Soft Comput (2017) 21:7363–7379
DOI 10.1007/s00500-016-2280-1

METHODOLOGIES AND APPLICATION

Evolutionary induction of a decision tree for large-scale data: a GPU-based approach

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

Published online: 22 August 2016

Abstract Evolutionary induction of decision trees is an emerging alternative to greedy top-down approaches. Its growing popularity results from good prediction performance and less complex output trees. However, one of the major drawbacks associated with the application of evolutionary algorithms is the tree induction time, especially for large-scale data. In the paper, we design and implement a graphics processing unit (GPU)-based parallelization of evolutionary induction of decision trees. We apply a Compute Unified Device Architecture programming model, which supports general-purpose computation on a GPU (GPGPU). The selection and genetic operators are performed sequentially on a CPU, while the evaluation process for the individuals in the population is parallelized. The data-parallel approach is applied, and thus, the parts of a dataset are spread over GPU cores. Each core processes the assigned chunk of the data. Finally, the results from all GPU cores are merged and the sought tree metrics are sent to the CPU. Computational performance of the proposed approach is validated experimentally on artificial and real-life datasets. A comparison with the traditional CPU version shows that evolutionary induction of decision trees supported by GPGPU can be accelerated significantly (even up to 800 times) and allows for processing of much larger datasets.

Keywords Evolutionary algorithms · Decision trees · Parallel computing · Graphics processing unit (GPU) · Large-scale data

1 Introduction

Decision trees (DTs) (Kotsiantis 2013; Rokach and Maimon 2008) are one of the most famous classification methods in data mining (Fayyad et al. 1996). Traditionally, DTs are induced with greedy top-down algorithms; however, in the recent past, an evolutionary approach for tree induction has attracted a great deal of interest. Application of evolutionary algorithms (EAs) (Michalewicz 1996) in DT induction results in simpler but still accurate trees in comparison with greedy strategies (Barros et al. 2012; Czajkowski and Kretowski 2014). The main downside of this approach is that EAs entail relatively high computational costs as they generally evaluate all candidate solutions in a population for every generation. Currently, data mining systems are faced with increasingly larger datasets (Bacardit and Llora 2013), and the issue of fast processing and analyzing often becomes crucial. The survey (Barros et al. 2012) of evolutionary induction of DTs stands at the fore of future trends the need of speeding up the tree-building process.

Fortunately, EAs are naturally prone to parallelism and the process of artificial evolution can be implemented in various ways (Chitty 2012). There are three main strategies that have been studied for the parallelization and/or distribution of computation effort in EAs:

– master–slave paradigm (Cantu-Paz 2000)—simple parallelization of the most time-consuming operations in each evolutionary loop; the master spreads usually independent tasks or chunks of data over the slaves and, finally, gathers and merges the results;

Communicated by V. Loia.

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

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

– island (coarse-grained) model (Bull et al. 2007)—grouping individuals into subpopulations that are distributed between islands and can evolve independently; some policies are also defined for the migration of individuals between islands each other;
– cellular (fine-grained) algorithm (Llora 2002)—redistribution of single individuals that can communicate only with the nearest individuals for selection and reproduction based on the defined neighborhood topology.

This manuscript concerns massive parallelization of evolutionary induction using graphics processing units (GPUs) (Tsutsui and Collet 2013). The GPUs of modern graphics cards are equipped with hundreds or even thousands of small, energy-efficient computing units (GPU cores) for handling multiple tasks in parallel and managing workloads efficiently. Moreover, they have an unmatched price/performance ratio and enable scale-up on a single workstation, which is simply not achievable using only multi-core CPUs. Thus, not only graphics applications but also general-purpose computation on GPUs (GPGPU) have gained in popularity (Yuen et al. 2013).

In this paper, a GPU-based parallelization of evolutionary induction of DTs is proposed. We focus on one of the most common data mining applications—classification. In particular, we concentrate on evolutionary-induced univariate classification trees (Kretowski and Grześ 2005). To the best of our knowledge, a study on speeding up the evolutionary induction of DTs using GPGPU, perhaps surprisingly, has not yet been attempted in the literature. Although the GPU computational model differs from the conventional CPU one, the strategy that we apply is similar to the master–slave paradigm. The CPU (master) executes EA steps and assigns computationally demanded tasks to a GPU. The GPU executes the tasks in parallel on its cores that could be considered as slaves. This way, so-called global parallelism (Alba and Tomassini 2002) is preserved and the original sequential algorithm does not change.

The proposed approach is applied to a system called global decision tree (GDT). Its framework can be used for the evolutionary induction of classification (Kretowski and Grześ 2007) and regression (Czajkowski and Kretowski 2014) trees, and the GDT solution concept can be applied in many real-life applications, such as finance (Czajkowski et al. 2015) and medicine (Grześ and Kretowski 2007). The main objectives of this work are to accelerate the GDT system and to enable efficient evolutionary induction of DTs on large-scale data. For these purposes, the proposed parallelization manages to exploit the potential of modern GPUs to handle computing intensive jobs like fitness calculation and leaves the evolutionary flow control and communication tasks to the CPU. This parallel computing model is an alternative to previous attempts to parallelize the GDT solution (Czajkowski et al. 2015), which were based on a hybrid MPI+OpenMP approach.

This paper is organized as follows: Section 2 provides a brief background on DTs, the GPGPU computing model, and most recent related works. Section 3 describes in detail our approach for parallel implementation of evolutionary tree induction. Section 4 presents experimental validation of the proposed solution on artificial and real-life datasets. In the last section, the paper is concluded and possible future work is outlined.

2 Background

Data mining (Fayyad et al. 1996) can reveal important and insightful information hidden in data. However, to effectively identify correlations and patterns within the data, appropriate tools and algorithms are required.

2.1 Decision trees

Decision trees (DTs) (Kotsiantis 2013; Rokach and Maimon 2008) represent one of the main techniques for discriminant analysis in data mining. They have a knowledge representation structure that is built of nodes and branches, where each internal node holds a test on one or more attributes; each branch represents the outcome of a test; and each leaf (terminal node) is designed by a class label. Most tree inducing algorithms partition the feature space with axis-parallel hyper-planes. These types of trees are called univariate decision trees as the split at each non-terminal node involves a single feature.

The success of tree-based approaches can be explained by their ease of application, fast operation, and effectiveness. Furthermore, the hierarchical tree structure, where appropriate tests from consecutive nodes are sequentially applied, closely resembles the human way of making decisions. All this makes DTs natural and easy to understand, even for an inexperienced analyst. Despite 50 years of research on DTs, some open issues still remain (Loh 2014).

Inducing an optimal DT is known to be NP-complete (Hyafil and Rivest 1976). Consequently, practical decision-tree learning algorithms must be heuristically enhanced. The most popular type of tree induction is based on a top-down greedy search (Rokach and Maimon 2005). It starts from the root node, where the locally optimal split (test) is searched according to the given optimality measure. Next, the training instances are redirected to the newly created nodes, and this process is repeated for each node until a stopping condition is met. Additionally, post-pruning (Esposito et al. 1997) is usually applied after the induction to avoid the problem of over-fitting the training data and to improve the generalization power of the predictive model. Some of the most popular representatives of top-down-induced decision trees are the solution proposed by Breiman et al. (1984) called *Classification And Regression Tree (CART)*, the *C*4.5 system proposed by Quinlan (1992), and the *CHAID* algorithm proposed by Kass (1980).

Inducing the DT through a greedy strategy is fast and generally efficient in many practical problems, but it usually produces locally optimal solutions. To mitigate some of the negative effects of locally optimal decisions, EAs were introduced for DT induction (Barros et al. 2012). The strength of such an approach lies in a global search for the tree structure and the tests in the internal nodes. This global induction is obviously much more computationally complex; however, it can reveal hidden regularities that are often undetectable by greedy methods.

2.2 GPGPU and CUDA

Recently, research on parallelization of various evolutionary computation methods (Bacardit and Llora 2013; Chitty 2012) has seemed to focus on GPUs as the implementation platform. The popularity of GPUs results from their high computational power at a relatively low cost. A single workstation equipped with a top-end GPU is more often able to provide a lower price/performance factor than a traditional computer cluster. Moreover, computer clusters are not always accessible and demand more maintenance.

The use of graphics hardware for generic problems has become known as general-purpose computation on GPUs (GPGPU). One of the first and most popular frameworks to facilitate GPGPU is a Compute Unified Device Architecture (CUDA) (Wilt 2013) created by the NVIDIA Corporation. In the CUDA programming model, a GPU (device) is considered as a co-processor that can execute thousands of threads in parallel to handle the tasks traditionally performed by the CPU (host) (see Fig. 1). The GPU engine is a scalable array of streaming multiprocessors (SMs). Each SM consists of a collection of simple streaming processors (called CUDA cores).

The CUDA GPU memory also has a hierarchical structure (NVIDIA 2015). Several types of memories are provided with different scopes, lifetimes, and caching behaviors. They can be grouped into two classes: small, fast on-chip memory (cache, resisters, etc.) and global memory with a larger capacity but much higher latency access. All SMs have access to the whole global memory. As regards on-chip memory, all CUDA cores inside the same SM share some memory space as well as having their own local memory.

CUDA employs a single-instruction multiple-data parallelism (Grama et al. 2003). From a programming perspective, when the CPU delegates a job to the GPU, it calls a kernel that is a function run on the device. Then, a grid of (threads) blocks is created and each thread executes the same kernel code in parallel. Each thread has an ID that allows an assigned part of the data to be computed and/or to make control decisions. Each block of threads is mapped to one of the SMs, and the threads inside the block are mapped to CUDA cores. Blocks can be organized into one- or two-dimensional grids, while threads can be organized into one-, two-, or three-dimensional blocks. The dimension and size in each dimension of grids and blocks are both important factors, and they should be set based on GPU specifications as well as parallelization granularity.

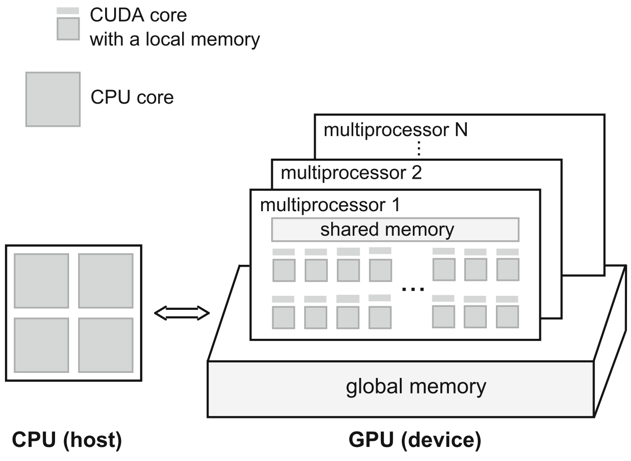

Fig. 1 CUDA hardware model. A GPU is considered as a co-processor to a CPU. The CPU delegates some jobs to the GPU and receives results. The GPU is equipped with two types of memory—small, fast on-chip memory and global memory with larger capacity but much higher latency access. Computational resources are grouped into multiple streaming processors (SMs) consisting of a collection of streaming processors (SPs) (called CUDA cores)

2.3 Parallelization of EA

GPGPU has widely been used to reduce the CPU load and boost the performance of different kinds of computational intelligence methods such as fuzzy systems (Anderson et al. 2008) and neural networks (Oh and Jung 2014). In the field of evolutionary computation (Tsutsui and Collet 2013), GPU parallelization has been applied to many computing techniques, such as ant colony optimization (Cano et al. 2013), evolutionary strategies (Zhu 2011), and differential evolution (Fabris and Krohling 2012; Veronese and Krohling 2010). EAs, due to their parallel nature, have been parallelized in many ways using various techniques (Langdon 2011; Oiso et al. 2011). Application of GPUs in EAs usually focuses on speeding up the evolutionary process (Chitty 2016; Cano et al. 2012) that is relatively slow due to high computational complexity or/and performing large-scale data mining (Bacardit and Llora 2013; Langdon 2013). GPUs have already been successfully applied in machine learning, specifically in speeding up the evaluation process for classification rules

(Cano et al. 2014, 2015) and evolutionary-mined association rules (Cano et al. 2013).

Various proposals have been made with regard to the design and implementation of EAs (Alba and Tomassini 2002). In this paper, we focus on a typical EA framework with a single, unstructured population. Two main decomposition techniques are used to parallelize EAs (Chitty 2012; Freitas 2002): a control approach (also known as a population approach) and a data approach. In the first approach, individuals from the population are evaluated at the same time on different processors. One of the main drawbacks of this approach is the weak scalability to very large datasets. In order to achieve a sufficient parallelization effect, the population size is often much larger than generally employed and can exceed even tens of thousands of individuals (Maitre et al. 2012; Oiso et al. 2011). Moreover, shared-memory systems (like multi-core architectures) can suffer from memory access contention and the number of available processors if often insufficient (Grama et al. 2003). On the other hand, distributed-memory systems (like computer clusters) may have problems with high inter-processor data traffic as well as with storing a copy of large datasets for each processing unit.

The second decomposition technique for parallelizing EAs (applied in this paper) focuses on distributing the dataset across the processors of the parallel system. In the data approach, the objects are evaluated by the individuals in parallel. This technique is considerably much more scalable with respect to the size of the dataset than is the population approach as the entire dataset can be gradually distributed among the local memories of all processors. However, issues with high inter-processor data traffic can still remain. The data parallelization approach for EAs became more popular with the success of GPGPU, which may eliminate or at least hide the communication overhead. The literature on GPGPU in EAs contains algorithms that apply both decomposition techniques (Chitty 2016). In addition, the current research on parallelization of EAs goes even further and proposes additional dimensions of parallelization. In a genetic programming system (Cano and Ventura 2014), the population and data approaches were extended with a GPU-parallel interpreter. A new technique for decomposition focuses on concurrent evaluation of individual's subtrees.

GPGPU has also been used in systems with other structures for EA populations. In Luong et al. (2010), the authors proposed schemes for the island model on GPU architectures in which the islands and the individuals within the islands were run in parallel. The coarse-grained strategy was applied in an evolutionary learning system called BioHEL (Franco et al. 2010), where the authors proposed two-dimensional parallelization that compared all the rules and the instances in the training set in parallel. In the literature, there is also a cellular EA framework on GPUs (Soca et al. 2010) that focuses on a control approach parallelization technique. Another study Franco and Bacardit (2016) proposed three-dimensional parallelization for the fine-grained parallelization strategy. The GPGPU parallelization not only covered the individuals and the instances, but also performed calculations for the attributes within each dataset instance in parallel.

2.4 Related work

Despite the fact that there is a strong need for parallelizing the EA tree-building process (Barros et al. 2012), the topic has not yet been adequately explored. In fact, it has hardly been studied in the literature. One of the reasons is that the straightforward application of GPGPU to EA may be insufficient. In order to achieve high speedup and exploit the full potential of GPGPU parallelization, there is a need to incorporate knowledge about DT specificity and its evolutionary induction.

In one of the few papers that cover parallelization of evolutionary induced DT, a hybrid MPI+OpenMP approach (Czajkowski et al. 2015) was investigated. The algorithm used the master–slave paradigm, and the most time-consuming operations, such as fitness evaluation and genetic operators, were executed in parallel on slaves. The authors applied the control parallelization approach in which the population was evenly distributed to the available nodes and cores. The experimental validation, performed on artificial datasets with different sizes (from 10,000 to 1,000,000 instances, and from 2 to 10 attributes), showed that the hybrid parallelization approach for evolutionary-induced decision trees managed to achieve a speedup of up to $\times 15$ with 64 CPU cores.

So far in the literature, two types of DT systems using GPU-based parallelization have been investigated. In the first one, GPU-based parallelization of the greedy induction of DTs was examined. One of the propositions was a CUDT system (Lo et al. 2014) that parallelized the top-down induction process of a single tree. The GPGPU was used to perform a parallel search through the attributes in each internal node in order to find the best locally optimal splits. The authors showed experimentally that their approach managed to reduce the induction time of a typical decision tree from 5 to 55 times when compared with the traditional CPU version. This approach was later extended with a new dataset decomposition strategy (Nasridonov et al. (2014)) as well as processing multiple tree nodes simultaneously (Strnad and Nerat 2016); however, the registered speedups remained similar.

The second type of DT systems covers the parallelization of ensembles of trees, such as random forests. The most straightforward idea was proposed in a CudaRF system (Grahn et al. 2011) that used one CUDA thread to build one tree in the forest. However, such implementation works only with a large number of trees. Experimental results showed that the induction time of the CudaRF trees was between 30 and 50 times faster than other systems with the assumption that the number of trees is 128 or higher. Another level of parallelization was proposed in the GPU random forests designed for data streams (Marron et al. 2014). The authors used a GPU approach for parallelization of the calculations the majority class in the leaves and the splits in the internal nodes.

However, it should be noted that all the aforementioned systems used the GPGPU approach for parallelization trees that were built with a greedy strategy through a process that is known as recursive partitioning. To the best of our knowledge, there are as yet no studies in the literature about the parallelization of evolutionary induced DTs using the GPU-based approach.

3 Globally induced decision trees

This section briefly recalls the GDT system whose general structure follows a typical EA framework (Michalewicz 1996) with an unstructured population and a generational selection. We have limited the GDT system description to a univariate binary classification tree version as this type of the tree is parallelized with the proposed GPU-based approach.

3.1 Representation

The type of EA may be identified by the way the individuals in the populations are represented. A genetic algorithm is typically considered when solutions are encoded in a fixed-length linear string. The tree-encoding schemes usually imply genetic programming (GP), where the solution encodes data and functions (Woodward 2003); however, the border between different types of EAs is vague and debatable.

DTs are complex tree structures in which the number of nodes, the type of tests, and even the number of test outcomes are not known in advance for a given dataset. This is why the tree representation may be more suitable, especially if the entire tree is searched in one EA run. Therefore, in the GDT system, DTs are not specially encoded and are represented in their actual form as typical univariate classification trees. Each test in a non-terminal node concerns only one continuous-valued attribute. Typical inequality tests with two outcomes are applied, but only precalculated candidate thresholds (Fayyad et al. 1996) are considered as potential splits. A candidate threshold for the given attribute is defined as the midpoint between such a successive pair of examples in the sequence sorted by the increasing value of the attribute, in which the examples are characterized by different classes. The GDT systems also allow univariate tests based on nominal attributes or multivariate (oblique) splits in the internal nodes; however, those variants are not considered in our solution.

Additionally, in every node, information about training instances associated with the node is stored. This enables the algorithm to more efficiently perform local structure and test modifications during applications of genetic operators.

3.2 Initialization, selection, and terminal condition

In general, an initial population (default size equals 64 individuals) should be randomly generated and cover the entire range of possible solutions (Crepinsek et al. 2013) to provide enough diversity of individuals. Due to the large search space, the application of greedy heuristics in the initialization phase is often considered to improve the EA computation time. The downside of this strategy is the possibility to trap EA in the local optima. Therefore, while creating the initial population, a good trade-off between a high degree of heterogeneity and a relatively low computation time is usually desired.

In the GDT system, the initial individuals are created by applying a simple top-down algorithm based on the dipolar principle (Kretowski 2004) to randomly chosen subsamples of the original training data (default: 10 % of data, but not more than 500 instances). Among instances located in the considered node, two objects from different classes are randomly chosen. An effective test that separates these two objects into subtrees is randomly created, taking into account only attributes with different feature values. Recursive partitions are repeated until the stopping criterion is met. Finally, the resulting tree is post-pruned based on the fitness function.

Ranking linear selection (Michalewicz 1996) is used as a selection mechanism. Additionally, in each iteration, a single individual with the highest value of fitness function in the current population is copied to the next one (elitist strategy). Evolution terminates when the maximum number of generations (default value: 1000) is reached.

3.3 Genetic operators

To maintain genetic diversity, the GDT system applies two specialized genetic meta-operators corresponding to classical mutation and crossover. Both operators influence the tree structure and the tests in non-terminal nodes. They are applied with a given probability to a tree (default value is 0.8 for mutation and 0.2 for crossover). Successful application of any operator results in the necessity for relocation of the learning instances between tree parts rooted in the modified nodes.

Each crossover begins by randomly selecting two individuals that will be affected. The next step is choosing the positions in both individuals. Depending on the recombination variant, randomly selected nodes may:

- exchange subtrees (if they exist) randomly or based on the mixed dipole principle (Kretowski and Grześ 2007);
- exchange tests associated with the nodes (only when non-terminal nodes are chosen and the numbers of outcomes are equal) randomly or based on the mixed dipole principle;
- exchange branches in random order, which starts from the selected nodes (only when non-terminal nodes are chosen and the numbers of outcomes are equal);
- transfer subtrees asymmetrically where the subtree of the first/second individual is replaced by a new one that was duplicated from the second/first individual. The replaced subtree starts in the node denoted as a receiver, and the duplicated subtree starts in the node denoted as a donor. In contrast to the symmetric crossovers, two nodes in each individual are modified as both trees. It is preferred that the receiver node has a high classification error because it is replaced by the donor node that should have a small value of classification error as it is duplicated. The application of this variant is more likely to improve the affected individuals because with higher probability, the good nodes are duplicated and they replace the weak nodes.

The mutation operator begins by randomly choosing the node type (equal probability of selecting a leaf node or an internal node). Next, the ranked list of nodes of the selected type is created, and a mechanism analogous to the ranking linear selection is applied to decide which node will be affected. Depending on the type of node, the ranking takes into account:

- 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. Therefore, internal nodes in the lower parts of the tree are mutated with a higher probability.
- number of misclassified objects—nodes with a higher error per instance are more likely to be mutated. In addition, pure nodes (nodes with all instances from one class) are not mutated.

Modifications performed by the mutation operator depend on the node type (i.e., if the considered node is a leaf node or an internal node) and cover different variants:

- shift the thresholds of the tests in the internal nodes;
- replace the test in the internal node with a new one based on the dipole principle;
- prune the internal nodes or expand the leaves that contain objects from different classes.

3.4 Fitness function

The evolutionary search process is very sensitive to proper definition of the fitness function, which drives the evolutionary search process by measuring how good a single individual is in terms of meeting the problem objective. In the context of DTs, a direct minimization of the reclassification quality measured on the learning dataset usually leads to an over-fitting problem and poor performance on unseen, testing observations because the trees are overgrown. In typical top-down induction of DTs (Rokach and Maimon 2005), this problem is partially mitigated by defining a stopping condition and by applying post-pruning (Esposito et al. 1997). In the case of evolutionary induced DTs, this problem may be mitigated by a complexity term incorporated into the fitness function. In the GDT system, the fitness function is maximized and has the following form:

$$\text{Fitness}\,(T) = Q_{\text{Reclass}}(T) - \alpha \cdot (\text{Comp}\,(T) - 1.0), \qquad (1)$$

where $Q_{\text{Reclass}}(T)$ is the reclassification quality of the tree T and α is the relative importance of the classifier complexity (default value is 0.001). The tree complexity term Comp (T) can be viewed as a penalty for over-parametrization. It equals the tree size, which is usually considered to be the number of nodes. The penalty associated with the classifier complexity increases proportionally with the tree size and prevents over-fitting. Subtracting the value 1.0 eliminates the penalty when the tree is composed of only one leaf. A similar idea is used in cost complexity pruning in the CART system (Breiman et al. 1984).

4 GPU-based approach for GDT

The proposed parallelization of the DT evolutionary inducer is based on the sequential GDT algorithm for univariate classification trees. The general flowchart of our GPU-based approach is illustrated in Fig. 2 and in Listing 1. It can be seen that the evolutionary induction is run in a sequential manner on a CPU, and the most time-consuming operations (evaluation of the individuals) are performed in parallel on a GPU. This way, the parallelization does not affect the behavior of the original EA.

The initialization phase (see Fig. 2) begins by sending the whole dataset to the GPU and saving it in the allocated space in the global memory. This CPU to GPU data transfer is performed only once, and the data are kept on the GPU till evolutionary induction stops. This way, data transfer is substantially reduced, especially for large datasets, and each GPU thread has access to these data. Next, the creating an initial population, as well as selection, is performed on the CPU. This step is not parallelized as it is performed only once,

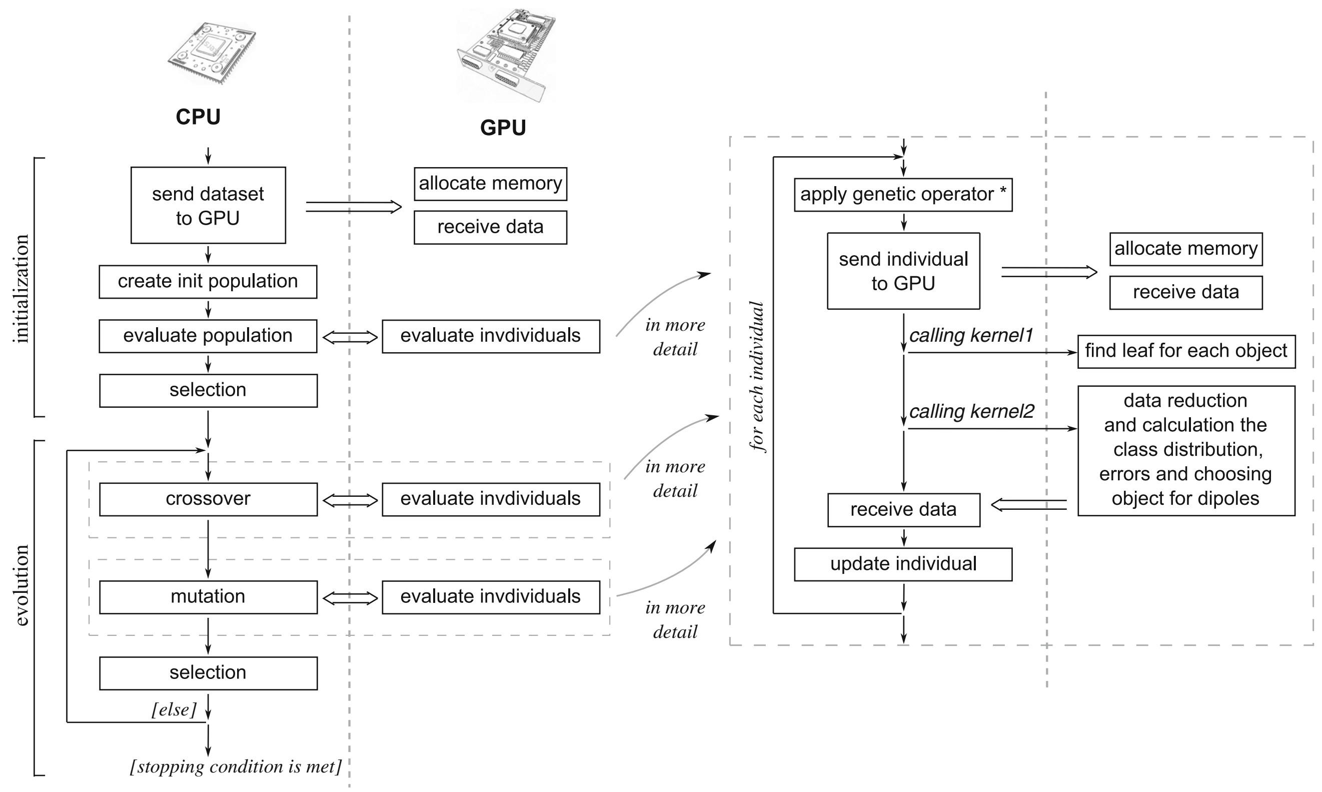

Fig. 2 Flowchart for the GPU-based approach of the evolutionary-induced DT algorithm (* genetic operator is not applied for the evaluation of the initial population)

and the population is created on a small fraction of the dataset. Only the evaluation of initial individuals (fitness calculation) is delegated to the GPU. In the evolutionary loop, genetic operators (without individual evaluation) and selection also run on the CPU as these operations are relatively fast.

In the GDT system, the evaluation (fitness calculation) of the individuals in the population is the most time-consuming operation, which is typical in EAs. In the case of DTs, all objects in the training dataset need to be passed through the tree starting from the root node to an appropriate leaf. Therefore, when there is a need to evaluate an individual after successive crossover or mutation, the GPU is called to perform the calculations. At first, the affected individual is sent to the GPU. Then, the CPU asks the GPU to take on some of its work. Two kernel functions are called. The first kernel (*kernel1* in Fig. 2) is called to propagate objects from the tree root to the leaves. The dataset is spread into smaller parts, first between different GPU blocks and then further between the threads.

Next, the second kernel function (*kernel2* in Fig. 2) merges information about the objects' location in the leaves. Then, the class distributions and classification errors are calculated and propagated from the leaves toward the tree root. In addition, the second kernel function stores in each tree node two randomly selected objects from each class that may take a part in the genetic operators (e.g., as a dipole) that will run in the next evolutionary loop. Both the tree statistics (class distribution, errors) and the selected objects are sent back to the CPU that uses them to update the affected individual. The following sections describe in greater detail the data decomposition and merge strategies (Sect. 4.1) and additional optimizations that shorten the evaluation time of the individuals on the GPU (Sect. 4.2).

4.1 Data decomposition and merge strategy

The first kernel function (see Fig. 2) uses the data decomposition strategy illustrated in Fig. 3. The dataset is decomposed at two levels. At first, the whole dataset is spread into smaller parts that are processed by different GPU blocks. Next, in each block, the objects from the fraction of the dataset are spread further over the threads.

In Fig. 4, which illustrates the GPU-based individual evaluation, we see that all blocks of threads process the same tree but with different data chunks. The role of each GPU block is to counter for each leaf the objects of each class from the assigned part of the data that reach the leaves. Every GPU block has a copy of the individual, which is loaded into shared memory, and the threads within the block count objects from different parts of the data in parallel [step (1)]. In addition,

Listing 1 Pseudo code of the main procedures of the GPU-based approach for the evolutionary induced DTs.

```
__global__
procedure kernel1(classDist, dipoles, indivTab)
  index=0;

  for i=1 to nObjectsToCheck do
    node=indivTab[index];
    while true do
      if node is leaf then
        //increment class counter
        temp=index*N_CLASSES+dataset[i].classId;
        atomicAdd(classDist[temp], 1);

        //save objects for dipoles
        setDipoles(dipoles, dataset[i]);
        break;
      else
        if dataset[i][node.attr] > node.value then
          index = index*2+1; //left child
        else
          index = index*2+2; //right child
        end if
      end if
    end while
  end for
end

__global__
procedure kernel2(classDist, dipoles, results)
  __shared__ classDistSum[N_NODE*N_CLASSES];
  __shared__ dipolesRandom[N_NODE*N_CLASSES*N_DIPOLES];

  //merge data collected in kernel1
  for i=1 to N_NODES*N_CLASSES do
    atomicAdd(classDistSum[i], classDist[i]);
  end for

  //merge data collected in kernel1
  for i=1 to N_NODES*N_CLASSES*N_DIPOLES do
    if dipoles[i]!=0 then
      atomicCAS(dipolesRandom[i], false, dipoles[i]);
    end if
  end for

  if threadIdx.x==0 then
    //calculate errors in the leafs
    for i=N_NODES to 1 do
      index=i*N_CLASSES;
      if classDistSum[index]>classDistSum[index+1] then
        results[index ]=classDistSum[index ];
        results[index+1]=classDistSum[index+1];
      else
        results[index ]=classDistSum[index+1];
        results[index+1]=classDistSum[index ];
      end if
    end for

    //propagate errors to the tree root
    for i=N_NODES to 1 do
      index = i*N_CLASSES
      if i\%2 then
        results[i-2]+=classDistSum[index ];
        results[i-1]+=classDistSum[index+1];
      else
        results[i-1]+=classDistSum[index ];
        results[i ]+=classDistSum[index+1];
      end if
    end for

    //propagate class distribution to the tree root
    //in analogy
    ...
    //propagate dipoles to the tree root
    //in analogy
    ...
  end if
end

procedure evaluateIndividual(indiv)
  copyTreeToTable(indiv, indivTab, indiv.getRoot());
  allocateMemoryAtGPU(indivTab);
  sendDataToGPU(indivTab);

  kernel1<<N_BLOCKS, N_THREADS>>(classDist, dipoles,
       indivTab);

  cudaDeviceSynchronize();

  kernel2<<N_INDIV, N_BLOCKS>>(classDist, dipoles,
                              results);
  cudaDeviceSynchronize();

  updateIndividual(indiv, results); //show in Listing 2
  deallocateMemoryAtGPU(indivTab);
end

procedure main()
  allocateMemoryAtGPU(dataset);
  sendDataToGPU(dataset);
  createInitPopulation();
  evaluatePopulation();
  selection();

  while !stopCondition do
    for all indiv in individuals do
      mutation(indiv);
      evaluateIndividualAtGPU(indiv);
      crossover(indiv);
      evaluateIndividualAtGPU(indiv);
    end for
    selection();
  end while

  deallocateMemoryAtCPU(dataset);
end
```

two objects of each class are randomly selected in each tree leaf [step (2)].

The role of the second kernel function is to merge information from multiple copies of the individual allocated in each GPU block. In Fig. 4, one can see that after the merge operation, there is only one single tree that gathered information for the whole dataset. The total number of objects of each class in each tree leaf is a sum of counters from copies of the individual [step (3)]. In addition, in each tree leaf, two objects of each class are randomly selected from the objects provided by the first kernel function [step (4)].

Finally, we calculate reclassification errors in each leaf [step (5)] and propagate all gathered information: class distribution, stored objects, and errors from the leaves toward the root node [step (6)]. The reason why the two objects from different classes are stored in each tree node is because the CPU does not have access to the objects that fall in particular nodes of the tree. Although the CPU does not need

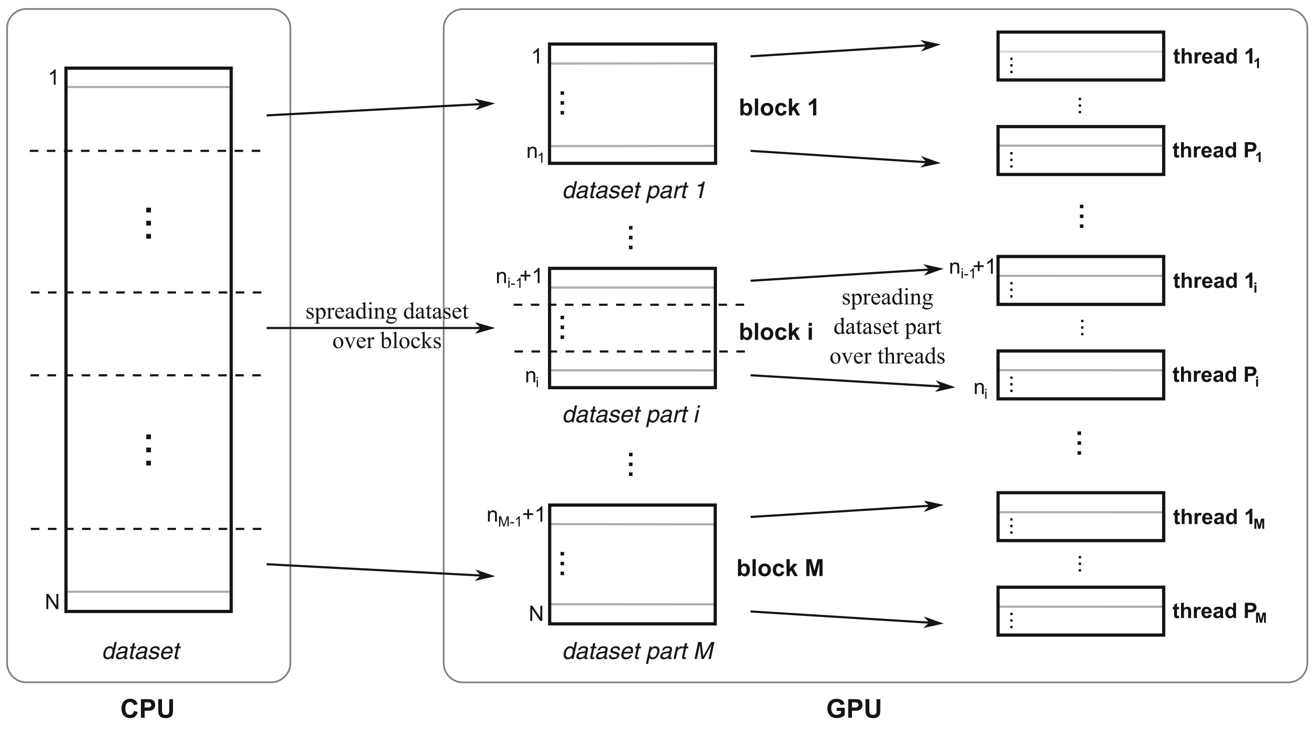

Fig. 3 Data decomposition strategy. The dataset is spread into smaller parts. Each part is processed by different GPU blocks. Objects inside the dataset parts are spread further over block threads

access to all objects from the dataset, pure and mixed (object) dipoles are required for some variants of genetic operators (Kretowski 2004). With two objects from different classes, the CPU can quickly and easily constitute such dipoles. When the CPU receives objects stored earlier by the GPU, dipoles are created and stored in each tree node when an individual is updated. This way, the CPU does not need to locate the objects in a node at all, which would take much more time.

4.2 Optimization and implementation aspects

After the successful application of a genetic operator on the CPU, the propagation of objects from the tree root toward the leaves is performed in the first kernel function on the GPU. This process of associating each object with an appropriate leaf is very time-consuming, especially on large datasets. In order to eliminate the propagation of all dataset objects, we propose processing only the modified part of the tree (see Fig. 5 and Listing 2). This way, instead of relocating all objects in the entire tree, we only update a part of the data in the node (together with all subnodes) that was affected.

The GDT assumption that internal nodes at lower parts of the tree are mutated with higher probability could improve the benefit of the proposed optimization, as it is expected that the lower parts of the tree held fewer objects that need to be assigned. When this optimization mechanism is switched on, the CPU sends to the GPU only information about the affected node (together with all subnodes) and the route from this node to the tree root to update tree statistics.

In order to better suit the GPU computation and yield efficient GPU memory management (Wilt 2013), the individuals' representation differs between the CPU and GPU. Before an individual is transferred to the GPU, its flat representation (one-dimensional array) is created based in its binary tree representation used by the CPU. Such a flat representation is also used during GPU computation. Figure 6 illustrates how each tree node is assigned to a specific position in the array. The size of the array is fixed to the height of the tree so it is large enough to hold any binary tree of this height. Each tree node can store t elements, such as information about the split, node statistics (e.g., errors, class distribution), and indexes of the stored objects. With the fixed size of the array, it is easy to find positions of a specific node together with its stored information, which for the i-th node are in the range $<t * i, t * i + t - 1>$. The position of the left and right child of the i-th node equals $(2 * i + 1)$ and $(2 * i + 2)$, respectively.

5 Experimental validation

This section shows the performance of the proposed parallel version of the GDT system. Experiments were performed on artificial and real-life datasets and with different NVIDIA

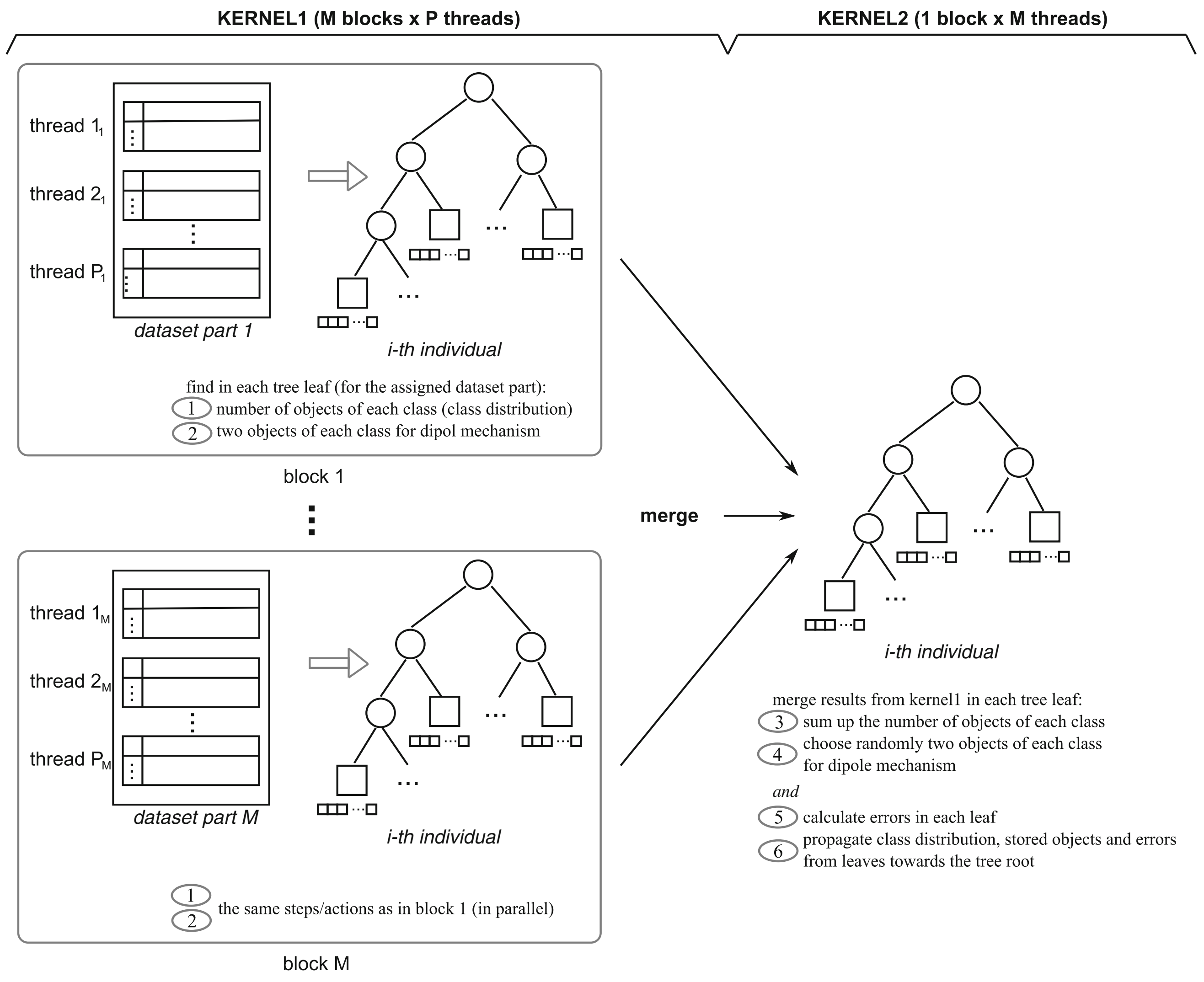
KERNEL1 (M blocks x P threads)
KERNEL2 (1 block x M threads)
thread 1_1
thread 2_1
thread P_1
dataset part 1
i-th individual
find in each tree leaf (for the assigned dataset part):
1 number of objects of each class (class distribution)
2 two objects of each class for dipol mechanism
block 1
thread 1_M
thread 2_M
thread P_M
dataset part M
i-th individual
1 2 the same steps/actions as in block 1 (in parallel)
block M
merge
i-th individual
merge results from kernel1 in each tree leaf:
3 sum up the number of objects of each class
4 choose randomly two objects of each class for dipole mechanism
and
5 calculate errors in each leaf
6 propagate class distribution, stored objects and errors from leaves towards the tree root

Fig. 4 The same individual is processed in parallel by successive blocks (of threads) that are responsible for the different parts of the dataset. Each block saves the results in its copy of the individual. Threads inside the blocks share the same copy of the individual. Kernel1 function is responsible for propagation of the objects to the tree leaves, while kernel2 merges the results from different blocks and propagates tree statistics from the leaves toward the tree root

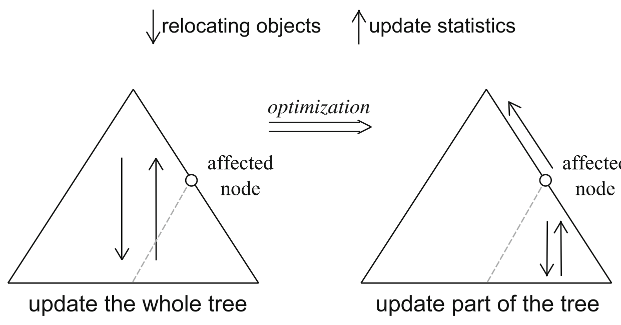

Fig. 5 The optimization mechanism to process only part of the tree when an internal node or a leaf is affected

graphics cards. We wanted to test large-scale datasets, and therefore, we have concentrated mainly on sets with at least 1 million objects. In this paper, we focused only on the time performance of the GDT system; therefore, results for the classification accuracy are not included. However, for all tested artificial datasets, the GDT system managed to induce trees with optimal structures and almost perfect accuracies (99–100 %). For detailed information about accuracy performance of the GDT system, we refer readers to our previous papers (Kretowski and Grześ 2005, 2007; Czajkowski and Kretowski 2014).

5.1 Setup

Experimental verification was performed with the univariate classification version of the GDT system. All presented results correspond to averages of 20 runs and were obtained with a default set of parameters from the sequential version of the GDT system. In the fitness evaluation, the pro-

Listing 2 Pseudo code of the main procedures improved to process only a part of the tree.

```
__global__
procedure kernel1(classDist, dipoles,indivTab)
  index=0;

  for i=1 to nObjectsToCheck do
    node=indivTab[index];
    while true do
      //check if we are already
      //in the non-affected tree part
      if node.attr == -2 then
      //TREE_PART_ALGORITHM
        break;
      end if

      if node is leaf then
      //the rest looks like kernel1 in Listing 1
       ...
    end while
  end for
end

procedure copyNodeToTable(node, indivTab, index)
  indivTable[index]=node;
  if node.hasLeftChild() then
    copyNodeToTable(node.getLeftChild(),
                    indivTab, index*2+1)
  end if
  if node.hasRightChild() then
    copyNodeToTable(node.getRightChild(),
                    indivTab, index*2+2)
  end if
end

procedure copyTreeToTable(indiv, indivTab,
    affectedNode)
  if affectedNode==root then
    copyNodeToTable(root, indivTab, 0);
  else//TREE_PART_ALGORITHM
    index=findIndex(indiv, affectedNode);
    copyNodeToTable(affectedNode, indivTab,
        index);
    markedPathFromNodeToRoot(affectedNode,
        root, -2);
  end if
end

procedure updateIndividual(indiv, results)
  if TREE_PART_ALGORITHM then
    node=indiv.getAffectedNode();
    fillFromAffectedNodeToRoot(indiv, node,
        results);
  else
    node=indiv.getRoot();
  end if

  fillSubTrees(node, results);
end
```

posed algorithm processed only the modified part of the tree (individual), as described in Sect. 4.2. The impact of this optimization on the overall algorithm speedup is discussed in Sect. 5.3. The size of the data processed in each block and thread was determined experimentally. We tested different blocks × threads configurations (see Sect. 5.3) and selected 256 × 1024 for the datasets with no more than 1 million instances and 1024 × 1024 for the larger datasets.

Experimental validation was performed on the datasets described in Table 1:

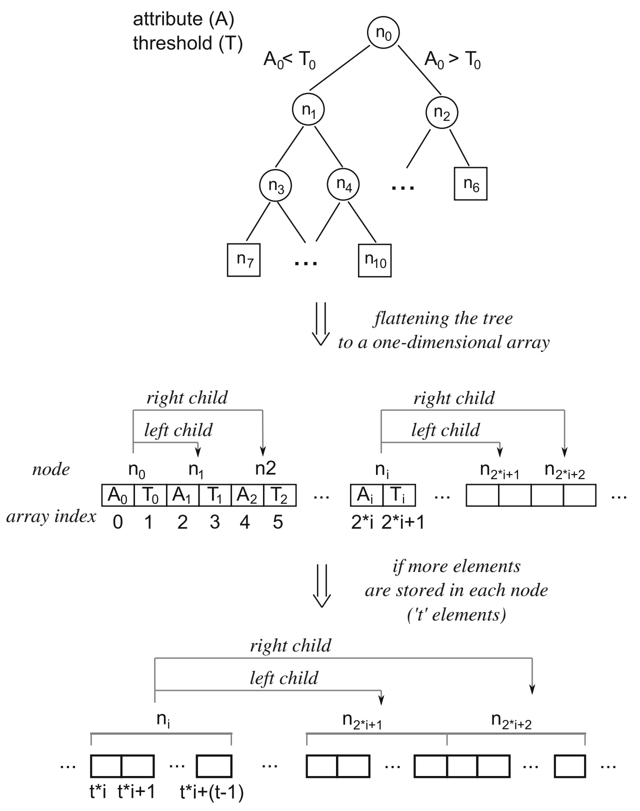

Fig. 6 Flattening of the tree structure on the GPU to a one-dimensional array [tree node (n), attribute (A), threshold (T), by default $t = 2$]

Table 1 Characteristics of the datasets: name, number of instances, number of attributes, and number of classes

Dataset	Instances	Attributes	Classes
Chess1M	1,000,000	2	2
Chess5M	5,000,000	2	2
Chess10M	10,000,000	2	2
Chess20M	20,000,000	2	2
Suzy	5,000,000	18	2
Higgs	11,000,000	28	2

– Chess dataset—an artificially generated dataset in which objects are arranged on a 3 × 3 chessboard (Czajkowski et al. 2015). Different sizes and different numbers of the dataset attributes were analyzed (additional attributes were randomly generated).
– Higgs dataset—one of the largest real-life datasets available in the UCI Machine Learning Repository (Blake et al. 1998). It concerns a classification problem to distinguish between a signal process that produces Higgs bosons and a background process that does not;
– Suzy dataset—a real-life dataset from the UCI repository that covers the problem of distinguishing between a signal process that produces super-symmetric particles and a background process that does not.

All the experiments were performed on a regular PC equipped with a quad-core processor (Intel Core i7-870, 8M Cache, 2.93 GHz), 32 GB RAM, and a single graphics card. We used a 64-bit Ubuntu Linux 14.04.02 LTS as an operating system. The sequential algorithm was implemented in C++ and compiled with the use of GCC version 4.8.2. The GPU-based parallelization was implemented in CUDA-C and compiled by nvcc CUDA 7.0 (NVIDIA 2015) (single-precision arithmetic was applied). We tested a group of different NVIDIA graphics cards, described in Table 2. For each graphics card, we gathered basic specifications that covered the number of CUDA cores, as well as clock rate, available memory, bandwidth, and price.

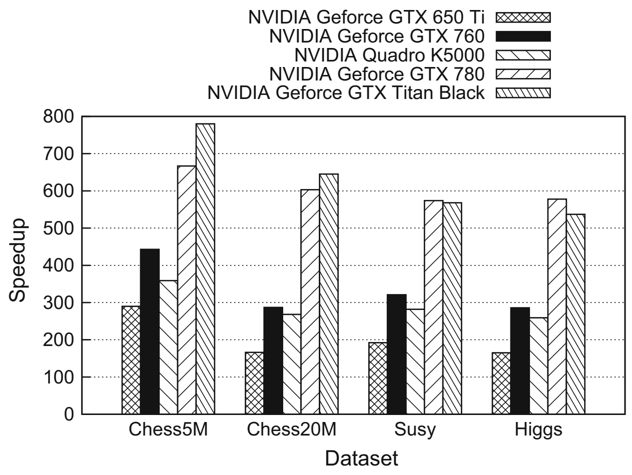

Fig. 7 Mean speedup for selected datasets and various GPUs

5.2 Obtained results

Figure 7 presents the mean speedup of the proposed GPU-accelerated approach in comparison with the sequential GDT solution for different datasets and various GPUs. More details, including the mean speedup for the multi-core OpenMP version (Czajkowski et al. 2015), are demonstrated in Table 3. In addition, at the bottom of the table, we provide the mean execution time for the sequential algorithm as well as for the GPU-based solution on the fastest GPU.

The experimental results suggest that with the proposed approach, even a regular PC with a cheap graphics card is sufficient for accelerating the GDT tree induction time over 100 times. As expected, better graphics cards manage to achieve much better improvement, the scale of which is quite surprising. The most expensive tested graphics card in comparison with the cheapest one is often more than 2 times faster and is able to induce trees almost 800 times faster than the sequential GDT solution. With such a high acceleration of the tree induction, the speedup achieved by the quad-core CPU using only OpenMP parallelization is—to put the mildly—not very impressive.

The scale of the improvement is even more visible when comparing the execution time between the sequential and parallel version of the GDT system. The results presented at the bottom of Table 3 show that the tree induction time for the proposed solution can be counted in seconds and minutes, whereas the original sequential solution often needs several days. Moreover, the GPU-based accelerated solution has lower memory consumption as there is no need to store in every node information about training objects associated with the node (used in the sequential version to accelerate the evaluation of individuals). We verified the execution times of the GPU-based induction for even larger datasets—for 50,000,000 and 100,000,000 objects, and they were about 64 min and 2.5 h, respectively. Because of very long computation times and high memory consumption, these datasets were not processed using the sequential version of the algorithm.

In the results reported in Table 3, we can also see that the speedup for the smallest tested dataset is much worse than for the rest of the analyzed datasets. The explanation of this observation can be found in Fig. 8, which illustrates in detail the time-sharing information of the proposed approach. It can be seen that the propagation of the dataset objects to the leaves (first kernel function), which is the most time-expensive algorithm phase, takes only one-third of the total evolutionary induction time on the *Chess1M* dataset, whereas

Table 2 Processing and memory resources of the NVIDIA graphics cards used in the experiments

NVIDIA graphics card	Engine		Memory		≈Price ($)
	No. CUDA cores	Clock rate (MHz)	Size (GB)	Bandwidth (GB/s)	
Geforce GTX 650 Ti	768	828	2	86.4	122
Geforce GTX 760	1152	980	2	192.2	139
Quadro K5000	1536	706	4	173.0	1679
Geforce GTX 780	2304	863	3	288.4	588
Geforce GTX Titan Black	2880	889	6	336.0	1899

Prices provided at http://www.videocardbenchmark.net/, updated November 29, 2015, are also included

Table 3 Mean speedup for different datasets and various GPUs

	Chess1M	*Chess5M*	*Chess10M*	*Chess20M*	*Suzy*	*Higgs*
Speedup						
Intel Core i7—4 cores (OpenMP)	×3.2	×3.3	×3.1	×3.0	×2.5	×2.4
NVIDIA Geforce GTX 650 Ti	×148	×290	×220	×166	×192	×165
NVIDIA Geforce GTX 760	×220	×443	×357	×287	×321	×286
NVIDIA Quadro K5000	×146	×359	×303	×268	×282	×259
NVIDIA Geforce GTX 780	×250	×667	×588	×603	×574	×578
NVIDIA Geforce GTX Titan Black	×302	×781	×669	×653	×568	×537
Time						
Intel Core i7—sequential	29,000 s ≈8 h	191,249 s ≈2 days	322,000 s ≈4 days	727,000 s ≈8 days	159,134 s ≈2 days	350,000 s ≈4 days
The shortest time	96 s ≈1.5 min	245 s ≈4 min	481 s ≈8 min	1113 s ≈18.5 min	277 s ≈4.5 min	606 s ≈10 min

The mean speedup obtained for the multi-core OpenMP version is also provided. Below speedups, the mean execution time of the sequential algorithm as well as its GPU-accelerated version on the fastest GPU is presented (in sec and days/h/min)

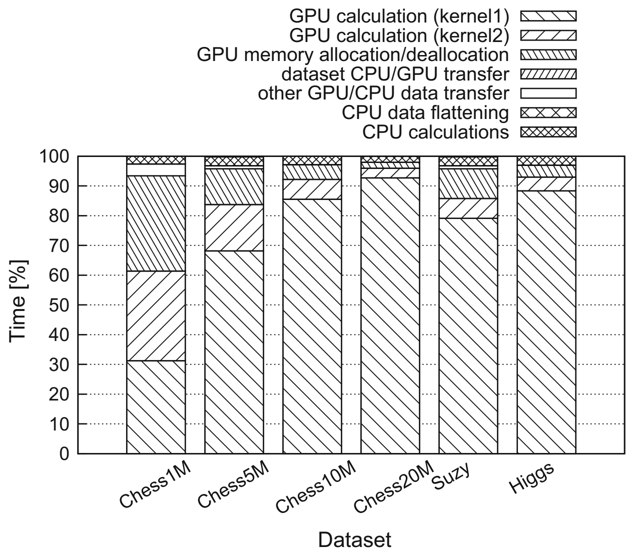

Fig. 8 Detailed time-sharing information (mean time as a percentage) of the GPU-accelerated execution on Geforce GTX Titan Black for various datasets

for the *Chess20M* dataset, it is almost 95 %. This means that other algorithm parts (like the second kernel function or GPU memory allocation/deallocation) are also important in the case of the smallest dataset. Because the execution times for these other parts of the algorithm are only tree size dependent (but not dataset size dependent), their time contribution in the whole algorithm execution decreases when the dataset size grows. Thus, for larger datasets, actions like propagating the statistics from the leaves to the root node (second kernel function) or memory allocation/deallocation on the GPU take a much smaller fraction of the total tree induction time. As expected, the time spent by the CPU on calculations is almost equal for all the datasets (about 2–3 %) and is very short in comparison with the GPU computation time. The influence of the rest of the operations, such as transferring the trees from/to GPU/CPU or the tree representation flattening, is almost unnoticeable.

In Sect. 4.2, we proposed an optimization of the GPU-accelerated algorithm that concerns processing only the modified part of the tree (individual) instead of the whole tree. To check the benefit (if any) of relocating only a part of the data that falls into the modified nodes, additional experiments were performed. In Fig. 9, we can observe noticeable improvement in the speedup when only a part of the tree is updated on all four variants of the *Chess* dataset using two different graphics cards (Geforce GTX Titan Black and Geforce GTX 780).

Performance of the proposed optimization strongly depends on the size of the individual. As the lower parts of the tree in the GDT system are modified more often, the benefit of the proposed improvement should be higher on larger trees. However, even for the tested *Chess* datasets, in which the induced trees are not very big (8 internal nodes and 9 leaves), the speedup increased almost 50 % in comparison with the algorithm without this improvement.

We also experimentally checked whether different sizes of the data processed in each block/thread influences the algorithm speedup. Figure 10 presents the mean speedup for the *Chess* dataset with a different number of objects for one of the graphics cards (Geforce GTX Titan Black). It can be observed that for all larger datasets (starting with *Chess5M*), the configuration of blocks × threads equal to 1024 × 1024 fits the best, whereas *Chess1M* configuration 256 × 1024 gives noticeably better results.

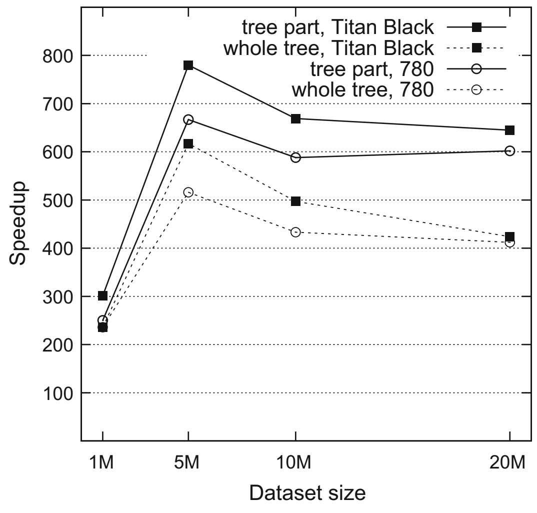

Fig. 9 The influence of the proposed improvement to process only the modified part of a tree (individual) at the GPU instead of the whole tree. The mean speedup when the improvement is switched on (*solid lines*)/off (*dotted lines*) for the different sizes of the *Chess* dataset (1, 5, 10, and 20 million instances) using Geforce GTX Titan Black as well as Geforce GTX 780 is shown

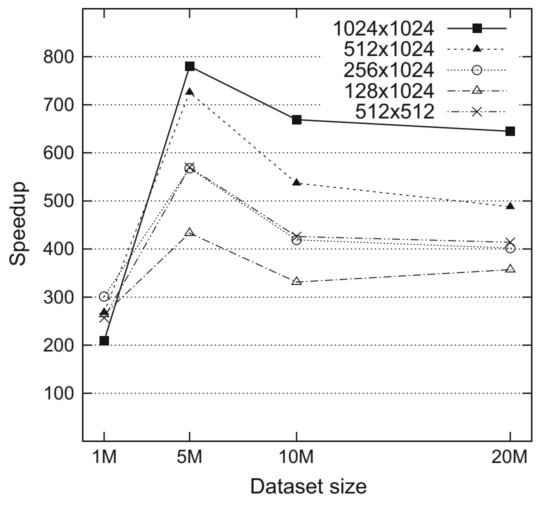

Fig. 10 The influence of the number of blocks × threads on the performance of the GPU-accelerated algorithm. The mean speedup for a few blocks × threads configurations and the different sizes of the *Chess* dataset (1, 5, 10, and 20 million instances) using Geforce GTX Titan Black is shown

The reason why the smaller number of blocks works better for smaller datasets can lie in the number of objects processed in each thread. If the number of objects in each thread is small (e.g., 1 or 2), the performance level falls. This is the case for the *Chess1M* dataset in which for the configuration of 1024 blocks × 1024 threads, each thread processes a maximum of 2 objects. On the other hand, processing too many objects per thread (e.g., 8 and above) seems to slow down the evolutionary speed as well (see the configuration of 128 blocks × 1024 threads). Unfortunately, the maximum number of threads per block is currently limited to 1024 by graphics card vendors. As regards the number of blocks, it is limited by the approach, as the number of blocks in the first kernel function is equal to the number of block threads in the second kernel function (see Fig. 4).

There are at least two reasons that may explain the described algorithm behavior. The first one concerns the problem with load balancing when the chunks of data are too big. On the other hand, too small data portions could cause more overhead as there are more threads to create, manage, and so on. Interestingly, reducing the number of threads in favor of blocks (512 × 512 instead of 256 × 1024) yielded worse results on the *Chess1M* dataset and almost the same for the rest of the datasets. More effort and research on blocks × threads configurations is needed as this issue is not only dataset dependent but also GPU dependent (Chitty 2016).

5.3 Additional discussion

Within the proposed solution, we tested the different parallelization techniques (population and/or data approach) mentioned in Sect. 2.3. One of the studied cases was a hybrid parallelization that was based on the decomposition of both the population and the data. The original GDT system uses a small number of individuals in the population; therefore, to achieve a better parallelization effect, it was necessary for the population size to be significantly increased (e.g., 100 times). However, performed experiments (not included) showed that even then, the solution proposed in this paper is more efficient.

With our approach, there are sufficient data to load the available multiprocessors and even to saturate the GPU. Moreover, the fine-grained parallelization and data decomposition itself facilitate load balancing. All threads in all blocks process the same tree (individual), and each thread usually processes the same number of objects. Therefore, individuals with different sizes do not affect the overall parallelization performance as all threads last a similar amount of time. The benefits of the population or population-data approach might be more visible with much smaller datasets than the ones validated in the manuscript.

Having the performance results for different graphics cards (Table 3) as well as their prices (Table 2) gives us an opportunity to estimate how much (in dollars) a single speedup unit for particular GPUs costs. Table 4 shows us that although Geforce GTX 780 and Geforce GTX Titan Black provide the best algorithm acceleration, the Geforce GTX 760 is the best graphics card when the price/speedup factor is analyzed. Quadro K5000 achieves the lowest score. Quadro family GPUs are designed to accelerate application to design, rendering, and 3D visual modeling (like CAD software) at the expense of lower computational performance in games

Table 4 Costs (in dollars) of a single speedup unit for various GPUs based on the mean speedups from all tested datasets in Table 3 and prices in Table 2

NVIDIA graphics card	Mean speedup	Price/speedup
Geforce GTX 650 Ti	197	0.62
Geforce GTX 760	319	0.44
Quadro K5000	270	6.23
Geforce GTX 780	543	1.08
Geforce GTX Titan Black	585	3.25

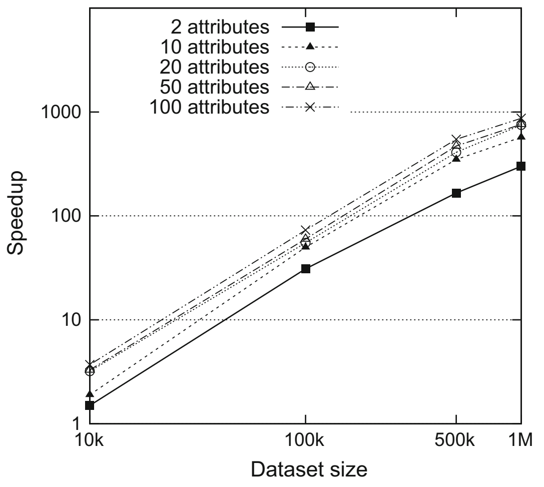

Fig. 11 The influence of the number of objects/attributes on the performance of the GPU-accelerated algorithm. The mean speedup for different sizes of the *Chess* dataset (10,000, 100,000, 5,000,000, and 1,000,000 instances) using Geforce GTX Titan Black is shown

and GPGPU. They offer lower power consumption but are usually more expensive than GTX series GPUs. Regardless, the obtained results show that even an engineering workstation equipped with a Quadro family GPU can be successfully used in the fast evolutionary inductions of DTs.

We also verified the performance of the solutions on datasets with fewer objects but with more attributes (see Fig. 11). It is seen that for 10,000 instances and 2 attributes, the speedup is the worst—a little higher than 1. However, in this case, the calculation time was only 1 min. It is also visible that the speedup grows with an increase in both the number of attributes and number of objects. Because we focus on large-scale data in this article, we left a thorough investigation of high-dimensional data (Cano et al. 2015) for future studies. For such data, a hybrid parallelization (including both population and date decompositions) or only population decomposition (particularly for small dataset objects) would probably be a more efficient approach.

6 Conclusions and future work

The growing popularity of evolutionary-induced DTs can be withheld if there are no effective solutions for improving their speed and ability to analyze large-scale data. In this paper, we propose GPU-based parallelization to extend the GDT system. Even a regular PC equipped with a medium-class graphics card is sufficient for our algorithm to reduce the tree induction time by more than two orders of magnitude. The experiments performed on artificial and real-life datasets presented in Table 3 show that our solution is fast, scalable, and can explore large-scale data (e.g., the tree induction for the dataset with 20,000,000 instances is performed under 20 min, which would take around 8 days for the sequential solution).

We see many promising directions for future research. A number of ideas for tuning the proposed solution, such as the computation and data transfer overlapping mechanism using concurrent execution design, can be investigated. Additional levels of data/population/individual decomposition (Cano and Ventura 2014; Nasridonov et al. 2014) are interesting directions to explore. We also plan to deal with a multi-GPU parallelization to speed up evolutionary induction even further. Hybrid parallelizations, such as MPI/CUDA, are also within the scope of our interest. In addition, we will continue to work with the presented approach to adapt it to the evolutionary induction of regression and model trees.

Acknowledgements This work was supported by the Grants W/WI/2/2014 (first author) and S/WI/2/2013 (third author) from Bialystok University of Technology founded by Ministry of Science and Higher Education as well as by the Polish National Science Center and a Grant allocated on the basis of decision 2013/09/N/ST6/04083 (second author).

Compliance with ethical standards

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

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

Alba E, Tomassini M (2002) Parallelism and evolutionary algorithms. IEEE Trans Evol Comput 6(5):443–462
Anderson DT, Luke RH, Keller JM (2008) Speedup of fuzzy clustering through stream processing on graphics processing units. IEEE Trans Fuzzy Syst 16:1101–1106
Bacardit J, Llora X (2013) Large-scale data mining using genetics-based machine learning. WIREs Data Min Knowl Discov 3:37–61
Barros RC, Basgalupp MP, Carvalho AC, Freitas AA (2012) A survey of evolutionary algorithms for decision-tree induction. IEEE Trans SMC C 42(3):291–312

Blake C, Keogh E, Merz C (1998) UCI repository of machine learning databases. http://www.ics.uci.edu/~mlearn/MLRepository.html

Breiman L, Friedman J, Olshen R, Stone C (1984) Classification and regression trees. Wadsworth Int. Group, Belmont

Bull L, Studley M, Bagnall A, Whittley I (2007) Learning classifier system ensembles with rule-sharing. IEEE Trans Evol Comput 11:496–502

Cano A, Zafra A, Ventura S (2012) Speeding up the evaluation phase of GP classification algorithms on GPUs. Soft Comput 16:187–202

Cano A, Olmo JL, Ventura S (2013) Parallel multi-objective ant programming for classification using GPUs. J Parallel Distrib Comput 73:713–728

Cano A, Luna JM, Ventura S (2013) High performance evaluation of evolutionary-mined association rules on GPUs. J Supercomput 66(3):1438–1461

Cano A, Luna JM, Ventura S (2014) Parallel evaluation of Pittsburgh rule-based classifiers on GPUs. Neurocomputing 126:45–57

Cano A, Ventura S (2014) GPU-parallel subtree interpreter for genetic programming. In: Proceedings of GECCO'14, pp 887–894

Cano A, Luna JM, Ventura S (2015) Speeding up multiple instance learning classification rules on GPUs. Knowl Inf Syst 44(1):127–145

Cantu-Paz E (2000) Efficient and accurate parallel genetic algorithms. Kluwer Academic, Norwell

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

Chitty DM (2016) Improving the performance of GPU-based genetic programming through exploitation of on-chip memory. Soft Comput 20(2):661–680

Crepinsek M, Liu S, Mernik M (2013) Exploration and exploitation in evolutionary algorithms: a survey. ACM Comput Surv 45(3):35:1–35:33

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

Czajkowski M, Czerwonka M, Kretowski M (2015) Cost-sensitive global model trees applied to loan charge-off forecasting. Decis Support Syst 74:55–66

Czajkowski M, Jurczuk K, Kretowski M (2015) A parallel approach for evolutionary induced decision trees. MPI+OpenMP implementation. In: Proceedings of ICAISC'15. Lecture notes in computer science, vol 9119, pp 340–349

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

Fabris F, Krohling RA (2012) A co-evolutionary differential evolution algorithm for solving min-max optimization problems implemented on GPU using C-CUDA. Expert Syst Appl 39(12):10324–10333

Fayyad U, Piatetsky-Shapiro G, Smyth P, Uthurusamy R (1996) Advances in knowledge discovery and data mining. AAAI Press, Palo Alto

Franco MA, Krasnogor N, Bacardit J (2010) Speeding up the evaluation of evolutionary learning systems using GPGPUs. In: Proceedings of GECCO 10. ACM, New York, pp 1039–1046

Franco MA, Bacardit J (2016) Large-scale experimental evaluation of GPU strategies for evolutionary machine learning. Inf Sci 330:385–402

Freitas AA (2002) Data mining and knowledge discovery with evolutionary algorithms. Springer, Secaucus

Grahn H, Lavesson N, Lapajne MH, Slat D (2011) CudaRF: a CUDA-based implementation of random forests. In: Proceedings of IEEE/ACS, pp 95–101

Grama A, Karypis G, Kumar V, Gupta A (2003) Introduction to parallel computing. Addison-Wesley, Reading

Grześ M, Kretowski M (2007) Decision tree approach to microarray data analysis. Biocybern Biomed Eng 27(3):29–42

Hyafil L, Rivest RL (1976) Constructing optimal binary decision trees is NP-complete. Inf Process Lett 5(1):15–17

Kass GV (1980) An exploratory technique for investigating large quantities of categorical data. Appl Stat 29(2):119–127

Kotsiantis SB (2013) Decision trees: a recent overview. Artif Intell Rev 39:261–283

Kretowski M (2004) An evolutionary algorithm for oblique decision tree induction. In: Proceedings of ICAISC'04. Lecture notes in computer science, vol 3070, pp 432–437

Kretowski M, Grześ M (2005) Global learning of decision trees by an evolutionary algorithm. In: Saeed K, Pejaś J (eds) Information processing and security systems. Springer, US, pp 401–410. http://link.springer.com/chapter/10.1007%2F0-387-26325-X_36

Kretowski M, Grześ M (2007) Evolutionary induction of mixed decision trees. Int J Data Wareh Min 3(4):68–82

Langdon WB (2011) Graphics processing units and genetic programming: an overview. Soft Comput 15:1657–1699

Langdon WB (2013) Large-scale bioinformatics data mining with parallel genetic programming on graphics processing units. In: Tsutsui S, Collet P (eds) Massively parallel evolutionary computation on GPGPUs, Springer, Berlin, Heidelberg, pp 311–347

Llora X (2002) Genetics-based machine learning using fine-grained parallelism for data mining. Ph.D. Thesis. Barcelona, Ramon Llull University

Lo WT, Chang YS, Sheu RK, Chiu CC, Yuan SM (2014) CUDT: a CUDA based decision tree algorithm. Sci World J 1–12. http://www.hindawi.com/journals/tswj/2014/745640/

Loh W (2014) Fifty years of classification and regression trees. Int Stat Rev 83(3):329–348

Luong TV, Melab N, Talbi E (2010) GPU-based island model for evolutionary algorithms. In: Proceedings of GECCO '10. ACM, New York, pp 1089–1096

Maitre O, Kruger F, Querry S, Lachiche N, Collet P (2012) EASEA: specification and execution of evolutionary algorithms on GPGPU. Soft Comput 16:261–279

Marron D, Bifet A, Morales GF (2014) Random forests of very fast decision trees on GPU for mining evolving big data streams. In: Proceedings of ECAI, pp 615–620

Michalewicz Z (1996) Genetic algorithms + data structures = evolution programs, 3rd edn. Springer, Berlin

Nasridonov A, Lee Y, Park YH (2014) Decision tree construction on GPU: ubiquitous parallel computing approach. Computing 96(5):403–413

NVIDIA (2015) CUDA C programming guide. Technical report. https://docs.nvidia.com/cuda/cuda-c-programming-guide/

NVIDIA (2015) CUDA C best practices guide in CUDA toolkit. Technical report. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

Oh KS, Jung K (2014) GPU implementation of neural networks. Pattern Recogn 37(6):1311–1314

Oiso M, Matsumura Y, Yasuda T, Ohkura K (2011) Implementing genetic algorithms to CUDA environment using data parallelization. Tech Gaz 18(4):511–517

Quinlan JR (1992) Learning with continuous classes. In: Proceedings of AI'92, World Scientific, pp 343–348

Rokach L, Maimon OZ (2005) Top–down induction of decision trees classifiers—a survey. IEEE Trans SMC C 35(4):476–487

Rokach L, Maimon OZ (2008) Data mining with decision trees: theory and application. Mach Percept Artif Intell 69. http://www.worldscientific.com/worldscibooks/10.1142/6604

Soca N, Blengio JL, Pedemonte M, Ezzatti P (2010) PUGACE, a cellular evolutionary algorithm framework on GPUs. In: Proceedings of IEEE congress on evolutionary computation (CEC), pp 1–8

Strnad D, Nerat A (2016) Parallel construction of classification trees on a GPU. Concurr Comput Pract Exp 28(5):1417–1436

Tsutsui S, Collet P (2013) Massively parallel evolutionary computation on GPGPUs. Springer, Berlin

Veronese L, Krohling R (2010) Differential evolution algorithm on the GPU with C-CUDA: In: Proceedings of IEEE congress on evolutionary computation (CEC), pp 1–7

Wilt N (2013) Cuda handbook: a comprehensive guide to GPU programming. Addison-Wesley, Reading

Woodward JR (2003) GA or GP? That is not the question. In: Proceedings of IEEE CEC, pp 1056–1063

Yuen D, Wang L, Chi X, Johnsson L, Ge W (2013) GPU solutions to multi-scale problems in science and engineering. Springer, Berlin

Zhu W (2011) Nonlinear optimization with a massively parallel evolution strategy–pattern search algorithm on graphics hardware. Appl Soft Comput 11:1770–1781