electronics

Article

Universal Reconfigurable Hardware Accelerator for Sparse Machine Learning Predictive Models

Vuk Vranjkovic †,‡, Predrag Teodorovic *,‡ and Rastislav Struharik ‡

Faculty of Technical Sciences, University of Novi Sad, 21000 Novi Sad, Serbia; bykbpa@uns.ac.rs (V.V.); rasti@uns.ac.rs (R.S.)
* Correspondence: t_pedja@uns.ac.rs; Tel.: +381-6310-28-109
† Current address: Trg Dositeja Obradovica 6, 21000 Novi Sad, Serbia.
‡ These authors contributed equally to this work.

Abstract: This study presents a universal reconfigurable hardware accelerator for efficient processing of sparse decision trees, artificial neural networks and support vector machines. The main idea is to develop a hardware accelerator that will be able to directly process sparse machine learning models, resulting in shorter inference times and lower power consumption compared to existing solutions. To the author's best knowledge, this is the first hardware accelerator of this type. Additionally, this is the first accelerator that is capable of processing sparse machine learning models of different types. Besides the hardware accelerator itself, algorithms for induction of sparse decision trees, pruning of support vector machines and artificial neural networks are presented. Such sparse machine learning classifiers are attractive since they require significantly less memory resources for storing model parameters. This results in reduced data movement between the accelerator and the DRAM memory, as well as a reduced number of operations required to process input instances, leading to faster and more energy-efficient processing. This could be of a significant interest in edge-based applications, with severely constrained memory, computation resources and power consumption. The performance of algorithms and the developed hardware accelerator are demonstrated using standard benchmark datasets from the UCI Machine Learning Repository database. The results of the experimental study reveal that the proposed algorithms and presented hardware accelerator are superior when compared to some of the existing solutions. Throughput is increased up to 2 times for decision trees, 2.3 times for support vector machines and 38 times for artificial neural networks. When the processing latency is considered, maximum performance improvement is even higher: up to a 4.4 times reduction for decision trees, a 84.1 times reduction for support vector machines and a 22.2 times reduction for artificial neural networks. Finally, since it is capable of supporting sparse classifiers, the usage of the proposed hardware accelerator leads to a significant reduction in energy spent on DRAM data transfers and a reduction of 50.16% for decision trees, 93.65% for support vector machines and as much as 93.75% for artificial neural networks, respectively.

Keywords: decision trees; support vector machines; artificial neural networks; hardware accelerator architectures; edge computing; sparse predictive models

check for updates

Citation: Vranjkovic, V.; Teodorovic, P.; Struharik, R. Universal Reconfigurable Hardware Accelerator for Sparse Machine Learning Predictive Models. *Electronics* **2022**, *11*, 1178. https://doi.org/10.3390/electronics11081178

Academic Editors: Iakovos Mavroidis, Sotiris Ioannidis and Konstantinos Georgopoulos

Received: 1 March 2022
Accepted: 25 March 2022
Published: 8 April 2022

1. Introduction

Until recent discoveries of convolutional neural networks and the other deep learning architectures, multi layer perceptron (MLP) artificial neural networks (ANNs), decision trees (DTs) and support vector machines (SVMs) were recognized as the most popular predictive models in the field of machine learning (ML). Although CNNs have replaced ANNs, DTs and SVMs in the fields of computer vision and natural language processing, ANNs, DTs and SVMs are still among the most widely used predictive models in the field of data mining [1–3].

A predictive model based on DTs was first presented in the literature in 1984 [4], while axis-parallel DTs were introduced a few years later [5]. SVMs were first introduced in 1995 [6], while ANNs were presented as a predictive model in [7], even though the computational model for ANNs was proposed a long time ago in [8]. Although ANNs, DTs and SVMs have long been known by the scientific community, they are still widely used for discovering patterns in large datasets (data mining) [1–3]. There are numerous software platforms used for this purpose, such as RapidMiner [9], R project [10] or Pentaho/WEKA [11]. However, when machine learning classifiers are used for solving large scale classification problems, like data mining, or in real time data processing applications, such as network anomaly detection [12] or real-time trading [13,14], the instance classification duration becomes a critical metric for the classifier performance evaluation. One way to improve the instance classification duration is to implement ML classifiers directly in hardware.

A significant effort has been made by the scientific community in this direction. Regarding DTs, a number of FPGA DT implementations were presented in the literature [15–20]. These proposed architectures dealt with the acceleration of the axis-parallel, oblique or nonlinear DTs and the ensembles of dense DTs. There are numerous SVM hardware implementations presented in the literature [21–27], while ANNs is, arguably, the predictive model with the most hardware implementations [28–32].

Even though the above-mentioned hardware implementations of ML predictive models outperform corresponding software solutions by several orders of magnitude, all of them can implement only a single type of the classifier: DT, SVM or ANN. In [33], the authors presented the universal coarse-grained reconfigurable hardware accelerator for hardware implementation of three ML predictive model types: ANNs, DTs and SVMs. The implementation is based on the fact that the DOT product of two multi-dimensional vectors, as the core data processing operation, is common for all three supported ML models. The same authors extended their solution to homogeneous and heterogeneous ensemble classifiers in [34].

Another way to improve classifier throughput performance is to compress and reduce the size of the predictive model, by using different sparsification approaches. Sparsification techniques have been mainly explored in the field of ANNs and convolutional neural networks (CNNs) [35–40] and have resulted in the significant reduction in model size. Compression has been significantly less explored in the fields of DTs and SVMs. Authors in [41,42] recognized the benefits of minimizing the number of non-zero hyperplane coefficients in oblique DTs. However, the focus in these studies was a feature/attribute selection and detection of irrelevant and noisy features and not the reduction in model size or inference process acceleration. Compression of the SVM size by using a smart selection of support vectors during the training was presented in [43], while in [44] authors proposed the complementary idea for the SVM size reduction through the removal of attributes from support vectors.

The benefits of having a sparse predictive model are fully exploited only when the model is being executed on a hardware accelerator that can process sparsified models. In the field of CNN accelerators this has been heavily exploited, resulting in numerous solutions being proposed [45–49]. Surprisingly, in the field of DTs, SVMs and ANNs, only a handful of hardware accelerators capable of directly processing sparse models have been proposed in [44,50], despite the obvious benefits of accelerating sparse ML predictive models. A hardware accelerator of sparse oblique DTs was presented in [50], where it was reported that oblique DT sparsification led to both instance processing speedup and memory reduction. A hardware accelerator for sparse SMVs was proposed in [44], with similar benefits regarding the improvements in inference speed.

To the best of our knowledge, there is no published result concerned with the development of a hardware accelerator capable of accelerating different sparse ML model types, like DTs, SVMs and ANNs. This could be of a great interest for the applications relying on using hybrid-classifier systems, for example, [51–56].

In this study, we present the Sparse Reconfigurable Machine Learning Classifier, SRMLC—an application specific hardware accelerator for efficient processing of sparsified decision trees, support vector machines and artificial neural networks. The SRMLC is based on the implementation proposed in [33] where the underlying core operation is MAC (multiply and accumulate); however, it is optimized in order to support the acceleration of sparse predictive models in which the majority of model weights are set to zero. Consequently, without any performance degradation in terms of classifier accuracy, the SRMLC processing latency is significantly reduced, as a result of skipping numerous MAC operations with zero-valued operands. Compared to previously published results in [33], there are five major contributions of our approach:

1. Sparsification—our design is the first universal reconfigurable machine learning classifier accelerator which is optimized to support sparse data representations and which benefits from such sparse data manipulations.
2. Scalability—one of the major design goals during the development of the SRMLC architecture was supporting better scalability on FPGA platforms. This is feasible due to the fact that one MAC unit, a basic building block within the SRMLC, uses one DSP, one BRAM and 300 LUTs, compared to previously published RMLC [33], where scalability was constrained by using too many BRAM blocks per one DSP.
3. Improved throughput—as a result of the classifier sparsification, the SRMLC has significantly improved throughput for DTs and SVMs. Regarding MLP ANNs, throughput is improved as a result of the fact that a single layer can be assigned to multiple MAC units for processing, which is not possible in the architecture proposed in [33].
4. Reduced processing latency—the SRMLC introduces a huge reduction in instance processing latency, as it allows for the usage of multiple MAC blocks for processing a single classification instance. The latency is reduced for all supported classifier types.
5. Energy efficiency—since the SRMLC uses sparse data representation, it suffers much less from the well known issue of power hungry data transfer between the DRAM and the accelerator. Improved energy efficiency is a consequence of the significantly reduced amount of data that needs to be transferred between the external DRAM and the accelerator core.

To the best of our knowledge, the SRMLC is the first reported hardware accelerator for sparse classifiers of this kind. In order to demonstrate functionality of the SRMLC, in this study we will also present algorithms for the training of sparse ANNs, SVMs and DTs and translate the compressed trained models into the sparse binary format, which can be directly handled by the SRMLC.

The remainder of this study is organized in the following way. In Section 2, we will present training algorithms for sparse classifiers which can be used to obtain predictive models that are significantly reduced in size. Three algorithms are presented: one for sparse ANN training, another for sparse DT induction and the third for sparse SVM training. In Section 3, a universal reconfigurable hardware accelerator for sparse classifiers is introduced. The proposed hardware accelerator benefits from the sparsity in predictive models and performs faster classifications by computing only multiplication operations with non-zero operands. In Section 4, we report the experimental results of the benchmarking of our SRMLC architecture performance using datasets from the UCI machine learning repository. The conclusion and final remarks are given in the Section 5.

2. Training of Sparse Predictive Models

In order to provide a sparse representation that will be efficiently processed by the SRMLC, a classifier's training process is modified by removing model parameters, according to the desired level of compression. However, the reduction in predictive model size has to take into account the resulting model accuracy drop. Usually, in the available literature, 1% of the absolute accuracy drop is acceptable when sparsifying the predictive model, so we have used this as a reference during our training process: model sparsification stops when the absolute accuracy of the sparse model is more than 1% below the absolute accuracy of a non-sparsified model. The same approach is used for all three classifiers, even though the training process itself is significantly different for each classifier type.

2.1. Pruning of ANN Model during Training

First, we will present a pruning algorithm of ANN model, which is used during the ANN training phase, in order to obtain sparse ANN model.

An ANN can be considered as a weighted directed graph, where nodes are artificial neurons, which are connected by directed weighted edges. Recurrent ANNs are ANNs which allow feedback connections, while feed-forward ANNs do not. In Multi-Layer Perceptron, a widely used type of feed-forward ANNs, individual neurons are organized into layers and the only connections that are allowed are the ones between adjacent layers of the network. Besides that, neurons are connected in a feed-forward manner, with no connections between neurons of the same layer and no feedback connections between the layers. The structure of the MLP ANN is shown in Figure 1.

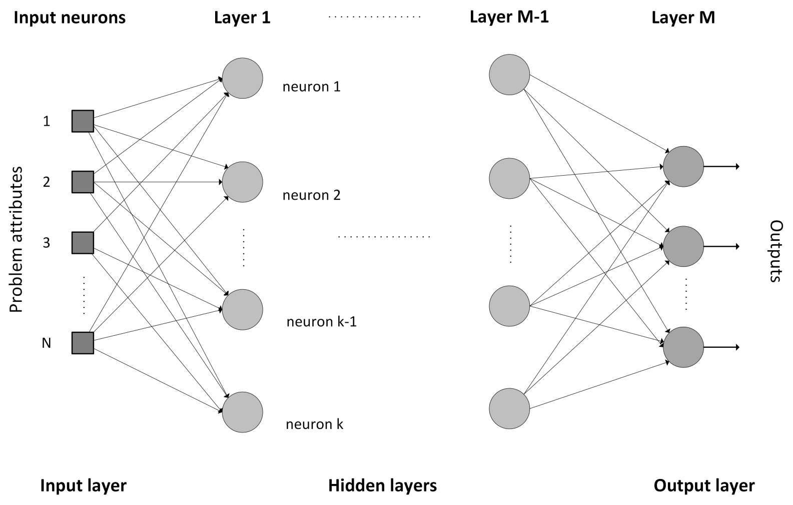

Figure 1. MLP ANN structure.

Three types of layers exist in MLP ANN: input, hidden and output. The input layer is composed of N input neurons, where N is the number of problem attributes. The output layer calculates the output values of the MLP ANN, while all layers between the input and the output are considered as hidden. Each hidden layer consists of an arbitrary number of neurons. Each (let us say k^{th}) neuron in the hidden or the output layer calculates its output as:

$$y_k = f(\mathbf{w} \cdot \mathbf{x} + b_k) \tag{1}$$

where $\mathbf{x}$ is the input vector for a neuron containing N neuron outputs from the previous ANN layer

$$\mathbf{x} = (x_1, x_2, \ldots, x_N),$$

$\mathbf{w}$ is a weight vector

$$\mathbf{w} = (w_{k1}, x_{k2}, \ldots, x_{kN}),$$

and b_k is a scalar, usually called the offset. For neurons in the input layer, $\mathbf{x}$ is a vector holding attribute values of the input instance. As mentioned above, for the neurons within hidden and output layers, vector $\mathbf{x}$ is the vector composed of the outputs from neurons in the previous layer of the network. Weight vector $\mathbf{w}$ for each layer has the same length as $\mathbf{x}$ for that particular layer. Function $f : \mathbf{R} \rightarrow \mathbf{R}$ is called the activation function and it can be either a linear or nonlinear real function. Many different activation functions can be used in MLP ANNs, for example, the hyperbolic tangent and sigmoid functions:

$$f_{tanh}(x) = tanh(x) \tag{2}$$

$$f_{sigmoid}(x) = \frac{1}{1+e^{-x}}. \tag{3}$$

In order to understand the ANN pruning process, let us observe a hidden or the output layer with M neurons and with N-dimensional input $\mathbf{x}$. Then, the output of this layer can be written as

$$\mathbf{Y} = F(\mathbf{x} \cdot \mathbf{W} + \mathbf{B}) \tag{4}$$

where $\mathbf{x_{1\times N}} = (x_1, x_2, \ldots, x_N)$ is an input for the layer, $\mathbf{W}$ is a matrix

$$\mathbf{W_{N\times M}} = \begin{bmatrix} w_{11}, w_{12}, \ldots, w_{1N} \\ w_{21}, w_{22}, \ldots, w_{2N} \\ \vdots \\ w_{M1}, w_{M2}, \ldots, w_{MN} \end{bmatrix} \tag{5}$$

and $\mathbf{B_{1\times M}} = (b_1, b_2, \ldots, b_M)$ is a vector consisting of the corresponding layer's neuron offsets. Finally, $\mathbf{Y_{1\times M}} = (y_1, y_2, \ldots, y_M)$ is an output vector consisting of the outputs from all neurons in the given layer.

The training of the ANN is the process by which the dataset consisting of N_{inst} problem instances is applied to the ANN. During the training, network parameters $\mathbf{W}$ and $\mathbf{b}$ (for each layer) are fine-tuned to provide the ANN output, for each given input instance, that is equal to the expected output.

The pruning ANN algorithm is run once the ANN training is complete. The goal of the pruning algorithm is to determine which network weights are the least significant and set corresponding elements of matrix $\mathbf{W}$ to zero, for each ANN layer. The pruning ANN algorithm iteratively repeats this procedure, as long as the accuracy drop of the reduced model is acceptable. As a result, the pruned ANN will have the majority of weights set to zero, allowing the SRMLC to skip all computations where these zero-valued weights are multiplicands in multiply operations. In order to prune the ANN, the pruning algorithm starts with a low pruning factor *MIN_PRUN_FACT* (for example 10%), increasing the pruning factor by *PRUN_FACT_INC_STEP* (for example 5%) in each iteration.

At the beginning, Algorithm 1 initializes the current pruning factor and evaluates the non-pruned trained ANN model. At line 5, a main loop starts and repeats as long as the accuracy of a pruned model is not degraded. Algorithm creates an empty array *abs_W_array* of the same size as *W_array* and populates it with the absolute values from corresponding elements in *W_array*, at lines 7–10. *W_array*, which is an input to the algorithm, represents all connections within the MLP ANN and hence holds W matrices from Equation (5) for all layers in the given MLP ANN. At line 11, a required number of zero elements is determined based on the current pruning factor, number of ANN layers and matrix $\mathbf{W}$ size. At lines 13–34, *num_zero_elem* minimum elements in *abs_W_array* are found and corresponding elements in *temp_W_array* are set to zero. In order to avoid selection of the same element several times, *min_elem_set* is used to store previously selected elements. Prior to updating *current_acc* at line 37, we retrain the MLP ANN with modified weights in order to increase the classifying accuracy, at line 36. Please note that *temp_W_array* is used for pruning, while *W_array* is only updated once we are confident that the accuracy of pruned model is still acceptable, at the beginning of a new iteration. If *W_array* was updated directly, the returned value from the algorithm would not be the correct one.

Algorithm 1 ANN_Pruning

```
Input: W_array
Output: pruned W_array
1: $current\_pruning\_factor \leftarrow MIN\_PRUN\_FACT$
2: $non\_pruned\_acc \leftarrow$ accuracy of model with weights $W\_array$
3: $current\_acc \leftarrow non\_pruned\_acc$
4: $temp\_W\_array \leftarrow W\_array$
5: while $current\_acc \geq non\_pruned\_acc - 1\%$ do
6:   $W\_array \leftarrow temp\_W\_array$
7:   $abs\_W\_array \leftarrow$ empty array same size as $temp\_W\_array$
8:   for $i = 1, 2, \ldots, \texttt{length}(temp\_W\_array)$ do
9:     $abs\_W\_array[i] = abs(temp\_W\_array[i])$
10:  end for
11:  $num\_zero\_elem = current\_pruning\_factor * \texttt{lenght}(temp\_W\_array) * \texttt{size}(W)$
12:  $min\_elem\_set \leftarrow$ empty set
13:  while $num\_zero\_elem > 0$ do
14:    $value_{min} \leftarrow MAX\_FLOAT$
15:    $i_{min} \leftarrow -1$
16:    $j_{min} \leftarrow -1$
17:    $k_{min} \leftarrow -1$
18:    for $i = 1, 2, \ldots, \texttt{length}(temp\_W\_array)$ do
19:      for $j = 1, 2, \ldots, \texttt{width}(W)$ do
20:        for $k = 1, 2, \ldots, \texttt{height}(W)$ do
21:          $W_{current} \leftarrow abs\_W\_array[i]$
22:          if $W_{current}[j][k] < value_{min}$ and $(i, j, k)$ not in $min\_elem\_set$ then
23:            Add $(i, j, k)$ into $min\_elem\_set$
24:            $i_{min} \leftarrow i$
25:            $j_{min} \leftarrow j$
26:            $k_{min} \leftarrow k$
27:            $value_{min} \leftarrow W_{current}[j][k]$
28:          end if
29:        end for
30:      end for
31:    end for
32:    $temp\_W\_array[i_{min}][j_{min}][k_{min}] = 0$
33:    $num\_zero\_elem \leftarrow num\_zero\_elem - 1$
34:  end while
35:  $current\_pruning\_factor \leftarrow current\_pruning\_factor + PRUN\_FACT\_INC\_STEP$
36:  retrain ANN with weights $temp\_W\_array$
37:  $current\_acc \leftarrow$ accuracy of model with weights $temp\_W\_array$
38: end while
39: return $W\_array$
```

2.2. DT Model Sparsification during Induction

For the given classification problem, represented by a set of n numerical attributes $A_i, i = 1, 2, \ldots, n$, axis-parallel DTs compare a single attribute A_i, from the test instance, against the corresponding threshold a_i and check if $A_i > a_i$. Such test is performed in each of the DT nodes. Inducing (training) of axis-parallel (or orthogonal) DT assumes the assignment of an attribute to each DT node, hence the order of comparisons, as well as the threshold value required for each comparison, a_i. Oblique DTs represent generalization of axis-parallel DTs, allowing multiple attribute testing in each DT node. As a result, oblique DTs are usually smaller in size, while providing higher classifying accuracy, when compared to corresponding axis-parallel DTs. In oblique DTs, multivariate testing is expressed with the following formula:

$$\sum_{i=1}^{n} a_i \cdot A_i + a_{n+1} > 0 \tag{6}$$

In Equation (6) coefficients $a_i, i = 1, \ldots, n+1$ are called the hyperplane coefficients. Finding the optimal oblique DT is proven to be an NP-complete problem [57], so many oblique DT induction algorithms use some kind of heuristic in order to find sub-optimal hyperplane coefficients, some of them being [58,59]. HereBoy evolutionary algorithm [60] was used in [61] to solve the hard oblique induction problem, while the authors in [50] extended the algorithm in order to support the induction of sparse oblique DTs. In order to obtain sparse DTs with high accuracy, which will be processed by the SRMLC, we decided to use the similar recursive evolutionary algorithm for the induction of sparse DTs in this study.

Algorithm 2 builds maximally sparsified DT which has an acceptable accuracy drop, compared to the non-sparsified DT. At lines 1–2, non-sparsified DT is built using Algorithm 3 and its accuracy is evaluated on a validation subset. Using the same Algorithm 3, at lines 5–10, sparsified DTs are built, each time with increased sparsification factor (starting from MIN_SPARS_FACT, increased by $SPARS\_INC\_FACT$ in each iteration), until the evaluated accuracy drops below the tolerated threshold. Algorithm 3, used by Algorithm 2, is a recursive algorithm which builds sparse oblique DT, based on a target sparsification factor (required percentage of zero-valued coefficients in the output DT), provided as an input parameter. The other input for the algorithm is a set of input instances, where each instance contains the instance attributes and output class. At line 1, a new node is created. Lines 2–4 represent the terminating condition of the recursive algorithm—once all input instances belong to the same class, that root is marked as a leaf and the corresponding label is added. If this is not the case, assign the most appropriate label to this node before entering the loop at lines 7–22. In this loop, after finding sub-optimal hyperplane position by using the HereBoy algorithm, a required number of hyperplane coefficients, calculated from $sparse\_coef\_perc$, is set to zero. This is completed in the loop at lines 11–19 where, one after the other, each coefficient from the copy of $hyperplane\_coefs$ is set to zero, followed by the calculation of resulting fitness. The coefficient which has the smaller impact on the fitness is considered the least important and set to zero after the complete set of coefficients has been investigated (lines 20–21). This is repeated until the percentage of zero-valued coefficients reaches $sparse\_coef\_perc$. When completed, at lines 23 and 24, input instance set $input\_instances$ is split into two disjoint subsets based on their position relative to the hyperplane. Two subsets are then used as an input for recursive calls of Algorithm 3 (lines 25–26) where the right and the left subtree from the node $root$ are built in the same manner. Once a complete DT is built, the $root$ node representing the tree is returned from the algorithm. Next, we will explain in more detail the evolutionary algorithm $Find\_best\_hyperplane\_pos$, which is invoked at line 8.

An input for Algorithm 4 is a set of input instances. First, it creates initial hyperplane coefficients providing that not all instances from $input\_instances$ are located on the same side of the hyperplane. The algorithm then iterates through a predefined number of generations, mutating hyperplane coefficients with certain probability in each generation. Mutation here refers to a random bit flip in the binary representation of hyperplane coefficients. If new, a mutated hyperplane has a better fitness compared to the old one, and it will proceed to the next generation as the best individual. At the end, after $NUM\_GENERATIONS$, the best individual will be returned back as the output of the algorithm.

In order to calculate a fitness, both in Algorithms 3 and 4, the Algorithm 5 is used.

Algorithm 5 first finds k as the total number of classes in the given classification problem. Then, the total number of input instances, N, is determined as the length of the input array $input\_instances$ which is used in the current DT node. N_i is also determined as the number of training instances that belong to class $i, i = 1, \ldots, k$, from the total of N instances. At the end, N_{1i} is calculated as the number of instances belonging to class $i, i = 1, \ldots, k$, which is located above the hyperplane represented by $hyperplane\_coefficients$. Algorithm 5 returns value $0 \leq fitness \leq 1$.

Algorithm 2 Sparsify_oblique_DT

```
    Input: input_instances
    Output: maximally sparsified DT
 1: $non\_sparse\_model \leftarrow$ Sparse_oblique_DT_induction(input_instances, 0)
 2: $non\_sparse\_acc \leftarrow$ evaluate $non\_sparse\_model$ on validation subset
 3: $current\_spars\_factor \leftarrow MIN\_SPARS\_FACT$
 4: $sparse\_acc \leftarrow non\_sparse\_acc$
 5: while $sparse\_acc \geq non\_sparse\_acc - 1\%$ do
 6:   $sparse\_model \leftarrow temp\_sparse\_model$
 7:   $temp\_sparse\_model \leftarrow$ Sparse_oblique_DT_induction(input_instances, $current\_spars\_factor$)
 8:   $sparse\_acc \leftarrow$ evaluate $temp\_sparse\_model$ on validation subset
 9:   $current\_spars\_factor \leftarrow current\_spars\_factor + SPARS\_INC\_FACT$
10: end while
11: return $sparse\_model$
```

Algorithm 3 Sparse_oblique_DT_induction

```
    Input: input_instances, sparse_coef_perc
    Output: sparsified dt
 1: $root \leftarrow$ new node in DT
 2: if for all elements of $input\_instances$ output class is the same then
 3:   Declare $root$ as a leaf and set $root.label = input\_instances.output\_class$
 4: else
 5:   Set $root.label$ to most frequent $input\_instances.output\_class$
 6:   $removed\_coefs \leftarrow$ emtpy set
 7:   repeat
 8:     $hyperplane\_coefs =$Find_best_hyperplane_pos($input\_instances$)
 9:     $best\_fitness =$ Calc_fitness($hyperplane\_coefs$)
10:     $fit\_diff \leftarrow best\_fitness$
11:     for all $index$ in $\{1,2,\ldots$ length($hyperplane\_coefs$)$\} \setminus removed\_coefs$ do
12:       Copy $hyperplane\_coefs$ to $new\_hyperplane\_coefs$
13:       $new\_hyperplane\_coefs[index] \leftarrow 0$
14:       $new\_fitness =$Calc_fitness($new\_hyperplane\_coefs$)
15:       if $best\_fitness - new\_fitness < fit\_diff$ then
16:         $worst\_coef\_index \leftarrow index$
17:         $fit\_diff \leftarrow best\_fitness - new\_fitness$
18:       end if
19:     end for
20:     $hyperplane\_coefs[worst\_coef\_index] \leftarrow 0$
21:     Add $worst\_coef\_index$ to $removed\_coefs$
22:   until percentage of zero elements in $hyperplane\_coefs \geq sparse\_coef\_perc$
23:   $input\_instances\_above \leftarrow$ all elements from $input\_instances$ above hyperplane
24:   $input\_instances\_below \leftarrow$ all elements from $input\_instances$ below hyperplane
25:   $root.right =$ Sparse_oblique_DT_induction($input\_instances\_above, sparse\_coef\_perc$)
26:   $root.left =$ Sparse_oblique_DT_induction($input\_instances\_below, sparse\_coef\_perc$)
27: end if
28: return $root$
```

Algorithm 4 Find_best_hyperplane_pos

```
    Input: input_instances
    Output: hyperplane_coefficients
 1: $hyperplane\_coef \leftarrow$ initial hyperplane coefficients
 2: for $i$ in 1,... $NUM\_GENERATIONS$ do
 3:     if random(0..100) $> mutation\_probability$ then
 4:         $new\_hyperplane\_coef \leftarrow$ Mutate($hyperplane\_coef$)
 5:     else
 6:         $new\_hyperplane\_coef \leftarrow hyperplane\_coef$
 7:     end if
 8:     if Calc_fitness($new\_hyperplane\_coef$) > Calc_fitness($hyperplane\_coef$) then
 9:         $hyperplane\_coef \leftarrow new\_hyperplane\_coef$
10:     end if
11: end for
12: return $hyperplane\_coef$
```

Algorithm 5 Calc_fitness

```
    Input: hyperplane_coefficients, input_instances
    Output: fitness
 1: $k \leftarrow$ number of classes of the classification problem
 2: $N \leftarrow$ len($input\_instances$)
 3: for $i = 1, \ldots, k$ do
 4:     $N_i \leftarrow$ number of training instances that belong to class $i$
 5:     $N_{1i} \leftarrow$ number of training instances that belong to class $i$ and are located above
        hyperplane defined by $hyperplane\_coefficients$
 6: end for
 7: $fitness \leftarrow 1 + \frac{1}{N}\left[\sum_{i=1}^{k} N_{1i} \cdot log_2 \frac{N_{1i}}{\sum_{j=1}^{k} N_{1j}} + \sum_{i=1}^{k} (N_i - N_{1i}) \cdot log_2 \frac{N_i - N_{1i}}{N - \sum_{j=1}^{k} N_{1j}}\right]$
 8: return $fitness$
```

2.3. Attribute Sparse Training of SVM

Similar to other supervised machine learning algorithms, SVMs are first trained during the learning phase, followed by the predicting phase when SVMs are used to classify input instances. During the learning phase, a training set TS with m instances is used: $TS = \{x_i, y_i\}$, $i = 1, 2, \ldots, m$, $x_i \in X \subseteq \mathbb{R}^n$, $y_i \in Y = \{+1, -1\}$. The goal of SVM training is to design a hyperplane which will separate "positive" input instances from the "negative" ones, while trying to maximize the margin between the hyperplane and the closest instances on both sides. Hyperplane equation is

$$\mathbf{w}^T \cdot \mathbf{x} + b = 0, \mathbf{w}, \mathbf{x} \in \mathbb{R}^n. \quad (7)$$

Even though the hyperplane is designed with constraint to maximize the distance from the closest input instances, in general, this condition cannot be fulfilled for all input instances. In order to solve this, the SVM algorithm allows the incorrect classification for some of the instances from the training set. The problem of finding an optimal separating hyperplane can be defined formally as the constrained quadratic programming (CQP) problem:

$$\min_{w,b}\left[\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{m}\varepsilon_i\right]$$
$$y_i(\mathbf{w}^T\mathbf{x}+b) \geq 1-\varepsilon_i,\ \varepsilon_i \geq 0,\ \forall i \in 1\dots m$$
$$C > 0$$

In the optimization problem given above, one part $\frac{1}{2}\|\mathbf{w}\|^2$ positions the hyperplane so that the margin is maximized, while the second part $C\sum_{i=1}^{m}\varepsilon_i$ relocates the hyperplane in order to minimize the number of misclassified training set instances. A parameter C defines the trade-off between those two complementary conditions. By using Lagrange multipliers, the original CQP problem is transformed into its dual QP form, which is easier to solve:

$$\max_{\alpha}\left[r^T\alpha - \frac{1}{2}\alpha^T\mathbf{W}\alpha\right]$$
$$\alpha = (\alpha_1, \alpha_2, \dots, \alpha_m), r = (r_1, r_2, \dots, r_m), y = (y_1, y_2, \dots, y_m)$$
$$w_{ij} = y_i y_j K(x_i, x_j),\ i = 1, \dots, m,\ j = 1, \dots, m$$
$$0 \leq \alpha_i \leq C,\ r_i = 1,\ i = 1, \dots, m$$
$$y^T \cdot \alpha = 0$$

In the dual QP problem definition from above, W is a symmetric positive semidefinite $m \times m$ matrix. Beside input instance labels y_i, y_j, each element w_{ij} from W is calculated by using a kernel function K. Some of the most popular kernel functions are

$$K(x_i, x_j) = (x_i \cdot x_j + 1)^d, \text{called polynomial}$$
$$K(x_i, x_j) = \tanh(ax_i \cdot x_j + b),\ \text{called sigmoid}$$

An efficient algorithm for solving the QP problem called Sequential Minimal Optimization (SMO) is proposed in [62]. Once a training phase is completed, a majority of Lagrange multipliers α_i will be zero, and only non-zero multipliers will be the ones corresponding to, so called, support vectors. These are the input instances that are located closest to the hyperplane, from both the "positive" and "negative" side. A number of support vectors (l) are usually significantly smaller than the total number of training instances: $l << m$. In predicting phase, only support vectors are used for the classification of a new input instance:

$$\mathbf{V}(\mathbf{x}) = b + \sum_{i \in SV} y_i \alpha_i K(\mathbf{x_i}, \mathbf{x})$$

where $\mathbf{x_i}$ is the support vector and $\mathbf{x}$ is the input instance to be classified. If the result of $\mathbf{V}(\mathbf{x})$ is negative, an input instance is classified as a class -1; otherwise, it is classified as +1. A generalization of the presented mathematical models and methods, which allows for multi-class prediction, is proposed in [63].

In order to decrease the number of multiplications during the predicting phase, Algorithm 6, presented below, will eliminate some of the input instance attributes from the input instance set. This will be done by setting "the least significant" attributes to zero, resulting in the reduction in the multiplication number during the input instance classification.

Algorithm 6 first trains SVM by using the original input dataset and then evaluates the model at lines 1–2. At line 3, an initial value of a target sparsification percentage is set using the MIN_SPARS_PERC parameter. A main loop (lines 6–22) is executed until the accuracy of a sparsified model drops below the tolerated lower threshold, calculated from the accuracy of the non-sparsified model. Within the loop, at line 8, this algorithm sorts the input dataset in descending order with respect to the number of non-zero attributes within each input instance. As a result, the first input instance in the dataset will be the one with the largest number of non-zero attributes. The algorithm then calculates *current_sparse_perc* at lines 9–11, prior to entering the inner loop (lines 12–18). In this loop, the algorithm starts with a selection of the first input instance from the dataset. Recall that this is the input instance with the largest number of non-zero attributes. Then, "the least significant" attribute from that input instance is set to zero at lines 14–15. This attribute is selected as the one with the smallest absolute value. After eliminating the least significant attribute, *current_sparse_perc* is updated (line 16) and *input_instances* is re-sorted since the input instances within the dataset have been modified. This inner loop is repeated until a desired *target_sparse_perc* is reached. When the loop completes, a standard SVM training is invoked on the sparsified input dataset. As a result, a trained SVM will consist of sparse support vectors, eventually leading to improved performance of the hardware accelerator that implements such SVM. Please note that, similar to the pruning/sparsification of ANNs and DTs, *sparse_svm* is updated based on *temp_sparse_svm* only when a current sparsified predictive SVM model has the acceptable accuracy (beginning of the main loop, line 7).

Algorithm 6 Attribute_sparse_SVM_training

```
Input: input_instances
Output: sparse support vectors
 1: non_sparse_svm ← Run SVM training on input_instances
 2: non_sparse_acc ← Evaluate SVM using non_sparse_svm
 3: target_sparse_perc ← MIN_SPARS_PERC
 4: sparse_acc ← non_sparse_acc
 5: temp_sparse_svm ← non_sparse_svm
 6: while sparse_acc ≥ non_sparse_acc − 1% do
 7:   sparse_svm ← temp_sparse_svm
 8:   Sort input_instances descending wrt number of non-zero attributes
 9:   current_zv ← number of zero-valued attributes within input_instances
10:   total_att ← total number of attributes within input_instances
11:   current_sparse_perc ← current_zv/total_att
12:   while current_sparse_perc < target_sparse_perc do
13:     selected_instance ← input_instances[0]
14:     selected_attribute ← non-zero attribute from selected_instance with the smallest
        absolute value
15:     selected_attribute ← 0
16:     Update current_sparse_perc accordingly
17:     Update/Sort input_instances
18:   end while
19:   sparse_svm ← Run SVM training on sparsified input_instances
20:   sparse_acc ← Evaluate SVM using sparse_svm
21:   target_sparse_perc ← target_sparse_perc + SPARS_INC_PERC
22: end while
23: return sparse_svm
```

3. Sparse Reconfigurable Machine Learning Classifier

In order to benefit from sparse classifier models, a dedicated hardware accelerator, called the SRMLC (sparse reconfigurable machine learning classifier), is developed. A proposed architecture was inspired by a previously published accelerator [33], which can be reconfigured to support different classifier types, but cannot take advantage of sparse classifier models. The SRMLC, proposed in this study, is a novel architecture, which is designed and optimized to work with such sparse classifiers.

A proposed SRMLC architecture has a reduced instance processing latency and enhanced throughput when compared to the RMLC architecture published in [33]. The instance processing latency is reduced as a result of parallel and simultaneous DOT product calculation. Additionally, as a consequence of sparse data processing, outputs are calculated faster, since a significant number of DOT product multiplications are skipped, which results in a higher throughput.

The presented SRMLC architecture contains three main hardware modules, as shown in Figure 2.

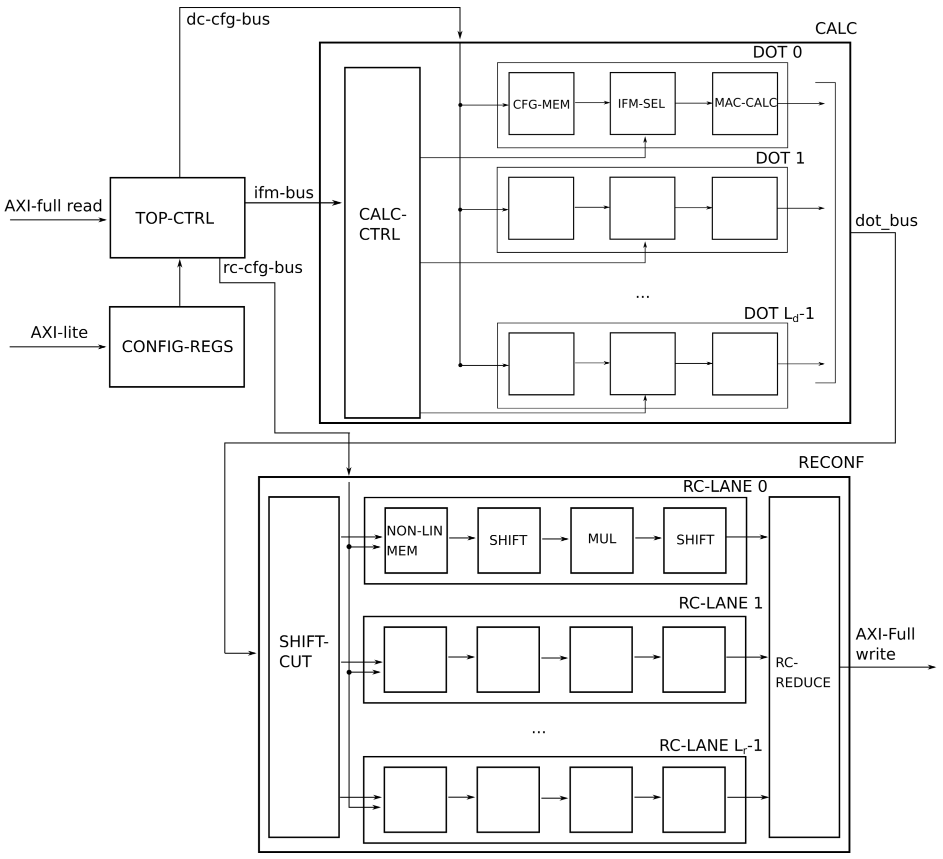

Figure 2. SRMLC top level architecture.

The first module (TOP-CTRL) controls the operation of the complete accelerator design. After reading configurations from the memory through the AXI-Full interface, TOP-CTRL configures other blocks via *dc-config-bus* and *rc-cfg-bus* interface busses, depending on the specific configuration.

The main CPU in the system can additionally configure the accelerator core through the AXI-lite interface. The configuration registers related to the control of the entire accelerator are located in the CONFIG-REGS module. One of the RMLC core features, especially important for understanding the section with the experimental results, is to fetch multiple input instances as a single memory block in a so-called batch mode. With this approach, the impact of DRAM memory latency is diminished and higher throughput is achieved.

For all types of supported predictive models, the second module (CALC) calculates DOT products, which are the core operations during the input instance classification. The CALC module receives the input instance from TOP-CTRL block. The output of this module is an array of calculated DOT products, which will be used by the RECONF module.

The third module (RECONF) is a reconfigurable block, which, according to the configuration, determines the type of currently active classifier. The RECONF module receives, as an input, an array of calculated DOT products from the CALC module and uses them to compute the classification results at the output.

The architecture of the CALC module is shown in Figure 3. The core of the CALC module is the array of DOT module instances, where each calculates a single DOT product between the input data vector and coefficient/weight vector, defined by the used predictive model (DT, SVM or ANN). As the SRMLC accelerator core can be parametrized, L_d, which is the number of DOT module instances within the array, is one of the most important system parameters. Each DOT module within the array can be configured through the configuration bus (*dc-config-bus*). The input data vector is transferred to DOT modules via the CALC module controller (CALC-CTRL). It is actually transferred in segments, as explained later in the text.

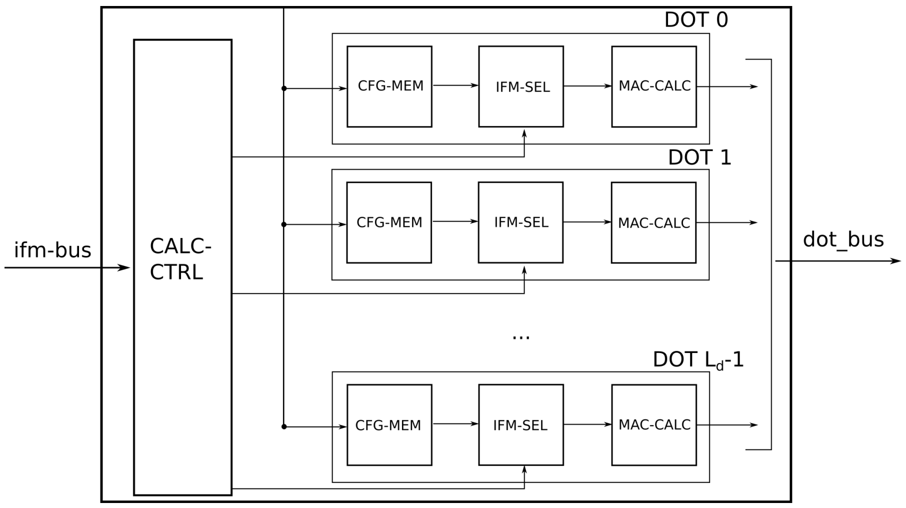

Figure 3. CALC architecture.

The architecture of the DOT module is shown in Figure 4. Please note that all submodules in the figure are shown conceptually, since the actual implementation contains more than one level of pipeline processing. The DOT module is the most important for the system performance and, in most configurations, it also utilizes the most hardware resources.

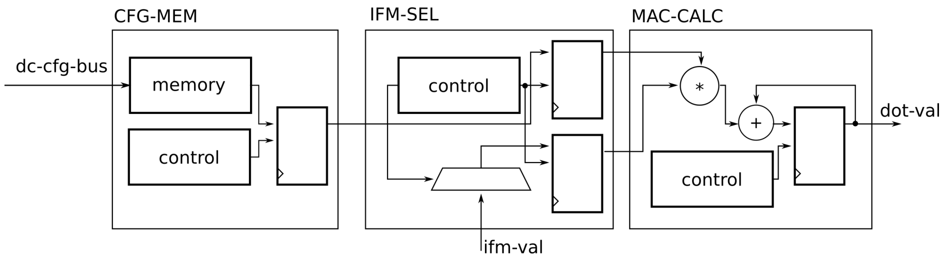

Figure 4. The DOT module architecture.

In order to support a sparse data processing, the configuration of each DOT module consists of classifier weights and increment values, stored within the internal DOT module memory (CFG-MEM). Classifier weights are coefficients that are used in DOT products for a given classifier type—weights in MLP ANNs, sparse hyperplane coefficients in DTs and sparse support vector coefficients in SVMs. Due to the compression of classifier weights, increment values are also stored in CFG-MEM to enable reconstruction of the actual positions within the original dense weight/coefficient vector. For example, assume that 10 pruned/sparsified classifier weights are 10, 0, 0, 25, 0, 0, 0, 0, 129, 38. For such a small compressed classifier weights segment, values stored in CFG-MEM would be:

- Classifier weights: 10, 25, 129, 38.
- Increment values: 0, 3, 5, 1.

Please note that due to a narrower dynamic range of increment values, they can be represented with fewer bits. Specifically, in the SRMLC architecture, weights are stored as 16-bit wide words and increments are stored as 4-bit wide nibbles, which results in the reduction in memory required for storing predictive model weights.

CFG-MEM memory is a main consumer of the memory resources within the DOT module. It is modeled so that the highest possible speed can be achieved with FPGA implementation. Additionally, the model is such that it can be easily mapped to the memory resources of FPGA devices, which is why this module is the main consumer of memory resources. The module also contains the control logic necessary to control multiple phases of a pipeline processing.

Figure 5 shows how the compressed classifier weights are stored in CFG-MEM memory. Within each memory location, a single non-zero classifier weight (denoted by V), and its corresponding increment value (denoted by I), is stored. CFG-MEM memory is divided into a number of sections. In the case of processing SVMs, each section holds the non-zero coefficients of a single support vector. In the case of processing MLP ANNs, each section holds the non-zero weights of one neuron. Finally, in the case of DT processing, one section holds the non-zero hyperplane coefficients of a single DT node. The section number is shown as the subscript of V and I in Figure 5, and the shaded locations indicate memory locations belonging to one section. The numbers shown below the V_i and I_i symbols indicate the index of a non-zero classifier weight and its corresponding increment value, within the current section. Please note that the width of the given section depends on the number of contained non-zero weights. Hence, it is not necessary that all sections within the CFG-MEM have the same width.

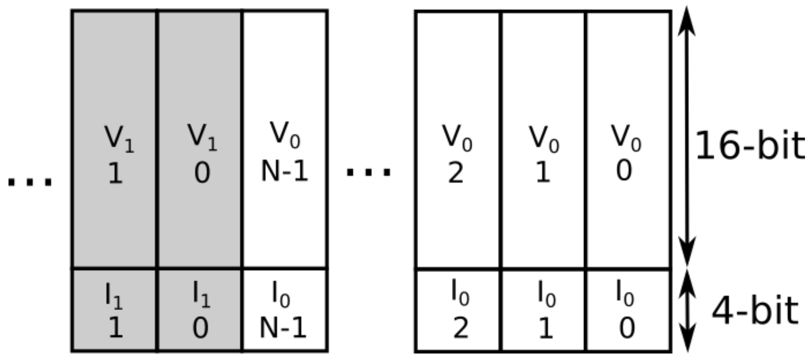

Figure 5. Layout of CFG-MEM memory.

The input vector values, obtained from the other input of the DOT module, are sent directly to the selection module (IFM-SEL). The IFM-SEL module, with respect to the increment values from CFG-MEM, determines which values from the input vector are required for the calculation, while remaining vector values are skipped in order to avoid multiplications with zero-valued operands.

The main component of the IFM-SEL module is the multiplexer, which selects the correct input vector value as an operand for multiplication. A multiplexer requires LUTs for implementation, so this module is the main consumer of LUTs in the FPGA implementation.

The outputs of the IFM-SEL module are used by the MAC-CALC module, which performs multiplication operations and accumulates the results in order to calculate the DOT product. The main goal when designing this module was to implement it within a single DSP block of the FPGA device, with the smallest possible utilized logic overhead. Therefore, this module is the main consumer of DSP blocks during FPGA implementation of the accelerator core.

Each of the submodules within the DOT module is designed to consume the critical FPGA resources (DSP, memory and random logic-LUTs) in a balanced way. With such an approach, the entire architecture scaled well, when increasing the available hardware resources of the FPGA device, since all resources were consumed equally. This is in stark contrast to the RMLC architecture which failed to balance the utilization of memory resources and, therefore, scaled poorly.

Figure 6 shows the architecture of the RECONF module. The SHIFT-CUT block receives the array of calculated DOT roduct values and stores them internally, allowing

DOT modules to process new input vector values, which results in a specific course grained parallelism within the SRMLC architecture. Buffered values are stored within the shift register, large enough to store L_d DOT product values. The lowest L_r out of those L_d values are passed to computing lanes (RC-LANE), where L_r is the number of used computing lanes within the design and is also an additional SRMLC configuration parameter. After distributing L_r values to appropriate computing lanes, the SHIFT-CUT block will shift contained DOT products in order to prepare the next batch of L_r points for processing.

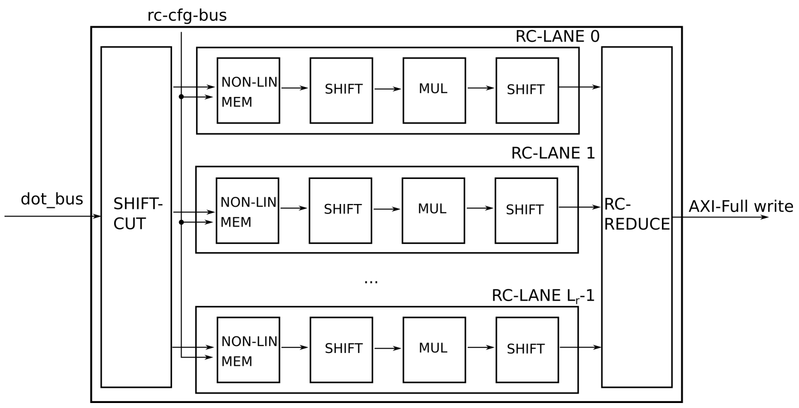

Figure 6. The RECONF architecture.

Each RC-LANE can perform multiple functions, depending on the configuration of the SRMLC accelerator. Figure 7 shows that the RC-LANE module contains several pipeline stages, where each is configurable via the configuration bus. Depending on the current configuration, the RC-LANE can perform non-linear function evaluation, multiplication or shift contained value in order to adjust its fixed point number format. Each of these stages can be skipped, if not needed for the calculation.

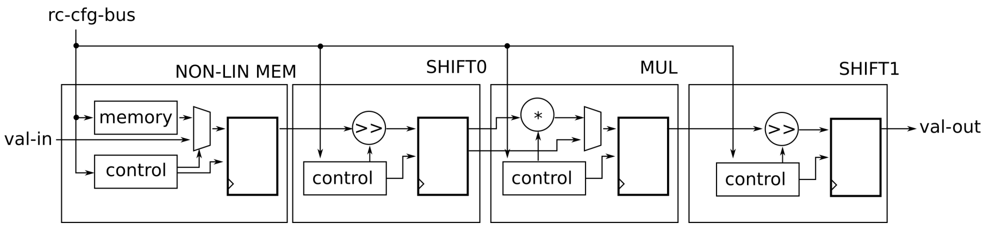

Figure 7. The RC-LANE architecture.

The reduction submodule from the RECONF (RC-REDUCE) receives all inputs from the RC-LANEs and performs different post-processing steps, depending on the current configuration. When the final stages of SMVs are processed, it will accumulate received values. In the case of MLP classifier processing, the RC-REDUCE module will pass calculated values from neurons or accumulate values depending on the configuration. At the end, the RC-REDUCE module is in charge of the selection process within decision trees, selecting the right or the left subtree, based on a previously calculated DOT product.

Next, we will show how the RC-LANE modules are configured in order to process SVM, DT or MLP ANN classifier types. Please notice that Figures 8–10 show the required configuration of a single RECONF block for simplification. In the examples below, active blocks within the RECONF module are shaded.

Figure 8 presents the configuration of one RECONF module in the case of DT processing. The figure also shows an example DT, as well as the way the DT is mapped to a corresponding configuration, stored inside the RC-REDUCE module. In this configuration, each DOT module computes a DOT product of the input instance and assigned DT node hyperplane coefficients and outputs the result for the RECONF module. The selected DT nodes' hyperplane coefficients are stored inside the CFG-MEM module, labeled with

$nd0, nd1\ldots$ in Figure 8. In this configuration, RC-LANE modules only adjust the number formats of computed DOT products. The RC-REDUCE module takes computed DOT products and iteratively compares them with the appropriate threshold values, t_i, while traversing a DT from the root node until a leaf node is reached.

In the example DT from Figure 8, a classification of input instance will go as follows. In the first step, the DOT product $d0$ will be compared to the threshold value $t0$. Depending on the outcome, the RC-REDUCE module will select which DOT product to use for the following comparison. For example, let us assume that the result of the $d0 > t0$ comparison is such that we should visit the DT node $n1$ next. This means that the RC-REDUCE module will next compare the dot product $d1$ with threshold $t1$. This procedure is repeated until RC-REDUCE reaches a leaf node. This completes the current input instance classification, and the computed input instance class membership value is transmitted through the val_out output port. For example, if the result of the $d1 > t1$ test leads to reaching the $c0$ leaf node, the classification of the input instance is completed and the RC-REDUCE module will output the $c0$ value on the val_out output port.

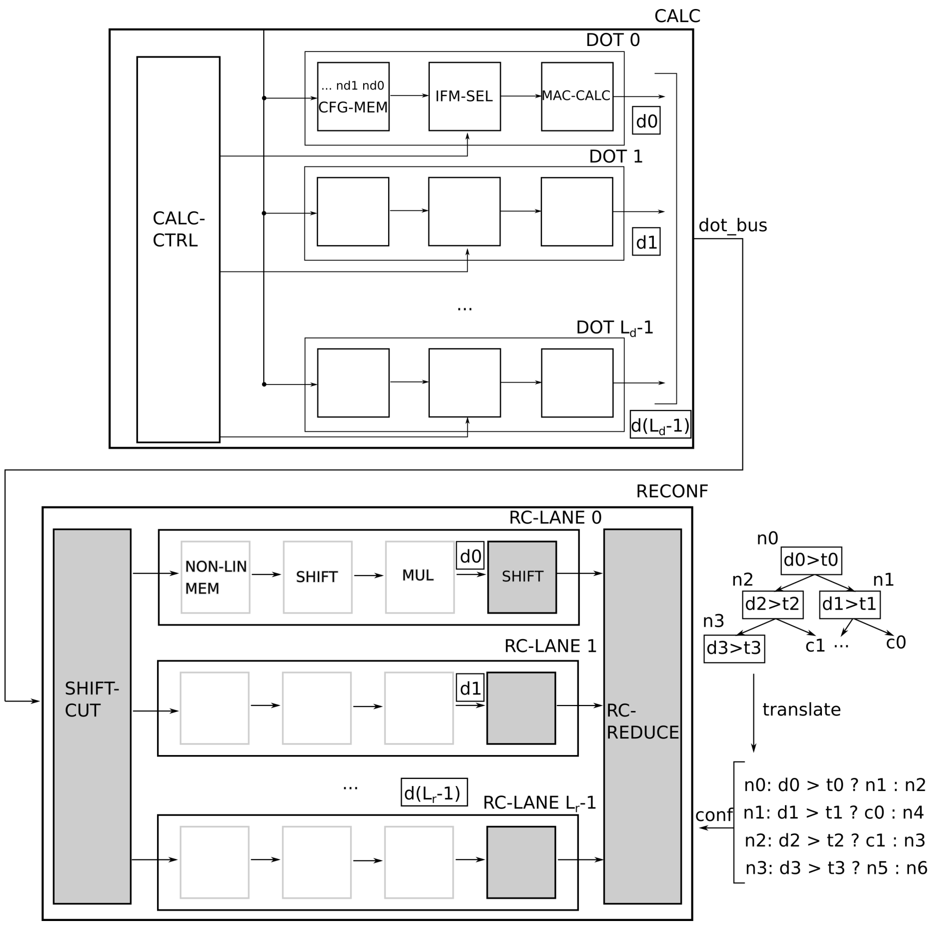

Figure 8. The RECONF architecture—DT configuration.

Figure 9 shows one non-linear SVM configuration. In this configuration, each DOT module calculates DOT product of the input instance and the subset of support vectors, associated with the corresponding DOT module. The subset of support vectors is stored within the CFG-MEM module, labeled with $sv0, sv1\ldots$ in the Figure 9. The calculated DOT values are passed from the CALC module to the RECONF module. In the Figure 9 these values are designated as $d_i, i = 0, 1, \ldots, L_r - 1$. When operating in SVM mode, RC-LANE modules use non-linear memory to calculate the kernel function, specified by the user. The values after calculating the non-linear function are labeled with $a_i, i = 0, 1, \ldots, L_r - 1$. The MUL submodule is used to multiply non-linear values with Lagrange multipliers,

$\alpha_i, i = 0, 1, \ldots, L_r - 1$. SHIFT modules are used to adjust number formats, in order to minimize a quantization loss during the calculation. The values $k_i, i = 0, 1, \ldots, L_r - 1$, obtained after multiplication with Lagrange multipliers, are sent to the RC-REDUCE module, which accumulates them in this operating mode. With such configuration, accumulated sum of values k_i is obtained. Additionally, this module adds offset value to the accumulated sum and compares it with zero to obtain the final SVM classification result of the current input instance. Lagrange multipliers, kernel function samples and the offset are an integral part of the SVM architecture configuration and they are set by the TOP-CTRL module.

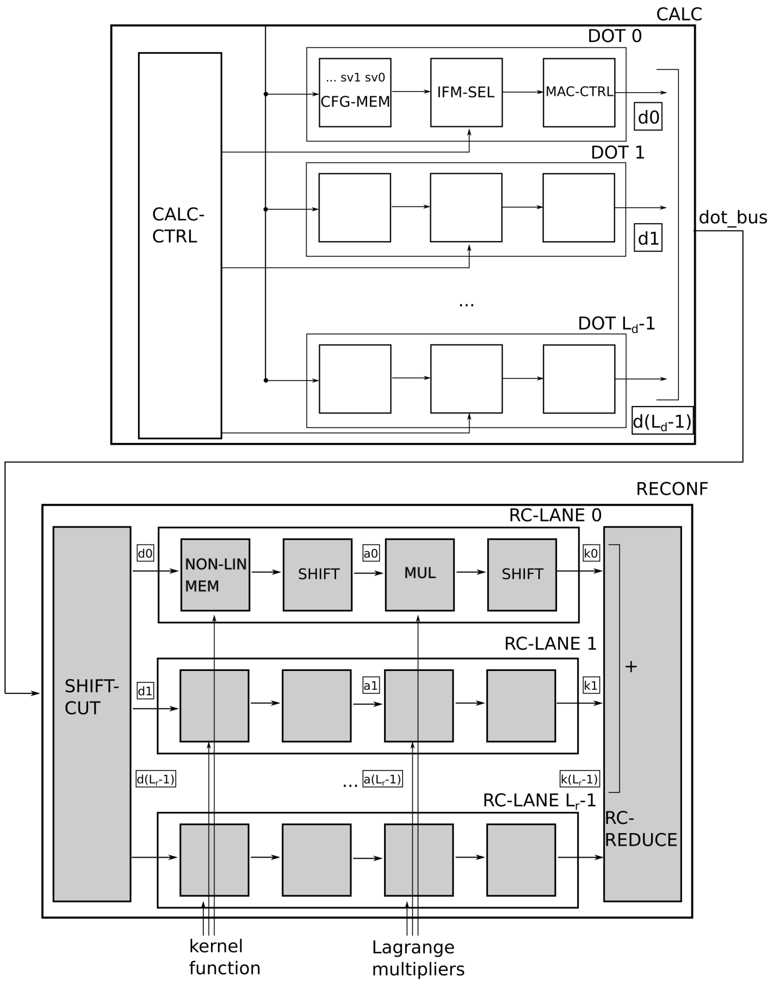

Figure 9. The RECONF architecture—SVM configuration.

Figure 10 shows the SRMLC configuration processing the sparse MLP classifier with a single hidden layer. As shown in Figure 10, when the architecture is configured to work as MLP, each DOT module calculates DOT products of the input instances and neuron weights for each neuron from a subset associated with a corresponding DOT module. The subset of neuron weights is stored inside the CFG-MEM module, labeled with $n0, n1 \ldots$ in Figure 10. The calculated neuron output values are forwarded to the RECONF module, marked with $d_i, i = 0, 1, \ldots, L_r - 1$ in the Figure 10. RC-LANE modules receive these values and pass them to non-linear memory, where the samples of specified activation function are stored. In this way, the output values $a_i, i = 0, 1, \ldots, L_r - 1$ are obtained after applying the activation function, as shown in Figure 10. These values are sent to the RC-REDUCE module which only forwards them to the output, without additional processing.

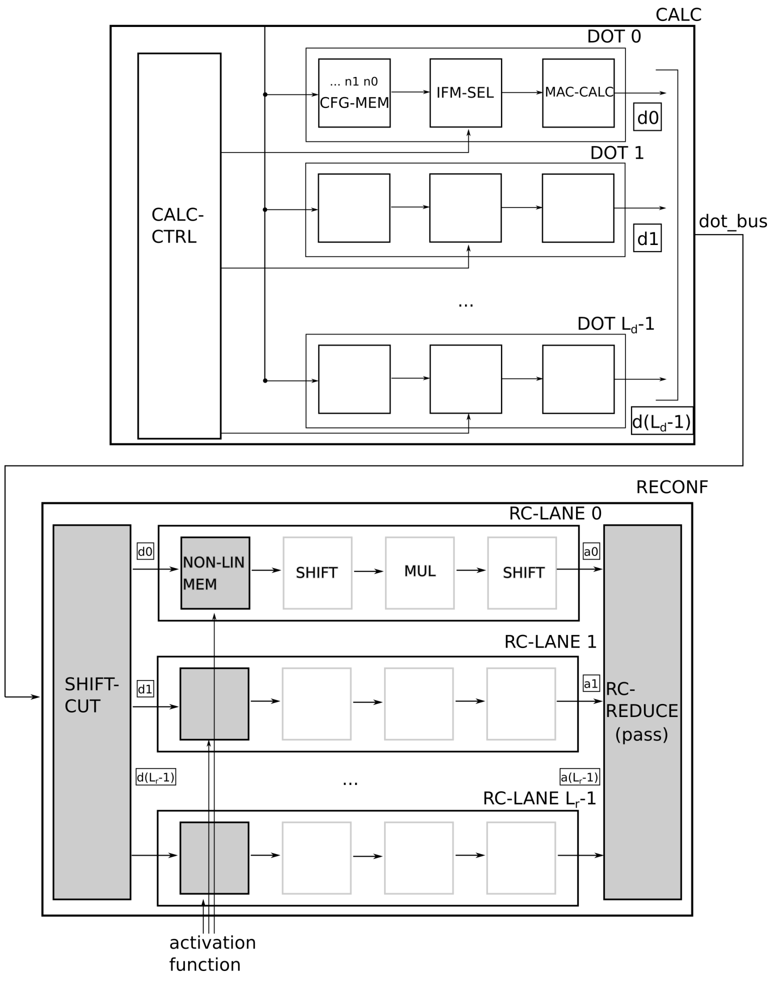

Figure 10. The RECONF architecture—MLP layer configuration.

Analytical Model of Instance Processing Throughput of SRMLC Architecture

In this section, we will provide an analytical model of the SRMLC architecture that shows the number of cycles required to classify a single input instance. The analysis will focus on the instance processing throughput of the architecture, so the latency required to deliver the data to the computing module will be neglected. The following description of the analytical model corresponds to the operation of the SRMLC architecture in a batch mode.

The architecture contains two main computing modules: the CALC module and the RECONF module. These two modules work in parallel, organized as a two-stage pipeline. Please note that both the CALC and RECONF modules internally also use pipelining. For different configurations of the SRMLC architecture, one of these modules will be a processing bottleneck during the instance classification.

Let N_{proc} be the number of tree nodes, the number of support vectors or the number of neurons, depending on the architecture configuration (DT, SVM or MLP). Let L_d be the number of DOT modules available in the current configuration of the SRMLC architecture. When $N_{proc} \leq L_d$, N_{proc} DOT modules will be active in parallel during the current input instance processing in a single run. However, if $N_{proc} > L_d$, the current input instance will be processed in $\lfloor N_{proc}/L_d \rfloor$ iterations during which all L_d DOT modules are engaged and the last iteration in which the number of active DOT modules equals:

$$D = N_{proc} \bmod L_d \tag{8}$$

Let L_r be the number of RC-LANE modules within the RECONF module, and let N_r be the number of cycles necessary to perform a single instance classification by the RECONF module. The architecture should always be configured so that relation $L_d > L_r$ holds. Then:

$$N_r = \begin{cases} \lfloor N_{proc}/L_d \rfloor \lceil L_d/L_r \rceil + \lceil D/L_r \rceil, N_{proc} > L_d \\ \lceil D/L_r \rceil, N_{proc} <= L_d \end{cases} \tag{9}$$

Let N_{nzv} be the number of non-zero weights and let N_c be the number of cycles required by the DOT module to compute all DOT products using the current instance. Then, due to a three cycles delay of pipeline processing:

$$N_c = (N_{nzv} + 3) \lceil N_{proc}/L_d \rceil \tag{10}$$

If N_a is the input instance attributes number, and P is the factor with which the classification models are pruned (percentage of weights that will be set to zero), then,

$$N_c = (N_a(1 - P) + 3) \lceil N_{proc}/L_d \rceil \tag{11}$$

Since the CALC and RECONF modules are connected in a pipeline, the number of cycles, N, required for the SRMLC architecture to classify a single input instance, equals

$$N = max(N_c, N_r) \tag{12}$$

Throughput with which the SRMLC architecture can process input instances can be calculated as:

$$Throu = \frac{f}{N} = \frac{f}{max(N_c, N_r)} \tag{13}$$

where f is the operating frequency of the SRMLC accelerator.

From the above analysis, it can be concluded that in order to obtain higher throughput, by increasing the pruning factor, the architecture needs to be configured so that N_c becomes a dominant factor in Equation (12).

For fixed architecture parameters (L_d and L_r), the values of N_r and N_c change as a staircase function when N_{proc} is changed. N_r is increased by 1 whenever N_{proc} is increased by L_r. N_c is increased by $(1 - P)N_a$, when N_{proc} is increased by L_d. As a result, the value of N_r increases more frequently in smaller steps. If we want the relation $N_c > N_r$ to hold, then the worst case scenario happens if N_{proc} reaches the value which causes the update of N_c. That is the case when $D = L_d$, so:

$$\lceil D/L_r \rceil = \lceil L_d/L_r \rceil \tag{14}$$

In that case, Equation (9) becomes:

$$N_r = \lceil N_{proc}/L_d \rceil \lceil L_d/L_r \rceil \tag{15}$$

From Equations (11) and (15), we can derive the condition that the CALC module is the real processing bottleneck, meaning that the sparsification/pruning factor has an impact on the processing throughput of the accelerator.

$$N_c > N_r \tag{16}$$

$$(N_a(1 - P) + 3) \lceil N_{proc}/L_d \rceil > \lceil L_d/L_r \rceil \lceil N_{proc}/L_d \rceil \tag{17}$$

$$N_a(1 - P) + 3 > \lceil L_d/L_r \rceil \tag{18}$$

$$P < 1 - \frac{\lceil L_d/L_r \rceil - 3}{N_a} \tag{19}$$

If Equation (19) holds, the sparsification/pruning factor P increases throughput for the given architecture parameters (L_d and L_r) and the problem instance (N_a). In the experimental results (Section 4), the SRMLC architecture is configured so that Equation (19) holds for almost all cases.

4. Experimental Results

In order to benchmark our approach, we have conveyed several experiments and the results are presented in this section. In the first subsection, we show the result of ANN pruning by using our Algorithm 1 and the results of DT and SVM sparsification by using Algorithms 2 and 6. In the second subsection, we compare our work with previously published work [33], while in the third subsection we present the comparison with embedded software implementations of ML predictive models.

4.1. Experiments for Pruning ANNs and Sparsification of DTs and SVMs

In order to be able to benchmark the performance of Algorithms 1, 2 and 6, UCI machine learning repository datasets from Table 1 were used. Short names shown in Table 1 correspond to the names used in Figures 11–22.

Table 1. Experimental datasets' basic characteristics.

Dataset Name	Short Name	Attributes	Instances
Australian—Statlog	australian	14	690
Pima Indians Diabetes	diabetes	8	768
Glass Identification	glass	9	214
Heart—Statlog	heart	13	270
Heart disease Cleveland	heart-disease	13	303
Hepatitis	hepatitis	19	155
Ionosphere	ionosphere	34	351
Page blocks	page-blocks	10	5473
Sonar	sonar	60	208
Statlog (Vehicle Silhouettes)	vehicle	18	846
Waveform 21	waveform21	21	5000
Waveform 40	waveform40	40	5000
Wisconsin Breast Cancer—WDBC	wdbc	10	683
Wine Recognition	wine	13	178
Zoo	zoo	17	101

The Tensorflow framework [64] has been used for evaluating Algorithm 1. The instances from the UCI machine learning repository with missing values are removed from datasets, while all results reported below are the averages of five ten-fold cross-validation experiments. This assumes that the original dataset D is divided into 10 non-overlapping subsets, $D_1, D_2, \ldots, D_{10}$, which consist of uniformly selected instances from D. During each cross-validation iteration, ANN is built by using the instances from the $D \setminus D_i$ set and tested on D_i the set ($i = 1, \ldots, 10$). By repeating this procedure five times, 50 ANNs are constructed in total for each dataset. Then, the average classification accuracy is calculated as the percentage of input instances, which are correctly classified. In order to obtain the ANN pruning curve, which shows how the accuracy of a pruned ANN drops as the pruning factor increases, we slightly modified Algorithm 1. The Algorithm 1 presented in Section 2 exits the main loop as soon as the accuracy of a pruned ANN drops more than 1% below the absolute accuracy of non-pruned ANN. Instead of this upper limit, for experimental purposes, we have swept *current_pruning_factor* from MIN_PRUN_FACT to MAX_PRUN_FACT. In our experiments, a MIN_PRUN_FACT parameter was set to 5% and a MAX_PRUN_FACT parameter was set to 99%.

Figures 11–14 show the results of training and pruning the ANN with a single hidden layer which has 64 neurons on 15 datasets from the UCI repository. The presented charts show the impact of a pruning factor (X-axis) on classification accuracy (Y-axis), where the name of the dataset is given above the chart. While the blue line shows the accuracy of the pruned ANN, which depends on a pruning factor and eventually drops, a red dashed line shows the lower limit of a pruning tolerance, since it is drawn 1% below the value of the non-pruned ANN absolute accuracy. Figures 11–14 show that $max\_pruning\_factor$ above 80% can be used with the accepted accuracy drop on 13 out of 15 datasets, which means that, most of the time, more than 80% of the ANN weights can be set to zero, without any loss in accuracy. During the training and pruning, several ANN architectures were used, containing multiple hidden layers and a varying number of neurons per hidden layer. Neither increasing number of hidden layers nor increasing number of neurons within the hidden layer above 64, resulted in better accuracy on the chosen datasets.

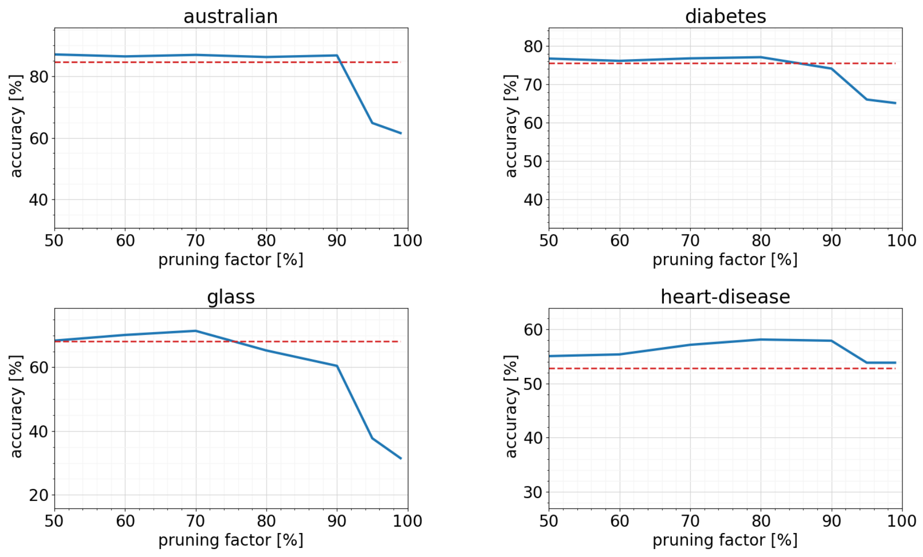

Figure 11. Results of pruning MLP ANN structure on Australian, diabetes, glass and heart-disease datasets.

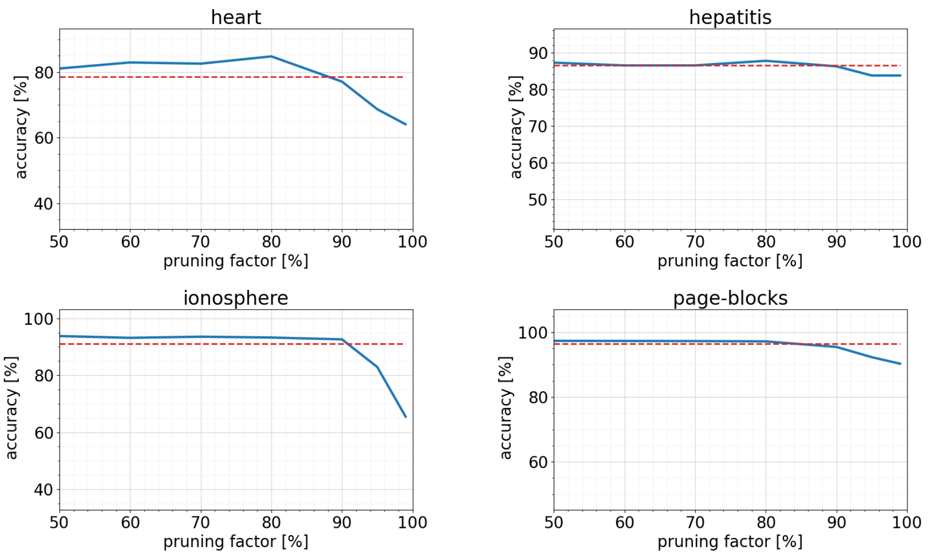

Figure 12. Results of pruning MLP ANN structure on heart, hepatitis, ionosphere and page-blocks datasets.

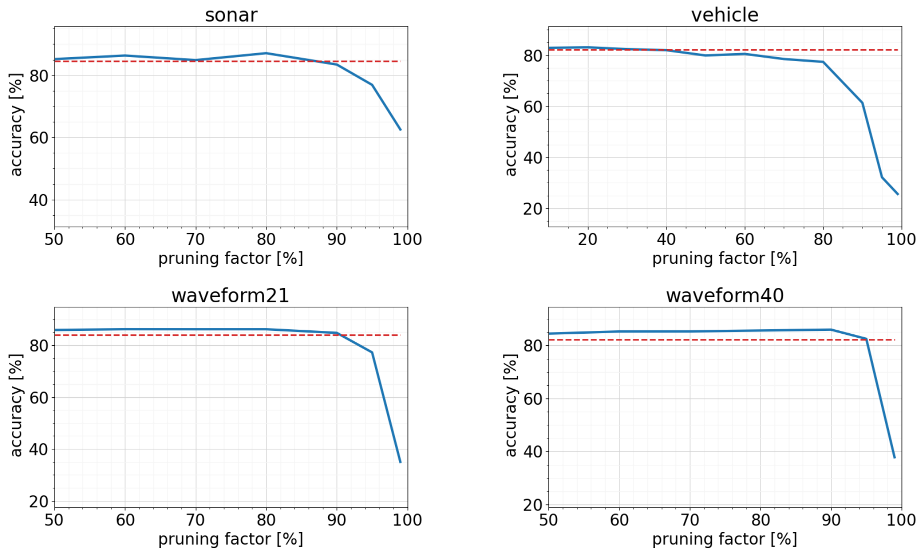

Figure 13. Results of pruning MLP ANN structure on sonar, vehicle, waveform21 and waveform40 datasets.

Figures 15–18 show results of the sparse oblique DT induction, performed on the same subset of datasets from the UCI repository which were used for the training and pruning of ANNs. Once again, we slightly modified Algorithm 2 in order to obtain the sparsification curve. Hence, instead of exiting the main loop once the accuracy of sparse DT drops below the tolerated lower accuracy limit, we sweep $current\_spars\_factor$ between MIN_SPARS_FACT and MAX_SPARS_FACT in our experiment set to 10% and 90%, respectively. In Figures 15–18, a blue line shows the accuracy of a sparse DT, which decreases as a sparsification factor increases (similar to pruning of ANNs). A red dashed line shows a lower limit for accepted accuracy, calculated as an absolute 1% drop from the non-sparse DT model accuracy. Although sparsification factors are lower compared to ANNs, the results show that a significant percentage of DT coefficients can be set to zero during induction, with the acceptable accuracy drop. However, opposite from ANN pruning, during DT sparsification, high percentages of sparsification factor could not be obtained for most of the datasets, since iterative evolutionary algorithm could not converge in those scenarios. This is the reason why, for some datasets, MAX_SPARS_FACT of 90% could not be reached in Figures 15–18. Similar to the experimental results for MLP ANNs, all reported results are calculated as averages of five ten-fold cross-validation experiments.

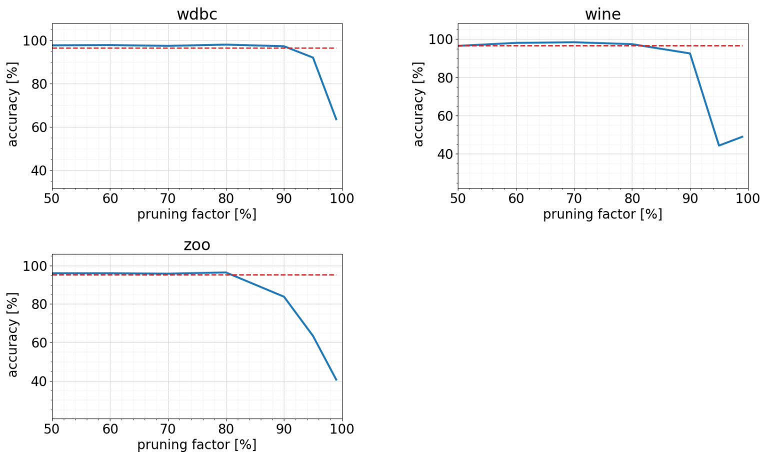

Figure 14. Results of pruning MLP ANN structure on wdbc, wine and zoo datasets.

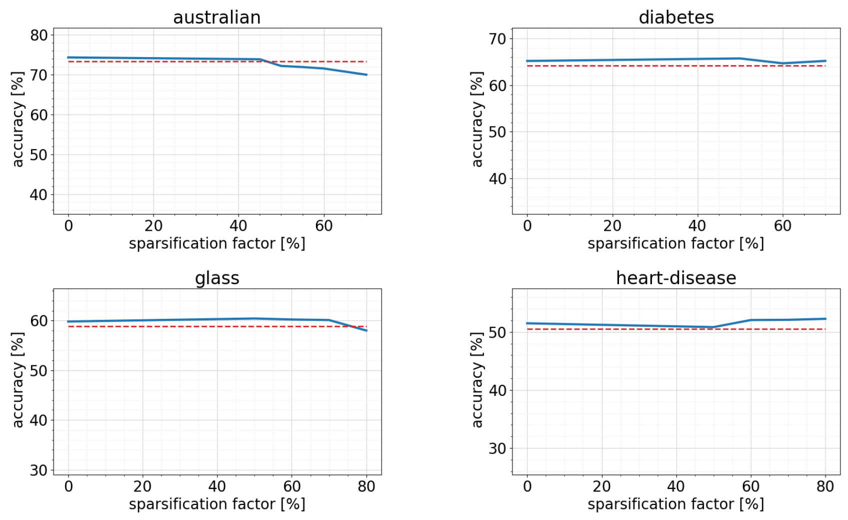

Figure 15. Results of pruning DT structure on Australian, diabetes, glass and heart-disease datasets.

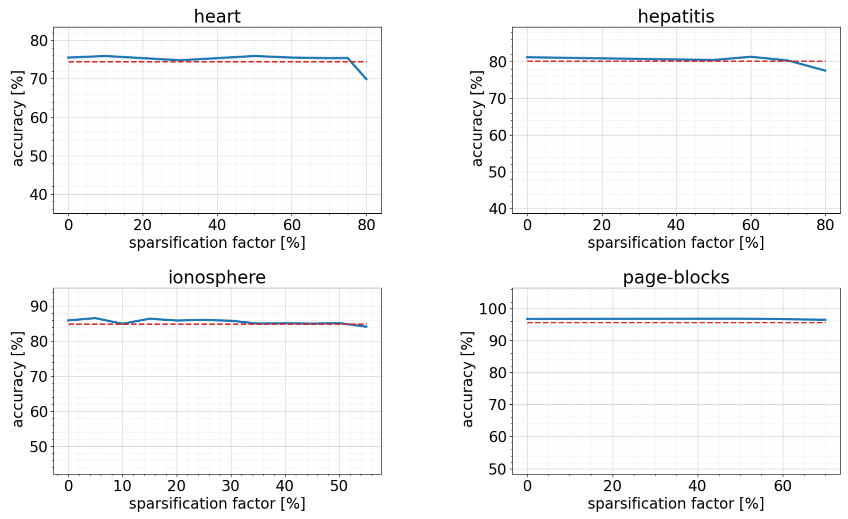

Figure 16. Results of pruning DT structure on heart, hepatitis, ionosphere and page-blocks datasets.

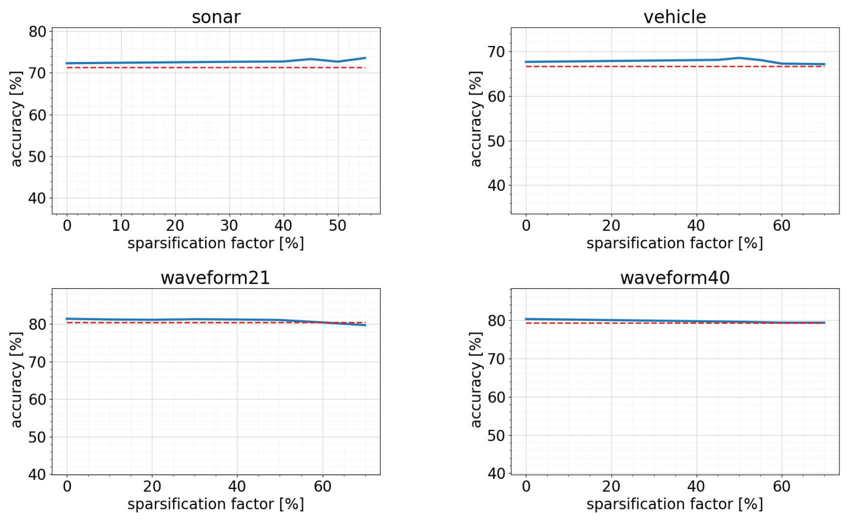

Figure 17. Results of pruning DT structure on sonar, vehicle, waveform21 and waveform40 datasets.

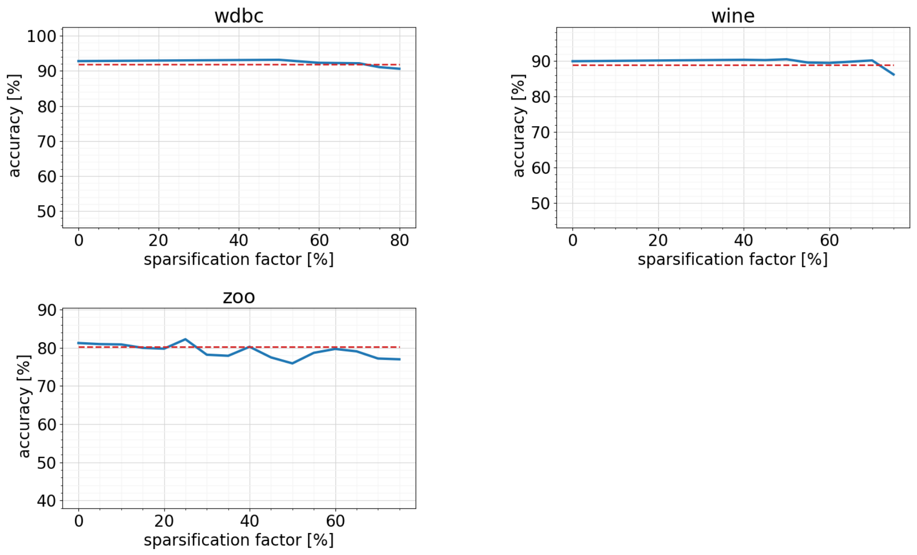

Figure 18. Results of pruning DT structure on wdbc, wine and zoo datasets.

Figures 19–22 show results of the attribute sparse SVM training, performed on the same 15 UCI datasets, which were used for the training and pruning of the ANNs and the sparse induction of DTs. As with other two algorithms, we had to modify Algorithm 6 in order to be able to draw the sparsification curve for SVM predictive models. Hence, instead of using the main loop exiting condition at line 6, we swept *target_spars_perc* between *MIN_SPARS_PERC* set to 5% and *MAX_SPARS_PERC* set to 95%. Similar to previous charts, a blue line was used to present the accuracy of an attribute sparse SVM for a given dataset, decreasing as the sparsification factor increases. A dashed red line shows the lower limit for tolerated accuracy and it is calculated after subtracting the 1% absolute accuracy drop from the non-sparse SVM model accuracy (accepting absolute 1% for the tolerated accuracy drop). As results show, at least 60% of the sparsification factor can be achieved for the majority of the used datasets. Similar to previously presented results for ANN pruning and DT sparsification, results shown here are calculated as averages of five ten-fold cross-validation experiments.

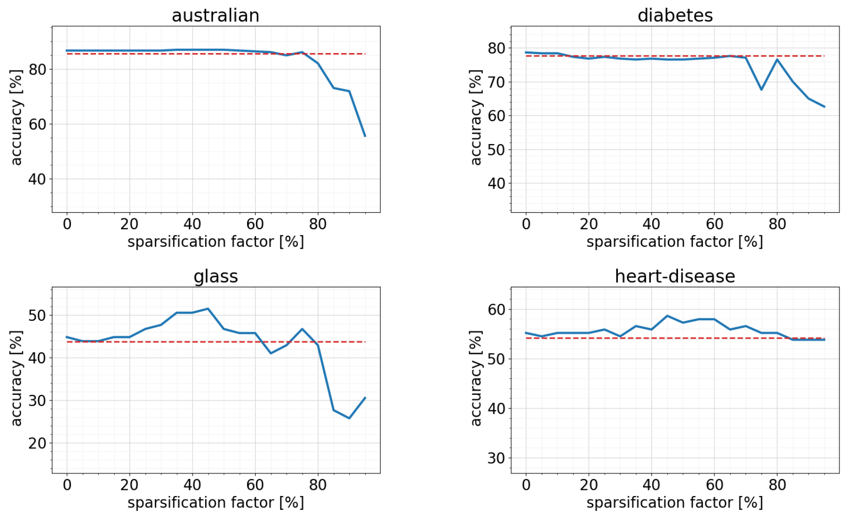

Figure 19. Results of pruning SVM structure on Australian, diabetes, glass and heart-disease datasets.

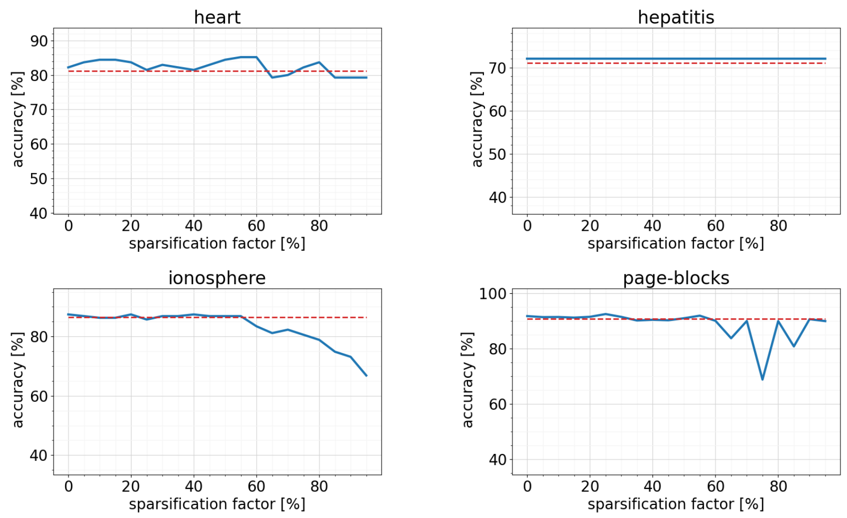

Figure 20. Results of pruning SVM structure on heart, hepatitis, ionosphere and page-blocks datasets.

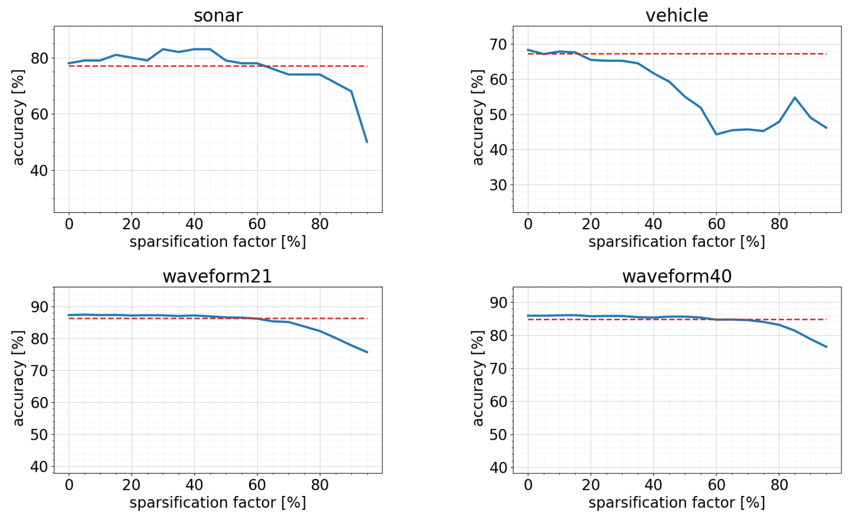

Figure 21. Results of pruning SVM structure on sonar, vehicle, waveform21 and waveform40 datasets.

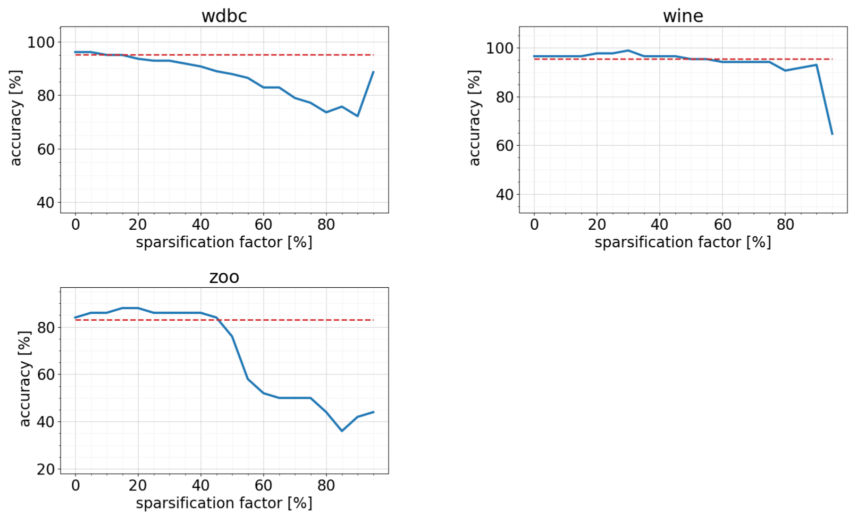

Figure 22. Results of pruning SVM structure on wdbc, wine and zoo datasets.

Predictive model evaluations are presented in Tables 2–4. For the dataset given in the first column, the achieved sparsification/pruning factor is shown in column 2. The accuracy of non-sparsified and sparsified models is presented in columns 3 and 4, respectively, while the last two columns show corresponding size, expressed as the number of model parameters (weights/coefficients).

From Tables 2–4, one can conclude that the ANN predictive model performs better than the other two with the average accuracy of 84.58% and the average maximum achieved pruning factor of 81.33%. For the SVM predictive model, an average accuracy on the given datasets is 79.64%, with an average maximum achieved sparsification factor of 57.67%. Finally, the DT classifier can be sparsified 61.71% on average, with a slightly worse average accuracy of 77.06%. Columns 3 and 4 from Tables 2, 3 and 4 also show that, for all three classifier types, even heavily sparsified predictive models can score a better predicting accuracy, when compared to a non-sparsified model, for some datasets.

Table 2. Sparse DT predictive model characteristics.

Dataset	Spars. [%]	Acc. [%]	Sparse Acc. [%]	Size	Sparse Size
australian	40	74.33	73.87	454.2	251.34
diabetes	66.67	65.17	65.18	262.44	343.74
glass	70	59.88	60.1	117.2	94.54
heart	71.43	75.52	75.41	87.08	87.6
heart-disease	78.57	51.5	52.28	195.44	97.4
hepatitis	70	81.12	80.25	26.4	21.2
ionosphere	48.57	85.98	85.15	164.5	246.56
page-blocks	63.64	96.75	96.52	262.46	334.04
sonar	54.1	72.33	73.58	414.8	351.56
vehicle	68.42	67.66	67.16	571.14	447
waveform21	59.09	81.43	80.47	1155.88	1593.76
waveform40	68.29	80.27	79.3	1808.1	1391.02
wdbc	63.64	92.78	92.15	55.44	45
wine	64.29	89.9	90.13	36.4	42.8
zoo	38.89	81.25	80.27	135.36	96.34

Table 3. Sparse SVM predictive model characteristics.

Dataset	Spars. [%]	Acc. [%]	Sparse Acc. [%]	Size	Sparse Size
australian	75	86.67	86.09	3122	1395
diabetes	65	78.65	77.66	3768	1223
glass	75	44.76	46.67	1503	430
heart	80	82.22	83.7	1612	353
heart-disease	80	55.17	55.17	2509	483
hepatitis	90	72	72	1292	82
ionosphere	55	87.43	86.86	5780	3319
page-blocks	55	91.66	91.81	8660	3385
sonar	60	78	78	10,200	5340
vehicle	15	68.33	67.62	11,700	11,963
waveform21	55	87.32	86.52	53,088	30,297
waveform40	55	85.92	85.36	113,720	66,263
wdbc	15	96.07	95.1	1730	1900
wine	45	96.47	96.47	1105	770
zoo	45	84	84	1003	777

Table 4. Sparse ANN predictive model characteristics.

Dataset	Spars. [%]	Acc. [%]	Sparse Acc. [%]	Size	Sparse Size
australian	90	85.74	86.72	1024	128
diabetes	80	76.83	77.08	640	160
glass	70	68.83	71.42	1024	384
heart	80	79.78	84.81	960	240
heart-disease	95	53.68	53.86	1152	72
hepatitis	80	88	87.75	1344	335
ionosphere	90	91.56	92.64	2304	288
page-blocks	80	97.44	97.18	960	240
sonar	80	85.77	87.1	3968	992
vehicle	40	82.07	81.96	1408	1055
waveform21	90	84.69	84.74	1536	192
waveform40	95	82.94	82.52	2752	172
wdbc	90	97.33	97.22	768	95
wine	80	97.75	97.31	1024	255
zoo	80	96.22	96.44	1536	384

4.2. Comparison with RMLC Architecture

In order to compare the resource utilization and scalability of the RMLC and SRMLC architectures, both of them were implemented by using FPGA technology. Xilinx Vivado Design Suite [65] was used for the implementation of both architectures, targeting the ZU9 FPGA device, with default values of settings for the synthesis and implementation. In order to measure an achievable instance processing performance, Zynq Ultrascale+ MPSoC ZCU102 Evaluation Board [66] was used as the test platform for conducting experiments. Please note that the ZCU102 evaluation board was used to conduct the experiments due to its availability. However, as results presented in Table 5 illustrate, smaller instances of the SRMLC architecture can easily fit even the entry level FPGA devices.

Table 5. SRMLC FPGA implementation results.

DOT#	Freq [Mhz]	LUT	BRAM	DSP	Power [W]
32	250	11,852	36.5	40	0.37
64	250	21,164	69	72	0.687
96	250	30,571	101.5	104	1.01
128	225	39,773	134	136	1.173
160	200	48,773	166.5	168	1.28
192	200	58,106	199	200	1.539
224	150	66,804	231.5	232	1.39
256	150	76,168	264	264	1.496
288	150	85,445	300	296	1.679
320	125	94,143	333	328	1.551
352	125	106,822	366	360	1.66
384	125	112,852	399	392	1.875
416	125	126,081	432	424	2.15
448	125	136,019	464.5	456	2.279

Table 5 presents a power consumption dependency from the number of used multiply-accumulate blocks within the design. The first column (DOT#) shows the number of used multiply-accumulate blocks, the second one shows the operating frequency after the synthesis and the following three columns show the usage of LUTs, BRAMs and DSPs, respectively. The last column (power) shows the estimated power consumption of the implementation. From Table 5, it can be seen that the SRMLC architecture is highly scalable. The smallest SRMLC instances (with 32 and 64 DOT modules) can be fitted to the cost-effective FPGA devices from the Spartan-7, Kintex-7 and Zynq-7000 families. If more performance is required, larger SRMLC instances can be used.

For comparison purposes, Table 6 shows resource utilization after the implementation of RMLC FPGA [33]. The first column (RB#) shows the number of used reconfigurable blocks in the RMLC architecture, which have a similar function to DOT modules in the SRMLC architecture. As it can be seen from Tables 5 and 6, the SRMLC architecture, presented in this study, provides significantly better scalability when compared to the RMLC architecture. In the SRMLC, a single DOT unit requires only 1 DSP block, 1 BRAM and 300 LUTs, which ensures an optimal utilization of available FPGA resources. This is not the case for the architecture proposed in [33], where the exceeding utilization of BRAM blocks is a limiting factor for efficient scalability of the RMLC architecture, clearly seen from Table 6, where the largest instance of the RMLC architecture that can fit inside the ZU9 FPGA device only has 96 RB units. This is significantly less than the largest SRMLC architecture that fits inside ZU9 FPGA and contains 448 DOT units, as can be seen from Table 5. This improved scalability on FPGA platforms was one of the main architecture design goals during the development of the SRMLC.

Table 6. RMLC FPGA implementation results.

RB#	Freq [Mhz]	LUT	BRAM	DSP	Power [W]
32	250	12,219	260.5	32	1.549
64	250	20,382	516.5	64	2.282
96	250	28,843	772	96	2.968

The ASIC implementation layout is shown in Figure 23, while the summary is provided in Table 7 for comparison purposes. Our block-level implementation was developed in the Genus [67] and Innovus [68] Cadence tools, using 40 nm TSMC process standard libraries.

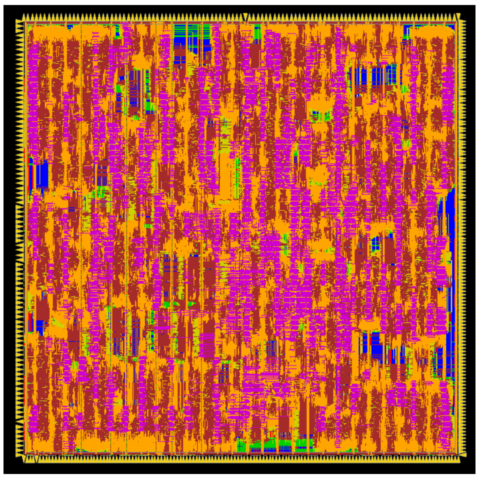

Figure 23. SRMLC ASIC layout.

Table 7. SRMLC ASIC TSMC 40 nm implementation results.

DOT#	Freq [Mhz]	Gate_COUNT	Area [mm^2]	Power [mW]
64	300	1,947,837	7.833	868.066

4.2.1. Comparison of Scalability

A scalability comparison of two architectures is shown in Figure 24. All architecture configurations are implemented for the ZCU102 development board.

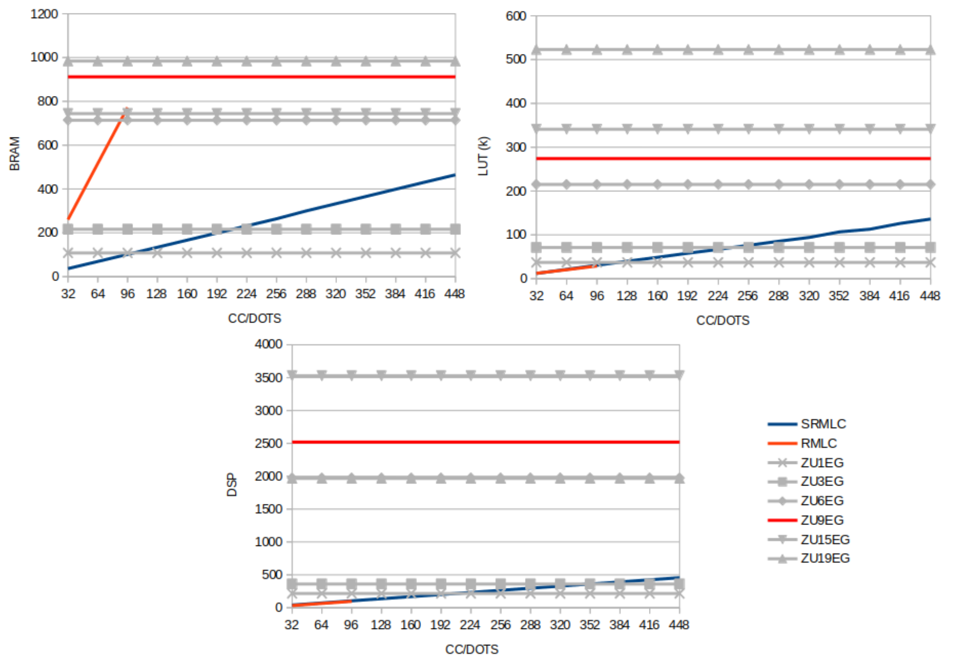

Figure 24. Resource scaling for the architectures.

We set the smallest configurations to have 32 RB / DOT modules. We then increased the number of RB / DOT modules with a step of 32. For RMLC configurations we stopped at 96, as this is the largest configuration that can fit on a ZCU102 development board. The graphs also show the limits for different FPGA devices from the Xilinx Ultrascale+ family. When the graph intersects one of these lines, it represents the largest possible number of RB / DOT modules that can be implemented on the corresponding FPGA device.

The weak point of the RMLC architecture is the consumption of BRAM resources. Therefore, we quickly approach the point on the graph (Figure 24) where the limit for the XCZU9EG device is crossed (XCZU9EG FPGA is used on the ZCU102 development board). As seen from Figure 24, all configurations for the SRMLC architecture can be implemented on all FPGA devices. The SRMLC architecture allows a significantly larger number of computing blocks to be accommodated because the resources required for implementation are balanced significantly better.

4.2.2. Comparison of the Achieved Throughput

From Table 8, we can see that the throughput of the proposed architecture is significantly improved, compared to RMLC architecture. The SRMLC architecture was parameterized so that the CALC module had 64 DOT modules, while the RECONF module had 8 RC-LANE modules. The RMLC architecture was parameterized to have 64 reconfigurable blocks.

For DTs, throughput is from 1.167 to 2 times higher (1.58 times on average), for SVMs it is from 0.467 to 2.3 times higher (1.65 times on average) and for MLP ANNs it is from 8.381 to 38 times higher (15.2 times on average), all compared to the corresponding RMLC implementations [33]. The results in Table 8 show, assuming that architectures contain the same number of compute blocks, so called RBs in RMLC and DOTs in SRMLC, that better throughput for DTs and SVMs in the SRMLC is achieved through the classifier sparsification.

When considering SVM models, for some datasets there was a decrease in throughput, as can be seen from Table 8. In these test cases, the bottleneck of the system is the RECONF module, which is not configured to have sufficient width. This can be seen from Equation (12) as the case where the value of N_r is higher than the value of N_c; hence, Equation (19) does not hold. It can be seen that this happens rarely and only for small datasets, showing that it is not a good compromise to spend additional resources to expand this module for scenarios that seldom happen.

Table 8. Throughput gain of SRMLC compared to RMLC [33].

Dataset	DTs	SVMs	MLPs
australian	2	1.636364	8.5
diabetes	1.166667	1.5	8.380952
glass	1.5	1.3	10.105263
heart	1.461538	2.125	12.8
hrtc	1.461538	2.125	8.533333
hepatitis	1.470588	2.3	13.538462
ionosphere	2	2.111111	25.191489
pblocks	1.6	1.076923	10.947368
sonar	1.833333	2.064516	38.037736
vehicle	1.5	1.157895	18.162162
wf21	1.5	1.470588	15.058824
wf40	1.483871	1.833333	25.962264
wdbc	1.6	0.466667	10.666667
wine	1.461538	2.125	12.8
zoo	1.642857	1.5	9.411765

Additionally, the results show that for MLP models the throughput is more improved. The reason is that the blocks in the RMLC architecture processed whole layers, while in the SRMLC architecture they processed individual neurons. This allows more than one neuron to be processed simultaneously when using the SRMLC architecture, which significantly contributes to increasing the architecture throughput.

The SRMLC architecture has another significant advantage, which these results do not show. When we implement both architectures (RMLC and SRMLC) on the same FPGA device, due to better SRMLC architecture scalability, it is possible to instantiate more DOTs, compared to RBs from RMLC architecture.

4.2.3. Comparison of the Achieved Processing Latency

As it can be seen from Table 9, the processing latency, as one of the most important performance metric parameters for most of the applications, is drastically improved in the SRMLC when compared to the previously published RMLC architecture [33]. For comparison purposes, the SRMLC architecture was parameterized so that the CALC module had 64 DOT modules, while the RECONF module had 8 RC-LANE modules. The RMLC architecture was parameterized to have 64 reconfigurable blocks.

Table 9. Processing latency reduction in SRMLC compared to RMLC [33].

Dataset	DTs	SVMs	MLPs
australian	3.349398	31.972603	3.192308
diabetes	2.989011	31.333333	3.551724
glass	3.011905	20.612903	5.376623
heart	3.174419	28.169492	6.860759
hrtc	3.174419	33.938462	3.098039
hepatitis	2.659091	28.339623	6.063492
ionosphere	3.695652	49.702703	14.116279
pblocks	3.213483	33.57764	5.792208
sonar	4.111111	66.032086	22.23913
vehicle	3.404255	41.588235	9.236842
wf21	3.730769	59.96347	6.624
wf40	4.428571	84.124524	15.282609
wdbc	2.97561	14.934783	5.717949
wine	2.845238	52.795181	6.860759
zoo	3.162791	24.105263	3.584906

From Table 9, we can see that the latency during the processing of DTs in the SRMLC is reduced from 2.66 to 4.43 times (3.33 on average), for MLP ANNs the latency is reduced from 3.1 to 22.2 times (7.84 on average) and for SMVs the processing latency can be from 14.9 to 84.1 times shorter (40.08 on average).

This significant latency reduction is obtained through the parallelization and sparsification. Most of the latency reduction was achieved from implementation in which a large number of DOTs processed a given input classification instance in parallel. This approach is in complete contrast to the RMLC architecture [33] and leads to significantly improved latency for all classifier types. Sparsification has less effect on improving latency compared to parallelization. The impact that sparsification has on improving throughput is proportional to the impact it has on reducing latency. For example, if the throughput is improved two times and the latency is reduced eight times, that means that the sparsification improved the latency two times, while the additional improvement of four times can be attributed to parallelization.

4.2.4. Comparison of the Energy Consumption

Table 10 shows the reduction in memory requirements for storing classifier coefficients needed by the SRMLC, when compared to the RMLC architecture, proposed in [33].

Table 10. SRMLC memory storage reduction compared to RMLC [33], expressed in % as a result of sparse data representation.

Dataset	DTs	SVMs	MLPs
australian	−44.66	−55.32	−87.50
diabetes	30.98	−67.54	−75
glass	−19.33	−71.39	−62.50
heart	0.60	−78.10	−75
hrtc	−50.16	−80.75	−93.75
hepatitis	−19.70	−93.65	−75.07
ionosphere	49.88	−42.58	−87.50
pblocks	27.27	−60.91	−75
sonar	−15.25	−47.65	−75
vehicle	−21.74	2.25	−25.07
wf21	37.88	−42.93	−87.50
wf40	−23.07	−41.73	−93.75
wdbc	−18.83	9.83	−87.63
wine	17.58	−30.32	−75.10
zoo	−28.83	−22.53	−75

Negative values in Table 10 indicate that the SRMLC requires fewer memory resources compared to RMLC [33], while positive values indicate that, for the corresponding datasets, the SRMLC requires additional memory resources. As it can be seen from Table 10, a memory reduction for DTs can be positive, at least for some datasets used in the experiments, which can be explained by the fact that sparsification of hyperplane coefficients leads to larger and “deeper” DTs, with a large number of nodes, where each node must store one hyperplane coefficients set. Table 10 also shows that for the majority of datasets, especially for SVMs and MLP ANNs, the required memory for storing model parameters is reduced and for several datasets it is severely reduced.

In the case of SVM classifiers, memory usage is generally improved, sometimes drastically, except in some cases. In these cases, the level of sparsification is small, so the extra space needed to accommodate the increments actually increases memory usage.

As a consequence of the model parameters sparsification, the energy consumption is significantly reduced as well. It is a well known fact from the available literature that data transfers from external DRAM are the most expensive in terms of the energy consumption [39]. As a result, reduced storage requirements in the SRMLC lead to significant energy saving, compared to solutions where dense data representation is used.

Due to the reduced number of DRAM accesses, as a result of sparsification, the energy consumption for DRAM data transfers is reduced from −49.88% up to 50.16% in the case of DT processing (5.16% on average), from −9.83% up to 93.65% in the case of SVM processing (48.22% on average) and from 25.07% up to 93.75% in the case of ANN processing (76.69% on average), compared to the scenario when non-sparsified models are being processed.

4.3. Comparison with Embedded Software Implementation

Tables 11–13 present the processing latency of different classifiers when they are being executed on the SRMLC hardware accelerator, compared to their software implementations being executed on the embedded processor. All three embedded software applications, providing results from Tables 11–13, were developed as GCC embedded Linux applications. The hardware platform used for testing was the ZCU102 evaluation board [66], so all embedded applications were executed on a quad-core Arm® Cortex®-A53 processor. The DT benchmarking application was implemented as a plain GCC application, without the usage of any specific libraries, where all underlying data structures were developed from scratch. However, SVM benchmarking GCC application was based on the LIBSVM library for Support Vector Machines [69], while the GCC embedded application for benchmarking ANNs used Tensorflow-lite framework [70] to model MLP ANNs.

Table 11. Processing latency comparison between the software implementation of DT classifier compared to SRMLC.

Dtst	Inst#	SWmin [ns]	SWmax [ns]	SWavg [ns]	HWb [ns]	HW [ns]	Gb	G
australian	138	1180	7601	3094	40	332	77.354	9.32
diabetes	154	860	5961	2240	48	364	46.669	6.154
glass	43	900	2840	1699	40	336	42.475	5.057
heart	54	620	17,062	1569	52	344	30.166	4.560
hrtc	60	1110	3371	1951	52	344	37.536	5.674
hepatitis	16	780	1610	1401	68	352	20.609	3.981
ionosphere	71	1180	3931	2916	80	368	36.453	7.925
pblocks	5001	1450	1086	1758	40	356	43.947	4.938
sonar	42	1920	5630	3755	144	432	26.078	8.693
vehicle	170	2001	6061	3748	64	376	58.56	9.968
waveform21	1000	2510	19,512	3886	72	416	53.975	9.341
waveform40	1000	3831	35,464	6717	124	448	54.171	14.993
wdbc	114	2021	6161	2461	40	328	61.516	7.502
wine	36	1100	1400	1156	52	336	22.236	3.441
zoo	20	1320	3180	1755	56	344	31.339	5.102

Table 12. Processing latency comparison between the software implementation of SVM classifier compared to SRMLC.

Dtst	Inst#	SWmin [ns]	SWmax [ns]	SWavg [ns]	HWb [ns]	HW [ns]	Gb	G
australian	138	8677.01	9101.15	8857.68	176	584	50.328	15.167
diabetes	153	10,699.51	11,215.45	10,943.79	256	792	42.749	13.818
glass	42	7956.03	8391.86	8146.1	120	496	67.884	16.424
heart	54	7401.94	7791.28	7600.95	96	472	79.177	16.104
hrtc	59	8553.5	8945.94	8691.98	128	520	67.906	16.715
hepatitis	31	6703.85	7043.12	6807.13	80	424	85.089	16.055
ionosphere	70	11,845.82	11,949.78	11,629.36	216	592	53.840	19.644
pblocks	1085	18,039.23	28,246.4	18,328.87	624	1288	29.373	14.230
sonar	41	18,805.5	19,426.11	19,042.93	372	748	51.191	25.458
vehicle	169	21,289.59	22,107.6	21,526.56	760	1360	28.324	15.828
waveform21	1000	76,443.91	82,371.23	77,257.10	2788	4380	27.711	17.639
waveform40	1000	154,107.09	317,612.89	155,688.16	4512	6296	34.505	24.728

Table 12. *Cont.*

Dtst	Inst#	SWmin [ns]	SWmax [ns]	SWavg [ns]	HWb [ns]	HW [ns]	Gb	G
wdbc	113	12,567.04	13,126.85	12,770.17	360	736	35.473	17.351
wine	35	7369.76	7758.14	7528.95	256	664	29.410	11.339
zoo	20	6816.63	7173.78	6963.81	112	456	62.177	15.272

Column *Dtst* shows the dataset which was used. Tests were conducted on different numbers of input classification instances and those are shown in column *inst#*. *SWmin*, *SWmax* and *SWavg* stand for the minimum, maximum and average processing latency of classifying one instance when executed in software, respectively. *HWb* shows the instance processing latency when the classifier is running on the SRMLC, while using the batch mode of processing, in which multiple input instances are read from external DRAM, which significantly reduces classification time. For the chosen datasets, the size of the batch was set to be the maximum for the corresponding experiments, which means that it was equal to the value of *inst#*. On the other side, the *HW* column presents processing latencies of the running classifier on the SRMLC in normal mode, where a single input instance is fetched each time, before starting classification.

Table 13. Processing latency comparison between the software implementation of ANN classifier compared to SRMLC.

Dtst	Inst#	SWmin [ns]	SWmax [ns]	SWavg [ns]	HWb [ns]	HW [ns]	Gb	G
australian	690	6860	143,471	8174.86	320	632	25.546	12.935
diabetes	768	5910	412,680	8027.00	296	608	27.118	13.202
glass	214	6970	149,393	9503.35	304	616	31.261	15.428
heart	270	6370	44,101	6946.23	320	632	21.707	10.991
hrtc	297	7090	529,399	14,229.04	312	624	45.606	22.803
hepatitis	80	7341	33,510	7868.64	336	648	23.419	12.143
ionosphere	351	8350	94,292	9466.01	376	688	25.176	13.759
pblocks	5427	6960	434,109	7819.18	304	616	25.721	12.693
sonar	208	10,150	57,852	11,048.09	424	736	26.057	15.011
vehicle	846	7300	441,552	12,000.27	296	608	40.541	19.737
waveform21	5000	7410	222,466	8307.54	332	644	25.023	12.900
waveform40	5000	8560	675,946	10,524.11	424	736	24.821	14.299
wdbc	569	8090	300,625	9415.08	312	624	30.177	15.088
wine	178	6991	1,370,359	17,856.08	320	632	55.800	28.253
zoo	101	8370	481,810	20,403.84	328	640	62.207	31.881

All processing latencies presented in Tables 11–13 are expressed in nanoseconds. The last two columns show the gain of the running classifier on the SRMLC, compared to the corresponding embedded software implementations, where the *Gb* column shows the gain of batch-mode processing latency and the *G* column presents the normal-mode gain. Table 11 shows that the processing latency gain of the SRMLC accelerated DT classifier in normal mode ranges from 3.98 up to 14.99 (7.11 on average). Similarly, from Table 12 it can be seen that, when run on the SRMLC in normal mode, the processing latency gain of the SVM classifier ranges from 11.34 up to 25.46 times (17.05 on average). Finally, Table 13 shows that the MLP ANN classifier, run in normal mode on the SRMLC accelerator, outperforms the one running in the software from 10.99 up to 31.88 times (16.74 on average), when the processing latency gain is used as a comparison metric. As it can be seen from the columns *Gb* in Tables 11–13 that when classifiers are run on the SRMLC accelerator in a batch mode, the gain is even higher: from 20.61 up to 77.35 (42.87 on average) for DTs, from 27.71 up to 85.09 (49.68 on average) for SMVs and from 21.71 up to 62.21 (32.68 on average) for MLP ANNs.

As can be seen from the results presented in Tables 11–13, the SRMLC architecture offers high-performance results when compared with traditional software implementations of ANN, DT, and SVM classifiers. There are several reasons why this was achieved. The SRMLC architecture uses a pipelining technique at several levels to increase the instance of processing throughput. Similarly, due to the parallelization of the main operations, implemented within the array of DOT modules, an additional increase in the processing throughput, but even more importantly, a decrease in the instance processing latency, has been achieved. Finally, by the fact that the SRMLC is able to directly process sparsified ML models, additional performance improvement was attained.

Apart from the obvious performance improvement due to the significant processing latency reduction, an additional benefit of using the SRMLC hardware accelerator is related to the fact that the classification processing latancy is deterministic, always taking exactly the same amount of time, which can be crucial for real-time applications in which the response latency is critical.

5. Conclusions

In this study, the universal reconfigurable hardware accelerator for sparse machine learning classifiers is proposed, which supports three classifier types: decision trees, artificial neural networks and support vector machines. While hardware accelerators for these classifiers can be found in the available literature, to the author's best knowledge, the accelerator presented in this study is the first hardware accelerator that is optimized to work with sparse classifier models, which results in a higher throughput, reduced processing latency, smaller memory footprint and decreased energy consumption due to reduced data movements, compared to the hardware implementations of traditional predictive models. In order to benchmark the proposed hardware accelerating platform, we have also presented the algorithms for sparse decision trees induction, sparsification of support vector machines and artificial neural networks pruning. Experiments carried out on standard benchmark datasets from the UCI Machine Learning Repository database show that the proposed sparsification algorithms allow a significant predictive models compression, with a negligible prediction accuracy drop: on average 61.7% for decision trees, 39.12% for support vector machines and 81.3% for artificial neural networks. Experimental results also show that using such compressed classifier models increases throughput up to 38 times, decreases processing latency up to 84 times and reduces energy consumption for DRAM data transfers up to 76.69%. Our hardware accelerator, as it is shown in the experimental results section, significantly outperforms machine learning classifiers implemented in embedded software. DT classification, accelerated by the SRMLC, is up to 77 times faster when compared to the corresponding DT classification implemented in the software. Similarly, when run on our accelerator, the SVM classification relative speedup is up to 85 times faster and the MLP ANN classification relative speedup is up to 62 times faster, when compared to corresponding embedded software implementations.

Author Contributions: Conceptualization, V.V. and R.S.; Methodology, V.V., P.T. and R.S.; Software, V.V. and P.T.; Supervision, R.S.; Visualization, V.V. and P.T.; Writing—original draft, V.V., P.T. and R.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement number 856967 and has been supported by the Ministry of Education, Science and Technological Development through project No. 451-03-68/2022-14/ 200156 "Innovative scientific and artistic research from the FTS (activity) domain".

Data Availability Statement: UCI Machine Learning Repository, Center for Machine Learning and Intelligent Systems. Available online: https://archive.ics.uci.edu/ml/index.php (accessed on 28 February 2022).

Acknowledgments: The authors would like to thank the editors and reviewers for their efforts and suggestions to improve our manuscript.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Olson, D.L.; Wu, D. *Predictive Data Mining Models*, 2nd ed.; Springer Nature: Singapore, 2020.
2. Kantardzic, M. *Data Mining: Concepts, Models, Methods, and Algorithms*, 3rd ed.; John Wiley & Sons: Hoboken, NJ, USA, 2020.
3. Zaki, M.J.; Wagner, M. *Data Mining and Machine Learning: Fundamental Concepts and Algorithms*, 2nd ed.; Cambridge University Press: Cambridge, UK, 2020.
4. Breiman, L.; Friedman, J.; Stone, C.; Olsen, R. *Classification and Regression Trees*; CRC Press: Boca Raton, FL, USA, 1984.
5. Quinlan, R. Induction of decision trees. *Mach. Learn.* **1986**, *1*, 81–106. [CrossRef]
6. Cortes, C.; Vapnik, V. Support-vector networks. *Mach. Learn.* **1995**, *20*, 273–297. [CrossRef]
7. Haykin, S. *Neural Networks and Learning Machines*; Pearson Education: Delhi, NCR, Noida, India, 2007.
8. McCullock, W.; Pitts, W. A Logical Calculus of Ideas Immanent in Nervous Activity. *Bull. Math. Biophys.* **1943**, *5*, 115–133. [CrossRef]
9. Mierswa, I.; Wurst, M.; Klinkenberg, R.; Scholz, M.; Euler, T. Yale: Rapid prototyping for complex data mining tasks. In Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, 20–23 August 2006; pp. 935–940.
10. The R Project for Statistical Computing. Available online: http://www.r-project.org (accessed on 1 September 2021).
11. Hall, M.; Frank, E.; Holmes, G.; Pfahringer, B.; Reutemann, P.; Witten, I. The WEKA data mining software: An update. *ACM SIGKDD Explor. Newsl.* **2009**, *11*, 10–18. [CrossRef]
12. Eltanbouly, S.; Bashendy, M.; AlNaimi, N.; Chkirbene, Z.; Erbad, A. Machine learning techniques for network anomaly detection: A survey. In Proceedings of the 2020 IEEE International Conference on Informatics, IoT, and Enabling Technologies (ICIoT), Doha, Qatar, 2–5 February 2020; pp. 156–162.
13. Rabhi, F.A.; Mehandjiev, N.; Baghdadi, A. State-of-the-Art in Applying Machine Learning to Electronic Trading. In *International Workshop on Enterprise Applications, Markets and Services in the Finance Industry*; Springer: Cham, Switzerland, 2020; pp. 3–20.
14. Dixon, M.F.; Halperin, I.; Bilokon, P. *Machine Learning in Finance*; Springer International Publishing: New York, NY, USA, 2020.
15. Zhao, S.; Chen, S.; Yang, H.; Wang, F.; Wei, Z. RF-RISA: A novel flexible random forest accelerator based on FPGA. *J. Parallel Distrib. Comput.* **2021**, *157*, 220–232. [CrossRef]
16. Malhotra, K.; Singh, A.P. Implementation of decision tree algorithm on FPGA devices. *IAES Int. J. Artif. Intell.* **2021**, *10*, 131. [CrossRef]
17. Alcolea, A.; Resano, J. FPGA accelerator for gradient boosting decision trees. *Electronics* **2021**, *10*, 314. [CrossRef]
18. Molina, R.; Loor, F.; Gil-Costa, V.; Nardini, F.M.; Perego, R.; Trani, S. Efficient traversal of decision tree ensembles with FPGAs. *J. Parallel Distrib. Comput.* **2021**, *155*, 38–49. [CrossRef]
19. Haytham, A. FPGA Acceleration of Tree-based Learning Algorithms. *Adv. Sci. Technol. Eng. Syst. J. Spec. Issue Multidiscip. Sci. Eng.* **2020**, *5*, 237–244.
20. Owaida, M.; Kulkarni, A.; Alonso, G. Distributed inference over decision tree ensembles on clusters of FPGAs. *ACM Trans. Reconfigurable Technol. Syst. (TRETS)* **2019**, *12*, 1–27. [CrossRef]
21. Ramadurgam, S.; Perera, D.G. An Efficient FPGA-Based Hardware Accelerator for Convex Optimization-Based SVM Classifier for Machine Learning on Embedded Platforms. *Electronics* **2021**, *10*, 1323. [CrossRef]
22. Younes, H.; Ibrahim, A.; Rizk, M.; Valle, M. Algorithmic-level approximate tensorial SVM using high-level synthesis on FPGA. *Electronics* **2021**, *10*, 205. [CrossRef]
23. Afifi, S.; GholamHosseini, H.; Sinha, R. FPGA implementations of SVM classifiers: A review. *SN Comput. Sci.* **2020**, *1*, 1–17. [CrossRef]
24. Batista, G.C.; Oliveira, D.L.; Saotome, O.; Silva, W.L. A Low-Power Asynchronous Hardware Implementation of a Novel SVM Classifier, with an Application in a Speech Recognition System. *Microelectron. J.* **2020**, *105*, 104907. [CrossRef]
25. Baez, A.; Himar, F.; Samuel, O.; Giordana, F.; Emanuele, T.; Abian, H.; Francesco, L.; Giovanni, D.; Gustavo, M.C.; Roberto, S. High-level synthesis of multiclass SVM using code refactoring to classify brain cancer from hyperspectral images. *Electronics* **2019**, *8*, 1494. [CrossRef]
26. Afifi, S.; GholamHosseini, H.; Sinha, R. A system on chip for melanoma detection using FPGA-based SVM classifier. *Microprocess. Microsystems* **2019**, *65*, 57–68. [CrossRef]
27. Luo, A.; An, F.; Zhang, X.; Mattausch, H.J. A hardware-efficient recognition accelerator using Haar-like feature and SVM classifier. *IEEE Access* **2019**, *7*, 14472–14487. [CrossRef]
28. Westby, I.; Yang, X.; Liu, T.; Xu, H. FPGA acceleration on a multi-layer perceptron neural network for digit recognition. *J. Supercomput.* **2021**, *77*, 14356–14373. [CrossRef]
29. Wu, C.; Fresse, V.; Suffran, B.; Konik, H. Accelerating DNNs from local to virtualized FPGA in the Cloud: A survey of trends. *J. Syst. Archit.* **2021**, *119*, 102257. [CrossRef]
30. Valencia, D.; Fard, S.F.; Alimohammad, A. An artificial neural network processor with a custom instruction set architecture for embedded applications. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2020**, *67*, 5200–5210. [CrossRef]
31. Medus, L.D.; Iakymchuk, T.; Frances-Villora, J.V.; Bataller-Mompeán, M.; Rosado-Muñoz, A. A novel systolic parallel hardware architecture for the FPGA acceleration of feedforward neural networks. *IEEE Access* **2019**, *7*, 76084–76103. [CrossRef]

32. Hwang, R.; Kim, T.; Kwon, Y.; Rhu, M. Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations. In Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), Valencia, Spain, 30 May–3 June 2020; pp. 968–981.
33. Vranjković, V.; Struharik, R.; Novak, L. Reconfigurable hardware for machine learning applications. *J. Circuits Syst. Comput.* **2015**, *24*, 1550064. [CrossRef]
34. Vranjković, V.; Struharik, R.; Novak, L. Hardware acceleration of homogeneous and heterogeneous ensemble classifiers. *Microprocess. Microsyst.* **2015**, *39*, 782–795. [CrossRef]
35. Chen, W.; Wilson, J.; Tyree, S.; Weinberger, K.; Chen, Y. Compressing neural networks with the hashing trick. In Proceedings of the International Conference on Machine Learning, Lille, France, 6–11 July 2015; pp. 2285–2294.
36. Han, S.; Mao, H.; Dally, W.J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv* **2015**, arXiv:1510.00149.
37. Han, S.; Pool, J.; Tran, J.; Dally, W.; Chen, Y. Learning both weights and connections for efficient neural network. In Proceedings of the Advances in Neural Information Processing Systems, Montreal, QC, Canada, 7–12 December 2015; pp. 1135–1143.
38. Iandola, F.N.; Han, S.; Moskewicz, M.W.; Ashraf, K.; Dally, W.J.; Keutzer, K. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size. *arXiv* **2016**, arXiv:1602.07360.
39. Han, S.; Liu, X.; Mao, H.; Pu, J.; Pedram, A.; Horowitz, M.A.; Dally, W.J. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Comput. Archit. News* **2016**, *44*, 243–254. [CrossRef]
40. Liang, T.; Glossner, J.; Wang, L.; Shi, S.; Zhang, X. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing* **2021**, *461*, 370–403. [CrossRef]
41. Kretowski, M. An evolutionary algorithm for oblique decision tree induction. In Proceedings of the International Conference on Artificial Intelligence and Soft Computing, Zakopane, Poland, 7–11 June 2004; pp. 54–68.
42. Kretowski, M.; Grześ, M. Evolutionary learning of linear trees with embedded feature selection. In Proceedings of the International Conference on Artificial Intelligence and Soft Computing, Zakopane, Poland, 25–29 June 2006; pp. 400–409.
43. Keerthi, S.S.; Chapelle, O.; DeCoste, D. Building support vector machines with reduced classifier complexity. *J. Mach. Learn. Res.* **2006**, *7*, 1493–1515.
44. Vranjkovic, V.; Struharik, R. Hardware Acceleration of Sparse Support Vector Machines for Edge Computing. *Elektron. Ir Elektrotechnika* **2020**, *26*, 42–53. [CrossRef]
45. Yang, J.; Fu, W.; Cheng, X.; Ye, X.; Dai, P.; Zhao, W. S2Engine: A novel systolic architecture for sparse convolutional neural networks. *IEEE Trans. Comput.* **2021**. [CrossRef]
46. Xu, H.; Shiomi, J.; Onodera, H. MOSDA: On-Chip Memory Optimized Sparse Deep Neural Network Accelerator with Efficient Index Matching. *IEEE Open J. Circuits Syst.* **2020**, *2*, 144–155. [CrossRef]
47. Liu, B.; Chen, X.; Han, Y.; Xu, H. Swallow: A versatile accelerator for sparse neural networks. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 4881–4893. [CrossRef]
48. You, W.; Wu, C. RSNN: A software/hardware Co-optimized framework for sparse convolutional neural networks on FPGAs. *IEEE Access* **2020**, *9*, 949–960. [CrossRef]
49. Liang, Y.; Lu, L.; Xie, J. OMNI: A framework for integrating hardware and software optimizations for sparse CNNs. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *40*, 1648–1661. [CrossRef]
50. Teodorovic, P.; Struharik, R. Hardware Acceleration of Sparse Oblique Decision Trees for Edge Computing. *Elektron. Ir Elektrotechnika* **2019**, *25*, 18–24. [CrossRef]
51. Serkani, E.; Gharaee Garakani, H.; Mohammadzadeh, N. Anomaly detection using SVM as classifier and decision tree for optimizing feature vectors. *ISC Int. J. Inf. Secur.* **2019**, *11*, 159–171.
52. Serkani, E.; Garakani, H.G.; Mohammadzadeh, N.; Vaezpour, E. Hybrid anomaly detection using decision tree and support vector machine. *Int. J. Electr. Comput. Eng.* **2018**, *12*, 431–436.
53. Lu, H.; Ma, X. Hybrid decision tree-based machine learning models for short-term water quality prediction. *Chemosphere* **2020**, *249*, 126169. [CrossRef]
54. Carson, J.; Hollingsworth, K.; Datta, R.; Clark, G.; Segev, A. A Hybrid Decision Tree-Neural Network (DT-NN) Model for Large-Scale Classification Problems. In Proceedings of the 2020 IEEE International Conference on Big Data (Big Data), Atlanta, GA, USA, 10–13 December 2020; pp. 4103–4111.
55. Jena, M.; Behera, R.K.; Dehuri, S. Hybrid Decision Tree for Machine Learning: A Big Data Perspective. In *Advances in Machine Learning for Big Data Analysis*; Dehuri, S., Chen, Y.W., Eds.; Springer: Cham, Switzerland, 2022; pp. 223–239.
56. Khraisat, A.; Gondal, I.; Vamplew, P.; Kamruzzaman, J.; Alazab, A. Hybrid intrusion detection system based on the stacking ensemble of c5 decision tree classifier and one class support vector machine. *Electronics* **2020**, *9*, 173. [CrossRef]
57. Heath, D.; Kasif, S.; Salzberg, S. Induction of oblique decision trees. In Proceedings of the IJCAI, Chambery, France, 28 August–3 September 1993; pp. 1002–1007.
58. Cantu-Paz, E.; Kamath, C. Inducing oblique decision trees with evolutionary algorithms. *IEEE Trans. Evol. Comput.* **2003**, *7*, 54–68. [CrossRef]
59. Otero, F.; Freitas, A.; Johnson, C.G. Inducing decision trees with an ant colony optimization algorithm. *Appl. Soft Comput.* **2012**, *12*, 3615–3626. [CrossRef]

60. Levi, D. HereBoy: A fast evolutionary algorithm. In Proceedings of the Second NASA/DoD Workshop on Evolvable Hardware, Palo Alto, CA, USA, 13–15 July 2000; pp. 17–24.
61. Struharik, R.; Vranjković, V.; Dautović, S.; Novak, L. Inducing oblique decision trees. In Proceedings of the 2014 IEEE 12th International Symposium on Intelligent Systems and Informatics (SISY), Subotica, Serbia, 11–13 September 2014; pp. 257–262.
62. Platt, J. *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*; MSRTR: Microsoft Research: Redmond, WA, USA, 1998; Volume 3, pp. 88–95.
63. Kreßel, U. Pairwise classification and support vector machines. In *Advances in Kernel Methods: Support Vector Learning*; Burges, C.J.C., Scholkopf, B., Smola, A.J., Eds.; MIT Press: Cambridge, MA, USA, 1999; pp. 255–268.
64. Tensorflow. Available online: http://www.tensorflow.org (accessed on 1 September 2021).
65. Xilinx Vivado Design Suite. Available online: https://www.xilinx.com/developer/products/vivado.html (accessed on 26 October 2021).
66. Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. Available online: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html# (accessed on 26 October 2021).
67. Genus Synthesis Solution. Available online: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html (accessed on 26 October 2021).
68. Innovus Implementation System. Available online: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html (accessed on 26 October 2021).
69. Chang, C.; Lin, C. LIBSVM: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.* **2011**, *3*, 27:1–27:27. [CrossRef]
70. Deploy Machine Learning Models on Mobile and IoT Devices. Available online: https://www.tensorflow.org/lite (accessed on 28 October 2021).