https://doi.org/10.5755/j01.eie.26.3.25796 ELEKTRONIKA IR ELEKTROTECHNIKA, ISSN 1392-1215, VOL. 26, NO. 3, 2020

Hardware Acceleration of Sparse Support Vector Machines for Edge Computing

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

***Abstract*—In this paper, a hardware accelerator for sparse support vector machines (SVM) is proposed. We believe that the proposed accelerator is the first accelerator of this kind. The accelerator is designed for use in field programmable gate arrays (FPGA) systems. Additionally, a novel algorithm for the pruning of SVM models is developed. The pruned SVM model has a smaller memory footprint and can be processed faster compared to dense SVM models. In the systems with memory throughput, compute or power constraints, such as edge computing, this can be a big advantage. The experiments on several standard datasets are conducted, which aim is to compare the efficiency of the proposed architecture and the developed algorithm to the existing solutions. The results of the experiments reveal that the proposed hardware architecture and SVM pruning algorithm has superior characteristics in comparison to the previous work in the field. A memory reduction from 3 % to 85 % is achieved, with a speed-up in a range from 1.17 to 7.92.**

***Index Terms*—Support vector machines; Hardware accelerator architectures; Edge computing.**

I. INTRODUCTION

Support Vector Machine (SVM) is one kind of machine learning algorithms, firstly introduced in [1]. SVMs were one of the most popular predicting models until Convolutional Neural Networks (CNN) have been proposed. Also, there is work on hybrid models of CNN and SVM, e.g., in [2], [3].

As with other supervised machine learning algorithms, SVM contains two phases: a learning phase and a predicting phase. The SVM model approximates unknown function U. In the learning phase, input to the SVM training algorithm is a training set with *m* instances, each of which has *n* attributes

$$TS = \{x_i, y_i\},\ i = 1..m,\ x_i \in X \subseteq R^n,\ y \in Y = \{+1, -1\}. \quad (1)$$

The output of the SVM learning phase is a linear classification function *L* in a form

$$L : X \to Y,\ L(x) = w \times x + b,\ w \in R^n,\ b \in R. \quad (2)$$

The function *L* approximates the unknown function : *X* → *Y*.

Function *L*, as previously defined, can correctly classify only the linearly separable training set. To be able to classify instances that belong to non-linearly separable classes, one can apply a non-linear mapping φ to input space *X* to transform the original input feature space to a high-dimensional feature space $Z \subseteq R^k,\ k >> n.$ The same linear classification function can be used to separate points in this high-dimensional feature space, achieving non-linear classification in the original space *X*. Using the kernel trick, the SVM classifier still can work in the original space, so the non-linear classification function can be expressed in the form

$$F(x) = w \times \phi(x) + b. \quad (3)$$

SVM splits points in the input feature space with a linear hyperplane. The hyperplane is located at a maximal distance from the training set instances of each class that is closest to the hyperplane. In general, this cannot be done without errors for all training set instances, so the SVM algorithm allows some training set instances to be incorrectly classified. The problem of finding an optimal splitting hyperplane can be described formally as the constrained quadratic programming (CQP) problem:

$$\min_{w,b}\left[\frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}\varepsilon_i\right], \quad (4)$$

$$y_i(w \times \phi(x) + b) \geq 1 - \varepsilon_i,\ C > 0,\ \varepsilon_i \geq 0,\ \forall i \in 1...m. \quad (5)$$

The first part of the minimization function puts the hyperplane at the maximal margin, while the second part changes the position of the hyperplane in a way that minimizes the number of training set instances that will be misclassified. These two criteria are contradictory, so the parameter C defines a trade-off between them.

Using a method of Lagrange multipliers, the original CQP problem can be transformed into its dual form, which is easier to solve (6)–(9):

Manuscript received 24 February, 2020; accepted 29 April, 2020.
This work was partially supported by Serbian Ministry of Education and Science under a grant (No. TR32016 - “Innovative electronic components and systems based on inorganic and organic technologies embedded in consumer goods and products”). This project has received funding from the European Union’s Horizon 2020 research and innovation programme under a Grant Agreement (No. 856697).

$$\min_{\alpha}\left[\frac{1}{2}\alpha^T W\alpha + r^T\alpha\right], \tag{6}$$

$$w_{ij} = y_i y_j K(x_i, x_j), \tag{7}$$

$$K(x_i, x_j) = \phi(x_i)\phi(x_j), \tag{8}$$

$$0 \le \alpha_i \le C,\ r_i = \pm 1,\ \forall i \in 1...m,\ y^T\alpha = 0. \tag{9}$$

The $m \times m$ matrix W is symmetric positive semidefinite, and its elements are w_{ij}. The function K is called kernel and it implicitly defines a non-linear mapping - φ. There are several popular kernels in use, some of them are:

$$K(x_i, x_j) = (x_i \times x_j + 1)^d : polynomial, \tag{10}$$

$$K(x_i, x_j) = e^{-\gamma\|x_i - x_j\|^2} : \text{ radial basis function}, \tag{11}$$

$$K(x_i, x_j) = \tanh(ax_i \times x_j + b) : sigmoid. \tag{12}$$

The parameters of the kernel are *d*, *γ*, *a*, and *b*.

More details about the theory behind the SVM training can be found in [4].

After the training procedure completes, some of the Lagrange multipliers will be zero, and others will be non-zero. The training set instances corresponding to the non-zero multipliers from input data set are called support vectors. Let *l* be the number of support vectors ($l \le m$).

The SVM training algorithm outputs a non-linear approximation function in the following form

$$V : X \to Y, V(x) = b + \sum_{i=1}^{l} y_i \alpha_i K(s_i, x), \tag{13}$$

where s_i is the support vector and *x* is the input instance.

The evaluation of the function *V(x)* for the unknown input instance is called classification. This is the second phase of the SVM algorithm. If the value of the function is greater than zero, the input instance is classified as a class with label +1, otherwise, it is labeled as a class -1.

The SVM algorithm can be generalized to do a multi-class classification also. One approach for this generalization is given in [5].

The efficient algorithm for the training of SVM is a Sequential Minimal Optimization (SMO), firstly introduced in [6]. Several additional optimizations of the SMO algorithm have been proposed, one example being [7].

Machine learning algorithms are being used in different kinds of applications, ranging from servers to embedded systems. Deploying machine learning models to embedded systems is especially difficult because of the size of the models. Therefore, the compression and reduction of the size of the models attracted a lot of research effort [8]. The compression of the size of SVM models is presented in [12]. The main idea in those papers for the reduction of SVM size is in a smart selection of support vectors during the training of SVMs. In this paper, we present a complementary idea for the size reduction of SVMs: the removal of attributes from support vectors. To the best of our knowledge, this approach to the reduction of SVM size has not been previously reported for SVMs.

We propose the AST-SVM algorithm for training of attribute sparse SVMs, and the ASA-SVM hardware accelerator that can take advantage of this kind of sparse SVMs. The presented accelerator is aimed for use in "FPGA" (Field-Programmable Gate Array) systems, and it is especially useful for embedded and edge applications, where designers are constrained with severe computational, memory throughput, and power limitations. The AST-SVM algorithm produces SVMs with an increased number of zeroes in model parameters. By operating on these sparse support vector representations, the ASA-SVM accelerator usually requires less memory for the storage of support vectors, and by skipping calculations of zero products during classification, it can reach higher performance than comparable dense accelerators.

The hardware acceleration of SVMs has been an interesting topic in the research community, resulting in several proposed hardware architectures [14]]–[[19]. In contrast to the ASA-SVM, all previously proposed architectures operate on dense SVMs. As far as we know, the ASA-SVM accelerator is the first to operate on sparse SVMs.

The rest of this paper is structured as follows. The algorithm for pruning SVMs is given in section II. The AST-SVM algorithm sets a predefined number of attributes of training instances to zero; therefore, reducing the size of the classifier after the training is done. Section III contains a description of the ASA-SVM architecture, which is designed to take advantage of sparse SVMs, which contains a large number of zeroes in each of the support vectors. In that way, a higher processing performance can be achieved, as well as less memory usage. In section IV, the experimental results of the AST-SVM algorithm and ASA-SVM architecture are presented for various standard datasets. Section V contains conclusions.

II. ATTRIBUTE SPARSE TRAINING SVM (AST-SVM) ALGORITHM

The AST-SVM algorithm uses a simple method for choosing which values to eliminate. Although simple, the method gives good results. The presented algorithm uses standard SVM training as the sub-task. In the AST-SVM algorithm, some of the values of attributes from some of the input instances of the input training set are set to zero. Then, the standard SVM training algorithm is called. After the SVM training is completed, the output is the sparse SVM model.

The algorithm starts with some training dataset. At every iteration, the algorithm keeps all training instances sorted by their number of non-zero values. The training instance with the biggest number of non-zero values is chosen for attribute elimination. Then, the attribute whose absolute value is closest to zero is set to zero, i.e., eliminated. This is repeated until a predefined percentage of attribute values is eliminated from the training set. The output of this elimination procedure is the pruned input training dataset. After the elimination phase is over, the standard SVM training is started with the pruned input dataset. This procedure, as a result, outputs the SVM model with sparse support vectors.

Algorithm 1. Attribute Sparse Training SVM (AST-SVM) Algorithm.

```
AST-SVM (DS, target_zv_pct)

DS - Input dataset used for the training of
       SVM. The DS contains training
       instances, which are potential support
       vectors.
target_zv_pct - Target percentage of zero
       values in DS

Sort DS instances by the number of attributes
       different from zero
current_zv_pct is the current percentage of
       zeroes in DS
Determine current_zv_pct
While current_zv_pct is less then
       target_zv_pct
do
{
  Take the instance with the biggest number of
       non-zero values.
  Determine the attribute value which is not
       zero, but whose absolute value is the
       closest to the zero.
  Set that attribute value to zero.
  Increase current_zv_pct accordingly.
  Keep DS instances sorted.
}

Run regular training SVM algorithm on modified
       DS and return SVM model as the result.
```

The listed algorithm uses an ordinary SVM training algorithm as one of its steps. We will call a model, which the ordinary algorithm outputs on the original, unmodified dataset, dense SVM model. The SVM model, which is the output of the AST-SVM algorithm, will be called "pruned SVM model". The dense model can contain some zero attributes in its support vectors. Please notice that the pruned SVM model always contains less non-zero attributes compared to the dense SVM model. However, that does not mean that the pruned SVM model always contains exactly target_zv_pct less non-zero attributes compared to the dense SVM model. The reason for this is that modification of the training dataset will change, which training instances will become support vectors. For example, the dense SVM model can have 0 % of zeroes in its support vectors. The value of target_zv_pct can be set to 5 %. The modified training dataset will have 5 % attribute values of the training instances set to zero. Nevertheless, after the ordinary SVM training algorithm is run on the modified training dataset, the resulting pruned SVM model can contain only 4 % fewer attributes compared to the dense SVM model. The pruned SVM model can use more support vectors than the dense SVM model, but still has less non-zero attributes overall.

III. Hardware Accelerator for Sparse SVMs

The digital architecture, which can take advantage of the sparse SVM model, called "ASA-SVM" (Attribute Sparse Accelerator - SVM) is the modification of the "RMLC" (Reconfigurable Machine Learning Classifier) architecture, proposed in [20].

In the RMLC architecture, the SVM is implemented by splitting the sum calculation to several identical modules, each of which calculates only one part of the complete sum $V(x)$ as shown in Fig. 1.

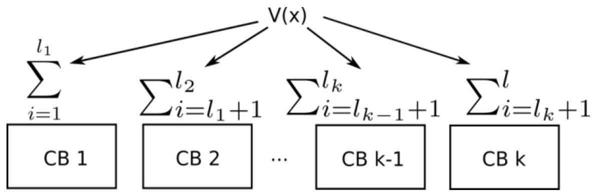

Fig. 1. Implementing SVM using RMLC architecture from [20].

In Fig. 1, there are K Computing Blocks (CB). Each CB calculates one part of the complete sum $V(x)$. CB 1 calculates sum up to l_1^{th} support vector. The partial sum it calculates is then passed to the CB 2 module, alongside with the current input instance. The CB 2 module then calculates the next part of the complete sum from $(l_1+1)^{th}$ support vector to l_2^{th} support vector and so on. The architecture is designed to achieve the high instance throughput by using pipelining. When the CB 1 module passes the partial sum of the first input instance to the CB 2 module, it immediately starts to calculate the partial sum using the next input instance. In the architecture with K CBs, up to K input instances can be processed at the same time.

The ASA-SVM architecture keeps only the SVM functionality part of the RMLC architecture and adds hardware support needed for the evaluation of sparse SVMs. The top-level block diagram of the ASA-SVM accelerator is shown in Fig. 2. The CBs are arranged in an array of identical pipeline stages. A configuration module (CM) reads the SVM configuration data from the main memory through the mm_rd interface and sends the data to a particular CB. The configuration for each CB module consists of three parts. The first part is stored inside the support vector memory (SV_M) and it contains support vectors and Lagrange multipliers for the part of the complete sum, which that CB calculates. The second part contains increments stored in INC_M memory, which determine which attributes of the support vectors and the input instance should be processed. This enables skipping all multiplications that use zero attributes from the support vectors. The third part of the configuration is the samples of the specified non-linear function needed for the evaluation of the selected kernel function.

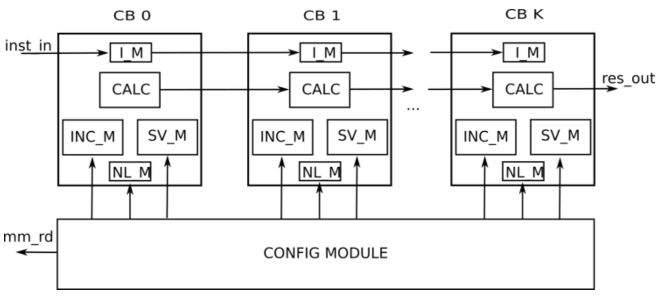

Fig. 2. Top-level block diagram of ASA-SVM accelerator.

In section II, the algorithm for the training of sparse SVMs was presented. The sparse SVMs have an advantage over ordinary SVM models since they contain fewer attributes with non-zero values. Let *frac* be the percentage of attributes, which are not zero. If the classification speed of the input instance is critical, then the sparse SVM models could be processed 1/*frac* times faster than the dense SVM models by skipping all product terms that use a zero-valued attribute. The hardware architecture needs to be designed to take advantage of this opportunity, and the ASA-SVM is the first SVM accelerator with this capability.

In Fig. 3, the skipping of product terms with zero-valued support vector attributes during kernel calculation is illustrated. Instead of storing the whole support vector in the memory, only non-zero values (NZV) are stored in the SV_M memory. Every NZV value corresponds to one increment value, which is stored in separate, INC_M memory. During the kernel calculation, for every NZV of the support vector, the corresponding value from the input instance is taken. All zero-valued attributes in the support vector are skipped during the kernel calculation evaluating the kernel faster. The second benefit of this technique is that usually less memory is needed for storing support vectors. Only the NZVs are stored with their corresponding increments. The increments can be coded with fewer bits. In the example, from Fig. 3, only 4 terms of the kernel calculation will be used instead of 12. Also, only 4 NZV will be stored, alongside with 4 increments.

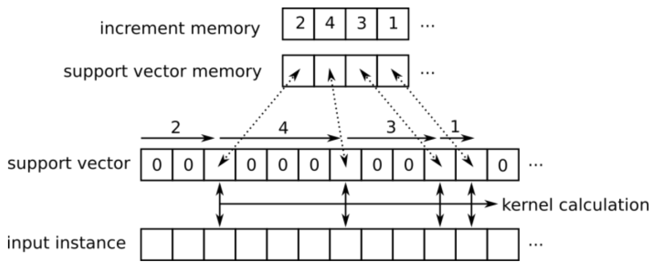

Fig. 3. Skipping in kernel calculation.

Each CB module is also pipelined. Figure 4 shows the architecture of the CB. A control unit reads the values from the increment memory and calculates the effective address of the attributes. The effective address is used to read the current attribute from the input memory. In the SV memory, NZV values of support vectors are stored. The NZVs are read sequentially. The data read from the input memory and SV memory are sent for further processing to the appropriate functional unit.

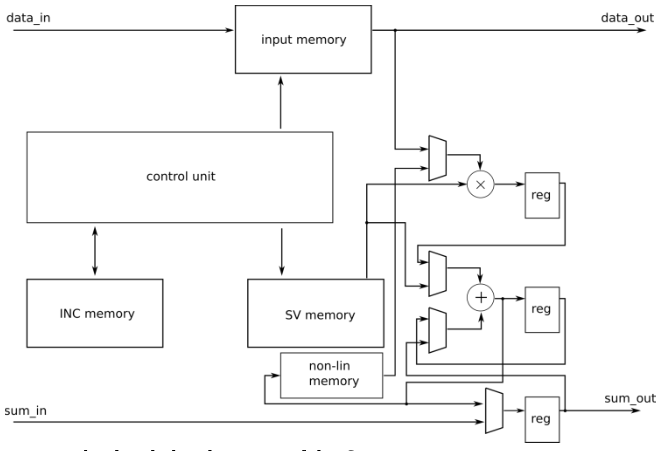

Fig. 4. The detailed architecture of the CB.

The control unit also controls the data selection multiplexers and enables signals of registers to configure pipelined data path required to implement the selected type of kernel needed for the classification. To perform the necessary computations, only one multiplier and adder are needed. After the vector calculation is finished, the final value is sent as an address to the non-lin memory. The value read from the non-lin memory is the kernel value for the current support vector and it is accumulated to the running sum after multiplication with the corresponding Lagrange multiplier stored in the SV memory. This process is then repeated for all support vectors stored in CB.

The RMLC architecture from [22] has a processing time of "RB" (Reconfigurable Block) – T_{RMLC}

$$T_{RMLC} = N_{sv}(N_{attr} + d)T_{clk}, \tag{14}$$

where N_{sv} is the number of SVs in a block, N_{attr} is the number of attributes in the SV, the value d is 4, depending on the type of kernel used in the SVM module, and T_{clk} is a period, on which the architecture works. The ASA-SVM architecture proposed in this paper has the processing time of one CB block - $T_{ASA-SVM}$

$$T_{ASA-SVM} = \sum_{i=1}^{N_{sv}} (N_{nzv}^{i} + d)T_{clk}, \tag{15}$$

where N_{nzv}^{i} is the number of NZVs in the ith SV of the CB. The value of N_{nzv}^{i} is always smaller than N_{attr}, so the processing speed of ASA-SVM is always greater than RMLC's.

The AST-SVM algorithm is designed to keep the number of NZVs evenly distributed along all support vectors. For every dataset, there is a threshold, when the maximum difference between N_{nzv}^{i} values will be 1. Let N_{nzv} be $\max(N_{nzv}^{i})$. Processing time of ASA-SVM architecture always satisfies the following condition

$$T_{ASA-SVM} \leq N_{sv}(N_{nzv} + d)T_{clk}. \tag{16}$$

The speed-up S of the ASA-SVM architecture, compared to RMLC, can be calculated as

$$S = \frac{T_{RMLC}}{T_{ASA-SVM}} \geq \frac{N_{attr} + d}{N_{nzv} + d}. \tag{17}$$

For large datasets, where $N_{attr} >> d$ and $N_{nzv} >> d$, the speed-up equation can be approximated as

$$S \approx \frac{N_{attr}}{N_{nzv}} = \frac{1}{frac}. \tag{18}$$

From the approximate equation for speed-up, it is clear that the efficiency of the ASA-SVM is directly proportional to the amount of pruning achieved during the training of the sparse SVM model.

IV. EXPERIMENTAL RESULTS

Xilinx Vivado Design Suite is used for the development of the ASA-SVM architecture. The default values for synthesis and implementation settings were set. Zynq Ultrascale+ MPSoC ZCU102 Evaluation Board [21] was a test platform, on which experiments were done.

The training of SVM models after pruning of the datasets was done using the LIBSVM library [22]. All standard procedures from the library were used with default parameters.

The ability of the AST-SVM algorithm to compress SVM classifiers was tested on several datasets from the LIBSVM dataset page [23]. The datasets of various sizes and characteristics were selected.

For every dataset, 19 pruning factors were used as inputs for the AST-SVM algorithm. The targeted pruning factors ranged from 5 % to 95 % with steps of 5 %. For every pruning factor, the dataset was pruned, and then the LIBSVM training procedure was called to create sparse SVM. The accuracy of the resulting sparse SVM model and its size was then determined. In all cases, if the dataset was split into training and validation sets, the training set was used for training, and the accuracy was measured on the validation dataset. In case when the dataset has not been split, the whole dataset was used for training, and also for the accuracy measurement. Table I presents the major characteristics of the datasets that were used in the experiments.

TABLE I. DATASETS' CHARACTERISTICS.

Dataset name	Short name	Attributes	Instances
Wisconsin Breast Cancer	bcancer	10	683
Pima Indians Diabetes	diabetes	8	768
Glass Identification	glass	9	214
Heart Disease	heart	13	270
Wine Recognition	wine	13	178
Mushrooms	mush	112	8124
USPS	usps	256	7291
Poker Hand	poker	10	25010
MNIST	mnist	780	60000
CIFAR10	cifar	3072	50000
SVHN	svhn	3072	73257

In the following Table II–Table XII and Fig. 5–Fig. 15, the results of the experiments are shown. The tables show how much it is possible to prune the SVM model trained using the AST-SVM algorithm on the given dataset. Also, real size reductions are shown.

TABLE II. BENCHMARKING RESULTS - BCANCER.

Target	ACC	NZV	ZV	Reduction	Speed-up	Memory
0	97.2182	814	0	1.00	1.00	1.25
0.05	97.0717	821	26	1.01	0.99	1.26
0.1	97.2182	750	75	0.92	1.06	1.15
0.15	97.0717	741	106	0.91	1.07	1.14
0.2	96.6325	710	159	0.87	1.10	1.09
0.25	96.3397	720	204	0.88	1.09	1.11
0.3	96.4861	648	243	0.80	1.17	1.00
0.35	96.3397	668	300	0.82	1.15	1.03
0.4	96.0469	623	356	0.77	1.20	0.96
0.45	96.1933	628	417	0.77	1.20	0.96
0.5	95.754	570	475	0.70	1.27	0.88
0.55	95.6076	561	539	0.69	1.29	0.86
0.6	95.4612	495	594	0.61	1.39	0.76
0.65	95.022	518	736	0.64	1.35	0.80
0.7	94.7291	495	869	0.61	1.39	0.76
0.75	94.2899	547	1213	0.67	1.31	0.84
0.8	94.7291	527	1409	0.65	1.34	0.81
0.85	91.9473	756	2797	0.93	1.05	1.16
0.9	76.2811	665	2998	0.82	1.15	1.02
0.95	89.1654	718	4551	0.88	1.09	1.10

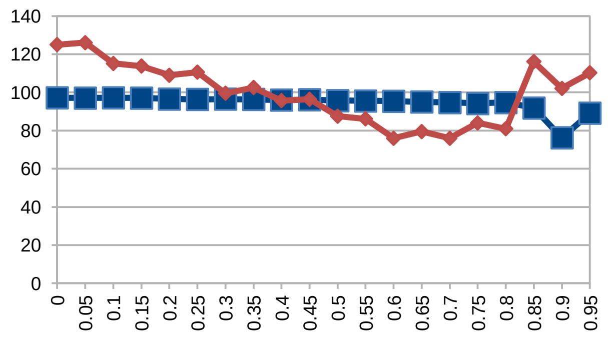

Fig. 5. Pruning-accuracy graph for bcancer.

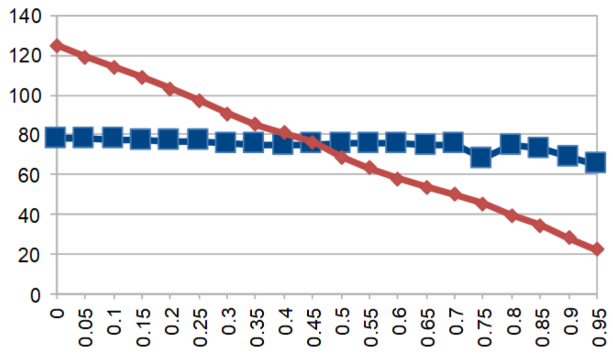

Fig. 6. Pruning-accuracy graph for diabetes.

TABLE III. BENCHMARKING RESULTS - DIABETES.

Target	ACC	NZV	ZV	Reduction	Speed-up	Memory
0	77.9948	4232	7	1.00	1.00	1.25
0.05	78.125	4040	190	0.95	1.03	1.19
0.1	77.474	3872	376	0.91	1.06	1.14
0.15	77.2135	3703	563	0.88	1.09	1.09
0.2	76.6927	3502	746	0.83	1.13	1.03
0.25	76.8229	3299	949	0.78	1.17	0.97
0.3	75.5208	3074	1111	0.73	1.22	0.91
0.35	75.2604	2892	1302	0.68	1.27	0.85
0.4	74.8698	2752	1541	0.65	1.31	0.81
0.45	75.2604	2584	1736	0.61	1.35	0.76
0.5	75.3906	2340	1881	0.55	1.43	0.69
0.55	75.5208	2144	2131	0.51	1.49	0.63
0.6	75.3906	1965	2274	0.46	1.56	0.58
0.65	74.6094	1826	2602	0.43	1.61	0.54
0.7	75	1700	2935	0.40	1.66	0.50
0.75	68.099	1540	3086	0.36	1.74	0.45
0.8	74.7396	1341	3483	0.32	1.84	0.40
0.85	73.1771	1167	3702	0.28	1.93	0.34
0.9	68.6198	949	3893	0.22	2.07	0.28
0.95	65.1042	751	4091	0.18	2.22	0.22

TABLE IV. BENCHMARKING RESULTS - GLASS.

Target	ACC	NZV	ZV	Reduction	Speed-up	Memory
0	59.3458	2334	536	1.00	1.15	1.02
0.05	56.0748	2234	608	0.96	1.18	0.97
0.1	58.8785	2139	703	0.92	1.21	0.93
0.15	60.2804	2059	797	0.88	1.24	0.90
0.2	56.5421	1973	883	0.85	1.28	0.86
0.25	61.6822	1885	985	0.81	1.31	0.82
0.3	52.8037	1799	1071	0.77	1.35	0.78
0.35	58.8785	1692	1164	0.72	1.40	0.74
0.4	60.2804	1629	1255	0.70	1.43	0.71
0.45	63.0841	1528	1342	0.65	1.48	0.67
0.5	53.7383	1432	1424	0.61	1.53	0.62
0.55	54.2056	1362	1550	0.58	1.57	0.59
0.6	50.9346	1259	1639	0.54	1.64	0.55
0.65	47.1963	1178	1720	0.50	1.69	0.51
0.7	43.9252	1072	1840	0.46	1.77	0.47
0.75	45.7944	1017	1909	0.44	1.81	0.44
0.8	45.3271	940	2056	0.40	1.87	0.41
0.85	35.514	852	2116	0.37	1.95	0.37
0.9	35.9813	751	2175	0.32	2.05	0.33
0.95	33.6449	645	2211	0.28	2.16	0.28

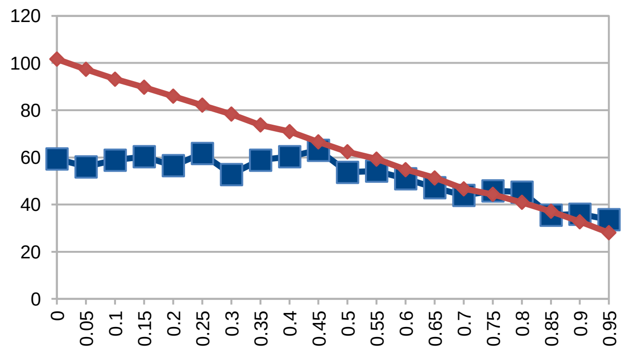

Fig. 7. Pruning-accuracy graph for glass.

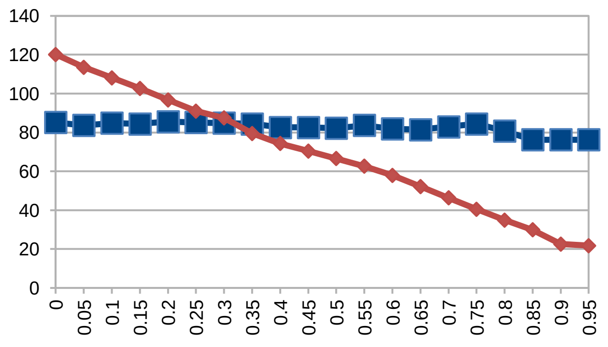

Fig. 8. Pruning-accuracy graph for heart.

TABLE V. BENCHMARKING RESULTS - HEART.

Target	ACC	NZV	ZV	Reduction	Speed-up	Memory
0	85.1852	1668	68	1.00	1.03	1.20
0.05	83.7037	1576	132	0.94	1.08	1.13
0.1	84.8148	1501	207	0.90	1.12	1.08
0.15	84.4444	1426	282	0.85	1.16	1.03
0.2	85.5556	1343	365	0.81	1.21	0.97
0.25	85.1852	1263	431	0.76	1.26	0.91
0.3	84.8148	1214	522	0.73	1.30	0.87
0.35	84.4444	1104	590	0.66	1.39	0.79
0.4	82.5926	1032	662	0.62	1.45	0.74
0.45	82.5926	978	758	0.59	1.50	0.70
0.5	82.2222	924	854	0.55	1.56	0.67
0.55	83.7037	870	964	0.52	1.62	0.63
0.6	81.8519	804	1072	0.48	1.70	0.58
0.65	81.4815	725	1151	0.43	1.80	0.52
0.7	82.963	643	1247	0.39	1.93	0.46
0.75	84.4444	562	1342	0.34	2.07	0.40
0.8	80.7407	485	1489	0.29	2.23	0.35
0.85	76.2963	415	1643	0.25	2.39	0.30
0.9	76.2963	314	1702	0.19	2.68	0.23
0.95	76.2963	302	2526	0.18	2.71	0.22

TABLE VI. BENCHMARKING RESULTS - WINE.

Target	ACC	NZV	ZV	Reduction	Speed-up	Memory
0	98.3146	1351	74	1.00	1.04	1.19
0.05	98.3146	1287	138	0.95	1.08	1.13
0.1	98.8764	1241	199	0.92	1.11	1.09
0.15	98.3146	1183	257	0.88	1.15	1.04
0.2	99.4382	1138	332	0.84	1.18	1.00
0.25	98.8764	1056	384	0.78	1.25	0.93
0.3	98.8764	1003	452	0.74	1.29	0.88
0.35	98.8764	939	501	0.70	1.35	0.82
0.4	97.7528	876	564	0.65	1.42	0.77
0.45	97.7528	824	646	0.61	1.48	0.72
0.5	97.7528	799	731	0.59	1.51	0.70
0.55	96.6292	750	825	0.56	1.57	0.66
0.6	96.0674	695	895	0.51	1.64	0.61
0.65	96.0674	629	946	0.47	1.75	0.55
0.7	94.9438	601	1079	0.44	1.79	0.53
0.75	96.0674	545	1165	0.40	1.89	0.48
0.8	93.2584	548	1462	0.41	1.89	0.48
0.85	92.1348	517	1673	0.38	1.95	0.45
0.9	94.382	502	1958	0.37	1.98	0.44
0.95	70.2247	424	2141	0.31	2.16	0.37

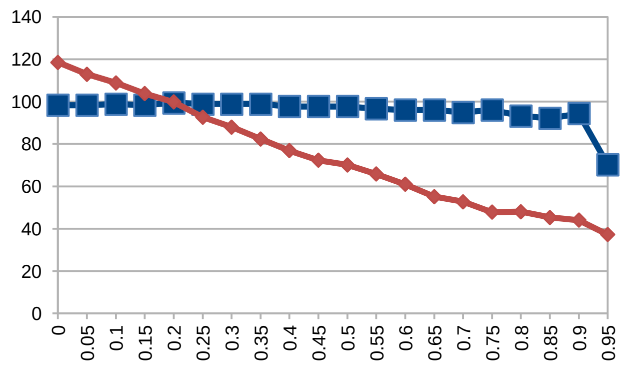

Fig. 9. Pruning-accuracy graph for wine.

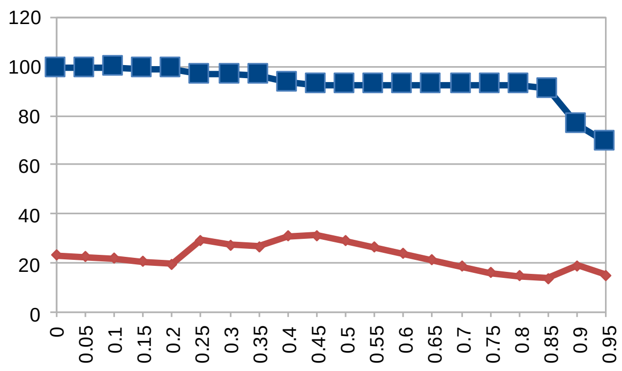

Fig. 10. Pruning-accuracy/size graph for mush.

TABLE VII. BENCHMARKING RESULTS – MUSH.

Target	ACC	NZV	ZV	Reduction	Speed-up	Memory
0	99.2122	17766	77832	1.00	4.67	0.23
0.05	99.2614	17120	79608	0.96	4.82	0.22
0.1	99.5076	16701	82626	0.94	4.92	0.22
0.15	99.0153	15732	83030	0.89	5.17	0.21
0.2	99.0153	14908	84193	0.84	5.40	0.19
0.25	96.8488	22336	135412	1.26	3.85	0.29
0.3	96.8488	21000	137200	1.18	4.06	0.27
0.35	96.4549	20551	145333	1.16	4.13	0.27
0.4	93.4023	23660	182000	1.33	3.66	0.31
0.45	92.6145	24023	202203	1.35	3.61	0.31
0.5	92.4175	22055	204510	1.24	3.89	0.29
0.55	92.4175	20129	207340	1.13	4.21	0.26
0.6	92.4175	18098	209145	1.02	4.60	0.24
0.65	92.4175	16184	212415	0.91	5.05	0.21
0.7	92.4175	14273	216134	0.80	5.60	0.19
0.75	92.4175	12329	219886	0.69	6.29	0.16
0.8	92.4668	11279	243649	0.63	6.74	0.15
0.85	90.9404	10431	284273	0.59	7.15	0.14
0.9	76.2186	14553	533610	0.82	5.51	0.19
0.95	69.03	11607	644245	0.65	6.59	0.15

TABLE VIII. BENCHMARKING RESULTS - USPS.

Target	ACC	NZV	ZV	Reduction	Speed-up	Memory
0	91.7289	668262	15438	1.00	1.02	1.22
0.05	91.5795	637712	48638	0.95	1.07	1.17
0.1	91.7788	607076	82189	0.91	1.12	1.11
0.15	91.7289	593198	119652	0.89	1.15	1.08
0.2	91.8784	563577	155368	0.84	1.21	1.03
0.25	91.6791	535909	192576	0.80	1.27	0.98
0.3	91.1809	525131	239129	0.79	1.30	0.96
0.35	89.3373	488840	276215	0.73	1.39	0.89
0.4	85.999	464114	321611	0.69	1.46	0.85
0.45	80.5182	446167	376658	0.67	1.52	0.82
0.5	76.8809	402136	411149	0.60	1.68	0.74
0.55	73.8416	396460	492880	0.59	1.71	0.72
0.6	71.4001	366015	554860	0.55	1.84	0.67
0.65	67.713	331885	617345	0.50	2.03	0.61
0.7	64.3747	302437	701118	0.45	2.22	0.55
0.75	60.289	268751	794429	0.40	2.48	0.49
0.8	56.2033	236489	920236	0.35	2.81	0.43
0.85	53.0144	203762	1097918	0.30	3.24	0.37
0.9	43.4479	161777	1319573	0.24	4.03	0.30
0.95	34.3797	102880	1517065	0.15	6.11	0.19

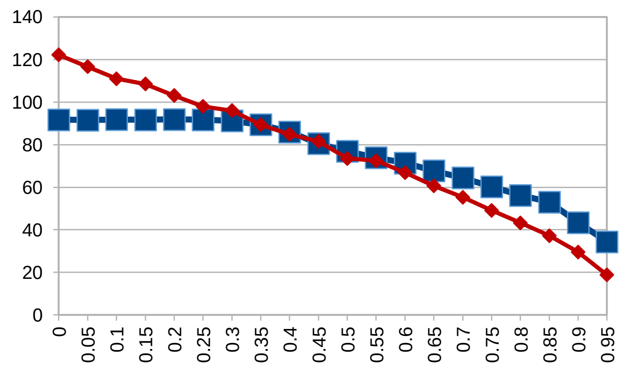

Fig. 11. Pruning-accuracy/size graph for USPS.

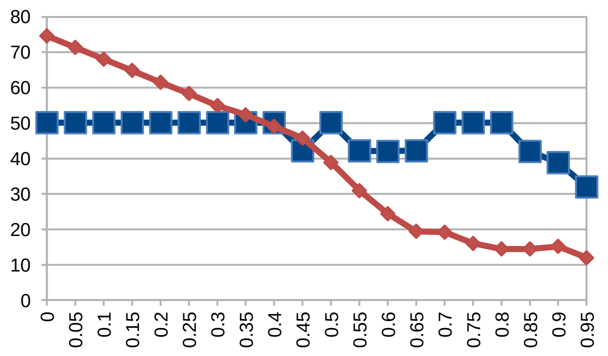

Fig. 12. Pruning-accuracy/size graph for poker.

TABLE IX. BENCHMARKING RESULTS - POKER.

Target	ACC	NZV	ZV	Reduction	Speed-up	Memory
0	50.1209	263110	177766	1.00	1.40	0.75
0.05	50.1209	251772	189237	0.96	1.44	0.71
0.1	50.1209	240094	201181	0.91	1.48	0.68
0.15	50.1209	228739	212593	0.87	1.52	0.65
0.2	50.1209	216969	224534	0.82	1.57	0.62
0.25	50.1209	205778	235972	0.78	1.62	0.58
0.3	50.1209	193716	247730	0.74	1.67	0.55
0.35	50.1209	184629	263258	0.70	1.71	0.52
0.4	50.1209	173009	276816	0.66	1.77	0.49
0.45	42.2479	161326	291083	0.61	1.83	0.46
0.5	50.1209	137262	276273	0.52	1.97	0.39
0.55	42.1724	109096	243696	0.41	2.16	0.31
0.6	42.0758	86266	220869	0.33	2.35	0.24
0.65	42.2498	68480	190110	0.26	2.52	0.19
0.7	50.1209	67689	224588	0.26	2.53	0.19
0.75	50.1209	56531	211977	0.21	2.65	0.16
0.8	50.1209	51010	224357	0.19	2.71	0.14
0.85	42.0069	51019	299227	0.19	2.71	0.14
0.9	38.8308	53604	381591	0.20	2.68	0.15
0.95	32.0346	42312	393415	0.16	2.82	0.12

TABLE X. BENCHMARKING RESULTS - MNIST.

Target	ACC	NZV	ZV	Reduction	Speed-up	Memory
0	93.13	3440972	14928526	1.00	5.22	0.23
0.05	93.19	3327687	15075738	0.97	5.40	0.23
0.1	93.17	3208746	15290148	0.93	5.59	0.22
0.15	93.14	3087787	15586265	0.90	5.80	0.21
0.2	93.1	2971934	15994048	0.86	6.02	0.20
0.25	93.04	2849282	16457548	0.83	6.27	0.19
0.3	92.94	2731426	17046437	0.79	6.53	0.19
0.35	92.66	2621558	17803285	0.76	6.80	0.18
0.4	92.23	2521740	18763902	0.73	7.06	0.17
0.45	91.91	2422939	19861701	0.70	7.34	0.16
0.5	91.65	2328891	21211033	0.68	7.62	0.16
0.55	91.31	2238182	22831250	0.65	7.92	0.15
0.6	90.81	2131474	24569118	0.62	8.30	0.15
0.65	90.28	2028023	26782833	0.59	8.70	0.14
0.7	89.47	1909403	29330857	0.55	9.22	0.13
0.75	87.18	1765278	32266078	0.51	9.93	0.12
0.8	83.78	1593953	35528837	0.46	10.94	0.11
0.85	76.85	1370830	38833718	0.40	12.60	0.09
0.9	71.66	1118770	42109334	0.33	15.22	0.08
0.95	52.75	851145	45191695	0.25	19.53	0.06

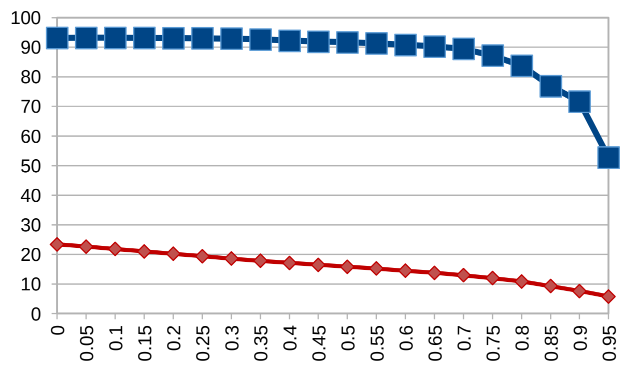

Fig. 13. Pruning-accuracy/size graph for MNIST.

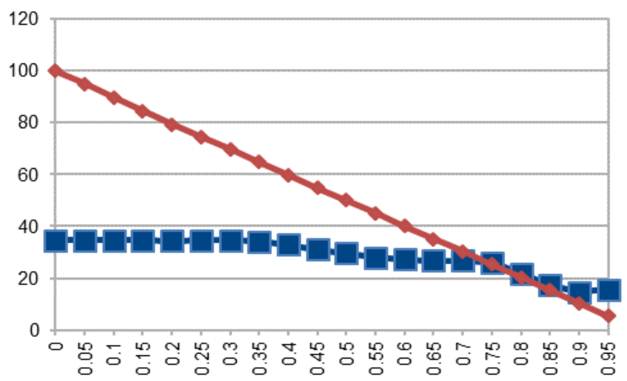

Fig. 14. Pruning-accuracy/size graph for CIFAR.

TABLE XI. BENCHMARKING RESULTS - CIFAR.

Target	ACC	NZV	ZV	Reduction	Speed-up	Memory
0	34.54	146841985	519164	1.00	1.00	1.25
0.05	34.52	139261741	7840604	0.95	1.06	1.18
0.1	34.66	131545686	15112995	0.90	1.12	1.12
0.15	34.69	124122919	22384793	0.85	1.19	1.05
0.2	34.36	116487853	29582357	0.79	1.26	0.99
0.25	34.57	109412240	36910612	0.75	1.35	0.93
0.3	34.71	102270755	44246200	0.70	1.44	0.87
0.35	34.09	95169529	51633959	0.65	1.55	0.81
0.4	32.94	87929910	58975251	0.60	1.67	0.75
0.45	31.13	80650407	66300969	0.55	1.83	0.68
0.5	29.61	73396121	73660009	0.50	2.01	0.62
0.55	28.03	66172232	81078001	0.45	2.22	0.56
0.6	27.22	58899163	88468148	0.40	2.50	0.50
0.65	26.75	51637803	95911287	0.35	2.85	0.44
0.7	26.79	44419751	103514464	0.30	3.31	0.38
0.75	26.03	37189070	111228862	0.25	3.95	0.32
0.8	21.81	29947728	119132619	0.20	4.90	0.25
0.85	17.2	22699792	127418852	0.15	6.45	0.19
0.9	14.61	15343490	135699454	0.10	9.50	0.13
0.95	15.64	7944104	144679393	0.05	18.14	0.07

TABLE XII. BENCHMARKING RESULTS - SVHN.

Target	ACC	NZV	ZV	Reduction	Speed-up	Memory
0	13.7754	205086232	191555	1.00	1.00	1.25
0.05	13.9098	194957987	10421473	0.95	1.05	1.19
0.1	14.2555	185743497	20763609	0.91	1.11	1.13
0.15	14.6512	176805075	31288746	0.86	1.16	1.08
0.2	15.7345	168223458	42101007	0.82	1.22	1.02
0.25	16.9215	160214044	53407091	0.78	1.28	0.98
0.3	18.5925	151307562	64803021	0.74	1.36	0.92
0.35	20.3058	141372009	76026432	0.69	1.45	0.86
0.4	20.8666	131534404	87530858	0.64	1.56	0.80
0.45	20.8052	121352380	99056198	0.59	1.69	0.74
0.5	20.2597	110768233	110447567	0.54	1.85	0.67
0.55	20.1099	99798629	121546573	0.49	2.05	0.61
0.6	20.179	88701366	132486705	0.43	2.31	0.54
0.65	20.3365	77667657	143501928	0.38	2.64	0.47
0.7	20.3288	66544984	154301096	0.32	3.08	0.41
0.75	20.3941	55487560	165164417	0.27	3.69	0.34
0.8	20.4057	44471245	176088302	0.22	4.59	0.27
0.85	20.2674	33512888	187262329	0.16	6.08	0.20
0.9	19.3838	22619933	199199743	0.11	8.98	0.14
0.95	18.5426	11648912	211649644	0.06	17.25	0.07

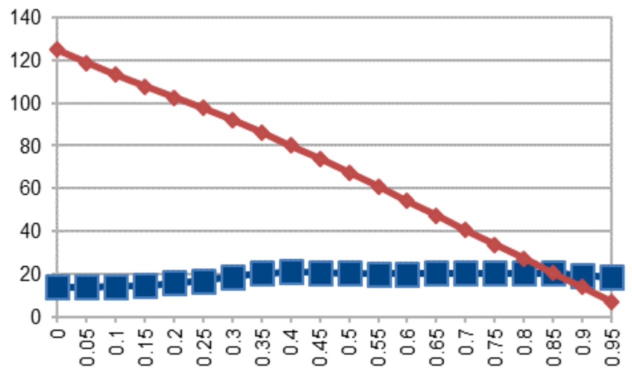

Fig. 15. Pruning-accuracy/size graph for SVHN.

The first column in the tables II–XII specifies the desired reduction in the size of the input dataset. The first row represents the SVM model trained without any pruning. The second column ("ACC") represents the accuracy of the trained SVM model. The third column ("NZV") presents the number of non-zero attributes in the support vectors, while the fourth ("ZV") contains the number of zero attributes introduced during the pruning process.

The fifth column ("Reduction") presents the relative size of the trained sparse SVM model compared to the original dense SVM model size shown in the first row of each table. This column shows how effective the pruning of SVM modules is when the AST-SVM algorithm. In an ideal case, the value of this column should be equal to 1 - "Target" value specified in the first column, but this is rarely the case. The reason for this is that when the input dataset is pruned, more support vectors are usually needed to achieve better accuracy as it was described in Chapter II. As Tables II-XII indicate, the reduction in the size of the trained SVM module is almost always possible. Please notice, when the reduction in trained SVM size is possible, it is always less than the specified target reduction value.

For some of the datasets, the reduction of the size of the trained SVM model can even help to achieve better classification accuracy. This is the case with the SVHN dataset, which is the biggest dataset in the experiments. In this case, the sparse SVM model can be almost 2 times smaller and achieve accuracy, which is over 7 % better than the accuracy of the dense SVM model. One explanation for this unexpected behavior could be that the input dataset contains a lot of noise in the data, which pruning helps to eliminate.

The sixth column shows the speed-up that is possible to achieve when using ASA-SVM architecture, compared to RMLC architecture, to implement the SVM model. With the reduction of trained SVM model size, fewer operations are needed to classify input instance. Every multiplication with the zero-valued support vector attribute can be skipped in the ASA-SVM accelerator, resulting in better performance and better energy usage. Please notice, that for some datasets, the original model also contains some zeros in its support vectors. In these cases, even without the pruning ASA-SVM architecture would enable speed-up over previously proposed RMLC architecture. From the Tables II–XII, it can be concluded that the speed-up between 1.17 and 7.92 is possible, taking into account the accepted classification accuracy reduction of 2 % during the SVM pruning process.

The seventh column shows achievable relative memory reduction, when only the NZVs of pruned SVM model are stored, together with their increment values, compared to memory size required to store the dense SVM model. Values greater than one indicate that memory usage is being increased, while values smaller than one represent a situation when memory footprint is decreased. In case when dense SVM model doesn't contain the significant number of zero-valued support vector attributes, and the SVM pruning percentage is close to zero percent, the memory size required to store SVM model parameters will increase. This is because every NZV value has to be stored together with its corresponding increment value. In this case, a significant number of support vector attribute values will be different from zero, and the increments will be just additional information needed to be stored without any benefit to speeding up the instance classification process. However, when the SVM model can be pruned with higher pruning rates, a significant reduction in memory will be possible. With the accepted trained SVM accuracy drop of 2 % compared to the dense SVM model, it can be concluded from the data shown in Tables II–XII that the SVM size memory reduction from 3 % up to 85 % is achievable.

V. CONCLUSIONS

This paper proposes a novel algorithm for training of sparse SVM models, called AST-SVM, and a hardware architecture for the acceleration of sparse SVM models, called ASA-SVM, which can take advantage of sparse SVMs and achieve better performance compared to the previously proposed hardware architectures that accelerate only dense SVMs.

Performed experiments, using 20 standard datasets, clearly indicate that using sparse over dense SVMs has two advantages, the memory size required to store the SVM model is reduced, and the input instances are processed faster.

When created SVM model size is considered, a reduction from 3% to 85% is possible, when using sparse instead of dense SVMs.

Additionally, the experiments clearly show that the sparse SVM models enable faster instance processing when using the hardware accelerator specifically designed to process sparse SVMs. Instance processing time speedup from 1.17 to 7.92 was reported on selected datasets.

CONFLICTS OF INTEREST

The authors declare that they have no conflicts of interest.

REFERENCES

[1] C. Cortes and V. Vapnik, "Support-vector networks", *Mach. Learn*. vol. 20, pp. 273–297, 1995. DOI: 10.1007/BF00994018.

[2] X.-X. Niu and Ch. Y. Suen, "A novel hybrid CNN-SVM classifier for recognizing handwritten digits", *Pattern Recogn.*, vol. 45, no. 4, pp. 1318–1325, Apr. 2012. DOI: 10.1016/j.patcog.2011.09.021.

[3] M. Elleuch, R. Maalej, and M. Kherallah, "A new design based-SVM of the CNN classifier architecture with dropout for offline Arabic handwritten recognition", *Procedia Comput. Sci.*, vol. 80, pp. 1712–1723, Jun. 2016. DOI: 10.1016/j.procs.2016.05.512.

[4] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge, Cambridge Univ. Press, 2000. DOI: 10.1017/CBO9780511801389.

[5] U. Kressel, "Pairwise classification and support vector machines", in *Advances in Kernel Methods: Support Vector Learning*. MIT Press, Cambridge, MA, USA, 1999, pp. 255–268.

[6] J. Platt, "Sequential minimal optimization: A fast algorithm for training support vector machines", Technical Report MSR-TR-98-14, Apr. 1998.

[7] R. Fan, P. Chen, and C. Lin, "Working set selection using second order information for training SVM", *Journal of Machine Learning Research*, vol. 6, pp. 1889–1918, 2005.

[8] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and J. Chen, "Compressing neural networks with the hashing trick", in *Proc. of International Conference on Machine Learning*, vol. 37, 2015, pp. 2285–2294. arXiv: 1504.04788.

[9] S. Han, H. Mao, and W. J. Dally. "Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding", in *Proc. of 4th International Conference on Learning Representations, ICLR 2016*, San Juan, Puerto Rico, May 2016. arXiv: 1510.00149.

[10] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural network", in in *Proc. of NeurIPS 2015 Twenty-ninth Conference on Neural Information Processing Systems*, 2015, pp. 1135–1143. arXiv:1506.02626.

[11] F. N. Iandola *et al*., "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and < 0.5 mb model size", in *Proc. of 5th International Conference on Learning Representations*, 2017. arXiv:1602.07360.

[12] S. S. Keerthi, O. Chapelle, and D. DeCoste, "Building support vector machines with reduced classifier complexity", *Journal of Machine Learning Research*, vol. 7, pp. 1493–1515, 2016.

[13] C. J. Burges and B. Scholkopf, "Improving the Accuracy and Speed of Support Vector Machines", in *Advances in neural information processing systems* 1997 pp. 375-381.

[14] D. Anguita, A. Boni, and S. Ridella, "A digital architecture for support vector machines: Theory, algorithm, and FPGA implementation", *IEEE Transactions on Neural Networks*, vol. 14, no. 5, pp. 993–1009, 2003. DOI: 10.1109/TNN.2003.816033.

[15] D. Anguita, L. Carlino, A. Ghio, and S. Ridella, "A FPGA core generator for embedded classification systems", *Journal of Circuits, Systems, and Computers*, vol. 20, no. 2, pp. 263–282, 2011. DOI: 10.1142/S0218126611007244.

[16] M. Davood, A. Soleimani, H. Khosravi, and M. Taghizadeh, "FPGA simulation of linear and nonlinear support vector machine", *Journal*

of Software Engineering and Applications, vol. 4, no. 5, pp. 320–328, 2011. DOI: 10.4236/jsea.2011.45036.

[17] M. Papadonikolakis and C. Bouganis, “Novel cascade FPGA accelerator for support vector machines classification”, *IEEE Transaction on Neural Networks Learning Systems*, vol. 23, no. 7, pp. 1040–1052, 2012. DOI: 10.1109/TNNLS.2012.2196446.

[18] C. Kyrkou and T. Theocharides, “A parallel hardware architecture for real-time object detection with support vector machines”, *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 831–842, 2012. DOI: 10.1109/TC.2011.113.

[19] V. Vranković and R. Struharik, “New architecture for SVM classifier and its application to telecommunication problems”, in *Proc of. 19th Telecommunications Forum (TELFOR)*, Belgrade, 2011, pp. 1543–1545. DOI: 10.1109/TELFOR.2011.6143852.

[20] V. Vranjkovic, R. Struharik, and L. Novak, “Reconfigurable hardware for machine learning applications”, *Journal of Circuits, Systems, and Computers*, vol. 24, no. 5, 2015. DOI: 10.1142/S0218126615500644.

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

[22] “LIBSVM - A Library for Support Vector Machines”. [Online]. Available: https://www.csie.ntu.edu.tw/~cjlin/libsvm/

[23] “LIBSVM Data: Classification, Regression, and Multi-label”. [Online]. Available: https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/