Compact In-Memory Representation of Decision Trees in GPU-Accelerated Evolutionary Induction

Krzysztof Jurczuk(✉), Marcin Czajkowski, and Marek Kretowski

Faculty of Computer Science, Bialystok University of Technology, Wiejska 45a, 15-351 Bialystok, Poland
{k.jurczuk,m.czajkowski,m.kretowski}@pb.edu.pl

Abstract. Decision trees (DTs) are popular techniques in the field of explainable machine learning. Traditionally, DTs are induced using a top-down greedy search that is usually fast; however, it may lead to sub-optimal solutions. Here, we deal with an alternative approach which is an evolutionary induction. It provides global exploration that results in less complex DTs but it is much more time-demanding. Various parallel computing approaches were considered, where GPU-based one seems to be the most efficient. To speed up the induction further, different GPU memory organization/layouts could be dealt with.

In this paper, we introduce a compact in-memory representation of DTs. It is a one-dimensional array representation where links between parent and children tree nodes are explicitly stored next to the node data (testes in internal nodes, classes in leaves, etc.). On the other side, when the complete representation is applied, children positions are calculated based on the parent place. However, it needs a spacious one-dimensional array as if all DT levels would be completely filled, no matter if all nodes actually exist. Experimental validation is performed on real-life and artificial datasets with various sizes and dimensions. Results show that by using the compact representation not only the memory requirements are reduced but also the time of induction is decreased.

Keywords: Evolutionary data mining · Decision trees · Compact in-memory representation · Graphics processing unit (GPU) · CUDA

1 Introduction

Explainable Machine Learning (XML) [2] is a new subfield of Machine Learning (ML) that aims to explain how ML models make predictions. Until recently, most research has focused on the predictive power of algorithms rather than on understanding rationale behind these predictions. The revival in this field reflects, as it were, an interest in and demand for understandable and interpretable methods for real-world applications. A learning model, to qualify as an XML algorithm, should be understandable using concepts related to human intelligence.

Decision trees (DTs) form a class of models that generally fall into the XML category. They are usually induced by top-down greedy methods. Such an induction is usually fast; however, it can lead to sub-optimal solutions [1]. One of the

R. Wyrzykowski et al. (Eds.): PPAM 2022, LNCS 13826, pp. 126–138, 2023.
https://doi.org/10.1007/978-3-031-30442-2_10

alternative approaches is the use of evolutionary algorithms (EAs). The incorporation of EAs into the DT induction allows for global solution-space exploration, leading to better solutions, that is, generated trees are much simpler and at least as accurate as those induced with traditional methods. Moreover, evolutionary induced DTs are less prone to overfitting, instability to changes in training data and attribute-selection bias [14]. At the same time, EA approach in the DT induction brings new challenges. Population-based and iterative calculations may be time-demanding, or even unachievable for big data [1,7,10].

To speed up the evolutionary induction of DTs, different parallel computing approaches were studied [10]. In this paper, we focus on the GPU-supported one that appeared to be the most efficient [6,7]. To boost the induction calculations we investigated different GPU memory layouts and representations, and we would like to propose a compact in-memory representation of DTs. It uses a one-dimensional array where links (corresponding to tree branches) between parent and children nodes are explicitly stored. In comparison to (previous) complete representation, it only holds the nodes that actually exist. There is no need to store all nodes as if all DT levels would be completely filled, no matter if a node really exists. We experimentally show that the compact representation not only saves memory resources but also speeds up the induction further.

The next section gives a brief overview of DTs, the ways of their induction as well as describes the Global Decision Tree (GDT) system that serves as the framework for our solution. Section 3 describes the GPU-boosted solution using the compact in-memory representation. Section 4 provides the evaluation, while Sect. 5 includes the conclusion and possible future works.

2 Background

2.1 Decision Trees

Despite more than 50 years of research [11], DTs are still being developed to address the various challenges they continue to face. They can be used as stand-alone single-tree solutions or as part of larger models such as random forests and gradient boosted DTs. In the latter case, however, it is not possible to speak of XML models, because in the pursuit of greater accuracy, the ease of interpreting and understanding ensemble models has been lost.

A typical DT consists of nodes and branches (see Fig. 1), where: each internal node is associated with a test on one or more attributes; each branch represents a test result, and each leaf (terminal node) contains a prediction [9]. Most tree-inducing algorithms partition the feature space using axis-parallel hyperplanes. Trees of this type are often called univariate because the test at each non-terminal node usually involves a single attribute that is selected according to the given goodness of split. Multivariate tests, which are generally based on linear combinations of many dependent qualities, are also used in some algorithms. The oblique split causes a non-orthogonal hyperplane to partition the feature space in a linear manner. DTs that enable multiple features to be tested

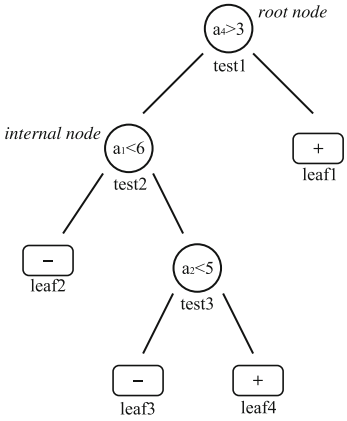


Fig. 1. An example of univariate decision tree.

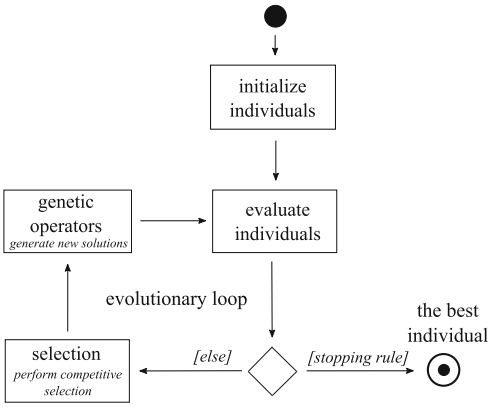


Fig. 2. Flowchart of the typical evolutionary algorithm.

at a node may be smaller than those confined to single univariate splits, but they have a substantially higher computing cost and are often difficult to interpret.

To make a prediction forecast, the new instance is followed down from a root node to a leaf, with the attribute values of each internal node being used to determine which branch to choose. The terminal node reflects the problem to which the DT is applied. In the case of classification trees, we are concerned with assigning a decision (class label) to each leaf. Typically, this is the class of the majority of all training instances that go into a given leaf. For the regression problem, DT models are used to approximate real-valued functions, so each leaf contains either a constant value or some linear (or nonlinear) regression model.

2.2 Decision Tree Induction

The complexity of inducing an optimal DT is NP-complete [5]. Therefore, heuristic improvements to practical DT learning algorithms are needed [9,11]. One of the major changes proposed in recent years for DTs concerns the induction process which has traditionally relied on a greedy partitioning strategy. Originally, the algorithm starts with a root node where a locally optimal split (test) is searched for based on a given criterion. The training instances are then redirected to the newly constructed nodes, and the procedure is repeated until a stopping condition is satisfied for each node. Furthermore, post-pruning is often used after induction to avoid the problem of over-fitting the training data and to improve the generalization ability of the predictive model. CART and C4.5/5.0 are the two most commonly applied top-down DT inducers.

To limit the impact of local, sub-optimal splits, alternative approaches based on metaheuristics, such as evolutionary algorithms (EAs), have been introduced to the tree induction process [1]. EAs belong to a family of meta-heuristic methods and represent techniques for solving a wide range of difficult optimization

problems [12]. The general framework (see Fig. 2) is based on biological evolution mechanisms. The typical EA works on the individuals, gathered in a population, that represent potential solutions to the target problem. In each evolutionary iteration, individuals are:

- transformed with genetic operators such as mutation and crossover that produce new offspring;
- evaluated according to a measure named the fitness function which determines its score;
- selected for reproduction - individuals with better fitness individuals being reproduced more frequently.

When the convergence criteria are met, the evolutionary loop is terminated.

The strength of the evolutionary approach lies in the global search in which tree structure and tests in internal nodes are searched simultaneously. It has been shown that evolutionary induced decision trees offer better suited, more stable, and simpler prediction models [1,10]. Of course, such a global induction is clearly more computationally demanding, but it can reveal underlying patterns that greedy approaches generally miss.

2.3 Global Decision Tree System

The proposed solution has been integrated into a system called the Global Decision Tree (GDT) [10]. The family of algorithms based on the GDT framework is very diverse and addresses almost every aspect related to evolutionary induced DTs like problem domain (classification, regression), tree representation (univariate, oblique, mixed), search (cost-sensitive, Pareto, memetic), real-world application (finance, medicine), parallelization and more [3,8].

GDT's overall structure is based on a typical EA schema [12] with an unstructured population and generational selection. The individuals are represented in their actual form as potential solutions using a tree-encoding schema. Initialization is performed in a simple greedy top-down manner with randomly selected samples of the training data. This way the population is fed with average solutions that should keep an initial balance between exploration and exploitation.

The selection mechanism is based on a ranking linear selection [12] with the elitist strategy, which copies the best individual found so far to the next population. Evolution terminates when a maximum number of generations is reached (default: 10 000) or the fitness of the best individual in the population does not improve during a fixed number of generations (default: 1 000).

To preserve genetic diversity, the GDT system applies two specialized meta-operators corresponding to the classical mutation and crossover. Both operators may have a two-level influence on the individuals as either the decision tree structure or a test in the splitting node can be modified. The type of node (internal, leaf), position in the tree (upper or lower parts), and node prediction error is taken into account to determine the crossover/mutation point. This way low quality nodes (or leaves) in the bottom parts of the tree are modified more

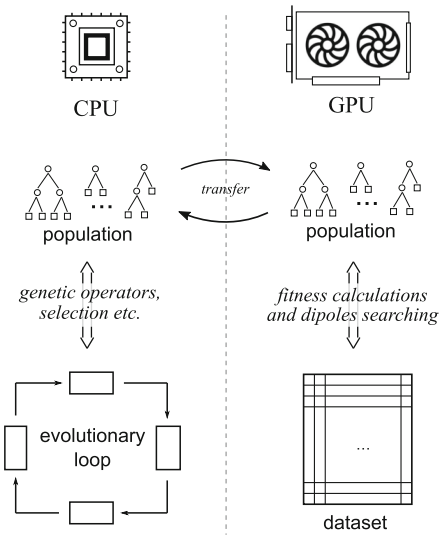


Fig. 3. General idea of the GPU-accelerated evolutionary induction. On the GPU, side the training dataset calculations are performed, while the CPU controls the evolution.

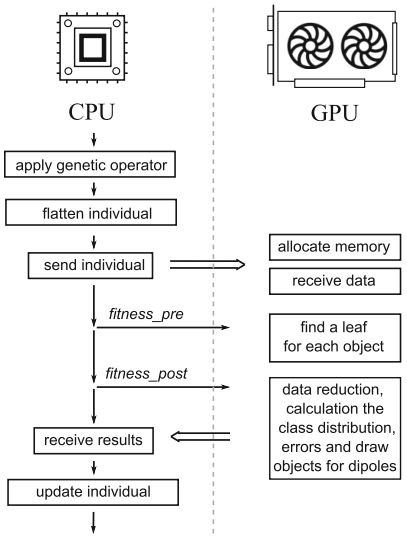


Fig. 4. Flow chart of updating an individual when a genetic operator is applied, including CPU-GPU communication, memory allocation and kernels' execution.

often. GDT offers dozens of specialized variants of crossover/mutations [10], often specific to the DT representation and problem domain, but the generic ones cover: (i) pruning nodes and expanding leaves; (ii) replacing, modifying or exchanging subtrees, branches, nodes, tests. New tests are created according to the dipolar strategy. A dipole is a pair of objects used to find the effective test.

The fitness function controls the accuracy and complexity of each individual. GDT offers various multi-objective optimization strategies [10]. Among them a weighted formula is the most universal one as it maximalizes the following fitness function: $Fitness(T) = Q(T) - \alpha * Complexity(T)$, where: Q(T) is the accuracy calculated on the training set, Complexity(T) is the tree complexity calculated as the sum of leaves and α is the relative importance of the complexity term (default: 0.001) and it is a user-supplied parameter.

3 GPU-Supported Evolution Using Compact In-Memory Representation of Decision Trees

The general idea of the GPU-supported solution (called cuGDT) is illustrated in Fig. 3. The most time-consuming operations (like fitness calculations or searching objects for dipoles, which are directly related to the training dataset) are isolated and delegated to the device [6]. The evolutionary induction is controlled by a CPU. Such a construction of cuGDT ensures that the parallelization does not affect the behavior of the original EA.

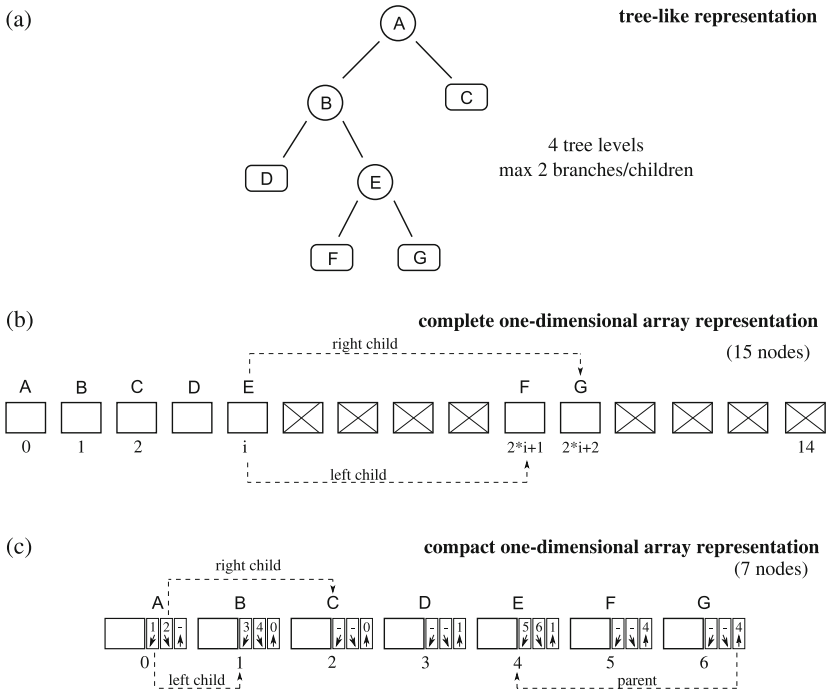


Fig. 5. Complete vs. compact one-dimensional array in-memory representation of a decision tree. Dotted lines indicate sample links between parent and children nodes. In the compact representation, each tree node contains three additional elements: array indexes of the left child, right child and parent nodes. In the complete representation, a parent node and its descendants can be found using the mathematical formula.

The dataset is transferred to the device before the evolution starts and it is kept till the evolutionary induction stops. This transfer time is negligible in relation to the evolution time. It was a conscious design decision to reduce the bottleneck of host/device memory transfers. However, this forced us to perform most of the dataset-related operations on the device, not only related to fitness calculations but also to searching for optimal splits. The CPU does not have direct access to the training dataset, it only receives sample objects to construct dipoles. During the evolution, the transfer between host and device includes sending the individuals to the GPU and sending back the results (class distribution, errors and objects for dipoles) to the CPU.

3.1 In-Memory Representation of Decision Trees

In the evolutionary loop, each time the genetic operator is successfully applied, the GPU is asked to help the CPU (see Fig. 4). Before transferring the modified individual, its flat representation is created based on its tree-like (using pointers) host representation (see Fig. 5). A one-dimensional array is built and then sent

to the device. A complete in-memory representation was previously used [6–8]. It did not require explicitly storing the links (as array indexes) between a parent node and its children. A simple mathematical formula was used to indicate the array indexes of children nodes or a parent node. The array index of the left child of the i-th node equals $(2 * i + 1)$, while for the right child, it is $(2 * i + 2)$. Unfortunately, the complete representation imposed to reserve memory space as if all DT levels would be completely filled, no matter if all the nodes really exist.

On the other side, the compact in-memory representation (see Fig. 5(c)) assumes that only the nodes that actually exist are put into the one-dimensional array. Thus, the array indexes of the parent node and descendants for each node have to be explicitly saved (next to the node data, like tests or classes). Obviously, this increases the memory requirements per node, but globally it may be compensated by keeping only actually existing nodes. The number of nodes in the complete representation grows fast, exponentially with the tree level. For a binary tree, in each successive tree level, it equals: $2\hat{}(tree\_level - 1)$, while the total number of nodes is: $2\hat{}(number\_of\_tree\_levels) - 1$. If we considered more than two children/branches then the growth would be even more prominent.

For DTs, the Structure-of-Arrays (SoA) data layout is used. In SoA [16], multi-value data are stored in separated arrays and the arrays are grouped in a structure. In our case, it is `struct DT{float thresholds[];int attributes[]; int leftChildNodesIdx[]; int rightChildNodesIdx[];int parentNodesIdx[];}`. The SoA layout is usually preferred from a GPU performance perspective because one thread may copy data to cache for other threads (coalesced memory access).

3.2 GPU Kernels Implementation

GPU computations are organized into two kernel functions: $fitness_{pre}$ and $fitness_{post}$ (see Fig. 4). The first kernel calculates the number of objects of each class located in each tree leaf. In addition, two randomly selected objects of each class are provided in each tree leaf. They may be later used to construct dipoles and finally effective tests. However, the results are scattered over separated copies of the individual created for each GPU block.

The $fitness_{post}$ function reduces the partial results collected by each GPU block. When the information about the class distribution is reduced, prediction errors in all leaves are found. Then, the class distribution, estimated errors and selected objects for dipoles are propagated from the leaves towards the tree root. Finally, all the results, in all tree nodes, are sent to the host.

The use of compact representation forced us to modify the way of traversing through DTs, among others. Considering the kernel $fitness_{pre}$, when objects are propagated from the tree root towards the leaves, children nodes are found based on stored indexes in the arrays `int leftChildNodesIdx[]` and `int rightChildNodesIdx[]`. For the $fitness_{post}$ kernel, when the results are propagated from the leaves towards the tree root, parent nodes are found based on

Table 1. Characteristics of the real-life and artificial datasets.

Dataset	No. samples	No. attributes	No. classes
Chess10K	10 000	2	2
Chess100K	100 000	2	2
Chess1M	1 000 000	2	2
Chess10M	10 000 000	2	2
*SDD_2C**	10 639	49	2
*SDD_4C**	21 277	49	4
*SDD_6C**	31 915	49	6
*SDD_8C**	42 553	49	8
*SDD_10C**	53 191	49	10
SDD	58 509	49	11

* Note: A subset of the *SDD* dataset containing objects of first 2, 4, 6, 8 and 10 classes.

the indexes in the array `int parentNodesIdx[]`. The reduction is similar but is performed on less (compact) array elements.

4 Experimental Validation

Validation was performed on both real-life and artificial datasets. The details of each one are presented in Table 1. The artificial dataset, called *Chess*, represents a classification problem with two classes, two real-values attributes and objects arranged on a 3×3 chessboard [10]. We used the synthetic dataset to scale it freely (from 10 000 to 10 000 000 objects). Concerning the real-life dataset, *Sensorless Drive Diagnosis* (*SDD*) from UCI Machine Learning Repository [4] was used. It contains 48 features extracted from the motor current signal and 11 different class labels. To check the solution behavior when the number of classes increases, we extracted from the *SDD* dataset five subsets, containing successively objects of the first 2, 4, 6, 8 and 10 classes. We called them *SDD_2C*, *SDD_4C*, *SDD_6C*, *SDD_8C* and *SDD_10C*.

Experiments were performed using two NVIDIA GPU cards installed on:

- server with two 8-core processors Intel Xeon E5-2620 v4 (20 MB Cache, 2.10 GHz), 256 GB RAM, NVIDIA Tesla P100 GPU card (3 584 CUDA cores and 12 GB of memory);
- server with two 24-Core processors AMD EPYC 7402 (128 MB Cache, 2.80 GHz), 1 TB RAM, NVIDIA Tesla A100 GPU card (13 824 CUDA cores and 40 GB of memory).

Servers run 64-bit Ubuntu Linux 18.04.6 LTS. The original GDT system was implemented in C++ and compiled with the use of gcc version 7.5.0. The GPU-based parallelization was implemented in CUDA-C [15] and compiled by nvcc

Table 2. Mean execution times of sequential, OpenMP and GPU-supported implementations (in seconds). Concerning GPU-supported ones, time for complete and compact in-memory representations of DTs is shown.

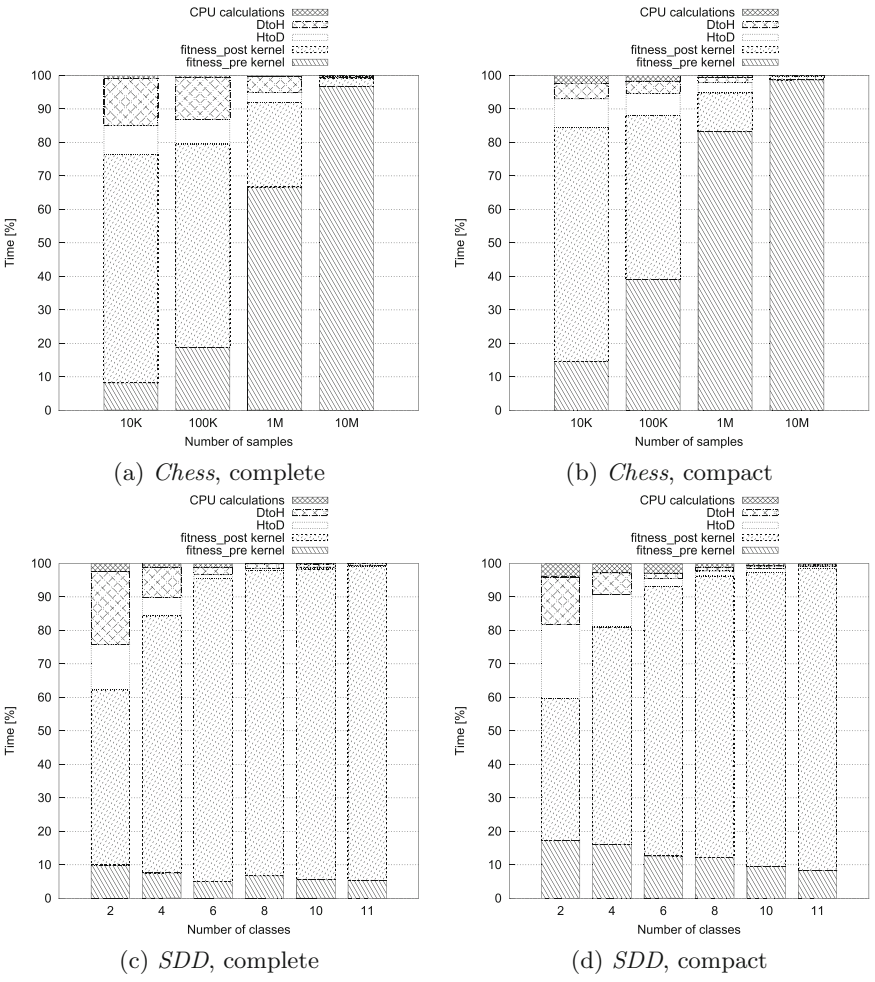
Dataset	Sequential	OpenMP	GPU			
			P100		A100	
			Complete	Compact	Complete	Compact
Chess10K	61	14.5	19.2	8.1	14.3	7.3
Chess100K	692	145.3	21.5	10.7	15.2	8.2
Chess1M	23 536	3 605.7	55.9	51.2	25.3	23.8
Chess10M	324 000	47 600.4	641.1	621.7	143.9	142.5
SDD_2C	38	10.7	10.1	8.53	8.9	8.1
SDD_4C	87	27.2	24.7	10.35	12.5	9.5
SDD_6C	411	128.8	119.9	19.74	51.7	12.1
SDD_8C	945	286.4	195.2	33.91	75.8	14.9
SDD_10C	1 984	548.4	420.4	57.64	301.1	27.2
SDD	2 858	766.8	561.2	75.76	447.5	36.1

CUDA 11.6 [13] (single-precision arithmetic was applied). All presented results correspond to averages of 5-10 runs and were obtained with a default set of parameters from the original GDT system [6,10]. As we are focused in this paper only on time and memory resources, results for the classification accuracy are not included (they can be found in [6,10]).

4.1 Results

Table 2 presents the preliminary results for all tested datasets. The mean execution times of cuGDT (using two different GPUs) as well as OpenMP-accelerated and sequential GDT versions (using the Intel CPU server) are shown. It is clearly visible that the use of compact representation gives an additional reduction in the evolution time. Concerning *Chess* dataset, the acceleration is more relevant for smaller datasets. When the number of objects grows, the difference between compact and complete representations becomes less important. The reason can be found when the run-time breakdown is analyzed (see Fig. 6(a,b)). We see that for 10M of objects, the $fitness_{pre}$ kernel dominates (more than 97% of time in both cases). The applied DT in-memory representation has no significant impact on the workload in this kernel. On the other side, for 10K of objects, the $fitness_{post}$ kernel is the most important, and here the representation makes a real difference. The reduction and propagation of results (towards the root node) are done through all elements of the one-dimensional array (representing DT) without any control decision. Thus, in the complete representation, many more operations have to be done.

As regards the *SDD* dataset, the compact representation also reduces the evolution time. However, the improvement grows when the number of classes



(a) *Chess*, complete (b) *Chess*, compact

(c) *SDD*, complete (d) *SDD*, compact

Fig. 6. The run-time breakdown of the GPU-accelerated algorithm using complete and compact in-memory representations of decision trees for NVIDIA Tesla P100 GPU card. The average time (as a percentage of total run-time) of the most relevant parts is shown, both communication between host and device (HtoD, DtoH) and GPU/CPU computations are included.

(and objects) increases. Similarly, the run-time breakdown in Fig. 6(c,d) can be used to explain the solution behavior. The kernel $\mathit{fitness}_{post}$ dominates and the compact representation is more efficient in it (as the reduction is performed on a smaller (compact) array). Another reason can be deduced from the results in Table 3 where the transfer size between the host and device is presented. We see that when the number of classes increases also the size of transfer grows. At the same time, the difference in the size of sent data becomes more important, particularly, for the transfer from the device to the host (transfer of the results).

Table 3. Transfer size in MB for Tesla P100 GPU card. Decision trees sent from host to device (HtoD) as well as results sent from device to host (DtoH) are included.

Dataset	Complete		Compact	
	HtoD	DtoH	HtoD	DtoH
Chess10K	18.05	72.19	15.36	24.49
Chess100K	17.71	70.85	14.83	23.62
Chess1M	17.17	68.68	15.64	24.99
Chess10M	20.07	80.29	14.39	22.99
SDD_2C	7.01	28.05	8.02	12.83
SDD_4C	16.57	115.61	11.81	33.05
SDD_6C	59.99	597.21	19.24	76.95
SDD_8C	71.71	929.37	27.46	142.77
SDD_10C	84.91	1 358.57	34.36	219.88
SDD	134.02	2 338.61	41.16	288.12

If there are more classes, there is a need to send back to the host results containing more data (in each DT node, for each class, the number of located training objects as well as two objects for constructing dipoles). Considering the transfer from the host to the device, the increase can be explained by bigger DTs when the problem is more difficult. For the *Chess* dataset, the transfer size is similar through the various number of objects as it does not influence the problem and DTs of similar size are transferred. Obviously, there are differences in transfer size between compact and complete representations but there are smaller than in the case of the *SDD* dataset as objects in *Chess* are only labeled by two classes.

NVIDIA Tesla A100 GPU card provides better results than P100 GPU one as it is more powerful, both from computational and memory (bandwidth) perspectives. If compared to the sequential GDT or even to its OpenMP-supported version [10], the GPU-boosted GDT is at least one order faster in most cases. cuGDT with the compact representation is always at least a little faster than using the complete one. Moreover, in some cases (8 and more classes), we had to limit the maximum size of DTs able to be processed by a GPU when the complete representation was used. It was the case when very deep DTs with sparsely filled nodes were verified. The memory size needed to store the structure of DTs was quite small, but the results (sent to the host) required too much GPU memory.

5 Conclusion

In this paper, we introduce a compact in-memory representation of DTs into the GPU-supported evolutionary induction. This representation stores explicitly links between parent and children nodes. It required allocating additional memory space for each tree node to save these links. However, in comparison to

the (previously used) complete representation, the compact one allowed us to globally decrease both the memory and time resources. It is only a preliminary investigation, and we are conscious that more research is needed, e.g. processing other datasets with different characteristics, checking kernel call settings and deeper profiling. Moreover, our plans include more research on other memory layouts and representations both for training data and DTs.

Acknowledgements. This work was supported by Bialystok University of Technology, Poland under the Grant WZ/WI-IIT/4/2023 founded by Ministry of Science and Higher Education.

References

1. Barros, R.C., Basgalupp, M.P., De Carvalho, A.C., Freitas, A.A.: A survey of evolutionary algorithms for decision-tree induction. IEEE Trans. SMC, Part C **42**(3), 291–312 (2012)
2. Belle, V., Papantonis, I.: Principles and practice of explainable machine learning. Front. Big Data. **4**, 39 (2021)
3. Czajkowski, M., Kretowski, M.: Decision tree underfitting in mining of gene expression data. An evolutionary multi-test tree approach. Expert Syst. Appl. **137**, 392–404 (2019)
4. Dua, D., Karra Taniskidou, E.: UCI machine learning repository (2022). https://archive.ics.uci.edu/ml
5. Hyafil, L., Rivest, R.L.: Constructing optimal binary decision trees is NP-complete. Inf. Process. Lett. **5**(1), 15–17 (1976)
6. Jurczuk, K., Czajkowski, M., Kretowski, M.: Evolutionary induction of a decision tree for large-scale data: a GPU-based approach. Soft. Comput. **21**(24), 7363–7379 (2017)
7. Jurczuk, K., Czajkowski, M., Kretowski, M.: Multi-GPU approach to global induction of classification trees for large-scale data mining. Appl. Intell. **51**(8), 5683–5700 (2021). https://doi.org/10.1007/s10489-020-01952-5
8. Jurczuk, K., Czajkowski, M., Kretowski, M.: GPU-based acceleration of evolutionary induction of model trees. Appl. Soft Comput. **119**, 108503 (2022)
9. Kotsiantis, S.B.: Decision trees: a recent overview. Artif. Intell. Rev. **39**(4), 261–283 (2013)
10. Kretowski, M.: Evolutionary Decision Trees in Large-Scale Data Mining. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21851-5
11. Loh, W.Y.: Fifty years of classification and regression trees. Int. Stat. Rev. **82**(3), 329–348 (2014)
12. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs, 3rd edn. Springer-Verlag, Berlin, Heidelberg (1996). https://doi.org/10.1007/978-3-662-03315-9
13. NVIDIA: NVIDIA Developer Zone - CUDA Toolkit Documentation (2022). https://docs.nvidia.com/cuda/
14. Rivera-Lopez, R., Canul-Reich, J., Mezura-Montes, E., Cruz-Chávez, M.A.: Induction of decision trees as classification models through metaheuristics. Swarm Evol. Comput. **69**, 101006 (2022)

15. Storti, D., Yurtoglu, M.: CUDA for Engineers: An Introduction to High-Performance Parallel Computing. Addison-Wesley, New York (2016)
16. Strzodka, R.: Abstraction for AoS and SoA layout in C++. In: Hwu, W.W. (ed.) GPU Computing Gems Jade Edition, pp. 429–441. Morgan Kaufmann (2012)