Applied Intelligence (2021) 51:5683–5700
https://doi.org/10.1007/s10489-020-01952-5

Multi-GPU approach to global induction of classification trees for large-scale data mining

Krzysztof Jurczuk[1] · Marcin Czajkowski[1] · Marek Kretowski[1]

Accepted: 16 September 2020 / Published online: 14 January 2021
© The Author(s) 2021

Abstract
This paper concerns the evolutionary induction of decision trees (DT) for large-scale data. Such a global approach is one of the alternatives to the top-down inducers. It searches for the tree structure and tests simultaneously and thus gives improvements in the prediction and size of resulting classifiers in many situations. However, it is the population-based and iterative approach that can be too computationally demanding to apply for big data mining directly. The paper demonstrates that this barrier can be overcome by smart distributed/parallel processing. Moreover, we ask the question whether the global approach can truly compete with the greedy systems for large-scale data. For this purpose, we propose a novel multi-GPU approach. It incorporates the knowledge of global DT induction and evolutionary algorithm parallelization together with efficient utilization of memory and computing GPU's resources. The searches for the tree structure and tests are performed simultaneously on a CPU, while the fitness calculations are delegated to GPUs. Data-parallel decomposition strategy and CUDA framework are applied. Experimental validation is performed on both artificial and real-life datasets. In both cases, the obtained acceleration is very satisfactory. The solution is able to process even billions of instances in a few hours on a single workstation equipped with 4 GPUs. The impact of data characteristics (size and dimension) on convergence and speedup of the evolutionary search is also shown. When the number of GPUs grows, nearly linear scalability is observed what suggests that data size boundaries for evolutionary DT mining are fading.

Keywords Evolutionary data mining · Big data · Decision trees · Parallel computing · Graphics processing unit (GPU) · CUDA

1 Introduction

In the last decade, the term Big Data has become extremely popular in business, industry, and science [14]. It refers to storing and handling of large or complex datasets that are almost impossible to manage using traditional tools. With such ever-growing volumes and velocity of data, new mining and knowledge discovery techniques need to be developed. Extracting and retrieving desired information and patterns from enormous quantity of data, known as Big Data Mining, give huge opportunities as well as great challenges [72].

This paper focuses on decision trees (DTs) which are one of the most established and well explored machine learning techniques. During the last 50 years of their applications [42], DTs have been mainly induced using greedy heuristics like a top-down approach [54]. Recent involvement of evolutionary algorithms (EAs) into the trees induction [2] can be seen as a breath of fresh air. Their main advantage is the global approach in which a tree structure, tests in internal nodes and predictions in leaves are searched simultaneously [38]. As a result, the generated trees are significantly simpler and at least as accurate as the greedy alternatives.

However, direct addressing the evolutionary DT induction to big data may be unachievable, as the population-based and iterative methods can be simply too demanding. To make it feasible, some forms of parallel or distributed processing are required. For this reason, we have

✉ Krzysztof Jurczuk
k.jurczuk@pb.edu.pl

Marcin Czajkowski
m.czajkowski@pb.edu.pl

Marek Kretowski
m.kretowski@pb.edu.pl

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

focused on graphics processing units (GPUs) to provide parallelization of evolutionary DT induction. GPUs of modern graphics cards/accelerators are known to afford massive computing resources at a relatively low cost. NVIDIA CUDA programming model [58], which supports general-purpose computation on GPUs (GPGPU), is applied.

In this paper, we design and implement a novel multi-GPU approach and attempt to define the scalability bounds of global induction of DTs. The main EA loop (selection, genetic operators, etc.) is performed on a CPU, while the fitness related calculations are isolated and performed on GPUs. This way, the GPUs handle compute-intensive jobs, while the evolutionary flow control and communication tasks are left to the CPU. We apply a data-parallel decomposition strategy [27]. Data is spread, first, between the GPUs and next, inside each GPU over computational cores. With efficient memory and device resources utilization as well as by incorporating the knowledge of DT induction and evolutionary parallelization, we manage to achieve stunning improvement in the algorithm speed. Moreover, the ability to analyze large-scale data of various characteristics has become possible.

The proposed approach is applied to a system called Global Decision Tree (GDT) that provides a possibility for evolutionary induction of various classification, regression and model trees [38]. The GDT system can be applied in many real-life applications, such as finance [17] or medicine [20]. The acceleration of the GDT would allow large-scale data processing and could provide a competitive solution to greedy state-of-the-art algorithms also in terms of DT induction time.

This paper is organized as follows. Section 2 provides a brief background on DTs and surveys the most closely related works. This section also includes a description of the GDT system. In the next section, our multi-GPU approach as well as its implementation are described in detail. Experimental validation of the proposed solution on artificially generated and real-life datasets is presented in Section 4. In the last section, we conclude the paper and outline possible future work.

2 Evolutionary induction of decision trees

In the beginning, basic information about decision trees, methods in their induction and evolutionary algorithms are presented. Then, related works concerning the parallelization of evolutionary algorithms and induction of decision trees are given. Finally, the Global Decision Tree system is briefly described.

2.1 Decision trees

Decision trees (DTs) [36] represent one of the key knowledge discovery methods [1, 6]. They are directed acyclic graphs built of nodes and branches [42]. The branches connect the nodes in a hierarchical manner. The first node is a root node (see Fig. 1). It does not have any parent/predecessor node. There are internal nodes and terminal nodes. Each internal node has at least two child/descendant nodes. Moreover, any internal node is associated with a test that can be performed on one or multiple attributes. The outcome of the test is represented by a branch.

The terminal nodes are called leaves and each such a node keeps the prediction value. In the case of classification trees, class labels are kept in the leaves. The majority of the tree-based inducers involve a single attribute in each test of the internal nodes. Such trees are referenced as univariate and they use axis-parallel hyperplanes to split the feature space.

There are also algorithms that apply multivariate tests based mainly on linear combinations of multiple attributes. The oblique split causes a linear division of the feature space by a non-orthogonal hyperplane. DTs, which allow multiple features to be tested in a node, are potentially smaller than those that are limited to single univariate splits, but have much higher computational cost and are often difficult to interpret [9].

The popularity of DTs can be explained by their easy application, effectiveness as well as fast operation. The hierarchical structure of tree-based approaches, in which the relevant tests from one node to the next one are applied one after the other, is similar to the human decision-making process. All this makes DTs natural and easy to understand, even for an inexperienced analyst. However, despite 50 years of research on DTs, some open issues still remain [42]. One of the emerging issues is to meet ever-growing computational demands (the execution times and resource utilization), especially when applied to big data.

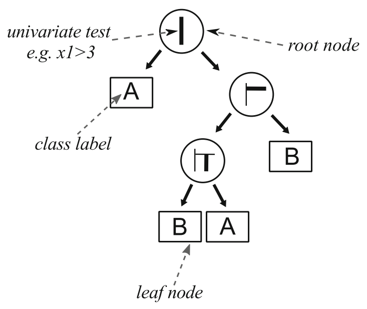

Fig. 1 An example of univariate decision tree

2.2 Induction of decision trees

Inducing an optimal DT from a dataset is an NP-complete [30] problem. Therefore, to induce decision trees in practice, various heuristics have been introduced. One of the most common examples are greedy algorithms. Such algorithms make the locally optimal splits at each node (using a defined goodness of split criterion) with the intent of finding a global optimum. The top-down approach [54] is the most popular method of decision tree induction. Starting from the root node, the best test is searched locally based on the given splitting criterion. The training instances are next redirected to newly created nodes and this procedure is repeated for each node until some stopping-rule is met. In addition, the application of post-pruning [22] is often performed to avoid (or at least to limit) the negative effects of overfitting to the training data and to improve the generalizing power of the predictive model.

CART [8] and C4.5 [52] are considered the two most popular top-down based DT inducers. In the first inducer, the goodness of split is measured by the Gini index or the Twoing criterion, whereas the C4.5 solution uses the gain ratio criterion. The CART algorithm generates binary trees, while the C4.5 also accepts the multi-way splits. The greedy strategy applied in both inducers is generally fast and efficient in many practical problems. However, it does not ensure that the globally best solution is searched. One of the attempts to reduce sub-optimal solutions is proposed in look-ahead algorithms like APDT [56]. Another alternative is the application of ensemble classifiers, like Random Forests [7] which induce a collection of trees called the forest. To make a final prediction, a majority voting is performed where each tree has a single vote. Such a technique of the ensemble prediction decreases the error [70], but at the same time, it increases the model complexity and makes the solution more difficult to understand and interpret. There were also studies that addressed the problem of the tree complexity (size), e.g., by a constrained number of leaf nodes [66] or by building a customized tree for each test instance that consists of only a single path from the root to a leaf node [61].

One of the alternative techniques that can be used to solve difficult computational problems are evolutionary algorithms (EAs) [69]. EAs belong to a family of metaheuristic methods that are inspired by biological mechanisms of evolution and are also known to be effective at escaping local optima. The strength of such an approach in context of DTs lies in a global search for the tree structure and the tests in the internal nodes. The typical EA operates on a population of individuals (trees) that represents possible solutions to the target problem. In each evolutionary iteration, individuals are modified with genetic operators such as mutation and crossover, and evaluated according to the fitness function (see Fig. 2). Next, individuals are reproduced to a new population of offspring whereas individuals with higher fitness are reproduced more often. The evolutionary loop is stopped when the convergence criteria are satisfied.

One of the first attempts to apply the global approach to DT was investigated in genetic programming (GP) community. In [37], LISP S-expressions (corresponding to DT with only nominal tests) were evolved. Currently, recognized population-based alternatives include EAs primarily [3]. However, in the literature, we may find other examples, like ant/bee colony optimizations [13, 40], differential evolution [5] or evolution strategy [4]. The review of global inducers of DTs [3] shows many benefits of such an approach, like finding more suitable models with new hidden patterns that are overlooked by the greedy solutions. However, the global DT induction is much more computationally demanded and, therefore, it takes relatively more time in comparison to the popular top-down methods.

2.3 Related works

2.3.1 Parallelization of EA

EAs can be accelerated using different parallelization techniques [16]. There are at least three basic parallel approaches on CPU architectures: master-slave model, island model and a cellular one. In the master-slave model, the management is centralized. The master processor

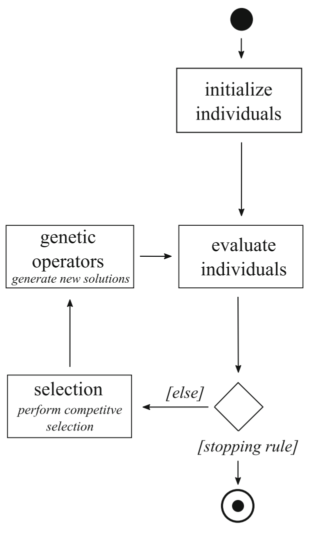

Fig. 2 Flowchart of the typical evolutionary algorithm

spreads the tasks (e.g., data chunks or individuals) over the slaves that process in parallel. Then, the results are collected by the master. On the other side, the island model is a distributed approach where individuals (or groups of individuals) can evolve independently, and infrequent migrations take place using some communication topology.

There are also different decomposition techniques in EAs parallelization [23]. In each evolutionary iteration, there are some algorithm parts where the individuals from the current population are considered separately. Thus, it is one of the ways to spread the computations over multiple processors. It is one of the traditional decomposition techniques and it is called a population approach (or a control approach) [16]. The main drawback of this solution is large population necessity in order to retain good scalability to very large datasets. In addition, high inter-processor data traffic may be a problem when distributed-memory systems are used, whereas shared-memory systems can suffer from insufficient number of available processors and memory access contention [27, 72].

2.3.2 GPU parallelization of EA

Recent research on the parallelization of EAs has seemed to focus on GPUs as the implementation platform [62]. The GPUs are popular parallel computing hardware because of their wide availability, relatively low cost and high computing power. GPU-accelerated solutions [11, 31] use mainly data parallel approach that is the second decomposition technique. The parallel evaluation of instances is considered much more scalable to the dataset size than the population approach. It focuses on spreading the entire dataset into chunks that are processed by multiple processors in parallel. However, despite the decrease of the communication overhead, some issues with high inter-processor data traffic (e.g., during the results reduction) can still remain. There are also algorithms that parallelize EAs using both decomposition techniques as well as additional dimensions of parallelization [11].

GPGPU has also been successfully used with other strategies for EAs parallelization [29, 35]. In the island model, the group of individuals in subpopulations distributed between islands were evolved in parallel [43]. In the BioHEL system [24], the authors proposed a two-dimensional parallelization. Both the rules and the instances in the training dataset were processed in parallel (coarse-grained strategy). GPGPU was also applied to the cellular EA framework [57]. Individuals were distributed over a lattice of processing units with the defined neighborhood topology. The communication only between the nearest individuals (for selection and reproduction) was possible. In another study [23], three-dimensional decomposition was tested (instances, classifiers and attributes). It was showed that a fine-grained paradigm could be more efficient on datasets with 10–50 discrete attributes.

2.3.3 Parallelization of DT induction

As regards the induction of DTs, we can find several GPU-based parallelizations, concerning greedy inducers [41, 49, 59], global ones [31] as well as DT ensembles, e.g. random forests [44] or gradient boosting DTs [55, 64]. In the CUDT system [41], the induction time of a typical decision tree was reduced up to 55 times. In order to find the locally best splits, a parallel search through the attributes in each internal node was performed. The solution was later extended with processing multiple tree nodes in parallel, providing, however, similar speedups [59].

Concerning the parallelization of ensembles of trees, the most straightforward approach was used in the CudaRF system [26], where each CUDA thread was responsible for building one tree in the forest. Another level of parallelism was tested for random forest building for data streams [44]. The authors proposed a GPU-accelerated solution in which the calculations of the majority class in leaves as well as the splits in internal nodes were parallelized. It provided at least 300× faster induction 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 which can be interpreted by a data analyst.

Concerning the evolutionary induction of DTs, a hybrid MPI+OpenMP approach [18] was investigated for classification trees. The master-slave paradigm and control parallelization approach were applied. The population was evenly distributed to the available slave processors (computer cluster nodes and further, inside nodes, over processor cores). This way, the most time-consuming operations, such as the fitness evaluation and genetic operators, were executed in parallel on the slave nodes. The experimental validation showed that such a hybrid parallelization was able to speed up the induction up to 15× for 64 CPU cores. Similar speedup but on much larger datasets was achieved using a Spark parallelization [53]. It was shown that the Spark boosted solution was able to process really large-scale data, even up to billions of objects. Moreover, an unmodified Spark solution can be easily scaled up just by adding more hardware to the cluster. However, a GPU-based acceleration appeared to be generally faster but limited by the size of the GPU memory [34].

To the best of our knowledge, in the literature, there are currently two studies that cover GPGPU parallelization of evolutionary DT induction - one for the classification trees [31] and one for simple regression trees [32]. In both cases, the experimental validation on real-life and artificial datasets showed that the GPU-supported algorithms were able to reduce the induction time significantly, even up to two orders of magnitude (in comparison to the original CPU version). However, this single GPU solution had certain limitations. The most important one concerns the limited size of GPU global memory. When the dataset size was above tens/hundreds of millions of objects, it could not be loaded into the GPU memory and efficient parallel processing was not possible [34]. To overcome this inherent limit, a multi-GPU solution is needed. Another issue to be faced is that GPU technology is developing so fast that next generation graphics accelerators enforce constant modifications of parallel algorithms to exploit full GPGPU potential.

Our preliminary attempts to create the multi-GPU solution were presented in the GECCO conference poster [33]. In this paper, the solution is developed further, described in detail and validated thoroughly. New contributions include, among others:

- GPU memory access patterns are considered and optimized GPU kernels are developed;
- more variants of genetic operators are considered;
- an in-depth analysis of the performance of the multi-GPU solution concerning execution time, speedup as well as dataset size scalability and scalability on multiple GPUs is performed;
- both artificial and real-life large-scale datasets are processed;
- scalability bounds of evolutionary induction of DTs for large-scale data are identified;
- dataset characteristics impact (size and dimension) on convergence and speedup is explored as well;
- a non-binding comparison of induction time with a typical greedy solution is presented.

2.4 Global decision tree system

In this paper, we adapted the Global Decision Tree system (GDT) [38] as it enables evolutionary induction of various DTs, and it has been already studied in terms of parallelism. Another benefit of the GDT system is that it uses a standard EA framework [47] with an unstructured, fixed size population, and a generational selection. In this study, we have deliberately focused only on a univariate GDT version designed for classification problems to facilitate understanding and to eliminate less important details.

2.4.1 Population and selection

Individuals in the population are processed in their actual form as univariate binary classification trees (without any additional encoding). The reason for this is that DTs are quite complex structures, and for a given dataset, the optimal structure that consists of nodes and splits is unknown in advance.

The generation of the initial population should be carried out randomly to include as many different solutions as possible (providing population diversity and exploration ability). In the initialization phase, greedy heuristics are often used due to the large search space [38]. The drawback of strategy is the possibility to trap EA in local optima. On the other side, it is considered as an easy way of reducing computation time. In the GDT system, initial individuals are generated by using a simple top-down approach, but each individual is induced using only a random fraction of the training dataset, to preserve the balance between exploration and exploitation.

Univariate tests at each non-terminal tree node depend on the type of the feature. In the case of nominal-valued features, the mutually exclusive groups are created and associated with the outcomes. For continuous attributes, splits with inequality tests and binary outcomes are applied. Tests are created by randomly selecting two training objects from different classes (so-called mixed dipole) that are located in the considered node [38]. Then, a test is randomly created in order to effectively separate these two objects into subtrees. The recursive partitioning ends when it reaches the pure leaf (a leaf with training objects from the same class) or the number of instances is insufficient (by default, a minimum number of instances equals 5).

The GDT system uses the linear ranking selection [47] as a selection mechanism. Moreover, the elitist strategy is applied. It means that, in each iteration, the best individual founded so far is copied to the next population. The evolutionary search ends when a terminal condition is met: the pre-defined number of generations without fitness improvement of the best individual or the maximum number of generations is reached.

2.4.2 Fitness function

The main role of the fitness function is to reflect the goal of the algorithm. At the same time, it has to drive the evolutionary search. There can often be more than just one goal, especially in prediction tasks where not only the accuracy but also the simplicity of a predictor is desired [38]. It is common that a highly complex model that works perfectly on training data may perform worse on unseen data, due to the over-fitting. Therefore, in real-life systems, a multi-objective optimization is often applied.

The GDT system [38] provides various multi-objective optimization strategies, including weight formula, lexicographic analysis, and Pareto-dominance. In the context of univariate classification trees, the model accuracy simultaneously with the tree complexity (represented as the number of nodes) is considered. The applied fitness function is maximized and is represented as a simple weighted form, as follows:

$$Fitness(T) = Accuracy(T) - \alpha * (Size(T) - 1.0), \quad (1)$$

where $Accuracy(T)$ is the classification quality of the tree T estimated on the learning dataset, $Size(T)$ is the number of nodes, and α is a user-defined parameter that reflects the relative importance of the complexity term. A similar concept may be found in a cost complexity pruning of the CART system [8].

2.4.3 Genetic operators

Genetic operators control evolutionary search and provide, at the same time, a necessary diversity and novelty. Two specialized operators that correspond to crossover and mutation are proposed in the GDT system. Each one has several variants [38] that impact the structure of the tree as well as tests in the split nodes. The general role of the crossover operator is to mix two existing individuals and create new solutions (offspring) with certain similarities to their ‘parents’. To perform the crossover, an exchange point (node) in each affected individual is randomly chosen. The typical variant exchanges the subtrees starting in the selected nodes, which is the equivalent of typical crossover that can be found in genetic programming [37]. In the case of non-terminal nodes and the equal number of outcomes, an exchange of randomly chosen branches or tests is also possible. An example where individuals exchange subtrees is illustrated in Fig. 3.

The mutation operates on a single individual. In the GDT system, it starts with the selection of a node type: a non-terminal node or a leaf. Both types have an equal probability of being chosen. Then, the algorithm creates the ranked list of expected nodes and selects the one that will be affected using a mechanism similar to the ranking linear selection [47]. If a non-terminal node type is selected, the ranking considers two metrics:

- Position of the node (in order to mutate more often the lower parts of the tree). Changes in the nodes located near the root (or the root itself) may shift the entire tree structure, while the mutation of the nodes in lower tree levels has just a local impact.
- Prediction performance of the node (in order to more likely mutate the nodes with a high error rate per instance).

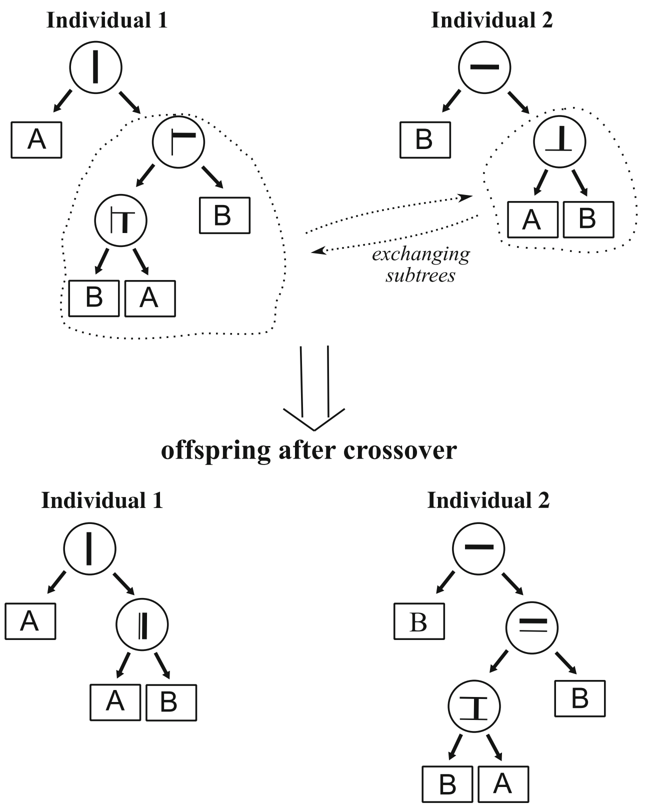

Fig. 3 An example crossover between two individuals and their resulting offspring. Internal nodes as well as a tree root are represented by circles, while leafs by rectangles. Bold line represents an univariate test in the current tree node

If a leaf-node type is selected, then only the second criterion applies. Pure leaves (with only objects from one class) are not considered in the ranking. This way, nodes in the lower parts of the tree and/or the ones with a higher number of misclassified objects are more often mutated.

There are several variants of mutation provided by GDT, which influence both the tree structure and the splits. Here, we include just a brief listing:

- *Prune* - reduces a subtree into a leaf (acts like a standard pruning);
- *Exchange parent with son* - a test of the parent node is replaced with a test from a random son (pruning of the internal node);
- *Expand* - a selected leaf is extended to a sub-tree with a randomly chosen test (expansion of the tree);
- *New test* - a test in the internal node is reinitialized using a new mixed dipole (another attribute can be used);
- *Shift threshold* - a splitting threshold (on the same attribute) is shifted.

For the detailed analysis of all variants, please refer to our previous works [38].

3 Multi-GPU accelerated solution

This section covers the proposed multi-GPU acceleration for the evolutionary induction of DTs. At first, a brief introduction to GPGPU and CUDA is given. Next, our GPU-boosted algorithm is described in detail.

3.1 GPGPU, CUDA

Modern graphics accelerators/cards are equipped with thousands of tiny computing units, called GPU cores (see Fig. 4). Each GPU core, though smaller, simpler, and slower than a CPU core, is tuned to be especially efficient at the basic mathematical operations. This simplicity allows many more GPU cores to be crammed into a single chip. Moreover, current GPU architectures are approaching terabytes per second memory bandwidth that, coupled with many computational units, creates an ideal device for running multiple tasks in parallel.

It is becoming more common that modern graphics accelerators have an unmatched price/performance ratio which is simply not achievable using only multi-core CPUs. Moreover, multiple GPUs can be installed in a single workstation saving space and energy in comparison to traditional CPU-based clusters. Thus, not only graphics applications, but also general-purpose computations on GPUs (GPGPU) have gained in popularity [63, 67].

Researchers and IT staff have seen the computational potential of graphics cards in general-purpose computations for many years [51]. However, the computational power of GPUs was difficult to exploit without efficient and intelligible development environments. Compute Unified Device Architecture (CUDA) [65] is a programming interface and parallel platform that has revolutionized GPGPU. Although there are some open alternatives (like OpenCL) and CUDA is vendor-specific, it is the most widespread platform.

In CUDA, a CPU works together with a GPU (a specialised co-processor to parallel processing). It means that a part of CPU's tasks can be delegated to the GPU and be processed by thousands of threads in parallel concurrently to the CPU operations. From a programming perspective, the CPU launches kernels that are the functions run on the GPU. For each kernel, many threads are created. By default, all threads execute the same kernel code. However, each thread has an ID which is used for taking control decisions and also for calculating memory addresses. The threads are hierarchically grouped into thread blocks, which are, in turn, arranged in a grid.

The structure of CUDA GPU memory is also hierarchical [58]. Several memories with different features (like latency, bandwidth, read-only access), various scopes (global, local) and lifetimes are provided. The global memory (residing in device DRAM) has the largest capacity, is globally accessible to all threads but has the highest latency. On the other side, there are registers, shared memory, caches, etc., that are fast but small on-chip memories. This multi-level memory allows programmers to further customize the throughput of many high-performance CUDA applications [10].

The CPU load reduction using GPGPU is recently widely adopted in many computational intelligence methods [11, 48, 71]. In evolutionary data mining, high computational complexity is naturally associated with the using of iterative and population-based search techniques, especially when large-scale data is handled. Thus, the application of GPUs usually focuses on boosting the performance of the evolutionary search directly [12, 15, 31].

3.2 Multi-GPU approach

The general idea of the GPU-accelerated approach is illustrated in Fig. 5 and in Listing 1. The CPU controls

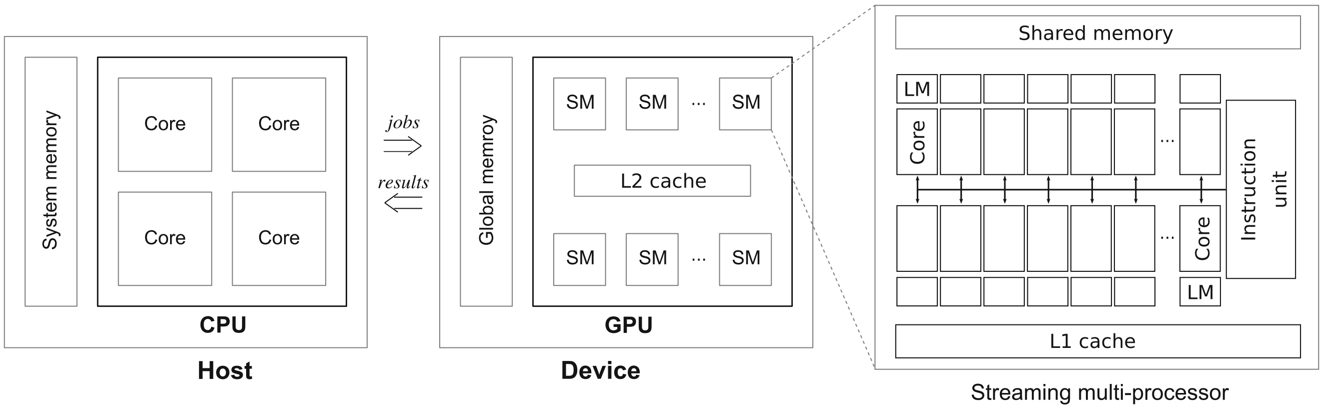

Fig. 4 The CUDA hardware architecture. A specialized microprocessor (GPU) is situated next to the CPU. The GPU consists of thousands of small computing units (cores) able to efficiently process many tasks in parallel concurrently to the CPU. The GPU cores are grouped into streaming multi-processors (SM)s. The structure of CUDA GPU memory is also hierarchical. There are registers, shared memory, caches that are fast but small on-chip memories, while the global memory (off-chip RAM) has the largest capacity but has the highest latency

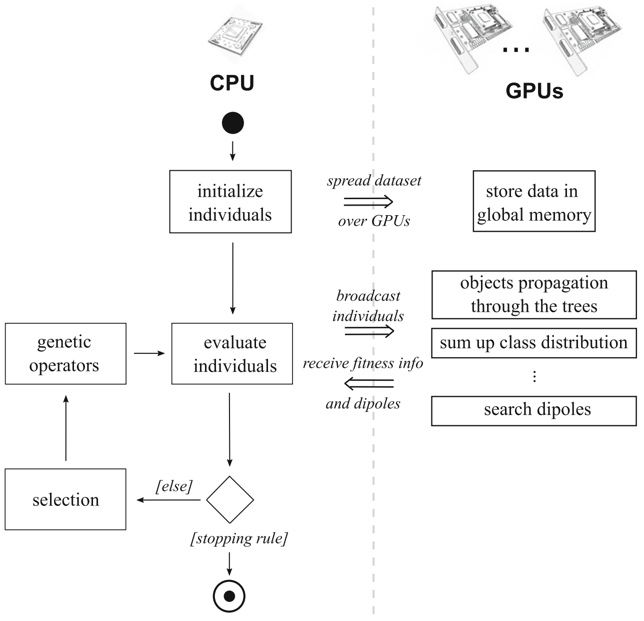

Fig. 5 General flowchart of the GPU-accelerated approach

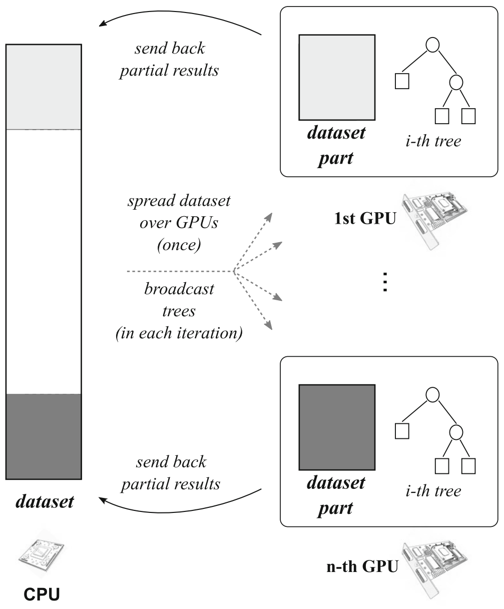
send back partial results
dataset part
i-th tree
spread dataset over GPUs (once)
1st GPU
broadcast trees (in each iteration)
send back partial results
dataset part
i-th tree
dataset
CPU
n-th GPU

Fig. 6 Work spreading over the GPUs. Each GPU processes the same individual but it is fed by a different chunk of the dataset. At the end, all GPU send back the partial results to the CPU that merges them to obtain the desired individual statistics

the main loop of the evolution process. Also, most of the algorithm phases that are not time-demanding are performed on the CPU (like the creation of an initial population, selection, genetic operators). The most time-consuming tasks (like error and fitness calculations as well as dipoles searching) are parallelized on GPUs. Such an architecture of the GPU-supported implementation ensures that the behavior of the original EA is not affected.

The initialization and selection steps remain unchanged. These steps are not parallelized because their execution times are negligible compared to the total execution time of the algorithm (less than 1%). Moreover, the initial population is built only once before the evolutionary loop on small fractions of the dataset.

We apply the data decomposition strategy [27]. The whole dataset is spread over the GPUs during the initialization. On each GPU, a space in the global memory is allocated to store a received part of the dataset. By default, the dataset is divided equally between the GPUs, leading to balancing the workload for homogeneous GPU resources. The dataset parts are kept on all GPUs till the evolutionary loop is finished. This way, the GPUs have constant access to the training objects, and the heaviest CPU-GPUs data transfer is done only once.

The GPUs are asked to do time-demanded calculations when a genetic operator is successfully applied. All available GPUs are requested. First, the CPU performs relatively fast operations (e.g., concerning changes in a tree structure). Second, the modified individual is broadcasted to all GPUs (see Fig. 6). Then, the GPUs are used to help in the individual evaluation and searching the dipoles. The CPU calls a kernel for each GPU. All GPUs process the same individual in parallel but operate on different chunks of the dataset. In the kernel, each GPU propagates the assigned objects of the training dataset through the tree starting from the root node to appropriate leaves. The next level of parallelism is provided inside each GPU where the dataset chunk is divided further over the blocks and threads.

Finally, each GPU holds a part of the results that are sent back to the CPU. All received parts are merged with the overall result which is finally used to update the affected individual. Each partial result concerns distributions of classes and randomly selected objects. Based on them, the CPU calculates errors and prepares pairs of objects for future dipoles in all leaves. Then, they are propagated from the leaves toward the tree root to determine the fitness value finally.

By default, each GPU is controlled by a separate CPU thread. The CPU threads are created by using OpenMP directives. When the CPU part is run on a multi-core/multi-processor environment, particular kernels can be launched in parallel, eliminating kernel launch overheads. The CPU threads need to be synchronized (using a critical section) when the partial results received from one of the GPUs are added to the overall result.

3.3 Single GPU parallelization

Each GPU is responsible for a chunk of the dataset assigned by the CPU. This chunk is divided further (at

```
procedure evaluateIndividualOnGPUs
input: indiv
output: updated_indiv
begin
 //flattening the tree into one-dimensional table
 indivTabHost=copyTreeToTable(indiv);

 //one OpenMP thread is created for each GPU
 #pragma omp parallel for
 for i=1 to nGPUs do
  //set device
  cudaSetDevice(i);

  //allocate memory on device for an individual
  cudaMalloc(indivTabDev, indivSize);
  //send an individual to GPU
  cudaMemcpy(indivTabDev, indivTabHost, indivSize,
       HostToDevice);

  //GPU calculations
  //propagate objects from the tree root toward leaves
  fitness_pre<<<N_BLOCKS, N_THREADS>>>(...);
  cudaDeviceSynchronize();
  //reduce results from blocks
  fitness_post<<<1, N_BLOCKS>>>(...);
  cudaDeviceSynchronize();
  //free memory for the individual on device
  cudaFree(indivTabDev):

  //copy partial results from device
  //class distributions and dipoles
  cudaMemcpy(classDistHost[i], classDistDev[i],
       indivSize, DeviceToHost);
  cudaMemcpy(dipolesHost[i], dipolesDev[i], indivSize,
        DeviceToHost);

  //merge partial results (in critical section)
  #pragma omp critical
   mergeClassDistribution(classDist,classDistHost[i])
        ;
   mergeDipoles(dipoles, dipolesHost[i]);
 end for
 updated_indiv=updateIndividual(indiv, classDist,
      dipoles);
end

procedure main
begin
 ...
 //spread data over GPUs
 for i=1 to nGPUs do
  //set device
  cudaSetDevice(i);

  //allocate memory on device for a part of the dataset
  cudaMalloc(datasetPartDev, datasetSize/nGPUs);

  //send dataset part to a GPU
  cudaMemcpy(datasetPartDev, dataset+i, datasetSize/
       nGPUs, HostToDevice);
 end for

 //evolutionary loop
 while !stopCondition do
  //loop over pairs of individuals
  for i=1 to nIdivPairs do
   crossover(indivPair[i]);
   evaluateIndividualOnGPUs(indivPair[i].indiv1);
   evaluateIndividualOnGPUs(indivPair[i].indiv2);
  end for
  //loop over individuals
  for i=1 to nIdividuals do
   mutation(indiv[i]);
   evaluateIndividualOnGPUs(indiv[i]);
  end for
  selection();
 end while
 ...
end
```

Listing 1 Pseudo code of the main evolutionary loop and multi-GPU loops

two levels) to spread calculations over the cores within a GPU. At first, the chunk is divided into smaller parts between different GPU blocks. Inside each GPU block, the objects (from the assigned fraction of the dataset) are spread further over the threads. We decided to focus only on data decomposition strategy (at three levels, in total), as it is a powerful and flexible (scalable) strategy for deriving concurrency when operating on large data [27, 31]. Other decomposition strategies (population-based or a hybrid) were not able to provide enough parallelism (consequently, better performance) due to small population sizes that we consider.

Inside each GPU, the calculations are organized into two kernel functions ($fitness_{pre}$ and $fitness_{post}$, see Fig. 7). The $fitness_{pre}$ function is called to propagate objects from the tree root to the leaves (see Listing 2). For each block, a copy of the evaluated individual is created. Thanks to this, the threads from different blocks accumulate results in the separated tree copies. They do not need to synchronize each other and can work independently. Threads inside each block are synchronized using atomic operations when needed. This way, the GPU threads (both inside and outside the blocks) process the same individual in parallel but perform calculations on different chunks of the data. Finally, this kernel outputs the number of objects of each class in each tree leaf. In addition, two randomly selected objects of each class are provided in each tree leaf, which is later required for dipoles. However, the results are scattered over separated copies of the individual created for each GPU block.

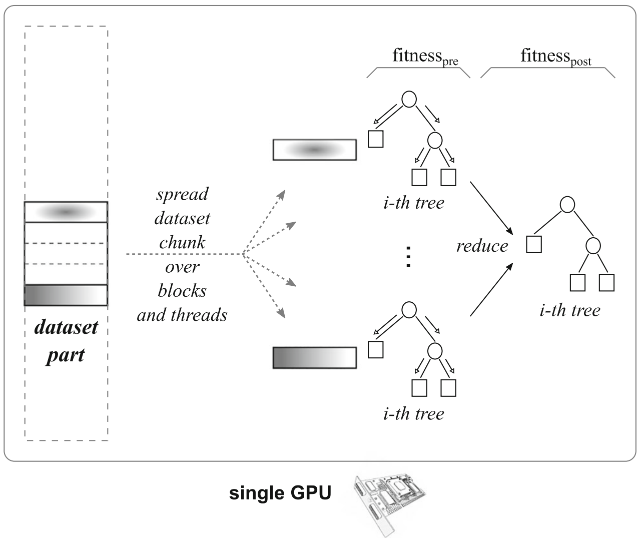

Fig. 7 Inside each GPU, the data decomposition strategy is also applied. A part of the dataset assigned to the GPU is spread further over GPU blocks and threads. Each thread processes different objects and has a roughly equal amount of work to do

```
__global__ procedure fitness_pre
input: indivTab, datasetTab
output: classDistTab, dipolesTab
begin
 int nObjsToCheck=nObjs/gridDim.x/blockDim.x;
 int startingObjIdx= threadIdx.x*(nObjs/blockDim.x);

 int idx=0;
 for i=1 to nObjsToCheck do
  node=indivTab[idx];
  int objIdx=startingObjIdx+i;

  while true do
   if node is a leaf then
    //increment class counter
    int temp=idx*N_CLASSES+datasetTab[objIdx].classId;
    atomicAdd(classDistTab[temp], 1);

    //store an object for dipoles
    setDipoles(dipolesTab, datasetTab[objIdx]);
    break;
   else
    if datasetTab[objIdx][node.attr] > node.value then
     idx = idx*2+1; //go to left child node
    else
     idx = idx*2+2; //go to right child node
    end if
   end if
  end while
 end for
end
```

Listing 2 Pseudo code of the first kernel function

```
__global__ procedure fitness_post
input: classDistTab, dipolesTab
output: results1Tab, results2Tab
begin
 __shared__ classDistReduce[N_NODES*N_CLASSES];
 __shared__ dipolesReduce[N_NODES*N_CLASSES*N_DIPOLES];

 //reduce class distributions
 for i=1 to N_NODES*N_CLASSES do
  atomicAdd(classDistReduce[i], classDistTab[i]);
 end for

 //reduce dipoles
 for i=1 to N_NODES*N_CLASSES*N_DIPOLES do
  if dipolesTab[i]!=0 then
   atomicCAS(dipolesReduce[i], false, dipolesTab[i]);
  end if
 end for

 //transfer from shared memory to global memory
 if threadIdx.x == 0 then
  for i=1 to N_NODES*N_CLASSES do
   results1Tab[i]=classDistReduce[i];
  end for
  for i=1 to N_NODES*N_CLASSES*N_DIPOLES do
   results2Tab[i]=dipolesReduce[i];
  end for
 end if
end
```

Listing 3 Pseudo code of the second kernel function

The purpose of the $fitness_{post}$ function is to reduce the partial results from multiple copies of the individual (see Listing 3). First, it reduces information about the distributions of classes in the leaves. Moreover, in each leaf, two objects of each class are randomly selected from the objects provided by the first kernel function. After the reduction, the obtained partial results are copied from the shared memory space to the global memory. Finally, the GPU sends the obtained tree statistics as well as objects for dipoles to the CPU. The CPU updates the affected individual, and if there is any GPU that still processes data, it waits for the rest of the results.

3.4 Memory and implementation aspects

To provide dataset size scalability and to improve overall solution performance, we decided not to store the arrangements of objects. Thus, the information which objects fall into which tree nodes during the evolutionary search is not accumulated on the CPU side (in contrast to the sequential version). However, some variants of genetic operators require mixed dipoles (e.g., reinitialization of a test). That is why the GPUs provide (by default) two objects from each class in each tree node. These objects are used by the CPU to constitute the dipoles. In order to store all objects located in each tree node on the CPU, GPUs would have to collect, save and finally transfer this information to the CPU. It would take much more time than drawing and transferring two objects in each tree node by the GPUs to the CPU. Moreover, the memory size requirements would significantly increase that could limit the dataset size scalability of the solution.

The representation of trees and dataset differs between the CPU and GPU. On the GPUs, to yield efficient GPU memory management [65], they are represented by one-dimensional arrays. The dataset is transferred to the GPUs only once before the evolution start. Thus, the time of transformation and sending is negligible even if the dataset is large. As regards the trees, in each iteration, tens of individuals are transferred to the GPUs. Before sending, their flat representation is created based on their references (pointers) CPU representation. Such a flat representation is also used during GPU computations. The array index of the left child of the i-th node equals $(2 * i + 1)$, while for the right child, it is $(2 * i + 2)$. When considering large datasets, the time of transformation and transfer of individuals, even in each iteration, is also negligible [31].

In GPU-accelerated applications, memory access patterns are critical for computational efficiency [45]. Thus, the choice of an appropriate data layout is an important issue. In our case, the most frequently read/written data is the information about the training objects and tree nodes. At the same time, this information is also the most massive data. The dataset objects as well as trees are stored in one-dimensional arrays. For arrays of items, there are two major layouts: Structure-of-Arrays (SoA) and Array-of-Structures (AoS) [60]. In SoA, multi-value data (e.g., values of the attributes of the dataset) are stored in the separated arrays (`struct dataset{ float tabAttr1[]; float tabAttr2[]; ...; tabAttrN[];}`). In AoS, there are grouped in a structure: (`struct Object{ float attr1; float attr2; ...; float attrN; }; Object dataset[];`.

The choice of data layout is not always obvious; however, it should primarily minimize the number of memory transactions on the off-chip (the slowest) global memory. In the algorithm, we decided to apply the SoA layout which is usually preferred from GPU performance perspective because one thread may copy data to cache for other threads. This is called a coalesced memory transaction when threads within the same warp access consecutive memory elements [65]. Such global memory access decreases the number of memory transactions and, as a result, minimizes DRAM bandwidth.

Concerning the GPU shared memory, it is applied when the reduction of tree statistics is performed. Finally, the merged results are copied from the shared memory to the global memory space before the $fitness_{post}$ ends. Nevertheless, the size of the GPU shared memory is usually limited to 48 KB per SM [50]. Thus, the trees of limited size can be handled by this mechanism. For larger trees, the algorithm uses the default global memory space. Moreover, at the beginning of each kernel, frequently used data (trees, objects) are copied to local variables explicitly.

4 Experiments

In this section, we present experimental analysis of the proposed multi-GPU approach, performed on artificially generated and real-life datasets.

4.1 Setup

An artificially generated problem called *Chess3x3* with two classes is used. It has two real-values attributes and objects arranged on a 3 × 3 chessboard (Fig. 8). It has a two-dimensional domain with three intervals in each dimension. We consider five regions labeled as ‘x’ and four regions labeled as ‘+’. All instances are created using a random number generator with uniform distribution. Decision borders are defined analytically. An ideal tree for this problem has a moderate size: 17 nodes and of which 9 leaves (5 with ‘+’ and 4 with ‘x’).

We deliberately used the synthetic dataset so we can scale it freely and have full control over the tree induction process, as we know exactly when the best individual reaches the optimal or near-optimal solution. We examined a various number of training objects (from dozens of thousands to a billion) as well as attributes (from two to

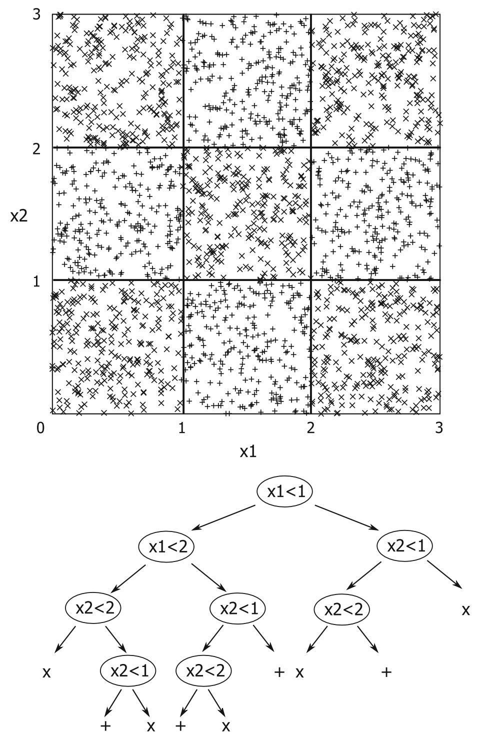

Fig. 8 An example of analyzed *Chess3x3* dataset variant and the corresponding ideal structure of a classification tree

ten thousand), where additional attributes were randomly generated.

Concerning real-life datasets, two large datasets from the UCI Machine Learning Repository [21] were tested:

- Suzy dataset (5 million of instances, 18 features, 2 classes) - the physics dataset that concerns the problem of classification of signal process which produces supersymmetric particles and a background process where no detectable particles are produced;
- Higgs dataset (11 million of instances, 28 features, 2 classes) - the physics dataset that has been produced using Monte Carlo simulations; refers to a classification problem to differentiate between a signal process which produces Higgs bosons and a background process which does not; one of the biggest datasets found in the UCI repository;

In the experiments, we used 4 NVIDIA Tesla P100 GPU cards, each equipped with 3 584 CUDA cores and 12 GB of memory. They were installed in a workstation with the dual processor Intel Xeon E5-2620 v4 (20 MB Cache) and 256 GB RAM. Each CPU contained 8 physical cores running at 2.10 GHz. The host operating system was a 64-bit Ubuntu Linux 16.04.02 LTS. The sequential algorithm was implemented in C++ and compiled with the use of gcc version 5.4.0. The GPU-based parallelization part was

Table 1 Default GDT parameters

Parameter	Value
Population size	64 individuals
Crossover rate	20% assigned to the tree
Mutation rate	80% assigned to the tree
Elitism rate	1 individual per generation
Max generations	10 000
Block/Thread numbers	256×256

Table 2 Mean induction times of the GPU-accelerated algorithm for various number of objects (from 10 thousands to 1 billion) of the *Chess* dataset running on 1, 2 or 4 GPUs (in seconds, and additionally in hours/minutes for 4 GPUs)

Dataset size	1 GPU	2 GPUs	4 GPUs	
10 000	20	17	15	(<0.5 min)
100 000	24	22	17	(<0.5 min)
1 000 000	44	34	24	(<0.5 min)
5 000 000	197	102	55	(≈ 1 min)
10 000 000	426	210	108	(≈ 2 min)
50 000 000	2512	1348	621	(≈ 10 min)
100 000 000	5067	2716	1285	(≈ 20 min)
1 000 000 000	56961	26747	13538	(≈ 4 h)

implemented in CUDA-C and compiled by nvcc CUDA 9.0 [50] (single-precision arithmetic was applied).

The GDT framework is regular generational EA, thus parameters such as population size, maximum number of generations, mutation and crossover probabilities as well as elitism rate have to be chosen before the evolution starts. We use a default, recommended set of GDT parameters, which are briefly listed in Table 1 [38]. For in-depth description and settings of additional parameters like probabilities of different mutation and crossover variants and GPU configurations, please refer to [31, 38].

As this paper aims to assess the size and time performance of the GPU-based solution, the exact results for the classification accuracy are not included. However, for the *Chess* dataset, the GDT system manages to induce trees with optimal structures (8 internal nodes and 9 leaves, see Fig. 8 and perfect (or near-perfect) accuracy, over 99.9%). Such a result was obtained for all tested combinations (number of objects and attributes), with the difference that the number of evolutionary iterations varied. Detailed information about accuracy performance of the GDT system can be found in our previous works, e.g. [38]. All presented results are the averages of 5-10 runs.

4.2 Multi-GPU speedup

This subsection presents the induction times and the speedup of the GPU-accelerated algorithm on the *Chess* dataset with 2 attributes as well as on two real-life datasets. Table 2 shows results of the algorithm running on 1, 2 or 4 GPUs for *Chess* datasets with various number of objects (from 10 thousand to 1 billion). As we can see, with 4 GPUs, we manage to successfully induce a classification tree for a 12 GB dataset with 1 billion instances in about 4 hours. We have estimated that the sequential GDT system would need over a year to calculate such a dataset, and using GDT with OpenMP parallelization [18] over 16 cores would decrease this time to a few months.

The huge gap between OpenMP and GPU parallelized versions, as well as the impact of various number of GPUs on the tree induction time is visualized in Fig. 9. One can easily see that the multi-GPU approach accelerates the global induction at least hundreds of times. For 50 million objects, the speedup almost reaches 3 000× for 4 GPUs, while the number of CUDA cores in a single Tesla P100 GPU is equal 3 584. In comparison to the OpenMP and hybrid MPI+OpenMP parallizations [38], the multi-GPU solution provides at least one order of magnitude better speedup.

It is clearly visible that using more GPUs in parallel calculation significantly decreases the computation time; however, the scale of improvement depends on the size of a dataset. For smaller datasets (below 10 million), the scalability is below linear, whereas for the larger datasets, we can observe linear or even a superlinear speedup (see Fig. 10). The difference in the obtained acceleration can be explained by the different ratio of computation time spent in GPU and CPU to the total evolutionary induction time. For smaller datasets, besides the most time expensive algorithm

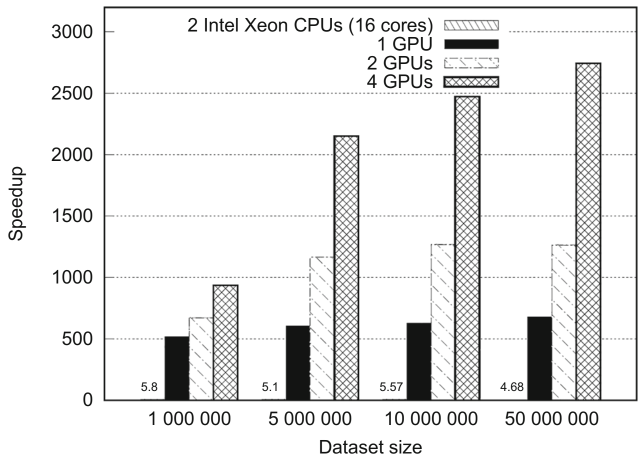

Fig. 9 Mean speedup of the GPU-accelerated algorithm for various number of objects of the *Chess* dataset running on 1, 2 or 4 GPUs. The mean speedup of an OpenMP parallelized version [18] (using 16 CPU cores) is also included

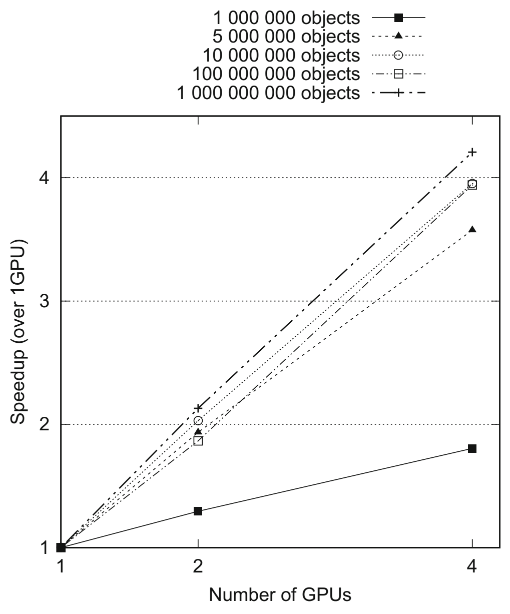

Fig. 10 Scalability of the GPU-accelerated algorithm: mean speedup (over 1 GPU) when the dataset size grows

phase that is parallelized, other algorithm parts (like GPU memory allocation/deallocation, CPU-GPU memory transfers, statistics propagation, and CPU calculations) contribute importantly to the overall execution time as it was previously observed in [31]. Moreover, smaller datasets are not able to saturate multiple GPU cores. With larger datasets, GPU calculations cover almost all the induction time and thus the time of other operations is unnoticeable. The value of linear speedup can be taken as the (upper) scalability bound.

Another scalability issue concerns the size of the dataset (the number of objects). Figure 9 and Table 2 show that: *(i)* the time of computation increases approximately linearly with the rise of the number of objects, and *(ii)* the speedup provided by the solution does not decrease when the size of the dataset grows. Thus, we can say that the solution also provides good scalability concerning the dataset size. In this case, the upper scalability bound results from the available GPUs' memory. For a single Tesla P100 GPU card equipped with 12 GB of memory, it is a little more than 1 billion of chess3x3 dataset objects (exactly 1 033 000 000 objects [34]). However, this dataset size limitation is easily shifted by adding more GPUs. On the other side, this limitation may be a source of the superlinear speedup observed in Fig. 10. A single GPU card with almost fully occupied memory can process less efficient than two half-occupied GPUs.

Concerning the real-life datasets, the time results are summarized in Table 3. We see that the multi-GPU approach also scales almost linearly. In order to feel the real difference

Table 3 Mean induction times (10 000 iterations) of the GPU-accelerated algorithm for two real-life datasets running on 1, 2 or 4 GPUs (in seconds)

Dataset	1 GPU	2 GPUs	4 GPUs
Suzy	1 976	903	480
Higgs	5 025	2 491	1 239

introduced by the GPU-based approach, we should look at induction times of the original sequential version. We have estimated that for the *Suzy* dataset, 10 000 evolutionary iterations need about two weeks, while for the *Higgs* dataset, over a month to calculate. On the other hand, the induction using 4 GPUs takes only 480 s (8 min) and 1 239 s (20 min), correspondingly.

4.3 Influence of the data characteristics

In most of the typical validations of the parallel data mining approaches, a silent assumption is accepted to scale the datasets by only adding more instances. However, in real-world applications, such an idealization is not always possible. A dataset may have a very high number of attributes that, in some cases, even exceeds the number of instances, which is a common case e.g., with genomic data [25]. Therefore, in order to show the impact of the data size (number of objects) and dimension (number of attributes) on the convergence and the speedup of the evolutionary searches, we added randomly generated (noisy) attributes to the tested dataset and performed analysis. Figures 11 and 12 show the influence of the number of attributes/instances on the convergence of the best individual during the evolution. As can be expected, a higher number of attributes or instances in the dataset increases the number of iterations required by the GDT system to find the best fitness value (Fig. 11).

Interestingly, the impact of these two data characteristics on the algorithm convergence is much different. For the *Chess* with 2 attributes, the increase of objects does not really influence the number of required iterations. However, when additional random features occur, the dataset size factor starts to matter. The noise influences drastically with the rising number of attributes. This is a result of unproductive or even harmful applications of variation operators used on the irrelevant attributes. The improvement of the best individual is much more likely when e.g., mutation variant considers tests performed on a meaningful attribute, but such attributes are drawn less frequently when the amount of noise rises. From Fig. 11, we can see that the linear increase in the number of attributes causes at least a polynomial increase in the number of iterations. It is especially visible for the datasets with a higher number of instances (please note that the number of iterations is presented on a logarithmic scale).

Similar conclusions can be drawn by analyzing Fig. 12 which illustrates the size of the best individual. The number of iterations required for GDT to find the best individual (with a proper size) firmly increases, especially with the growth of the number of attributes. Nevertheless, for all tested cases, the GDT solution is capable of finding the optimal (or near-optimal) solution. In addition, we managed to verify the efficiency of the multi-GPU parallelization when the number of attributes grows. The additional experiments (not included) showed that the dataset dimension does not influence the performance of the GPU-based algorithm on condition that the dataset fits into GPUs memories.

4.4 Top-down vs multi-GPU comparison

For a general and non-binding comparison, we have also checked how a typical top-down solution would perform on the tested datasets. Validation was performed with a Weka software [28] with the state-of-the-art C4.5 decision tree learner. We were curious whether the evolutionary GPU-accelerated solution can be competitive with the sequential version of the greedy inducer. Table 4 shows the mean induction times (in seconds) of the sequential GDT, C4.5 algorithm (Weka default implementation under the name of J48), Spark-accelerated GDT, GPU-accelerated GDT [31] as well as multi-GPU supported one using 4 GPUs for the *Suzy* and different *Chess* variants.

As we can see, starting from the *Chess* with 1 million instances, the proposed solution is much slower than J48.

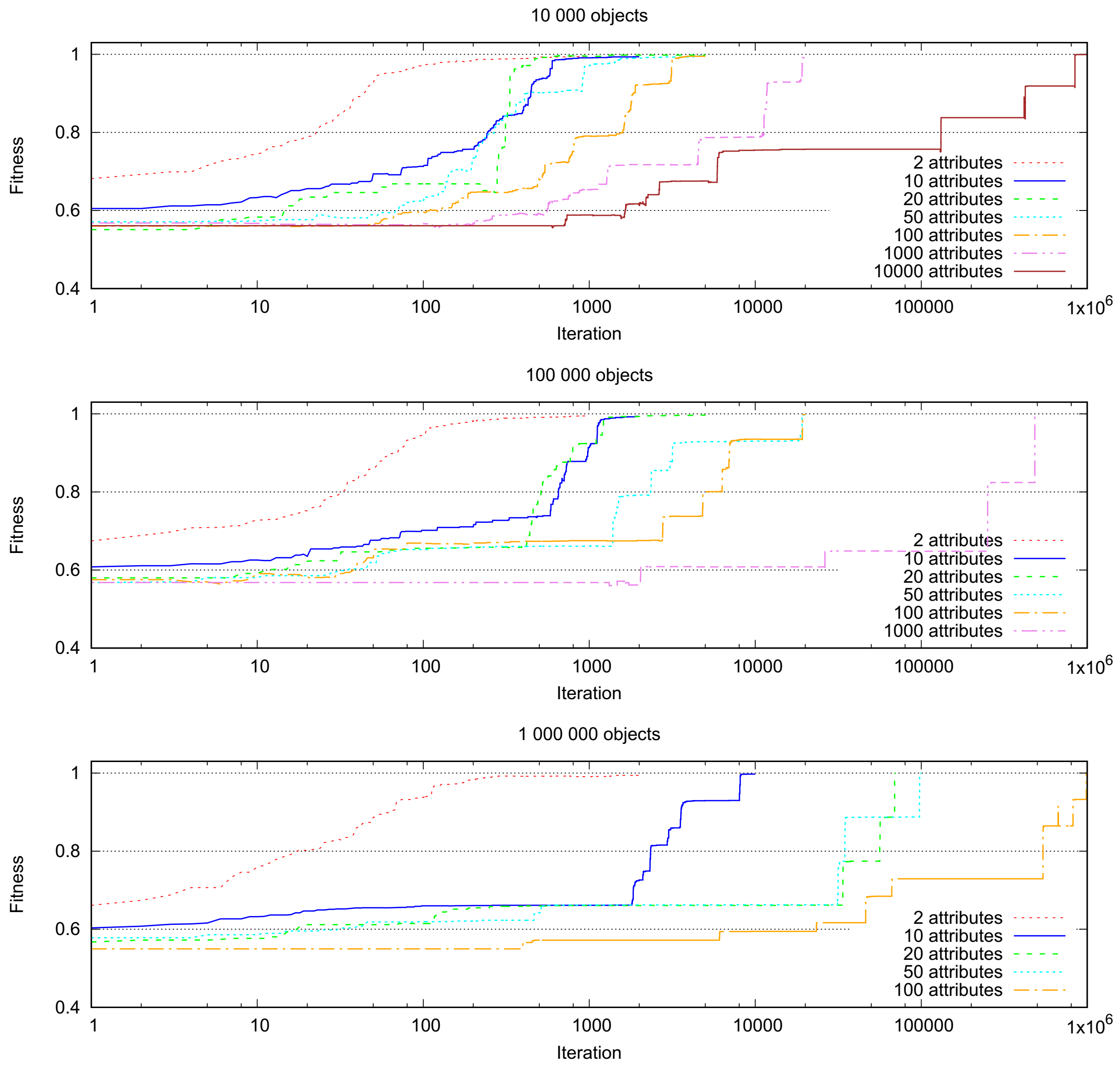

Fig. 11 Influence of the number of attributes/instances on the performance of the best individual during the evolution on the training set - mean fitness function for: 10 000 objects (top), 100 000 objects (middle) and 1 000 000 objects (bottom)

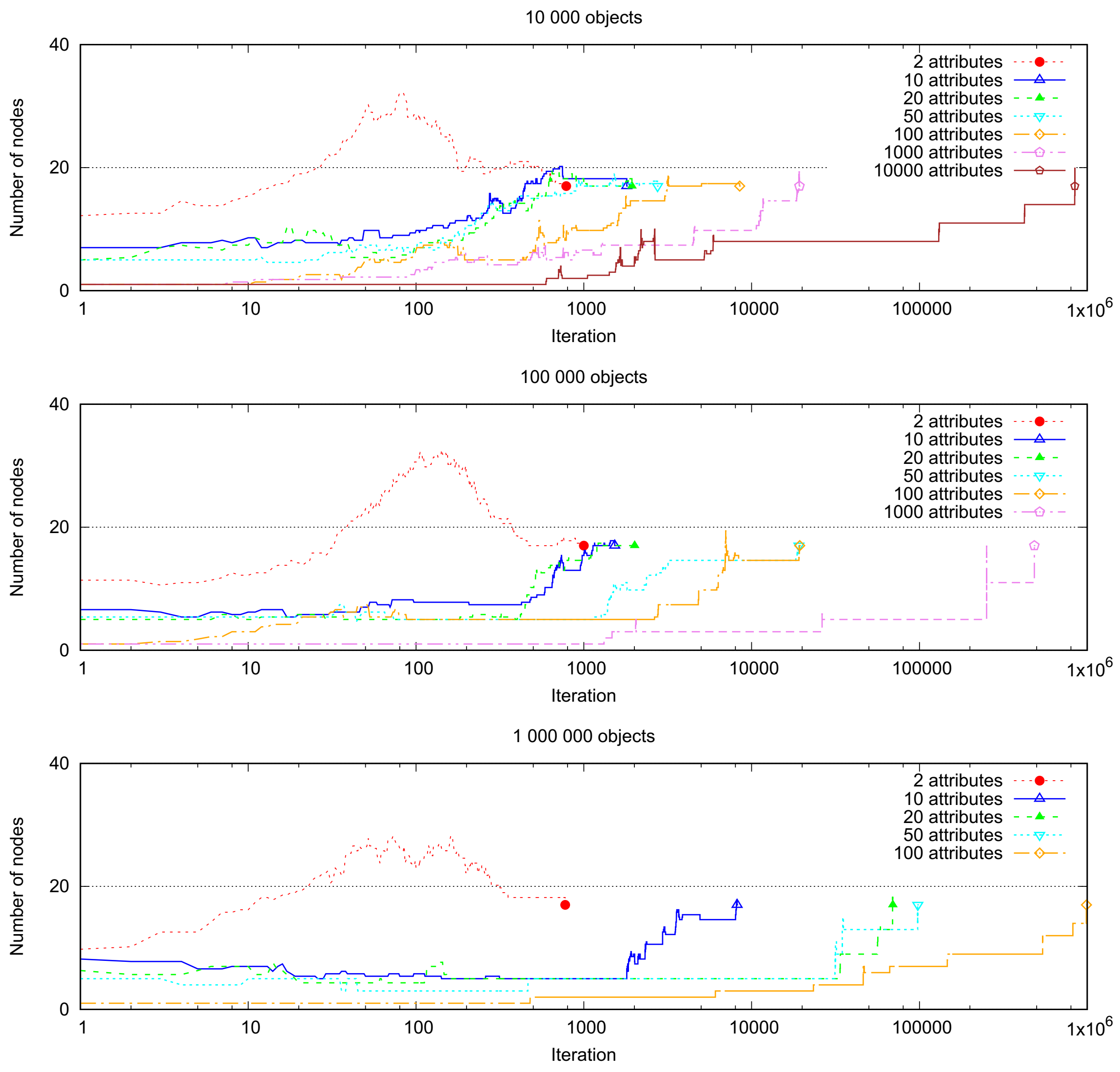

Fig. 12 Influence of the number of attributes/objects on the performance of the best individual during the evolution on the training set - mean number of tree nodes for: 10 000 objects (top), 100 000 objects (middle) and 1 000 000 objects (bottom)

However, when the number of instances grows (e.g., for 100 million), it starts to outperform the greedy approach. Tests for *Chess* with 1 billion instances as well as for *Higgs* dataset were not performed as the J48 algorithm ran out of memory during the tree induction. Table 4 gives us a general impression about the speed of the multi-GPU GDT solution, and we see that our approach is capable of competing with the sequential greedy inducers but rather for big data.

The obtained induction times seem to be also promising when compare with the Spark-accelerated version of the global inducer [34, 53]. The Spark-based solution was able to process really huge datasets (e.g., 4 billions of

Table 4 Mean induction times (in seconds) of the sequential top-down (J48 algorithm) [28], sequential GDT [38], Spark-accelerated GDT [53], GPU-accelerated GDT [31] and multi-GPU solution for various datasets

Dataset	J48 (Weka)	GDT	GDT Spark	GDT 1 GPU	GDT 4 GPUs
Chess 1 000 000	6.5	22 897	1 471	44	24
Chess 10 000 000	93	266 464	2 028	426	108
Chess 100 000 000	1 350	≈four weeks	4 917	5 067	1 285
Suzy	3 109	≈two weeks	1 595	1 976	480

instances). However, we see that it is much slower (see Table 4) and provides worse performance per watt as well as performance per price factors than the multi-GPU solution. In addition, we have attempted to confront the multi-GPU solution with the distributed version of the classical greedy inducer boosted by the well-known Spark's MLlib library [46, 68]. MLlib on 64 cores needed about 100 seconds to mine the real-life *Suzy* dataset. However, DTs generated by MLlib were clearly overgrown in comparison with ones generated by GDT (from 5 to 7 times more number of nodes). Naturally, we understand that other parallel/distributed version of the top-down inducers could outperform multi-GPU GDT in terms of speed. Anyway, we would like to emphasize that our goal was not to race with greedy inducers but to accelerate the global (EA-based) approach as that it can be directly applied in big data mining.

The accuracy and complexity of the induced trees by global and greedy approaches for the *Chess* dataset are the same. Both algorithms manage to find an optimal tree structure and almost perfectly classify the tested instances. In the case of the *Suzy* dataset, the estimated accuracies are practically equal for both the GDT system and J48. However, the trees generated by the J48 learner are overgrown (containing even about 8 000 leaves), which makes it almost impossible to analyse and interpret. The GDT system provides a much smaller predictor (over two orders of magnitude). Additional experiments (not included) have shown that by adjusting α term, the GDT system can also induce larger trees (with even higher accuracy). However, we believe that building such a complex predictor is not what 'white box' learners (like DTs) are designed for.

5 Conclusions

This paper investigates the scalability bounds of the evolutionary induction of decision trees in the context of big data. We have demonstrated the importance and effectiveness of the multi-GPU approach for the large-scale datasets with different sizes and dimensions. Experimental validation reveals how the characteristics of the dataset impact the convergence of evolutionary learning.

Our novel multi-GPU parallelization (that incorporates the knowledge of the global DT induction and EAs) is capable of accelerating the induction significantly. Experiments show that we manage to induce classification trees for a 12 GB dataset with 1 billion instances in less than 4 hours and achieve speedup almost 3 000 $\times$, to the sequential version, using 4 GPUs. The solution scales linearly when more GPUs are added. We also provide a loose, non-binding comparison with the sequential version of a greedy top-down solution and show that the GPU-accelerated global approach can be highly competitive.

We see many promising directions for future research. We plan to extend our approach to the rest of GDT framework variants of decision trees, including regression trees with linear models in the leaves (model trees) [19]. Hybrid parallelizations, such as MPI/CUDA, are also within the scope of our interest. Then, the number of GPUs will not be limited by a single workstation, and we suppose that the obtained speedup allows us to process even bigger datasets or to deal with data streams [39]. Moreover, additional levels of population/individual decomposition as well as computation and data transfer overlapping mechanisms are interesting directions to explore.

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

Compliance with Ethical Standards

Conflict of interests The authors declare that they have no conflict of interest.

References

1. Baranauskas JA, Netto OP, Nozawa SR, Macedo AA (2018) A tree-based algorithm for attribute selection. Appl Intell 48(4):821–833
2. Barros RC, Basgalupp MP, De Carvalho AC, Freitas AA (2012) A survey of evolutionary algorithms for decision-tree induction. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 42(3):291–312
3. Barros RC, Basgalupp MP, Freitas AA, De Carvalho AC (2014) Evolutionary design of decision-tree algorithms tailored to microarray gene expression data sets. IEEE Trans Evol Comput 18(6):873–892
4. Beyer HG, Finck S, Breuer T (2014) Evolution on trees: on the design of an evolution strategy for scenario-based multi-period portfolio optimization under transaction costs. Swarm and Evolutionary Computation 17:74–87
5. Biswal B, Behera H, Bisoi R, Dash P (2012) Classification of power quality data using decision tree and chemotactic differential evolution based fuzzy clustering. Swarm and Evolutionary Computation 4:12–24

6. Bogawar PS, Bhoyar KK (2018) An improved multiclass support vector machine classifier using reduced hyper-plane with skewed binary tree. Appl Intell 48(11):4382–4391
7. Breiman L (2001) Random forests. Machine Learning 45(1):5–32
8. Breiman L, Friedman JH, Olshen RA, Stone CJ (1984) Classification and regression trees. Wadsworth
9. Cai Y, Zhang H, He Q, Duan J (2020) A novel framework of fuzzy oblique decision tree construction for pattern classification. Appl Intell 50:2959–2975
10. Candel F, Petit S, Sahuquillo J, Duato J (2018) Accurately modeling the on-chip and off-chip GPU memory subsystem. Futur Gener Comput Syst 82:510–519
11. Cano A (2018) A survey on graphic processing unit computing for large-scale data mining. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 8(1):e1232
12. Cano A, Zafra A, Ventura S (2015) Speeding up multiple instance learning classification rules on GPUs. Knowl Inf Syst 44(1):127–145
13. Cao J, Yin B, Lu X, Kang Y, Chen X (2018) A modified artificial bee colony approach for the 0-1 knapsack problem. Appl Intell 48(6):1582–1595
14. Chen CP, Zhang CY (2014) Data-intensive applications, challenges, techniques and technologies: a survey on big data. Inf Sci 275:314–347
15. Chitty D (2016) Improving the performance of GPU-based genetic programming through exploitation of on-chip memory. Soft Comput 20(2):661–680
16. Chitty DM (2012) Fast parallel genetic programming: multi-core CPU versus many-core GPU. Soft Comput 16(10):1795–1814
17. Czajkowski M, Czerwonka M, Kretowski M (2015) Cost-sensitive Global Model Trees applied to loan charge-off forecasting. Decis Support Syst 74:57–66
18. Czajkowski M, Jurczuk K, Kretowski M (2015) A parallel approach for evolutionary induced decision trees. MPI+openMP implementation. In: Rutkowski L, Korytkowski M, Scherer R, Tadeusiewicz R, Zadeh LA, Zurada JM (eds) Artificial intelligence and soft computing, LNCS, vol 9119. Springer, pp 340–349
19. Czajkowski M, Kretowski M (2014) Evolutionary induction of global model trees with specialized operators and memetic extensions. Inf Sci 288:153–173
20. Czajkowski M, Kretowski M (2019) Decision tree underfitting in mining of gene expression data. an evolutionary multi-test tree approach. Expert Syst Appl 137:392–404
21. Dua D, Karra Taniskidou E (2017) UCI machine learning repository. http://archive.ics.uci.edu/ml
22. Esposito F, Malerba D, Semeraro G (1997) A comparative analysis of methods for pruning decision trees. IEEE Trans Pattern Anal Mach Intell 19(5):476–491
23. Franco MA, Bacardit J (2016) Large-scale experimental evaluation of gpu strategies for evolutionary machine learning. Inf Sci 330(C):385–402
24. Franco MA, Krasnogor N, Bacardit J (2010) Speeding up the evaluation of evolutionary learning systems using GPGPUs. In: Proceedings of the 12th annual conference on genetic and evolutionary computation, GECCO '10. ACM, New York, pp 1039–1046
25. Gligorijevic V, Malod-Dognin N, Przulj N (2016) Integrative methods for analyzing big data in precision medicine. Proteomics 16(5):741–758
26. Grahn H, Lavesson N, Lapajne MH, Slat D (2011) CudaRF: a CUDA-based implementation of Random Forests. In: 2011 9th IEEE/ACS international conference on computer systems and applications (AICCSA), pp 95–101
27. Grama A, Karypis G, Kumar V, Gupta A (2003) Introduction to parallel computing. Addison-Wesley, Boston
28. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The WEKA data mining software: an update. SIGKDD Explor Newsl 11(1):10–18
29. Hofmann J, Limmer S, Fey D (2013) Performance investigations of genetic algorithms on graphics cards. Swarm and Evolutionary Computation 12:33–47
30. Hyafil L, Rivest RL (1976) Constructing optimal binary decision trees is NP-complete. Inf Process Lett 5(1):15–17
31. Jurczuk K, Czajkowski M, Kretowski M (2017) Evolutionary induction of a decision tree for large-scale data: a GPU-based approach. Soft Comput 21(24):7363–7379
32. Jurczuk K, Czajkowski M, Kretowski M (2017) GPU-accelerated evolutionary induction of regression trees. In: Martín-vide C, Neruda R, Vega-Rodríguez MA (eds) Theory and practice of natural computing. Springer, pp 87–99
33. Jurczuk K, Czajkowski M, Kretowski M (2019) Multi-GPU approach for big data mining - global induction of decision trees. In: Proceedings of the genetic and evolutionary computation conference companion, GECCO 2019, Prague, Czech Republic, pp 175–176
34. Jurczuk K, Reska D, Kretowski M (2018) What are the limits of evolutionary induction of decision trees?. In: Auger A, Fonseca CM, Lourenço N, Machado P, Paquete L, Whitley D (eds) Parallel problem solving from nature – PPSN XV. Springer, pp 461–473
35. Kalantzis G, Shang C, Lei Y, Leventouri T (2016) Investigations of a GPU-based levy-firefly algorithm for constrained optimization of radiation therapy treatment planning. Swarm and Evolutionary Computation 26:191–201
36. Kotsiantis SB (2013) Decision trees: a recent overview. Artif Intell Rev 39(4):261–283
37. Koza JR (1991) Concept formation and decision tree induction using the genetic programming paradigm. In: Schwefel HP, Männer R (eds) Parallel problem solving from nature. Springer, Berlin, pp 124–128
38. Kretowski M (2019) Evolutionary decision trees in large-scale data mining. Springer, Berlin. https://doi.org/10.1007/978-3-030-21851-5
39. Le T, Vo B, Fournier-Viger P, Lee MY, Baik SW (2019) SPPC: a new tree structure for mining erasable patterns in data streams. Applied Intelligence 49(2):478–495
40. Lee CY, Lee ZJ, Lin SW, Ying KC (2010) An enhanced ant colony optimization (EACO) applied to capacitated vehicle routing problem. Appl Intell 32(1):88–95
41. Lo WT, Chang YS, Sheu RK, Chiu CC, Yuan SM (2014) CUDT: a CUDA based decision tree algorithm. Scientific World Journal
42. Loh WY (2014) Fifty years of classification and regression trees. Int Stat Rev 82(3):329–348
43. Luong TV, Melab N, Talbi EG (2010) GPU-Based island model for evolutionary algorithms. In: Proceedings of the 12th annual conference on genetic and evolutionary computation, GECCO '10. ACM, New York, pp 1089–1096
44. Marron D, Bifet A, Morales GDF (2014) 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, ECAI'14. IOS Press, Amsterdam, pp 615–620
45. Mei G, Tian H (2016) Impact of data layouts on the efficiency of GPU-accelerated IDW interpolation. SpringerPlus 5(1):1–18
46. Meng X, Bradley J, Yavuz B, Sparks E, Venkataraman S, Liu D, Freeman J, Tsai D, Amde M, Owen S, et al. (2016) MLlib: machine learning in Apache Spark. The Journal of Machine Learning Research 17(1):1235–1241
47. Michalewicz Z (1996) Genetic algorithms + data structures = evolution programs, 3rd edn. Springer, Berlin
48. Mohebbi H, Mu Y, Ding W (2017) Learning weighted distance metric from group level information and its parallel implementation. Appl Intell 46(1):180–196

49. Nasridinov A, Lee Y, Park YH (2014) Decision tree construction on GPU: ubiquitous parallel computing approach. Computing 96:403–413
50. NVIDIA (2019) NVIDIA developer zone - CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/
51. Owens JD, Luebke D, Govindaraju N, Harris M, Krueger J, Lefohn AE, Purcell TJ (2007) A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 26(1):80–113
52. Quinlan JR (1992) Learning with continuous classes. World Scientific, Singapore, pp 343–348
53. Reska D, Jurczuk K, Kretowski M (2018) Evolutionary induction of classification trees on spark. In: Rutkowski L, Scherer R, Korytkowski M, Pedrycz W, Tadeusiewicz R, Zurada JM (eds) Artificial Intelligence and Soft Computing, LNCS, vol 10841. Springer, pp 514–523
54. Rokach L, Maimon O (2005) Top-down induction of decision trees classifiers - a survey. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 35(4): 476–487
55. Rory M, Eibe F (2017) Accelerating the XGBoost algorithm using GPU computing. PeerJ Computer Science 3: e127
56. Shah S, Sastry PS (1999) New algorithms for learning and pruning oblique decision trees. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 29(4):494–505
57. Soca N, Blengio JL, Pedemonte M, Ezzatti P (2010) PUGACE, a cellular evolutionary algorithm framework on GPUs. In: IEEE Congress on evolutionary computation, pp 1–8
58. Storti D, Yurtoglu M (2016) CUDA for engineers : an introduction to high-performance parallel computing. Addison-Wesley, New York
59. Strnad D, Nerat A (2016) Parallel construction of classification trees on a GPU. Concurrency and Computation: Practice and Experience 28(5):1417–1436
60. Strzodka R, Hwu WW (2012) Abstraction for AoS and SoA layout in C++. In: GPU computing gems jade edition, Morgan Kaufmann, pp 429–441
61. Su C, Cao J (2019) Improving lazy decision tree for imbalanced classification by using skew-insensitive criteria. Appl Intell 49(3):1127–1145
62. Tsutsui S, Collet P (eds) (2013) Massively parallel evolutionary computation on GPGPUs. Natural Computing Series. Springer, Berlin
63. Wang J, Cao J, Li W, Yu P, Huang K (2019) A novel parallel accelerated CRPF algorithm. Appl Intell 50:849–859
64. Wen Z, Shi J, He B, Chen J, Ramamohanarao K, Li Q (2019) Exploiting gpus for efficient gradient boosting decision tree training. IEEE Transactions on Parallel and Distributed Systems 30(12):2706–2717
65. Wilt N (2013) CUDA handbook: a comprehensive guide to GPU programming. Addison-Wesley, Upper Saddle River
66. Wu CC, Chen YL, Liu YH, Yang XY (2016) Decision tree induction with a constrained number of leaf nodes. Appl Intell 45(3):673–685
67. Yuen D, Wang L, Chi X, Johnsson L, Ge W, Shi Y (2013) GPU solutions to multi-scale problems in science and engineering. Springer, Berlin
68. Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin MJ, et al. (2016) Apache spark: a unified engine for big data processing. Commun ACM 59(11):56–65
69. Zelinka I (2015) A survey on evolutionary algorithms dynamics and its complexity – mutual relations, past, present and future. Swarm and Evolutionary Computation 25:2–14
70. Zhang H, Cao Q (2019) Fast 6D object pose refinement in depth images. Appl Intell 49(6):2287–2300
71. Zhang Z, Sun Y, Xie H, Teng Y, Wang J (2019) GMMA: GPU-based multiobjective memetic algorithms for vehicle routing problem with route balancing. Appl Intell 49(1): 63–78
72. Zhou L, Pan S, Wang J, Vasilakos AV (2017) Machine learning on big data: opportunities and challenges. Neurocomputing 237: 350–361

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

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

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