2014 International Joint Conference on Neural Networks (IJCNN)
July 6-11, 2014, Beijing, China

Hardware Implementation of KLMS Algorithm using FPGA

Xiaowei Ren, Pengju Ren, Badong Chen, Tai Min and Nanning Zheng

***Abstract*— Fast and accurate machine learning algorithms are needed in many physical applications. However, the learning efficiency is badly subjected to the intensive computation. Knowing that hardware implementation could speed up computation effectively, we use a FPGA hardware platform to implement an on-line kernel learning algorithm, namely the kernel least mean square (KLMS) which adopts the simple survival kernel as the Mercer kernel. By using an on-line quantization method and pipeline technology, the requirement of hardware resources and computation burden can be reduced significantly and the data processing speed can be accelerated apparently without losing accuracy. Finally, a 128-way parallel FPGA platform which works at 200MHz is implemented. It could achieve an average speedup of 6553 versus Matlab running on a 3GHz Intel(R) Core(TM) i5-2320 CPU.**

I. INTRODUCTION

KERNEL adaptive filters (KAFs) [1] are a family of nonlinear adaptive filtering algorithms, which have been applied to machine learning [2] and signal processing [3] successfully during the past few years, including KLMS [4] [5], kernel recursive least squares (KRLS) [6] and kernel affine projection algorithms (KAPAs) [7] etc. Among these algorithms, KLMS is the simplest, which is easy to implement without losing effectiveness.

However, when we make use of kernel adaptive filters, two critical issues should be concerned cautiously. The first one is to choose a dedicated kernel, such as Gaussian kernel [8] and multiple-kernel [9], to ensure good performance of the algorithms. The second one is that all kernel adaptive filters suffer from the constantly growing network size, leading to a serious memory and computation burden. Approximate linear dependency criterion (ALD) [6], surprise criterion (SC) [10], prediction variance criterion [11] and quantization methods [12] are some main techniques that have been put forward to constrain the network size.

While various kinds of techniques have been put forward to reduce the complexity of machine learning methods, intensive computation is still the critical restriction of on-line (real-time) learning. Note that hardware devices could accelerate mathematical operations in orders of magnitude [13] [14] [15], we consider to implement some algorithms with hardware platform, instead of conventional software methods. Meanwhile, some specific work have been done to improve the performance of software algorithms with hardware implementation. For example, in order to deal with the intensive computation, a VLSI of dynamic codebook generator and encoder for image compression applications is described in [16]. Furthermore, a VLSI is realized in [17] to make the HVQ (hierarchical vector quantization) cost-effective and computationally efficient.

In this paper, we implement a FPGA processing element (PE) of KLMS. The kernel what we choose is a new Mercer kernel, namely the survival kernel [18], which is suitable for on-line KLMS because it is parameter-free and computationally simple. Meanwhile, we adopt a quantization approach [12] (so this new KLMS is called QKLMS) to relax the memory and computation burden yet guarantee the accuracy of algorithm. Moreover, pipeline technology [19] [20] is applied to explore the concurrency in or between each operation and augment resource reuse rate. Finally, at very low hardware cost, we finish the implementation of a parallel FPGA platform which is used to process 128-way data training simultaneously. When it works at 200MHz, it's 6553 times faster than Matlab running on a 3GHz Intel(R) Core(TM) i5-2320 CPU.

The remained part of this paper is organized as follows. Section II gives a brief description of the KLMS algorithm, quantization method and survival kernel. The architecture of processing element is elaborated in section III. In section IV, performance evaluation and implementation results are shown. Finally, this work is concluded in section V.

II. KLMS, QKLMS AND SURVIVAL KERNEL

In this section, we will present some basic background information related to our work. The first one is KLMS, a kernel learning method what we implement with FPGA in this paper. Then a quantization approach QKLMS used to reduce the network size is briefly described. We also introduce the survival kernel that we choose.

A. KLMS

In fact, KLMS is a stochastic gradient algorithm to solve the least-square (LS) problem in reproducing kernel Hilbert spaces (RKHS). A Mercer kernel is a continuous, symmetric, positive-definitive function defined on $X \times X$, i.e. $\kappa : X \times X \rightarrow \mathbb{R}$. It could be expressed in the formula of $\kappa(x_m, x_n)$. By the Mercer's theorem, any Mercer kernel induces a mapping Ψ between input space X and a feature space F (which is an inner product space) such that:

$$\kappa(x_m, x_n) = \Psi(x_m)^T \Psi(x_n) \tag{1}$$

Xiaowei Ren, Pengju Ren, Badong Chen and Nanning Zheng are with the Institute of Artificial Intelligence and Robotics, Xi'an Jiaotong University, 28 Xianning West Road, Xi'an 710049, China (email: {renxiaowei66, pengjuren}@gmail.com, {chenbd, nnzheng}@mail.xjtu.edu.cn).

Tai Min is with the IMEC, Kapeldreef 75, B-3001 Leuven, Belgium (email: {tmdfz}@hotmail.com).

This research was supported by NSFC grant No.61372152 and No.610303036, China Postdoctoral Science Foundation No.2012M521777, Specialized Research Fund for the Doctoral Program of Higher Education of China No.20130201120024, Natural Science Basic Research Plan in Shaanxi Province of China No.2013JQ8029 and the Fundamental Research Funds for the Central Universities.

978-1-4799-1484-5/14/$31.00 ©2014 IEEE

If we identify $\Psi(x) = \kappa(x, .)$, feature space F is actually the same as RKHS induced by the kernel. Then, the KLMS is the LMS algorithm performed on the example sequence $\{(\Psi(x_1), y_1), ..., (\Psi(x_n), y_n)\}$, which is expressed as follows:

$$\begin{cases} f_0 = 0 \\ e(n) = d(n) - f_{n-1}(x_n) \\ f_n = f_{n-1} + \eta e(n)\Psi(x_n) \end{cases} \tag{2}$$

where $e(n)$, $d(n)$, f_n are the prediction error, desired signal and learned nonlinear mapping at iteration n respectively, η is the step size. We can get the access of $f_{n-1}(x_n)$ through:

$$f_{n-1}(x_n) = \eta \sum_{j=1}^{n-1} e(j)\kappa(x_j, x_n) \tag{3}$$

Once an input sample comes, we need to allocate a new kernel unit for input space with x_i as the center and $\eta e(i)$ as the corresponding coefficient. That's to say, we will get a continuously growing radial basis function (RBF) network during the period of data training. This tricky issue requires significant memory and computation burden, a solution should be found to solve this problem.

B. QKLMS

As we said above, a technique that is able to compact the RBF network structure of kernel adaptive filter is urgently demanded. One can apply a quantization method to KLMS (called QKLMS) so that the network size (the number of centers) can be significantly decreased [12].

The key problems in quantization method including: 1) how to decide whether a new input data should be omitted or not, and 2) how to update the existing centers and their coefficients. So far, there have been many quantization algorithms in literature [21] [22] [23] [24], but their off-line training of codebook (dictionary) does not lend them fitness for on-line implementation. A quantization method in [12] could train codebook directly from on-line samples and is adaptively growing.

As for the first key issue said above, the distance between the new input and the current codebook could determine whether this input could be discarded or not. Let $C(n)$ and $a(n)$ be the codebook and its corresponding coefficient vector after the n^{th} iteration, the distance between a new input x_n and codebook $C(n-1)$ could be calculated as:

$$dis(x_n, C(n-1)) = \min_{1 \le j \le size(C(n-1))} \|x_n - C_j(n-1)\| \tag{4}$$

where $C_j(n-1)$ is the jth element of $C(n-1)$.

When we get the distance, the next-step is to update the codebook and the corresponding coefficient according to the distance. If $dis(x_n, C(n-1))$ is greater than the threshold ε_X, it means that x_n is not a "redundant" data. Then the codebook and coefficient should be updated as:

$$C(n) = \{C(n-1), x_n\} \tag{5}$$

$$a(n) = \{a(n-1), \eta e(n)\} \tag{6}$$

Oppositively, if $dis(x_n, C(n-1))$ is less than the threshold ε_X, it means that x_n is closely related to the existing codebook $C(n-1)$ and x_n is a "redundant" data. Then the only work we need to do is to update the coefficient vector $a(n-1)$ as follows:

$$a_{j^*}(n) = a_{j^*}(n-1) + \eta e(n) \tag{7}$$

$$j^* = arg \min_{1 \le j \le size(C(n-1))} \|x_n - C_j(n-1)\| \tag{8}$$

where $a_{j^*}(n-1)$ is the j^*th element of coefficient $a(n-1)$. Other elements of $a(n-1)$ keep unchanged. Finally, the output of the kernel adaptive filter could be computed as:

$$f_n(x_n) = \sum_{j=1}^{size(C(n-1))} a_j(n-1)\kappa(C_j(n-1), x_n) \tag{9}$$

It's obvious that the threshold ε_X is a very important parameter for QKLMS. However, there is not a definite way to determine the value for it. In general, we should make a tradeoff between accuracy and complexity(network size). It depends on the specific applications and cross validation is an effective way to select the ε_X.

C. Survival Kernel

We select survival kernel to implement in our FPGA platform. Assuming $X = \mathbb{R}_+^m$, where $\mathbb{R}_+^m = \{\mathbf{x} \in \mathbb{R}^m : \mathbf{x} = (x_1, ..., x_m), x_i > 0, i = 1, ..., m\}$. The survival kernel is defined by [18]:

$$\kappa_{sur}(\mathbf{x}, \mathbf{y}) = \int_{\mathbb{R}_+^m} (\boldsymbol{I}(\mathbf{x} > \mathbf{t})\boldsymbol{I}(\mathbf{y} > \mathbf{t}))d\mathbf{t} \tag{10}$$

where $\boldsymbol{I}(.)$ denotes the indicator function and if $x_i > t_i, i = 1, ..., m$, we say that $\mathbf{x} > \mathbf{t}$. The survival kernel of (10) can also be expressed as:

$$\kappa_{sur}(\mathbf{x}, \mathbf{y}) = \prod_{i=1}^{m} min(x_i, y_i) \tag{11}$$

If we reduce input space X to a uni-dimensional space, (11) will become:

$$\kappa_{sur}(x, y) = min(x, y) \tag{12}$$

For the sake of simplicity, we only implement the one-dimensional survival kernel in this paper.

Remark 1: The survival kernel is strictly positive-definitive (SPD), parameter-free and easy to compute (just by operations of comparing and multiplication), it has great potential to be used in on-line kernel learning. In the present work, we focus mainly on the KLMS.

Remark 2: Even though survival kernel is defined on $\mathbb{R}_+^m$, we can still apply it on $\mathbb{R}^m$. Because the sample data of physical applications are always bounded, we can get positive data through simple translation.

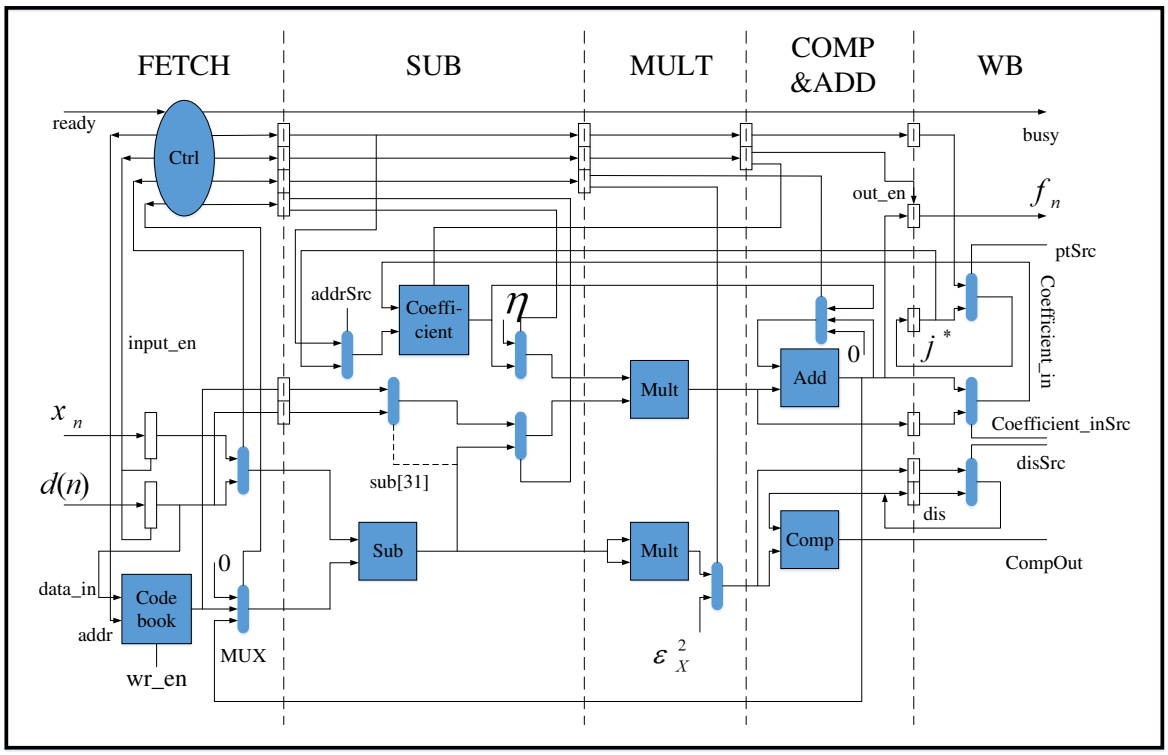

Fig. 1. Microarchitecture of PE

III. HARDWARE IMPLEMENTATION

High efficiency and throughput are the primary concerns for our hardware implementation. QKLMS algorithm and pipeline technology are employed to reduce the size of required memory and enhance the reuse efficiency of functional units. Meanwhile, benefiting from the usage of pipeline technology, concurrent data training procedure is allowed, accelerating the data training rate. Firstly, we will introduce the microarchitecture of our proposed FPGA PE. Secondly, data training procedure with our PE is explained step by step.

A. Microarchitecture of PE

In our PE, all the data are single precision float with the format of IEEE-754 standard, so all data registers are $32bits$. Figure 1 shows that our FPGA PE is composed of two RAMs, one control unit and five functional units. These hardware devices are arranged into five pipeline stages, FETCH, SUB, MULT, COMP&ADD and WB are included.

Two RAMs are used to realize the codebook and its corresponding coefficient. Simulation results show that if we could set a reasonable threshold ε_X, QKLMS [12] could reduce the size of codebook and coefficient to a constant around 80 still with very high accuracy, regardless of the time-length of data training period. However, memory size needed in conventional KLMS is proportional to the data training iterations. More precisely, if we conduct 1000-time data training, the required memory size should be at least $1000 \times 32bits$ (both codebook and coefficient RAM is $4KB$,$8KB$ in summary). Worse still, the size of RAMs will be unaffordable as the time-length increases dramatically. Hence, QKLMS makes a considerable reduction of memory consumption, especially for a long data training interval.

Control unit implemented by finite state machine (FSM) is in charge of the generation of different control signals during the procedure of data training. At the beginning of data training, it's responsible for the initialization of codebook and coefficient RAMs. Then, it guides functional units to calculate the learned result $f_n(x_n)$ and the distance from input x_n to codebook $C(n-1)$. After that, the codebook and coefficient are updated if required, the process continues until the data training is accomplished.

As shown in Figure 1, all pipeline stages are named after operations they conduct. It's obvious that input data x_n and desired learning result $d(n)$ go into the circuit system through FETCH stage whenever PE is available. Stage SUB makes all subtraction operations needed during data training, such as the computation of learning error $e(n)$ and the operation $(x_n - C_j(n-1))$, the latter is prepared for the calculation of $(x_n, C_j(n-1))^2$ in stage MULT. At the same time, SUB is also responsible for the implementation of survival kernel, i.e. the comparison between x_n and $C_j(n-1)$ as shown in

formula (12). The less one is selected by the most significant bit (MSB) of subtraction result. If the result is positive and MSB is zero, it means that x_n is larger than $C_j(n-1)$. Therefore, by using a subtractor to substitute the comparator need in formula (12), an extra comparator is saved. All accumulation items of learning result $f_n(x_n)$ in equation (9) and $(x_n - C_j(n-1))^2$ which is the square of distance from x_n to $C_j(n-1)$ are obtained from stage MULT. Other multiplication operations, such as $\eta e(n)$, are also made in MULT. Two primary works are done in the COMP&ADD stage. The first one is to complete the accumulation of learned result and the second one is to evaluate $dis(x_n, C(n-1))$ and j^* in formula (4) and (8) with a comparator. In addition, the comparison between $dis(x_n, C(n-1))$ and ε_X is also performed with the comparator. In the last stage of PE, we finish the updating of codebook and coefficient according to formula (5), (6) and (7). If x_n is "redundant", i.e. $dis(x_n, C(n-1))$ is smaller than ε_X, we only update the coefficient. Otherwise, codebook and coefficient are updated with x_n and $\eta e(n)$ respectively.

B. Data Training Procedure

After we introduced the microarchitecture of PE, let's take a look at how data are trained in the system. Data training procedure can be divided into three steps in coarse granularity.

Step 1. Initialization: At the beginning, when the circuit is turned on, we should initialize the codebook and coefficient with x_1 and $\eta d(1)$ respectively. In order to get $\eta d(1)$, operation "$d(1) - 0$" is carried out with subtraction unit and the result is multiplied by η at stage MULT. Then, control unit enables x_1 and $\eta d(1)$ to be written in codebook and coefficient as the first element.

Step 2. Evaluation of $f_n(x_n)$, j^* and $dis(x_n, C(n-1))$: Theoretically, these three results are calculated sequentially. However, we plan to get them simultaneously so as to accelerate data training rate. This performance improvement is achieved by adding one more multiplier in stage MULT and let adder and comparator work concurrently in stage COMP&ADD. Furthermore, not only the parallelism among these three calculations is developed, but also the concurrency of each computation is exploited because of the adoption of pipeline technology.

Following the first data, subsequent input data are trained in sequential order. Here, we will illustrate this step with x_n. Firstly, x_n is subtracted by all elements of codebook $C(n-1)$. We choose the smaller one according to the MSB of subtraction result after each comparison. If the MSB is "1", x_n is our option, otherwise the compared element of codebook is selected out. Then the smaller one is multiplied with the corresponding coefficient. Functional unit adder will accumulate each product and output the final learning result $f_n(x_n)$.

According to formula (4) and formula (8), we need to calculate $dis(x_n, C(n-1))$ and find out the location of one element which has the least distance to x_n among the codebook, i.e. the value of j^*. Note that $dis(x_n, C(n-1))$ is only used to compare with threshold ε_X and the square of a positive dataset has the same monotonicity as itself, so we only need to compute the square of distance instead of calculating the distance itself. Only $(x_n - C_j(n-1))^2$ for every n and j is calculated and compared. Then we could avoid the design of a functional unit used to extract the square root of $(x_n - C_j(n-1))^2$ and save more hardware cost. Meanwhile, because we cut down an operation, data training could be faster. If the new value of distance is smaller than the prior one, the older distance is replaced and the value of j^* is updated. Otherwise, they all keep their original values.

Step 3. Updating: After we get the result of $dis(x_n, C(n-1))$, we can determine whether need to update the codebook and coefficient. But before the updating operation, the value of $e(n)$, $\eta e(n)$ and $a_{j^*}(n-1) + \eta e(n)$ need to be calculated firstly. These three values are computed sequentially by pipeline stage SUB, MULT and COMP&ADD. Then we should compare the square of $dis(x_n, C(n-1))$ and ε_X^2. In the light of comparison result, updating of codebook and coefficient are conducted. If $dis(x_n, C(n-1))$ is the smaller one, codebook keep unchanged and the j^*th element of coefficient is added with $\eta e(n)$. Otherwise, we have to allocate a new space in codebook and coefficient to store x_n and $\eta e(n)$ just as described in formula (5) and (6).

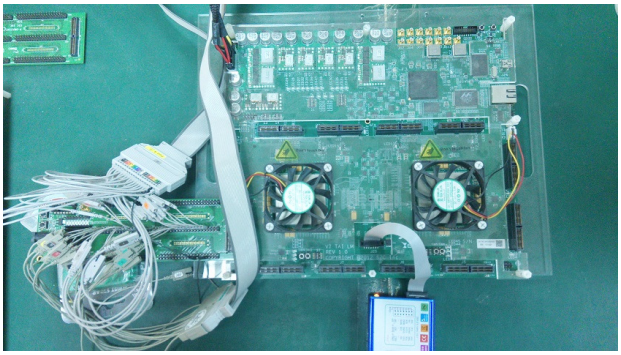

Fig. 2. FPGA platform

IV. EVALUATION

This section presents the evaluation results to illustrate the accuracy and data training rate of our FPGA implementation. The FPGA platform we used is Xilinx Virtex-7 XC7V2000T as shown in Figure 2, the state-of-the-art FPGA device.

A. Accuracy of Hardware Implementation

At first, we use a cosine function to testify the correctness and accuracy of our processing element. The desired data are generated by:

$$d(n) = cos(\pi x_n) + v(n) \tag{13}$$

Where input x_n is uniformly distributed over [2,4], and a zero-mean white Gaussian noise $v(n)$ with variance 0.001 is added. ε_X^2 and η are set to $1/1024$ and 0.25 respectively. The learned and desired mappings are plotted in Figure 3, prediction error of every data training is plotted in Figure 4.

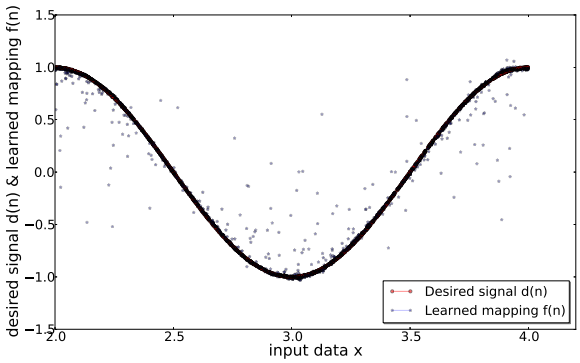

Fig. 3. Learned and desired mappings

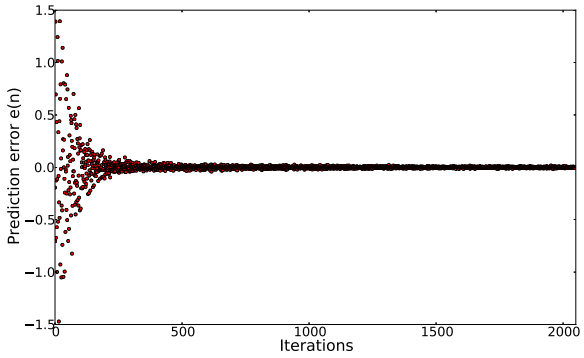

Fig. 4. Prediction error of every training iteration

In the experiment, 2000-time data trainings are processed. In Fig.3, red dots represent the desired mappings and the learned signals are shown with blue stars. As the data training continues, blue stars becomes brighter and brighter. It is evident that the blue stars which deviate from the desired mappings are all lightly colored. Also, with the proceeding of data training, learned signals are matched with desired mappings more and more exactly. The curve of desired mappings has even been fully covered by brighter blue stars. We also record the variation trend of prediction error in every data training, as shown in Fig. 4. At the beginning of data training, prediction error vibrate in the range of [-1.5,1.5]. But after a very short period of time, prediction error is converged to zero precisely and keep this trend during the following part of data training. As been observed, the hardware implementation of the QKLMS algorithm functions accurately.

B. Speedup versus Software

Furthermore, by simply instantiating our PE 128 times, a hardware platform which could process 128-way data learning in parallel is achieved. ε_X^2 and η are set to $1/1024$ and 0.25 respectively for every PE. Then we make an experiment in which 128 data trainings are needed to complete with our hardware platform and Matlab respectively. By the comparison of their execution time for the different training iterations, we can know about how many times our hardware platform is faster than Matlab. In our 128-way parallel hardware platform, these data trainings could be performed simultaneously. Even though Matlab runs on a computer which is configured with $2GB$ main memory and Intel(R) Core(TM) i5-2320 CPU that works at 3.00GHz in the experiment, it still cannot achieve the full parallelism like us. Our parallel hardware platform works at 200MHz. Table I shows that an average speedup of 6553 is achieved with our hardware platform. Furthermore, the hardware platform performs well in real-time at a sample rate of 2.4MHz approximately.

TABLE I
SPEEDUP OF THE PARALLEL HARDWARE VERSUS MATLAB

Iterations	FPGA(ms)	Matlab(ms)	Speedup
1024	0.400	2681.429	6704
2048	0.830	5524.324	6656
4096	1.690	11032.853	6528
8192	3.410	22007.986	6454
16384	6.850	43987.562	6422

C. Implementation Results

The proposed 128-way parallel hardware platform is implemented in Verilog and synthesized with Xilinx EDA tool called Vivado. Table II summarizes our hardware cost. 29.72% block RAMs are used to store the codebook and the coefficient. This two memories in each PE could record as much as 1024 data items. All functional units are implemented by $32bits$ float DSP IPs, and 47.41% DSP devices is sacrificed. Besides, the hardware platform requires merely 3.23% Flip-flop and 10.09% Lookup Table (LUT) which are two main resources of FPGA. Therefore, we can definitively claim that a hardware platform of KLMS algorithm has been implemented at very low cost without losing accuracy.

TABLE II
IMPLEMENTATION RESULTS OF FPGA

Resources	Utilized	Available	Utilization Rate(%)
Flip-Flop	78976	2443200	3.23
LUT	123264	1221600	10.09
Memory LUT	768	344800	0.22
I/O	100	1200	8.33
BRAM	384	1292	29.72
DSP48	1024	2160	47.41
BUFG	2	128	1.562
MMCM	1	24	4.167

V. CONCLUSION

In this paper, the KLMS algorithm used for machine learning is implemented with FPGA at a very low hardware cost. The survival kernel we used is a parameter-free, strictly positive definitive and simple Mercer kernel. Additionally, benefiting from the adoption of a quantization method, we reduce the burden of memory requirement and computation significantly. Moreover, pipeline technology is

used to enhance the hardware reuse efficiency and operation concurrency. When the 128-way parallel hardware platform works at 200MHz, an average speedup of 6553 versus Matlab running on a 3GHz Intel(R) Core(TM) i5-2320 processor is achieved. Meanwhile, benefiting from such a large speedup, less time will be spent for cross validation which is used to select the ε_X as described in the last paragraph of section II.B.

There are many work that need to be done in the future. The first one is to optimize the microarchitecture of our PE in order to get higher parallelism and speedup. For example, if we could use the median comparison method to reduce the number of comparison in survival kernel, the PE will be sped up much more. As for the 128-way parallel hardware platform, we want to apply it in the on-line analysis of electroencephalograms (EEG) signals through processing different data training in parallel, such as the detection of causality among EEG signals.

VI. ACKNOWLEDGMENT

We thank all the past and present members of our team, they give us many precious suggestions and discussions. We also thank all the foundations that give us important support. Meanwhile, the authors will appreciate it if the reviewers could point out some related references what we had overlooked.

REFERENCES

[1] J. C. Príncipe, W. Liu, and S. Haykin, *Kernel Adaptive Filtering: A Comprehensive Introduction*. John Wiley & Sons, 2011, vol. 57.

[2] J. Kivinen, A. J. Smola, and R. C. Williamson, "Online learning with kernels," *Signal Processing, IEEE Transactions on*, vol. 52, no. 8, pp. 2165–2176, 2004.

[3] H. Takeda, S. Farsiu, and P. Milanfar, "Kernel regression for image processing and reconstruction," *Image Processing, IEEE Transactions on*, vol. 16, no. 2, pp. 349–366, 2007.

[4] W. Liu, P. P. Pokharel, and J. C. Principe, "The kernel least-mean-square algorithm," *Signal Processing, IEEE Transactions on*, vol. 56, no. 2, pp. 543–554, 2008.

[5] P. Bouboulis and S. Theodoridis, "Extension of wirtinger's calculus to reproducing kernel hilbert spaces and the complex kernel lms," *Signal Processing, IEEE Transactions on*, vol. 59, no. 3, pp. 964–978, 2011.

[6] Y. Engel, S. Mannor, and R. Meir, "The kernel recursive least-squares algorithm," *Signal Processing, IEEE Transactions on*, vol. 52, no. 8, pp. 2275–2285, 2004.

[7] W. Liu and J. Príncipe, "Kernel affine projection algorithms," *EURASIP Journal on Advances in Signal Processing*, vol. 2008, no. 1, p. 784292, 2008.

[8] J. C. Principe, *Information theoretic learning: Rényi's entropy and kernel perspectives*. Springer, 2010.

[9] M. Gönen and E. Alpaydın, "Multiple kernel learning algorithms," *The Journal of Machine Learning Research*, vol. 999999, pp. 2211–2268, 2011.

[10] W. Liu, I. Park, and J. C. Príncipe, "An information theoretic approach of designing sparse kernel adaptive filters," *Neural Networks, IEEE Transactions on*, vol. 20, no. 12, pp. 1950–1961, 2009.

[11] L. Csató and M. Opper, "Sparse on-line gaussian processes," *Neural Computation*, vol. 14, no. 3, pp. 641–668, 2002.

[12] B. Chen, S. Zhao, P. Zhu, and J. C. Principe, "Quantized kernel least mean square algorithm," *Neural Networks and Learning Systems, IEEE Transactions on*, vol. 23, no. 1, pp. 22–32, 2012.

[13] H. Shojania and B. Li, "Parallelized progressive network coding with hardware acceleration," in *Quality of Service, 2007 Fifteenth IEEE International Workshop on*. IEEE, 2007, pp. 47–55.

[14] M. Edwards and J. Forrest, "Software acceleration using programmable hardware devices," in *Computers and Digital Techniques, IEE Proceedings-*, vol. 143, no. 1. IET, 1996, pp. 55–63.

[15] M. Hopf and T. Ertl, "Accelerating 3d convolution using graphics hardware," in *Visualization'99. Proceedings*. IEEE, 1999, pp. 471–564.

[16] K. Tsang and B. W. Wei, "A vlsi architecture for a real-time code book generator and encoder of a vector quantizer," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, no. 3, pp. 360–364, 1994.

[17] M. Bracco, S. Ridella, and R. Zunino, "Digital implementation of hierarchical vector quantization," *Neural Networks, IEEE Transactions on*, vol. 14, no. 5, pp. 1072–1084, 2003.

[18] B. Chen, N. Zheng, and J. C. Principe, "Survival kernel with application to kernel adaptive filtering," in *Neural Networks (IJCNN), The 2013 International Joint Conference on*. IEEE, 2013, pp. 1–6.

[19] J. F. Duluk Jr, R. E. Hessel, V. T. Arnold, J. Benkual, J. P. Bratt, G. Cuan, S. L. Dodgen, E. S. Fang, Z. Gong, Y. Y. Thomas *et al.*, "Deferred shading graphics pipeline processor having advanced features," Oct. 5 2010, uS Patent 7,808,503.

[20] W. Gongli and W. Li, "Development status and prospective trend of pipeline technology at home and abroad [j]," *Petroleum Planning & Engineering*, vol. 4, p. 000, 2004.

[21] Y. Linde, A. Buzo, and R. Gray, "An algorithm for vector quantizer design," *Communications, IEEE Transactions on*, vol. 28, no. 1, pp. 84–95, 1980.

[22] P. A. Chou, T. Lookabaugh, and R. M. Gray, "Entropy-constrained vector quantization," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 37, no. 1, pp. 31–42, 1989.

[23] T. Lehn-Schiøler, A. Hegde, D. Erdogmus, and J. C. Principe, "Vector quantization using information theoretic concepts," *Natural Computing*, vol. 4, no. 1, pp. 39–51, 2005.

[24] S. Craciun, D. Cheney, K. Gugel, J. C. Sanchez, and J. C. Principe, "Wireless transmission of neural signals using entropy and mutual information compression," *Neural Systems and Rehabilitation Engineering, IEEE Transactions on*, vol. 19, no. 1, pp. 35–44, 2011.