Dataflow Design for Optimal Incremental SVM Training

Shengjia Shao*, Oskar Mencer*,† and Wayne Luk*
*Department of Computing, Imperial College London, †Maxeler Technologies
United Kingdom
E-mail: {shengjia.shao12, o.mencer, w.luk}@imperial.ac.uk

***Abstract*—This paper proposes a new parallel architecture for incremental training of a Support Vector Machine (SVM), which produces an optimal solution based on manipulating the Karush-Kuhn-Tucker (KKT) conditions. Compared to batch training methods, our approach avoids re-training from scratch when training dataset changes. The proposed architecture is the first to adopt an efficient dataflow organisation. The main novelty is a parametric description of the parallel dataflow architecture, which deploys customisable arithmetic units for dense linear algebraic operations involved in updating the KKT conditions. The proposed architecture targets on-line SVM training applications. Experimental evaluation with real world financial data shows that our architecture implemented on Stratix-V FPGA achieved significant speedup against LIBSVM on Core i7-4770 CPU.**

I. Introduction

Support Vector Machine (SVM) is one of the most widely-used supervised machine learning techniques with successful application to various classification and regression tasks [1]. To train an SVM, a Quadratic Programming (QP) problem constructed from the training dataset needs to be solved. The QP problem has a unique global optimal solution, determined by the Karush-Kuhn-Tucker (KKT) conditions [2].

Traditionally, SVM is trained using *batch training* methods, in which the training dataset is gathered, then the corresponding QP problem is solved [2]. Its drawback is whenever the training dataset changes, the QP problem needs to be solved from scratch again. For on-line tasks like financial trading in which the training dataset itself is evolving, *incremental training* methods are more suitable than batch training methods. Incremental training updates the SVM from existing KKT conditions in obtaining the solution for the new training dataset, without solving QP from scratch again. Incremental SVM training algorithms can be *optimal* or *approximate*, depending on whether the new SVM obtained from incremental updating corresponds to the *exact* new optimal QP solution or an *approximate* one. In this paper, we develop an optimal incremental SVM training algorithm [3] on FPGA.

Much existing work accelerating SVM training on FPGA focuses on batch training [4]. This paper presents the first hardware accelerated SVM for incremental training. In particular, we focus on the ϵ-insensitive SVM for Regression (ϵ-SVR). The main novelty is a parametric parallel dataflow architecture with customisable arithmetic units for the dense linear algebraic operations involved in updating the KKT conditions. Specifically, we make the following contributions:

- A novel dataflow architecture addressing the challenges of incremental SVM training on FPGA: random memory access, numerical accuracy, and list manipulation.
- A parallel data path for updating KKT conditions. Parallelism is adjustable to make the design scalable, and to trade-off between parallelism and resource usage.
- Implementation on Maxeler MPC-X2000 with an Altera Stratix-V FPGA and evaluation using real financial data. The proposed system is significantly faster than software.

The rest of this paper is organised as follows. Section II reviews related background. Section III details our dataflow architecture. Section IV presents experimental evaluation. Finally, Section V provides conclusion and suggests future work.

II. Background

We focus on the ϵ-insensitive SVM for Regression (ϵ-SVR). With training dataset $D = \{(\mathbf{x}_i, y_i) \mid i = 1, \cdots, N\}$, where $\mathbf{x}_i \in \mathbb{R}^d$ the input feature and $y_i \in \mathbb{R}$ the regression target, ϵ-SVR gives the following regression function:

$$f(\mathbf{x}) = \sum_{i=1}^{N} \theta_i K(\mathbf{x}, \mathbf{x}_i) + b \tag{1}$$

Here, θ_i and b are obtained from the KKT conditions. $K(\mathbf{x}, \mathbf{x}') = \langle \Phi(\mathbf{x}), \Phi(\mathbf{x}') \rangle$ is the kernel function [2].

We define training error as $h(\mathbf{x}_i) = f(\mathbf{x}_i) - y_i$. For the SVM training problem, the KKT conditions can be expressed as a relation between training error and coefficients [3]. These relations divide the training dataset into three subsets:

$$\begin{cases} \text{Set S} = \{i \mid |h(\mathbf{x}_i)| = \epsilon, 0 < |\theta_i| < C\} \\ \text{Set E} = \{i \mid |h(\mathbf{x}_i)| \geq \epsilon, |\theta_i| = C\} \\ \text{Set R} = \{i \mid |h(\mathbf{x}_i)| \leq \epsilon, \theta_i = 0\} \end{cases} \tag{2}$$

When a sample $(\mathbf{x}_c, y_c)$ joins or leaves the training dataset, a new optimal solution could be obtained incrementally by adjusting θ_i and b such that the KKT conditions are still satisfied. This is the idea of incremental SVM training algorithms [3], [5]. The major computation involved is calculating the sensitivities of θ_i and $h(\mathbf{x}_i)$ with respect to θ_c. These sensitivities tell us how existing training samples are affected if we change the weight θ_c of the new sample. These computations involve two dense matrices $\mathbf{Q}$ and $\mathbf{R}$. $\mathbf{Q}$ is the kernel matrix $Q_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$. $\mathbf{R}$ is a matrix that depends on $\mathbf{Q}$ and set S. Using $\{s_1, s_2, \cdots, s_{ls}\}$ to denote all samples in S, matrix $\mathbf{R}$ can be expressed as follows:

978-1-5090-5602-6/16/$31.00 ©2016 IEEE

$$\mathbf{R} = \begin{bmatrix} 0 & 1 & \cdots & 1 \\ 1 & Q_{s_1,s_1} & \cdots & Q_{s_1,s_{ls}} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & Q_{s_{ls},s_1} & \cdots & Q_{s_{ls},s_{ls}} \end{bmatrix}^{-1} \tag{3}$$

With matrices $\mathbf{Q}$ and $\mathbf{R}$, sensitivity of b, θ_i and $h(\mathbf{x}_i)$ of all training samples with respect to θ_c and b can be calculated via matrix-vector multiplications [5] [3]. The first one is vector $\vec{\beta}$, the sensitivity of b and $\theta_i (i \in S)$ with respect to θ_c. $\vec{\beta}$ is a matrix-vector product of $\mathbf{R}$ and some $\mathbf{Q}$ elements:

$$\begin{bmatrix} \Delta b \\ \Delta\theta_{s_1} \\ \vdots \\ \Delta\theta_{s_{ls}} \end{bmatrix} = \vec{\beta}\Delta\theta_c = \left(-\mathbf{R} \begin{bmatrix} 1 \\ Q_{s_1,c} \\ \vdots \\ Q_{s_{ls},c} \end{bmatrix} \right) \Delta\theta_c \tag{4}$$

For the elements in set E, R, consider the sensitivity between training error $h(\mathbf{x}_i)$ $(i \in E \cup R)$ and θ_c. Denote the samples in $E \cup R$ as $\{n_1, n_2, \cdots, n_{ln}\}$ and let $\Delta h(\mathbf{x}_i) = \gamma_i \Delta\theta_c$. The coefficient vector $\vec{\gamma}$ is calculated as follows:

$$\vec{\gamma} = \begin{bmatrix} Q_{n_1,c} \\ Q_{n_2,c} \\ \vdots \\ Q_{n_{ln},c} \end{bmatrix} + \begin{bmatrix} 1 & Q_{n_1,s_1} & \cdots & Q_{n_1,s_{ls}} \\ 1 & Q_{n_2,s_1} & \cdots & Q_{n_2,s_{ls}} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & Q_{n_{ln},s_1} & \cdots & Q_{n_{ln},s_{ls}} \end{bmatrix} \vec{\beta} \tag{5}$$

Using the sensitivity vectors $\vec{\beta}$ and $\vec{\gamma}$, the change of θ_c or b in each iteration can be calculated.

The two matrices need to be updated when necessary. $\mathbf{Q}$ is updated when a new sample joins the training data set. As $\mathbf{Q}$ is symmetrical, only one row needs to be updated. $\mathbf{R}$ will be enlarged or shrunken when a sample joins or leaves set S [5].

The incremental training algorithm needs a starting point. It can be initialised using either an existing trained SVM (warm start) or two data samples (cold start) [3]. The incremental training procedure can also be 'reversed' when an existing training sample is removed from the training dataset, which is called *decremental training*. In decremental training, the coefficient θ of the leaving sample is gradually reduced to 0 while preserving KKT conditions [3], [5].

III. Hardware Design

The general system architecture is shown in Figure 1. Our system supports both incremental and decremental SVM training as they share the same data path. As most of the computations involve dense linear algebra, we introduce parallelism by using blocked matrix-vector arithmetic. The idea is to divide an n-by-n matrix into K×K blocks sized at (n/K)-by-(n/K) each and process them in parallel. K is a compile time parameter. We will first discuss the design challenges.

A. Design Challenges

1) Random Memory Access: While most of the dense linear algebra operations involved can be easily parallelised using blocked matrix-vector arithmetic, the computation of vector $\vec{\gamma}$, shown in eq. (5), brings challenges due to its random memory access pattern. Denote the number of elements of set S by N_S, and that of set $E \cup R$ by N_{ER}. $\vec{\gamma}$ is a vector of N_{ER} elements. Note that eq. (5) implies random access of $\mathbf{Q}$, as both row and column addresses depend on the set membership of S, E, R at run-time. As matrix $\mathbf{Q}$ is divided into K×K blocks and each BRAM block only has two ports in hardware, we are unable to truly parallelise the random memory accesses, i.e. what if the K elements needed in a certain cycle happened to be in the same $\mathbf{Q}$ block?

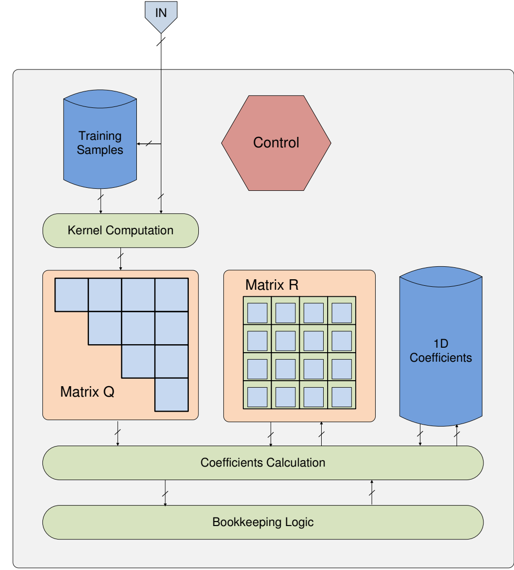

Fig. 1. The overall system block diagram. The system is composed of a central control block and parallelised computational and storage blocks. The proposed architecture supports both incremental and decremental SVM training.

We address this challenge by exploiting problem specific properties to re-arrange the loop. We notice that in many cases $N_{ER} \gg N_S$, i.e. the majority of the training samples belong to $E \cup R$. Thus, we extend the outer loop (the loop over set $E \cup R$ with random access $\{n_1, n_2, \cdots, n_{ln}\}$) to loop over the entire dataset (the sequential loop over sample $\{1, 2, 3, \cdots, n\}$), and parallelise it with factor K. The inner loop over set S is still computed sequentially due to random memory access. The time complexity of computing $\vec{\gamma}$ without parallelisation is $O(N_S \times N_{ER})$. With the proposed scheme, the time complexity is $O(N_S \times n/K)$. In a typical scenario that $N_{ER} \gg N_S$, the scheme is highly effective.

2) Numerical Accuracy: The incremental SVM iteratively updates itself, and such procedure is sensitive to numerical errors. In our FPGA design we use fixed-point numbers to reduce resource usage, and it can be challenging to maintain good numerical accuracy. We handle this issue in two ways:

- Using more fractional bits. Theoretically, `double` has 15-17 decimal digits precision. Thus we use 50 fractional bits in our fixed-point data type (˜15 decimal digits).
- Exploiting problem specific features. As shown in eq. (2), a sample in set S has $h(\mathbf{x}_i) = \pm\epsilon$; a sample in set E has $\theta_i = \pm C$; a sample in set R has $\theta_i = 0$. These values are fixed. Although the incremental algorithm will compute

them when updating, there may be small deviations due to limited precision. To correct such deviations we always write these fixed values explicitly ($\pm\epsilon, \pm C, 0$) when we know them.

By combining the two methods above, our FPGA design achieves the same level of accuracy as the double precision LIBSVM software [6] in our experiment. For better accuracy the number of fractional bits can be further increased.

3) List Manipulation: The training samples are divided into S, E, R sets. Each set is a list. When a sample moves from one set to another we need to update the lists. Set membership update involves inserting/removing an element in a random position in a list. A straightforward implementation on FPGA is storing each list in an array and put it in a BRAM block. In this way the set membership update will have $O(n)$ time complexity. For better performance we implement the list using shift registers. In each clock cycle, the new value for each register can be either: 1) its current value; 2) the value from the register on its left; 3) the value from the register on its right; 4) the external input. As all these registers are synchronous and they operate in parallel, the insertion/removal of an element in a random position in the list can finish within one clock cycle. In this way we reduce the time complexity of list manipulation from $O(n)$ to $O(1)$.

B. Training with Limited Resources - The Sliding Window

In the proposed architecture for incremental and decremental SVM training, all related coefficients are stored in FPGA's BRAM for efficient access. Among these coefficients, matrix $\mathbf{Q}$ and $\mathbf{R}$ are the most memory consuming.

Consequently, the number of training samples that the system can store is determined by the BRAM space available in the FPGA chip. To train SVM with limited resources, we adopt a sliding window approach: when the window is full and a new sample arrives, an existing sample needs to be removed before the new sample can be added to the training dataset. Similar approach of learning with limited resources has been reported [7]. In this paper, as we are evaluating the system with high-frequency financial data, we choose to remove the oldest sample from the training dataset, as it is considered to be out-of-date. For other applications, the control logic can be modified to deploy a different strategy of selecting the item to be removed.

IV. Experimental Evaluation

In this section we evaluate the proposed hardware design running on FPGA against LIBSVM, one of the most widely-used SVM software libraries [6]. LIBSVM uses the Sequential Minimal Optimisation (SMO) algorithm to train SVM. SMO is an efficient batch training algorithm. LIBSVM is single-threaded. We use high-frequency financial order book data in our experiments.

A. Hardware Platform

The proposed system is implemented using the MaxJ dataflow computing language by Maxeler Technologies and built on Maxeler's MAX4 platform with an Altera Stratix-V 5SGSD8 FPGA (28nm technology). The FPGA runs at 150MHz. The LIBSVM software runs on a computer with Intel Core-i7 4770 CPU at 3.4GHz (22nm technology) and with 16GB DDR3-1600 memory.

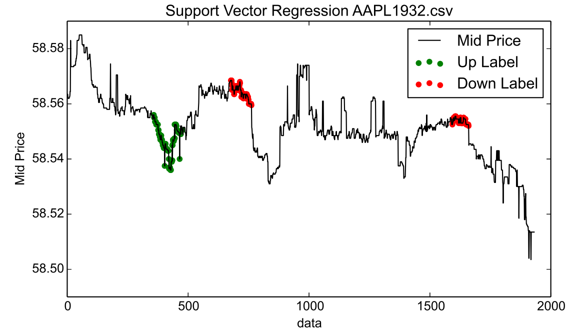

Fig. 2. Mid-Price Prediction using Incremental SVM with WinSize 420

B. Benchmark Problem

We use 5-level financial order book data in our experiments. An order book is a list of buy and sell orders for a certain financial instrument. Level 1 correspond to the best bid and ask, level 2 the second best, and so on. The order book keeps evolving as market participants buy and sell. Our application predicts future mid-price at level 1 (average of best bid and best ask). We construct 16 features from the order book data as follows (EMA is Exponential Moving Average):

- 1-2: Ask(Bid) Size at Level 1
- 3: Bid Size at Level 1 - Ask Size at Level 1
- 4-5: Total Ask(Bid) Size in Level 1-5
- 6-10: Mid-Price at Level 1-5
- 11: Ask Price at Level 1 - Bid Price at Level 1
- 12-13: Weighted Average Ask(Bid) Price of Level 1-5
- 14: 10-period EMA of Mid Price at Level 1
- 15: 20-period EMA of Mid Price at Level 1
- 16: (Bid Price at Level 1 * Ask Size at Level 1 + Ask Price at Level 1 * Bid Size at Level 1)/(Ask Size at Level 1 + Bid Size at Level 1)

This SVM model has the potential to predict stock price movements. Figure 2 shows the mid-price prediction using our window-based incremental SVM with window size 420. A green dot is plotted when SVM mid-price prediction is continuously higher than the recent mid-price for 150 events; and a red dot is plotted when it is continuously lower for 150 events. They can be used to make trading decisions.

C. Resource Usage

Our incremental SVM system has a window size n $=$ 420, and RSize $=$ 120. RSize controls the allocated memory space (RSize $\times$ RSize elements) for matrix $\mathbf{R}$, which can be smaller than window size. This is because $\mathbf{R}$ is determined by set S and set S can be small for many real world problems. With $(420, 120)$ fixed, we parallelise our system with K=3,4,5,6 and report resource usage in Table I. FPGA clock frequency is set

TABLE I
RESOURCE USAGE OF STRATIX-V 5SGSD8 FPGA

K	QBDim	RBDim	LUT	FF	BRAM	DSP
3	140	40	185942	270687	1446	770
4	105	30	201825	313540	1479	1143
5	84	24	224748	365129	1657	1486
6	70	20	253331	423728	1892	1916
Available			524800	1049600	2567	1963

TABLE II
PERFORMANCE COMPARISON BETWEEN LIBSVM AND FPGA (K=6)

Samples	T_{LIB}(s)	Cycles	$T_{Exp.}$(s)	$T_{Act.}$(s)	$Acc_{Exp.}$	$Acc_{Act.}$
402	0.1826	836953	0.0056	0.0308	32.61x	5.93x
802	0.5991	1712549	0.0114	0.0406	52.55x	14.76x
1202	1.9214	2620524	0.0175	0.0469	109.79x	40.97x
1602	2.1240	3785907	0.0252	0.0564	84.29x	37.66x
1902	2.3672	4873275	0.0325	0.0672	72.84x	35.22x

to 150MHz. In the table, QBDim $= n/K$ is the dimension of each matrix $\mathbf{Q}$ block, RBDim $=$ RSize$/K$ is the dimension of each matrix $\mathbf{R}$ block. Larger K will lead to fewer kernel cycles, thus better performance. DSP is the critical resource in our system. Most of the DSPs are used by: a) Gaussian RBF kernel $K(\mathbf{x}, \mathbf{x}') = exp(-\frac{\|\mathbf{x}-\mathbf{x}'\|^2}{2\sigma^2})$, because of its complexity; and b) matrix $\mathbf{R}$ enlarging and shrinking, because $\mathbf{R}$ is divided into $K \times K$ blocks.

We notice that although the total size of matrix $\mathbf{Q}$, $\mathbf{R}$ and other coefficients stay the same for different configurations as $n = 420$ and RSize $= 120$ are fixed, there is an increase in BRAM usage with parallelism K. This is because blocked storage is used for parallel access and higher parallelism means more blocks, even though the overall size is unchanged.

D. Performance Evaluation and Discussion

We compare the elapsed time for LIBSVM and FPGA to perform the training task. Our dataset contains 1902 items, corresponding to 88 seconds of trading. We use the sliding window approach with window size $n = 420$ and RSize $= 120$. This means as the stock trading goes on, we always use the latest 420 prices to train the SVM. In the beginning when there are fewer than 420 items, all data are used. Table II shows the performance results. Below are the descriptions of table items:

- T_{LIB}: the run time of LIBSVM to perform the task
- Cycles: the number of FPGA cycles needed for the task
- $T_{Exp.}$: Expected FPGA run time $T_{Exp.} =$ Cycles/Freq.
- $T_{Act.}$: Actual FPGA run time
- $Acc_{Exp.}$: Expected speed-up $Acc_{Exp.} = T_{LIB}/T_{Exp.}$
- $Acc_{Act.}$: Actual speed-up $Acc_{Act.} = T_{LIB}/T_{Act.}$

As we see from the table, up to 40.97 times speed-up has been achieved. However, the expected speed-up calculated using the number of cycles to run and the FPGA clock frequency (150MHz) is much greater than the actual speed-up. This means our system is bounded by CPU-FPGA communication.

When constructing the SVM we use 16 features, so together with the prediction target (future mid-price), there are 17 items. As they are double-precision numbers, the size of each training sample is therefore $17 * 64 = 1088$ bits. The total size of 1902 samples is 252.61KB. In the Maxeler MAX4 system we used, CPU and FPGA communicate via an Infiniband connection with 2GB/s bandwidth. If we divide 252.61KB by 2GB/s, then data transfer only needs 0.00012s; but this is certainly not the case as the difference between $T_{Exp.}$ and $T_{Act.}$ is much larger than that. A reasonable explanation is that bandwidth is a function of transfer data size and pattern. In our experiment, the data are transferred in small pieces (1088 bits) at random times (when the training of previous sample finishes). It is possible that for this kind of data transfer the actual Infiniband bandwidth available may be much smaller than expected.

V. CONCLUSION AND FUTURE WORK

In this paper, we have introduced a novel dataflow design for incremental SVM training. The proposed design addresses three challenges of implementing incremental SVM efficiently on FPGA: random memory access, numerical accuracy, and list manipulation. Experimental evaluation using high frequency financial data shows the proposed design running on Stratix-V FPGA achieves up to 40.97 times speed up against LIBSVM software on Core-i7 4770 CPU. The proposed design is suitable for scenarios in which on-line SVM training is needed, such as financial time series prediction.

Possible future work includes using data compression to reduce communication overhead. Assuming the I/O overhead is removed with such compression techniques, we should be able to reach the expected speed-up (up to 109.79 times).

ACKNOWLEDGEMENTS

This work is supported in part by the Lee Family Scholarship, European Union Horizon 2020 research and innovation programme under Grant Number 671653, by the UK EPSRC (EP/N031768/1, EP/I012036/1, EP/L00058X/1 and EP/K503733/1), the Maxeler University Programme, the HiPEAC NoE, and Altera.

REFERENCES

[1] H. Byun and S.-W. Lee, "Applications of support vector machines for pattern recognition: A survey," in *Pattern Recognition with Support Vector Machines*. Springer, 2002, pp. 213–236.

[2] A. J. Smola and B. Schölkopf, "A tutorial on support vector regression," *Statistics and Computing*, vol. 14, no. 3, pp. 199–222, 2004.

[3] J. Ma, J. Theiler, and S. Perkins, "Accurate on-line support vector regression," *Neural Computation*, vol. 15, no. 11, pp. 2683–2703, 2003.

[4] S. M. Afifi, H. GholamHosseini, and R. Sinha, "Hardware implementations of SVM on FPGA: A state-of-the-art review of current practice," *International Journal of Innovative Science, Engineering & Technology*, vol. 2, 2015.

[5] T. Poggio and G. Cauwenberghs, "Incremental and decremental support vector machine learning," *NIPS*, vol. 13, p. 409, 2001.

[6] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[7] P. Laskov, C. Gehl, S. Krüger, and K.-R. Müller, "Incremental support vector learning: Analysis, implementation and applications," *Journal of Machine Learning Research*, vol. 7, no. Sep, pp. 1909–1936, 2006.