A Reconfigurable Parallel FPGA Accelerator for the Kernel Affine Projection Algorithm

Xiaowei Ren[†‡], Qihang Yu[†], Badong Chen[†], Nanning Zheng[†] and Pengju Ren[†]
[†]Institute of Artificial Intelligence and Robotics, Xi'an Jiaotong University, Xi'an, China, 710049
[‡]State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, China, 214125
Email: {renxiaowei, yqh777cn}@stu.xjtu.edu.cn {chenbd, nnzheng, pengjuren}@mail.xjtu.edu.cn

***Abstract*— Kernel affine projection algorithm (KAPA) is an efficient online kernel learning method, because it not only inherits the advantages of other kernel adaptive filtering (KAF) algorithms, but also reduces the gradient noise significantly. More importantly, it provides a unifying framework for many KAF algorithms. However, suffering from huge computational load, especially when network size is large, it is not suitable for real-time applications. In order to extend its availability, we design a reconfigurable parallel FPGA accelerator for it. The generally used Gaussian kernel is chosen. Moreover, a novel quantization method is adopted to constrain the network size, so as to further reduce computational load and storage overhead. The proposed accelerator allows multiple input data to be processed simultaneously, accelerating the execution rate. Shift registers are used to record the results of different input data. The codebook and coefficients are updated for each input in sequential order along with the shifting of registers constantly. Finally, the FPGA accelerator with eight datapaths, which works at 100MHz, achieves an average speedup of 404.47 versus C code running on a 3GHz Intel(R) Core(TM) i5-2320 CPU.**

I. INTRODUCTION

Kernel tricks make many linear methods yield more powerful nonlinear extensions by recasting them into high dimensional reproducing kernel Hilbert spaces (RKHS), so they gain large popularity in various applications. Among these kernel methods, benefiting from its simplicity and online nature, kernel least mean square (KLMS) [1] is a well-known one. It has been successfully used in the machine learning and signal processing. Moreover, kernel affine projection algorithm (KAPA) [2], which is the combination of famed kernel trick and affine projection algorithms [3], is an improvement of KLMS. KAPA not only inherits the advantages of KLMS, but also reduces gradient noise significantly. What's more, KAPA is a unifying framework for kernel adaptive filtering (KAF). If parameters in KAPA are set to different values, it will reduce to KLMS, sliding-window kernel recursive least square (SW-KRLS) [4], kernel adaline [5] and regularization networks [6] etc. That's to say, we can understand the relationship among different kernel methods and the tradeoff between complexity and performance through KAPA effectively. Therefore, KAPA has better performance and wider applicability than KLMS.

However, both KLMS and KAPA suffer from huge computational load, especially when network size is very large. This critical issue prevents them from using in some applications where real-time processing is needed. So we have to find some solutions to accelerate them. In [7], Ren et al. designed a parallel acceleration platform for KLMS with FPGA. With the adoption of a novel quantization method [8] [9] [10] [11] [12], a simple and effective survival kernel [10] and pipeline technology, it achieves a speedup in orders of magnitude compared with software implementation. Because KAPA is much more general and flexible than KLMS, we extend the previous work to KAPA in this paper. First, we change the survival kernel to Gaussian kernel, which is far more generally used. In order to reduce the hardware cost, we transform the Gaussian kernel into a look-up table which has 2048 data entries, instead of implementing specific functional units for it. Second, we replicate the datapaths multiple times to allow numerous input data in a window to be processed simultaneously. We also add some corresponding shift-registers to record the results of different input. When shifting these registers constantly, the codebook and coefficients can be updated accordingly. Finally, we implement the accelerator with eight datapaths on a Xilinx Virtex-7 FPGA board. When it works at 100MHz, compared with C code running on a 3GHz Intel(R) Core(TM) i5-2320 CPU, it is 404.47 times faster on average.

The organization of this paper is as follows. Section II briefly describes the quantized KAPA (QKAPA), in which a quantization method is used to constrain the network size. The microarchitecture of accelerator is elaborated in section III, followed by the evaluation results in section IV. Finally, this work is concluded in section V.

II. QUANTIZED KAPA

The network size of the original KAPA will increase continuously along with the data training, usually leading to very large computational load and storage overhead. Therefore, a quantization method [8] is adopted in this work, which could reduce the computation load and hardware cost significantly. The detailed description of the quantized KAPA (QKAPA) is shown in Algorithm I. For simplicity, we don't consider the regularization term in this paper.

Let $\{x_n\}$ $n = 1, 2, 3...$ be the input set, $\{d_n\}$ is the corresponding desired output. A window, whose size is K, slides across the input set, so $\{x_k, x_{k+1}, ...x_n\}$ are grouped into the n_{th} window, where $k = max(1, n - K + 1)$. First, we initialize the codebook and coefficients with x_1 and ηd_1

978-1-4799-8058-1/15/$31.00 ©2015 IEEE.

respectively (line1), where $C(n,k)$ and $a(n,k)$ are the codebook and coefficients updated by the x_k in the n_{th} window. For every single element in the n_{th} window, we calculate their output $f(n,K)$ and error $e(n,k)$ as line 7 and 9. Line 8 is the formula for Gaussian kernel. The distance between x_k and codebook $C(n,k-1)$ is used to judge how to update the codebook and coefficients, so the distance is calculated in line 10 beforehand. If $dis(x_k, C(n,k-1))$ is smaller than or equal to the quantization threshold ε_X, x_k is "redundant". Then codebook keep unchanged and only the coefficient is updated as line 14. Otherwise, x_k is not "redundant". A new space is allocated in both codebook and coefficients. The value for this new space is x_k and $\eta e(n,k)$ respectively.

Algorithm 1: Quantized KAPA with Gaussian Kernel

Input: $\{x_n \in X, d_n\}$, $n = 1, 2, ...$

1 **Initialization**: learning step $\eta > 0$, quantization size $\varepsilon_X > 0$, codebook $C(1,1) = x_1$, coefficients $a(1,1) = \eta d_1$

2 **Computation**:

3 **while** $\{x_n, d_n\}$ *is available* **do**

4 **for** *k = max(1, n-K+1) to n* **do**

5 (1) Compute the output:

6
$$f(n,k) = \sum_{j=1}^{size(C(n,k-1))} a_j(n,k-1)\kappa(x_k, C_j(n,k-1))$$

7
$$\kappa(x_k, C_j(n,k-1)) = exp(-\frac{(x_k - C_j(n,k-1))^2}{2\sigma^2})$$

8 where $C(n,k-1)$ and $a(n,k-1)$ are the codebook and coefficients updated by the x_{k-1} in the n_{th} window $\{x_{n-K+1}, ..., x_{k-1}, x_k, x_{k+1}, ...x_n\}$

9 (2) Compute the error: $e(n,k) = d_n - f(n,k)$

10 (3) Compute the distance between x_k and $C(n,k-1)$:

$$dis(x_k, C(n,k-1)) = \min(\|x_k - C_j(n,k-1)\|)$$

where $1 \le j \le size(C(n,k-1)$

11 (4) Update the codebook and coefficients:

12 **if** $dis(x_k, C(n,k-1)) \le \varepsilon_X$ **then**

13 $C(n,k) = C(n,k-1)$

14 $a_{j^*}(n,k) = a_{j^*}(n,k-1) + \eta e(n,k)$

$$j^* = arg \min_{1 \le j \le size(C(n,k-1))} \|x_k - C_j(n,k-1)\|$$

15 **else**

16 $C(n,k) = \{C(n,k-1), x_k\}$

17 $a(n,k) = \{a(n,k-1), \eta e(n,k)\}$

III. HARDWARE IMPLEMENTATION

The FPGA accelerator is a follow-up study of [7], which is designed to accelerate the KLMS algorithm specifically. If the window size is set to 1, our platform will work the same as [7]. As for the detailed microarchitecture, all of these two works adopt a quantization method [8] to constrain the network size. Besides, there are two main differences between this accelerator and [7]. First, the kernel used in [7] is survival kernel [10], while we applied Gaussian kernel in this paper, so two more pipeline stages FFIX and GAUSSIAN are added for KAPA. Second, the codebook and coefficients in [7] are updated based on only one input datum. However, in KAPA, K input elements are grouped together to form a sliding window, codebook and coefficients are updated by all of the elements in this window. Therefore, we replicate the datapath, which is similar to [7], eight times, enabling eight elements in a window to be processed simultaneously. Also, the window size is a user-defined parameter, that can be configured according to specific application requirements.

A. Microarchitecture of the Accelerator

The microarchitecture of accelerator is shown in Fig 1. In this circuitry, the window size could be any value determined by different applications, and control unit could be adjusted accordingly by different window sizes to generate corresponding control signals. We change the survival kernel to Gaussian kernel and transform the Gaussian kernel into a look-up table with 2048 data entries. Therefore, two more pipeline stages FFIX and GAUSSIAN are added.

Functional units, SRAMs and other hardware resources are arranged into eight pipeline stages. Every pipeline stage is named after the operations they conduct. K input data of the n_{th} window are stored in RAM x(n) and RAM d(n). Eight input data and codebook element $C_j(n,k-1)$ are read out in FETCH stage. Meanwhile, eight x_k are loaded into the shift register x(n)_buf. $x_k - C_j(n,k-1)$ and error $e(n,k)$ are calculated in SUB stage. In order to reduce the hardware cost, Gaussian kernel is transformed into a look-up table which has 2048 data entries. The look-up table is accessed by the result of $x_k - C_j(n,k-1)$, so we have to change its format from float point to fixed point in stage FFIX. Stage MULT is responsible for the computation of $a_j(n,k-1)\kappa(x_k, C_j(n,k-1))$, $(x_k - C_j(n,k-1))^2$ and $\eta e(n,k)$. Every product term of output $f(n,k)$ is accumulated in stage COMP&ADD. At the same time, we can get the results of $dis(x_k, C(n,k-1))$ and j^* with comparators in this stage. After the error $\eta e(n,k)$ is calculated for every input x_k, these eight results are recorded in a shift register of stage UPDATE. Then, they are added with the corresponding $a_{j^*}(n,k-1)$ one by one. According to the comparison result between $dis(x_k, C(n,k-1))$ and ε_X, codebook and coefficients are updated in the final stage WB.

B. Data Training Procedure

The detailed data training procedure for KAPA is divided into three steps in coarse granularity. Eight input data are processed in parallel for KAPA.

Step 1. Initialization: At the beginning of data training, we only have one input datum. Therefore, even though we replicate the datapaths eight times, only one need to work in this step. ηd_1 is calculated the same as [7]. Then x_1 and ηd_1 are written into the codebook and coefficients.

Step 2. Evaluation of $f(n,k)$***,*** j^* ***and*** $dis(x_k, C(n,k-1))$***:*** As for a single element of the n_{th} window, operations performed in this step is the same as [7]. In order to develop the parallelism of KAPA adequately, we allow eight input elements in the n_{th} window to be processed in parallel. If the window size K is smaller than eight, K datapaths work in this step. Otherwise, input elements in a single window are processed in the group of eight. In Algorithm I, every

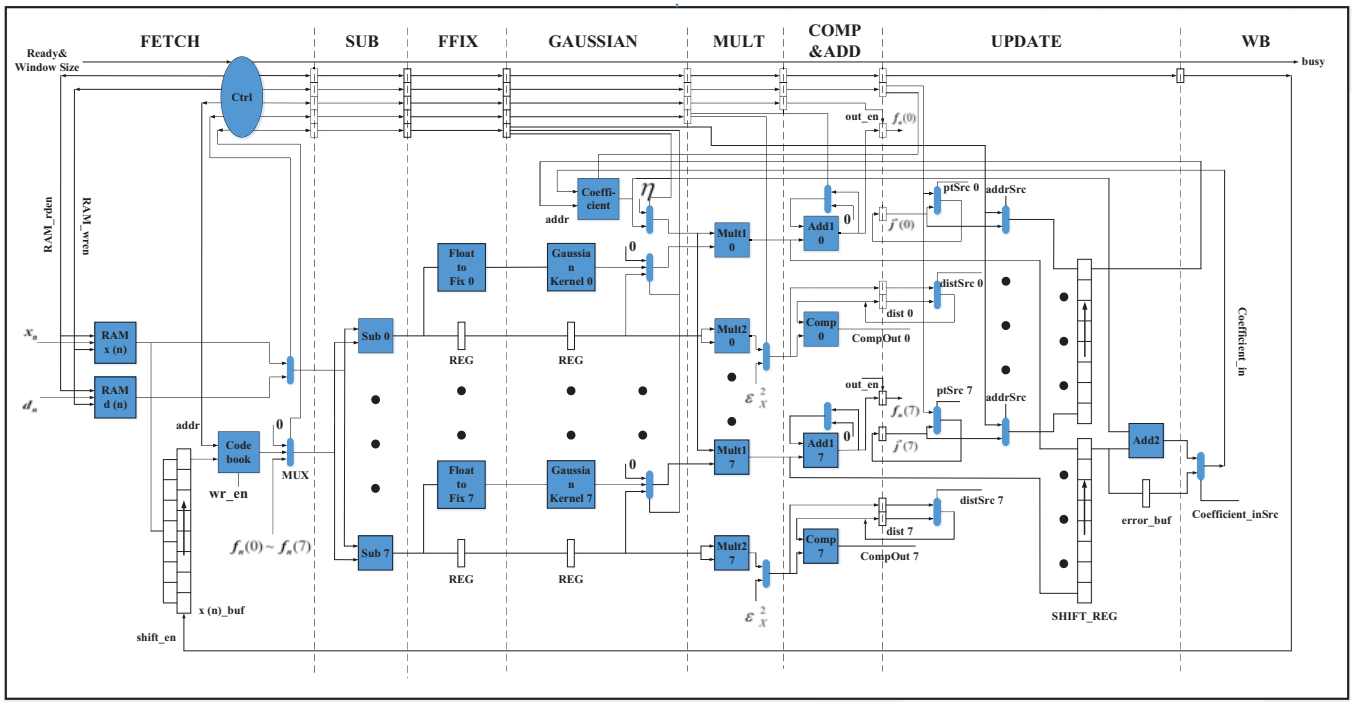


Fig. 1: Microarchitecture of the Accelerator

input x_k uses the codebook and coefficients updated by the previous input x_{k-1}. However, in our FPGA platform, the codebook and coefficients used by x_k in the n_{th} window are updated by the $(n-1)_{th}$ window or the former eight input elements. Evaluation results in section IV.A shows that this minor difference will not lead to intolerable prediction error. For various window sizes, prediction error could be converged to zero quickly.

Step 3. Updating: Based on the comparison results between $dis(x_k, C(n,k-1))$ and ε_X, we can determine whether the j^* or a new data entry should be written in coefficients. If a new data entry is allocated in coefficients, so does the codebook. Then the reasonable writing addresses for codebook and coefficients are loaded into a shift register. Similarly, the results of $\eta e(n,k)$ for every input are also stored in a shift register. These shift registers are shifted time after time, either $\eta e(n,k)$ or $a_{j^*}(n,k-1)+\eta e(n,k)$ is selected out to update the coefficients. Similarly, codebook is also updated accordingly with the shifting of x(n)_buf.

C. Extensions

This FPGA accelerator also yields great scalability. Even though we demonstrate an eight-datapath implementation on FPGA board, it is worth noting that the number of datapath is not limited to eight. It can be extended to more (e.g. 16, 32), which is constrained by hardware resource and clock frequency. With the increasing of datapaths, current updating mechanism may be a bottleneck, because we have to wait for a large number of cycles to update the codebook and coefficients sequentially for every input datum. However, another updating mechanism could conquer this bottleneck easily. The more input data are processed in parallel, the more of them need to update the same address in codebook and coefficients. We can find out these input data, sum their $\eta e(n,k)$ up and finish the updating simultaneously, instead of sequential operations. In this way, the updating mechanism will not impact the performance of FPGA accelerator any more.

IV. EVALUATION

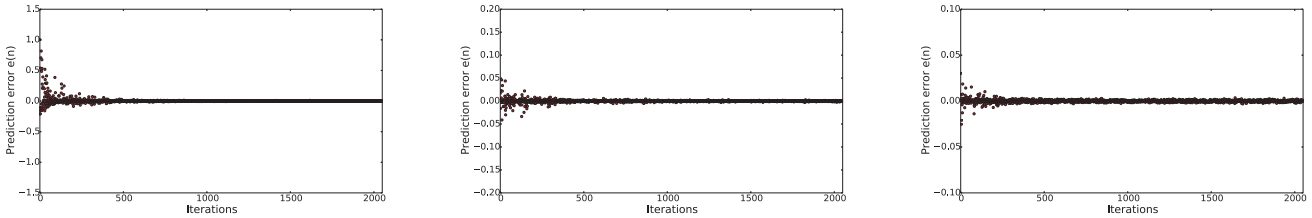
We implement our KAPA accelerator with Xilinx Virtex-7 XC7V2000T, state-of-the-art FPGA device. We test its prediction error and speedup with the following function:

$$d_n = \frac{sin(x_n)}{x_n} + v(n) \tag{1}$$

Where x_n is uniformly distributed over $[-2\pi, 2\pi]$, $v(n)$ is a zero-mean white Gaussian noise with variance 0.001. In the experiment, η and ε_X^2 are set to 0.25 and $1/64$ respectively. Finally, we analyse its hardware cost. All of the evaluation results show that we have implemented an accurate and effective accelerator for KAPA at low hardware cost.

A. Prediction Error of the Accelerator

Two main causes will impact the prediction error of this FPGA accelerator. Firstly, as we said in section III.B step 2, the updating of codebook and coefficients is a little bit different from Algorithm I. Second, quantized Gaussian kernel could impact the accuracy of FPGA accelerator as well. Therefore, we test its accuracy with formula (1) firstly. Results in Fig. 2 show that, for different window sizes, prediction error could



(a) window size = 1 (b) window size = 10 (c) window size = 50

Fig. 2: The change of prediction error for different window sizes

be converged to zero in a very short period of time. So we can conclude that our KAPA accelerator is accurate.

B. Speedup versus C Code

TABLE I: Speedup versus C Code

Window Size	# of Iterations	FPGA (ms)	C (s)	Speedup
1	2048	1.730	0.086	49.71
	4096	3.606	0.345	95.67
	8192	7.372	1.281	173.76
10	2048	3.888	0.808	207.82
	4096	8.066	3.234	400.94
	8192	16.462	13.064	793.58
50	2048	14.312	4.075	284.73
	4096	29.698	16.496	555.46
	8192	60.622	65.386	1078.58

This FPGA platform is designed to accelerate the execution of KAPA so that it could be applied in real-time situations. Therefore, we test its speedup versus C code in this subsection. In the experiment, C code runs on a PC which is configured with a 3GHz Intel(R) Core(TM) i5-2320 CPU and $4GB$ main memory, our FPGA accelerator works at 100MHz. From the results in Table I, we can conclude that our FPGA accelerator is 404.47 times faster on average. What's more, the larger the window size, the larger the speedup we can get. So the execution rate of KAPA is accelerated considerably.

C. Implementation Result

TABLE II: Implementation Result of FPGA Accelerator

Resources	Utilized	Available	Utilization Rate(%)
FF	4131	2443200	0.169
LUT	8950	1221600	0.733
BRAM	26	1292	2.012
DSP48	66	2160	3.056
BUFG	2	128	1.562
MMCM	1	24	4.167

Results shown in Table II demonstrate the cost-efficiency of our FPGA accelerator. Especially the Flip-Flop (FF) and Look-up Table (LUT), two main resources in FPGA, we only consume 0.169% and 0.733% of them respectively. Sometimes, we need to group several instances of Algorithm I together in physical applications. It means that several processing elements shown in Fig 1 should operate simultaneously. However, benefiting from the low cost of our accelerator, we can implement a large number of processing elements on the FPGA board easily. Thar's to say, not only the parallelism in Algorithm I could be developed in our design, but also the parallelism between different instances of Algorithm I. In this case, BRAM and DSP48 will limit the number of processing element we can implement. However, the DSP unit can be implemented with abundant FF and LUT, then DSP48 will not be the bottleneck. Therefore, constrained by the BRAM, we can replicate the processing element 50 times approximately on FPGA board.

V. CONCLUSION

In this paper, we extend [7] to the KAPA accelerator. It could be reconfigured by different window sizes. With the replication of datapaths eight times, we allow eight elements in a single window to be processed in parallel. We also adopt a quantization method to constrain the network size, reducing storage cost and computation load. Gaussian kernel is transformed into a Look-up Table to lower the hardware cost further more. Finally, the FPGA accelerator, which works at 100MHz, is 404.47 time faster on average than the C code runs on a PC configured with 3GHZ Intel(R) Core(TM) i5-2320 CPU and $4GB$ main memory. What's more, the design of this work yields great scalability. It can be extended easily in terms of the number of datapath and processing element.

ACKNOWLEDGMENT

This work was supported by 973 program (No.2015CB351703), NSFC grants (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) and the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing.

REFERENCES

[1] 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.

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

[3] A. H. Sayed, *Fundamentals of adaptive filtering*. John Wiley & Sons, 2003.

[4] S. Van Vaerenbergh, J. Via, and I. Santamaría, "A sliding-window kernel rls algorithm and its application to nonlinear channel identification," in *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, vol. 5. IEEE, 2006, pp. V–V.

[5] T.-T. Frieß and R. F. Harrison, "A kernel based adaline." in *ESANN*, 1999, pp. 245–250.

[6] T. Evgeniou, M. Pontil, and T. Poggio, "Regularization networks and support vector machines," *Advances in computational mathematics*, vol. 13, no. 1, pp. 1–50, 2000.
[7] X. Ren, P. Ren, B. Chen, T. Min, and N. Zheng, "Hardware implementation of klms algorithm using fpga," in *Neural Networks (IJCNN), 2014 International Joint Conference on*. IEEE, 2014, pp. 2276–2281.
[8] 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.
[9] B. Chen, S. Zhao, S. Seth, and J. C. Principe, "Online efficient learning with quantized klms and l 1 regularization," in *Neural Networks (IJCNN), The 2012 International Joint Conference on*. IEEE, 2012, pp. 1–6.
[10] 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.
[11] S. Zhao, B. Chen, P. Zhu, and J. C. Príncipe, "Fixed budget quantized kernel least-mean-square algorithm," *Signal Processing*, vol. 93, no. 9, pp. 2759–2770, 2013.
[12] B. Chen, S. Zhao, P. Zhu, and J. C. Principe, "Quantized kernel recursive least squares algorithm," *Neural Networks and Learning Systems, IEEE Transactions on*, vol. 24, no. 9, pp. 1484–1491, 2013.