Memory Optimisation for Hardware Induction of Axis-parallel Decision Tree

Chuan Cheng and Christos-Savvas Bouganis
Department of Electrical and Electronic Engineering
Imperial College London
South Kensington Campus, London, UK
Email: {chuan.cheng09,christos-savvas.bouganis}@imperial.ac.uk

***Abstract*—In data mining and machine learning applications, the Decision Tree classifier is widely used as a supervised learning method not only in the form of a stand alone model but also as a part of an ensemble learning technique (i.e. Random Forest). The induction of Decision Trees (i.e. training stage) involves intense memory communication and inherent parallel processing, making an FPGA device a promising platform for accelerating the training process due to high memory bandwidth enabled by the embedded memory blocks in the device. However, peak memory bandwidth is reached when all the channels of the block RAMs on the FPGA are free for concurrent communication, whereas to accommodate large data sets several block RAMs are often combined together making unavailable a number of memory channels. Therefore, efficient use of the embedded memory is critical not only for allowing larger training dataset to be processed on an FPGA but also for making available as many memory channels as possible to the rest of the system. In this work, a data compression scheme is proposed for the training data stored in the embedded memory for improving the memory utilisation of the device, targeting specifically the axis-parallel decision tree classifier. The proposed scheme takes advantage of the nature of the problem of the decision tree induction and improves the memory efficiency of the system without any compromise on the performance of the classifier. It is demonstrated that the scheme can reduce the memory usage by up to 66% for the training datasets under investigation without compromise in training accuracy, while a 28% reduction in training time is achieved due to extra processing power enabled by the additional memory bandwidth.**

I. INTRODUCTION

In data mining and machine learning applications, the Decision Tree classifier is widely used as a supervised learning method not only in the form of a stand alone model but also as a part of an ensemble learning technique (i.e. Random Forest). The process consists of two major stages: training and classification. During the training stage, pre-collected examples (i.e. training data set) belonging to various object classes are processed by one of many training methods available (i.e. ID3, CART and C4.5 [1]–[3] to name a few). Based on the training data set, a tree-structure predicting model is induced, which is later used to classify unseen examples during the classification stage. The induction of a decision tree involves consecutive splittings of a training dataset in the feature space. The Axis-parallel split is the most basic but commonly used strategy along with oblique and non-linear weak classifier strategies. It is adopted in well known training methods including CART and C4.5 and has been incorporated in various software tools such as R, scikit-learn and WEKA [4]–[6] etc.

978-1-4799-5944-0/14/$31.00 ©2014 IEEE

To fulfill performance requirements in training speed for applications with large training data sets or with small time-window for training, critical stages or the entire training process have been mapped onto FPGAs for acceleration [7], [8]. FPGAs offer high memory bandwidth and flexible re-configurable fabric which makes them suitable for the target application of the induction of decision trees, which involves intense memory communication. Ideally, the entire training dataset should be stored in the embedded memory blocks of the FPGA to avoid repeated external memory access. However, in practise, the size of embedded memory on a modern FPGA device is relatively small (about 3MB for Altera Stratix series devices [9]), which significantly limits the size of the training problems that can be mapped, while possible strategies that are based on splitting the training dataset will produce overheads in the form of external memory access. Moreover, the memory bandwidth on FPGA is maximised when the channels of all block RAMs are free for concurrent communication. However, quite often, several block RAMs are often combined together to hold large dataset, reducing the maximum number of available memory channels. Therefore, efficient use of the embedded memory is critical not only for allowing larger training dataset to be processed on FPGA but also for making available as many memory channels as possible, resulting to shorter training times.

In this work, a data compression scheme is proposed for reducing the memory space required by the training data used at the training stage. Under the proposed scheme, the training dataset that is sent to the FPGA device is compressed before being stored in the embedded memory blocks, without compromising the performance of the generated predicting model. The proposed compression method takes advantage of the internal workings of the training algorithm for the induction of the decision tree, resulting in a lossless compression scheme. The freed memory space can be used to hold a larger training dataset, or to allow extra parallel processing power to further reduce the training time of the existing problem. The proposed scheme is evaluated under a selection of training datasets. It is demonstrated that the datasets can consume up to 66% less memory bits when compared with custom floating-point or custom fixed-point formats without having adverse impact on the accuracy. Moreover, further experiments demonstrate a 28% reduction in training time due to extra memory channels that are available enabled by the compression scheme.

II. Background

A. Training Decision Tree

A decision tree, like the one that is shown in Fig. 1, can be induced by many training methods which share the same basic framework i.e. splitting the training dataset recursively into many partitions in the feature space. The purpose of the training process is to find a set of *Test* nodes which is used for the construction of the predicting model. Each *Test* node in the tree corresponds to a split for the partition of the training set that reaches the node. The splitting depends on a weak learner resulted from the optimisation of an objective function I:

$$\theta_i = \arg_\theta \max I(S_i, \theta) \tag{1}$$

where θ_i is the optimal weak leaner selected out of all candidate learners θ for node i and S_i is the partition of training data reaching the node i. θ is defined by:

$$\theta = (\phi(v), \tau), v \in S_i \tag{2}$$

where v is the training data selected from the partition S_i. Function ϕ and threshold τ form an indicator function $[\phi(v) \geq \tau]$ based on which partition S_i is split into two parts. Both ϕ and τ are commonly determined based on the training method. In this work we target axes-parallel split which is defined as:

$$\phi = v_a, a \in A \tag{3}$$

Given a partition of training data S_i, v_a is one of the values in S_i corresponding to attribute a and A is the set of attributes in the feature space. Meanwhile τ are determined by taking the median of two adjacent v_a. One example of this is visualised in Figure. 2 where the data points in a partition within a $\mathbb{R}^3$ feature space are projected to attribute axis $a3$. The projection shows there are five different $v_a(a = 3)$ along axis $a3$, from which four different indicator functions can be derived. Then an optimal one is selected ($\tau = (d3+d4)/2$) based on which a hyper-plane separate the partition into two new partitions, one contains blue points and the other one contains red points.

The objective function I used to determine the optimal weak learner is also method dependent. Take CART method as an example, the function is defined by:

$$I = i(N) - P_L i(N_L) - P_R i(N_R) \tag{4}$$

where N is the partition reaching a *Test* node. N_L and N_R represent the the left and right branches derived from N. P_L and P_R are the proportion of data points allocated to the left and right branches respectively. $i()$ is the Gini impurity measurement defined by:

$$i(N) = \sum_{i \neq j} P(\omega_i)P(\omega_j) = \frac{1}{2}\left[1 - \sum_j P^2(\omega_j)\right] \tag{5}$$

where $P(\omega_i)$ and $P(\omega_j)$ refer to the proportion of data points with class i and j in the partition N respectively.

III. Optimisation for Embedded Memory

A. Data Compression

The key idea behind the proposed compression method in the Decision Tree training exploits the fact that for axes-parallel weak classifier, the real values of the training examples are not directly involved in the optimisation of the objective function I in Eqn. 4, but they are only needed to calculate the threshold τ for the weak learner ϕ as defined in Eqn. 2 by using the median of two adjacent values in the sorted sequence of the data. Thus, only the relative position of the data is required rather than the actual values. The proposed methodology works as follows. Given a training dataset (number of examples $\times$ number of attributes), consider each column as a list of values corresponding to one attribute. The proposed method replaces the actual values in the list with their corresponding sorted indices. In case there are data with the same value, the same index is used. Thus, a mapping to integer values is performed.

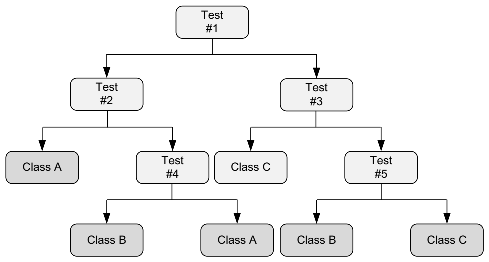

Fig. 1: A decision tree predicting model

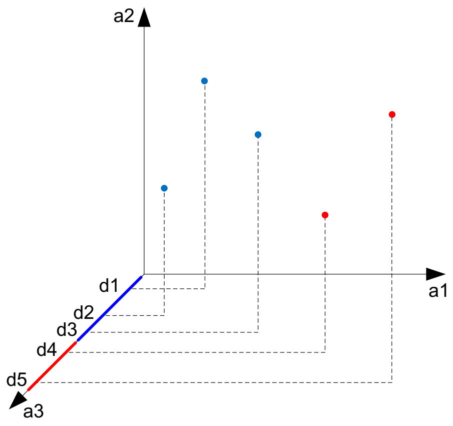

Fig. 2: Visualisation of axes-parallel weak classifier

B. Hardware Implementation

The hardware implementation of the proposed compression method is depicted in Figure. 3. The compression module is placed before the embedded memory block so the imported training data are compressed before being stored in the memory. The data are separated into several lists (i.e. each list corresponds to an attribute) and enter the compression module in a sequential manner. Each list contains values belonging to one attribute. For each list, a counter (counter #1) attaches an index i_n to each element, recording its original position. The list is then sent to a sorting module where the elements are re-ordered based on their values. A second counter (counter #2) is in place to assign another index i'_n to each re-ordered element, indicating its new position (rank). i'_n are used as the compressed data and are written to the embedded memory, with i_n being the writing addresses. By doing so, the internal

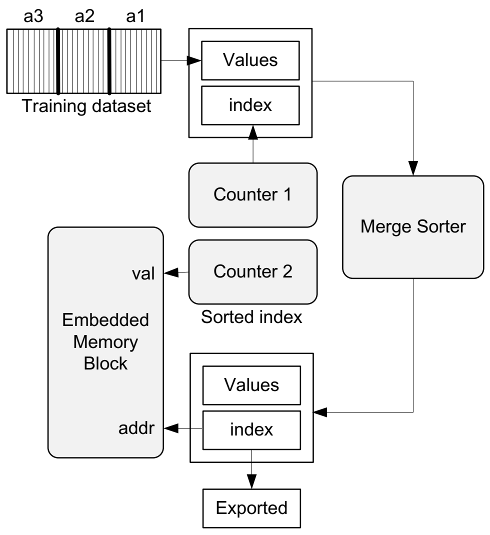

Fig. 3: Architecture of hardware implementation

layout of the training dataset before and after the compression remains the same.

Note that the threshold τ generated from the compressed data is represented by a pair of sorted indices i, j and can not be used directly in the predicting process. The indices pair indicates that real value threshold $\tau' = (d_i + d_j)/2$, where d_i and d_j are real values corresponding to the indices. A mechanism is designed to link i to d_i as illustrated in Figure. 4. Data array a_r on the right-side stores the uncompressed training data d. Data array a_l on the left receives the list of indices that are previously used as writing addresses for the memory block in the compression module (indices that are exported in Fig. 3). The elements in a_l serves like pointers storing the locations of the real values, so given an index i, the ith element in a_l contains the index(address) of d_i stored a_r.

The maximal rank produced for a list also indicates the number of different values N_d in it. The architecture in Fig. 3 assumes that the compression module is incorporated into the training circuitry and are synthesised together. In this case N_d is unknown when the word-length for the data is selected, so the word-length is set to be long enough to represent the number of elements in the list N_s. We notate this as *long* method since it requires longer word-length than it may be necessary in the given problem. If the compression process is performed before the synthesis of the training circuitry, the word-length can be configured to bound N_d, which guarantees the minimum possible word-length. We refer to this as the *short* method. However, the *short* method requires some prior knowledge of the dataset, which may not be available during the design time.

IV. PERFORMANCE EVALUATION

The data compression scheme is evaluated under two approaches. In the first approach, the performance of the proposed method is compared against other possible compression methods. In this work the reference designs are based on custom precision number representations aiming at the reduction of the required memory resources. In the second approach, the proposed scheme is incorporated with one of our previous works to evaluate its impact on a decision tree training system with regards to the resource utilisation, overhead and improvement in performance.

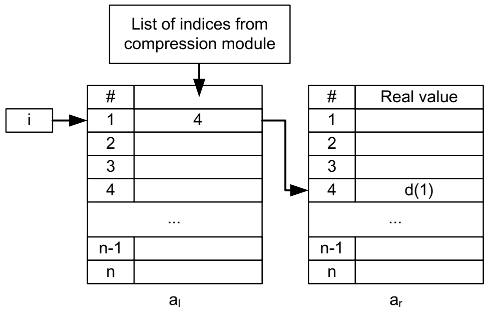

Fig. 4: Mechanism linking sorted index i to real value d_i

A. Memory Compression Performance

In this performance evaluation, several datasets from the UCI repository were selected, where all of them contain only numeric elements. By preprocessing the datasets, the required word-length for custom floating point precision and fixed point precision were deducted along with the number of bits required under the proposed scheme. Table I lists the required word-length for representing a data value corresponding to one attribute of the dataset using custom floating-point, custom fixed-point and the proposed compression scheme respectively. For the proposed scheme, results based on both methods are reported.

The word-length needed for the first two formats are calculated as follows. For the floating-point format, the length of mantissa is the minimum length that meets the sufficient and necessary condition for maintaining distinction among elements in the dataset [10]; the length of exponent is chosen to accommodate the dynamic range for all the values in the dataset. Regarding the fixed-point format, the fraction part length is determined as to have enough precision to represent the minimum distance between the sorted values in the dataset; the integer length is determined based on the absolute maximum value in the dataset.

Note that the word-length based on both floating-point and fixed-point format depends on the range or the precision of the values in the dataset whereas for the compression scheme it is only proportional to the number of (different) elements in the dataset. The compression is more effective when the distribution of the values is either extremely sparse or extremely dense. From the table, reduction for the first four datasets ranges from 33% to 53% [1]. For the following two datasets with relatively larger size, 'Segment' has both high precision and large range so after compression 66% of the word-length can be reduced, however for 'Waveform' due to larger size of the dataset and more dense distribution of the elements, *long* method is no longer efficient when compared with floating-point format. *short* method on the other side is

[1]The results are based on comparison between $min\{float, fixed\}$ and $min\{short, long\}$

TABLE I: Comparison of memory usage (bits per attribute per example)

Datasets	Trainining size	Custom float	Custom fixed	Proposed	
				short	*long*
Sonar	208	14	15	8	8
Breast cancer	699	19	33	10	10
Ionosphere	351	19	19	9	9
Vowel	990	18	15	10	10
Segment	2310	36	32	11	12
Waveform	5000	14	12	10	13

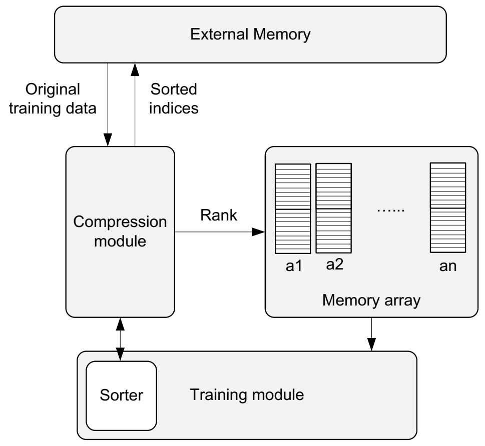

Fig. 5: Architecture of the compression module incorporated into an RF training system

able to reduce 17% of the word-length and guarantees the minimum word-length in all circumstances.

B. Impact on a Random Forest Training System

The hardware implementation of the *long* method is incorporated into an FPGA architecture that is used to accelerate the training process of Random Forest (RF) predicting model. The incorporation is depicted in Figure. 5. The training dataset is originally stored in the external memory, and when the training process is initiated, the dataset is sent to the compression module. The compression is done by utilising the existing sorter circuitry in the training module. The compressed dataset (rank) is then written to a memory array from which the data will be repetitively loaded for the training purpose. The memory array consists of a set of block RAMs with each one containing a list of training data corresponding to one or more attribute(s). In addition, the list of the sorted indices is exported by the compression module and is sent back to the external memory as explained in Section. III-B. The whole system is implemented in a Terasic DE4 board with Altera EP4SGX70HF35C2 FPGA with a working frequency of 200MHz.

Table II lists the resource utilisation of the compression module without taking into account the sorting circuitry (i.e. sorter), as this is already part of the original system. The required resources are compared to that of the whole system as well as the total resource available on the device. The architecture is configured for the "Sonar" dataset and is scaled down to use the minimal parallel processing elements (i.e. one) to see the maximum overhead of resource utilisation added by the compression module to such a system. The results show that the compression module accounts for a very small part of the total resource utilisation and is negligible on large FPGAs such as Stratix IV. The bottom row of the table shows the block RAM usage without using the compression scheme. A total of 28.8% of block RAMs are freed by the proposed compression technique.

TABLE II: Hardware utilisation of the compression module

	Compression module excl. sorter	Whole system	Percentage wrt. whole system	Percentage wrt. FPGA
LUTs	157	5924	2.7%	<1%
Logic register	173	7559	2.3%	<1%
Block RAMs (bits)	8528	204815	4.2%	<1%
Block RAMs (M9Ks)	2	37	5.4%	<1%
Block RAMs (M9Ks) no compression	-	52	-	-

Table. III presents the processing time of the system before and after incorporating the proposed compression module. The table is separated into two parts. In the top part, a practical training is performed, generating an ensemble of 500 decision trees with $mtry = \sqrt{n}$, where n is the number of attributes in the training dataset. The architecture is also configured to mimic an ordinary single decision tree training process by training single tree ensemble with $mtry = n$. The corresponding results are reported in the bottom part of the table. Note that access time to external memory is not given for the architecture with compression module, as it is already accounted for in the data compression time.

From the table it can be seen that, for the multiple trees training task (500 trees, $mtry = 8$), the total training time is reduced by 28.3% by using the data compression, as extra embedded memory allows more parallel processing power to spread the work load. However for the single tree training task (single tree, $mtry = 60$), the architecture without the compression module consumes 14.6% less time than the system that includes the proposed compression module. This is expected since no parallelism can be exploited at the decision tree construction level, so the available memory bandwidth gained by the proposed method cannot be exploited. However, the memory space saved by the proposed compression method can still be used to hold a bigger training dataset.

V. Conclusion

In this work, a data compression scheme is proposed to optimise the use of embedded memory on FPGA for axes-parallel decision tree induction. Depending on the nature of training dataset, the scheme can effectively reduce the word-length needed to represent the training data without any loss of information. The evaluation demonstrates that the data compression can produce two benefits. Firstly, the embedded memory saved may support extra parallel processing elements which improves training speed; and secondly, the freed memory can be used to hold bigger training dataset so that the range of applications can be extended.

TABLE III: Decomposed processing time (ms)

500 tree, $mtry = 8$		
	Without compression	With compression
Ext. memory to FPGA	0.056	-
Data compression	-	0.17
Training	7.04	4.92
Total	7.10	5.09
1 tree, $mtry = 60$		
Ext. memory to FPGA	0.056	-
Data compression	-	0.17
Training	0.58	0.58
Total	0.64	0.75

References

[1] J. R. Quinlan, *Discovering rules by induction from large collections of examples*. Edinburgh university press, 1979.

[2] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[3] J. R. Quinlan, *C4.5: programs for machine learning Vol.1*. Morgan kaufmann, 1993.

[4] "R project," Aug. 2014. [Online]. Available: www.r-project.org

[5] F.Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *The Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[6] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "the weka data mining software: an update," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[7] G. Memik and A. Choudhary, "An fpga implementation of decision tree classification," in *2007 Design, Automation&Test in Europe Conference&Exhibition*, 2007, pp. 1–6.

[8] G. Chrysos, P. Dagritzikos, I. Papaefstathiou, and A. Dollas, "Hc-cart: A parallel system implementation of data mining classification and regression tree (cart) algorithm on a multi-fpga system," *ACM Transactions on Architecture and Code Optimisation (TACO)*, vol. 9, no. 4, 2013.

[9] *TriMatrix embedded memory blocks in Stratix IV devices*, Altera Corporation.

[10] J. M. Muller, N. Brisebarre, F. de Dinechin, C. P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehle, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhauser Basel, 2010, ch. 2.7.