Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

A parallel and scalable digital architecture for training support vector machines*

Kui-kang CAO†1, Hai-bin SHEN†‡1, Hua-feng CHEN2
([1]Institute of VLSI Design, Zhejiang University, Hangzhou 310027, China)
([2]Zhejiang University of Media and Communications, Hangzhou 310027, China)
†E-mail: {caokk, shb}@vlsi.zju.edu.cn

Received Aug. 14, 2009; Revision accepted Dec. 4, 2009; Crosschecked May 4, 2010; Published online May 24, 2010

Abstract: To facilitate the application of support vector machines (SVMs) in embedded systems, we propose and test a parallel and scalable digital architecture based on the sequential minimal optimization (SMO) algorithm for training SVMs. By taking advantage of the mature and popular SMO algorithm, the numerical instability issues that may exist in traditional numerical algorithms are avoided. The error cache updating task, which dominates the computation time of the algorithm, is mapped into multiple processing units working in parallel. Experiment results show that using the proposed architecture, SVM training problems can be solved effectively with inexpensive fixed-point arithmetic and good scalability can be achieved. This architecture overcomes the drawbacks of the previously proposed SVM hardware that lacks the necessary flexibility for embedded applications, and thus is more suitable for embedded use, where scalability is an important concern.

Key words: Support vector machine (SVM), Sequential minimal optimization (SMO), Field-programmable gate array (FPGA), Scalable architecture
doi:10.1631/jzus.C0910500 **Document code:** A **CLC number:** TN79

1 Introduction

Support vector machines (SVMs) are one of the most powerful and important supervised machine learning methods (Vapnik, 1998; Schölkopf *et al.*, 1999). They are effective tools for solving problems like face detection, speaker identification, text categorization and so forth (Burges, 1998). Although SVMs are popular on general-purpose computing platforms, where very high performance can be achieved with hardware accelerators such as graphics processing units (GPUs) (Catanzaro *et al.*, 2008) and field-programmable gate arrays (FPGAs) (Graf *et al.*, 2008), their applications in embedded systems are hindered by the complexity of the algorithm, especially when the capability for on-line learning is needed.

On-line learning is indispensable for applications with adaptive capabilities. For these types of applications, very short training time is usually required for fast adaptation. For instance, in the application of SVM for adaptive channel equalization (Sebald and Bucklew, 2000), the SVM classifier needs to be retrained on-line to adapt to a changing environment. To track fast channel variations, very low latency SVM training is required. When SVMs are used to build smart human-computer interfaces such as the sketch recognizer (Sun *et al.*, 2005), fast on-line training is also required to adapt to new users or new user habits swiftly.

To facilitate the application of SVMs in embedded systems where a dedicated and efficient device is preferred, several digital architectures that implement SVMs on hardware have been proposed in recent years. Anguita *et al.* (2003) proposed a VLSI-friendly SVM training algorithm, based on which a digital architecture is presented and implemented on

‡ Corresponding author
* Project (No. 60720106003) supported by the National Natural Science Foundation of China

an FPGA. Wee and Lee (2004) proposed a concurrent architecture for training SVMs based on the kernel Adatron (KA) algorithm (Frieß *et al.*, 1998) and implemented this on an FPGA chip for disease diagnosis. Choi *et al.* (2006) implemented a hardware SVM training unit and incorporated it into a speaker verification system. There are some other hardware implementations of SVMs (Biasi *et al.*, 2005; Anguita *et al.*, 2006; Manikandan *et al.*, 2009), but only the less complicated testing phase was implemented.

Unfortunately, the previously proposed digital architectures for training SVMs lack the necessary flexibility for embedded applications. The SVM hardware described by Choi *et al.* (2006) works only serially. On the other hand, the architectures of both Wee and Lee (2004) and Anguita *et al.* (2003) involve a lot of processing elements (PEs) that process training samples in a fully parallel way, and the number of PEs used in hardware is determined by the size of the training set. Both of the serial and fully parallel architectures are difficult to scale, and thus a trade-off between performance and hardware costs is not possible, making these architectures less suitable to be used in embedded systems, where scalability is an important concern.

To better address this problem, in this paper we propose a parallel and scalable digital architecture based on the widely used sequential minimal optimization (SMO) algorithm for training support vector machines. The error cache updating task in the SMO algorithm is mapped into multiple processing units working in parallel. The number of processing units is adjustable so that scalability can easily be achieved. Unlike previously proposed fully parallel architectures, by adjusting the memory size of the hardware, this architecture can solve SVM training problems of different sizes.

2 Support vector machine classifier and sequential minimal optimization

Given a training set $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{m}$, where $\boldsymbol{x}_i$ is an input pattern and $y_i \in \{\pm 1\}$ is the associated class label, an SVM classifier predicts the class label of a new pattern $\boldsymbol{x}$ with the decision function

$$f(\boldsymbol{x}) = \operatorname{sgn}\left(\sum_{i=1}^{m} a_i y_i k(\boldsymbol{x}, \boldsymbol{x}_i) + b\right), \tag{1}$$

where a_i's are Lagrange multipliers, b is a bias term, and $k(\cdot,\cdot)$ is the kernel function that maps input patterns into the feature space. One of the most widely used kernel functions is the Gaussian radial basis function (RBF):

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(-\left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\|^2 \Big/ (2\sigma^2)\right). \tag{2}$$

The Lagrange multipliers are determined during SVM training, which amounts to solving the following quadratic programming (QP) problem:

$$\begin{aligned} &\max\ L(\boldsymbol{a}) = \sum_{i=1}^{m} a_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} a_i a_j y_i y_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) \\ &\text{s.t.}\ \sum_{i=1}^{m} a_i y_i = 0 \ \text{ and } \ 0 \le a_i \le C,\ i = 1, 2, \ldots, m, \end{aligned} \tag{3}$$

where C is a predefined regularization constant.

Although many algorithms can be applied to solve the QP problem Eq. (3), SMO (Platt, 1999) has proven to be one of the most popular and successful approaches. SMO is an extreme case of the decomposition algorithms aimed at solving large scale problems. By restricting the working set to have only two elements, SMO can solve the sub-QP problems analytically, thus avoiding the use of numerical solvers, which might cause numerical instability issues. To improve the efficiency of the working set selection method of Platt's SMO algorithm, Keerthi *et al.* (2001) suggested two modified versions of SMO. The architecture proposed in this paper is based on the second modification, which is very efficient and is widely used.

To describe Keerthi's SMO algorithm, first divide the training samples into five sets: $I_0=\{i: |y_i|=1, 0<a_i<C\}$, $I_1=\{i: y_i=1, a_i=0\}$, $I_2=\{i: y_i=-1, a_i=C\}$, $I_3=\{i: y_i=1, a_i=C\}$, $I_4=\{i: y_i=-1, a_i=0\}$. Then define the error array $e_i = \sum_{j=1}^{m} a_j y_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) - y_i$, $i=1, 2, \ldots, m$, the upper and lower bounds $b_{\text{up}}=\min\{e_i: i\in I_0\cup I_1\cup I_2\}$, $b_{\text{low}}=\max\{e_i: i\in I_0\cup I_3\cup I_4\}$, and their associated indices $i_{\text{up}} = \arg\min_i e_i,\ i \in I_0 \cup I_1 \cup I_2$ and $i_{\text{low}} = \arg\max_i e_i,\ i\in I_0\cup I_3\cup I_4$.

The SMO algorithm optimizes two Lagrange multipliers at each step according to Eqs. (4) and (5):

$$a_p^{\text{new}} = a_p^{\text{old}} - y_p(e_q^{\text{old}} - e_p^{\text{old}})/\eta, \tag{4}$$

$$a_q^{\text{new}} = a_q^{\text{old}} + s(a_p^{\text{old}} - a_p^{\text{new}}), \tag{5}$$

where $p=i_{\text{low}}$, $q=i_{\text{up}}$, $s=y_p y_q$, $\eta=2k(\boldsymbol{x}_p, \boldsymbol{x}_q)-k(\boldsymbol{x}_p, \boldsymbol{x}_p)-k(\boldsymbol{x}_q, \boldsymbol{x}_q)$. a_p^{new} and a_q^{new} are then clipped to $[0, C]$; i.e., a_p^{new} = min(max(0, a_p^{new}), C), a_q^{new} = min(max(0, a_q^{new}), C).

After optimizing a_p and a_q, the error array e_i is updated according to Eq. (6):

$$\begin{aligned} e_i^{\text{new}} &= e_i^{\text{old}} + (a_p^{\text{new}} - a_p^{\text{old}}) y_p k(\boldsymbol{x}_p, \boldsymbol{x}_i) \\ &\quad + (a_q^{\text{new}} - a_q^{\text{old}}) y_q k(\boldsymbol{x}_q, \boldsymbol{x}_i), \quad i=1,2,\ldots,m. \end{aligned} \tag{6}$$

Then update b_{low}, i_{low}, b_{up}, and i_{up}, based on the new e_i array, and start the next iteration of the procedure.

The training process stops if $i_{\text{low}}=i_{\text{up}}$ or $b_{\text{low}}< b_{\text{up}}+2\tau$ (τ is the tolerance, which is usually set to 10^{-3}), which means all Lagrange multipliers satisfy the Karush-Kuhn-Tucker (KKT) conditions.

The skeleton of the modified SMO algorithm is outlined in Table 1. The computations required for each task are also listed in this table. It is assumed that an inner product between training pattern vectors is required for kernel evaluation, which is the case for the widely used RBF, polynomial and tanh kernels.

Table 1 Skeleton of the modified SMO algorithm and computations required for each task

Task	Description	Computation
1	Initialize: $a_i=0$, $e_i=-y_i$, $i=1,2,\ldots,m$	$2m$
2	Determine: b_{low}, i_{low}, b_{up}, and i_{up}	$4m$
	Repeat	
3	Optimize $a_{i_{\text{low}}}, a_{i_{\text{up}}}$	$(2d+c_1)I$
4	Update e_i, $i=1,2,\ldots,m$	$(4dm+c_2)I$
5	Update b_{low}, i_{low}, b_{up}, and i_{up}	$4mI$
6	Until $i_{\text{low}}=i_{\text{up}}$ or $b_{\text{low}}<b_{\text{up}}+2\tau$	4
7	Calculate $b=(b_{\text{low}}+b_{\text{up}})/2$	2

I: number of iterations; m: number of training samples; d: dimension of the training pattern vectors; c_1 and c_2 are some constants, which depend on the kernel used in the algorithm

3 Architecture design

3.1 Algorithm analysis

The SMO algorithm is a sequential algorithm in its original form. To implement it with parallel hardware, a parallel version of this algorithm should be developed first. By analyzing the characteristics of the SMO algorithm listed in Table 1, we can see that the computation of task 4 dominates the computation for large dimensions d of the training pattern vectors. If d is not large enough (smaller than around 10), task 5 can become a bottleneck. Therefore, it is desirable that both task 4 and task 5 should be parallelized to train SVM efficiently.

In order to parallelize task 4, we should first analyze its data dependency. According to Eq. (6), each element of array e_i is updated independently with one training sample at a time. Thus, we can first partition the entire training samples into several smaller subsets and then update the corresponding subset of array e_i simultaneously. Thus, e_i should be partitioned accordingly. With this technique, the processing time of task 4 in parallel hardware can be reduced almost linearly.

The updating of b_{low}, i_{low}, b_{up}, and i_{up} requires iterative comparisons over the entire array e_i. Based on the fact that array e_i will be partitioned into several subsets, the local results of b_{low}, i_{low}, b_{up}, and i_{up} associated with each subset can be achieved simultaneously. The global b_{low} and b_{up} are the maximum and minimum values of all local b_{low}'s and b_{up}'s respectively, and the corresponding i_{low} and i_{up} can also be determined. Thus, task 4 can also be parallelized.

3.2 Scalable architecture

Based on the above analysis, tasks 4 and 5 can be mapped into multiple processing units to be performed in parallel. This leads to the architecture shown in Fig. 1. It is characterized by one Lagrange multiplier optimizer (LMO) and multiple error cache updating units (ECU), all being connected to the cache unit controller (CUC). To support parallel processing, each ECU has its own memory units, holding a portion of training samples. Operation phases of this architecture include device configuration, training data loading, SVM training, and outputting of the result. The global finite state machine (FSM) is presented in Fig. 2.

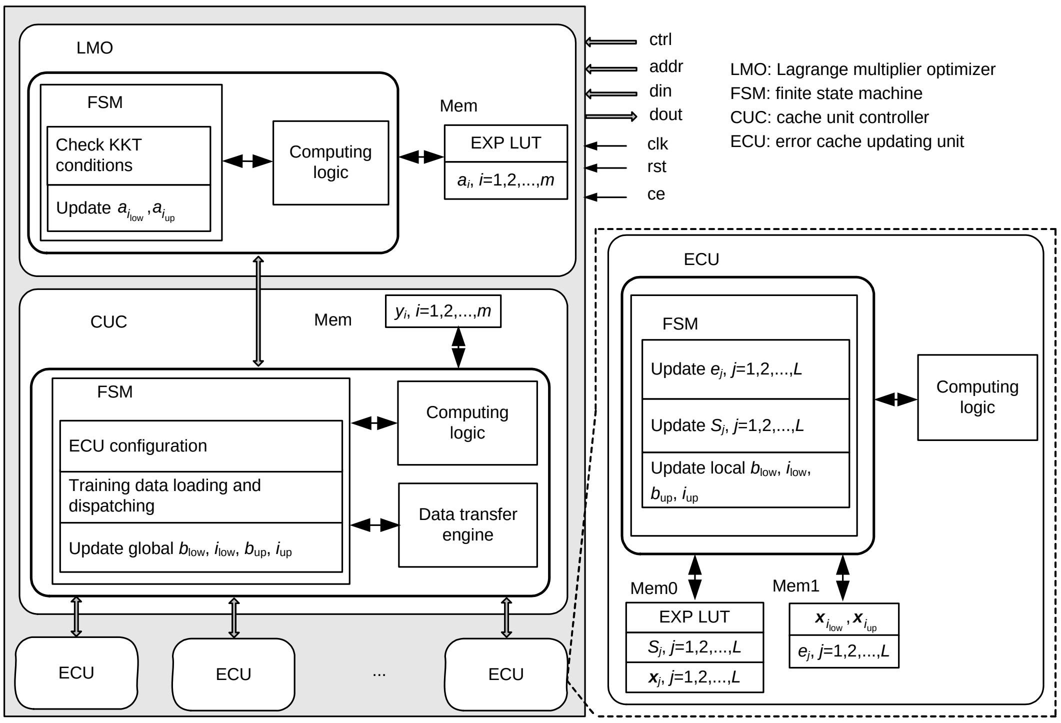
LMO
FSM
Check KKT conditions
Update $a_{i_{\rm low}}, a_{i_{\rm up}}$
Computing logic
Mem
EXP LUT
a_i, i=1,2,...,m
ctrl
addr
din
dout
clk
rst
ce
LMO: Lagrange multiplier optimizer
FSM: finite state machine
CUC: cache unit controller
ECU: error cache updating unit
CUC
Mem
y_i, i=1,2,...,m
FSM
ECU configuration
Training data loading and dispatching
Update global $b_{\rm low}$, $i_{\rm low}$, $b_{\rm up}$, $i_{\rm up}$
Computing logic
Data transfer engine
ECU
ECU
...
ECU
ECU
FSM
Update e_j, j=1,2,...,L
Update S_j, j=1,2,...,L
Update local $b_{\rm low}$, $i_{\rm low}$, $b_{\rm up}$, $i_{\rm up}$
Computing logic
Mem0
Mem1
EXP LUT
S_j, j=1,2,...,L
x_j, j=1,2,...,L
$x_{i_{\rm low}}, x_{i_{\rm up}}$
e_j, j=1,2,...,L

Fig. 1 Scalable digital architecture for training SVMs

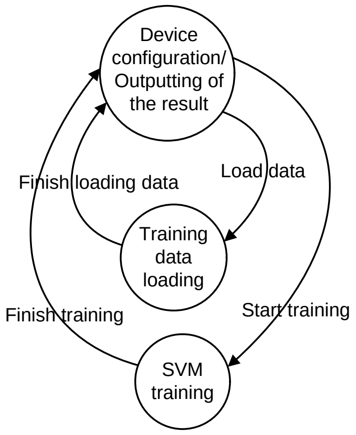

Fig. 2 Global finite state machine of the architecture

The LMO controls the entire process of training. It updates the Lagrange multipliers $a_{i_{\rm low}}$ and $a_{i_{\rm up}}$ and checks the KKT conditions at each iteration step. The indices $i_{\rm low}$, $i_{\rm up}$ and associated training samples are input by CUC. Because the Gaussian kernel evaluator is incorporated in this architecture, a lookup table (LUT) is contained in this module for exponential operations.

ECU is the basic parallel processing unit of this architecture. Tasks 4 and 5 are mapped into ECUs and performed in parallel during SVM training. The most important design consideration of this module is memory organization. To improve the processing performance, the memory unit of each ECU is divided into two blocks to support parallel memory accessing. Before SVM training, the input patterns of training samples should be stored into the ECUs, each holding a different portion of the input patterns $\boldsymbol{x}_j$, j=1,2,...,L, where L is the number of training samples in the current ECU. At each iteration step, $\boldsymbol{x}_{i_{\rm low}}$ and $\boldsymbol{x}_{i_{\rm up}}$ are loaded from the corresponding ECUs by CUC and then stored to all ECUs concurrently before updating e_j, j=1,2,...,L. $\boldsymbol{x}_{i_{\rm low}}$ and $\boldsymbol{x}_{i_{\rm up}}$ should be stored in a different memory block other than the one that holds $\boldsymbol{x}_j$'s to support concurrent memory accessing. With this design, the ECU is able to perform a multiply-accumulate (MAC) operation in a single clock cycle, and the loading and storing of $\boldsymbol{x}_{i_{\rm low}}$ and $\boldsymbol{x}_{i_{\rm up}}$ can be pipelined.

Apart from task 4, task 5 is processed in parallel with multiple ECUs. The array S_j ($S_j=k$ for $j\in I_k$, $j=1,2,...,L$; $k=0,1,...,4$) is used to denote which set the training sample belongs to, and it should be updated after updating array e_j. After arrays e_j and S_j are updated, b_{low}, i_{low}, b_{up}, and i_{up} are determined locally with the training samples in the current ECU and then submitted to CUC. Arrays S_j and e_j should also be stored in different memory blocks to support concurrent memory accessing.

CUC manages the data transfer among different modules. After all ECUs finish their updating tasks in the current iteration step, the CUC determines the final b_{low}, i_{low}, b_{up}, and i_{up} by merging the partial results achieved in each ECU. These values are then passed to the LMO module to start the next iteration step. The interconnection between CUC and ECUs is based on a bus-like structure with broadcasting ability. The CUC is the only master device on the bus, while all ECUs act as slave devices, and thus no arbiter is needed for the bus, making it simple, efficient, and easy to extend.

The task mapping and clock cycles required by each task with this architecture are presented in Table 2. The clock cycles are a combination of time for both computation and IO, with overlapped cycles removed.

Note that currently only the Gaussian kernel is implemented in the hardware, but other kernels can be incorporated into the architecture easily as well. Moreover, to reduce the hardware cost and power dissipation, fixed-point arithmetic is used in the architecture.

Table 2 Task mapping and clock cycles required by each task

Task	Description	Clock cycles	Processing unit(s)
1	Initialize: $a_i=0$, $e_i=-y_i$, $i=1, 2, ..., m$	m	LMO, ECU
2	Determine: b_{low}, i_{low}, b_{up}, and i_{up}	m/N	ECU, CUC
	Repeat		
3	Optimize $a_{i_{\text{low}}}$ and $a_{i_{\text{up}}}$	$(2d+c_1)I$	LMO
4	Update e_i, $i=1,2,...,m$	$(2dm/N+2d+c_2)I$	ECU
5	Update b_{low}, i_{low}, b_{up}, and i_{up}	$(m/N+N)I$	ECU, CUC
6	Until $i_{\text{low}}=i_{\text{up}}$ or $b_{\text{low}}<b_{\text{up}}+2\tau$	1	LMO
7	Calculate $b=(b_{\text{low}}+b_{\text{up}})/2$	1	LMO

I: number of iterations; m: number of training samples; d: dimension of the training pattern vectors; N: number of ECUs; c_1 and c_2 are some implementation-dependent constants. LMO: Lagrange multiplier optimizer; ECU: error cache updating unit; CUC: cache unit controller

4 Experiment results and comparisons

Experiments were conducted to test the effectiveness, scalability, and other characteristics of the architecture, such as the required LUT size and word length. The first set of experiments was based on a benchmarking dataset, the Sonar dataset, and the second was based on the dataset from a telecommunication problem, using SVM for the purpose of adaptive channel equalization, where a dedicated hardware can be very useful.

A bit-true and cycle-accurate model of the digital architecture was first implemented with Simulink Stateflow®, which is an efficient tool to model and simulate event-driven systems. The synthesizable Verilog code was then generated automatically from the Stateflow model with Simulink HDL Coder®. Architectural properties such as scalability and performance were evaluated based on simulation results with the Stateflow model. The Verilog code was then adapted and synthesized for FPGA to obtain the hardware related information, such as the clock rate and hardware cost.

4.1 Application to the Sonar dataset

The Sonar dataset consists of 208 patterns, each with 60 features. The task is to predict whether an object is a mine or a rock based on the features of the Sonar signals. The dataset is usually subdivided evenly into the training set and the testing set. In our experiments, the variance σ^2 of the Gaussian kernel was set to 0.5 and the regularization constant C was set to 10.

As mentioned before, the hardware was implemented with the LUTs and fixed-point arithmetic. Thus, classification performance could be affected by the LUT size and word length. Table 3 shows how classification error rates change with the LUT size and word length. The same classification error rates as those with floating-point arithmetic were obtained when the LUT size was not less than 1024 entries and the word length was not less than 16 bits, which were much less than the requirements proposed in Wee and Lee (2004).

Table 3 Classification error rates on the Sonar dataset for different LUT sizes and word lengths

Word length	Classification error rate (%)			
(bits)	S_{LUT}=256	512	1024	2048
13	5.76	8.65	6.73	6.73
14	7.69	6.73	6.73	6.73
15	6.73	5.76	5.76	5.76
16	5.76	6.73	5.76	5.76
17	7.69	5.76	5.76	5.76

S_{LUT}: lookup table size

To evaluate the scalability of the architecture, we consider two criteria, speedup and efficiency, which are defined respectively as follows:

$$\text{speedup} = \frac{\text{clock cycles with multiple ECUs}}{\text{clock cycles with a single ECU}}, \quad (7)$$

$$\text{efficiency} = \frac{\text{speedup}}{\text{number of ECUs}}. \quad (8)$$

Scaling of speedup and efficiency of the architecture with respect to a different number of ECUs on the Sonar dataset is illustrated in Fig. 3. The training procedure takes about 5 720 000 clock cycles with a single ECU, which is about 114 ms if the hardware runs at 50 MHz. The figure shows that up to 8 ECUs, the architecture scales almost linearly with the number of ECUs. After that, the speedup slows down gradually. The reason is that, according to Amdahl's law, when N parallel processing units are used, the speedup is limited to $1/(1-P+P/N)$, where P is the portion of parallelizable workload. Fortunately, pattern classification problems with a large dataset are very common, which means that the portion of parallelizable workload is usually quite high, leading to better scalability than the one obtained with this small dataset.

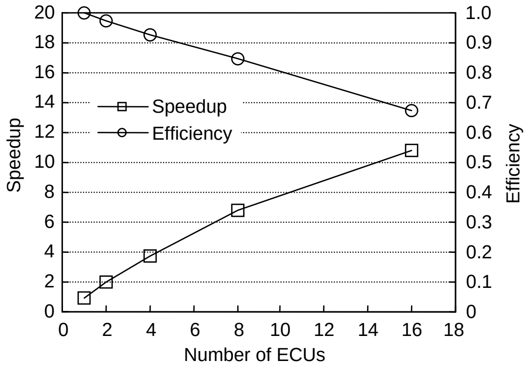

Fig. 3 Speedup and efficiency on the Sonar dataset

4.2 Application to adaptive channel equalization

As pointed by Chen *et al.* (1990), channel equalization can be treated as a classification problem. The classifier takes a vector of d consecutive channel outputs $\boldsymbol{x}_n=[x(n), x(n-1), ..., x(n-d+1)]^{\mathrm{T}}$ as the input pattern and the transmitted symbol $u(n-D)\in\{\pm1\}$ as the output class label, where D denotes the delay. The SVM-based adaptive equalizer is trained periodically with the following training sequences for channel estimation:

$$\{(\boldsymbol{x}_{n-i}, u(n-D-i))\}_{i=0}^{m-1}. \quad (9)$$

This is a typical application where a dedicated device for on-line SVM training can be effectively used.

In our experiment, the following polynomial channel, studied by Sebald and Bucklew (2000), was considered:

$$\begin{cases} \tilde{x}(n) = u(n) + 0.5u(n-1), \\ \hat{x}(n) = \tilde{x}(n) - 0.9\tilde{x}^3(n), \\ x(n) = \hat{x}(n) + e(n), \end{cases} \quad (10)$$

where $e(n)$ is an additive white Gaussian noise with variance σ_e^2.

The model with D=0, d=2, and $\sigma_e^2 = 0.2$ was used in simulation to generate training samples. The variance σ^2 of the Gaussian kernel was set to 0.5 and the regularization constant C was set to 0.25. A total of 2900 instances were generated, 500 of which were used for training, and the rest for testing.

Table 4 shows how classification error rates change with the LUT size and word length on this problem. The least classification error rate was 14.8%, slightly better than the results on the same problem reported by Anguita *et al*. (2003). Table 4 also shows that the quantization effect of the fixed-point hardware sometimes can be a benefit for generalization capability. Smaller classification error rates were obtained with the fixed-point hardware than with the floating-point simulator, where the classification error was 15.25%. This property accords with the results of Anguita *et al.* (2003).

Scaling of the speedup and efficiency of the architecture with respect to different numbers of ECUs on the channel equalization problem is illustrated in

Fig. 4. The training procedure takes about 3653000 clock cycles with a single ECU, which is about 73 ms if the hardware runs at 50 MHz. Fig. 4 also shows that the scalability of the architecture on this problem is better than that on the Sonar dataset. This occurs because this problem has more training samples, and the portion of parallelizable computation is greater. As the size of the problem goes up, better scalability can be achieved with this architecture.

Table 4 Classification error rates on the channel equalization problem for different LUT sizes and word lengths

Word length	Classification error rate (%)				
(bits)	S_{LUT}=256	512	1024	2048	4096
13	15.0	15.0	14.8	15.4	15.3
14	15.0	15.2	15.0	14.9	15.0
15	15.0	15.0	15.0	15.0	15.1
16	15.2	15.0	15.0	15.0	15.2
17	15.2	15.2	15.1	15.0	15.0

S_{LUT}: lookup table size

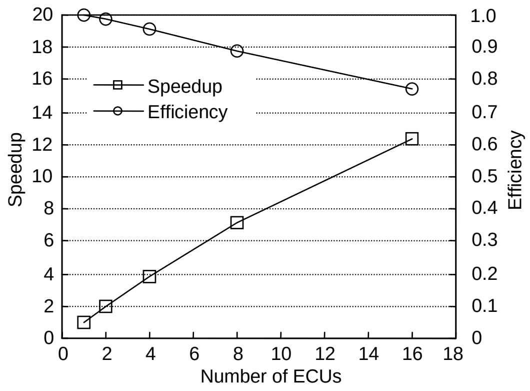

Fig. 4 Speedup and efficiency on the channel equalization problem

4.3 FPGA synthesis results

The Verilog code generated from the Stateflow model of the architecture was adapted for FPGA synthesis. It was designed in a parameterized way, where the number of ECUs can be defined with macros. In our current implementation, the CUC can work with up to 16 ECUs, but it can be extended easily to support more ECUs. The design was synthesized for the Xilinx Virtex-4 XC4VLX100 FPGA, which contains 240 18-kb block RAMs, 49152 logic slices, and 96 XtremeDSP$^{\text{TM}}$ slices for arithmetic operation. Critical paths were extracted, indicating that the hardware was able to run at more than 75 MHz.

Fig. 5 shows how the hardware cost changes with the number of ECUs incorporated in the hardware. The LUT size was set to 1024 entries, and the word length was set to 16 bits. The ECU was configured with different sizes of memory when different numbers of ECUs were defined. A training set with up to 256 patterns, each having up to 64 features, could be handled with the implemented SVM hardware. By increasing the memory size of each module, larger training sets can be handled with this architecture. If the word length is 18 bits and the dimension of the training sample is 16, a maximum of about 12000 training samples can fit into this FPGA. This is usually sufficient for adaptive applications with tight latency constraints, to which the proposed architecture is targeted. Too many training samples may lead to very long training time, which is undesirable for such applications.

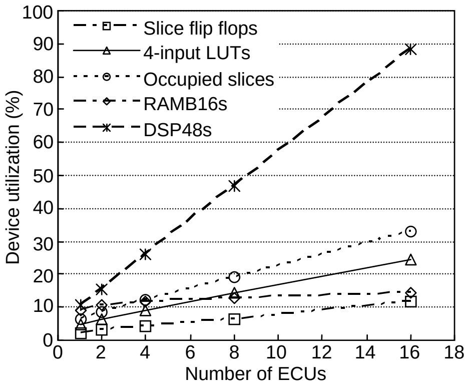

Fig. 5 Device utilization of the architecture with a different number of ECUs

4.4 Comparisons

Comparisons of the proposed architecture to the previously proposed ones are presented in Table 5. Anguita *et al.* (2003) trained SVM in a fully parallel fashion, where the number of PEs should be no less than the number of training samples. This kind of architecture is difficult to scale with a large number of training samples, because too many PEs would be needed. Moreover, this architecture stores the kernel matrix $\boldsymbol{K}$ explicitly, where $K_{ij}=k(\boldsymbol{x}_i, \boldsymbol{x}_j)$, but it is prohibitive to do so for large-scale problems because the number of elements of $\boldsymbol{K}$ increases quadratically with the number of training samples. The architecture of Wee and Lee (2004) is based on a simplified SMO algorithm, which can handle only a non-standard type

Table 5 Comparisons among several digital architectures for training SVMs

Architecture	Algorithm	Parallelization	Hardware resource usage	Clock cycles per iteration
Anguita *et al*. (2003)	DSVM with fixed bias	Fully parallel	Depends on the number of PEs	$m+c_1$
Wee and Lee (2004)	Kernel Adatron	Fully parallel	Depends on the number of PEs	$d+c_2$
Choi *et al*. (2006)	Gauss-Jordan elimination	Serial	Depends on the memory size	Variable
This paper	SMO	Customizable	Depends on the number of ECUs and the memory size	$4d+(2d+1)m/N+c_3$

m: number of training samples; *d*: dimension of the training pattern vectors; *N*: number of ECUs; c_1, c_2, and c_3 are some implementation-dependent constants. DSVM: digital support vector machine; SMO: sequential minimal optimization; PE: processing element; ECU: error cache updating unit

of SVM where the bias *b* is set to zero. This architecture suffers from the same drawback as the one of Anguita *et al*. (2003) because of its fully parallel nature. Choi *et al*. (2006) trained SVM by solving a system of linear equations with the Gauss-Jordan elimination algorithm; no parallel processing technique is used in this architecture and the kernel matrix is also required to be stored explicitly.

The architecture proposed in this paper is based on the popular and mature SMO algorithm. It is designed with flexibility and scalability in mind, and thus is more suitable to be used in embedded environments, where a trade-off is usually needed to balance the performance and hardware costs.

5 Conclusions

In this paper, a parallel and scalable architecture based on the SMO algorithm for training SVMs is proposed, tested, and mapped to an FPGA chip. To the best of our knowledge, this is the first time Keerthi's SMO algorithm has been implemented with digital hardware. By taking advantage of the popular SMO algorithm, which avoids the numerical instability issues, inexpensive fixed-point arithmetic can be used with the proposed architecture to solve SVM training problems effectively. Experiments show that the performance of this architecture scales well with respect to the number of the parallel processing units. The drawbacks of the previously proposed SVM hardware that lacks flexibility are overcome in this scalable architecture based on the SMO algorithm, thus making this architecture more suitable to be used in embedded environments. Future work will focus on the optimization of the ECU module to make it more cost-effective.

References

Anguita, D., Boni, A., Ridella, S., 2003. A digital architecture for support vector machines: theory, algorithm, and FPGA implementation. *IEEE Trans. Neur. Networks*, **14**(5):993-1009. [doi:10.1109/TNN.2003.816033]

Anguita, D., Pischiutta, S., Ridella, S., Sterpi, D., 2006. Feed-forward support vector machine without multipliers. *IEEE Trans. Neur. Networks*, **17**(5):1328-1331. [doi:10.1109/TNN.2006.877537]

Biasi, I., Boni, A., Zorat, A., 2005. A Reconfigurable Parallel Architecture for SVM Classification. Proc. IEEE Int. Joint Conf. on Neural Networks, p.2867-2872. [doi:10.1109/IJCNN.2005.1556380]

Burges, C.J.C., 1998. A tutorial on support vector machines for pattern recognition. *Data Min. Knowl. Discov.*, **2**(2):121-167. [doi:10.1023/A:1009715923555]

Catanzaro, B., Sundaram, N., Keutzer, K., 2008. Fast Support Vector Machine Training and Classification on Graphics Processors. Proc. 25th Int. Conf. on Machine Learning, p.104-111. [doi:10.1145/1390156.1390170]

Chen, S., Gibson, G.J., Cowan, C.F.N., Grant, P.M., 1990. Adaptive equalization of finite nonlinear channels using multilayer perceptrons. *EURASIP Signal Process.*, **20**(2): 107-119.

Choi, W.Y., Ahn, D., Pan, S.B., Chung, K.I., Chung, Y.W., Chung, S.H., 2006. SVM-based speaker verification system for match-on-card and its hardware implementation. *ETRI J.*, **28**(3):320-328. [doi:10.4218/etrij.06.0105.0022]

Frieß, T.T., Cristianini, N., Campbell, C., 1998. The Kernel-Adatron Algorithm: A Fast and Simple Learning Procedure for Support Vector Machines. Proc. 15th Int. Conf. on Machine Learning, p.188-196.

Graf, H.P., Cadambi, S., Durdanovic, I., Jakkula, V., Sankaradass, M., Cosatto, E., Chakradhar, S.T., 2008. A Massively Parallel Digital Learning Processor. 22nd Annual Conf. on Neural Information Processing Systems, p.529-536.

Keerthi, S.S., Shevade, S.K., Bhattacharyya, C., Murthy, K.R.K., 2001. Improvements to Platt's SMO algorithm for SVM classifier design. *Neur. Comput.*, **13**(3):637-649. [doi:10.1162/089976601300014493]

Manikandan, J., Venkataramani, B., Avanthi, V., 2009. FPGA Implementation of Support Vector Machine Based Isolated Digit Recognition System. Proc. 22nd Int. Conf. on VLSI Design, p.347-352. [doi:10.1109/VLSI.Design.2009.23]

Platt, J.C., 1999. Fast Training of Support Vector Machines Using Sequential Minimal Optimization. *In*: Schölkopf, B., Burges, C., Smola, A. (Eds.), Advances in Kernel Methods: Support Vector Learning. MIT Press, Cambridge, MA, p.185-208.

Schölkopf, B., Burges, C.J.C., Smola, A.J., 1999. Advances in Kernel Methods: Support Vector Learning. MIT Press, Cambridge, MA, p.1-16.

Sebald, D.J., Bucklew, J.A., 2000. Support vector machine techniques for nonlinear equalization. *IEEE Trans. Signal Process.*, **48**(11):3217-3226. [doi:10.1109/78.875477]

Sun, Z., Zhang, L., Tang, E., 2005. An incremental learning method based on SVM for online sketchy shape recognition. *LNCS*, **3610**:655-659. [doi:10.1007/11539087_82]

Vapnik, V.N., 1998. Statistical Learning Theory. Wiley, New York, p.493-520.

Wee, J.W., Lee, C.H., 2004. Concurrent support vector machine processor for disease diagnosis. *LNCS*, **3316**:1129-1134. [doi:10.1007/b103766]