Braiding: a Scheme for Resolving Hazards in Kernel Adaptive Filters

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

***Abstract*—Computational cost presents a barrier in the application of machine learning algorithms to large-scale real-time learning problems. Kernel adaptive filters (KAFs) have low computational cost with the ability to learn online and are hence favoured for such applications. Unfortunately, dependencies of the outputs on the weight updates prohibit pipelining.**

This paper introduces a combination of parallel execution and conditional forwarding, called braiding, which overcomes dependencies by expressing the output as a combination of the earlier state and other examples in the pipeline.

To demonstrate its utility, braiding is applied to the implementation of classification, regression and novelty detection algorithms based on the Naive Online regularised Risk Minimization Algorithm (NORMA). Fixed point, open source implementations are described which can achieve data rates of around 130 MSamples/s with a latency of 10 to 13 clock cycles. This constitutes a two orders of magnitude increase in throughput and one order of magnitude decrease in latency compared to a single core CPU implementation.

I. Introduction

In recent years, there has been an exponential rise in the amount of data being acquired and generated. Attempting to extract useful information from the complex relationships present in such data has led to an increasing interest in machine learning. Some algorithms, such as support vector machines (SVM) [1], random forests (RF) [2], Gaussian processes (GP) [1] and neural networks [3], have demonstrated consistently good performance on various datasets [4]. However, their computational cost prevents their application to problems which require high throughput or low latency training. Examples include channel equalisation and machine prognostics [5], [6], [7].

In online and streaming applications, not all of the data is available a priori. As such, the model deduced by a machine learning algorithm must be updated as new data becomes available. Kernel adaptive filters (KAFs) [8] are a class of online, non-linear learning algorithms with substantially reduced computational requirements for an online setting. The reduction is achieved by finding a closed form solution to a model update step based on the previous model and a new data example. Unfortunately, the recursive nature of the update step creates a dependency which can limit the data rates of hardware implementations.

In this paper, we propose a hardware architecture for the implementation of the well-known Naive Online regularised Risk Minimization Algorithm (NORMA) [9], a KAF capable of regression, classification and novelty detection. The architecture features a fully pipelined datapath, with parallel execution and conditional forwarding to overcome all data hazards whilst incurring minimal increase in the number of arithmetic operations. A technique called *braiding*, is introduced in which: (1) all partial results are computed in parallel; (2) when the information regarding which should be included becomes available, an appropriate select signal is generated; and (3) a multiplexer selects the appropriate partial sum to add to the full sum. A related approach for the least mean square algorithm which employs correction terms has been proposed in [10].

978-1-4673-9091-0/15/$31.00 ©2015 IEEE

The contributions of this paper are as follows:

- The braiding technique which addresses data dependencies in sliding window based recursive algorithms.
- An open source, fixed point, fully pipelined implementation of NORMA which can learn a single non-linear predictive model at very high data rates. Using different loss functions, classification, regression and novelty detection are implemented.
- A comparison of the performance of different implementations of NORMA on reconfigurable computing platforms and a central processing unit (CPU). To the best of our knowledge, this has higher throughput and lower latency than any previously reported implementation of NORMA.

The paper is organised as follows: Section II describes NORMA and summarises previous works; Section III describes our proposed architecture and introduces the braiding concept; Section IV describes the performance of our implementation and makes a comparison with a single core CPU and a previously reported design; finally, conclusions are drawn in Section V.

II. Background

In this section, NORMA [9] is described along with a brief introduction to kernel methods and a review of prior works.

A. Kernel Methods

Kernel methods are a popular class of machine learning algorithms which are capable of modelling any continuous function with arbitrary accuracy [11]. Given a set of training pairs, $\{x_i, y_i\}$, where $x_i \in \mathbb{R}^m$ is the input vector and $y \in \mathbb{R}$ is the output, target or label. In general, the goal of kernel methods is to create a function (or model), $f(x)$, which can accurately predict y, given x. Kernel methods create $f(x)$ through a kernel function, $\kappa : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$, and the following learned parameters:

- a *dictionary*, $\mathcal{D}$, a subset of input vectors; and
- a vector of weights, $\boldsymbol{\alpha}$. One weight is required for each entry in the dictionary.

Using the dictionary and weights, $f(x)$, is defined as follows:

$$f(x) = \sum_{i=1}^{D} \alpha_i \kappa(x, d_i) \quad , \tag{1}$$

where D is the number of entries in the dictionary, α_i is the i^{th} element of $\boldsymbol{\alpha}$ and d_i is the i^{th} entry of $\mathcal{D}$. Their ability to learn a model is due to the kernel function, $\kappa(x_i, x_j)$, which computes the inner product between x_i and x_j after applying a mapping function, $\phi : \mathbb{R}^m \to \mathbb{F}$, to x_i and x_j, i.e. $\kappa(x_i, x_j) = \phi(x_i)^T \phi(x_j)$. This is often referred to as the *kernel trick* [1].

Some common kernel functions include:

- the linear kernel, $\kappa(x_i, x_j) = x_i^T x_j$;
- the Gaussian kernel, $\kappa(x_i, x_j) = \mathrm{e}^{-\gamma \|x_i - x_j\|_2^2}$; and
- the polynomial kernel, $\kappa(x_i, x_j) = (x_i^T x_j + c)^b$.

where γ, b and c are parameters chosen to suit the given problem. In this work, we use the Gaussian kernel due it being a universal approximator [11].

Several algorithms, such as SVM and kernel recursive least squares (KRLS), have been proposed to find $\mathcal{D}$ and $\boldsymbol{\alpha}$. However, the computational complexity of SVM ($O(n^3)$) [1] and KRLS ($O(n^2)$) [12] are not scalable for use in high frequency, real-time applications.

B. NORMA

NORMA is an $O(n)$ stochastic approximation of the SVM [9]. It is an online algorithm which can be applied to classification, regression or novelty detection. NORMA is based on the concept of minimizing the instantaneous risk of predictive error by taking a step in that direction given by:

$$f_{t+1} = f_t - \eta_t \partial_f R_{inst,\lambda}[f, x_{t+1}, y_{t+1}]\Big|_{f=f_t} \tag{2}$$

where f_t is the function, f, at time t, η is the step size, ∂ is the partial derivative operator and $R_{inst,\lambda}$ is the instantaneous risk function. Equation 2 is then expressed in terms of a loss function, $l(f_t(x_t), y_t)$, to penalize misclassifications or predictions of the function given by:

$$f_{t+1} = (1 - \eta\lambda) f_t - \eta_t l'(f_t(x_{t+1}), y_t) \kappa(x_{t+1}, \cdot) \quad . \tag{3}$$

Depending on the application, a suitable loss function can be provided to achieve a specific goal. For example, in this paper:

$$l(f(x) + b, y) = \max(0, \rho - y(f(x) + b)) - \nu\rho \tag{4}$$

$$l(f(x)) = \max(0, \rho - f(x)) - \nu\rho \tag{5}$$

$$l(f(x), y) = \max(0, |y - f(x)| - \epsilon) + \nu\epsilon \tag{6}$$

where η, λ, $\nu \in \mathbb{R}$ are parameters, are used for classification (4), novelty detection (5), and regression (6). It would be trivial to extend this work to other loss functions.

Reference [9] provides a complete derivation of equations 2 and 3. In this paper the factor decaying the weights of f_t or $(1 - \eta\lambda)$ in equation 3 is defined as the forgetting factor Ω. Differentiating the loss functions then substituting them into equation 3 leads to the following update expressions:

$$(\alpha_i, \alpha_t, b, \rho) = \begin{cases} (\Omega\alpha_i, 0, b, \rho + \eta\nu) \text{ if } y(f(x) + b) \geq \rho \\ (\Omega\alpha_i, \eta y, b + \eta y, \rho - \eta(1 - \nu)) \text{ otherwise} \end{cases} \tag{7}$$

$$(\alpha_i, \alpha_t, \rho) = \begin{cases} (\Omega\alpha_i, 0, \rho + \eta\nu) \text{ if } f(x) \geq \rho \\ (\Omega\alpha_i, \eta, \rho - \eta(1 - \nu)) \text{ otherwise} \end{cases} \tag{8}$$

$$(\alpha_i, \alpha_t, \epsilon) = \begin{cases} (\Omega\alpha_i, 0, \epsilon - \eta\nu) \text{ if } |y - f(x)| \leq \epsilon \\ (\Omega\alpha_i, \delta\eta, \epsilon + \eta(1 - \nu)) \text{ otherwise} \end{cases} \tag{9}$$

where $\delta = sign(y - f(x))$, $\Omega \leq 1$ and $f(x) = \sum \alpha_i \kappa(x, d_i)$. Equations 7, 8 and 9 are the update expressions for classification ($NORMA_c$), novelty detection ($NORMA_n$) and regression ($NORMA_r$) respectively. The expression determining the constant value Ω with respect to other parameters varies for the application. For the classification and novelty detection loss functions in this paper $\Omega = 1 - \eta$ whereas $\Omega = 1 - \lambda\eta$ for regression. In this paper Ω is used for simplicity to be consistent for all applications to refer to the value that decays the weights.

NORMA is used with truncation, commonly known as a sliding window, in which $\mathcal{D}$ is updated according to

$$[d_1, \cdots d_D] \to [x_t, d_1, \cdots d_{D-1}] \tag{10}$$

The examples retained by the function $f(x)$ are called support vectors for machine learning techniques such as support vector machines (SVM) but are more commonly referred to as the *dictionary* in the KAF literature. The example i in the dictionary is referred to d_i in this paper. The corresponding weights are referred to as α_i as defined in equations 7, 8 and 9. One significant observation in this work is that after the computation of $f(x)$, the update of the weights is a scalar multiplied by a vector while rest of the update expression consists of trivial scalar expressions easily implemented in hardware.

C. Related Work

FPGA-based implementations of KAFs have previously been proposed to achieve higher throughput, lower latency or decreased power usage. Matsubara et al. [10] described an implementation of the least mean square (LMS) algorithm which computed a solution ignoring data hazards, and then applied correction terms. A soft vector processor optimised for implementation of the KRLS algorithm was reported by Pang et al. [13], this having the advantage that different KAFs can be implemented with only software changes. Ren et al. [14] implemented a simpler algorithm, quantized kernel least mean squares (QKLMS) [15], which has many similarities to NORMA. Their approach computed the update sequentially, leading to small area, however the design described in our paper has $50\times$ higher throughput.

The highest throughput KAF architecture reported to date is an implementation of kernel normalized least mean squares (KNLMS)[16]. The design presented achieves a lower throughput however does not require multiple parallel problems. This allows application beyond parameter search or parallel problem domains.

III. Architecture

The pipelined hardware architecture of NORMA is based on three properties of its formulation:

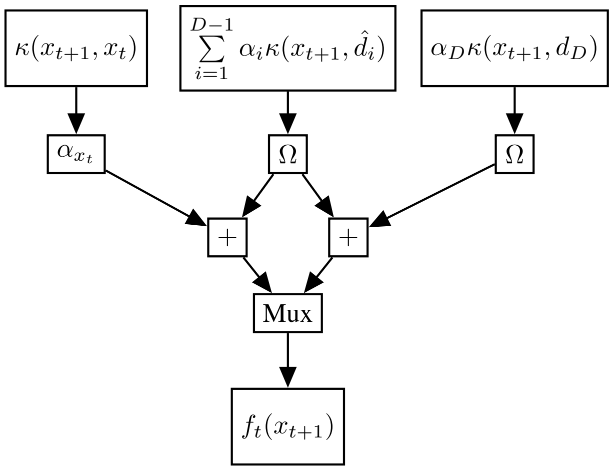
1) NORMA is a sliding window algorithm
2) each iteration, the weights decrease by a multiplicative factor ($\alpha_i \to \Omega\alpha_i$); and
3) the computational cost of the update is small compared to the evaluation of the decision function $f_t(x_{t+1})$.

The first property allows the decision function to be expressed as a combination of the examples currently in the pipeline, x_i, and a previous dictionary, $\hat{\mathcal{D}}$. The second provides a simple relationship to update the weights for the final dictionary reducing the computation to be completed each clock cycle. The third is necessary as it determines the critical path and hence the effectiveness of pipelining as one update is computed each clock cycle for a fully pipelined design. By using braiding, $f_t(x_{t+1}) = \sum_{i=1}^{D} \alpha_i \kappa(x_{t+1}, d_i)$ can be rewritten for a single pipeline stage as a function of the dictionary in the previous clock cycle ($\hat{\mathcal{D}}$):

$$f_t(x_{t+1}) = \sum_{i=1}^{D-1} \Omega\alpha_i \kappa(x_{t+1}, \hat{d}_i) \quad (11)$$
$$q\left\{ \quad + \begin{cases} 0 \text{ if } x_t \text{ is not added} \\ \alpha_{x_t}\kappa(x_{t+1}, x_t) \text{ otherwise} \end{cases} \right.$$
$$+ \sum_{i=D}^{D-q} \Omega\alpha_i \kappa(x_{t+1}, \hat{d}_i)$$

where q is 1 if x_t is added to the dictionary and 0 otherwise, Ω is the forgetting factor, D is the size of the dictionary and x_t is the example ahead in the pipeline. When the example x_t is added, the sliding window shifts to include this new example and discards the oldest example in the dictionary; this is represented by the third term in equation 11. If x_t is not added, the dictionary remains unchanged.

Figure 1 shows a hardware datapath which implements equation 11. The multiplexer chooses whether to add x_t to the dictionary, or discard it so that the oldest dictionary entry can now be merged into the sum. This design is analogous to braiding as the core of the sliding window has either $\kappa(x_{t+1}, x_t)$ or $\alpha_i\kappa(x_{t+1}, \hat{d}_i)$ braided into it each clock cycle. It can be expanded to an arbitrarily pipelined implementation as expressed in equation 12.



$\kappa(x_{t+1}, x_t)$
$\sum_{i=1}^{D-1} \alpha_i \kappa(x_{t+1}, \hat{d}_i)$
$\alpha_D \kappa(x_{t+1}, d_D)$
α_{x_t}
Ω
Ω
$+$
$+$
Mux
$f_t(x_{t+1})$

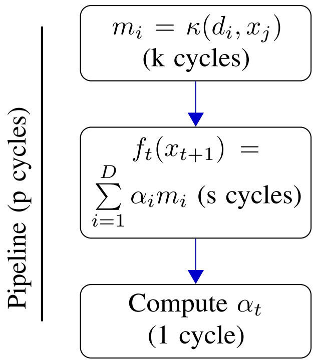
Fig. 1. Hardware representation of equation 11

$$f_t(x_{t+1}) = \sum_{i=1}^{D-p} \Omega^p \alpha_i \kappa(x_{t+1}, \hat{d}_i) \quad (12)$$
$$q\left\{ \begin{array}{l} + \begin{cases} 0 \text{ if } x_{t+1-p} \text{ is not added} \\ \Omega^{p-1}\alpha_{x_{t+1-p}}\kappa(x_{t+1}, x_{t+1-p}) \text{ otherwise} \end{cases} \\ + \begin{cases} 0 \text{ if } x_{t+2-p} \text{ is not added} \\ \Omega^{p-2}\alpha_{x_{t+2-p}}\kappa(x_{t+1}, x_{t+2-p}) \text{ otherwise} \end{cases} \\ \qquad\qquad \vdots \\ + \begin{cases} 0 \text{ if } x_t \text{ is not added} \\ \alpha_{x_t}\kappa(x_{t+1}, x_t) \text{ otherwise} \end{cases} \end{array} \right.$$
$$+ \sum_{i=D-p+1}^{D-q} \Omega^p \alpha_i \kappa(x_{t+1}, \hat{d}_i)$$

In this case q is the number of examples in the pipeline added to the dictionary, p is the number of pipeline stages and $\hat{d}_i$ are examples in the dictionary at time $t - p$. The first term is the portion of the decision function that depends on recent examples in the sliding window such that in p cycles time they are guaranteed to still be in the dictionary and hence must be used in the computation of $f_t(x_{t+1})$. The dependence of the decision function on the examples currently in the pipeline is expressed in the second component. Using the shift of q, the final term in equation 12 sums the contributions from the uncertain portion of the dictionary.

For a hardware implementation, the computation of $f_t(x_{t+1})$ is separated into two sections. The first is the computation of the kernel function, $\kappa(x_{t+1}, \hat{d}_i)$. This involves $D + p$ parallel kernel evaluation blocks to compute this function for each of the examples in the dictionary and pipeline.

Following this is a multiplication and sum of $\alpha_i\kappa_i$. This completes the computation of $f_t(x_{t+1})$ which is then passed to the update section to determine if the example is to be added.



$m_i = \kappa(d_i, x_j)$ (k cycles)
$f_t(x_{t+1}) = \sum_{i=1}^{D} \alpha_i m_i$ (s cycles)
Compute α_t (1 cycle)
Pipeline (p cycles)

Fig. 2. The pipelined calculation

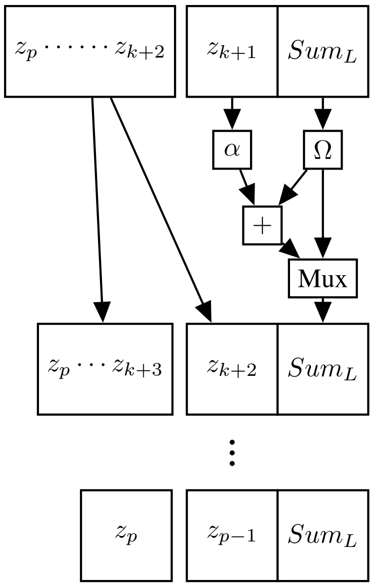


Fig. 3. The pipeline for Sum_L

A. Pipelining

Figure 2 shows a high level view of the proposed pipeline for the online training of the NORMA algorithm implementing equation 12. This pipeline assumes that $D > p = k+s+1$ using the notation for p, k, s defined in Figure 2. This is reasonable because the number of cycles for a single kernel evaluation, k, depends only on the number of features and, using a method such as an adder tree, s, scales as $O(\log_2(D))$. The three stages of the pipeline vary greatly in terms of the hardware requirements for their implementation. The kernel evaluation scales as $O(F(D+p))$ where F is the number of features, followed by the summation stage which is $O(D)$ additions and multiplications. The final stage that computes the update is $O(1)$ in hardware and time. However, as the update must take place in a single cycle to avoid data hazards, this final stage is the critical path restricting the clock frequency.

The kernel evaluation must begin by calculating $\kappa(x_{t+1}, \cdot)$ with the dictionary entries and all the examples in the pipeline as any of them could be needed in the computation of $f_t(x_{t+1})$. As x_{t+1} progresses along the pipeline, the possible combinations are constrained, either discarding the example at the final stage of the pipeline or the oldest example in the dictionary each clock cycle. Hence, each stage of the pipeline in the kernel evaluation contains one less example than the previous. After k cycles, this will result in $D+p-k = D+s+1$ values to propagate to the summation stage. These scalar values that are the result of the kernel evaluation are given the label of z_i for the pipelined examples and v_i for the dictionary examples.

The kernel evaluation stage outputs $D+s+1$ scalar values which are the results of evaluating the kernel function of an example with the current dictionary and the $s+1$ examples ahead in the pipeline. As shown in Figure 2, this is followed by the summation stage of $\sum \alpha_i z_i + \sum \alpha_i v_i$ where z_i is defined as the result of the kernel evaluation for pipelined examples and v_i for the dictionary entries. In terms of equation 12, z_i are the evaluations of the kernel function for terms within the q bracket and v_i are all other kernel evaluations. The summation stage does not use all values of z_i and v_i, discarding one each cycle as in equation 12. Additionally, for the examples ahead in the pipeline, the values of α_i have yet to be calculated. This requires the summation to be split into sum left ($SUM_L = \sum \alpha_i z_i$) and sum right ($SUM_R = \sum \alpha_i v_i$).

SUM_L is the summation of terms ahead in the pipeline shown in Figure 3 or the q terms in equation 12. This diagram shows the cumulative sum spread over $s-1$ cycles to first multiply the kernel evaluation with freshly computed α values and then added it to a running total of SUM_L which is initialized to 0 for the first cycle. Ω is the forgetting factor which is multiplied into the sum each clock cycle. The 'Mux' in Figure 3 and in all other figures in this paper refer to a multiplexor with two inputs and the control line connected to the decision of whether or not to add an example to the dictionary. SUM_L finishes with two remaining z values and the sum giving a total of three output values into the final stage of the sum. The forgetting factor is multiplied into SUM_L each clock cycle.

SUM_R differs from SUM_L as all the α values are readily available from the current dictionary. In the first stage the value $w_i = \alpha_i v_i$ is calculated for all examples followed by a pipelined adder tree to sum w_i. However, all w_i values cannot be summed immediately as some may be evicted from the dictionary and hence give the incorrect result for SUM_R. Under the assumption that the oldest example is removed, the examples in the dictionary that have an index less than $D-s-1$ or the examples that are 'young enough' are guaranteed to be in the final sum and therefore can be added in parallel. The other w_i values above this threshold index are questionable as they are in range of being shifted out of the dictionary by the sliding window. As they cannot be immediately added into SUM_R they are *braided* into the sum each clock cycle when it is determined that it is impossible for that dictionary entry to be shifted out before the calculation $f_t(x_{t+1})$ completes.

The term braiding is used to describe this structure as each clock cycle another term is combined into the sum analogous

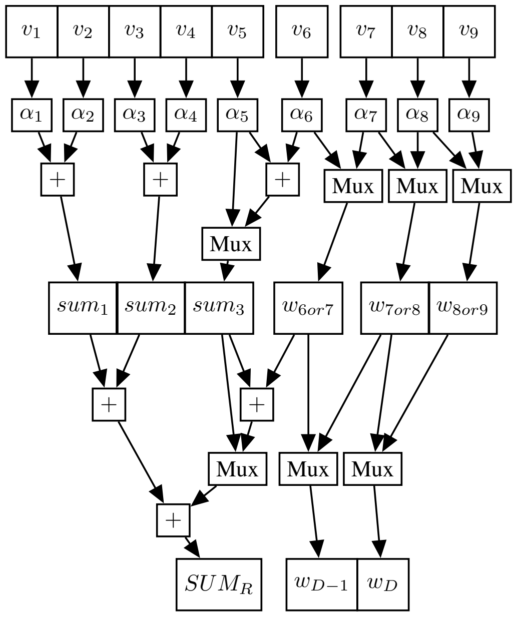


Fig. 4. An example 3 stage pipelined braiding sum

to a hair or rope braid. Figure 4 shows an example of SUM_R for a dictionary size of 9 with 3 pipeline stages. v_1 to v_5 are the 'young enough' examples as within the next 4 clock cycles they are certainly a component in the computation of $f_t(x_{t+1})$. These examples are also referred to as the core of the sliding window in this paper. v_6 to v_9 are uncertain as in 4 clock cycles they could all be removed if all the pipelined examples are added. In the first stage in the figure if an example at the end of the pipeline is added v_9 is discarded from the dictionary leaving three uncertain examples. Conversely, if it is not added then example v_6 is *braided* into the sum also leaving three uncertain examples. These are passed along to the next stage while the others are summed in an adder tree with braiding until only two unbraided examples remain.

The final stage of the summation combines SUM_L ($= \sum \alpha_i z_i$) and SUM_R ($= \sum \alpha_i v_i$) and the computation of α is shown in Figure 5. In Figure 4 the forgetting factor was temporarily ignored. Instead it is computed separately over $s-1$ clock cycles so when SUM_L and SUM_R are combined, this value of Ω^{s-1} is used to compensate as $\sum \Omega^{s-1}\alpha_i \cdot v_i = \Omega^{s-1}\sum \alpha_i \cdot v_i$. Furthermore, to account for an additional cycle, Ω is applied once more so the factor becomes Ω^s. SUM_R and SUM_L are then combined and the remaining task is to incorporate a single dictionary entry (w_D) and pipeline example (z_p) in the sum. In the final cycle, the value of $f_t(x_{t+1})$ can be computed. This is passed into the block $\alpha(f_t(x_{t+1}))$ which computes the weight α based on the application. α is then forwarded to the stages in Figure 5 and to the calculation of SUM_L. During the calcuation of α, whether to add an example to the dictionary or not must be determined. The result of this decision is passed to the select lines of all the multiplexers on the next clock cycle.

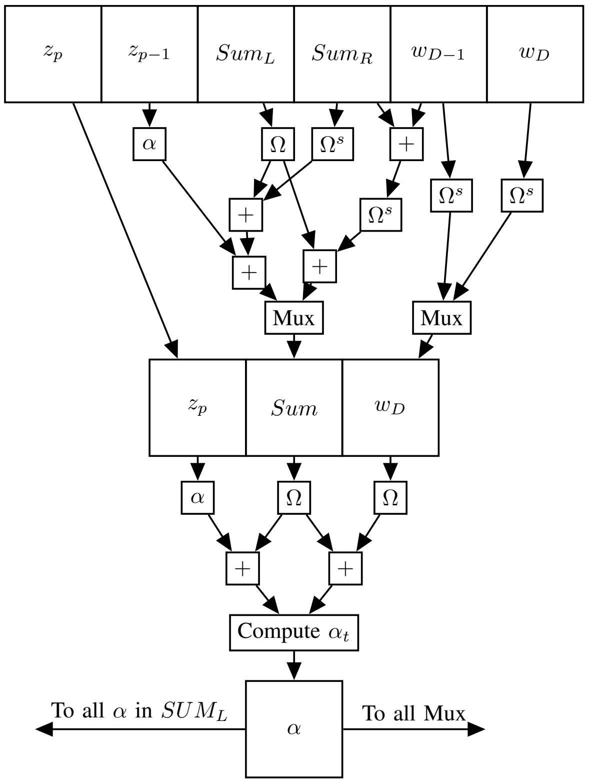


Fig. 5. Data path of the last stage of sum and the update computation

B. Fixed Point

To reduce hardware usage and latency on the critical path involving computing the update, the implementation uses fixed point arithmetic.

As the weights and output of the kernel evaluation are tightly bounded to be scalar values less than a magnitude of 1, the impact of fixed point on the accuracy of the algorithm should not be severe if sufficient precision is used. The update step of NORMA is examined to assess the impact of a fixed point implementation with 12 fractional bits. It is characterized by an initial value η multiplied by the forgetting factor Ω each clock cycle. Under the assumption that all examples are added leads to the maximum value of the oldest dictionary weight being $\eta\Omega^{D-1}$. Replacing Ω with $1-\eta$ for classification and novelty detection [9] leads to the calculation of $\eta = \frac{1}{D}$ to obtain the maximum value for the last dictionary entry. This results in $\frac{1}{D}(1-\frac{1}{D})^{D-1}$ which for 12 fractional bits and $D = 200$ is $\frac{7.55}{2^{12}}$. The more realistic assumption that half the examples are added further reduces this to $\frac{2.77}{2^{12}}$. Using fixed point multiplication with truncation erodes this further demonstrating that the use of fixed point can restrict the effective dictionary size of NORMA when insufficient precision is used. A brute force search for the maximum dictionary for each possible fractional width (f_w) was performed under the assumption that every example is added. From these results the expression $D \leq 3 * 2^{\frac{f_w-3}{2}}$ approximates the maximum size of the dictionary when using fixed point. There is little point in having a larger dictionary than this as all older dictionary entries will have a weight

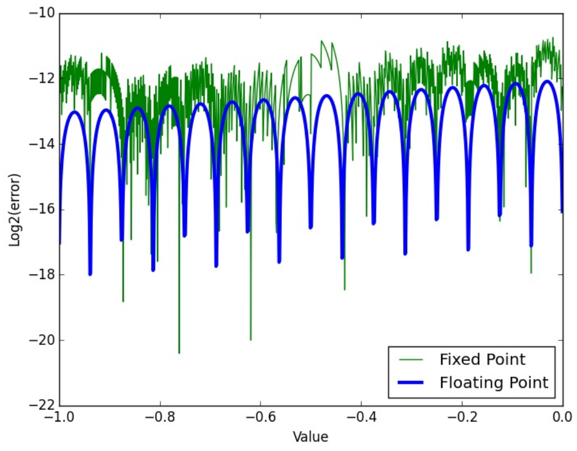


Fig. 6. Approximation error of a 16 value lookup table

of zero. In reality, the size of the dictionary should be much smaller than this bound as η likely will not be optimal for the dictionary size and the assumption that all examples are added is strong. To achieve reasonable utilization from the dictionary it should be quite a bit smaller than this bound.

Without loss of generality, the Gaussian kernel, $\kappa(x_i, x_j) = e^{-\gamma\|x_i - x_j\|_2^2}$, was implemented in our design since it is a commonly used kernel. Our fixed point implementation uses the property that $e^{bx} = 2^{ax}$ for a given b [17]. The value of ax is split into an integer and fractional component, giving $2^{ax} = 2^{int(ax)}2^{frac(ax)}$ where $-1 \leq frac(ax) \leq 0$. Using the restriction that $ax \leq 0$ for the Gaussian kernel allows the value of $2^{frac(ax)}$ to be calculated using a lookup table with linear interpolation between -1 and 0. 2^{ax} is then obtained by bit shifting this result with the integer component. This allows the exponentiation to be computed in 5 operations or a couple of clock cycles while the calculation of $\|x-y\|_2^2$ depends on the size of the adder tree constrained by the number of features used. Figure 6 shows the error associated with using this method. The two traces represent linear interpolation computed in floating point and fixed point with fractional width of 12. For a 16 value lookup table and 12 fractional bits, the error only affects the least significant one or two bits.

IV. Results and Discussion

The proposed architecture was implemented using CHISEL [18] with an open source implementation available at github.com/da-steve101/chisel-pipelined-olk.git. A modified Kernlab implementation of NORMA for use with the caret package to obtain the baseline floating point learning performance of NORMA on various datasets. A C implementation of NORMA was created to measure the speed of a CPU implementation for comparison. This is also available in our repository. All other datasets and scripts used to obtain the results in this paper are available in the repository with instructions in the README. The code was synthesised for a Xilinx Virtex-7 XC7VX485T-2FFG1761C on a VC707 development board with speed grade

TABLE I
Performance of $NORMA_n$ for F = 8 and Lookup Table = 16 points on a Virtex 7

D =	**16**	**32**	**64**	**128**	**200**
Fixed 8.10					
DSPs (/2800)	309	514	911	1679	2556
Freq (MHz)	133.0	137.8	137.4	131.0	127.3
Latency (clocks)	10	11	12	12	13
Slices (/75900)	4615	8194	14663	29113	46443
Fixed 8.16					
DSPs (/2800)	595	988	1749	2800	*
Freq (MHz)	115.2	114.1	93.2	98.2	*
Latency (clocks)	10	11	12	12	*
Slices (/75900)	6188	11622	20512	58889	*
Fixed 8.22					
DSPs (/2800)	1236	2056	*	*	*
Freq (MHz)	101.2	97.29	*	*	*
Latency (clocks)	10	11	*	*	*
Slices (/75900)	10971	18976	*	*	*
Fixed 8.28					
DSPs (/2800)	1236	2056	*	*	*
Freq (MHz)	97.2	89.8	*	*	*
Latency (clocks)	10	11	*	*	*
Slices (/75900)	13819	23931	*	*	*

2.

The resource usage and clock frequency was obtained for a varying dictionary size with $F = 8$ features for the $NORMA_n$ application are summarised in Table I. This table shows the growth in resource usage with dictionary size and fixed point precision. From this table it is clear that the number of DSP's is the bottleneck for this architecture. This is because approximately $D * (F + 1)$ multiplications are required for the kernel evaluation. The number of DSP's used also grows with the bitwidth as the full result can no longer be computed in a single DSP for more than 18 bits. An increasing fixed point bitwidth also results in a longer delay in the multiplication stage, reducing the maximum clock frequency also shown in the table.

Tables II, III and IV compare fixed point implementations with the Kernlab R implementation [19] as a floating point reference. The caret package [20] facilitated this comparison, and parameters for the floating point implementation were chosen using a parameter search with 10 fold cross validation [21]. Training was conducted using the H and Area Under the Receiver Operating Characteristic Curve (AUC) measures for classification and novelty detection, and the L1 and L2 error for regression. The AUC and H measures were chosen as they provide a better overview of the classifier or novelty detector than accuracy [22]. Both AUC and H measures are used as there is a lack of consensus on which should be used [23]. An optimal classifier has $AUC = 1$ and $H = 1$ whereas a classifier that randomly guesses should have an $AUC = 0.5$ and $H = 0$. The fixed point implementation was then tested using the same parameters found with the floating point cross validation to assess its impact. A lookup table with 16 points as shown in Figure 6 was also used unless otherwise stated. In Tables II, III and IV, i_w denotes the integer width that was used. This was determined by finding the minimum value

TABLE II
RESULTS FOR $NORMA_c$ IN FIXED VS FLOATING POINT

Dataset	Artificial ($i_w = 7$)		Satellite ($i_w = 8$)	
Measure	AUC	H	AUC	H
Floating $NORMA_c$	0.893	0.550	0.995	0.947
Fixed 18 bits	0.618	0.108	0.673	0.145
Fixed 24 bits	0.745	0.255	0.996	0.922
Fixed 30 bits	0.899	0.579	0.998	0.954
Fixed 36 bits	0.903	0.586	0.997	0.959

TABLE III
RESULTS FOR $NORMA_n$ IN FIXED VS FLOATING POINT

Dataset	Artificial ($i_w = 8$)		Satellite ($i_w = 8$)	
Measure	AUC	H	AUC	H
Floating $NORMA_n$	0.641	0.140	0.836	0.358
Fixed 18 bits	0.664	0.174	0.5	0
Fixed 24 bits	0.658	0.162	0.503	0.130
Fixed 30 bits	0.658	0.162	0.800	0.295
Fixed 36 bits	0.658	0.162	0.825	0.348

before overflow occurs.

The datasets chosen to benchmark for classification and novelty detection were an artificial dataset generated using *sklearn.make_classification* [24] and the mlbench dataset *Satellite* [25]. The parameters chosen for the artificial dataset were *n_samples* = 1000, *n_features* = 8, *n_informative* = 4, *n_redundant* = 2 and *random_state* = 101. For the *Satellite* dataset, the multiclass problem was simplified to a two class problem by taking the largest class out and combining the others. The novelty detector for this dataset trains on one of the classes in the artificial dataset and for the satellite dataset it trains on the combined classes to detect the 'anomalous' red soil class. The regression datasets were an artificial dataset generated using *sklearn.make_regression* and the UCI *Combined Cycle Power Plant* dataset from [26]. The parameters chosen for the artificial dataset were *n_samples* = 1000, *n_features* = 8, *n_informative* = 6 and *random_state* = 101. All the datasets were then partitioned into a test and training set using the *createDataPartition* function in caret with an 80% split. The preprocessing functionality in caret was then used to center and scale the datasets using the training set with the exception of the artificial classification dataset. These four datasets are available from our github repository, together with four parameters files which enable the results to be reproduced.

The results in Tables III and IV show that the use of fixed point does not have an impact on the learning performance of the algorithm for bitwidths greater than 18 bits. The

TABLE IV
RESULTS FOR $NORMA_r$ IN FIXED VS FLOATING POINT

Dataset	Artificial ($i_w = 6$)		Combined Cycle Power Plant Dataset ($i_w = 6$)	
Measure	L1	L2	L1	L2
Floating $NORMA_r$	0.773	0.934	0.700	0.697
Fixed 18 bits	0.760	0.899	0.637	0.621
Fixed 24 bits	0.776	0.939	0.666	0.643
Fixed 30 bits	0.777	0.943	0.653	0.628
Fixed 36 bits	0.777	0.943	0.653	0.628

TABLE V
SINGLE CORE CPU LEARNING PERFORMANCE WITH F=8 COMPARED WITH $NORMA_n$ USING 8.10 FIXED POINT

D =	16	32	64	128	200
Freq (MHz)	2.83	1.51	0.77	0.38	0.25
Latency/example (ns)	353.2	660.9	1293.2	2625.2	4025.8
FPGA Speedup (×)	47.0	91.3	178.44	344.7	509.2
Latency Reduction (×)	4.69	8.296	14.87	28.7	39.2

TABLE VI
IMPACT OF GAUSSIAN KERNEL APPROXIMATION (7.29 ON ARTIFICIAL TWO CLASS)

Table Size	1	2	4	8	16
AUC	0.907	0.906	0.904	0.903	0.903
H	0.588	0.590	0.586	0.586	0.586

cases in which fixed point actually improves the learning performance of the algorithm are attributed to noise in the results. For classification Table II shows that using fixed point with insufficient precision does have a detremental impact on the results which is the expected case for general datasets. For implementations with similar fractional width precision to IEEE 754 (24 bits) there is no noticable difference between fixed and floating point implementations.

As the fixed point increases in precision the learning performance is expected to improve. This is a trade off with the amount of hardware used hence affecting the maximum dictionary size as shown in Table I. Due to fixed point changing the nature of the algorithm, performing the cross validation in fixed point may achieve better results than a cross validation in floating point.

Table I shows the resource usage and clock frequency for NORMA with novelty detection in fixed point. Novelty detection has a shorter critical path in the update step compared to classification and regression resulting in a reduction in clock frequency of around 10-15 MHz compared to the other applications. The resource usage is however very similar as the majority of the calculation is the same.

Table V shows the results for a single core C implementation of $NORMA_n$ compiled using gcc 4.9.2 with '-O3' and run on an Intel i7-4510U @ 2.00GHz compared with the results in Table I for the 18 bit fixed point implementation. It is noted that, by increasing the number of features used (F), the speedup of the FPGA implementation improves dramatically as all additional features are calculated in parallel with minimal impact on the clock frequency. However, this increases the amount of hardware needed for the implementation hence restricting the dictionary size.

Table VI shows the effectiveness of the Gaussian kernel approximation. There is no noticable difference in learning performance on the artificial two class dataset even when there is a single line from 0 to -1 that is shifted for integer powers. It is suggested by the authors that, due to the fixed initial value of the weights, NORMA is insensitive to the accuracy of the kernel function. The impact on other algorithms will be investigated in future work.

The NORMA implementation in this paper can be compared to the implementation of a KRLS vector processor designed for low latency machine learning [13]. While KRLS based algorithms have been shown to have superior learning performance to NORMA [27] the latency in the online learning is about two orders of magnitude lower with three orders of magnitude increase in throughput for the implementation described in this paper. Compared to the approach in [14], which reports a sample rate of 2.4 MHz, braiding provides a $50\times$ increase in throughput. The algorithms are similar enough that a direct comparision is meaningful as the braiding technique could be applied to QKLMS. A high throughput pipelined implementation of KNLMS is described in [16] achieved by time multiplexing multiple problems. For a single problem or dataset the throughput of this architecture is much higher. Braiding relies on properties of NORMA that present in other algorithms such as [28] and hence could be applied there as well.

V. Conclusion

A novel braiding technique for sliding window algorithms was presented. Our study showed that when applied to a fixed-point implementation of NORMA, extremely low latency and high throughput can be achieved with similar learning ability to a floating-point implementation. NORMA was chosen as an example application for braiding as it allows the flexibility to implement classification, regression and novelty detection algorithms.

In this work, the same fixed point wordlength was used throughout each design. A more efficient scheme could vary the precision along the datapath to achieve similar accuracy with smaller wordlengths. Implementations of NORMA with multiple FPGAs could enable larger dictionary sizes. Finally, the braiding scheme described relies on properties of NORMA which are present in other algorithms. Generalisation to these problems would also be promising candidates for further research. This work has been made open source and the results reproducible, with a desire that others make improvements and utilise it in real-world machine learning problems.

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] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[3] C. M. Bishop, *Neural networks for pattern recognition*. Oxford university press, 1995.

[4] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim, "Do we need hundreds of classifiers to solve real world classification problems?" *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3133–3181, 2014.

[5] Y. Ji, T. Heinze, and Z. Jerzak, "HUGO: real-time analysis of component interactions in high-tech manufacturing equipment (industry article)," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM, 2013, pp. 87–96.

[6] R. Daniels and R. W. Heath Jr, "Online adaptive modulation and coding with support vector machines," in *Wireless Conference (EW), 2010 European*. IEEE, 2010, pp. 718–724.

[7] F. Perez-Cruz, J. J. Murillo-Fuentes, and S. Caro, "Nonlinear channel equalization with gaussian processes for regression," *Signal Processing, IEEE Transactions on*, vol. 56, no. 10, pp. 5283–5286, 2008.

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

[9] 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.

[10] K. Matsubara, K. Nishikawa, and H. Kiya, "A new pipelined architecture of the LMS algorithm without degradation of convergence characteristics," in *Acoustics, Speech, and Signal Processing, 1997. ICASSP-97., 1997 IEEE International Conference on*, vol. 5. IEEE, 1997, pp. 4125–4128.

[11] B. Hammer and K. Gersmann, "A note on the universal approximation capability of support vector machines," *Neural Processing Letters*, vol. 17, no. 1, pp. 43–53, 2003.

[12] Y. Engel, S. Mannor, and R. Meir, "The kernel recursive least squares algorithm," *IEEE Transactions on Signal Processing*, vol. 52, pp. 2275–2285, 2003.

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

[14] 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.

[15] 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.

[16] N. J. Fraser, D. J. Moss, J. Lee, S. Tridgell, C. T. Jin, and P. H. Leong, "A fully pipelined kernel normalised least mean squares processor for accelerated parameter optimisation," in *Proc. International Conference on Field Programmable Logic and Applications (FPL), page to appear*, 2015.

[17] M. Wielgosz, E. Jamro, and K. Wiatr, "Highly efficient structure of 64-bit exponential function implemented in FPGAs," in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2008, pp. 274–279.

[18] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 1216–1225.

[19] A. Karatzoglou, A. Smola, K. Hornik, and M. A. Karatzoglou, "Package kernlab," 2015.

[20] M. Kuhn, "caret: Classification and regression training," *Astrophysics Source Code Library*, vol. 1, p. 05003, 2015.

[21] R. Kohavi *et al.*, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Ijcai*, vol. 14, no. 2, 1995, pp. 1137–1145.

[22] D. J. Hand and C. Anagnostopoulos, "A better beta for the H measure of classification performance," *Pattern Recognition Letters*, vol. 40, pp. 41–46, 2014.

[23] C. Ferri, J. Hernández-Orallo, and P. A. Flach, "A coherent interpretation of AUC as a measure of aggregated classification performance," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 657–664.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[25] F. Leisch, M. F. Leisch, and Z. No, "The mlbench package," 2007.

[26] P. Tüfekci, "Prediction of full load electrical power output of a base load operated combined cycle power plant using machine learning methods," *International Journal of Electrical Power & Energy Systems*, vol. 60, pp. 126–140, 2014.

[27] S. Van Vaerenbergh and I. Santamarıa, "A comparative study of kernel adaptive filtering algorithms," in *Digital Signal Processing and Signal Processing Education Meeting (DSP/SPE), 2013 IEEE*. IEEE, 2013, pp. 181–186.

[28] G. Li, C. Wen, Z. G. Li, A. Zhang, F. Yang, and K. Mao, "Model-based online learning with kernels," *Neural Networks and Learning Systems, IEEE Transactions on*, vol. 24, no. 3, pp. 356–369, March 2013.