On the Design of a Hardware-Software Architecture for acceleration of SVMs training phase

Lázaro Bustio-Martínez[1,2], René Cumplido[2], José Hernández-Palancar[1], and Claudia Feregrino-Uribe[2]

[1] Advanced Technologies Application Center.
7^a ♯ 21812 e/ 218 y 222, Rpto. Siboney, Playa, C.P. 12200, Havana, Cuba.
{lbustio,jpalancar}@cenatav.co.cu
[2] National Institute for Astrophysics, Optics and Electronic.
Luis Enrique Erro No 1, Sta. Ma. Tonantzintla, 72840, Puebla, México.
{rcumplido,cferegrino}@inaoep.mx

Abstract. Support Vector Machines (SVM) is a new family of Machine Learning techniques that have been used in many areas showing remarkable results. Since training SVM scales quadratically (or worse) according of data size, it is worth to explore novel implementation approaches to speed up the execution of this type of algorithms. In this paper, a hardware-software architecture to accelerate the SVM training phase is proposed. The algorithm selected to implement the architecture is the Sequential Minimal Optimization (SMO) algorithm, which was partitioned so a General Purpose Processor (GPP) executes operations and control flow while the coprocessor executes tasks than can be performed in parallel. Experiments demonstrate that the proposed architecture can speed up SVM training phase 178.7 times compared against a software-only implementation of this algorithm.

Key words: SVM, SMO, FPGA, Parallel, hardware-software architectures

1 Introduction

Support Vector Machines (SVM) is a recent technique that has been widely used in many areas showing remarkable results, specially in data classification [5]. It was developed by Vladimir Vapnik in the early 90's and created an explosion of applications and theoretical analysis that has established SVM as a powerful tool in Automatic Machine Learning and Pattern Recognition [10].

Due to SVM's training time scales quadratically (or worse) according to training database size [2], the problems that can be solved are limited. Many algorithms have been proposed to avoid this restriction, although at present there are three basic algorithms for training SVM [11]: *Chunking* [9], *Sequential Minimal Optimization* (SMO) [8] and SVM^{Light} [6] (this algorithm is an improvement to [7]). SMO has proved to be the best of them because it reduces the training time, it does not need expensive computational resources as the others, it is easily programmable and it does not require complex math libraries to solve Quadratic Programming (QP) problems that SVM involves.

SVM is inadequate for large scale data classification due to the high training times and computational resources that it requires. Because of this, is very important to explore techniques that can help to improve SVM's performance. This is the case of hardware-software architectures, especially, GPPs that can enhance their instruction set by using an attached coprocessor.

To prove the feasibility using hardware-software architectures to accelerate algorithms, a *Field Programmable Gates Arrays* (FPGA) is used as a prototyping platform. A FPGA is an integrated circuit that can be configured by the user making possible to build circuits. FPGAs are formed by logic blocks wired by reprogrammable connections, who can be configured to perform complex combinational functions (even to implement a GPP). FPGAs are used in many areas obtaining significant speed ups, such as automatic target recognition, string pattern matching, transitive closure of dynamic graphs, Boolean satisfiability, data compression and genetic algorithms [3], among others.

In this paper, SMO's performance was analyzed to identify those sections that are responsible of the processing bottleneck during its execution. To accelerate SMO, a hardware-software architecture was designed and implemented. In this architecture, hardware executes the most time-consuming functions while the software executes control flow and iterative operations.

This paper is organized as follows: in Section II describes the different approaches to implement processing algorithms, including a brief description of the FPGAs. In Section III, the SVM and their theoretical foundation are revised as well as the most cited algorithms that train SVM are described, explaining their characteristics and particularities, specially for the SMO algorithm. In Section IV the architecture proposed is described, detailing software and hardware implementations while in Section V the results are shown. The work is concluded in Section VI.

2 Platforms for algorithms implementation

There are two main approaches to implement algorithms. The first one consists in building *Application Specific Integrated Circuits* (ASICs)[3]. They are designed and built specifically to perform a given task, and thus they are very fast and efficient. ASICs can not been modified after fabrication process and this is their main disadvantage. If an improvement is needed, the circuit must be re-designed and re-builded, incurring in the costs that this entails.

The second one consists in using a GPP which is programmed by software; it executes the set of instructions that are needed by an algorithm. Changing the software instructions implies a change in the application's behavior. This results in a high flexibility but the performance will be degraded. To accomplish certain function, the GPP, first must read from memory the instructions to be executed and then decode their meaning into native GPP instructions to determine which actions must be done. Translating the original instructions of an algorithm introduces a certain delay.

The hardware-software architectures combines the advantages of those two approaches. It aims to fills the gap between hardware and software, achieving potentially much higher performance than software, while maintaining a higher level of flexibility than hardware.

In classification tasks, many algorithms are expensive in terms of processing time when they are implemented in GPP and they classifies large scale data. When a classification algorithm is implemented, it is necessary to perform a high amount of mathematical operations that can not be done without the flexibility that software provides. So, a hardware-software architectures offer an appropriate alternative to implement this type of algorithms.

FPGAs appeared in 1984 as successors of the *Complex Programmable Logic Devices* (CPLDs). The architecture of a FPGAs is based on a large number of logic blocks which performs basic logic functions. Because of this, an FPGA can implement from a simple logical gate, to a complex mathematical function. FPGAs can be reprogrammed, that is, the circuits can be "erased" and then, a new algorithm can be implemented. This capability of the FPGAs allow us to create fully customized architectures, reducing cost and technological risks that are present in traditional circuits design.

3 SVM for data classification

SVM is a set of techniques based on convex quadratic programming for data classification and regression. The main goal of SVM is to separate training data into two different groups using a decision function (separating hyperplane) which is obtained from training data. The separating hiperplane can be seen, in its simplest way, as a line in the plane whose form is $y = \boldsymbol{w} \cdot \boldsymbol{x} + b$ or $\boldsymbol{w} \cdot \boldsymbol{x} - b = 0$ for the canonical hyperplane. SVM classification (in a simple two-class problem) simply looks at the sign of a decision function for an unknown data sample.

Training a SVM, in the most general case, is about to find those λ's that maximizes the Lagrangian formulation for the dual problem L_D according to the following equation:

$$L_D = \sum_{i=1}^{l} \lambda_i - \frac{1}{2} \sum_{i,j=1}^{l} y_i y_j K\left(\mathbf{x_i} \cdot \mathbf{x_j}\right) \lambda_i \lambda_j \tag{1}$$

subject to:

$$\sum_{i=1}^{l} y_i \lambda_i = 0; 0 \leqslant \lambda_i \leqslant C, i = 1, 2, ..., l \tag{2}$$

where $K(\mathbf{x_i} \cdot \mathbf{x})$ is a positive definite kernel that maps input data into a high dimension feature space where linear separation becomes more feasible [12]. $\mathbf{x_i}, \mathbf{x_j} \in R^d$ are the input vectors of the i^{th} and j^{th} training data respectively, l is the number of training samples; $y \in \{-1; 1\}$ is the class label; $\lambda = \lambda_1, \lambda_2 ... \lambda_n$ are the Lagrange multipliers for the training dataset in the Lagrangian formulation. So,the unknown data can be classified using $y = sign\left(\sum_{i=1}^{l} y_i \lambda_i K\left(\mathbf{x_i} \cdot \mathbf{x}\right) - b\right)$ where b is the SVM's threshold and is obtained using $\lambda_i\left(y_i\left(\mathbf{w} \cdot \mathbf{x_i} - b\right) - 1\right) = 0, i = 1, 2, ..., l$ for those data samples with $\lambda_i > 0$ (those data samples are called *Support Vectors*).

The kernel function depends on the user's choice, and the resultant feature space determines the functional form of the support vectors; thus, different kernels behave differently. Some common kernels can be found on [7]. Many of the kernel functions are formed by Linear Kernel, except RBF one. Mathematically, to accelerate the Linear Kernel implies to accelerate the others. Because of this, the Linear Kernel is focused in this paper.

4 Architectural design

SMO is basically a sequential algorithm: heuristic hierarchy is formed by a set of conditional evaluations which decides the algorithm behavior, with every evaluation depending on the result of the previous evaluation. Because of this sequentiality, SMO can not be implemented as it is in hardware. In addition, the highly time-consuming functions are fully parallelizable, as it is the case of kernel function computation. Thus, a hardware-software architecture that implements in hardware the most time-consuming functions and heuristic hierarchy in software could be the right approach for reducing execution time in SVM training.

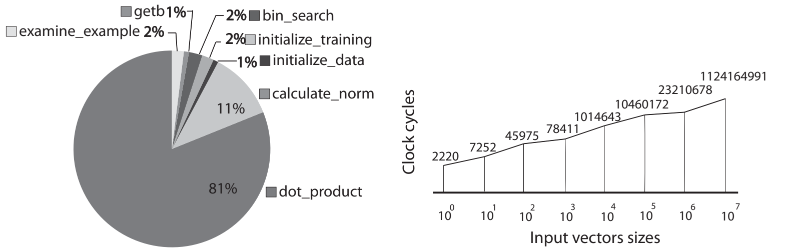
4.1 SMO's performance profiling

There are a few SMO's performance analyses in the literature. Only Dey et al. in [4] analyze SMO's performance and identify the most time-consuming functions. In their paper, Dey et al. demonstrate the convenience of using hardware-software architectures to speed up algorithms and use SVM as an example to prove this approach. In order to identify task level hot spots in SMO's execution and to validate Dey's results, a performance profiling was made. The results are shown in Fig. 1(a).

It was observed that 77% of the total calls in SMO corresponds to the *dot_product* function. The time profile analysis shows that 81% of the total execution time was spent by the *dot_product* function. As a result of performance analysis it is evident that the *dot_product* function is responsible of bottleneck in SMO's execution. Fig. 1(b) supports this conclusion. From the performance analysis we concluded that using a hardware-software architecture to implement SMO algorithm, where software implements heuristic hierarchy, and hardware implements the *dot_product* function could obtain an speed up of at least one order of magnitude when is compared to software implementations.

4.2 Architecture description

Fig. 2 shows a diagram of the proposed architecture. The architecture is formed



(a) Performance profile for SMO.

(b) Performance profile for dot product calculation in software.

Fig. 1. Performance profile analysis.

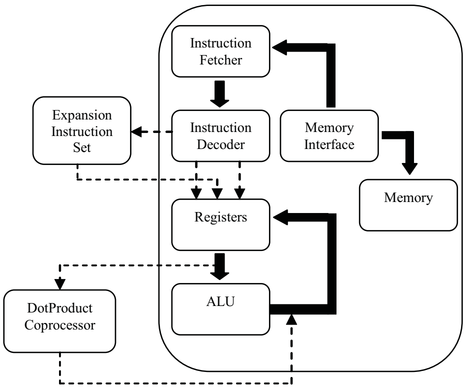
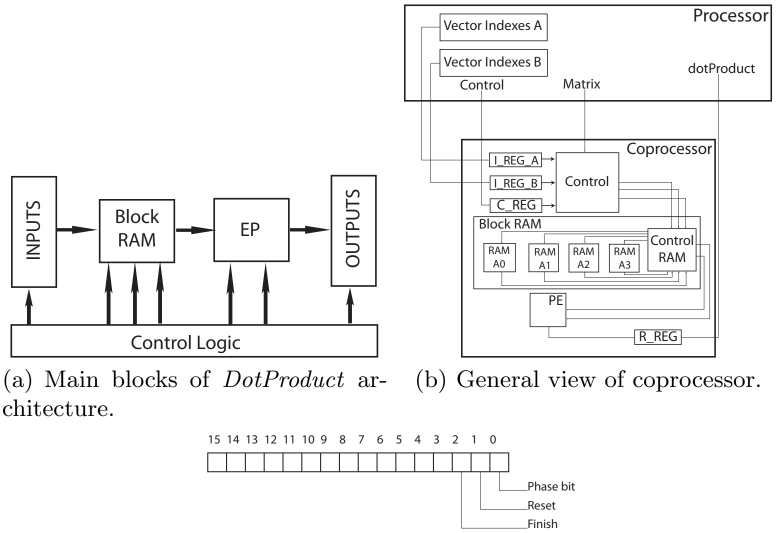


Fig. 2. Diagram of the proposed architecture.

by a GPP that can enhance its performance by using a coprocessor, where the control structures are executed in software on the GPP and the dot product computations are executed on the coprocessor. The software reads the training file, initializes the data structures and receives the parameters for the SMO. Thus, when training starts, the software executes control mechanisms and the coprocessor executes high time-consuming functions.

4.3 Software Implementation

To accomplish the proposed architecture, the software implementation must first load a training file and algorithm parameters. After that, the application executes the SMO algorithm and selects the correct branch from the heuristic hierarchy that SMO implements. When a dot product is needed, the application indicates the vectors that will be sent to the coprocessor. When the computation is finished, the application obtains the resulting dot product from the coprocessor, generates the output file with the training results and finishes the training process.



(a) Main blocks of *DotProduct* architecture.

(b) General view of coprocessor.

(c) Structure of C-REG.

Fig. 3. Description of the proposed architecture.

4.4 Hardware Implementation

The hardware architecture for the *dot_product* calculation will be named *DotProduct*, while SMO with the *dot_product* function implemented in hardware will be named *FSMO*. For this architecture, the training dataset will be represented as a matrix without using any compression method and requires that values of the matrix to be 1 or 0. Since the dot product is calculated many times and the values for this calculation remains constant, the right strategy to avoid unwanted delays is to map the training dataset inside the coprocessor. The dot product is $dotProduct = \sum_{i,j=1}^{l} \mathbf{x_i} \cdot \mathbf{x_j}$ where $\mathbf{x_i}$ and $\mathbf{x_j}$ are training vectors and l is the number of elements on vectors. The digital architecture that implements this mathematical expression consists of 5 main blocks as shown in Fig. 3(a).

INPUTS represents control signals, registers and data necessary for the architecture to work. BLOCK RAM is a memory block that contains the training dataset. Each row corresponds to one training data sample. The Processor Element (PE) is the basic computation unit which calculates the dot products of two input vectors. OUTPUT is the element that addresses the dot product computation results, and CONTROL LOGIC are those elements that permit to control and data flow inside the architecture.

Through INPUTS, the *DotProduct* architecture obtains the indexes that will be used on the dot product calculation. INPUTS is used for mapping training data into BLOCK RAM. At this point, all data necessary to calculate a dot product of input vectors are inside the *DotProduct* architecture. Those two vectors whose indexes were given through INPUTS are delivered to the PE where

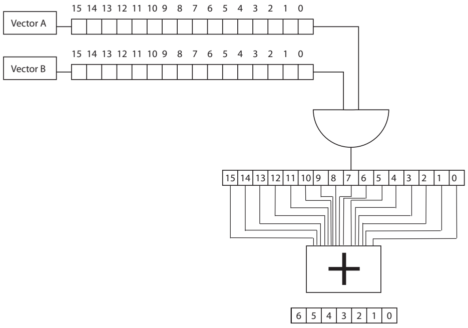


Fig. 4. Hardware implementation of *DotProduct* architecture.

the dot product is calculated and then, the result is stored in OUTPUTS. A general view of architecture is shown in Fig. 3(b).

There are two registers, *I_REG_A* and *I_REG_B*, which hold indexes of training data samples that will calculate the dot product. Register *C_REG* controls when to load data, when to read from BLOCK RAM or when to start a dot product calculation. *C_REG* is shown in Fig. 3(c). The *Phase* bit states whether the architecture is in the Initialization and Load Data Phase (set to 0) or in the Processing Phase (set to 1). When *Reset* is active (set to 1), all registers are initialized to 0, Initialization and Load Data Phase are enabled and the PE is ready to process new data. The *Finish* bit indicates when processing is finished and it is active at 1.

When *FSMO* starts, the Initialization and Load Data Phases are activated (the *Phase* bit of *C_REG* is set to 0). After this, register *I_REG_B* is disabled and the address bus of BLOCK RAM is connected to *I_REG_A* indicating the address where the value stored in matrix will be written (see Fig. 3(b) for more details) ensuring data transfer from training dataset into BLOCK RAM. When BLOCK RAM is filled, the architecture stays at this state while the *Phase* bit of *C_REG* is 0. When the *Phase* bit is changed to 1, matrix input is disabled and *I_REG_B* is enabled and connected to BLOCK RAM. At this moment, the training data samples whose indexes are stored in *I_REG_A* and *I_REG_B* are delivered to the PE where the dot product is calculated. The result of the dot product computation is stored in *R_REG*, *Finish* bit is activated and the architecture is ready to calculate a new dot product.

The PE calculates the dot product of two given training data samples. For this training dataset representation, the dot product computation is reduced to apply a logical AND operation between input vectors and counts the number of 1's in resulting vector. In this way, the architecture that implements the PE is shown in Fig. 4. Notice that the PE can calculate a dot product using three clock cycles; so, the processing time for the dot product calculation is: $t = 3 \cdot v$ where v is the number of dot products. To prove the validity of the architecture proposed, the *DotProduct* architecture was implemented using VHDL language over ISE 9.2 Xilinx suite, and was simulated using ModelSIM SE 6.5. Hardware

Table 1. Experimental results of training *Adult* with *FSMO*

Corpus	Objs.	Iter.	Training Time		b	Non Bound	Bound
Adult			sec.	C.C.(10^{12})		Support Vectors	Support Vectors
1	1605	3474	364	1.089	0.887	48	631
2	2265	4968	746	2.232	1.129	50	929
3	3185	5850	1218	3.628	1.178	58	1212

architecture was implemented on an XtremeDSP Virtex IV Development Kit card. The software application was written using Visual C++ 6.0 and ANSI C.

Based on the fact that the Linear Kernel are used by many others, the *Dot_Product* architecture is suitable to perform others kernel functions. Using the *Dot_Product* architecture as starting point, any of most used kernel are obtained just adding some blocks that implement the rest of their mathematical formulation.

5 Experiments and Results

Since the dot product is the responsible of the bottleneck in SMO execution, a performance profile for this function was made. Eight experiments were carried out using a Pentium IV processor running at 3GHz and the results are shown in Fig. 1(b). The number of clock cycles required grows with the size of the input vectors.

In hardware, the dot product calculation is independent of input vector size. The *DotProduct* architecture can handle input vectors of 128-bits wide in 3 clock cycles: 1) receives data samples indexes, 2) fetches data sample vectors and 3) calculates the dot product. If the dot product calculation in software of two input vectors of 128-bits wide is compared with hardware implementation, the second one will be completed at 3 clock cycles while the first one will be completed between 45957 and 78411 clock cycles.

5.1 Experiments on Adult dataset

Adult dataset [1] was used by Platt in [8] to prove the feasibility of SMO, and the same dataset was used here to prove the feasibility of proposed architecture. *Adult* dataset consists of 9 corpuses which contain between 1605 and 32562 data samples of 123 characteristics each one. *DotProduct* can manage training datasets of 4096 training data samples of 128 characteristics because of area limitations of the chosen FPGA. Only *Adult-1*, *Adult-2* and *Adult-3* have sizes that can be handled by the *DotProduct* architecture and the results of training those datasets are shown in table 1. In those tables, *C.C.* means *Clock Cycles.*

Table 2. Experimental results of *Adult's* training with Platt's SMO

Corpus	Objs.	Iter.	Time	b	Non Bound	Bound
Adult			sec		Support Vectors	Support Vectors
1	1605	3474	0.4	0.884	42	633
2	2265	4968	0.9	1.127	47	930
3	3185	5850	1.8	1.173	57	1210

Table 3. Deviation in *Adult* training for *FSMO* and Platt's SMO.

Corpus	Threshold b		Dif.	%
Adult	*FSMO*	SMO(Platt)		
1	0.887279	0.88449	0.0027	0.257
2	1.129381	1.12781	0.0015	0.139
3	1.178716	1.17302	0.0056	0.483

Table 2 shows the results for Platt's SMO. There is a deviation in threshold b for this implementations when is compared to *FSMO.* Platt in [8] does not present any implementation detail so it is not possible explain exactly the reason of this deviation: the *epsilon* value of the PC could be responsible for that behavior. Table 3 shows that in the worst case, the deviation incurred is less than 0.5% when is compared to Platt's SMO. So, the proposed architecture trains correctly the SVM.

5.2 Analysis of results

In this paper the hardware architecture to speed up the dot product computation was implemented taking advantage of parallel capabilities of hardware. Also, the heuristic hierarchy of SMO was implemented in software and it uses the hardware architecture for the dot product calculations. *FSMO* trains correctly a SVM, and it accuracy is over 99% compared to Platt's implementation [8].

After the synthesis of the *DotProduct* architecture, it was determined that this architecture can run at 35 MHz of maximum frequency. Since the dot product in hardware takes three clock cycles is then the *DotProduct* architecture could calculate 11666666 dot products of 128-bits wide input vectors in a second. Meanwhile, the same operation for input vectors of 128-bits wide using a Pentium IV processor running at 3GHz of frequency requires 45957 clock cycles, so in this processor, we can calculate 65278 dot products in a second. This demonstrates that the *DotProduct* architecture can run up to 178.7 times faster than its implementation in a modern GPP. The *DotProduct* architecture requires 33% of the available reprogrammable area, thus we can extend it to handle training datasets three times bigger. Larger training datasets can be handled if external memories are used, in this case the architecture can be extended 10 more times.

6 Conclusions

In this paper we proposed a hardware-software architecture to speed up SVM training. SMO algorithm was selected to be implemented in our architecture. SMO uses a heuristic hierarchy to select two candidates to be optimized. The dot product calculation in SMO spent 81% of the total execution time so this function was implemented in hardware while heuristic hierarchy was implemented in software, on the GPP. To validate the proposed architecture we used an XtremeDSP Virtex IV Development Kit card as coprocessor obtaining a speed up of 178.7x for the dot product computations when compared against a software-only implementation running on a GPP.

References

1. D.J. Newman A. Asuncion. UCI machine learning repository, 2007.
2. Christopher J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Min. Knowl. Discov.*, 2(2):121–167, 1998.
3. Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.
4. Soumyajit Dey, Monu Kedia, Niket Agarwal, and Anupam Basu. Embedded support vector machine: Architectural enhancements and evaluation. In *VLSID '07: Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference*, pages 685–690, Washington, DC, USA, 2007. IEEE Computer Society.
5. Isabelle Guyon. Svm application list, 2006.
6. Thorsten Joachims. Making large-scale support vector machine learning practical. pages 169–184, 1999.
7. E. Osuna, R. Freund, and F. Girosi. An improved training algorithm for support vector machines. *Neural Networks for Signal Processing [1997] VII. Proceedings of the 1997 IEEE Workshop*, pages 276–285, 1997.
8. John C. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. Technical report, Microsoft Research, MST-TR-98-14, 1998.
9. Vladimir Vapnik and S. Kotz. *Estimation of Dependences Based on Empirical Data: Empirical Inference Science (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
10. Vladimir N. Vapnik. *The nature of statistical learning theory.* Springer-Verlag New York, Inc., New York, NY, USA, 1995.
11. Guosheng Wang. A survey on training algorithms for support vector machine classifiers. In *NCM '08: Proceedings of the 2008 Fourth International Conference on Networked Computing and Advanced Information Management*, pages 123–128, Washington, DC, USA, 2008. IEEE Computer Society.
12. Eric W. Weisstein. Riemann-lebesgue lemma. online.