 2080 Ti, the induction time is reduced about 85× (*Colormoments*), 130× (*Cooctexture*), and 278× (*Layout*).

Considering each tested GPU separately, Table 3 shows that all of them are able to speed up the induction of model trees. The best results are obtained by RTX 2080 Ti GPU, as it is the strongest one. In three cases (*Colorhistogram*, *Elnino*, *Year*), the RTX 2080 Ti card is even 2× faster than Tesla P100.

For most of the datasets, Tesla P100 is slightly better than GTX Titan X. The exceptions are *Cooctexture*, *Layout*, *Mv* and *Pol* datasets, where Tesla P100 provides more than 15% better acceleration. Considering the theoretical single-precision computing power, Tesla P100 is about 30% stronger than the former GPU, which may explain the time differences. The GTX 780 GPU card is the oldest among the tested GPU cards and yet generates enough to accelerate the induction significantly. The GTX 780 is also the cheapest GPU and provides the best performance per price factor among the tested GPUs. At the same time, it consumes as much energy as other GPUs, and thus, has the worst performance per watt factor. We can also read from Table 2 that each succeeding (newer) GPU card is equipped with faster memory that certainly helps to obtain better speedups.

Using the same workstation equipped with two 8-core Xeon CPUs (besides a GPU), we also verified how much speedup can be provided by a multi-threaded CPU implementation. In Table 3, the execution times of the OpenMP-based parallelization [12] are given. We see that it is able to accelerate the induction time a few times using 16 OpenMP threads (run on 16 CPU cores). It is competitive with the GPU-based solution only for the smallest dataset (*Pol*). This multi-threaded CPU acceleration, when combined with MPI and deployed on a computing cluster, was able to speed up the GMT more [12]. Using 64 cores, such a hybrid MPI+OpenMP solution provided speedups of up to 23 times for the real-life and artificial datasets. For small- and medium-size datasets (e.g., *Pol*, *Fried*), the results are competitive or even better than those provided by cuGMT. When the number of samples grows, the hybrid MPI+OpenMP solution is far below the possibilities of the GPU-supported algorithm.

6.2. Artificial datasets

Table 4 presents the average time of induction for different variants of the *Armchair* dataset. The results include the execution times of three GMT versions: cuGMT (using four different GPUs, from low- to high-end), an OpenMP-accelerated one [12] and a sequential version. We clearly see that, with the increase in the number of samples, the time of evolutionary induction grows (linearly, in most cases). The GPU-supported solution decreases the time of calculation substantially, e.g., from ∼16 h to ∼11 min (*Archmar100K*) or from ∼6.5 days to ∼48 min (*Archmar500K*). For 5 and 10 million training samples, the exact time of the sequential induction was hard to obtain because of very long computation times. On the other hand, we see that cuGMT manages with such large-scale datasets, and could handle even larger data.

To show the scale of the improvement, the speedup of cuGMT and OpenMP-supported GMT over the sequential version is presented in Fig. 16. It is clearly visible that the acceleration grows when more samples are considered. For the smallest *Armchair* configuration (10 thousand samples), the provided acceleration is about 16× for RTX 2080 Ti GPU. It is enough to reduce the induction time from ∼1.2 h to ∼4.5 min. When 1 million samples are processed, the acceleration reaches its peak of about 200×, decreasing the calculation time from ∼15 days to only ∼2 h. For a larger number of samples, the speedup saturation is observed.

The observed speedup behavior may be explained by the different time contributions of algorithm parts when the number of samples grows, which is illustrated in Fig. 17. It visualizes the average time (as a percentage of total run-time) of data transfer, GPU procedures and the most time-demanding CPU jobs. For the smallest *Armchair* configuration, CPU calculations and data transport overhead take an important fraction of the total tree induction time (about 20%). It is one of the main reasons why the acceleration is only about 16× in this case. The time of these two operations decreases (in relation to other activities) when the number of samples grows. Simultaneously, the `calc_models` procedure increases its time contribution and, thus, the speedup grows.

In Fig. 17, we also see that the time contribution of two GPU kernels (`redistribute` and `estimate_errors`) getting more important (rises from ∼5% to ∼20%) when the number of samples grows. Starting from 5 million samples, their time of execution increases so much that the time factor of the `calc_models` procedure begins to decrease. This is exactly the point where the speedup saturation is observed. For the two above-mentioned GPU kernels, the occupancy of CUDA cores depends a lot on the number of blocks and threads used, among others. We preliminarily verified that by using more CUDA blocks/threads, it is possible to shift the speedup peak further (e.g., to about 220× for 512 × 512 configuration). We also observed that the time of `calc_models` kernel increased slower starting from 5 million samples (not proportionally to the number of samples, as for fewer ones). Reasons may be found in the internal implementation of applied cuBLAS and cuSOLVER libraries, which we left for future investigation.

Table 4 and Fig. 17 show that all GPUs provide substantial speedup, similarly as observed for the real-life datasets. Here, however, the differences between GPUs are more prominent. For example, RTX 2080 Ti is almost consistently about 2× better than Titan X GPU, which coincides exactly with difference of the computational power between them.

The use of an artificial problem gave us the possibility to scale datasets freely. Thus, not only the impact of the number of samples but also the dimension (number of attributes) was investigated. Fig. 18(a) shows the speedup of cuGMT when the number attributes grows (from 2 to 10) while the number of samples is

Table 4
Execution time (in seconds) of the GPU-accelerated, OpenMP-accelerated [12] and sequential versions of GMT [13], using the artificial *Armchair* dataset. For the fastest GPU and the sequential version, the time is also rounded in minutes/hours/days and presented in brackets.

Dataset	RTX 2080 Ti		Tesla P100	Titan X	GTX 780	OpenMP	Sequential	
Armchair10K	267	(4.5 min)	431	439	601	1 038	4 221	(1.2 h)
Armchair50K	432	(7 min)	733	769	1 046	6 372	22 691	(6.5 h)
Armchair100K	654	(11 min)	1 128	1 206	1 670	16 843	57 109	(16 h)
Armchair500K	2 870	(48 min)	4 846	5 884	7 420	169 601	557 545	(6.5 days)
Armchair1M	6 404	(2 h)	10 214	12 842	14 964	407 287	1 303 450	(15 days)
Armchair5M	33 955	(9.5 h)	57 417	71 673	83 799	1 987 815[a]	6 158 273[a]	(71 days)
Armchair10M	70 648	(20 h)	127 022	142 125	175 978	4 129 295[a]	12 269 675[a]	(142 days)

[a]Note: Execution time estimated based on 1000 first iterations.

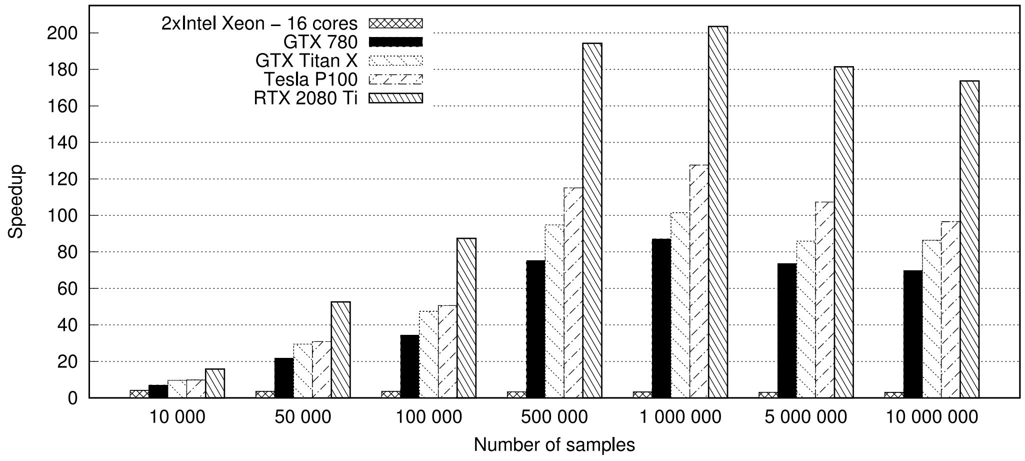

Fig. 16. Mean speedup for the artificial *Armchair* dataset when the number of samples grows.

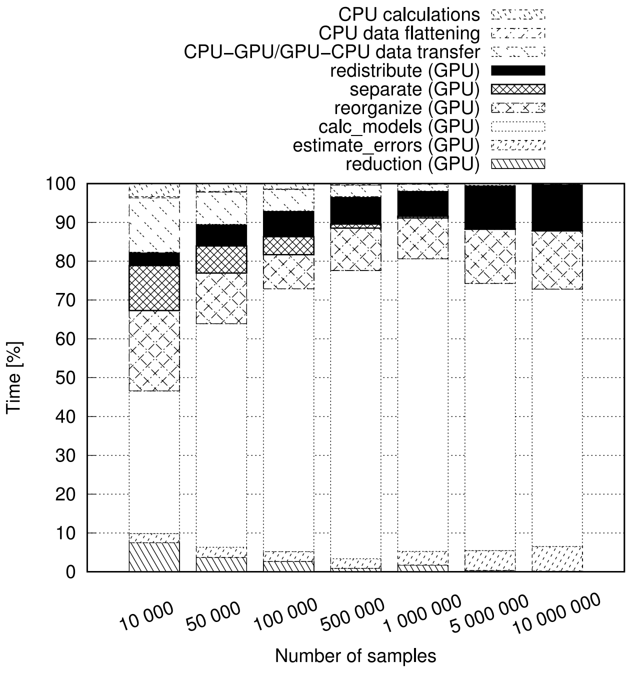

Fig. 17. The run-time breakdown of the GPU-accelerated algorithm using RTX 2080 Ti GPU for the increasing number of samples of the *Armchair* dataset. The average time (as a percentage of total run-time) of the most relevant parts is shown, both communication between CPU–GPU/CPU–GPU and GPU/CPU computations are included.

constant. Due to a very long induction time of the sequential GMT, two cases were calculated (100 thousand and 1 million samples). We see that the speedup increases when the dataset dimension rises (e.g., to 200× and 550×, correspondingly). The obtained acceleration is very satisfactory. When, for example, 6 attributes are considered, it allows the induction to be reduced from ∼39 h to ∼19 min (*Armchair100K*), and from ∼41 days to ∼3 h (*Armchair1M*).

Fig. 18(b) may explain the continuous speedup increase when the number of attributes grows. It shows the breakdown of execution time of the most time-demanding algorithm's parts for 100 thousand samples. Each time when the dataset dimension increases, the parallel overhead (data flattening and data transfer) is less important. At the same time, the time contribution of the `calc_models` procedure increases (as it is the only activity directly related with the number for attributes), while other GPU procedures have less affect on the induction time.

6.3. Discussion on GPU-based approaches

Other GPU-supported parallelizations of DT inducers can be found in the literature. Some of them concern the greedy (top-down) approach [51,52]. Typically, they parallelized the search of optimal tests (over attributes and nodes) in successive tree levels during the recursive partitioning of dataset samples. They speeded up the induction up to 55 times. Although a similar parallel decomposition strategy (over nodes) could be adapted in the EA-based inducers, it would not be scalable with respect to the dataset size. In the global inducer, the size of trees is usually smaller as well as there is a need for multiple redistributions of training samples over many different trees. Moreover, the previously published greedy solutions considered classification problems (in the leaves classes were located, not regression models). Thus, the calculation burden was concentrated on finding optimal splits according to a given optimality measure, but not on the regression models and finally fitness function value calculations.

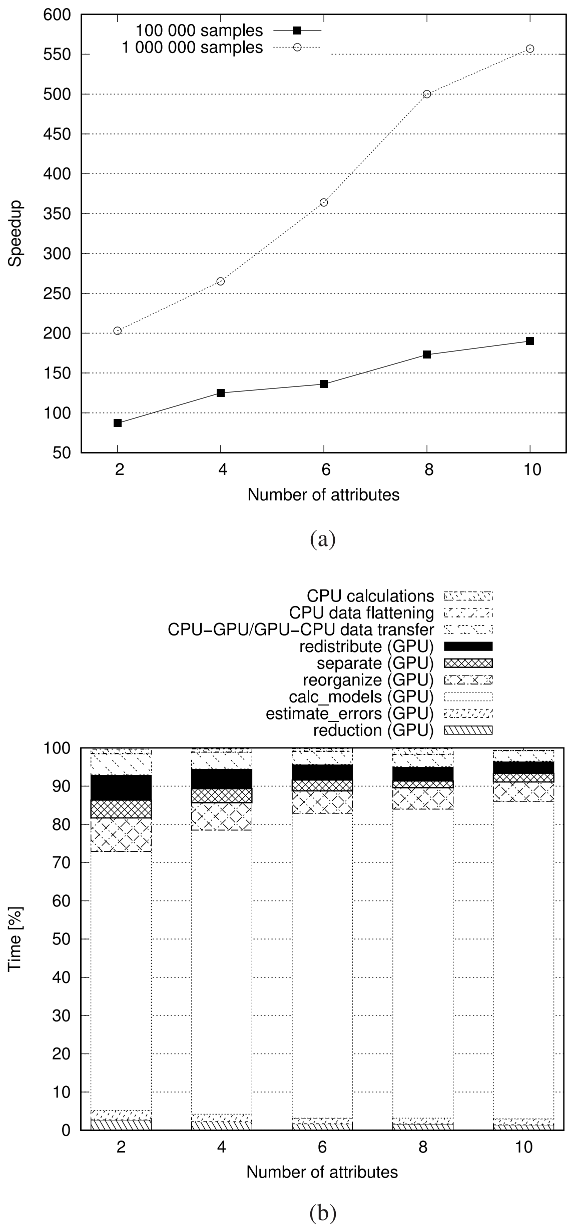

Fig. 18. The influence of the number of attributes on the time performance of the cuGMT using GeForce RTX 2080 Ti GPU: (a) mean speedup for 2, 4, 6, 8 and 10 attributes for the *Armchair* dataset (100 000 and 1 000 000 samples), (b) the run-time breakdown (average time as a percentage of total run-time) for *Armchair* dataset (100 000 samples).

In the literature, we can also find GPU-accelerated inducers of ensemble (multi-tree) classifiers, e.g., Random Forests [53,54] or gradient boosting DTs [55,56]. They used the simple top-down recursive partition approach to build each DT, while the idea was to combine the predictions of multiple trees together (as it should better predict than a single tree). The GPU-supported accelerations divided calculations in different DTs over CUDA cores. If only such a decomposition was applied [53], the provided speedup was up to 30×. In addition, it was achieved only for a large number of DTs. When additional decomposition strategies were added (similar to being used in the single tree inducers [51,52]) [54,55] and multiple GPUs were exploited [56], the induction time was accelerated more (even to 300 times). For the EA-based inducer, we also tested different decomposition techniques. One of the studied cases was a hybrid approach: both individuals and the dataset samples were spread over CUDA cores. However, we had to significantly increase (artificially) the population size, to achieve satisfying speedups. Despite that, the solution proposed in this paper (two-level dataset decomposition) was much more efficient, particularly for large datasets.

Swarm intelligence (SI) based DT inducers [37] also realize a global-search strategy. Similarly to EA-based solutions, they use population-based and iterative calculations. Thus, they suffer similar problems, like ever-growing computational demands when applied to complex problems. Although, to the best of our knowledge, there was no direct study on GPU-supported SI-based DT inducers, many GPU-accelerated implementations of SI algorithms were proposed, e.g., for ant colony optimization (ACO) [78], bee swarm optimization (BSO) [79] and particle swarm optimization (PSO) [80]. They can be categorized into four major categories: naive parallel model, multiphase parallel model, all-GPU parallel model and multiswarm parallel one [59]. Most of them are primarily focused on offloading the fitness evaluations onto the GPU for parallel running (as our solution also does) because it is the most time-consuming operation. On the conceptual level, our parallelization fits the most into two categories: multiphase parallel model and all-GPU parallel one.

In multiphase parallel models, not only a naive parallel fitness evaluation is applied but also the remaining parts of the algorithm (e.g., velocity and position update in PSO [81]) further exploit the GPU parallelism. Similarly, in our solution, multiple tasks (phases) are GPU-powered, e.g., fitness evaluation, dipoles searching or regression model construction, providing better leverage of GPU computing resources and thus, better speedup. Results in Figs. 15, 17 and 18(b) show that the execution time of remaining serial (not parallelized) parts of the algorithm is negligible.

On the other side, all-GPU parallel models also move all serial code onto the GPU. This way, the overhead of communication between CPU–GPU/GPU–CPU and frequent kernel launch can be decreased significantly, and the overall performance can get improved. However, combining multiple kernels into a single one may lead to difficulties with data synchronization between different thread blocks. To tackle this issue, all threads can be organized into one block, as it was done in a GPU-based PSO [82]. Such a mechanism limits the usage of GPU resources to a single streaming multi-processor and can be reasonable only for a small swarm size. In our inducer, we also move most calculations onto the GPU, while the CPU mainly controls the evolutionary process. However, in order to provide data synchronization between subsequent phases and use full GPU resources, we decided to divide calculations into multiple kernels. What we could improve in the future is to move a serial code of genetic operators onto the GPU. Then, the transfer of DTs between CPU–GPU would not be needed. Results in Figs. 15, 17 and 18(b) suggest that then the communication overhead would decrease. It could be reasonable for smaller datasets. However, no meaningful performance improvement would be observed for large-scale data because then the communication overhead is negligible.

7. Conclusion

The paper concerns an evolutionary approach in induction of model trees. Such a population-based and iterative way of induction provides global exploration that searches significantly simpler decision trees and at least as accurate as induced by the state-of-the-art greedy alternatives. However, the global approach is much more computationally demanding, and thus hard to apply to big data mining directly. We address this problem and propose a GPU-supported inducer. Experimental validation is performed with real-life and artificial datasets. Both the data size and dimension are investigated. The results clearly show that the computational barrier can be overcome. The GPU-based solution speeds up the inducer significantly (induction time decreases from hours to minutes or from days to hours). The global induction of model trees is now also possible for large-scale data (in a reasonable time), which until now was reserved for greedy methods. Moreover, it is achieved without costly computer clusters but using only a single workstation with at least a medium-class GPU.

We see many promising directions for future research. One of these directions focuses on speeding up the solution further. For example, a multi-GPU approach, a Spark-based one, or a hybrid CPU/GPU [83] may provide additional time performance improvement and make the evolutionary approach as fast as top-down inducers. Another direction is to apply a reuse strategy that was shown to efficiently accelerate the evolutionary induction of classification trees [9]. Moreover, such an approach may allow us to observe and better understand the evolutionary process, e.g., to follow the similarity factor between trees in subsequent generations. On the other hand, a multi-tree representation would also be an attractive field of research. It assumes that similar parts of trees would be represented by partially shared structures in memory. It would provide an additional way to observe the dynamism of evolution.

CRediT authorship contribution statement

Krzysztof Jurczuk: Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Marcin Czajkowski:** Data curation, Software, Writing – original draft. **Marek Kretowski:** Conceptualization, Funding acquisition, Validation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

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

Appendix A. Pseudo code of main procedures/kernels of the GPU-accelerated approach

Listing 1: Pseudo code of the evolutionary loop and CPU–GPU interaction.

```
procedure evaluateIndividualAtGPU
input: indiv
output: updatedIndiv
begin
 //data (decision tree) flattening (SoA) and coping to GPU
 indivTab=copyTreeToTable(indiv);
 allocateMemoryAtGPU(indivTab); //if no optimization
 sendDataToGPU(indivTab);

 redistribute<<N_BLOCKS, N_THREADS>>(indivTab,...);
 cudaDeviceSynchronize();

 separate(objToLeafTab, objIdxTab, ...);

 reorganize<<N_BLOCKS, N_THREADS>>(objToLeafTabSorted, ...);
 cudaDeviceSynchronize();

 recvDataFromGPU(nObjsInNodeTab);
 indiv.setNObjsInNodes(nObjsInNodeTab);

 calc_models(indiv.getRoot(), datasetTabMatrixX, ...);

 estimate_errors<<N_BLOCKS, N_THREADS>>(datasetTabMatrixC,..);
 cudaDeviceSynchronize();

 reduction<<1, N_BLOCKS>>(indivErrorTabG, ...)
 cudaDeviceSynchronize();

 recvDataFromGPU(errors);
 recvDataFromGPU(dipoles);

 updatedIndiv=updateIndividual(indiv, errors, dipoles);
 deallocateMemoryAtGPU(indivTab); //if no optimization
end

procedure main
  ...
  //if optimization
  //allocate memory for individuals
  //init and allocate memory for CUSOLVER, CUBLAS

  //transform dataset to SoA and coping to GPU
  datasetTab=copyDatasetToSoA(dataset);
  allocateMemoryAtGPU(datasetTab);
  sendDataToGPU(datasetTab);

  createInitPopulation();
  evaluatePopulation();
  selection();

  //evolutionary loop
  while !stopCondition do

    //loop over pairs of individuals
    for indivPair in idividualPairs do
      crossover(indivPair);
      evaluateIndividualAtGPU(indivPair.indiv1);
      evaluateIndividualAtGPU(indivPair.indiv2);
    end for

    //loop over individuals
    for indiv in individuals do
      mutation(indiv);
      evaluateIndividualAtGPU(indiv);
    end for
    selection();
  end while

  deallocateMemoryAtGPU(datasetTab);
  ...
end
```

Listing 2: Pseudo code of the GPU-supported procedures and kernels.

```
__global__ procedure redistribute
input: indivTab, datasetTab
output: objToLeafTab, nObjsInNodeTab
begin
 int nObjsToCheck=nObjs/gridDim.x/blockDim.x;
 int startingObjIdx=blockIdx.x*nObjs/gridDim.x+threadIdx.x;

 int nodeIdx=0;
 int objIdx=startingObjIdx;
 Sample sample;
 Node node;

 //redistribute samples among leaves
 for i = 1 to nObjsToCheck do
  sample=dataset[objIdx]; //first attribute value
  nodeIdx=0; //root index
  while true do
   node=indivTab[nodeIdx]; //attribute in a node
   if node is a leaf then
    //remember leaf index
    objToLeafTab[objIdx]=nodeIdx;
    //increment number of samples in a leaf
    atomicAdd(&nObjsInNodeTab[nodeIdx], 1);
    break;
   else
    //move to one of the children nodes
    if sample[attrShift(node)] > node[valueShift] then
     nodeIdx=nodeIdx*2+1; //left child
    else
     nodeIdx=nodeIdx*2+2; //right child
    end if
   end if
  end while
  objIdx+=blockDim.x;
 end for
end

procedure separate
input: objToLeafTab, objIdxTab={0,1,2,...,nObjs-1}
output: objToLeafTabSorted, objIdxTabSorted
begin
 setTempStorage(tempStor, tempStorSize);//if no optimization
 allocateTempStorage();//if no optimization

 //sorting
 cub::DeviceRadixSort:SortPairs(tempStor, tempStorSize,
       objToLeafTab, objToLeafTabSorted, objIdxTab,
       objIdxTabSorted);

 deallocTempStorage();//if no optimization
end

__global__ procedure reorganize
input: objToLeafTabSorted, objIdxTabSorted
ouput: datasetTabMatrixX, datasetTabMatrixY
begin
 int nObjsToMove=nObjs/gridDim.x/blockDim.x;
 int startingPos=blockIdx.x*nObjs/gridDim.x+threadIdx.x;

 int objPos=startingPos;
 Sample sample;
 Node leaf;

 //reorganize samples into matrices for MLR
 for i=1 to nObjsToMove do
  leafIdx=objToLeafTabSorted[objPos];
  objIdx=objIdxTabSorted[objPos];

  sample=datasetTab[objIdx]; //first attribute value
  leaf=indivTab[leafIdx];
  //if the sample is in a leaf to be updated
  if leaf[updateShift] then
   //starting point for sample attributes
   leafMatrixX=sampleMatrixXShift(leafIdx,objIdx);
   datasetTabMatrixX[leafMatrixX]=1.0;
   int k=1;
   //rewrite attribute values
   for j=1 to nAttrs-1 do
    //if the attribute is currently used in a model
    if leaf[onoffShift(j)] then
     datasetTabMatrixX[leaftMatrixX+leafAttrShift(k,leafIdx)]=
           sample[attrShift(j)];
     k++;
    end if
   end for
   datasetTabMatrixY[objIdx]=sample[attrShift(nAttrs)];
  end if
  objPos+=blockDim.x;
 end for
end

procedure calc_models
input: node, datasetTabMatrixX, datasetTabMatrixY
output: datasetTabMatrixC //model coefficients
begin
 const float alpha=1;

 if node is a leaf then
  if node is modified then
   int m=node->nSamples;
   int n=node->nModelAttrs;
   X=datasetTabMatrixX+shiftMLRMatrix(node,n);
   Y=datasetTabMatrixY+shiftMLRMatrix(node);
   const int ldx=m, ldy=m;

   // initialize the CUSOLVER, CUBLAS, if no optimization
   initMLRLibraries();
   allocateMemoryAtGPU(tau,work,lwork,devInfo,mt_cublasH);

   //calculate the size of work buffer needed
   cusolverDnSgeqrf_bufferSize(handle, m, n, X, ldx, lwork);

   //MLR calculations
   //step1: X = QR with CUSOLVER
   cusolverDnSgeqrf(handleS, m, n, X, ldx, tau, work, lwork,
         devInfo);
   cudaDeviceSynchronize();

   //step2: z = (Q^T)Y with CUSOLVER, z is m x 1
   cusolverDnSormqr(handleS, CUBLAS_SIDE_LEFT, CUBLAS_OP_T, m,
         1, MIN(m,n), X, ldx, tau, Y, ldy, work, lwork,devInfo);
   cudaDeviceSynchronize();

   //step3: Solve RC = z for C with CUBLAS, C is n x 1
   cublasStrsm(handleB, CUBLAS_SIDE_LEFT,
         CUBLAS_FILL_MODE_UPPER, CUBLAS_OP_N,
         CUBLAS_DIAG_NON_UNIT, n, 1, alpha, X, ldx, Y, ldy);
  end if
 else
  if node->left then //has a left child
   calc_models(node->left);
  end if
  if node->right then //has a right child
   calc_models(node->right);
  end if
 end if
 datasetTabMatrixC=datasetTabMatrixY;
end

__global__ procedure estimate_errors
input: datasetTabMatrixC
output: indivErrorTabG, indivDipolesTabG
begin
 int nObjsToCalc=nObjs/gridDim.x/blockDim.x;
 int startingPos=blockIdx.x*nObjs/gridDim.x+threadIdx.x;

 __shared__ float indivErrorTab[N_NODES];
 __shared__ unsigned int indivDipolesTab[N_NODES];

 int objPos=startingPos;
 Sample sample;
 Node leaf;
 C=datasetTabMatrixC;
 for i=1 to nObjsToCalc do
  leafIdx=objToLeafTabSorted[objPos];
  objIdx=objIdxTabSorted[objPos];

  sample=datasetTab[objIdx]; //first attribute value
  leaf=indivTab[leafIdx];
  startLeafMLR=leaf[startMLRShift];

  //calc error
  int prediction=C[startLeafMLR];
  int k=1;
  for j=1 to nAttrs-1 do
   //if the attribute is currently used in a model
   if leaf[onoffShift(j)] then
    prediction+=C[startLeafMLR+k]*sample[attrShift(j)];
    k++;
   end if
  end for
  errPow2=(prediction-sample[attrShift(nAttrs)])^2;

  //sum of errors
  atomicAdd(&indivErrorTab[leafIdx], errPow2);

  //sample for a dipole
  atomicCAS(&indivDipolesTab[leafIdx], 0, objIdx);
  objPos+=blockDim.x;
 end for

 __syncthreads();
 //copy data from shared to global memory
 if threadIdx.x==0 then
  for i=0 to N_NODES-1 do
   indivErrorTabG[blockIdx.x*N_NODES+i]=indivErrorTab[i];
   indivDipolesTabG[blockIdx.x*N_NODES+i]=indivDipolesTab[i];
  end for
  //or by memcpy
 end if
end

__global__ procedure reduction
input: indivErrorTabG, indivDipolesTabG
output: indivDipolesTabG, indivErrTabG
begin
 __shared__ float indivErrTab_Sum[N_NODES];
 __shared__ unsigned int indivDipolesTabMin[N_NODES];
 __shared__ unsigned int indivDipolesTabMax[N_NODES];

 //reduce errors
 for i=0 to N_NODES-1 do
  atomicAdd(&indivErrTab_Sum[i], indivErrorTabG[threadIdx.x*
        N_NODES+i]);
```

```
end for

//reduce dipoles
predTab = datasetTab[attrShift(nAttrs)];
for i=0 to N_NODES-1 do
 if indivDipolesTabG[threadIdx.x*N_NODES+i] is not empty
  oldMinObjIdx=indivDipolesTabMin[i];
  oldMaxObjIdx=indivDipolesTabMax[i];
  newObjIdx=indivDipolesTabG[threadIdx.x*N_NODES+i];

  if predTab[oldMinObjIdx] > predTab[newObjIdx] then
   atomicExch(&indivDipolesTabMin[i], newObjIdx);
  end if
  if predTab[oldMaxObjIdx] < predTab[newObjIdx] then
   atomicExch(&indivDipolesTabMax[i], newObjIdx);
  end if
 end if
end for

__syncthreads();
//propagate
if threadIdx.x==0 then
 //propagate error
 for i=N_NODES-1 to 1 do
  if i%2 then
   indivErrTab_Sum[i/2-1]+=indivErrTab_Sum[i];
  else
   indivErrTab_Sum[i/2]+=indivErrTab_Sum[i];
  end if
 end for
 //propagate dipoles
 for i=N_NODES/2-1 to 0 do
  if rand(1) then
   indivDipolesTabMin[i]=indivDipolesTabMin[2*i+1];
   indivDipolesTabMax[i]=indivDipolesTabMax[2*i+1];
  else
   indivDipolesTabMin[i]=indivDipolesTabMin[2*i+2];
   indivDipolesTabMax[i]=indivDipolesTabMax[2*i+2];
  end if
 end for

 //copy data from shared to global memory
 //in analogy like in estimate_errors kernel
 //indivDipolesTabG, indivErrTabG
end if
end
```

References

[1] T. Condie, P. Mineiro, N. Polyzotis, M. Weimer, Machine learning on big data, in: 2013 IEEE 29th International Conference on Data Engineering (ICDE), 2013, pp. 1242–1244.

[2] S. Kotsiantis, Decision trees: a recent overview, Artif. Intell. Rev. 39 (4) (2013) 261–283..

[3] W.Y. Loh, Fifty years of classification and regression trees, Internat. Statist. Rev. 82 (3) (2014) 329–348.

[4] R.C. Barros, M.P. Basgalupp, A.C. De Carvalho, A.A. Freitas, A survey of evolutionary algorithms for decision-tree induction, IEEE Trans. SMC C 42 (3) (2012) 291–312.

[5] M. Kretowski, Evolutionary Decision Trees in Large-Scale Data Mining, Springer, 2019.

[6] R.C. Barros, D.D. Ruiz, M.P. Basgalupp, Evolutionary model trees for handling continuous classes in machine learning, Inform. Sci. 181 (5) (2011) 954–971.

[7] K. Jurczuk, M. Czajkowski, M. Kretowski, Multi-GPU approach to global induction of classification trees for large-scale data mining, Appl. Intell. 51 (2021) 5683–5700.

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

[9] K. Jurczuk, M. Czajkowski, M. Kretowski, Fitness evaluation reuse for accelerating GPU-based evolutionary induction of decision trees, Int. J. High Perform. Comput. Appl. 35 (1) (2021) 20–32.

[10] D. Storti, M. Yurtoglu, CUDA for Engineers : An Introduction to High-Performance Parallel Computing, Addison-Wesley, New York, 2016.

[11] N. Wilt, CUDA Handbook: A Comprehensive Guide to GPU Programming, Addison-Wesley, Upper Saddle River, NJ, 2013.

[12] M. Czajkowski, K. Jurczuk, M. Kretowski, Hybrid parallelization of evolutionary model tree induction, in: L. Rutkowski, M. Korytkowski, R. Scherer, R. Tadeusiewicz, L.A. Zadeh, J.M. Zurada (Eds.), Artificial Intelligence and Soft Computing - ICAISC 2016, in: Lecture Notes in Computer Science, vol. 9692, Springer, Cham, 2016, pp. 370–379.

[13] M. Czajkowski, M. Kretowski, Evolutionary induction of global model trees with specialized operators and memetic extensions, Inform. Sci. 288 (2014) 153–173.

[14] M. Czajkowski, M. Kretowski, The role of decision tree representation in regression problems – an evolutionary perspective, Appl. Soft Comput. 48 (2016) 458–475.

[15] M. Czajkowski, M. Czerwonka, M. Kretowski, Cost-sensitive global model trees applied to loan charge-off forecasting, Decis. Support Syst. 74 (2015) 57–66.

[16] M. Czajkowski, M. Kretowski, Decision tree underfitting in mining of gene expression data. An evolutionary multi-test tree approach, Expert Syst. Appl. 137 (2019) 392–404.

[17] A. Mukhopadhyay, U. Maulik, S. Bandyopadhyay, C.A.C. Coello, A survey of multiobjective evolutionary algorithms for data mining: Part I, IEEE Trans. Evol. Comput. 18 (1) (2014) 4–19.

[18] L. Rokach, O. Maimon, Data Mining with Decision Trees: Theory and Applications, World Scientific, Singapore, 2014.

[19] R. Rivera-Lopez, J. Canul-Reich, E. Mezura-Montes, M.A. Cruz-Chávez, Induction of decision trees as classification models through metaheuristics, Swarm Evol. Comput. 69 (2022) 101006.

[20] J.R. Quinlan, Learning with Continuous Classes, World Scientific, 1992, pp. 343–348.

[21] F.M. Ortuño, O. Valenzuela, B. Prieto, M.J. Saez-Lara, C. Torres, H. Pomares, I. Rojas, Comparing different machine learning and mathematical regression models to evaluate multiple sequence alignments, Neurocomputing 164 (2015) 123–136.

[22] A. Fakhari, A.M.E. Moghadam, Combination of classification and regression in decision tree for multi-labeling image annotation and retrieval, Appl. Soft Comput. 13 (2) (2013) 1292–1302.

[23] J. Liu, C. Sui, D. Deng, J. Wang, B. Feng, W. Liu, C. Wu, Representing conditional preference by boosted regression trees for recommendation, Inform. Sci. 327 (2016) 1–20.

[24] L. Hyafil, R.L. Rivest, Constructing optimal binary decision trees is NP-complete, Inform. Process. Lett. 5 (1) (1976) 15–17.

[25] L. Rokach, O. Maimon, Top-down induction of decision trees classifiers - a survey, IEEE Trans. Syst. Man Cybern. C (Appl. Rev.) 35 (4) (2005) 476–487.

[26] F. Esposito, D. Malerba, G. Semeraro, A comparative analysis of methods for pruning decision trees, IEEE Trans. Pattern Anal. Mach. Intell. 19 (5) (1997) 476–491.

[27] L. Breiman, J.H. Friedman, R.A. Olshen, C.J. Stone, Classification and regression trees, Wadsworth, 1984.

[28] Q. Liu, X. Li, H. Liu, Z. Guo, Multi-objective metaheuristics for discrete optimization problems: A review of the state-of-the-art, Appl. Soft Comput. 93 (2020) 106382.

[29] J.R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, 1992.

[30] J.R. Koza, Concept formation and decision tree induction using the genetic programming paradigm, in: H.-P. Schwefel, R. Männer (Eds.), Parallel Problem Solving from Nature, Springer Berlin Heidelberg, Berlin, Heidelberg, 1991, pp. 124–128.

[31] J. Petke, S.O. Haraldsson, M. Harman, W.B. Langdon, D.R. White, J.R. Woodward, Genetic improvement of software: A comprehensive survey, IEEE Trans. Evol. Comput. 22 (3) (2018) 415–432.

[32] G. Fan, J.B. Gray, Regression tree analysis using TARGET, J. Comput. Graph. Statist. 14 (1) (2005) 206–218.

[33] B. Biswal, H. Behera, R. Bisoi, P. Dash, Classification of power quality data using decision tree and chemotactic differential evolution based fuzzy clustering, Swarm Evol. Comput. 4 (2012) 12–24.

[34] H.-G. Beyer, S. Finck, T. Breuer, Evolution on trees: On the design of an evolution strategy for scenario-based multi-period portfolio optimization under transaction costs, Swarm Evol. Comput. 17 (2014) 74–87.

[35] F.E. Otero, A.A. Freitas, C.G. Johnson, Inducing decision trees with an ant colony optimization algorithm, Appl. Soft Comput. 12 (11) (2012) 3615–3626.

[36] J.E. Fieldsend, Optimizing decision trees using multi-objective particle swarm optimization, in: C.A.C. Coello, S. Dehuri, S. Ghosh (Eds.), Swarm Intelligence for Multi-Objective Problems in Data Mining, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 93–114.

[37] I. Bida, S. Aouat, A new approach based on bat algorithm for inducing optimal decision trees classifiers, in: A. Rocha, M. Serrhini (Eds.), Information Systems and Technologies To Support Learning, Springer International Publishing, Cham, 2019, pp. 631–640.

[38] D. Yuen, J. Wang, L. Johnsson, C.-H. Chi, Y. Shi, GPU Solutions to Multi-Scale Problems in Science and Engineering, Springer, 2013.

[39] Y. Djenouri, D. Djenouri, Z. Habbas, Intelligent mapping between GPU and cluster computing for discovering big association rules, Appl. Soft Comput. 65 (2018) 387–399.
[40] A. Cano, A survey on graphic processing unit computing for large-scale data mining, Wiley Interdiscip. Rev.: Data Min. Knowl. Discov. 8 (1) (2018) e1232.
[41] W.-B. Qiao, J.-C. Créput, Component-based 2-/3-dimensional nearest neighbor search based on Elias method to GPU parallel 2D/3D Euclidean Minimum Spanning Tree Problem, Appl. Soft Comput. 100 (2021) 106928.
[42] D.M. Chitty, Improving the performance of GPU-based genetic programming through exploitation of on-chip memory, Soft Comput. 20 (2) (2016) 661–680.
[43] A. Cano, A. Zafra, S. Ventura, Speeding up multiple instance learning classification rules on GPUs, Knowl. Inf. Syst. 44 (1) (2015) 127–145.
[44] D.M. Chitty, Fast parallel genetic programming: multi-core CPU versus many-core GPU, Soft Comput. 16 (10) (2012) 1795–1814.
[45] A. Grama, G. Karypis, V. Kumar, A. Gupta, Introduction to Parallel Computing, Addison-Wesley, 2003.
[46] S. Tsutsui, P. Collet, Massively Parallel Evolutionary Computation on GPGPUs, Springer, Berlin, 2013.
[47] T.V. Luong, N. Melab, E.-G. Talbi, GPU-based island model for evolutionary algorithms, in: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, GECCO '10, 2010, pp. 1089–1096.
[48] M.A. Franco, N. Krasnogor, 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, 2010, pp. 1039–1046.
[49] N. Soca, J.L. Blengio, M. Pedemonte, P. Ezzatti, PUGACE, a cellular evolutionary algorithm framework on GPUs, in: IEEE Congress on Evolutionary Computation, 2010, pp. 1–8.
[50] M.A. Franco, J. Bacardit, Large-scale experimental evaluation of GPU strategies for evolutionary machine learning, Inform. Sci. 330 (C) (2016) 385–402.
[51] W.T. Lo, Y.S. Chang, R.K. Sheu, C.C. Chiu, S.M. Yuan, CUDT: A CUDA based decision tree algorithm, Sci. World J. 2014 (2014).
[52] D. Strnad, A. Nerat, Parallel construction of classification trees on a GPU, Concurr. Comput.: Pract. Exper. 28 (5) (2016) 1417–1436.
[53] H. Grahn, N. Lavesson, M.H. Lapajne, D. Slat, CudaRF: A CUDA-based implementation of random forests, in: 2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA), 2011, pp. 95–101.
[54] D. Marron, A. Bifet, G.D.F. Morales, Random forests of very fast decision trees on GPU for mining evolving big data streams, in: Proceedings of the Twenty-First European Conference on Artificial Intelligence, in: ECAI'14, Amsterdam, The Netherlands, 2014, pp. 615–620.
[55] M. Rory, F. Eibe, Accelerating the xgboost algorithm using GPU computing, PeerJ Comput. Sci. 3 (2017) e127.
[56] Z. Wen, J. Shi, B. He, J. Chen, K. Ramamohanarao, Q. Li, Exploiting GPUs for efficient gradient boosting decision tree training, IEEE Trans. Parallel Distrib. Syst. 30 (12) (2019) 2706–2717.
[57] K. Jurczuk, M. Czajkowski, M. Kretowski, GPU-accelerated evolutionary induction of regression trees, in: C. Martín-Vide, R. Neruda, M.A. Vega-Rodríguez (Eds.), Theory and Practice of Natural Computing, in: LNCS, vol. 10687, Springer, 2017, pp. 87–99.
[58] D. Reska, K. Jurczuk, M. Kretowski, Evolutionary induction of classification trees on spark, in: L. Rutkowski, R. Scherer, M. Korytkowski, W. Pedrycz, R. Tadeusiewicz, J.M. Zurada (Eds.), Artificial Intelligence and Soft Computing, in: LNCS, vol. 10841, Springer, 2018, pp. 514–523.
[59] Y. Tan, K. Ding, A survey on GPU-based implementation of swarm intelligence algorithms, IEEE Trans. Cybern. 46 (9) (2016) 2028–2041.
[60] Z. Michalewicz, Genetic Algorithms + Data Structures=Evolution Programs, Springer, 1996.
[61] M. Črepinšek, S.-H. Liu, M. Mernik, Exploration and exploitation in evolutionary algorithms: A survey, ACM Comput. Surv. 45 (3) (2013) 35.
[62] M. Kretowski, M. Grzes, Evolutionary induction of mixed decision trees, Int. J. Data Warehous. Min. 3 (4) (2007) 68–82.
[63] R.O. Duda, P.E. Hart, D.G. Stork, Pattern Classification, second ed., Wiley-Interscience, 2000.
[64] M. Czajkowski, M. Kretowski, A multi-objective evolutionary approach to Pareto-optimal model trees, Soft Comput. 23 (5) (2019) 1423–1437.
[65] N. Satish, M. Harris, M. Garland, Designing efficient sorting algorithms for manycore GPUs, in: Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing, IPDPS '09, 2009, pp. 1–10.
[66] D.P. Singh, I. Joshi, J. Choudhary, Survey of GPU based sorting algorithms, Int. J. Parallel Program. 46 (6) (2018) 1017–1034.
[67] D. Merrill, CUB V1.8.0 a library of warp-wide, block-wide, and device-wide GPU parallel primitives, NVIDIA Res. (2020) URL http://nvlabs.github.io/cub/.
[68] G. Golub, C. Van Loan, Matrix Computations, third ed., The Johns Hopkins University Press, Baltimore, 1996.
[69] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes: The Art of Scientific Computing, third ed., Cambridge University Press, New York, 2007.
[70] G. Mei, H. Tian, Impact of data layouts on the efficiency of GPU-accelerated IDW interpolation, SpringerPlus 5 (1) (2016) 1–18.
[71] R. Strzodka, Abstraction for AoS and SoA layout in C++, in: W.W. Hwu (Ed.), GPU Computing Gems Jade Edition, Morgan Kaufmann, 2012, pp. 429–441.
[72] L. Torgo, Regression datasets, URL https://www.dcc.fc.up.pt/~ltorgo/Regression/DataSets.html.
[73] D. Dua, E. Karra Taniskidou, UCI machine learning repository, 2017, URL http://archive.ics.uci.edu/ml.
[74] I.H. Witten, E. Frank, M.A. Hall, C.J. Pal, The WEKA Workbench. Online Appendix for "Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques", Morgan Kaufmann, 2016.
[75] NVIDIA, NVIDIA developer zone - CUDA toolkit documentation, 2020, URL https://docs.nvidia.com/cuda/.
[76] O. Sagi, L. Rokach, Ensemble learning: A survey, WIREs Data Min. Knowl Discov. 8 (4) (2018) e1249.
[77] G. Schwarz, Estimating the dimension of a model, Ann. Statist. 6 (2) (1978) 461–464.
[78] J.M. Cecilia, A. Llanes, J.L. Abellan, J. Gomez-Luna, L.-W. Chang, W.-M.W. Hwu, High-throughput Ant Colony Optimization on graphics processing units, J. Parallel Distrib. Comput. 113 (2018) 261–274.
[79] Y. Djenouri, P. Fournier-Vigerand, J.C.-W. Lin, D. Djenouri, A. Belhadi, GPU-Based swarm intelligence for Association Rule Mining in big databases, Intell. Data Anal. 23 (1) (2019) 57–76.
[80] H. Liu, Z. Wen, W. Cai, FastPSO: TOwards efficient swarm intelligence algorithm on GPUs, in: 50th International Conference on Parallel Processing, in: ICPP 2021, 2021.
[81] Y. Zhou, Y. Tan, GPU-based parallel particle swarm optimization, in: 2009 IEEE Congress on Evolutionary Computation, 2009, pp. 1493–1500.
[82] L. Mussi, F. Daolio, S. Cagnoni, Evaluation of parallel particle swarm optimization algorithms within the CUDA™ architecture, Inform. Sci. 181 (20) (2011) 4642–4657.
[83] M. Gowanlock, Hybrid KNN-join: Parallel nearest neighbor searches exploiting CPU and GPU architectural features, J. Parallel Distrib. Comput. 149 (2021) 119–137.