Hardware Acceleration of Sparse Oblique Decision Trees for Edge Computing

Predrag Teodorovic*, Rastislav Struharik
Department of Power, Electronic and Telecommunication Engineering,
Faculty of Technical Sciences, University of Novi Sad,
Trg Dositeja Obradovica 6, 21000, Novi Sad, Serbia
t_pedja@uns.ac.rs

***Abstract*—This paper presents a hardware accelerator for sparse decision trees intended for FPGA applications. To the best of authors' knowledge, this is the first accelerator of this type. Beside the hardware accelerator itself, a novel algorithm for induction of sparse decision trees is also presented. Sparse decision trees can be attractive because they require less memory resources and can be more efficiently processed using specialized hardware compared to traditional oblique decision trees. This can be of significant interest, particularly, in the edge-based applications, where memory and compute resources as well as power consumption are severely constrained. The performance of the proposed sparse decision tree induction algorithm as well as developed hardware accelerator are studied using standard benchmark datasets obtained from the UCI Machine Learning Repository database. The results of the experimental study indicate that the proposed algorithm and hardware accelerator are very favourably compared with some of the existing solutions.**

***Index Terms*—Decision trees; Hardware accelerator architectures; Genetic algorithms; Edge computing.**

I. INTRODUCTION

Until recent discoveries of Convolutional Neural Networks and other Deep Learning architectures, Decision trees (DTs) had been recognized as one of the three most popular predicting models in machine learning field, together with Artificial Neural Networks and Support Vector Machines.

The decision tree predicting model was first presented in the literature more than 30 years ago [1], while axis-parallel DTs were introduced only few years after [2]. Assuming that the classification problem is represented by a set of n attributes, A_i (i = 1, …, n), axis-parallel DTs in each tree node perform testing of a single attribute A_i from a test instance against the threshold a_i: $A_i > a_i$. Inducing axis-parallel (also called orthogonal), DT assumes the selection of the attribute to be assigned and tested in each DT node (A_i) as well as the threshold value required for a comparison (a_i). ID3 and C4.5, the two most commonly used algorithms for inducing axis-parallel DTs, are presented in [2]. Although proposed quite long ago, axis-parallel DTs are still a topic of interest for the academic community [3]–[6].

Oblique decision trees are the generalization of axis-parallel DTs allowing multiple attribute testing in every DT node. As a result, oblique DTs are usually much smaller in size providing higher classifying accuracy when compared to axis-parallel DTs. In oblique DTs, this multivariate testing has a form, which is expressed as follows

$$\sum_{i=1}^{n} a_i \times A_i + a_{n+1} > 0, \tag{1}$$

where a_i, i = 1, …, n + 1 are called hyperplane coefficients.

The most important oblique DT induction algorithms are CART, proposed in [1], and OC1, which is presented in [7]. After the authors in [8] proved that finding the best oblique DT is a NP-complete problem, many oblique DT induction algorithms use some kind of heuristics in order to find sub-optimal hyperplane coefficients [9]–[12]. The authors in [13] use HereBoy evolutionary algorithm [14] for solving this hard oblique DT induction problem. In our research, we use this approach as the starting point and modify it in order to support the induction of sparse oblique DTs.

Recently, a huge effort in machine learning community has been spent on the compression and size reduction of the prediction models, mainly Artificial Neural Networks (ANNs) and Convolutional Neural Networks (CNNs) [15]–[19]. The authors in [20], [21] exploit the benefits of minimizing the number of non-zero hyperplane coefficients in oblique DTs as well. However, the focus in these papers is on the feature/attribute selection and the elimination of irrelevant or noisy features/attributes, while the percentage of non-zero elements after DT induction is not reported.

In this paper, we present SDTI algorithm for the induction of sparse DTs and SDTA hardware accelerator for sparse DTs. The proposed accelerator, intended for an implementation in FPGA, is particularly interesting for embedded and edge applications, where the storage capacity, memory consumption, and bandwidth as well as computing capabilities are severely constrained. As a result of the DT sparsification and increased number of zero-valued hyperplane coefficients, SDTA requires less memory for storing DT parameters and provides significant speedup by skipping zero-valued product terms in (1) and avoiding

Manuscript received 26 November, 2018; accepted 30 June, 2019.

This work was partially supported by Serbian Ministry of Education and Science (Project title: "Innovative electronic components and systems based on inorganic and organic technologies embedded in consumer goods and products", No. TR32016).

unnecessary operations.

The problem of hardware implementation of DTs has been in the focus of the research community for more than 15 years, resulting in a number of proposed architectures [22]–[28]. However, all previously proposed architectures are concerned with the acceleration of dense univariate, oblique or non-linear DTs, or ensembles composed out of dense DTs. To the best of the authors' knowledge, the hardware architecture for the acceleration of sparse DTs, presented in this paper, is the first of its kind.

The rest of the paper is organized as follows. In Section II, we present SDTI algorithm that induces sparse DTs by forcing high percentage of zero hyperplane coefficients in each DT node (1) without decreasing DT accuracy. In Section III, we introduce SDTA hardware accelerator for the sparse oblique decision trees intended for FPGA applications. The proposed hardware accelerator is going to benefit from the sparsity in DTs and provide faster classifications by computing operations only with non-zero operands. In Section IV, we give the experimental results of benchmarking our SDTI algorithm and SDTA accelerator performance using datasets from the UCI machine learning repository. Conclusions are given in Section V.

II. SPARSE DECISION TREE INDUCTION (SDTI) ALGORITHM

In this section, we present the application of HereBoy evolutionary algorithm for solving the sparse DT building problem. In our SDTI algorithm, iteratively for each node in DT, one by one hyperplane coefficients from (1) are masked and set to zero, while HereBoy algorithm is used to fine tune remaining non-masked coefficients in order to change and improve the position of the hyperplane. This incremental sparsification is repeated until the desired percentage of the hyperplane coefficients is set to zero. DT itself is built using the classical DT building algorithm with an iterative sparsification explained above and performed at each DT node.

In order to evaluate position of the hyperplane, during incremental sparsification procedure performed in each DT node, the fitness function from [13] is used

$$fitness = 1 + \frac{1}{N}\left\lfloor \sum_{i=1}^{k} N_{1i} ld \frac{N_{1i}}{\sum_{j=1}^{k} N_{1j}} + \sum_{i=1}^{k} \left(N_i - N_{1i}\right) ld \frac{N_i - N_{1i}}{N - \sum_{j=1}^{k} N_{1j}} \right\rfloor, \quad (2)$$

where *ld* stands for a binary logarithm, $log_2(n)$. The symbols in proposed fitness function are:

– k (number of classes of the classification problem);

– N (number of training instances associated with the current node);

– N_i (number of training instances that belong to the class i (from the total of N instances);

– N_{1i} (number of instances that belong to the class i, placed above the current hyperplane).

In SDTI implementation, a binary encoding scheme, where HereBoy is working with a single chromosome similar to [13], is also accepted. Given n attributes from a dataset, the chromosome consists of $n + 1$ values that encode coefficients a_i from (1), where each coefficient is encoded with l bits (typical value for l being 20). Also, like in [13], the initial chromosome for the evolutionary algorithm is generated in a way that corresponding hyperplane divides training set and ensures that at least one instance from the training set can be found on both sides of the hyperplane.

The basis for our SDTI algorithm is HBDT algorithm presented in [13]. Similar to HBDT algorithm, in every step, we use HereBoy algorithm to find the sub-optimal hyperplane, which splits instances from different classes in two disjoint regions. This is recursively repeated until the current region contains only the instances from a single class, which results in creating the leaf and setting the output class label to that particular class value. Modification of HBDT algorithm, introduced in our SDTI algorithm, is related to the incremental sparsification of a hyperplane in each node of DT. For each hyperplane obtained by evolutionary algorithm, the worst hyperplane coefficient is found, with the property that the fitness value calculated using (2) minimally decreases after the replacement of that coefficient with a zero value. The worst coefficient, when found, is masked and set to zero. This procedure is repeated with remaining non-masked hyperplane coefficients until the desired percentage of sparsification is reached.

Algorithm 1. Sparse Decision Tree Induction (SDTI) algorithm.

```
SDTI (TI, sparsity)
TI - Set of instances used to build a DT. Each
       instance is a vector of n numerical
       attributes plus the output class value
sparsity – sparsification level for each DT node
       (percentage of hyperplane coefficients
       that will be masked and set to zero)
→ Create node root
→  If the values of output_class for all
  instances from the TI set belong to the same
  class, make root node a leaf, with output
  label matching the appropriate class value
→  Otherwise
   {
    → Find out the dominating class in the TI
      set, and set the class label for the node
      root to that value
    →  Repeat
      {
        →  Using the HereBoy algorithm and
          fitness function (2) find the optimal
          position of the dividing hyperplane
        → Find among non - masked hyperplane
          coefficients coefficient hc_worst
          that, when replaced with zero,
          minimally reduces fitness of the
          hyperplane, calculated by fitness
          function (2)
        → Mask and set to zero hc_worst
        → Calculate cs (current sparsity) as a
          percentage of zero elements within
          hyperplane coefficients
       } until cs is greater or equal to
    sparsity

    → Using the best hyperplane, divide the TI
     set into two subsets, one containing
     instances located above the hyperplane,
     TIabove and the other, containing instances
     located below the hyperplane, TIbelow.
    → Create a new branch descending from the
     node root, and create a sub-tree by calling
     the algorithm SDTI(TIabove, sparsity)
    → Create a new branch descending from the
     node root, and create a sub-tree by calling
     the algorithm SDTI(TIbelow, sparsity)
   }
→ Return root
```

Since the iterative sparsification, explained above, is repeated for each DT node, the resulting DT should have exactly the same percentage of zero values per each DT node defined by the *sparsity* argument. As it is shown in section IV, for all 10 benchmarked datasets from the UCI repository, we are able to reach the sparsification percentage above 60 % (in some cases, even close to 80 %) without the loss in accuracy.

Similar to HBDT algorithm, in order to reach even better classification results, pruning of a DT at the end is performed. As a result, the size of the DT is decreased as a consequence of the redundant nodes removal. For this purpose, *Prune_DT* algorithm, reported in [13], is used.

III. HARDWARE ACCELERATOR FOR SPARSE DTS

The architecture for hardware acceleration of sparse decision trees, called SDTA (Sparse Decision Tree Accelerator), is the evolution of SMpL architecture introduced in [24].

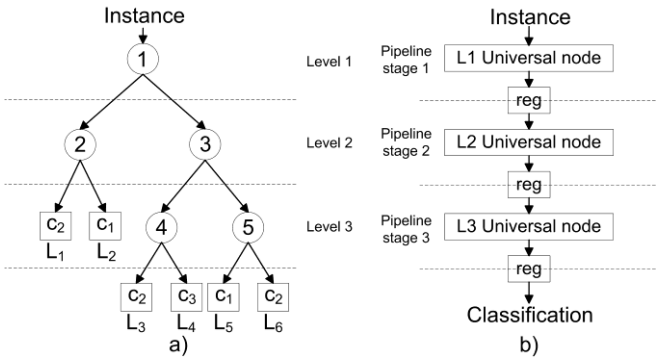

Fig. 1. Implementing DT using SMpL architecture from [24]: a) Structure of typical oblique DT; b) SMpL architecture for hardware implementation of DT.

Let us consider the DT shown in the Fig. 1(a). A straightforward approach to the hardware realization would be to implement the complete DT in hardware, which is originally presented in [22]. However, this approach is not the optimal because of the following key property of DTs. While the instance is being processed by a DT, only a subset of nodes from the complete DT is going to be visited. This is because each instance is taking exactly one path from the root node to one of the DT leaves. For example, if the current input instance will eventually be classified into the leaf node L_4 from Fig. 1(a), then it will be processed only by DT nodes 1, 3 and 4 while nodes 2 and 5 will not be active during the current instance processing. Therefore, to classify an instance, there is no reason to evaluate every node in the DT, but only a selected subset of nodes, one from each DT level, at most. SMpL architecture, originally presented in [24], that evaluates only visited DT nodes during the classification of an instance, is more efficient than the architecture in [22] in terms of separate node modules that need to be implemented in hardware. For example, Fig. 1(b) shows the implementation of the DT from Fig. 1(a) based on the SMpL architecture. The SMpL architecture consists of three separate modules, called Universal Nodes. They can evaluate any DT node located on the same depth within a DT. We need three of these modules since the depth of the DT from the Fig. 1(a) is three. Operation of the SMpL architecture is pipelined, so the instance processing throughput is not influenced by the depth of implemented DT and is only dependent on the time required to evaluate a single DT node, which, in case of SMpL architecture, is proportional to the number of attributes, *n*, in the underlying classification problem DT was created to solve.

As shown in Section II, for any classification problem, instead of learning dense oblique DT, in which we would have to process all *n* attributes in every DT node, we can learn a sparse oblique DT, which processes only a fraction *p* of *n* problem attributes in each node. Using the sparse DT can be beneficial, particularly when instance processing speed is concerned since computation of each sparse DT node will be *1/p* times faster compared to a dense DT node computation time. However, all previously proposed architectures for the DT acceleration are not able to benefit from this optimization opportunity.

The SDTA architecture that we are proposing is explicitly designed to take an advantage of such an optimization. SDTA architecture is closely related to SMpL architecture presented in [24], but modified in order to be able to skip all unnecessary computations due to the fact that some hyperplane coefficients a_i becomes equal to zero as a result of the sparsification introduced in each DT node.

The top-level architecture of SDTA sparse DT accelerator is presented in Fig. 2.

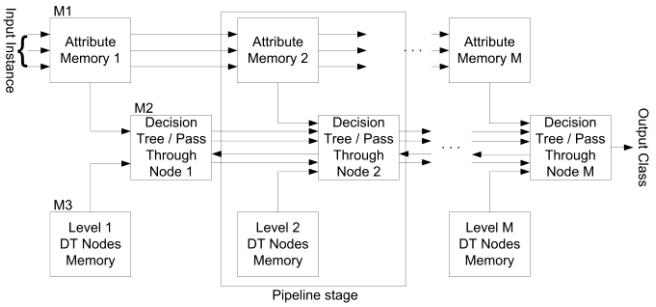

Fig. 2. Top-Level Architecture of SDTA Sparse DT accelerator.

The SDTA sparse DT accelerator consists of *M* identical pipeline stages, where *M* is the maximum depth of a sparse DT that it can accelerate. Each pipeline stage is capable of evaluating sparsified tests from all nodes found at the same depth in the DT and consists of three major modules: attribute memory (module M1), Decision Tree or Pass Through (DTPT) node (module M2), and memory for storing relevant information about the nodes found at the same depth in the DT (module M3). Attribute memories from the pipeline stages create a chain, shifting the attribute values related to different instances that are being processed by the SDTA accelerator through the system. Each attribute memory is also connected to the corresponding DTPT node. DTPT nodes create a pipeline chain as well. Figure 3 presents the detailed architecture of one pipeline stage. Attribute memory, M1, is used to store attribute values for the current instance. Module M2 normally calculates the position of the instance relative to the hyperplane associated to the selected node while skipping all product terms from (1), where coefficients a_i are equal to zero, and calculates the address of the node from the next level that should be visited next.

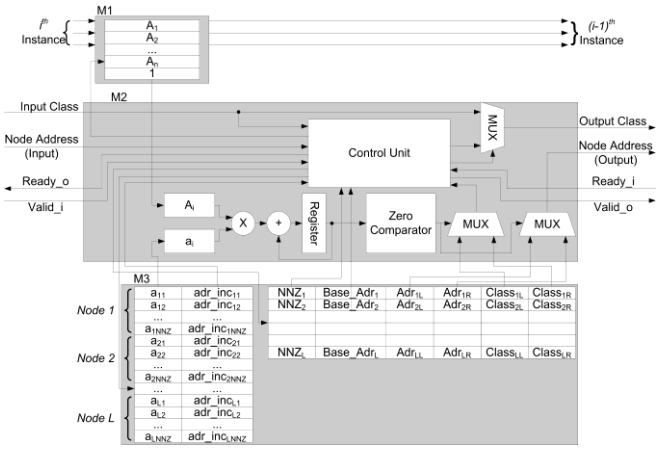

Fig. 3. Detailed architecture of the SDTA pipeline stage.

This happens when the current instance has not been classified yet, i.e., module M2 operates in the "Decision Tree" mode. In case the current instance has already been classified by some previous pipeline stage, module M2 simply transfers the input class value to the next pipeline stage operating in the "Pass Through" mode. Module M3 stores the necessary information about all nodes from the same level in DT and consists of two memory units.

The first memory unit, SHCM, stores sparse hyperplane coefficients for all nodes. For each non-zero hyperplane coefficient a_i, its value together with the attribute address increment is stored. The address increment is used to fetch the appropriate attribute from M1 memory, which should be multiplied with the current hyperplane coefficient. For example, if there is a total of 10 problem attributes and in the current DT node non-zero hyperplane coefficients are 1st, 3rd, 6th, 7th and 10th coefficient, their corresponding attribute address increment values would be 0, 2, 3, 1, and 3, respectively.

The second memory unit stores the following data for each node associated to the current pipeline stage: number of non-zero hyperplane coefficients *NNZ*; base address *Base_Addr*, where the block of non-zero coefficients for the current node begins in the SHCM memory; addresses of left and right child nodes of the current node, Adr_L and Adr_R; class values for all child nodes that are actually leaves, $Class_L$ and $Class_R$. In case a child node is not a leaf, its corresponding class value is set to zero.

During the evaluation of the DT node, module M2 calculates (1), but multiplying and adding only attribute/hyperplane coefficient product terms, for which the hyperplane coefficient a_i is different from zero. All product terms, where a hyperplane coefficient a_i is zero, are skipped effectively speeding up the hyperplane computation process.

Compared to SMpL architecture from [24], the node processing time of the SDTA architecture is measured in clock cycles, equals:

$$\begin{aligned} SMpL\_Node\_\mathrm{Pr}ocessing\_Time &= (n+1)\,T_{clk}, \\ SDTA\_Node\_Processing\_Time &= (NNZ+1)\,T_{clk}, \end{aligned} \tag{3}$$

where n is the number of problem attributes and *NNZ* is the number of non-zero hyperplane coefficients, a_i.

Clearly, since $NNZ < n$, the node processing time of the SDTA architecture is always strictly shorter than the node processing time of the SMpL architecture. The amount of the node processing time reduction depends on the number of hyperplane coefficients that is set to zero during the process of inducing the sparse DT, using SDTI algorithm. The more coefficients are set to zero, i.e., the more DT node tests are sparsified, greater DT processing speedup can be reached when using SDTA instead of SMpL architecture.

Since the instance processing throughput of both SMpL and SDTA architectures directly depends on the individual node processing time, from previous discussion, it is obvious that the instance processing throughput of SDTA architecture is higher than the one of SMpL architecture.

The instance processing throughput of SMpL architecture is given by a following equation

$$SMpL\_Throughput = \frac{1}{(n+1)\,T_{clk}}. \tag{4}$$

The instance processing throughput of SDTA architecture is more difficult to calculate since it depends on the amount of sparsity present in DT nodes along the path that the current instance is taking through a DT while it is being processed. However, the worst case for instance processing throughput for SDTA architecture can be calculated as follows

$$SDTA\_Throughput\_WC = \frac{1}{\left(\max\limits_{i\in DT\_Nodes}\{NNZ_i\}+1\right)T_{clk}}. \tag{5}$$

In the special case, when all hyperplanes from a DT are sparsified by removing identical number of coefficients (not necessarily located at the same positions), it is easy to calculate the instance processing throughput for SDTA architecture also

$$SDTA\_Throughput = \frac{1}{(NNZ+1)\,T_{clk}}. \tag{6}$$

In this case, we can easily calculate the expected speedup of SDTA over the existing SMpL architecture as follows

$$Speedup = \frac{n+1}{NNZ+1} \approx \frac{1}{\frac{NNZ}{n}} = \frac{1}{p},\ when\ n >> 1. \tag{7}$$

From equation (7), it can be concluded that the amount of the speedup that is reachable when using SDTA architecture directly depends on the amount of DT sparsification that is achievable during the DT induction phase.

IV. EXPERIMENTAL RESULTS

To be able to compare the performance of the SDTI algorithm with the HDBT algorithm, the following datasets from the UCI machine learning repository [29] are used: Glass Identification (gls), Vehicle Silhouettes (veh), Statlog Heart Disease (hrts), Hepatitis Domain (hep), Wine Recognition (wine), Pima Indians Diabetes (pid), Page

Blocks Classification (page), Waveform 40 (wav40), Heart Disease Cleveland (hrtc), and Wisconsin Diagnostic Breast Cancer (wdbc).

The instances with missing values are removed from the dataset, while all reported results are the averages of the five ten-fold cross-validation experiments. This assumes the dividing the original dataset D into 10 non-overlapping subsets, D_1, D_2, …, D_{10}, which consist of uniformly selected instances from D. During each cross-validation iteration, DT is built using the instances from $D \backslash D_i$ set and tested on D_i set (i = 1, ..., 10). By repeating this procedure 5 times, 50 DTs are constructed in total for each dataset. Then, average classification accuracy is calculated as the percentage of instances, which are correctly classified. Additionally, the average DT size is expressed as the average number of leaves. Both average classification accuracy and average DT size are calculated along with the corresponding 95 % confidence intervals. Similar to the approach from [13], 30 % of the instances from the training set are selected randomly to build the pruning set since DT pruning algorithm requires a separate pruning set.

Table I presents the results of experiments designed to compare the accuracy and the size of dense DTs created using HBDT algorithm [13] and sparse DTs with increasing sparsification value created using SDTI algorithm proposed in this paper. In Table I, for each UCI dataset, results of the HBDT algorithm correspond to the results of SDTI algorithm with the sparsity value of 0 %, since those two algorithms are identical when the sparsification is not applied.

TABLE I. BENCHMARKING RESULTS ON UCI DATASETS.

Dataset	Spars.	Attr	Tree size	Mem	Spd	Accuracy
hrts	0 %	13	7.22 ± 0.72	0 %	1.00	75.52±1.74
hrts	28.57 %	13	8.92 ± 0.78	-9 %	1.40	74.81 ± 2.05
hrts	50 %	13	11.44 ± 1.2	-16.1 %	2.00	75.93 ± 1.53
hrts	57.14 %	13	13.02 ± 1.11	-17.2 %	2.33	75.52 ± 1.72
hrts	71.43 %	13	18.4 ± 1.54	-17.9 %	3.50	75.41 ± 1.63
pid	0 %	8	30.16 ± 1.49	0 %	1.00	65.17 ± 1.18
pid	44.44 %	8	55.4 ± 2.82	3.6 %	1.80	65.71 ± 1.09
pid	55.56 %	8	72.22 ± 3.0	8.6 %	2.25	64.67 ± 1.09
pid	66.67 %	8	92.58 ± 3.49	4.7 %	3.00	65.18 ± 1.1
gls	0 %	9	12.72 ± 0.73	0 %	1.00	59.80 ± 2.01
gls	50 %	9	25.68 ± 1.59	5.3 %	2.00	60.41 ± 1.86
gls	60 %	9	25.56 ± 1.59	-16.2 %	2.50	60.21 ± 1.79
gls	70 %	9	26.18 ± 1.73	-35.5 %	3.33	60.10 ± 1.94
wav40	0 %	40	45.10 ± 1.19	0 %	1.00	80.27 ± 0.41
wav40	48.78 %	40	51.26 ± 1.89	-41.6 %	1.95	79.55 ± 0.33
wav40	58.54 %	40	64.10 ± 2.14	-40.7 %	2.41	79.30 ± 0.35
wav40	68.29 %	40	86.54 ± 3.06	-38.5 %	3.15	79.30 ± 0.39
wine	0 %	13	3.6 ± 0.25	0 %	1.00	89.90 ± 1.49
wine	35.71 %	13	5.42 ± 0.56	9.3 %	1.56	90.31 ± 1.58
wine	50 %	13	5.7 ± 0.66	-9.6 %	2.00	90.45 ± 1.48
wine	57.14 %	13	6.84 ± 0.72	-3.7 %	2.33	89.42 ± 1.68
wine	64.29 %	13	7.76 ± 0.92	-7.1 %	2.80	90.13 ± 1.65
hep	0 %	19	2.32 ± 0.35	0 %	1.00	81.12 ± 2.31
hep	50 %	19	2.78 ± 0.46	-32.6 %	2.00	80.38 ± 2.4
hep	60 %	19	3.18 ± 0.47	-33.9 %	2.50	81.25 ± 2.84
hep	70 %	19	3.70 ± 0.55	-38.6 %	3.33	80.25 ± 2.51
veh	0 %	18	31.06 ± 1.09	0 %	1.00	67.66 ± 1.12
veh	47.37 %	18	55.40 ± 2.56	-4.8 %	1.90	68.55 ± 1.03
veh	57.89 %	18	59.56 ± 3.22	-18 %	2.38	67.27 ± 1.12
veh	68.42 %	18	60.50 ± 2.78	-37.5 %	3.17	67.16 ± 0.99
page	0 %	10	24.86 ± 1.24	0 %	1.00	96.75 ± 0.15
page	36.36 %	10	37.86 ± 2.4	-1.7 %	1.57	96.83 ± 0.15
page	45.45 %	10	41.74 ± 2.63	-6.9 %	1.83	96.84 ± 0.15
page	54.55 %	10	53.20 ± 2.7	-0.6 %	2.20	96.71 ± 0.17
page	63.64 %	10	67.76 ± 3.26	1.7 %	2.75	96.52 ± 0.15
wdbc	0 %	30	6.04 ± 0.63	0 %	1.00	92.78 ± 0.76
wdbc	48.39 %	30	6.82 ± 0.69	-40.4 %	1.94	93.15 ± 0.76
wdbc	58.06 %	30	7.96 ± 0.83	-42.1 %	2.38	92.29 ± 0.66
wdbc	67.74 %	30	10.00 ± 0.98	-42.4 %	3.10	92.15 ± 0.7
hrtc	0 %	13	14.96 ± 1.24	0 %	1.00	51.50 ± 1.65
hrtc	50 %	13	17.42 ± 1.22	-41.2 %	2.00	50.84 ± 1.55
hrtc	57.14 %	13	19.10 ± 1.27	-44.4 %	2.33	52.06 ± 2.15
hrtc	64.29 %	13	20.60 ± 1.41	-49.9 %	2.80	52.10 ± 1.93
hrtc	78.57 %	13	26.80 ± 2.12	-60.4 %	4.67	52.27 ± 1.59

The SDTA architecture is designed and implemented using Xilinx Vivado Design Suite using default parameters for synthesis and implementation, while the experimental measurements are performed on Zynq Ultrascale+ MPSoC ZCU102 Evaluation Board [30].

The first column of the Table I represents UCI dataset. For each dataset, multiple sparsification percentages are reported in the second column. The given percentage refers to the percentage of the zero hyperplane coefficients in each DT node after running the SDTI algorithm. The fact that all nodes use the same percentage of sparsification is convenient for pipelining in SDTA hardware accelerator due to the fact that each node requires exactly the same number of computations, maximizing throughput. The third column shows the number of attributes for each dataset, which directly determines the maximum number of the hyperplane coefficients in each DT node: without the sparsification, a number of hyperplane coefficients in each DT node equal to the number of attributes plus one according to (1). The fourth column presents the average DT size, which is equal to the average number of leaves in the DT. Please note that the DT size is increased as a consequence of the sparsification in DT nodes. The DT size is larger as DTs become sparser.

The fifth column, *Mem*, shows the relative gain in terms of the memory required to store the complete DT calculated as the average number of internal nodes multiplied by the number of hyperplane coefficients. The negative percentages stand for the decreased memory requirements while positive represent the opposite. It is interesting to notice that, in some cases, even with high percentages of sparsification, there is no reduction of memory requirements for storing DTs, which is a consequence of larger DTs after the sparsification. However, in most cases of the test, memory requirements are decreased after the sparsification. The column *Spd* presents the speedup in the classification throughput expressed with respect to the DT with the sparsification 0 %. This is an effective comparison between the achievable instance classification throughput when DT is accelerated by SMpL dense DT accelerator presented in [24] and the SDTA sparse DT accelerator presented in this paper. The speedup is calculated using the percentage of the sparsification presented in the second column of the Table I according to (7). Higher the sparsification percentage results in a reduced number of the required computations in each pipeline stage, and higher is the speedup of the SDTA architecture compared to the SMpL architecture.

Finally, the last column of the Table I shows the averaged accuracies of the DTs built for different sparsification percentages and different UCI datasets. In the conducted tests, only sparsifications resulting in accuracy drop lower than 1 % are accepted compared to the accuracy of a non-sparse DT for the same dataset. The highlighted rows mark the highest sparsification percentages achieved with the acceptable accuracy drop for each given dataset.

As it can be observed from the Table I, it is possible to achieve the significant sparsification DT levels for each of used UCI datasets without degrading the original, dense DT accuracy. For all presented UCI datasets, the sparsification levels greater than 60 % can be achieved. For some of them, the sparsification levels reach the value of almost 80 %.

As for the DT storage memory requirements, it can be observed that for most of UCI datasets used a significant reduction in memory size can be achieved compared to the memory size required to store a dense DT. The amount of the memory size reduction reaches up to 60 %. For some datasets and sparsification levels, the output DTs are significantly deeper when compared to the dense DTs resulting in slightly increased requirements for the storing hyperplane coefficients, even after eliminating majority of them during the SDTI algorithm run.

However, if we analyse the throughput speedup from the Table I, it is clear that for all datasets used the DT sparsification is helping in improving the instance classification throughput. The instance processing speedup, when using sparse DTs over traditional dense DTs, ranges from 2.75 up to 4.67 times, which is a significant improvement.

V. CONCLUSIONS

In this paper, a novel algorithm for inducing sparse DTs, SDTI, and hardware accelerator architecture for accelerating sparse DTs, SDTA, are presented. Using sparse DTs over standard dense oblique DTs can be beneficial, since sparse DTs usually require less memory resources for storing their parameters and can be processed faster when compared to dense oblique DTs.

To validate the performance of SDTI, sparse DT building algorithm and SDTA, sparse DT hardware accelerator, a set of experiments using standard UCI machine learning repository datasets are used. Results of experiments seem to indicate that sparse DTs usually require significantly less memory resources, up to 60.4 % less storage capacity compared to dense oblique DTs, without any loss in DT accuracy.

Speedup experiments also indicate that working with the sparse DT hardware accelerator results in the instance processing speedup of up to 4.67 times compared to previously proposed dense DT hardware accelerator.

REFERENCES

[1] L. Breiman, J. H. Freidman, R. A. Olshen, and C. J. Stone, "Classification and Regression Trees", *Chapman and Hall/CRC*, 1984. DOI: 10.1201/9781315139470-8.

[2] J. R. Quinlan, "Induction of decision trees", *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986. DOI: 10.1007/bf00116251.

[3] A. M. Mahmood, K. M. Rao, and K. K. Reddi, "Novel algorithm for scaling up the accuracy of decision trees", *International Journal on Computer Science and Engineering*, vol. 2, no. 2, pp. 126–131, 2010.

[4] M. Z. Islam, "EXPLORE - A novel decision tree classification algorithm", in *Proc. Of BNCOD 2010 - 27th British National Conference on Databases*, 2012, pp. 55–71. DOI: 10.1007/978-3-642-25704-9_7.

[5] O. T.Yildiz, "Univariate decision tree induction using maximum margin classification", *The Computer Journal*, vol. 55, no. 3, pp. 293–298, 2012. DOI: 10.1093/comjnl/bxr020.

[6] A. López-Chau, J. Cervantes, L. López-García, and F. G. Lamont, "Fisher's decision tree", *Expert Systems with Applications*, vol. 40, no. 16, pp. 6283–6291, 2013. DOI: 10.1016/j.eswa.2013.05.044.

[7] S. K. Murthy, S. Kasif, and S. Salzberg, "A System for induction of oblique decision trees", *Journal of Artificial Intelligence Research*, vol. 2, pp. 1–32, 1994. DOI: 10.1613/jair.63.

[8] D. Heath, S. Kasif, and S. Salzberg, "Induction of oblique decision

trees", in *Proc. of the 13th International Joint Conference on Artificial Intelligence*, 1993, pp. 1002–1007.

[9] E. Cantú-Paz and C. Kamath, "Inducing oblique decision trees with evolutionary algorithms", *IEEE Trans. on Evolutionary Computation*, vol. 7, no. 1, pp. 54–68, 2003. DOI: 10.1109/tevc.2002.806857.

[10] R. C. Barros, M. P. Basgalupp, A. C. P. L. F. de Carvalho, and A. A. Freitas, "A survey of evolutionary algorithms for decision tree induction", *IEEE Trans. on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 42, no. 3, pp. 291–312, 2012. DOI: 10.1109/tsmcc.2011.2157494.

[11] F. E. B. Otero, A. A. Freitas, and C. G. Johnson, "Inducing decision trees with an ant colony optimization algorithm", *Applied Soft Computing*, vol. 12, no. 11, pp. 3615–3626, 2012. DOI: 10.1016/j.asoc.2012.05.028.

[12] S. H. Cha and C. Tappert, "A Genetic algorithm for constructing compact binary decision trees", *Journal of Pattern Recognition Research*, vol. 4, no. 1, pp. 1–13, 2009. DOI: 10.13176/11.44.

[13] R. Struharik, V. Vranjković, S. Dautović, and L. Novak, ''Inducing oblique decision trees", in *Proc. of 12th Int. Symp. Intell. Syst. Inform. (SISY)*, Subotica, Serbia, 2014, pp. 257–262. DOI: 10.1109/sisy.2014.6923596.

[14] D. Levi, "HereBoy: A fast evolutionary algorithm", in *Proc. of The Second NASA/DoD Workshop on Evolvable Hardware*, 2000, pp. 17–25. DOI: 10.1109/eh.2000.869338.

[15] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and J. Chen, "Compressing neural networks with the hashing trick", in *Proc. of International Conference on Machine Learning*, 2015, pp. 2285–2294.

[16] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding", in *Proc. of International Conference on Learning Representations*, 2016.

[17] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural network", is *Proc. of the 28th International Conference on Neural Information Processing Systems*, 2015, vol. 1, pp. 1135–1143.

[18] F. N. Iandola *et al.*, "Squeezenet: alexnet-level accuracy with 50x fewer parameters and < 0.5 mb model size", in *Proc. of 5th International Conference on Learning Representations*, 2017.

[19] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network", *in Proc. of 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, Seoul, 2016, pp. 243–254. DOI: 10.1109/ISCA.2016.30.

[20] M. Kretowski, "An evolutionary algorithm for oblique decision tree induction", in *Proc. of International Conference on Artificial Intelligence and Soft Computing – ICAISC 2004*, pp. 432–437. DOI: 10.1007/978-3-540-24844-6_63.

[21] M. Krętowski and M. Grześ, "Evolutionary learning of linear trees with embedded feature selection", in *Proc. of International Conference on Artificial Intelligence and Soft Computing - ICAISC 2006*, pp. 400–409. DOI: 10.1007/11785231_43.

[22] A. Bermak and D. Martinez, "A compact 3D VLSI classifier using bagging threshold network ensembles", *IEEE Transactions on Neural Networks*, vol. 14, no. 5, pp. 1097–1109, 2003. DOI: 10.1109/tnn.2003.816362.

[23] S. Lopez-Estrada and R. Cumplido, "Decision tree based FPGA-architecture for texture sea state classification", in *Proc. of IEEE International Conference on Reconfigurable Computing and FPGA's*, 2006, pp. 1–7. DOI: 10.1109/reconf.2006.307770.

[24] R. Struharik and L. Novak, "Intellectual property core implementation of decision trees", *IET computers & digital techniques*, vol. 3, no. 3, pp. 259–269, 2009. DOI: 10.1049/iet-cdt.2008.0055.

[25] R. Struharik and L. Novak. "Hardware implementation of decision tree ensembles", *Journal of Circuits, Systems and Computers*, vol. 22, no. 05, 2013. DOI: 10.1142/s0218126613500321.

[26] J. Barba *et al.*, "FPGA acceleration of semantic tree reasoning algorithms", *Journal of Systems Architecture*, vol. 61, no. 3–4, pp. 185–196, 2015. DOI: 10.1016/j.sysarc.2015.01.001.

[27] F. Saqib, A. Dutta, J. Plusquellic, P. Ortiz, and M. S. Pattichis, "Pipelined decision tree classification accelerator implementation in FPGA (DT-CAIF)", *IEEE Trans. Computers*, vol. 64, no. 1, pp. 280–285, 2015. DOI: 10.1109/tc.2013.204.

[28] X. Lin, R. D. Blanton, and D. E. Thomas, "Random forest architectures on FPGA for multiple applications", *In Proc. of the on Great Lakes Symposium on VLSI 2017*, pp. 415–418. DOI: 10.1145/3060403.3060416.

[29] C. L. Blake and C. J. Merz, "UCI Repository of Machine Learning Databases". Available. [Online]: www.ics.uci.edu/~mlearn/ML-Repository.http

[30] Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. Available. [Online]: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html