A Fully Pipelined Kernel Normalised Least Mean Squares Processor For Accelerated Parameter Optimisation

Nicholas J. Fraser, Duncan J.M. Moss, JunKyu Lee, Stephen Tridgell, Craig T. Jin and Philip H.W. Leong
School of Electrical and Information Engineering, Building J03
The University Of Sydney, 2006, Australia

***Abstract*—Kernel adaptive filters (KAFs) are online machine learning algorithms which are amenable to highly efficient streaming implementations. They require only a single pass through the data during training and can act as universal approximators, i.e. approximate any continuous function with arbitrary accuracy. KAFs are members of a family of kernel methods which apply an implicit nonlinear mapping of input data to a high dimensional feature space, permitting learning algorithms to be expressed entirely as inner products. Such an approach avoids explicit projection into the feature space, enabling computational efficiency. In this paper, we propose the first fully pipelined floating point implementation of the kernel normalised least mean squares algorithm for regression. Independent training tasks necessary for parameter optimisation fill L cycles of latency ensuring the pipeline does not stall. Together with other optimisations to reduce resource utilisation and latency, our core achieves 160 GFLOPS on a Virtex 7 XC7VX485T FPGA, and the PCI-based system implementation is 70$\times$ faster than an optimised software implementation on a desktop processor.**

I. Introduction

Machine learning and data mining focus on the development of mathematical ideas and algorithms to learn from data. Interest in these fields has been steadily increasing in recent years as advancements have addressed previously intractable problems such as speech recognition, handwriting recognition, image processing, credit card fraud and automatic fault detection. One important class of machine learning algorithms are kernel methods which include support vector machines (SVMs) [1], Gaussian processes (GPs) [1], and kernel adaptive filters (KAFs) [2].

Reconfigurable computing, the application of field programmable gate arrays (FPGAs) to computing problems, has been successfully applied in accelerating certain classes of problems. The following computational conditions are desirable for efficient FPGA implementation: (1) static data structures requiring small amounts of memory, (2) instruction and task level parallelism, (3) high ratio of computation to memory accesses (arithmetic intensity) (4) modest precision requirements, and (5) low input/output bandwidth.

SVMs and GPs are batch-mode algorithms which require multiple passes over the training set and do not satisfy conditions (1)-(3) because storage of the entire training set is required; because the result of one iteration is required before the next iteration can proceed; and because many memory accesses are required per data input and processing time increases with data size. In contrast, KAFs are recursive algorithms which perform a small, fixed amount of computation per data input, and meet all the above conditions, making them amenable to efficient FPGA implementations.

Different KAF algorithms have been proposed for classification, regression and anomaly detection tasks [2]. In this paper, we describe a particularly efficient implementation of the kernel normalised least mean squares (KNLMS) algorithm. KNLMS was chosen because of its simple computational structure and its ability to approximate any continuous function with arbitrary accuracy [2]. The computational bottleneck for KNLMS is the evaluation of an inner product in the feature space. Our implementation is fully pipelined and uses single-precision floating point which leads to high latency which is normally undesirable. However, in machine learning, a parameter search is usually performed over a grid, and if B different values of each of P parameters are explored, the search space is B^P. In our implementation, we exploit this property to evaluate L independent parameter settings in parallel, neatly filling the KNLMS pipeline. As a result, our implementation achieves very high computational efficiency. Although acceleration could also be achieved using a GPU, FPGA implementations are advantageous in large scale systems where power consumption is an issue, such as data centres. Also, FPGA implementations are preferred in embedded, real-time multichannel applications, such as machine prognostics, where the proposed design could also be applied.

The key contributions of this work are:

- The first description of fully pipelined datapaths for KAFs. Compared with previous vector-processor architectures, much higher performance can be attained because all pipeline stages do useful work and never stall. This is achieved by creating a deeply pipelined module of a significant, non-recursive portion of the KNLMS algorithm. A scheduler is then employed to ensure that no data dependencies exist within the pipeline, for L parallel problems.
- A number of optimisations for the KNLMS algorithm: pipelining, memory optimisations, scheduling and mixed-precision processing are combined to achieve a 575$\times$ speedup over a naïve implementation for parameter optimisation.
- A complete PCI-based system implementation with a speedup of 70$\times$ over a processor and a speedup of 660$\times$ over a previous microcoded kernel recursive

least squares implementation, by Pang et al. [3].

This paper is organised as follows: Section II describes the KNLMS algorithm [4] and summarises previous implementations of kernel based machine learning algorithms; Section III describes the proposed architecture; Section IV shows the performance and accuracy results of the proposed architecture compared with a CPU implementation; and conclusions are drawn in Section V.

II. Background

A. Kernel Normalised Least Mean Squares

In this section, the KNLMS algorithm [4] is summarised with particular attention to aspects which affect a hardware implementation.

In a standard supervised learning problem training examples are input/output pairs $\{\mathbf{x}_i, y_i\}$, where $\mathbf{x}_i \in \mathbb{R}^M$ is the input vector and $y_i \in \mathbb{R}$ is the output or target. In regression, the goal is to estimate a function, $f(\mathbf{x}_i)$, which maps $\mathbf{x}_i \to y_i$. Kernel regression attempts to estimate this function by learning a dictionary $\mathcal{D}$, containing a subset of input vectors, and corresponding weights, $\boldsymbol{\alpha}$. A prediction, $\tilde{y}_i$, is then calculated as follows: $\tilde{y}_i = \sum_{n=1}^{N} \alpha_n \kappa(\mathbf{x}_i, \tilde{\mathbf{x}}_n)$, where $\tilde{\mathbf{x}}_n$ is the n^{th} entry in $\mathcal{D}$, α_n is the n^{th} entry of $\boldsymbol{\alpha}$, N is the maximum size of $\mathcal{D}$, and $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ is the kernel function, specified at design time. Although different kernels can be accommodated, in this paper we focus on the commonly used radial basis function (RBF) kernel, $\kappa(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|^2}$, where γ is a free parameter chosen to suit the problem at hand.

The KNLMS algorithm is a stochastic gradient descent based algorithm which learns its model by taking small steps in the direction of the instantaneous gradient, to minimise the error in the current training example. Similar to algorithms such as the least mean squares (LMS) algorithm [5], it slowly converges to a solution over time.

The coherence criterion [4] is used to select the entries in the dictionary. For unit norm kernel functions, the coherence criterion is defined as follows: given a new input example at iteration t, $\mathbf{x}_t$ is added to the dictionary if $\max(|\mathbf{k}_t|) \le \mu_0$, where $\mathbf{k}_t$ is the kernel vector with the n^{th} element being given by $\kappa(\mathbf{x}_t, \tilde{\mathbf{x}}_n)$, μ_0 is the coherence parameter chosen at design time. The weights for each iteration are then calculated by solving the instantaneous approximation to the following affine projection problem:

$$\min_{\boldsymbol{\alpha}} \|\boldsymbol{\alpha} - \hat{\boldsymbol{\alpha}}_{t-1}\|^2 \quad \text{subject to} \quad y_t = \mathbf{k}_t^{\dagger}\boldsymbol{\alpha} \quad , \tag{1}$$

where $\hat{\boldsymbol{\alpha}}_{t-1}$ is the set of weights obtained from the previous iteration and † denotes the vector transpose operation. Assuming that the current input vector, $\mathbf{x}_t$, can be adequately represented by the current dictionary and is not added to the dictionary, Eq. (1) can be solved by minimising the following Lagrangian function:

$$J(\boldsymbol{\alpha}, \lambda) = \|\boldsymbol{\alpha} - \hat{\boldsymbol{\alpha}}_{t-1}\|^2 + \lambda(y_t - \mathbf{k}_t^{\dagger}\boldsymbol{\alpha}) \quad . \tag{2}$$

A solution, $\hat{\boldsymbol{\alpha}}_t$, is found by differentiating Eq. (2) with respect to $\boldsymbol{\alpha}$ and λ and setting the derivatives to zero, giving:

$$\begin{aligned} 2(\hat{\boldsymbol{\alpha}}_t - \hat{\boldsymbol{\alpha}}_{t-1}) &= \mathbf{k}_t\lambda \\ y_t &= \mathbf{k}_t^{\dagger}\hat{\boldsymbol{\alpha}}_t \ . \end{aligned} \tag{3}$$

```
Initialise the step-size, $\eta$, and the regularization factor, $\epsilon$.
Insert $\mathbf{x}_1$ into the dictionary, denote it as $\tilde{\mathbf{x}}_1$. $\mathbf{k}_1 = \kappa(\mathbf{x}_1, \tilde{\mathbf{x}}_1)$,
$\hat{\boldsymbol{\alpha}}_1$ = 0, n = 1.
while $t > 1$ do
  Get $\{\mathbf{x}_t, y_t\}$.
  Calculate $\mathbf{k}_t = [\kappa(\mathbf{x}_t, \tilde{\mathbf{x}}_1), \cdots, \kappa(\mathbf{x}_t, \tilde{\mathbf{x}}_n)]^{\dagger}$.
  if $\max(|\mathbf{k}_t|) > \mu_0$ then
    Calculate $\hat{\boldsymbol{\alpha}}_t$ using Eq. (4).
  else
    $n = n + 1$.
    Append $\kappa(\mathbf{x}_t, \mathbf{x}_t)$ to $\mathbf{k}_t$.
    Insert $\mathbf{x}_t$ into the dictionary, denote it as $\tilde{\mathbf{x}}_n$.
    Calculate $\hat{\boldsymbol{\alpha}}_t$ using Eq. (5).
  end if
end while
```

Fig. 1. KNLMS algorithm with coherence criterion.

Multiplying each term in the first equation by $\mathbf{k}_t^{\dagger}$ and substituting in for y_t, we get $\lambda = 2(\mathbf{k}_t^{\dagger}\mathbf{k}_t)^{-1}(y_t - \mathbf{k}_t^{\dagger}\hat{\boldsymbol{\alpha}}_{t-1})$. This yields the following recursive update equation:

$$\hat{\boldsymbol{\alpha}}_t = \hat{\boldsymbol{\alpha}}_{t-1} + \frac{\eta}{\epsilon + \mathbf{k}_t^{\dagger}\mathbf{k}_t}(y_t - \mathbf{k}_t^{\dagger}\hat{\boldsymbol{\alpha}}_{t-1})\mathbf{k}_t \tag{4}$$

where η is a step-size parameter and ϵ is a regularisation factor.

For the case where the current training example cannot be adequately represented by the dictionary, the current input, $\mathbf{x}_t$, is appended to the dictionary and the update equation becomes:

$$\hat{\boldsymbol{\alpha}}_t = \begin{bmatrix} \hat{\boldsymbol{\alpha}}_{t-1} \\ 0 \end{bmatrix} + \frac{\eta}{\epsilon + \mathbf{k}_t^{\dagger}\mathbf{k}_t}(y_t - \mathbf{k}_t^{\dagger}\begin{bmatrix} \hat{\boldsymbol{\alpha}}_{t-1} \\ 0 \end{bmatrix})\mathbf{k}_t \tag{5}$$

Pseudocode for the KNLMS algorithm, adapted from [4] and [6], is shown in Figure 1.

B. Literature Review

Kernel methods are eminently amenable to efficient hardware implementations and several implementations of SVM have been reported. Anguita et al. [7] described an SVM core generator which allowed for different speed, resource and accuracy tradeoffs and utilised fixed point arithmetic. Papadonikolakis and Bouganis [8] described a scalable SVM module generator which supported different kernel types. Their design supported different numbers of parallel computing tiles which allowed for performance/resource tradeoffs, and was partitioned into fixed point and floating point sections to achieve high performance while maintaining high accuracy. The MAPLE architecture as described by Majumdar et al. [9] was designed to improve many learning algorithms, including SVM. MAPLE utilised two-dimensional vector processing elements to accelerate linear algebra routines. The architecture also supported off-chip memory, allowing it to accommodate large learning problems. While much of the computation is similar to KNLMS, SVM implementations are batch-mode and hence lower speed than online algorithms.

Pang et al. [3] proposed a compact and low latency microcoded soft vector processor. Parallelism was achieved using up to 128 floating-point vector processing elements, and kernel evaluations were accelerated through the inclusion of a hardware exponentiation unit. An implementation of the

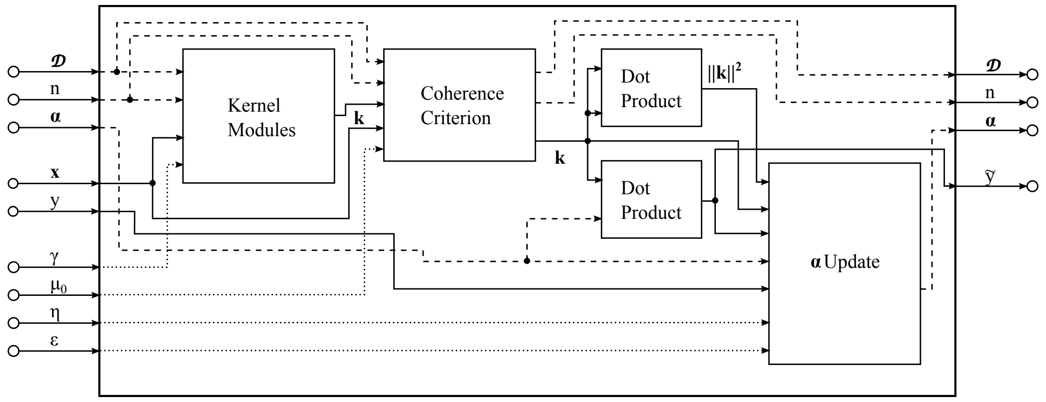

Fig. 2. A high level diagram of the KNLMS processor showing the various submodules.

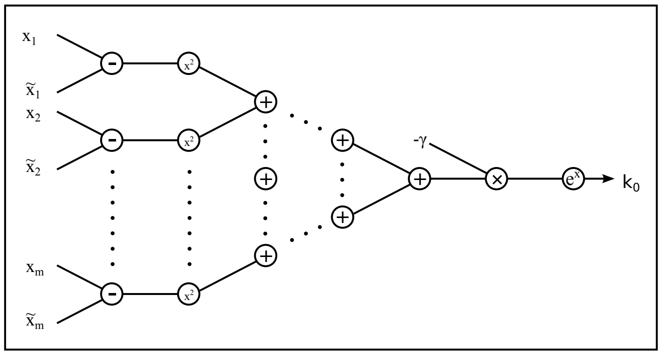

Fig. 3. Dataflow diagram of a kernel module.

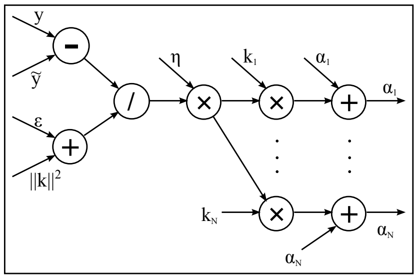

Fig. 4. Dataflow diagram of the $\boldsymbol{\alpha}$ update module.

sliding window kernel recursive least squares (SW-KRLS) algorithm [10] was used as an example in the paper. The floating-point implementation of the quantized kernel least mean squares (QKLMS) algorithm [11] utilising the survival kernel [12] by Ren et al. [13] is most comparable to this work. Differences include: (1) their work was limited to a single dimensional kernel vs arbitrary dimensions, (2) they employed the survival kernel compared to the much more commonly used Gaussian kernel in this design, (3) they achieve parallelism through pipelining and 128 parallel processing elements whereas our efficiency is by virtue of a fully pipelined design.

III. Architecture

In this section, our fully pipelined KNLMS architecture is described, highlighting areas suitable for optimisation. In addition, scalability of the design is explored.

A. High Level Description

The idea behind the design is to create a module to accelerate the *forward path* of the KNLMS algorithm. The forward path consists of the mathematical operations required to update the kernel regression model from time step $t-1$ to t, which are those within the while loop of Figure 1. A high level diagram showing the basic structure of the processor is shown in Figure 2. The submodules are responsible for the following functions: (1) the kernel modules calculate the kernel vector, $\mathbf{k}_t$; (2) the coherence criterion module decides whether to add the latest input example to the dictionary, and updates it if necessary; (3) the dot product modules, which produce the a-priori estimate of y_t, denoted by $\tilde{y}_t$, and the normalisation term, $||\mathbf{k}_t||^2$; and (4) the $\boldsymbol{\alpha}$ update module produces the updated weights, $\boldsymbol{\alpha}_t$. In order to compute multiple iterations of the KNLMS algorithm, the forward path module needs to be connected to a scheduler, which is described in Section III-E.

B. Kernel Module

Figure 3 shows the dataflow graph of a kernel module. The kernel module computes the Gaussian kernel, given by: $\kappa(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma||\mathbf{x}_i - \mathbf{x}_j||^2}$. Each kernel module requires $2M-1$ adders, $M+1$ multipliers and 1 exponential unit, where M is the feature length. N kernel modules are required for a design supporting a maximum dictionary size of N. The most computationally expensive part of the KNLMS processor is the calculation of the kernel vector.

C. Alpha Update Module

The $\boldsymbol{\alpha}$ update module finishes the training step by calculating $\boldsymbol{\alpha}_t$ as shown in Eq. (4). The dataflow graph for the $\boldsymbol{\alpha}$ update module is shown in Figure 4. The $\boldsymbol{\alpha}$ update module first calculates the prediction error and the normalisation term. This is followed by a scalar vector product and an elementwise vector addition. The $\boldsymbol{\alpha}$ update module operates on vectors of length N and as such, requires $N+1$ multipliers, $N+2$ adders and 1 divider.

D. Coherence and Dot Product Modules

The coherence module is a simple control module. It takes $\mathbf{k}_t$, $\mathbf{x}_t$, μ_0, n and $\mathcal{D}$ as inputs. $\mathbf{k}_t$ is padded with zeros for each unused entry in $\mathcal{D}$. If $\max(|\mathbf{k}_t|) \leq \mu_0$, then n becomes $n+1$ and $\mathbf{x}_t$ is appended to $\mathcal{D}$. Otherwise, n and $\mathcal{D}$ are unchanged.

The two dot product modules are made using parallel multipliers followed by an adder tree. Each module operates on vectors of length N and as such, require N multipliers and $N-1$ adders.

E. Optimisations

In order to maximise performance, the optimisations described in this subsection were implemented.

A fully pipelined design cannot be directly synthesised from the algorithm in Figure 1, due to the dependency of the updated dictionary $\mathcal{D}$ and weights $\hat{\boldsymbol{\alpha}}_t$ on the new kernel vector $\mathbf{k}_t$. By describing it in a non-recursive manner, we can turn the datapath into an acyclic one. The feedback connection is then made by externally connecting outputs to corresponding inputs in Figure 2. This results in a fully-pipelined design with initiation interval of 1. Dictionary and weight updates are delayed by the total pipeline latency, L.

The main bottleneck in developing machine learning models is parameter optimisation. Even if the kernel function is fixed to be the Gaussian kernel, a search is required for the following parameters: μ_0, γ, η, and ϵ. This involves performing regression over a test data set using different parameter settings. Since these are independent problems, they can be executed in parallel as batches of L independent tasks. Each task is executed in a different pipeline slot so all hardware units in the KNLMS forward path evaluation pipeline can be fully utilised.

In order to perform regression on L independent problems, we require storage for $L\times$ dictionaries of size MN, and $L\times$ length-M weight vectors. This is achieved by indexing them with a counter $l \in [0, \ldots, L)$ so that every L cycles, we return to the same dictionary and weight vector. This arrangement removes the need for an $L:1$ multiplexer per dictionary and weight entry.

Computing the kernel function $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ for the Gaussian kernel requires an M-input floating-point adder tree which has a total latency of $\lceil \log_2 M \rceil$ times the latency of a single floating-point adder. We observe that: (1) the inputs to this adder are strictly positive, so unsigned arithmetic can be used; (2) the output is passed through a function $e^{-\gamma \sum x^2}$ which is not sensitive to small input errors; and (3) computation can be done in fixed point. This can reduce latency and allow accuracy-speed tradeoffs.

F. Estimated Growth and Latency

We estimated the scalability of the architecture with the key parameters N and M. The number of required operators and estimated latency is shown in Table I. The operator latency is given in parentheses next to the operator symbol. In order to estimate the latency for a given design, the operator latency is multiplied by the expression in the latency row. In terms of worst case scalability, the area of arithmetic operators is $\mathcal{O}(MN)$, memory usage is $\mathcal{O}(MN)$, and latency $\mathcal{O}(\log_2 N + \log_2 M)$.

TABLE I. FORMULAE FOR THE NUMBER OF FLOATING-POINT OPERATORS REQUIRED AND LATENCY IN CYCLES

	+ (11)	× (7)	/ (30)	exp (20)	< (4)
Operation	$2MN + 2N$	$MN+$ $4N+1$	1	N	$N-1$
Latency	$\log_2 N+$ $\log_2 M + 3$	5	1	1	$\log_2 N$

TABLE II. SUMMARY OF PLACE AND ROUTE OUTPUT FOR EACH OF THE KNLMS DESIGNS

	Naïve	Float	Fused
BRAM18K (2060)	8 (0.4%)	145 (7.0%)	145 (7.0%)
DSP48 (2800)	12 (0.4%)	1267 (45.3%)	787 (28.1%)
LUTs (304K)	4550 (14.9%)	150,494 (49.5%)	174,857 (57.5%)
Latency (cycles)	756	207	167
II (cycles)	757	1	1
$F_{max}(MHz)$	96.7	314	289
GOPS	0.07	161.1	148.3

TABLE III. AREA UTILISATION OF DIFFERENT DESIGNS OBTAINED FROM SYNTHESIS

Type	M	N	LUTs	DSPs	L (Estimate)	F_{max}
Float	2	16	77K	595	185 (189)	385
	4	16	109K	819	196 (200)	385
	16	16	307K	2163	218 (222)	385
	8	2	23K	161	162 (166)	385
	8	4	46K	319	177 (181)	385
	8	8	95K	635	192 (196)	385
	8	16	173K	1267	207 (211)	385
Fused	2	16	102K	494	161	303
	4	16	119K	595	163	303
	16	16	440K	1171	175	303
	8	2	33K	101	131	303
	8	4	64K	199	143	303
	8	8	130K	395	155	303
	8	16	247K	787	167	303

IV. RESULTS

This section describes the resource utilisation, performance and accuracy of the implementation written in C. The design was synthesised and implemented using Xilinx Vivado HLS 2014.4. The target platform was a Xilinx VC707 evaluation board using a Xilinx Virtex 7 XC7VX485TFFG1761-2 FPGA.

A. Simulation Results

Table II demonstrates the difference between three different designs: (1) Naïve - an unoptimised C implementation derived from KAFBOX [14], representing a design without consideration of the resulting hardware datapath; (2) Float - a single precision floating point design which uses the Xilinx floating point cores throughout and contains all optimisations described in Section III except for the fixed point adder tree; and (3) Fused - all optimisations including the fixed point adder tree.

Floating-point operations are single precision and IEEE-754 compliant with the exception that denormalised numbers are not supported. Although our design is parameterised, results for the settings $N = 16$ and $M = 8$, are reported unless stated otherwise. The GOPS are estimated using F_{max}, the initiation interval (II) and the number of operations required for single update. With reference to Table I, the number of operations is 513. Note that for the Naïve and Float designs, GOPS is equivalent to GFLOPS.

Table III describes the relationship between the design parameters M and N, and the latency and hardware resources. The numbers in brackets refer to latency estimates, which are

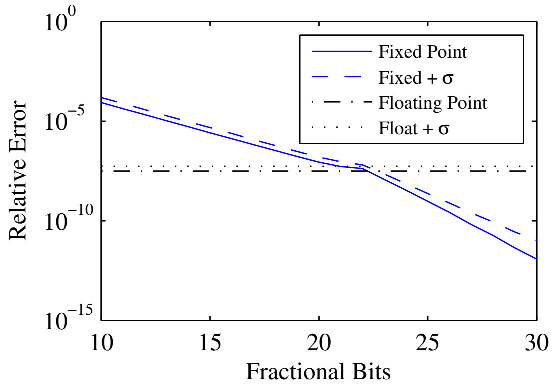

Fig. 5. Relative error introduced by using a fixed point adder tree.

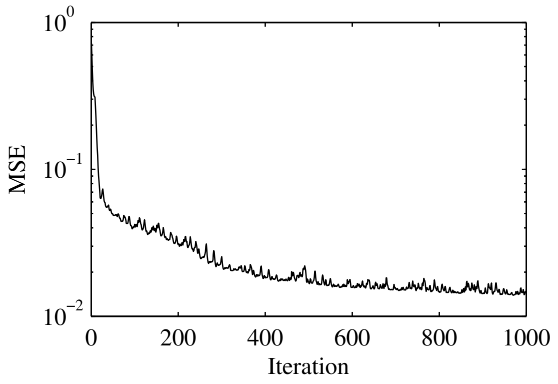

Fig. 6. Convergence of KNLMS using the fixed point adder tree.

calculated using the equations in Table I. These numbers are different to Table II because they are post-synthesis estimates rather than place and route results. The latency in the design increases in proportion to $\log_2$ of the dictionary or feature length and is accurately predicted by the model of Table I. While the area model in Table I accurately reflects the usage of the total number of different operators, estimating individual LUT and DSP usage is not straightforward so the simple model can only be applied to total usage.

From Table III, it can be concluded that both latency and DSP48 usage are improved in the Fused design, but for the N and M values evaluated, the differences are not large. Fused requires more LUT resources because of two data conversions (to and from fixed point) which are not implemented efficiently in Vivado HLS.

B. Learning Accuracy

For the experiments in this paper, the chaotic MG-30 Mackey-Glass benchmark modelling the differential equation $dx(t)/dt = ax(t) + bx(t\tau)/(1 + x(t\tau)^{10})$ with $(a = 0.1, b = 0.2, \tau = 30)$, as implemented in KAFBOX [14] was used.

To isolate and test the fixed point adder tree, uniformly random input vectors were generated in the range of the MG-30 data, i.e. $[0, 2)$. The relative error compared with double precision (i.e. $(y - \tilde{y})/y$) over 10,000 trials was then measured. Figure 5 shows the mean and standard deviation (σ) for 8-input Float and Fused adder trees. The x-axis refers to the number of fractional bits of the Fused implementation, and sufficient integer bits are included to avoid overflow. The relative error and standard deviation of single precision floating point is also shown for comparison. As expected, at 24 bits, the mean error of both implementations are similar.

In terms of learning accuracy, Figure 6 shows the convergence of the learning accuracy of the KNLMS using Fused arithmetic, over 1100 examples of the MG-30 series, generated using a feature length of 8. The first 1000 examples were used as a training set and subsequent 100 examples as a test set. After each training example was learned, the mean squared error (MSE) of the predicting the test set was calculated. The KNLMS algorithm was configured with the following parameters: $\gamma = 0.5$, $\eta = 0.1$, $\epsilon = 0.001$ and $\mu_0 = 0.5$. Using a fractional length of 16 for the fixed point adder resulted in a maximum difference in MSE of less than 1×10^{-5} between single precision floating point and fused arithmetic. Since this was around 0.1% of the learning MSE in Figure 6, the Fused arithmetic design used 16 fractional bits for its fixed point configuration.

C. Performance

An evaluation of the performance of this work compared with other KAF implementations is challenging since designs are not directly comparable. SW-KRLS [10] requires a matrix inversion per update and has $\mathcal{O}(N^2 + NM)$ time complexity. This is in contrast to KNLMS which is $\mathcal{O}(NM)$ stochastic gradient descent techniques [2]. Moreover, one should be careful in comparing Altera and Xilinx LUTs and DSP blocks as they are different. Nevertheless, a summary of previous online KAF implementations of which we are aware is presented in Table IV. Clearly, the different versions of our KNLMS processor have much higher throughput when compared to the other implementations. The KAFBOX implementation is in MATLAB and hence inefficient since the vector length N is insufficiently large for performance improvements through vectorisation. The CPU (C) is a C version of the KNLMS algorithm which uses the multithreaded Intel MKL for linear algebra. However, while the linear algebra library utilised multiple threads, the parameters were searched sequentially for both CPU implementations. The system implementation is described in detail in the following sub-section.

D. System Implementation

The Float KNLMS core was integrated with a RIFFA 2.2.0 [15] PCI Express (PCIe) interface as illustrated in Figure 7. Data ingress and egress are controlled by 512-word FIFOs, and a P-word memory (P must be a multiple of L) for each of the 4 parameters shown in the bottom left module of Figure 2 was used to store the parameters to be searched. Separate memories indexed by l (as detailed in Section III-E) are used to store dictionary and weight values.

When the input FIFO becomes non-empty, the finite state machine (FSM) will read it and convert it to an input vector of the M most recent samples via the serial to parallel converter. Then, a sequence of L independent optimisations with different parameter values is streamed through the KNLMS processor with the results being saved on the output FIFO. The FSM

TABLE IV. COMPARISON OF ONLINE KERNEL METHOD IMPLEMENTATIONS.

Implementation	Algorithm	Device	M	N	DSPs	LUTs	BRAM	Freq MHz	Time ns	Slowdown rel. to Float
Naïve	KNLMS	Xilinx XC7VX485-2	8	16	12	4550	8	96.7	7,829	2,462
Float	KNLMS	Xilinx XC7VX485-2	8	16	1267	150,494	145	314	3.18	1
Fused	KNLMS	Xilinx XC7VX485-2	8	16	787	174,857	145	289	3.46	1.1
System	KNLMS	VC707 dev board	8	16	691	212,665	146	250	13.6	4.3
CPU (C)	KNLMS	Intel i7-4790	8	16	-	-	-	3,600	940	296
CPU (KAFBOX) [14]	KNLMS	Intel i7-4790	8	16	-	-	-	3,600	73,655	23,162
Pang et al. [3]	SW-KRLS	Altera Stratix V 5SGXEA7C2	7	16	30	41,476	227	237	9,000	2,830

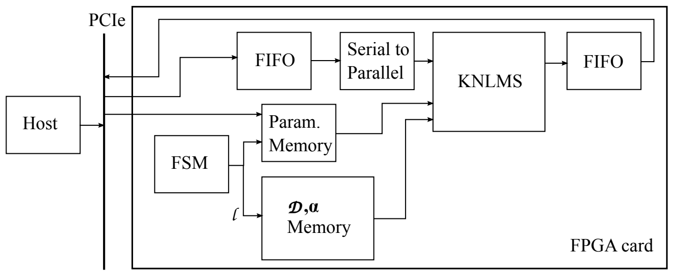

Fig. 7. Block diagram illustrating system integration of the KNLMS processor.

disables the KNLMS core and output FIFOs appropriately, and the host computer ensures that the FIFOs never overflow by controlling the amount of data being sent to the board.

Using the above arrangement, the same MG-30 benchmark set with 38000 samples was trained over a parameter space of $P = 280$ values. Execution time was 136.4ms, this corresponding to a processing time of 13.55ns per data per parameter, and a speedup of 70$\times$ over the CPU version. While this efficiency is only 23% of the highest achievable for the core, the total throughput is still far higher than the other implementations shown in Table IV. Note that, the system implementation used a different clock constraint in Vivado HLS in order to meet the timing requirements of the 250MHz clock on the VC707 dev board. This accounts for the difference in DSP and LUT utilisation between the float and system implementations in Table IV. The main sources of inefficiencies in our interface can be attributed to turnaround time of the PCIe bus and overheads associated with the host computer. Larger buffers should allow this to be greatly improved.

V. CONCLUSION

A fully pipelined FPGA implementation of KNLMS was presented which achieves higher performance than any previously reported design. Pipeline stages are filled with multiple independent tasks, corresponding to different machine learning parameter values, allowing high utilisation of resources. Using this approach a 70$\times$ speedup over an optimised software implementation was measured. This work demonstrated the feasibility of performing parameter search at nanosecond periods and opens the way for Big Data applications which were previously computationally intractable.

Future work will focus on techniques to further increase parallelism, explore precision tradeoffs and reduce the latency of the design. Other platforms will also be considered, such as GPU and heterogeneous architectures. Future work also includes systematically comparing KAFs in terms of their computational complexity and arithmetic intensity.

ACKNOWLEDGMENT

This research was supported under the Australian Research Councils Linkage Projects funding scheme (project number LP130101034).

REFERENCES

[1] B. Scholkopf and A. J. Smola, *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Cambridge, MA, USA: MIT Press, 2001.

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

[3] Y. Pang, S. Wang, Y. Peng, N. J. Fraser, and P. H. Leong, "A low latency kernel recursive least squares processor using FPGA technology," in *FPT*, 2013, pp. 144–151.

[4] C. Richard, J. C. M. Bermudez, and P. Honeine, "Online prediction of time series data with kernels," *Signal Processing, IEEE Transactions on*, vol. 57, no. 3, pp. 1058–1067, 2009.

[5] B. Widrow and M. J. Hoff, "Adaptive switching circuits," in *IRE WESCON Convention Record*, 1960, pp. 96–104.

[6] M. Yukawa, "Multikernel adaptive filtering," *Signal Processing, IEEE Transactions on*, vol. 60, no. 9, pp. 4672–4682, Sept 2012.

[7] D. Anguita, L. Carlino, A. Ghio, and S. Ridella, "A FPGA core generator for embedded classification systems," *Journal of Circuits, Systems and Computers*, vol. 20, no. 02, pp. 263–282, 2011.

[8] M. Papadonikolakis and C. Bouganis, "A scalable FPGA architecture for non-linear SVM training," in *ICECE Technology, 2008. FPT 2008. International Conference on*, 2008, pp. 337–340.

[9] A. Majumdar, S. Cadambi, M. Becchi, S. T. Chakradhar, and H. P. Graf, "A massively parallel, energy efficient programmable accelerator for learning and classification," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, pp. 6:1–6:30, Mar. 2012.

[10] S. Van Vaerenbergh, J. Via, and I. Santamaria, "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, 2006, pp. 789–792.

[11] 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.

[12] 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.

[13] 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.

[14] S. Van Vaerenbergh, "Kernel methods toolbox KAFBOX: a Matlab benchmarking toolbox for kernel adaptive filtering," 2012, software available at http://sourceforge.net/p/kafbox.

[15] M. Jacobsen, Y. Freund, and R. Kastner, "RIFFA: A reusable integration framework for FPGA accelerators." in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 2012, pp. 216–219. [Online]. Available: http://dblp.uni-trier.de/db/conf/fccm/fccm2012.html#JacobsenFK12