A high-performance FPGA architecture for Acceleration of SVM Machine Learning Training

Charalampos Kardaris
Department of Electrical and Computer Engineering
NTUA, Athens, Greece
ckardaris@outlook.com

Christoforos Kachris
Institute of Communications and Computer Systems
NTUA, Athens, Greece
kachris@microlab.ntua.gr

Dimitrios Soudris
Department of Electrical and Computer Engineering
NTUA, Athens, Greece
dsoudris@microlab.ntua.gr

***Abstract*—Support Vector Machines (SVMs) is one of the most popular machine learning algorithms as it provides high-performance and needs minimal tuning. It can be used for classification, regression and other learning tasks. Its design makes the training of large datasets time-consuming, so it is important that solutions are developed that attempt to tackle this problem. One such solution is presented in this paper. Based on the LIBSVM library, one of the most popular and widely used implementations of SVMs, we developed the design that can be executed on an heterogeneous computing system, consisting of an FPGA accelerator card, along with CPU.**

The first step of the process is to determine the most computationally intensive and parallelizable part of the algorithm. That is found to be a matrix row computation, involving inner products and a mathematical function (one of exp, pow, tanh) for each of the elements. Then, we develop a kernel, that can fully utilize the abilities of the FPGA accelerator card, regarding data processing and computations. In our case that is the Alveo U200 Data Center accelerator card. The results show that the proposed implementation achieves up to 14x speedup in the row computation part of the algorithm compared to a multi-threaded CPU execution on a Ryzen3 2200G, a CPU with a base clock speed of 3.5 GHz.

***Index Terms*—machine learning, FPGA acceleration, support vector machines, SVM, LIBSVM, high level synthesis**

I. Introduction

Machine Learning is one of the biggest trends of our technological era. The optimization efforts in the whole field are based, as is usually the case, in two pillars. Development of better algorithms and acceleration of currently established ones. The efforts presented in this paper, take the second route. This paper describes the process of accelerating SVM algorithms using an efficient FPGA architecture and provides a performance evaluation, demonstrating the benefits of such implementation.

The main contributions of the paper are the following:

- A thorough profiling and analysis of the most time-consuming functions of SVM algorithms
- A high-performance implementation of the SVM algorithm utilizing the hardware resources of the FPGAs
- A detailed performance evaluation and assesment of the proposed scheme and comparison with CPU-based implementations.

979-8-3503-9958-5/22/$31.00 ©2022 IEEE

II. LIBSVM

The SVM algorithm was introduced as a method to solve two-class classification problems. Different formulations of the initial algorithm have been proposed in order to perform multi-class classification, regression analysis and other learning tasks. The LIBSVM library supports a number of these formulations:

- C-Support Vector Classification
- ν-Support Vector Classification
- Distribution Estimation (One-class SVM)
- ϵ-Support Vector Regression (ϵ-SVR)
- ν-Support Vector Regression (ν-SVR)

Each of the above is a quadratic minimization problem.
For example C-SVC is defined as follows.
Given training vectors $\boldsymbol{x}_i \in \mathbb{R}^n, i = 1, ..., l$, in two classes, and an indicator vector $\boldsymbol{y} \in \mathbb{R}^l$ such that $y_i \in \{1, -1\}$, C-SVC solves the following primal optimization problem.

$$\begin{aligned} \min_{\boldsymbol{w},b,\boldsymbol{\xi}} \quad & \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{l}\xi_i \\ \text{subject to} \quad & y_i(\boldsymbol{w}^T\phi(\boldsymbol{x}_i) + b) \geq 1 - \xi_i, \\ & \xi_i \geq 0, i = 1, \ldots, l \end{aligned} \tag{1}$$

where $\phi(\boldsymbol{x}_i)$ maps $\boldsymbol{x}_i$ into a higher-dimensional space and $C > 0$ is the regularization parameter. Due to the possible high dimensionality of the vector variable $\boldsymbol{w}$, usually we solve the following dual problem.

$$\begin{aligned} \min_{\boldsymbol{a}} \quad & \frac{1}{2}\boldsymbol{a}^T Q\boldsymbol{a} - \boldsymbol{e}^T\boldsymbol{a} \\ \text{subject to} \quad & \boldsymbol{y}^T\boldsymbol{a} = 0, \\ & 0 \leq a_i \leq C, \quad i = 1, \ldots, l \end{aligned} \tag{2}$$

where $e = [1, \ldots, l]^T$ is the vector of all ones, Q is an $l \times l$ positive semi-definite matrix, $Q_{ij} \equiv y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ and $K(\boldsymbol{x}_i, \boldsymbol{x}_j) \equiv \phi(\boldsymbol{x}_i)^T\phi(\boldsymbol{x}_j)$ is the kernel function. After

TABLE I
PROFILING OF SVM-TRAIN

functions \ datasets	a9a	skin	ijcnn1	w8a	Avg.
dot	79.03	37.41	63.71	72.9	63.26%
kernel_rbf	8.18	21.63	11.66	12.11	13.4%
get_Q	4.51	20.95	10.29	5,65	10.35%
select_working_set	5.31	14.45	8.31	5.08	8.29%

problem (2) is solved, using the primal-dual relationship, the optimal w satisfies

$$\boldsymbol{w} = \sum_{i=1}^{l} y_i a_i \phi(\boldsymbol{x}_i) \tag{3}$$

and the decision function is

$$sgn(\boldsymbol{w}^T \phi(\boldsymbol{x}) + b) = sgn\left(\sum_{i=1}^{l} y_i a_i K(\boldsymbol{x}_i, \boldsymbol{x}) + b\right)$$

The definitions of the other SVM formulations can be found in [1]. The main difficulty of solving such problems is that Q may be too large to be stored. To address that, the LIBSVM library implements a decomposition method called Sequential Minimal Optimization (SMO), which requires the solution of a simple two-variable problem for each iteration. The library implementation also makes use of other techniques that further aid in the management of large datasets without any need for optimization software (i.e. caching, shrinking).

III. Software Analysis

FPGAs have the benefit that they are able to execute highly parallel code, due to the fact that they can place on the hardware chip and utilize many instances of the same compute module (e.g an adder or a multiplier). In order to fully take advantage of the capabilities of FPGA design, we first need to determine the part of the code that is best suited for FPGA execution. The characteristics of that code segment need to be:

- computationally intensive
- highly parallelizable

Both are important to be present. It has to make sense to dispatch the execution of the code to the FPGA, as in most cases there is some overhead in doing so. Dispatching code that is not that computationally intensive would most certainly result in slower execution.

A. Original Code

Our implementation is based on version 3.24 of the LIBSVM library.

B. Profiling

We used the `GNU gprof` utility in order to get a timing report from various executions of the `svm-train` program using different datasets. The results are presented in Table I.

C. Hardware Function Selection

The function that takes most of the execution time is the dot product function named `dot`. Nevertheless, this function is not well suited to be executed on the FPGA on its own. The profiles reported it being called a huge number of times, with each execution not taking almost zero time. The overhead of dispatching it to the FPGA and getting back the results would be too big. Upon further inspection of the report and the source code, we detected the following call stack: `get_Q() -> kernel() -> dot()`.

The fact that all calls of the `dot` function are coming for the `get_Q` function mean that the latter is a better candidate for hardware acceleration. The relevant code inside `get_Q` is as follows.

```
for(j = start; j < len; j++) {
    data[j] = (float)(y[i]*y[j]*
        (this->*kernel_function)(i,j));
}
```

The existence of this loop with count `len - start` (with `len` in the order of the number of training vectors) and a variable `i` that stays constant throughout creates for a nice setting for FPGA execution. What this code actually does is compute row i of matrix Q presented in Equation 2 of Section II. The loop starts from index `start`, because values preceding that and starting from index `0` are cached in software and are available instantly. More information about the way this is done exists in [1] and the source code is straightforward, as well.

IV. Accelerator Design

In order to explain the design choices that we made we have to take into account:

- the mechanics of the original algorithm
- the capabilities provided to us by utilizing the Alveo™ U200 accelerator card
- the aspect of retaining the existing functionality

Expanding the last item, our goal from the beginning was to alter the original code as little as possible. Our attempt was to accelerate the original algorithm using the available tools and not make modifications that would may give significant speedups, but would alter core parts of it. The results of our FPGA version would have to match the results of the original software version. We were able to produce two versions of the kernel code that are doing exactly that. The first version stores data in the FPGA global memory in `double` format, exactly like the original code, and the second version stores them in `float` format. The first version produces identical results to the original software, while the second version trades some accuracy with speed (more on the design specifics can be found in Subsection IV-B and on the performance results in Section V). We considered other versions, looking to further exploit this trade-off, but the loss in accuracy was deemed too much, which would be only be avoided with a whole restructuring of the original code. This effort

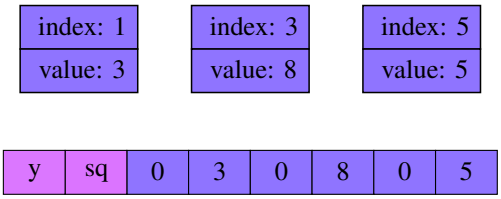

Fig. 1. Linked list to array conversion - 6 feature training vector

would also require a more detailed accuracy analysis of the training, something that was out of the scope of our work. Our implementation does only one basic change to the original source code (along with some other necessary additions in order to facilitate that change), which is to substitute the code of the loop in Section III-C, with a call to our `Host` function called `callRowKernel` and add 3 function calls needed to translate the logic of the original program for FPGA execution.

The design choices will be presented in the following manner. We will provide the relevant information regarding the original implementation in software and the capabilities of the accelerator card and then explain our design choices. Subsection IV-A will deal with the `Host` side code and Subsection IV-B with the `Kernel` side code.

A. Host Code

The first thing we need to address is the way the training vector data was stored in the original code. The authors, in that version of the code, had decided to put more weight on sparse data. This lead to the usage of linked lists to store the data, with each element storing the `index` and `value` of the non-zero dimensions of the training vectors. Consequently, the `dot product` function would have to traverse the relevant linked lists. The need for repetitiveness inside the `Kernel` code made us abandon that way and instead store the data inside the `DMA global memory` of the accelerator card in sequential fashion using arrays, padding with zeros when needed. Also, to cover the case of an SVM formulation using classification labels and maybe using the `RBF Kernel` we added 2 elements at the beginning of our arrays. The first storing y_i and the second storing the sum of squares $\sum_{i=0}^{dimensions} x_i^2$ for the given training vector (see Figure 1).

One of the major advantages of using the Alveo™ U200 is the fact that its `global memory` has 4 banks. This permits `RW` operations to happen from 4 different kernels at the same time. To properly use this feature we divided the training vector data in 4 arrays to be stored in the `global memory`.

Another major advantage is the `512-bit` width of the transfer bus between the FPGA and the `global memory` of the accelerator card. This makes possible `512-bit` transfers in one clock cycle. In our algorithm, this translates to the transfer of 8 `doubles` per clock cycle on the first version or 16 `floats` in the second version. To accommodate this feature, we extend the dimensions of the training vector to a multiple of 8 (or 16 respectively), padding with zeros when

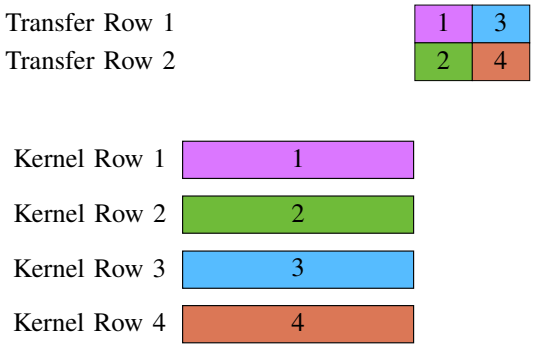

Fig. 2. Initial approach to parallel kernel execution

needed. That way, transfers in the `Kernel` code would never cross training vector dimension boundaries.

Without explaining how the `Kernel` code works (see Subsection IV-B), we need to explain what it does, so that we can describe the process of calling it in the `Host` side. Each of the 4 kernels is assigned to one of the 4 banks of the `global memory`. Each kernel takes as input parameters the training vector i (supposing we are performing the computation of row i of matrix Q), a `start` parameter (denoting the first vector in the bank that we want to compute the SVM kernel function for - this is closely related to the `start` variable of the original code) and a `products` parameter (denoting the number of training vectors that we want to compute the SVM kernel function for, starting beginning from `start` and return the results). Carefully selecting these parameters for the 4 kernels we can partition the `len - start` computation of the original loop into more chunks that can be executed in parallel, thus achieving a good speedup.

The initial approach was to partition the computation in 4 chunks, one for each kernel. This was fairly logical to do, since we can only execute up to 4 kernels in parallel. The attempt was successful for small datasets, but for large datasets the overhead of transferring the results from the FPGA global memory back to the host after the kernels had finished executing was not permitting the acceleration that we had anticipated (see Figure 2).

To tackle this problem we further partitioned the computation designated for each kernel into more chunks. That way the transfer of the results happens in parallel with the next kernel execution in line (see Figure 3).

The final item we had to address from the original source code was `swapping`. The algorithm at various points during the execution performs a `Shrinking` procedure. This procedure marks some training vectors as unnecessary for the rest of the training process. To do that it swaps these vectors with others from the "end" of the list and reduces the'working length". Using a linked list makes this quite easy, by manipulating their pointers. What we had to do to update the data in the `global memory` of the accelerator card, was keep track of those swaps, and before a new row computation was needed, make the necessary changes to the arrays and migrate them to the `global memory`, replacing the old

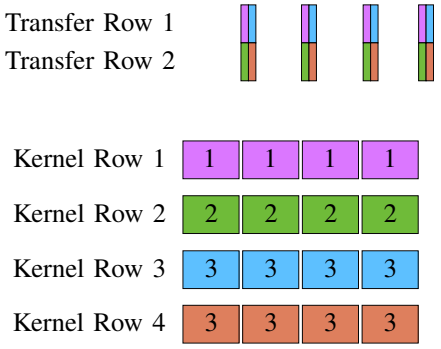

Fig. 3. Final approach to parallel kernel execution

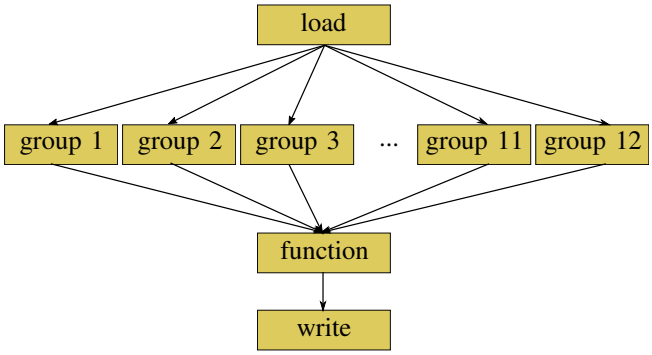

Fig. 4. Kernel dataflow

sequence of values. This creates some overhead, compared to the original software version, but it happens so rarely that it doesn't affect the acceleration of the training.

B. Kernel Code

The functionality of the code was first mentioned in Subsection IV-A. In a little more detail, what each kernel has to do is start reading the training vector values from the global memory, do the necessary computations and return the results back to the global memory. From there they are transferred back to the host.

This process of reading from the memory and writing back to it, and considering that only one DMA transfer operation can be happening at any given time (read or write), sets a lower bound regarding the time complexity of the whole task. As mentioned before, the `data bus` is able to transfer 8 `doubles` or equivalently 16 `floats` per clock cycle. We took advantage of this feature, not only for the reading process, but for the writing process as well. The LIBSVM algorithm expects float values to be returned, so we make `writes` of 16 `floats` at a time, so as to not disrupt the reading process that much. The resulting latency of our implementation is approximately $n * d + \frac{n}{16}$ cycles, where n is the number of training vectors to compute the SVM kernel function for and d is the number of dimensions divided by 8 (or 16 in the case of the second version), as they were selected in the `Host` side code. For example, to compute a row of 1 million elements of 32 dimensions each, each kernel would take a little more than 4.0625 (or 2.0625 for the second version) million clock cycles.

In order to achieve this result, where the actual time of execution is only determined by the time needed to read the data and write the results back to the global memory, we made use of the `pipeline` and `dataflow` directives. In the code these are specified using `#pragma HLS PIPELINE` and `#pragma HLS DATAFLOW` in the required areas. The `pipeline` directive permits the parallel execution of a loop's body, starting each iteration as soon as possible, even before the previous one has finished. The `dataflow` directive, along with the usage of `hls::streams`, facilitates the passing of data from one hardware module to the next (in code terms, output from one function that is input to another), before the first module has finished its execution. To better explain the `dataflow` directive usage we will describe the route of data through the hardware modules in our design. Figure 4 provides a visual representation of the functions/modules defined in each kernel.

Module `load` reads data from the `global memory`; 8 `doubles` or 16 `floats` on every clock cycle. We know the total dimensions for every training vector, so we know how to group those reads per training vector. Every clock cycle, each of the 8 (or 16) values is multiplied with the corresponding value from the `base vector` (the vector with index i), a tree-style addition of the 8 (or 16) results starts (3 levels or 4 levels respectively) and the result of these is streamed to one of the 12 `group` modules. These functions start adding the incoming values in sequential manner in groups of size $\frac{dimensions}{8}$ (or $\frac{dimensions}{16}$). These sums are the dot products for every training vector with the `base vector`. The sum is streamed to the `function` module, which computes the SVM kernel function. From there the results are streamed one by one to the `write` module, which groups them by 16 and writes them back to the `global memory`.

The interesting part of the dataflow model is the presence of the 12 `group` modules. In our attempt to make the kernels as general as possible in order to be able to train datasets with different number of features using the same FPGA bitstream, we had to abandon the notion of a tree-style addition for the $\frac{dimensions}{8}$ (or $\frac{dimensions}{16}$) values coming from the `load` module, as the Vivado HLS compiler would simply not pipeline the whole process, not being able to determine the depth of the tree at compile time. For that reason, a sequential addition process was selected. The problem was that each addition takes more clock cycles to be completed, while the `load` module is pipelined and can produce an output value in every clock cycle. In order to not have the values of the following training vectors wait idle in a queue and stall the whole pipeline, we thought about feeding these values to a different module. Each of these `group` modules is computing the dot product for a different training vector. The trick is that by the time the first value of the 13th training vector is ready to passed to a `group` module, the addition process of the 1st will have been completed. The math that supports this is simple.

TABLE II
RESOURCES PER KERNEL - DOUBLE VERSION

Resource	Used	Available	Utilization
BRAM_18K	278	319	87.1%
DSP48E	426	1132	37.6%
FF	93777	361686	26%
LUT	62764	177415	35.4%
URAM	8	80	10%

TABLE III
RESOURCES PER KERNEL- FLOAT VERSION

Resource	Used	Available	Utilization
BRAM_18K	242	319	75.9%
DSP48E	391	1132	34.5%
FF	83087	361686	23%
LUT	53867	177415	30.4%
URAM	8	80	10%

The compile reports showed that each addition takes at most 12 cycles to be completed. In that case the dot product of a training vector with the base vector takes $12 \cdot \frac{dimensions}{8}$ (or $12 \cdot \frac{dimensions}{16}$) cycles to be completed. In that time, the `load` module has produced exactly that many values, corresponding to 12 different training vectors. At the end, the number of cycles needed to make an addition dictated the number of modules needed to compute the dot products in parallel.

Up to now, we have only described the process of computing the dot product, but the data stored for each training vector contain also its label y and its sum of squares sq. These, after being multiplied and added respectively with the corresponding values of the base vector, are streamed directly from the `load` module to the `function` module. The latter computes one of the following SVM kernel functions:

- Linear: $y \cdot dot$
- Polynomial: $y \cdot (gamma \cdot dot + coef)^{degree}$
- RBF: $y \cdot e^{-gamma \cdot (sq - 2 \cdot dot)}$
- tanh: $y \cdot tanh(gamma \cdot dot + coef)$

where $gamma, degree, coef$ are SVM parameters.

C. FPGA Resources

The only parameter that can be changed and affects the resource usage of the kernels is the maximum number of dimensions per training vector supported. For `MAX_DIMENSIONS = 8000` the resource usage for each kernel is shown in Table II for the double version and Table III for the float version.

V. PERFORMANCE EVALUATION

The objective of this section is to present the speedups achieved using our implementation compared to the original software version. The FPGA speedups are relative to a multi-threaded execution on the CPU that utilizes all 4 available cores. We try to explore how the training set size and the number of features affect these speedups.

In order to do that, we did a grid-like exploration, defined by different training set sizes and number of features. To achieve that in an objective manner we created some custom datasets. The original dataset was Epsilon. This is a dense dataset that

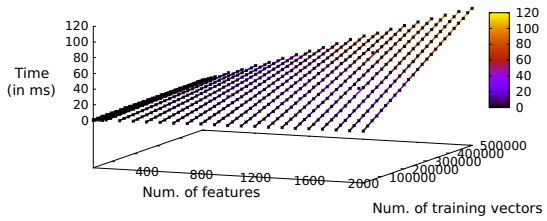

Fig. 5. Double version: How the training size affects the execution time

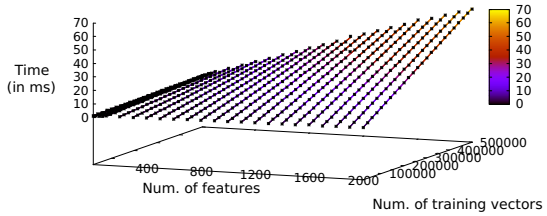

Fig. 6. Float version: How the training size affects the execution time

contains 400000 training vectors with 2000 features each. What we did was a take a subset of this dataset with 20000 training vectors. For each number of features that we wanted to test (5, 10, 25, 50, 75, 100, 200, ..., 2000), we removed the features that we didn't need and we copied this new base set of 20000 training vectors many times in order to create new custom datasets with (20000, 40000, ..., 500000) training vectors. The final step was to measure execution times for 1 row computation on the CPU and the FPGA.

Figures 5, 6 and 7 contain information about the execution time measured for every set of parameters (number of dimensions and training size) for the 3 different versions we checked (double FPGA version, float FPGA version and multi-threaded CPU version). The black points denote the actual measurements and the lines are fitted to the data in each case. The fitting of the lines is almost perfect for all 3 versions, with the multi-threaded one having only some minor deviances, that are mostly created by measurement accuracy errors. There is a linear relationship between the number of features, the training

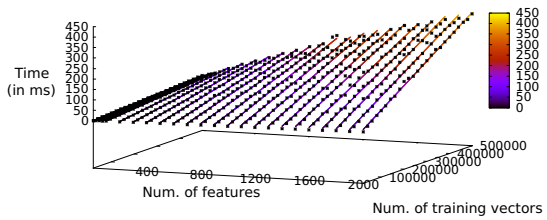

Fig. 7. Multiple threads: How the training size affects the execution time

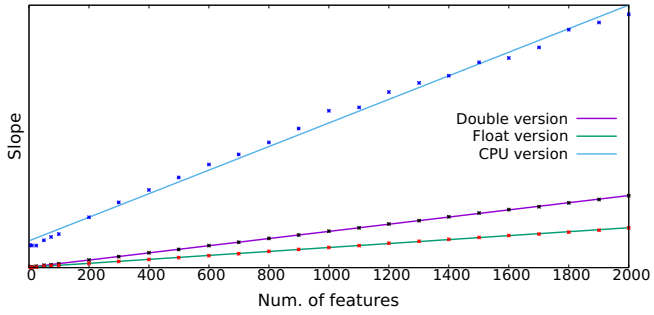

Fig. 8. Slope of execution time lines by number of features

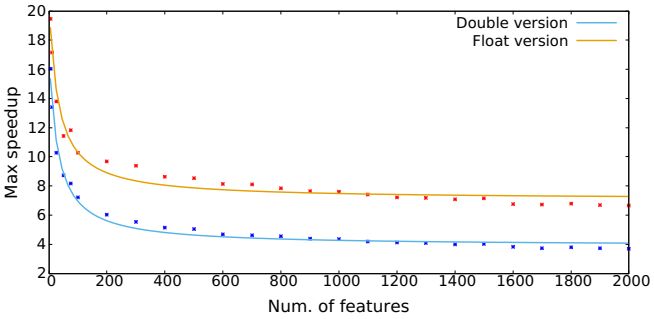

Fig. 9. Maximum speedup by number of features

size and the actual execution time of one row computation.

We can also observe that the maximum speedup that can be achieved varies different for every different number of features. In Figure 8 we have the graphs of the functions of the slope of the lines of Figures 5, 6, 7, that show the execution time of the different version we measured. In the same way that the functions of speedup for every number of features are fractions of the linear functions of execution time, the function of the maximum speedup by number of features (see Figure 9) is a fraction of these linear functions of the slopes. This relationship defines the shape of the graph in Figure 9 (again it's a hyperbola), that shows that for small number of features the maximum speedup is higher. The limits are about 3.5x and 7x for the double and the float version respectively.

VI. Related Work

To the best of our knowledge and according to the list of "Interfaces and Extensions to LIBSVM" available on the library website, there has not been implemented a direct extension to LIBSVM, utilizing FPGAs. Of course, Machine Learning training using Support Vector Machines is a very popular research field, so there exist many other SVM implementations on FPGAs.

Sequential Minimal Optimization is not well-scalable for huge data applications. In [2] Stochastic Gradient Descent is used as an alternative. This work also experiments with both single-precision floating point and fixed-point(5 bits integer and 20 bits fractional part) numerical representations. Their speedups seem to be very high, but their limitation is the low number of features supported by their design.

Instead of replacing SMO altogether there have been efforts to improve it for hardware acceleration. One of the disadvantages of the conventional SMO implementation used in LIBSVM is the need of data from only 2 row computations in each iteration. Caching further reduces this amount at times and only one new row computation is needed per iteration. This prevents the parallelization of more computations in the FPGA and thus it is technically a bottleneck of the original algorithm. The work in [3] addresses this limitation, by creating a variant of SMO called Hybrid Working Set (HWS), that creates working sets of bigger size of which the computations are grouped in columns, thus increasing the spatial locality of data.

The SVM kernels supported in LIBSVM are not all well-tailored for parallel hardware execution. Functions such as exp and $tanh$ do not exploit all the capabilities of reconfigurable architecture. In [4] an implementation is proposed utilizing the Hardware Friendly Kernel (HFK). As it name implies, this kernel is better suited for hardware parallelization, having the advantage of being able to be computed with only shifts and additions rather than multiplications. This work also produces exciting speedups, but is again limited by the supported number of features (up to 64) of their design.

VII. Conclusion

In this paper, we present the results of our work in an attempt to accelerate the LIBSVM library for Machine Learning training on FPGAs. Extra care was taken in order to parallelize the FPGA kernel code to a point where the only bottleneck was memory transfer operations to and from the DMA memory of the accelerator card, a bottleneck that could not be avoided, since, by providing support for training of a large amount of data, we could not store the necessary values in the space restricted FPGA local memory. The timing experiments show that, compared to a multithreaded CPU execution on a Ryzen™ 3 2200G, a CPU with a base clock speed of 3.5 GHz, we can achieve speedups of about 7x in the general case of our fastest version and up to 14x in some edge cases. This edge cases refer to datasets with few number of features and a substantial number of training vectors.

References

[1] Chih-Chung Chang and Chih-Jen Lin, LIBSVM: a library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm

[2] Felipe Fernandes Lopes, João Ferreira and Marcelo Fernandes. Parallel Implementation on FPGA of Support Vector Machines Using Stochastic Gradient Descent. Electronics. 8. 10.3390/electronics8060631, 2019.

[3] Sriram Venkateshan, Alap Patel and Kuruvilla Varghese. Hybrid Working Set Algorithm for SVM Learning With a Kernel Coprocessor on FPGA. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on. 23. 2221-2232. 10.1109/TVLSI.2014.2361254, 2015.

[4] Daniel Holanda Noronha, Matheus Torquato and Marcelo Fernandes. A Parallel Implementation of Sequential Minimal Optimization on FPGA. Microprocessors and Microsystems. 69. 10.1016/j.micpro.2019.06.007, 2019.

[5] Xilinx® Inc., Vivado Design Suite User Guide: High-Level Synthesis (UG902 v2019.2), January 13, 2020

[6] Xilinx® Inc., Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393 v2019.2), February 28, 2020