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. A 128-way parallel hardware platform, which could achieve an average speedup of 6553 versus Matlab at 200MHz, is implemented with FPGA.

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 operation in orders of magnitude [13] [14] [15], we consider to implement some algorithms with hardware platform, instead of conventional software methods.

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 [16], 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 [17] [18] 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 128-way parallel hardware platform that is 6553 times faster than Matlab at 200MHz.

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-definite 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 $\Psi$ between input space $X$ and a feature space $F$ (which is an inner product space) such that:

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

If we identify $\Psi(x) = \kappa(x, \cdot)$, 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:

$$\begin{cases} f_0 = 0 \\ e(n) = d(n) - f_{n-1}(x_n) \\ f_n = f_{n-1} + \eta e(n) \Psi(x_n) \end{cases}$$ (2)
where \( e(n) \), \( d(n) \), \( f_n \) are the prediction error, desired signal and learned nonlinear mapping at iteration \( n \) respectively, \( \eta \) is the step size. We can get the access of \( f_{n-1}(x_n) \) through:

\[
f_{n-1}(x_n) = \eta \sum_{j=1}^{n-1} e(j)\kappa(x_j, x_n)
\]

(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.

\section*{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 [19] [20] [21] [22], 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:

\[
dis(x_n, C(n-1)) = \min_{1\leq j\leq \text{size}(C(n-1))} \|x_n - C_j(n-1)\|
\]

(4)

where \( C_j(n-1) \) is the \( j \)th 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 \( \text{dis}(x_n, C(n-1)) \) is greater than the threshold \( \varepsilon_X \), it means that \( x_n \) is not a “redundant” data. Then the codebook and coefficient should be updated as:

\[
C(n) = \{C(n-1), x_n\}
\]

\[
a(n) = \{a(n-1), \eta e(n)\}
\]

(5)

(6)

Oppositely, if \( \text{dis}(x_n, C(n-1)) \) is less than the threshold \( \varepsilon_X \), 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:

\[
a_{j^*}(n) = a_{j^*}(n-1) + \eta e(n)
\]

(7)

\[
j^* = \arg \min_{1\leq j\leq \text{size}(C(n-1))} \|x_n - C_j(n-1)\|
\]

(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:

\[
f_n(x_n) = \sum_{j=1}^\text{size}(C(n-1)) a_j(n-1)\kappa(C_j(n-1), x_n)
\]

(9)

\section*{C. Survival Kernel}

We select survival kernel to implement in our FPGA platform. Assuming \( X = \mathbb{R}_+^m \), where \( \mathbb{R}_+^m = \{x \in \mathbb{R}^m : x = (x_1, ..., x_m), x_i > 0, i = 1, ..., m\} \). The survival kernel is defined by [16]:

\[
\kappa_{\text{sur}}(x,y) = \int_{\mathbb{R}_+^m} (I(x > t)I(y > t))dt
\]

(10)

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

\[
\kappa_{\text{sur}}(x,y) = \prod_{i=1}^m \min(x_i, y_i)
\]

(11)

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

\[
\kappa_{\text{sur}}(x,y) = \min(x, y)
\]

(12)

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

\section*{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.

\section*{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.

\section*{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.

\section*{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 \( \varepsilon_X \). 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 \( 1000 \times 32 \text{bits} \) (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 \( \varepsilon_X \) 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 \( \varepsilon_X \), we only update the coefficient. Otherwise, codebook and coefficient are updated with \( x_n \) and \( \eta e(n) \) respectively.

Fig. 1. Microarchitecture of PE
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 $\eta$ 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 $\text{dis}(x_n, C(n - 1))$:** 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 $\text{dis}(x_n, C(n - 1))$ 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 $\text{dis}(x_n, C(n - 1))$ is only used to compare with threshold $\varepsilon_X$ 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 $\text{dis}(x_n, C(n - 1))$, 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 $\text{dis}(x_n, C(n - 1))$ and $\varepsilon_X^2$. In the light of comparison result, updating of codebook and coefficient are conducted. If $\text{dis}(x_n, C(n - 1))$ 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](image-url)

**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:

$$d(n) = \cos(\pi x_n) + v(n)$$

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. 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](image-url)

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.

<table>
<thead>
<tr>
<th>Iterations</th>
<th>FPGA(ms)</th>
<th>Matlab(ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.400</td>
<td>2681.429</td>
<td>6704</td>
</tr>
<tr>
<td>2048</td>
<td>0.830</td>
<td>5524.324</td>
<td>6656</td>
</tr>
<tr>
<td>4096</td>
<td>1.690</td>
<td>11032.853</td>
<td>6528</td>
</tr>
<tr>
<td>8192</td>
<td>3.410</td>
<td>22007.986</td>
<td>6454</td>
</tr>
<tr>
<td>16384</td>
<td>6.850</td>
<td>43987.562</td>
<td>6422</td>
</tr>
</tbody>
</table>

**TABLE I**

SPEEDUP OF THE PARALLEL HARDWARE VERSUS MATLAB

**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. 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 made with our hardware platform at 200MHz. Furthermore, the hardware platform performs well in real-time at a sample rate of 2.4MHz approximately when the threshold \(\varepsilon_X\) is set as 1/1024 in our experiment.

**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

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

**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 definite 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. At last, we realize a 128-way parallel hardware platform which has an average speedup of 6553 versus Matlab at 200MHz.

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 speed 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