Kernel Normalised Least Mean Squares with Delayed Model Adaptation

NICHOLAS J. FRASER and PHILIP H. W. LEONG, The University of Sydney, Australia

Kernel adaptive filters (KAFs) are non-linear filters which can adapt temporally, and have the additional benefit of being computationally efficient through use of the "kernel trick". In a number of real world applications, such as channel equalisation, the non-linear mapping provides significant improvements over conventional linear techniques such as the least mean squares (LMS) and recursive least squares (RLS) algorithms. Prior works have focused mainly on the theory and accuracy of KAFs, with little research on their implementations. This paper proposes several variants of algorithms based on the kernel normalised least mean squares (KNLMS) algorithm which utilise a delayed model update to minimise dependencies. Subsequently, this work proposes corresponding hardware architectures which utilise this delayed model update in order to achieve high sample rates and low latency, while also providing high modelling accuracy. The resultant delayed KNLMS (DKNLMS) algorithms can achieve clock rates up to 12× higher than the standard KNLMS algorithm, with minimal impact on accuracy and stability. A system implementation, achieves 250 GOps/s and a throughput of 187.4 MHz on an Ultra96 board, 1.8× higher throughput than previous state-of-the-art.

CCS Concepts: • **Computing methodologies** → **Kernel methods**; • **Computer systems organization** → **Data flow architectures**; • **Theory of computation** → *Online learning algorithms*; Algorithm design techniques;

Additional Key Words and Phrases: Kernel Adaptive Filters, Kernel Normalized Least Mean Squares, Field Programmable Gate Array

ACM Reference Format:
Nicholas J. Fraser and Philip H. W. Leong. 2018. Kernel Normalised Least Mean Squares with Delayed Model Adaptation. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (January 2018), 30 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

In recent years, several online kernel-based adaptive filters have been proposed to solve non-linear regression problems. These kernel adaptive filters (KAFs) utilise the Mercer kernel [2] to allow linear techniques be applied to non-linear feature spaces without directly computing the feature vectors [9]. Most KAFs are conceptually similar to their linear counterparts, e.g., the least mean squares (LMS) and recursive least squares (RLS) algorithms, but have been reformulated in the *dual space* [24] and utilise Mercer kernels. KAFs form a growing research field related to machine learning and digital signal processing, particularly in real-time environments. KAFs are popular in applications that require: non-linear modelling capability; adaptation to changing conditions; and less computational demand than deep neural networks.

Authors' address: Nicholas J. Fraser; Philip H. W. Leong, The University of Sydney, School of Electrical and Information Engineering, Sydney, NSW, 2006, Australia.

1936-7406/2018/1-ART1 $15.00
https://doi.org/0000001.0000001

The potential benefits of KAFs have been demonstrated by several previous works including applications, such as channel equalisation [35] and time series prediction [30], where high data rates may be required. However very few works, with the notable exception of Tridgell et al. [32], have considered whether high throughput implementations are feasible. All KAFs are online algorithms with recursive expressions in order to update their models based on new examples. These create dependencies which must be addressed in order for high performance hardware to be created.

This paper addresses this problem by considering the effect of pipelining the kernel normalised least mean squares (KNLMS) algorithm. We show that by modifying the KNLMS algorithm to have *delayed model adaptation* (described in Section 3), high frequency implementations can be realised using field programmable gate array (FPGA) technology. Specifically, the contributions of this paper are:

- a technique for modifying KAFs, delayed model adaptation, to allow them to be pipelined;
- an application of delayed model adaptation to the KNLMS algorithm, producing the delayed KNLMS (DKNLMS) algorithm and further generalisation to multiple delays multi-DKNLMS (MDKNLMS);
- two further variations of DKNLMS, dictionary guarding DKNLMS with dictionary guarding (DKNLMS-DG) and correction terms DKNLMS with correction terms (DKNLMS-CT), the latter of which can achieve higher accuracy while maintaining high performance;
- a description of hardware architectures to implement the KNLMS and DKNLMS algorithms with comparisons of speed, area and scalability;
- an empirical analysis of the learning accuracy of the DKNLMS algorithm including comparisons with LMS, KNLMS and other KAFs;
- an empirical analysis of the effect of using 18-bit fixed point precision and some function approximation (exponential, division) on the overall accuracy and stability of DKNLMS; and
- all source code used to generate the results in Section 6 has been open sourced[1], so that interested readers may easily reproduce and extend the work.

The delayed model adaptation used in this work, can be considered as an extension to similar work on pipelining linear adaptive filters. In particular, the work by Long et al. [25], Poltmann [28], Douglas et al. [13] and Yi et al. [39]. The architecture described in the this work, can be considered as an extension to the work by Fraser et al. [19] to implement the DKNLMS variants.

The paper is arranged as follows: Section 2 briefly explains kernel methods, the KNLMS algorithm and previous pipelined adaptive filtering architectures; Section 3 describes the delayed model adaptation method and the application of it to the KNLMS algorithm; Sections 4 and 5 describes the proposed architecture and implementation details respectively; Section 6 shows accuracy results of the algorithm and the performance of the implementation; and finally, conclusions are drawn in Section 7.

2 BACKGROUND

The KNLMS algorithm is a type of KAF [24], which are variants of adaptive filters [21], such as LMS and RLS-based algorithms [11, 12, 38], utilising the kernel trick [9]. In this section, the KNLMS algorithm [30] is described and summarised. Issues associated with hardware implementations are highlighted including: dictionary growth and data dependencies. Previous implementations of KAFs and high performance linear adaptive filters are also briefly reviewed.

[1] https://bitbucket.org/nick_fraser/knlms_core_gen

2.1 Kernel Adaptive Filtering

KAFs are online algorithms which create non-linear models to fit a set of training examples. The training examples consist of input/output pairs $\mathbf{x}_n \in \mathbb{R}^M$ and $y_n \in \mathbb{R}$, representing the input vector and desired output value. The kernel adaptive filter model is represented by the following:

- A positive definite kernel function, $\kappa(\mathbf{x}_i, \mathbf{x}_j)$, chosen at design time. Examples of kernel functions are: the Gaussian kernel, $\kappa(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|_2^2}$, where γ is a parameter chosen at design time; the polynomial kernel, $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \left(\mathbf{x}_i^T \mathbf{x}_j + c\right)^d$, where c and d are chosen at design time; and the linear kernel, $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \mathbf{x}_j$.
- A dictionary, given by $\mathcal{D}$, which is a subset of training example inputs.
- A vector of weights, $\boldsymbol{\alpha}$, where one weight corresponds to a single dictionary entry.

Given a new example, $\mathbf{x}_n$, a prediction, $\tilde{y}_n$, is calculated as follows:

$$\tilde{y}_n = \sum_{i=1}^{\tilde{N}_{n-1}} \alpha_i \kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_i) \quad , \tag{1}$$

where $\tilde{N}_{n-1}$ is the number of entries in $\mathcal{D}$ at time $n-1$, henceforth denoted as $\mathcal{D}_{n-1}$, $\tilde{\mathbf{x}}_i$ is the i^{th} entry of $\mathcal{D}$, and α_i is the i^{th} entry of $\boldsymbol{\alpha}$. In general, the goal of KAFs is to find $\boldsymbol{\alpha}$ and $\mathcal{D}$ which accurately predicts $y_n \; \forall \; n$.

2.2 Kernel Normalised Least Mean Squares

Given a new example, $\{\mathbf{x}_n, y_n\}$, the goal of training is to update the model, $(\mathcal{D}_{n-1}, \hat{\boldsymbol{\alpha}}_{n-1})$, from time $n-1$ to time n, where $\hat{\boldsymbol{\alpha}}_{n-1}$ is the approximation to $\boldsymbol{\alpha}$ at time $n-1$.

In order to calculate the prediction of y_n and to evaluate the *coherence criterion* [30] (which decides whether to add $\mathbf{x}_n$ to the dictionary), a kernel evaluation must be made between $\mathbf{x}_n$ and all entries in the dictionary. We first define $\mathbf{k}_{n-1}(\cdot) \in \mathbb{R}^M \rightarrow \mathbb{R}^{\tilde{N}_{n-1}}$ as a function which, when applied to a new input, $\mathbf{x}_n$ is given by:

$$\mathbf{k}_{n-1}(\mathbf{x}_n) = [\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_1), \ldots, \kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_{\tilde{N}_{n-1}})]^T \quad , \tag{2}$$

Conceptually, this kernel vector can be thought of as a vector of kernel evaluations between input $\mathbf{x}_n$ and each entry of $\mathcal{D}_{n-1}$. The coherence between $\mathbf{x}_n$ and $\mathcal{D}_{n-1}$ is given by:

$$\mu = \max_i \frac{|\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_i)|}{\sqrt{\kappa(\mathbf{x}_n, \mathbf{x}_n)\kappa(\tilde{\mathbf{x}}_i, \tilde{\mathbf{x}}_i)}} \;\; \text{s.t.} \;\; i \in \left\{1, \cdots, \tilde{N}_{n-1}\right\} \quad . \tag{3}$$

The example, $\mathbf{x}_n$ is added to the dictionary if $\mu < \mu_0$, where μ_0 is a parameter specified at design time to control the size and coherence of the dictionary. The idea of the coherence criterion is to prevent redundant training examples from being added to the dictionary, i.e., if $\mathcal{D}_{n-1}$ contains an entry very similar to $\mathbf{x}_n$, then there would be limited benefit to adding $\mathbf{x}_n$ to $\mathcal{D}_{n-1}$. However, adding $\mathbf{x}_n$ would cost us more in memory and computational requirements of KNLMS at time n, so if $\mu \geq \mu_0$, then we prefer not to add $\mathbf{x}_n$ to the dictionary. Note that if $\kappa(\cdot, \cdot)$ is a unit norm kernel, Eq. (3) can be simplified to:

$$\mu = \max_i |\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_i)| \;\; \text{s.t.} \;\; i \in \left\{1, \cdots, \tilde{N}_{n-1}\right\} \quad . \tag{4}$$

The Gaussian kernel used in this work is a unit norm kernel, so we use Eq. (4), rather than Eq. (3).

The kernel vector is then updated, creating $\mathbf{k}_n(\mathbf{x}_n) = [\mathbf{k}_{n-1}(\mathbf{x}_n)^T, \kappa(\mathbf{x}_n, \mathbf{x}_n)]^T$ if $\mathbf{x}_n$ is added, or else simply $\mathbf{k}_n(\mathbf{x}_n) = \mathbf{k}_{n-1}(\mathbf{x}_n)$. Similarly, $\hat{\boldsymbol{\alpha}}_{n-1}$ is appended with a zero, if $\mathbf{x}_n$ is added to the

dictionary. The KNLMS update step is derived by solving the following optimization problem:

$$\hat{\boldsymbol{\alpha}}_n = \underset{\boldsymbol{\alpha}}{\operatorname{argmin}} \|\boldsymbol{\alpha} - \hat{\boldsymbol{\alpha}}_{n-1}\|_2 \text{ s.t. } y_n = \mathbf{k}_n(\mathbf{x}_n)^T \boldsymbol{\alpha} \ . \tag{5}$$

The solution to which can be found by minimising the Lagrangian function:

$$J(\boldsymbol{\alpha}, \boldsymbol{\lambda}) = \|\boldsymbol{\alpha} - \hat{\boldsymbol{\alpha}}_n\|_2 + \lambda(y_n - \mathbf{k}_n(\mathbf{x}_n)^T \boldsymbol{\alpha}) \ . \tag{6}$$

where λ is a Lagrangian multiplier. Differentiating Eq. (6) with respect to $\boldsymbol{\alpha}$ and λ and solving for zero, results in the following expressions for $\hat{\boldsymbol{\alpha}}_n$:

$$2(\hat{\boldsymbol{\alpha}}_n - \hat{\boldsymbol{\alpha}}_{n-1}) = \lambda \mathbf{k}_n(\mathbf{x}_n) \tag{7}$$

$$\mathbf{k}_n(\mathbf{x}_n)^T \hat{\boldsymbol{\alpha}}_n = y_n \ . \tag{8}$$

Solving these equations for $\hat{\boldsymbol{\alpha}}_n$ leads to following update equation:

$$\hat{\boldsymbol{\alpha}}_n = \hat{\boldsymbol{\alpha}}_{n-1} + \eta \frac{y_n - \mathbf{k}_n(\mathbf{x}_n)^T \hat{\boldsymbol{\alpha}}_{n-1}}{\epsilon + \|\mathbf{k}_n(\mathbf{x}_n)\|_2^2} \mathbf{k}_n(\mathbf{x}_n) \ , \tag{9}$$

where η is a step size parameter and ϵ is a regularisation parameter, both of which are chosen at design time.

The KNLMS algorithm can be described using the psuedocode given is Algorithm 1.

ALGORITHM 1: KNLMS algorithm with coherence criterion and a unit norm kernel function.

Choose values for the step-size, η, and the regularization factor, ϵ.
Initialise $\mathcal{D} = \{\tilde{\mathbf{x}}_1\}$, $\boldsymbol{\alpha}_1 = [\frac{\eta}{1+\epsilon} y_1]$
while $n > 1$ **do**
  Get $\{\mathbf{x}_n, y_n\}$.
  Calculate $\mathbf{k}_{n-1}(\mathbf{x}_n) = [\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_1), \cdots, \kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_{\tilde{N}})]^T$.
  $\mu = \max(|\mathbf{k}_{n-1}(\mathbf{x}_n)|)$.
  if $\mu < \mu_0$ **then**
    $\tilde{N} = \tilde{N} + 1$.
    Append $\mathbf{x}_n$ to $\mathcal{D}$.
    Append 0 to $\hat{\boldsymbol{\alpha}}_{n-1}$.
    $\mathbf{k}_n(\mathbf{x}_n) = [\mathbf{k}_{n-1}(\mathbf{x}_n)^T, \kappa(\mathbf{x}_n, \mathbf{x}_n)]^T$.
  else
    $\mathbf{k}_n(\mathbf{x}_n) = \mathbf{k}_{n-1}(\mathbf{x}_n)$.
  end if
  Calculate $\hat{\boldsymbol{\alpha}}_n$ using Eq. (9).
end while

2.3 KNLMS In Hardware

In order to make the KNLMS algorithm more amenable to high-performance hardware implementations, the following modifications are made:

- a maximum dictionary size is chosen, $\tilde{N}$;
- the kernel vector is the same length, $\tilde{N}$, at each iteration - if the dictionary is not full, the unused entries are padded with zeros; and
- similarly, the weights remain the same length, $\tilde{N}$, at each iteration - since the kernel vector is padded with zeros, unused weights will remain zero.

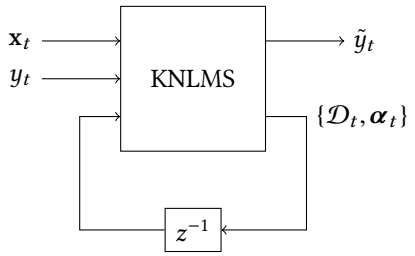

Fig. 1. An illustration of the bottleneck in KAFs.

If $\tilde{N}$ is larger than the number of entries that would be allowed into the dictionary due to the coherence criterion, then KNLMS will perform exactly the same with or without the above modifications. The restriction on the maximum dictionary size means we can ensure that $\mathcal{D}$ and $\boldsymbol{\alpha}$ can always fit in on-chip memory, avoiding potential bottlenecks reading and writing to off-chip memory. Furthermore, this places a restriction on the maximum number of operations that are required per training example, which allows us to design hardware to meet minimum throughput requirements of a given target application. The padding of the kernel and weight vectors to be of size $\tilde{N}$ mean that all vector sizes in every iteration are the same, regardless of the number of entries in $\mathcal{D}$. This allows us to simplify our datapath design for fixed vector sizes and reduce control logic overhead.

2.4 Pipelined Adaptive Filters

The recursive nature of adaptive filters means that they suffer from a computational bottleneck, as illustrated in Figure 1 for KNLMS. In order to start calculating the model at time n, we must know the model at time $n-1$. Typically, this means that our data rate is limited by the latency of the critical path of the architecture for a single update step.

For linear adaptive filters, such as LMS, this problem has been considered in great detail. Long et al. [25] introduced the first analysis of the LMS algorithm with delayed coefficient adaptation, i.e., the delayed least mean squares (DLMS) algorithm. Yi et al. [39] proposed variants of the DLMS algorithm together with implementation architectures. In particular, their transpose-form delayed least mean squares (TF-DLMS) implementation utilises a pipeline with different delay factors depending on the spatial location of filter weights. The architectures proposed by Yi et al. [39] implement delayed LMS, transpose-form LMS and some combinations in-between along with analysis of operation counts and expected propagation delay. Furthermore, Yi et al. [39] then systematically compares these algorithms and architectures and is able to achieve data rates up to 182 MS/s on a Virtex-II FPGA for a variant they propose called transpose-form retimed delayed least mean squares (TF-RDLMS). Douglas et al. [13] describe an architecture which utilises error correction terms, proposed by Poltmann [28], to create a pipelined LMS architecture which has the same behaviour as the standard LMS algorithm. The error correction terms allow the weights to be updated after some latency without diverging from the original algorithm.

To our knowledge, no prior work has studied the effects of model adaptation delay on KAFs. However, there are several works which propose architectures for executing various types of KAFs. Ren et al. [29] propose an architecture for the kernel LMS (KLMS) algorithm which is designed to consume a small amount of resources while using floating point arithmetic. A high degree of parallelism is achieved by replicating the design across an FPGA device. The design also utilised a very efficient kernel, the survival kernel [8], which can achieve high accuracy but is not as common as the Gaussian kernel used in this work.

Fraser et al. [18] describe a deeply pipelined processor generator which can produce dataflow implementations of the KNLMS algorithm. Architecturally, it is similar to this work. The authors provide resource and performance results for the architecture using both floating point and fixed point arithmetic. However, the architecture and implementation described by Fraser et al. [18] requires deep pipelines which would result in very high delay factors, if the hardware were repurposed for implementing one of the DKNLMS-based algorithms described in this work. Despite having vastly different architectures, the designs of both Ren et al. [29] and Fraser et al. [18] achieve parallelism across hyperparameter search and/or multi-channel systems but are not optimized for the single-channel, high-throughput case. The present work directly addresses this problem.

Pang et al. [27] describe a pipelined, microcoded vector processor. The processor is compact, power efficient and capable of achieving high performance on sliding window kernel RLS (SW-KRLS) [35], fixed budget kernel RLS (FB-KRLS) [34] and KNLMS [30]. The processor used floating point arithmetic, and is one of the few KAFs architectures capable handling multiple algorithm types.

Fox et al. [16] describe an architecture which implements the Fastfood algorithm [23] which approximates $\mathbf{k}_{n-1}(\mathbf{x}_n)$ using random projections. The matricies used to compute $\mathbf{k}_{n-1}(\mathbf{x}_n)$ can be defined in a way which constrains them to be Hadamard matricies, allowing the most computationally expensive part of the update step to be computed using the fast Walsh-Hadamard transform (FWHT). This reduces the computational complexity from $O(\tilde{N}M) \rightarrow O(\tilde{N}\log_2 M)$. A systolic structure of Hadamard blocks, with an internal butterfly structure to efficiently implement the FWHT was utilised for its implementation. The work targeted larger scale problems, which means they can achieve high processing element (PE) efficiency, but lower sample rates, compared to this work.

Tridgell et al. [32] describe the *braiding* technique to handle dependencies in high performance KAFs. To minimise latency, braiding calculates both branches of a conditional in parallel, and inserts the result of the appropriate branch into a reduction tree once the condition is known. The authors demonstrate the technique with a high throughput implementation of naive online regularised risk minimization algorithm (NORMA) [22], utilising fixed point arithmetic. Their provided post-place-and-route results demonstrate cores capable of achieving a throughput of 138 MS/s. The design suffers from only being able to work on sliding window algorithms. In contrast, this work can be applied to arbitrary dictionary storage schemes.

3 KERNEL NORMALISED LEAST MEAN SQUARES WITH DELAYED MODEL ADAPTATION

In this section, we provide a mathematical description of the DKNLMS algorithm. This has KNLMS-like behaviour, with dependencies shifted temporally into the future to facilitate pipelining. Motivations behind these choices are further elaborated upon in the Architecture section (Section 4).

The DKNLMS algorithm is conceptually similar to the delayed LMS algorithm [25]. The key idea is to introduce a delay factor, $d \in \mathbb{N}^+$, which removes most of the dependencies between iterations up to time d. At time n, given a new training pair, $\{\mathbf{x}_n, y_n\}$, and a delayed version of the model, given by $\mathcal{D}_{n-d}$ and $\tilde{\boldsymbol{\alpha}}_{n-d}$, we wish to calculate $\mathcal{D}_n$ and $\tilde{\boldsymbol{\alpha}}_n$. This is achieved by first, conceptually breaking the KNLMS algorithm up into two components:

(1) the *gradient* calculation; and
(2) the *model adaptation*.

The gradient calculation is the most computational intensive part of the KNLMS algorithm, whereas the model adaptation does not necessarily have long critical paths. Specifically, we consider the gradient calculation to involve: the calculation of $\mathbf{k}$, the coherence criterion, the a-priori prediction

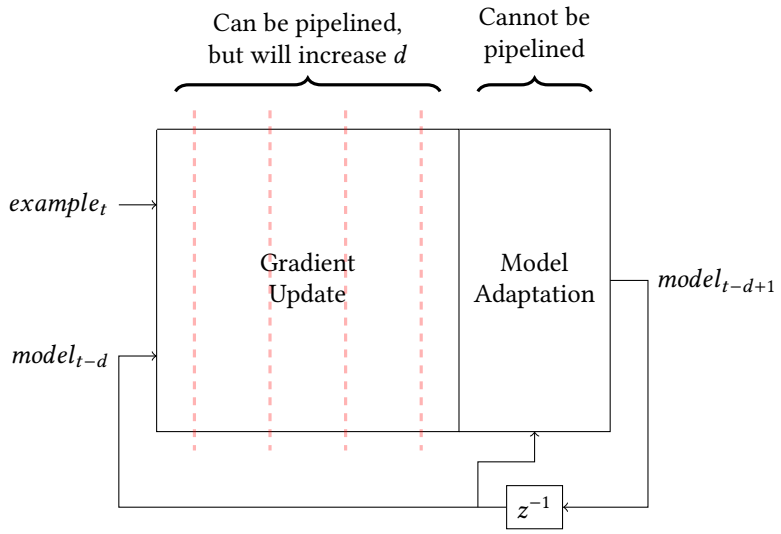

Fig. 2. Pipelining the gradient calculation increases d, but will increase the sample rate. Pipelining the model adaptation will increase the number of channels while decreasing the sample rate.

and the change in weights. The model adaptation is the storing of $\mathbf{x}_n$ into $\mathcal{D}_{n-d}$ (if required) and the accumulation of weights. If we design our dataflow so that that the algorithmic loop is contained to the model adaptation, then we can pipeline the gradient calculation arbitrarily, with each pipeline stage increasing d by 1. Figure 2, illustrates this mapping of the dataflow. Note that the recursive loop is coupled to the model adaptation, allowing the gradient calculation to be pipelined.

The kernel vector, $\mathbf{k}_{n-d}(\mathbf{x}_n)$, is calculated for the new input example. Similarly, the coherence between $\mathcal{D}_{n-d}$ and $\mathbf{x}_n$ is used to decide whether or not to add $\mathbf{x}_n$ to $\mathcal{D}_{n-1}$.

In order to update the weights, we need a prediction of y_n using the model at time $n-d$. This can be expressed as:

$$\hat{y}_n = \mathbf{k}_{n-d}(\mathbf{x}_n)^T \hat{\boldsymbol{\alpha}}_{n-d} \quad . \tag{10}$$

A partially updated kernel vector, $\tilde{\mathbf{k}}_n(\mathbf{x}_n)$, if $\mathbf{x}_n$ is added to dictionary is given by:

$$\tilde{\mathbf{k}}_n(\mathbf{x}_n) = \left[\mathbf{k}_{n-d}(\mathbf{x}_n)^T, \mathbf{0}^T, \kappa(\mathbf{x}_n, \mathbf{x}_n)\right]^T \quad , \tag{11}$$

where $\mathbf{0}$ is an $(\tilde{N}_{n-1} - \tilde{N}_{n-d})$ vector of zeros to align the latest dictionary entry with the corresponding entry in $\tilde{\boldsymbol{\alpha}}_n$. Alternatively, if $\mathbf{x}_n$ is not added then $\tilde{\mathbf{k}}_n(\mathbf{x}_n) = \left[\mathbf{k}_{n-d}(\mathbf{x}_n)^T, \mathbf{0}^T\right]^T$. Finally, $\tilde{\boldsymbol{\alpha}}_n$ is calculated as follows:

$$\tilde{\boldsymbol{\alpha}}_n = \tilde{\boldsymbol{\alpha}}_{n-1} + \eta \frac{y_n - \hat{y}_n}{\epsilon + \left\|\tilde{\mathbf{k}}_n(\mathbf{x}_n)\right\|_2^2} \tilde{\mathbf{k}}_n(\mathbf{x}_n) \quad , \tag{12}$$

Note that if $d = 1$, then Eq. (10), (11) and (12) are equivalent to KNLMS, i.e., Eqs. (1), (2) and (9). As such, DKNLMS can be considered as a generalisation to KNLMS to support $d > 1$.

The DKNLMS algorithm is described as psuedocode in Algorithm 2. Note that $\mathcal{D}_i = [\,]$, and $\tilde{\boldsymbol{\alpha}}_i = [\,]$ if $i \in \mathbb{Z}^-$ and a unit norm kernel is assumed for the coherence calculation.

3.1 Multiple Delays

When studying the dataflow in Algorithm 2 it is clear that the values for the dictionary delay, $d_{\mathcal{D}} \in \mathbb{N}^+$, and the weight delay, $d_{\boldsymbol{\alpha}} \in \mathbb{N}^+$, need not be the same. A version of DKNLMS with multiple

ALGORITHM 2: DKNLMS algorithm with coherence criterion.

Initialise the step-size, η, and the regularization factor, ϵ, and delay factor, d.
Define a kernel function, $\kappa(\cdot,\cdot)$.
Initialise $\mathcal{D}_0 = [\,]$, $\tilde{\boldsymbol{\alpha}}_0 = [\,]$, and $\tilde{N}_0 = 0$.
while $n > 0$ **do**
  Get $\{\mathbf{x}_n, y_n\}$.
  if $\tilde{N}_{n-d} == 0$ **then**
    $\mathbf{k}_{n-d}(\mathbf{x}_n) = [\,]$.
    $\mu = 0$.
  else
    Calculate $\mathbf{k}_{n-d}(\mathbf{x}_n) = [\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_1), \cdots, \kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_{\tilde{N}_{n-d}})]^T$.
    $\mu = \max(|\mathbf{k}_{n-d}(\mathbf{x}_n)|)$.
  end if
  if $\mu < \mu_0$ **then**
    $\tilde{N} = \tilde{N} + 1$.
    Append $\mathbf{x}_n$ to $\mathcal{D}_{n-1}$.
    Append 0 to $\tilde{\boldsymbol{\alpha}}_{n-1}$.
    $\tilde{\mathbf{k}}_n(\mathbf{x}_n) = [\mathbf{k}_{n-d}(\mathbf{x}_n)^T, \mathbf{0}^T, \kappa(\mathbf{x}_n, \mathbf{x}_n)]^T$.
  else
    $\tilde{\mathbf{k}}_n(\mathbf{x}_n) = \mathbf{k}_{n-d}(\mathbf{x}_n)$.
  end if
  Calculate $\tilde{\boldsymbol{\alpha}}_n$ using Eq. (12).
end while

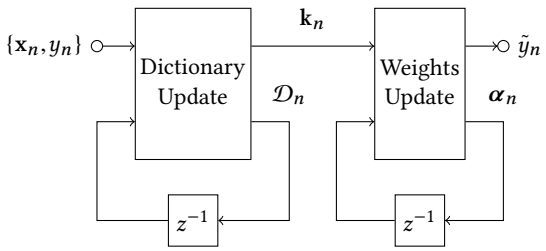

Fig. 3. A high level diagram of MDKNLMS.

delays, which is henceforth referred to as the MDKNLMS algorithm, can easily be implemented in hardware by splitting the DKNLMS into the two parts shown in Figure 3. This modification incurs no further computation cost, and actually reduces the number of registers required within the design, which may reduce the resource usage of the design. However, it is unclear as to whether this will have a positive or negative affect on the maximum clock frequency of an implementation, we discuss this further in Section 6. Comparing the DKNLMS and MDKNLMS algorithms, for the same number of pipeline stages, $N_p = d$, MDKNLMS will reduce the effective delay of either $d_{\mathcal{D}}$, $d_{\boldsymbol{\alpha}}$, or both. That is, for DKNLMS the delay factors are given by $N_p = d = d_{\mathcal{D}} = d_{\boldsymbol{\alpha}}$, while for MDKNLMS are $N_p = d = d_{\mathcal{D}} + d_{\boldsymbol{\alpha}}$. Furthermore, if a higher model adaptation delay introduces an instability in the algorithm, as can occur for delayed LMS [25], then MDKNLMS will be more stable than the equivalent DKNLMS with an equivalent delay.

3.2 Dictionary Guarding

An issue with applying delayed model adaptation to KNLMS is that the choice of whether or not to add an example to $\mathcal{D}_{n-1}$ is based on an outdated dictionary, $\mathcal{D}_{n-d_\mathcal{D}}$. This can result in a dictionary which contains *redundant* entries. For example, imagine that two consecutive examples are equal, i.e., $\mathbf{x}_n = \mathbf{x}_{n+1}$ (the outputs y_n, y_{n+1} can be ignored for now), and the coherence between $\mathbf{x}_n$ and all earlier examples are less than μ_0. Clearly, when training occurs on $\mathbf{x}_n$, the coherence between it and $\mathcal{D}_{n-d_\mathcal{D}}$ will be less than μ_0, and it will be added to the dictionary, $\mathcal{D}_{n-1}$, to create $\mathcal{D}_n$. When training occurs on $\mathbf{x}_{n+1}$, the coherence between $\mathbf{x}_{n+1}$ and $\mathcal{D}_{n-d_\mathcal{D}+1}$ will also be less than μ_0. As such, $\mathbf{x}_{n+1}$ is also added to the dictionary, creating $\mathcal{D}_{n+1}$. Given that the prediction function, Eq. (1), is based the weighted sum of kernel evaluations between the dictionary entries and the latest input, we can see that $\mathcal{D}_{n+1}$ provides no improved predictive power over $\mathcal{D}_n$, i.e., a prediction made using $\mathcal{D}_{n+1}$ could be rewritten as:

$$\sum_{i=1}^{\tilde{N}_{n+1}} \alpha_i \kappa(\mathbf{x}_j, \tilde{\mathbf{x}}_i) = \sum_{i=1}^{\tilde{N}} \alpha_i \kappa(\mathbf{x}_j, \tilde{\mathbf{x}}_i) + (\alpha_{\tilde{N}_n} + \alpha_{\tilde{N}_{n+1}}) \kappa(\mathbf{x}_j, \tilde{\mathbf{x}}_{\tilde{N}_n}) \ . \tag{13}$$

This means that $\mathcal{D}_{n+1}$ uses more memory and has a higher computational cost, for no improved modelling accuracy over $\mathcal{D}_n$.

To address this effect, we propose a method to prevent redundant entries from being added to the dictionary, which we call *dictionary guarding*. If $\mathbf{x}_n$ is added, based on the dictionary at time $n - d_\mathcal{D}$, it is not used to assist in making a decision of whether or not a new example should be added until $\mathbf{x}_{n+d_\mathcal{D}}$ arrives, i.e., we suggest, that the examples $\{\mathbf{x}_{n+1}, \cdots, \mathbf{x}_{n+d_\mathcal{D}-1}\}$ are prevented from being added to the dictionary.

Dictionary guarding prevents the dictionary from containing redundant entries and it can be implemented easily in hardware, using a simple counter, as described in Section 4. The possible downsides are:

- A decrease in modelling accuracy;
- An increase in convergence time; and
- A reduction in coherence between the dictionary and previously seen input examples, i.e., the coherence between $\mathcal{D}$ and all inputs is no longer guaranteed to be greater than μ_0.

How these issues relate to the resultant DKNLMS-DG algorithm are examined empirically in Section 6.

3.3 Correction Terms

If the accuracy of MDKNLMS is not satisfactory, we can modify it to exhibit equivalent learning behaviour to the original KNLMS algorithm. These modifications come at the cost of extra hardware, and potentially, a decrease in clock frequency. This section describes this modified version of MDKNLMS, DKNLMS-CT. These correction terms can be considered as extension to the work by Poltmann [28] to KAFs, a significant difference being that for KAFs the contents of the dictionary also need to be considered. These correction terms also share similarity to braiding [32], the difference being the correction terms described in this work can work for non-sliding window based KAFs.

Firstly, let us consider the dictionary update step. In KNLMS, the current example, $\mathbf{x}_n$, is added to the dictionary if $\mu < \mu_0$, where μ is calculated using Eq. (4). The same applies to MDKNLMS, however, the calculation of μ is now based on an older version of the dictionary, $\mathcal{D}_{n-d_\mathcal{D}}$, rather than $\mathcal{D}_{n-1}$. At each time step, either the current input vector is added to the dictionary, or it remains unchanged. Therefore, $\mathcal{D}_{n-d_\mathcal{D}}$ is a subset of $\mathcal{D}_{n-1}$. $\mathcal{D}_{n-1}$ can also potentially contain any of the vectors from $\mathbf{x}_{n-d_\mathcal{D}+1} \rightarrow \mathbf{x}_{n-1}$. Given this, an expression for the decision value, ζ_n, the decision

whether or not to add $\mathbf{x}_n$ to the dictionary, can be written as:

$$\zeta_n = ! \left[(\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_1) < \mu_0) \ \& \ \cdots \ \& \ (\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_{\tilde{N}_{n-1}}) < \mu_0) \right] \ , \tag{14}$$

where '&', '!' are the Boolean functions 'AND' and 'NOT' respectively. If ζ_n is true, then $\mathbf{x}_n$ is added to the dictionary. Note that unit norm kernel functions are assumed in Eq. (14). If only $\mathcal{D}_{n-d_\mathcal{D}}$ is available, Eq. (14) can be rewritten as:

$$\begin{aligned} \zeta_n = &![(\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_1) < \mu_0) \ \& \ \cdots \ \& \ (\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_{\tilde{N}_{n-d_\mathcal{D}}}) < \mu_0) \\ &\& \ (\kappa(\mathbf{x}_n, \mathbf{x}_{n-d_\mathcal{D}+1}) < \mu_0 + !\zeta_{n-d_\mathcal{D}+1}) \ \& \ \cdots \ \& \ (\kappa(\mathbf{x}_n, \mathbf{x}_{n-1}) < \mu_0 + !\zeta_{n-1})] \ . \end{aligned} \tag{15}$$

Note that in this equation, '+' denotes the 'OR' function. While these expressions are equivalent, Eq. (15) can be implemented easily using pipelined hardware and therefore allows tradeoffs between area usage and throughput to be made. Conceptually, Eq. (15) can be though of as checking the coherence between $\mathbf{x}_n$, all dictionary entries and all training examples which are in the pipeline ahead of the current entry. The final decision is then corrected if any proceeding training examples are added to the dictionary. This correction only takes place if the proceeding entry is also coherent with the current training example, $\mathbf{x}_n$.

Now that the dictionary, $\mathcal{D}_n$, and kernel vector, $\mathbf{k}_n(\mathbf{x}_n)$, found by DKNLMS-CT are equivalent to KNLMS, we need to apply correction terms so that the weights, $\hat{\boldsymbol{\alpha}}$, are the same as those found by KNLMS. Let us consider Eqs. (9) and (12). Since the kernel vectors have been corrected, the only differences between these equations are the a-priori predictions. Defining the weight update step, δ_n, for KNLMS to be:

$$\delta_n = \frac{\eta \left(y_n - \mathbf{k}_n(\mathbf{x}_n)^T \boldsymbol{\alpha}_n \right)}{\epsilon + \|\mathbf{k}_n(\mathbf{x}_n)\|_2^2} \ , \tag{16}$$

and similarly, $\tilde{\delta}_n$ to be:

$$\tilde{\delta}_n = \frac{\eta \left(y_n - \mathbf{k}_n(\mathbf{x}_n)^T \boldsymbol{\alpha}_{n-d_{\boldsymbol{\alpha}}} \right)}{\epsilon + \|\mathbf{k}_n(\mathbf{x}_n)\|_2^2} \ , \tag{17}$$

for MDKNLMS with a corrected kernel vector. To create the DKNLMS-CT algorithm, we must find a way to modify Eq. (17) to become Eq. (16). In order to realise a benefit from pipelining, these modifications to Eq. (17) should only depend on data available at sample $n - d_{\boldsymbol{\alpha}}$. Given this requirement, an expression for δ_n can be formed using Eqs. (9), (16) and (17):

$$\begin{aligned} \delta_n &= \frac{\eta}{\epsilon + \|\mathbf{k}_n(\mathbf{x}_n)\|_2^2} \left(y_n - \mathbf{k}_n(\mathbf{x}_n)^T [\boldsymbol{\alpha}_{n-d_{\boldsymbol{\alpha}}} + \sum_{i=1}^{d_{\boldsymbol{\alpha}}-1} \delta_{n-i} \mathbf{k}_{n-i}(\mathbf{x}_{n-i})] \right) \\ \delta_n &= \frac{\eta}{\epsilon + \|\mathbf{k}_n(\mathbf{x}_n)\|_2^2} \left(y_n - \mathbf{k}_n(\mathbf{x}_n)^T \boldsymbol{\alpha}_{n-d_{\boldsymbol{\alpha}}} - \sum_{i=1}^{d_{\boldsymbol{\alpha}}-1} \delta_{n-i} \mathbf{k}_n(\mathbf{x}_n)^T \mathbf{k}_{n-i}(\mathbf{x}_{n-i}) \right) \ . \end{aligned} \tag{18}$$

Using this expression for δ_n, we can rewrite Eq. (9) as:

$$\hat{\boldsymbol{\alpha}}_n = \hat{\boldsymbol{\alpha}}_{n-1} + \delta_n \mathbf{k}_n(\mathbf{x}_n) \ . \tag{19}$$

Clearly, if $\hat{\boldsymbol{\alpha}}_n$ is updated using Eq. (18) rather than Eq. (9), then more operations are required. However, as we will see in Section 6, this way of computing the weights is preferable since we can still receive the benefits of pipelining the architecture. This is due to the fact that: (1) the vectors, $\{\mathbf{k}_{n-1}(\mathbf{x}_{n-1}), \cdots, \mathbf{k}_{n-d_{\boldsymbol{\alpha}}+1}(\mathbf{x}_{n-d_{\boldsymbol{\alpha}}+1})\}$, are available at time n; and (2) the only dependencies within the pipeline are scalar values, $\{\delta_{n-1}, \cdots, \delta_{n-d_{\boldsymbol{\alpha}}+1}\}$, which can be easily fed back where appropriate with only a small overhead.

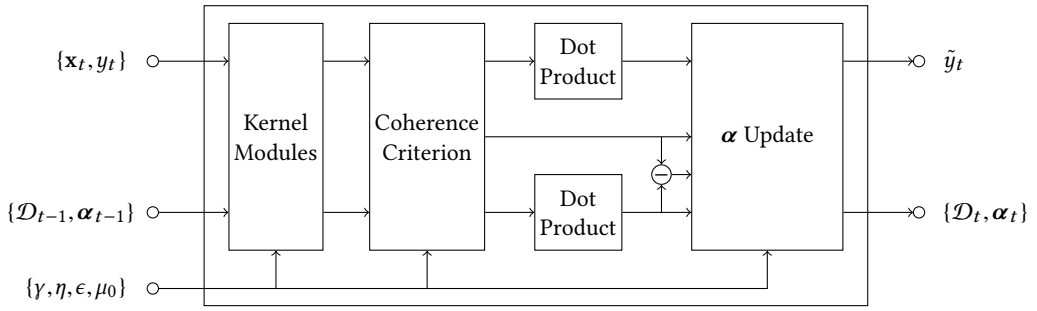

Fig. 4. High level view of the proposed architecture.

The correction terms also do not need to be added completely, a subset of the correction terms in Eq. (18) may be used instead. This can allow for further trade-offs to be made between area, accuracy and throughput.

Note that due to space restrictions, we only analyse the theoretical costs and benefits of DKNLMS-CT in the subsequent sections. As we will see in Section 6, the other DKNLMS variants achieve high accuracy and throughput and as such, empirically we deem that the addition of correction terms is not a requirement. However, this may change for future KAFs algorithms and datasets, so this theoretical analysis could be useful for future works.

4 ARCHITECTURE

In this section, the architecture of the DKNLMS algorithm is described in detail. The variants MDKNLMS and DKNLMS-DG are simple extensions to DKNLMS, as such, we described the modifications where they should occur. The architecture for DKNLMS-CT is not described, rather we theoretically analyse the cost of implementing it. Finally, in this section we omit the subscripts denoting time to reduce the notation complexity.

4.1 High Level View

In order to understand the architecture at a high level, firstly consider the *forward path* of the KNLMS algorithm. The forward path consists of the operations required for the KNLMS algorithm to update its model (i.e., the contents of the while loop in Algorithm 1), given a new training example, $\{\mathbf{x}_n, y_n\}$. Since the forward path contains no loops, it can be implemented using combinatorial logic, which is later converted to implement DKNLMS. The design method can be summarised as follows:

(1) begin with a combinatorial KNLMS design, created from a dataflow graph of the KNLMS forward path with a maximum dictionary size of $\tilde{N}$;
(2) add registers (or memories) to store the dictionary and weights internally to the design (i.e., add the loops with the smallest possible critical path to Eq. (12) and to the dictionary update step); and
(3) excluding the loops described in (2), pipeline the rest of the dataflow graph until a desired d value (or clock frequency) is achieved.

A block diagram of the basic DKNLMS implementation is shown in Figure 4. The design can be thought of as the combination of four different submodules: kernel modules, the coherence criterion module, dot product modules and the $\boldsymbol{\alpha}$-update module. Note that the coherence criterion

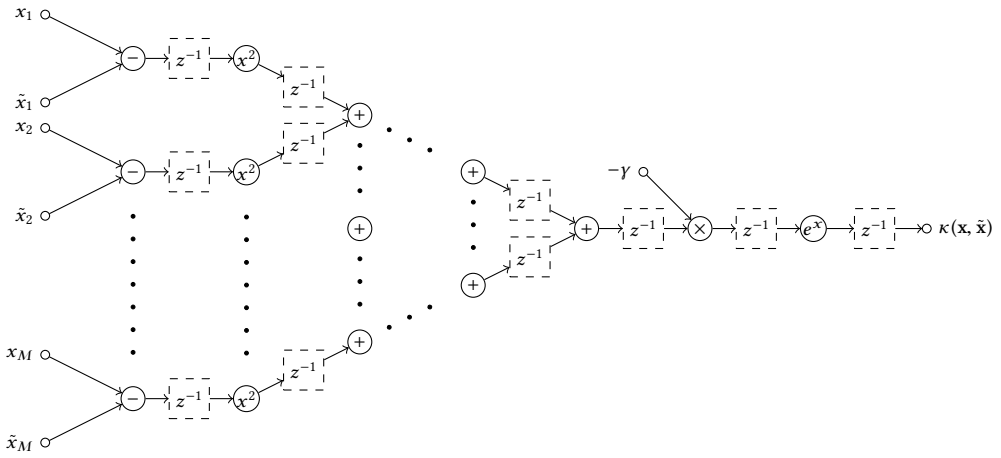

Fig. 5. Exponential kernel evaluation module.

module and the α update module contain loops and registers (or memories) to store and update the dictionary ($\mathcal{D}$) and weights ($\boldsymbol{\alpha}$) respectively. Throughout the design, 'optional' registers are placed between each arithmetic operation. Moreover, all arithmetic operations can also be pipelined while only affecting the delay parameters of the DKNLMS variants, with the exclusion of the accumulation on $\boldsymbol{\alpha}$.

4.2 The Submodules

In this subsection, the submodules of the design (as shown in Figure 4) are explained in detail. The kernel modules form the largest portion of the architecture and are designed to calculate the kernel vector, $\mathbf{k}$, given in Eq. (2). The i^{th} element of the kernel vector is evaluated using the Gaussian kernel, given by: $\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_i) = \exp(\|\mathbf{x}_n - \tilde{\mathbf{x}}_i\|_2^2)$. Although there are several useful kernel functions described in the literature, including several which are hardware friendly (notably the Laplacian kernel, used by Anguita et al. [1]), we chose to use the Gaussian kernel because: (1) it's a unit norm kernel; (2) it's a universal approximator [20]; and (3) it's a very popular kernel function, known to many familiar with kernel methods. In order to calculate $\mathbf{k}$, $\tilde{N}$ kernel modules are used, and Figure 5 illustrates the data path of a kernel module. Each kernel module takes the current input sample and an element of the dictionary as inputs, and produces the Gaussian kernel evaluation as an output. An adder tree is used to reduce latency. Note that we use z^{-1} with a broken line border to denote the optional registers. In order to use another unit norm kernel function, this module can simply be replaced with a module implementing your kernel function of choice. If a non-unit norm kernel is desired, the coherence module would also need to be modified to implement the coherence criterion described by Richard et al. [30].

The dot product module is given in Figure 6. It features elementwise multiplication followed by an adder tree. The dot product modules are used to calculate the a-priori prediction and $\|\mathbf{k}\|_2^2$, which is used the find the normalisation factor. For both the kernel and dot-product modules, an adder tree was preferred to perform accumulation, as we wish to minimise the overall latency.

The coherence criterion module calculates the coherence between the current input and the dictionary. The coherence (given in Eq. (4)) is then used to determine whether or not to add the current input to the dictionary. Figure 7 shows the architecture of the coherence criterion module. It features comparison modules, connected to a binary 'OR' tree, which produces the

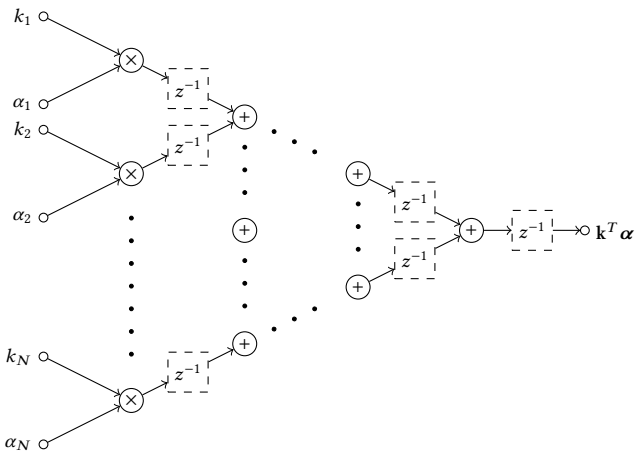

Fig. 6. The dot product block was made using a simple binary tree structure.

'AddToDict' signal. This signal feeds a multiplexer which determine whether or not the current example gets added to the dictionary, and subsequently, whether or not the counter is incremented. The dictionary can be stored in registers or memories, but the current design requires that all dictionary entries must be read every cycle. This module also pads the kernel vector, $\mathbf{k}$, with zero entries for unused dictionary entries, and appends '1' to $\mathbf{k}$ if the current input is added. In order to realise DKNLMS-DG, an extra counter must be included which entry corresponds to the number of elements seen since the last entry was added to the dictionary. If this value is below $d_{\mathcal{D}}$, then 'AddToDict' is set to false.

Figure 8 shows the $\boldsymbol{\alpha}$ update module. This module takes the prediction error, e_n, the normalisation factor, $\|\mathbf{k}\|_2^2$, and the kernel vector, $\mathbf{k}$, as inputs. The output is the updated weights, which are also stored internally in registers. The $\boldsymbol{\alpha}$ update module calculates the weight update using Eq. (9).

Both the coherence and $\boldsymbol{\alpha}$ modules feature valid and reset signals. The valid signal prevents unwanted entries from being added to the dictionary and from modifying the weights. The reset signal restores the counter, dictionary and weights to zero values.

4.3 Resource Usage and Latency

In this subsection, the resource usage and propagation delay of the proposed architecture are given as a function of: (1) the maximum dictionary size, $\tilde{N}$; (2) the feature length, M; and (3) the operators, exp, $\times$, $+$, $/$ and $<$. Since the design is fully pipelined, modules do not share resources. Also, as shown in Figure 4, the four difference modules are connected in series. Therefore, the total latency of the design will be the sum of the latency of each of the four modules described above, plus a subtraction. The total resource usage of the design will be sum of all resources of all modules used within the architecture, plus a subtraction.

By careful inspection of Figures 4, 5, 6, 7 and 8 the total resource usage can be calculated, remembering that $\tilde{N}$ kernel modules, two dot product modules, one coherence module and one α update module are used. Table 1 shows the resource usage for each module and the total for the whole design. Since $\tilde{N}$ kernel modules are required, the kernel modules will dominate for large values of $\tilde{N}$ and M. The worst case scalability of the resource usage of the complete design is $O(\tilde{N}M)$. The extra operations required to implement DKNLMS-CT are also shown in Table 1. Note that the number of correction terms are proportional to the delay factors, $d_{\mathcal{D}} - 1$ and $d_{\boldsymbol{\alpha}} - 1$

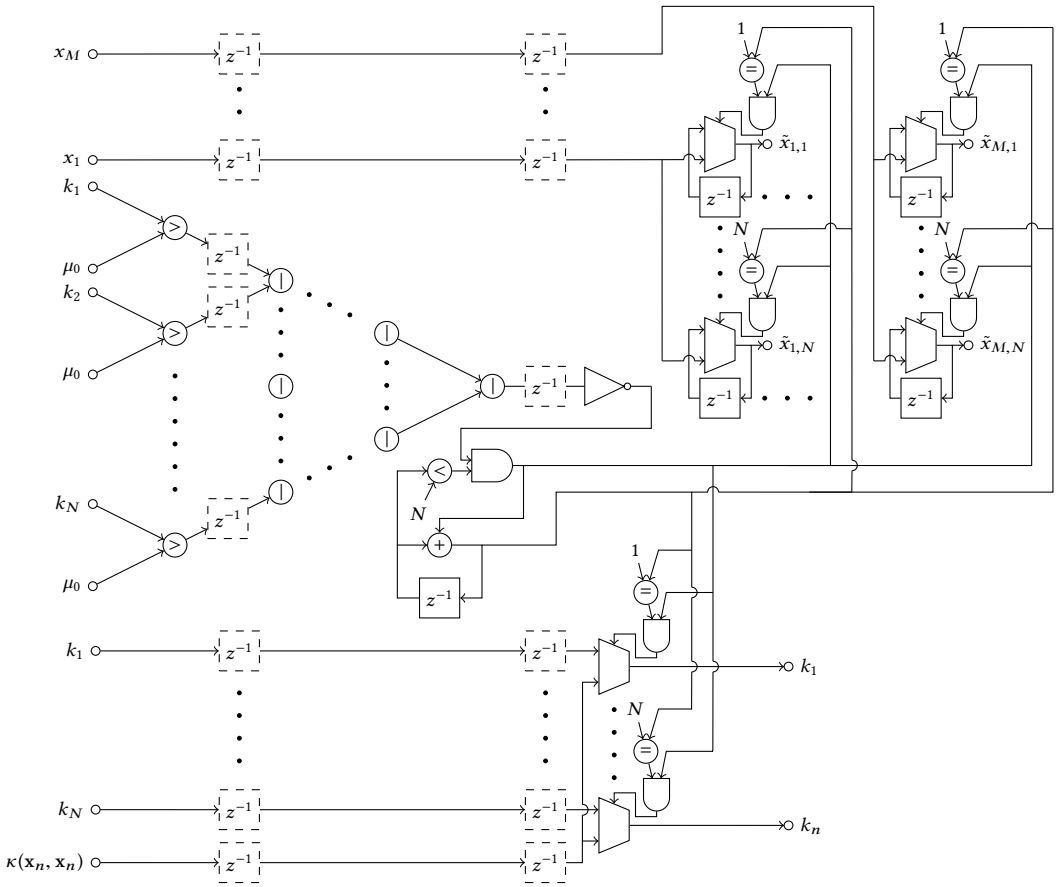

Fig. 7. Coherence criterion module.

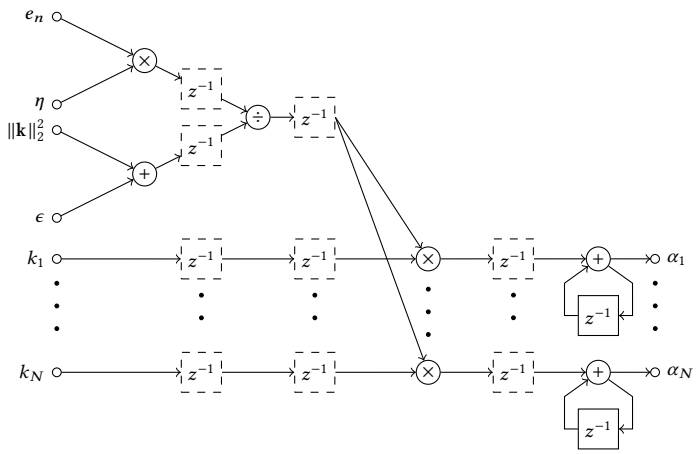

Fig. 8. Incremental update of alpha.

Table 1. Resource usage of each module and the complete design

Module	Num. of Modules	exp	$\times$	$+$	$\div$	$<$
Kernel	$\tilde{N}$	1	$M+1$	$2M-1$	0	0
Dot Product	2	0	$\tilde{N}$	$\tilde{N}-1$	0	0
Coherence Criterion	1	0	0	0	0	$\tilde{N}$
α Update	1	0	$\tilde{N}+1$	$\tilde{N}+1$	1	0
Design Total	-	$\tilde{N}$	$(M+4)\tilde{N}+1$	$2(M+1)\tilde{N}$	1	$\tilde{N}$
Correction Terms ($d_{\mathcal{D}}$)	$d_{\mathcal{D}}-1$	1	$M+1$	$2M-1$	0	1
Correction Terms ($d_{\boldsymbol{\alpha}}$)	$d_{\boldsymbol{\alpha}}-1$	0	$\tilde{N}+1$	$\tilde{N}$	0	0

Table 2. Expected propagation delay of each module and the complete design

Module	exp	$\times$	$+$	$\div$	$<$
Kernel	1	2	$\lceil\log_2(M)\rceil+1$	0	0
Dot Product	0	1	$\lceil\log_2(\tilde{N})\rceil$	0	0
Coherence Criterion	0	0	0	0	1
α Update	0	2	2	1	0
Design Total	1	5	$\lceil\log_2(\tilde{N})\rceil+\lceil\log_2(M)\rceil+4$	1	1
Correction Terms ($d_{\mathcal{D}}$)	1	2	$\lceil\log_2(M)\rceil+1$	0	1
Correction Terms ($d_{\boldsymbol{\alpha}}$)	0	2	$\lceil\log_2(\tilde{N})\rceil+\lceil\log_2(d_{\boldsymbol{\alpha}})\rceil$	0	0

respectively. The kernel vector evaluation scales the worst, $O(\tilde{N}M)$, while the number of correction terms will also incur significant overhead if $d_{\mathcal{D}}-1$ and $d_{\boldsymbol{\alpha}}-1$ are large.

The expected propagation delay of each module and the whole design are given in Table 2. The propagation delay is expressed as a sum of arithmetic operations along the critical path. The overall expected propagation delay is the sum of the expected propagation delay of each arithmetic operation which lie on this critical path. Using this basic approximation of propagation delay, we can decide which optional registers to enable in the design, shown in Figures 4, 5, 6, 7 and 8, or in the arithmetic operations themselves. Furthermore, α update and coherence criterion modules appear to have constant latency, while the kernel and dot product modules scale as $\log_2(M)$ and $\log_2(\tilde{N})$ respectively. As such, the worse case scalability of the expected propagation delay is $O(\log_2(M)+\log_2(\tilde{N}))$. In practice, larger designs will also likely suffer from placement and routing issues which will also affect the propagation delay. The expected propagation delay of the correction terms are also shown in Table 2. For the most part, the correction terms are computed in parallel with other modules, so the overall propagation delay does not change with the inclusion of correction terms. The exclusion for this, is the $\lceil\log_2(d_{\boldsymbol{\alpha}})\rceil$ which increases the overall expected propagation delay.

5 IMPLEMENTATION

In this section, several details / methods related to the implementation of the different DKNLMS configurations are specified. Chisel v2.2.27 [3] was used to create a core generator capable of generating all the architectures described in Section 4. Chisel is a hardware construction language

embedded in Scala which generated Verilog. It's a low abstraction domain specific language (DSL) which benefits from meta-programming in Scala to allow core generators to be made easily. Being a low abstraction DSL, Chisel allows complete control over the datapath while allowing the design to be scalable. As such, a design made in Chisel is comparable with an RTL design, but with increased productivity. All bitstreams and implementations in this work were generated using Vivado 2017.4 with aggressive optimizations enabled. Fixed point arithmetic was used throughout the design, unless specified otherwise, a wordlength of 18bits was used, although the design is parameterised for any fixed point format. The integer length was chosen to avoid overflows on the training dataset and the specific algorithm hyperparameters. In order to minimise the overhead of arithmetic operations, all fixed point arithmetic was implemented with a custom fixed point library in which the input and output every arithmetic operation was the same fixed point format. When necessary, conversion from higher precision to lower precision formats, e.g., after a multiplication, was achieved using simple truncation. Furthermore, no saturation logic was implemented to avoid overflows. 18bit wordlengths were chosen as they map well to DSP blocks within FPGAs, while providing little degradation in the accuracy of the algorithm, this is explored further in Section 6. A further reduction in bitwidth may be achieved with a finer grained approach to bitwidth and integer length selection, but this is not explored in this work. The exponential evaluation and division operations can consume a significant amount of hardware, and cause a significant increase in the overall expected propagation delay. As such, signification attention was paid to the implementation of these operations.

5.1 Exponential and Division Approximation

The exponential and division operators represent areas where significant performance improvements can be made to the DKNLMS designs. Some specific improvements made to this design were based on the following observations:

- the range of the inputs is significantly less than the range of the fixed point datatype;
- both functions are differentiable;
- division can be implemented as an inversion and a multiplication; and
- the accuracy of DKNLMS is not very sensitive to approximations in both functions.

Given the above, several different implementations were considered, including:

(1) a simple lookup table;
(2) a lookup table with linear interpolation; and
(3) the Remez algorithm [31] implemented using Estrin's polynomial evaluation method [15].

The exponential function was implemented as a simple single input function approximation, while division was implemented a reciprocal function, followed by a multiplication. Although Remez and linear interpolation provide many benefits, the rest of the design already used a significant amount of DSP resources and as such, a simple lookup table was used for both reciprocal and exponential functions with 1024 elements, we found this sufficient to provide good accuracy. We further discuss the implications on accuracy in Section 6.1.3.

5.2 Register Parameterisation

As the number of pipeline stages directly affects d, the more pipeline stages are added the more a DKNLMS variant will behave differently to KNLMS. As such, for small values of d the *placement* of the registers becomes vitally important. As stated in Section 4, the design was created with optional registers placed between each operation. When a particular core is generated, an array of Boolean values control whether or not a particular set of registers (i.e., a pipeline stage) is actually used in the design. This directly controls the delay parameters of DKNLMS and the variants, i.e., d, $d_{\mathcal{D}}$ and

d_α. Also, when operations themselves are pipelined, shift registers are created to ensure that other signals (e.g., the valid signal) arrive at subsequent locations at the correct times.

In order to generate the array of Boolean values, we break the design up into approximate unit delays. Let us denote a unit delay with τ. For the given 18bit fixed point implementation, the estimated propagation delay of several operations is as follows:

- Multiply - τ;
- Addition/Subtraction - τ;
- Exponentiation - 3τ; and
- Division - 4τ.

The above delay values were approximated by synthesising and place & routing the individual arithmetic operators on the target FPGA device. The pipeline stages are then placed so they have approximately equal delay values separating them. For example, using Table 2, one can determine the expected propagation delay of each submodule. This estimate can then be used to decide which pipeline stage to enable to best improve the performance of the design. This is done by enabling the pipeline register which produces paths with the smallest possible estimated propagation delay.

6 RESULTS

In this section, we compare the accuracy and performance of all the KNLMS-like algorithms discussed within: KNLMS, DKNLMS, MDKNLMS, DKNLMS-DG and DKNLMS-CT. For accuracy, we study the effect of delayed model adaptation on the modelling capability of the KNLMS. Issues with convergence, stability and model size when delayed model adaptation is introduced are also explored. In terms of performance, the throughput of the architectures described in Section 4 are provided for varying dictionary size ($\tilde{N}$), feature length (M) and latency (d). These performance figures are compared to a C implementation of KNLMS running on a PC.

6.1 Accuracy

In this subsection, the accuracy the DKNLMS algorithms are analysed empirically. In the first part of this section, we study the accuracy, convergence and stability of all DKNLMS variants. Firstly, we consider the DKNLMS variants using floating point arithmetic and without any of the function approximations described in Section 5. The extra effect of using fixed point arithmetic and function approximations is then considered in Section 6.1.3. For the accuracy experiments in this paper, the chaotic Mackey-Glass [26] time series was used. The Mackey-Glass time series benchmark is generated using the following differential equation: $dx(t)/dt = -ax(t) + bx(t-\tau)/(1+x(t-\tau)^{10})$ with $(a = 0.1, b = 0.2, \tau = 30)$, this configuration is identical to the benchmark used by Engel et al. [14]. Our implementation was based off the KAF toolbox (KAFBOX) [33].

6.1.1 Hyperparameter Optimisation. In practice, we suggest optimising the parameters for the DKNLMS variants independently of each other, allowing tradeoffs for convergence time and stability to be captured within the cross validation process. In order to adequately compare the different algorithms, hyperparameter optimisation must be performed for each algorithm. Otherwise, we may accidentally compare one algorithm at optimal settings to another at sub-optimal settings. The problem of hyperparameter optimisation is non-convex and therefore, we cannot be certain of achieving optimal settings, but a simple random search [6] will help us avoid extremely sub-optimal settings. We can also quantify our effort in finding good hyperparameters in terms of the number of samples required to achieve good accuracy. In this work, we use the following hyperparameter optimization procedure:

Table 3. Cross-validation accuracy and hyperparameters for each DKNLMS variant

Algorithm	d	γ	η	ϵ	μ_0	Optimised Average $\tilde{N}$	Optimised Average MSE ($\times 10^{-2}$)	Baseline Average $\tilde{N}$	Baseline Average MSE ($\times 10^{-2}$)
KNLMS	-	1.48	0.15	0.050	0.69	19.9	1.37	87.3	1.49
DKNLMS	4	1.48	0.15	0.050	0.69	25.7	1.35	88.2	1.44
	8	1.48	0.15	0.050	0.69	26.1	1.31	88.7	1.37
	16	0.88	0.068	0.090	0.75	23.6	1.58	88.3	1.61
	32	2.14	0.036	0.061	0.77	66.9	1.60	103.4	3.16×10^{5}
MDKNLMS	4	1.23	0.13	0.063	0.87	54.1	1.53	88.2	1.54
	8	1.48	0.15	0.050	0.69	25.7	1.41	88.2	1.44
	16	1.48	0.15	0.050	0.69	26.1	1.32	88.7	1.39
	32	0.89	0.063	0.089	0.85	32.9	1.63	88.3	1.70
DKNLMS-DG	4	1.36	0.27	2.55	0.94	128	1.28	90.5	2.07
	8	2.21	0.38	3.06	0.70	36.7	1.21	80.3	4.23
	16	1.92	0.49	3.72	0.42	9.1	1.49	66.7	6.96
	32	3.24	0.72	3.89	0.28	10	2.28	48.3	4.60
DKNLMS-CT	4	1.48	0.15	0.05	0.69	19.9	1.37	87.3	1.49
	8	1.48	0.15	0.05	0.69	19.9	1.37	87.3	1.49
	16	1.48	0.15	0.05	0.69	19.9	1.38	87.3	1.50
	32	1.48	0.15	0.05	0.69	19.9	1.40	86.9	1.53

(1) 10-fold cross validation was used, separating the training set into ten different training and validation sets;
(2) for each training example, the resultant model was tested, using mean squared error (MSE), on the entire validation set to produce a model convergence series;
(3) after removing an initial number of examples during the convergence period, the remaining entries in the model convergence series are averaged; and
(4) the model with the lowest average MSE in the convergence period, was assumed the be the best hyperparameters.

For the examples in Section 6.1.2 the training set size was 1000 and the convergence period was 500 samples. When there was negligible accuracy difference, sometimes the 2^{nd} best hyperparameters where used if they provided: 1) a more compact model; or 2) a more stable convergence series. Table 3, shows the optimized hyperparameters for each DKNLMS configuration, along with average model size, $\tilde{N}$, and average MSE over the cross validation procedure. The baseline average MSE in the final columns, which uses some common values for each of the hyperparameters of KNLMS without specific optimization for each algorithm. Specifically, the values of the hyperparameters for the baseline were as follows: $\gamma = 1.39$, $\eta = 0.1$, $\epsilon = 10^{-4}$ and $\mu_0 = 0.9$. Interestingly, when the baseline results are compared, we get very similar performance between KNLMS, DKNLMS ($d \leq 8$) and MDKNLMS ($d \leq 16$), while DKNLMS-DG suffers significant accuracy degradation for all values of d. As expected, all algorithms for all values of d perform better when their hyperparameters are optimised specifically for them. By comparing the baseline and optimised results for DKNLMS-DG it is clear this algorithm behaves very different to the original KNLMS algorithm. Through hyperparameter optimisation, the accuracy of the DKNLMS-DG variants are significantly improved to the point where it appears to perform better than the other variants for $d = 4$ and $d = 8$. However, it should be noted that since the behaviour of DKNLMS-DG was significantly different

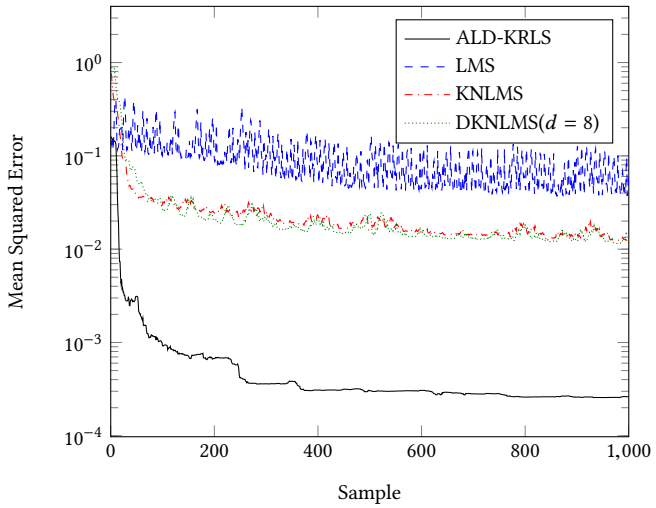

Fig. 9. Comparison of prediction performance between DKNLMS ($d = 8$) and other adaptive filtering algorithms.

to the others, more time was spent optimising the hyperparameters in order to find acceptable values. The search space was broader and roughly 20× points were sampled, compared to the other configurations. If the other algorithms were allowed a larger search space, this difference may be less pronounced. Lastly, the baseline DKNLMS ($d = 32$) appears to have unstable behaviour, but, after hyperparameter optimisation it appears to have achieved stability with a some degradation in accuracy compared KNLMS. As expected, DKNLMS-CT performed almost identically to KNLMS.

6.1.2 Accuracy Comparison. The cross validation accuracy is useful for hyperparameter optimisation, but the real test is the accuracy on the test set. Furthermore, when dealing with time series data, it may be necessary to plot the convergence of a learning algorithm to better understand its behaviour. In this subsection, we plot the convergence of each algorithm, firstly with the baseline hyperparameters, secondly, with optimised hyperparameters and compare the differences.

However, before we look closely at the proposed algorithms, let us first put some of the results in context. Figure 9 shows the accuracy of DKNLMS ($d = 8$) versus KNLMS and some other common adaptive filtering algorithms in the literature. Firstly, we can see that approximate linear dependency kernel RLS (ALD-KRLS) [14] outperforms KNLMS by almost two orders of magnitude. However, the increased complexity of the dependencies of ALD-KRLS mean that it's not the subject of our study for now. The purpose of the figure is to provide context around different types of adaptive filters and their costs and benefits. ALD-KRLS scales with $O(\tilde{N}^2 + \tilde{N}M)$ in operations and memory, as such it would be a much more difficult task to implement at sample rates in the order of 100s of MHz as we do in this work. Secondly, we see that KNLMS outperforms LMS [38] by almost an order of magnitude. Currently, if high frequency (in the order of 100s of MHz) adaptive filtering is required, variants of LMS are one of the few available options.

Figures 10 & 11 show the convergence pattern of DKNLMS while varying the delay parameters. Figure 10 uses the baseline hyperparameters, while 11 uses the optimised hyperparameters for each configuration. Figure 10 shows that for DKNLMS ($d \leq 8$), the accuracy and convergence are very similar to KNLMS. For $d = 16$, some instability begins to show, while for $d = 32$ DKNLMS becomes completely unstable. In Figure 11 we see the stability for $d = 32$ significantly improve,

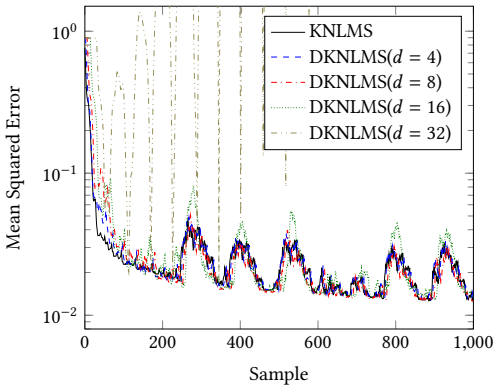

Fig. 10. KNLMS vs DKNLMS using the same parameters for each configuration.

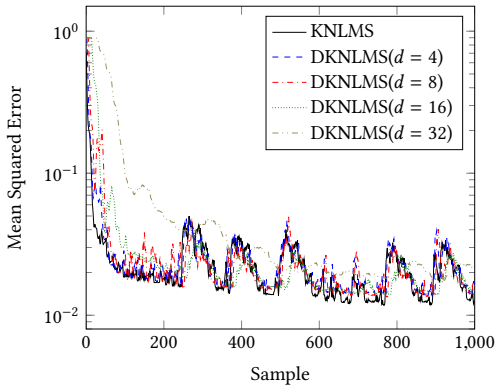

Fig. 11. KNLMS vs DKNLMS with parameters adjusted for each configuration.

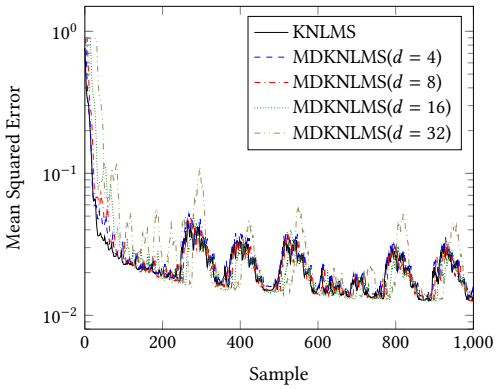

Fig. 12. KNLMS vs MDKNLMS using the same parameters for each configuration.

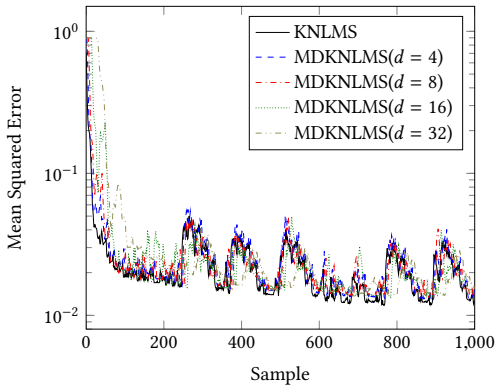

Fig. 13. KNLMS vs MDKNLMS with parameters adjusted for each configuration.

at the cost of convergence speed. For $d = 16$ with optimised hyperparameters, $d = 16$ seems to perform effectively as good as KNLMS for this benchmark.

Moving on to the second variant, MDKNLMS, Figures 12 & 13 show the convergence pattern of MDKNLMS while varying the delay parameters. Again, Figure 12 uses the baseline hyperparameters, while 13 uses the optimised hyperparameters for each configuration. Figure 12 shows similar convergence for all configurations of MDKNLMS ($d \leq 16$), while some instability begins to show when $d = 32$. After hyperparameter optimisation, all configurations of MDKNLMS perform well, with $d = 32$ only show a slight deterioration of convergence speed.

Now for DKNLMS-DG, Figures 14 & 15 show the convergence pattern of DKNLMS-DG while varying the delay parameters. Similar to the previous examples, Figure 14 uses the baseline hyperparameters, while Figure 15 uses the optimised hyperparameters for each configuration. In Figure 14, we see a significant difference between the accuracy of KNLMS and DKNLMS-DG for all factors of d. In terms of accuracy, all configurations are visibly less accurate and have slower convergence than KNLMS. For $d \geq 16$, the DKNLMS-DG also show visible instability in the convergence region, without exhibiting the extremely unstable behaviour of DKNLMS ($d = 32$). After hyperparameter

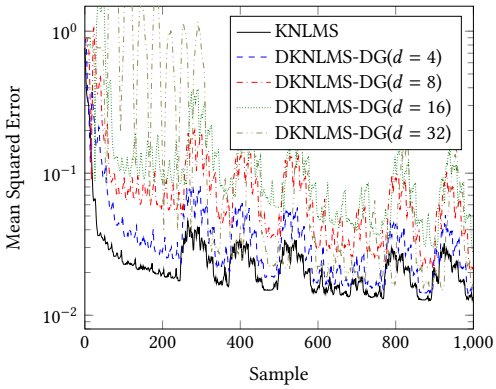

Fig. 14. KNLMS vs DKNLMS-DG using the same parameters for each configuration.

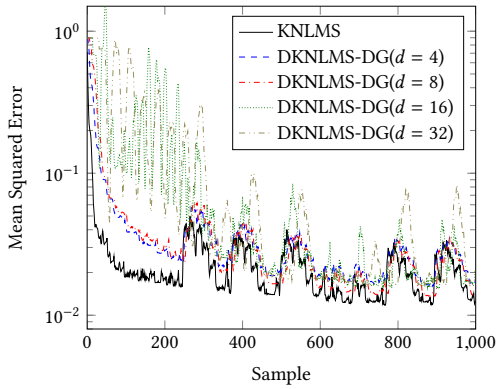

Fig. 15. KNLMS vs DKNLMS-DG with parameters adjusted for each configuration.

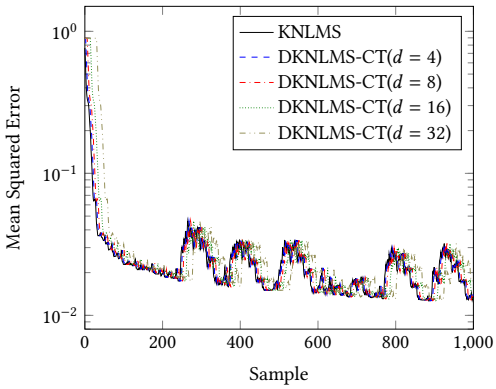

Fig. 16. KNLMS vs DKNLMS-CT using the same parameters for each configuration.

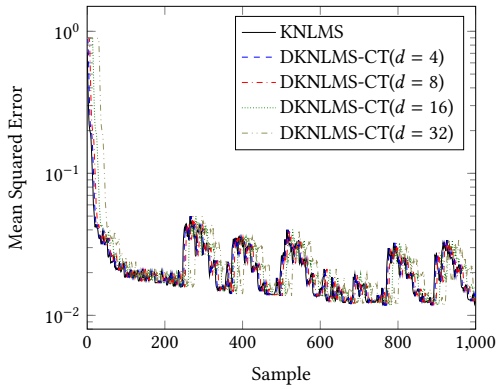

Fig. 17. KNLMS vs DKNLMS-CT with parameters adjusted for each configuration.

optimisation, shown in Figure 15, all configurations of DKNLMS-DG show significantly better accuracy, at the expense of significantly worse convergence speeds. Furthermore, even after hyperparameter optimisation, instability can be seen when $d \geq 16$. We suspect this instability may be caused by the fractional part of Eq. (9) tending towards zero, which can occur for DKNLMS-DG configurations. We also note that the ϵ value that was found for DKNLMS-DG configurations was significantly higher than the other DKNLMS variants, which may be to counteract this effect. Overall, the accuracy results of the DKNLMS-DG configurations were underwhelming, and as such will often not be used for comparisons in the subsequent sections.

Finally, Figures 16 & 17 show the convergence pattern of DKNLMS-CT while varying the delay parameters. Again, similar to previous examples, Figure 16 uses the baseline hyperparameters, while Figure 17 uses the optimised hyperparameters for each configuration. As expected, DKNLMS-CT shows the exact same convergence pattern as KNLMS itself, just shifted by the delay amount d. While from an accuracy point of view, DKNLMS-CT is promising, one must remember that this comes at the cost of increased hardware resources and increased critical path delay. In practical terms, this means reduced dictionary size and higher latency. Consider the results shown in Table 3

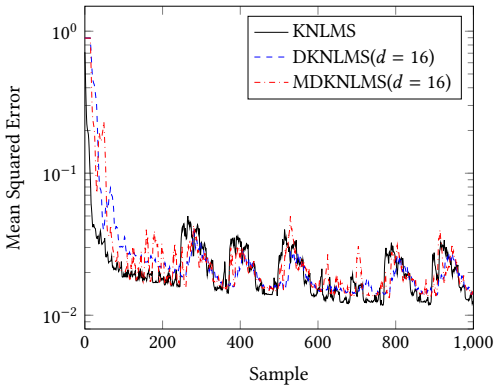

Fig. 18. DKNLMS and MDKNLMS when $d = 16$.

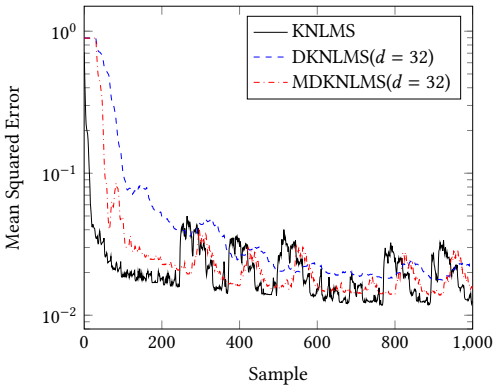

Fig. 19. DKNLMS and MDKNLMS when $d = 32$.

for MDKNLMS and DKNLMS-CT when $d = 16$. Rounding the values, MDKNLMS / DKNLMS-CT would require dictionary sizes of 20 / 26 respectively. Using the feature length of Mackey-Glass of $M = 7$, and the arithmetic counts in Table 1, the number of operations per update can be calculated for MDKNLMS / DKNLMS-CT as 756 / 1023 respectively, assuming $d_{\mathcal{D}} = d_{\boldsymbol{\alpha}} = 8$. In all, that would mean that for this example DKNLMS-CT would require a 35% increase in Ops for no accuracy benefit over MDKNLMS. Although this may be useful in some circumstances, in this benchmark we do not see a significant advantage in using DKNLMS-CT over DKNLMS or MDKNLMS.

Now that each DKNLMS variant has compared against KNLMS, let us now compare these variants against each other. Figures 18 & 19 shows DKNLMS and MDKNLMS convergence patterns plotted directly against each other using high values of d. Specifically, Figure 18 uses $d = 16$ while Figure 19 uses $d = 32$. In Figure 18 we see both algorithms working quite well at $d = 16$, with MDKNLMS perhaps showing a small amount of instability. In Figure 19 we see a larger difference between each configuration, both DKNLMS variants exhibiting slower convergence speeds than KNLMS, DKNLMS being much slower than MDKNLMS. More importantly, DKNLMS clearly shows some accuracy degradation when $d = 32$, while MDKNLMS behaves very similar to DKNLMS. Given this, we consider MDKNLMS to be the most promising DKNLMS variant when it comes to achieving high throughput learning models. In the subsequent sections, in order to save space we'll mostly provide results for MDKNLMS.

6.1.3 Finite Precision and Function Approximation. In this section, we test accuracy with a full architecture simulation, as such finite precision effects and function approximation is taken into account. For the Mackey-Glass time series and the hyperparameters described in Section 6.1.1 a wordlength of 18bits with a 5bit integer length was used. Figure 20 shows the modelling accuracy of the floating point vs several different fixed point versions with function approximations. Note, all fixed point formats use 5bits for the integer length which avoids overflow and Mackey-Glass time series using MDKNLMS ($d = 16$). Figure 21 shows the difference in accuracy between the floating point and fixed point versions. Note, floating point refers to double precision, i.e., 64bit float. In Figure 20 we can see that using 14bit fixed point arithmetic causes some accuracy issues, affecting the stability of MDKNLMS. Meanwhile, 18bit and 22bit does not significantly affect the accuracy of MDKNLMS ($d = 16$). In terms of the quantifying the change in modelling error, Figure 21 shows the accuracy difference between fixed point and floating point formats. In particular, with reference to both Figures 20 & 21, we can see that the difference in modelling accuracy is approximately an order of magnitude lower than the modelling error incurred by the algorithm itself, for both 18bit

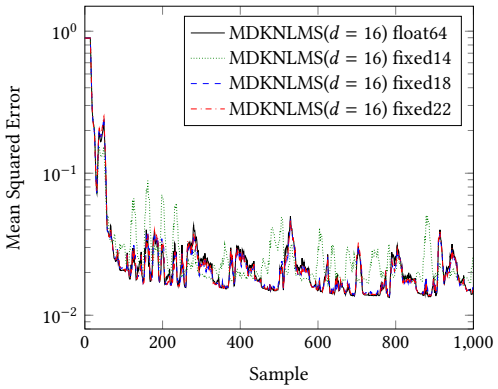

Fig. 20. Comparison between single precision floating point and 18-bit fixed point MDKNLMS when $d = 16$.

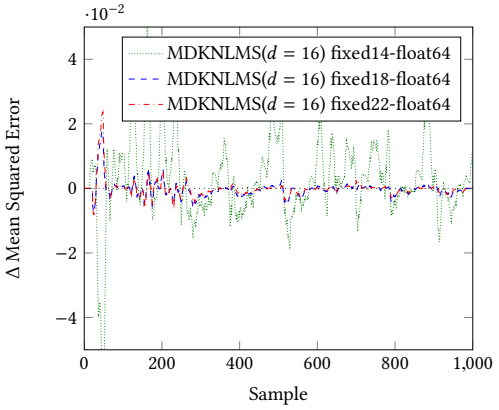

Fig. 21. Accuracy difference between single precision floating point and 18-bit fixed point MDKNLMS when $d = 16$.

and 22bit fixed point formats. The 14bit format introduces some error which would noticeably affect the overall accuracy of MDKNLMS. Interestingly, in many parts of the convergence curve, the 18bit and 22bit fixed point model actually outperforms the floating point model, in terms of accuracy. This could be due to the rounding error perhaps behaving as a regulariser in this model, by injecting noise in the form of quantisation error. This injected quantisation could actually behave like $l2$ regularisation and help the algorithm avoid overfitting [7]. Similar effects have been observed by prior works, relating quantisation error to noise injection [4, 5] and relating quantisation to regularisation [10]. Further study of this effect is future work.

6.2 Performance

In this section, we look at the potential performance and scalability of the DKNLMS architectures described Section 4. This section contains post-place-and-route results on a Xilinx VU9P FPGA. Full system implementations are found in Section 6.4.

Figure 22 shows how the throughput of DKNLMS, MDKNLMS and DKNLMS-DG varies with the latency (clock cycles) of the design. For $\tilde{N} = 64$ and $M = 8$, $d = 4$ pipeline stages are required to achieve a throughput of 100 MHz and a throughput of 280 MHz can be achieved if all of the optional registers described in Section 4 are enabled, i.e., $d = 31$. These represent speedups of 4.4×/12.0× over a combinatorial KNLMS design (i.e. $d = 1$), respectively. We do see a slight difference in maximum clock frequency between the different DKNLMS variants, but this appears to be some noise from the place-and-route tool, rather than any particular trend. Furthermore, Figure 23 provides insights into the space of frequencies which can be achieved for different values of $\tilde{N}$ and d. Overall, frequencies up to 420 MHz can be achieved for high latencies ($d \geq 26$) and low dictionary sizes ($\tilde{N} \leq 4$). However, with reference to Section 6.1, for more practical values of $\tilde{N}$ ($32 \leq \tilde{N} \leq 64$), and d ($16 \leq d \leq 32$), frequencies of between 214 MHz to 296 MHz can be achieved.

6.3 Area Usage

Figure 24 shows DSP usage as a function of M and $\tilde{N}$. All of the DKNLMS variants have the same DSP usage so this chart representative of the entire design space described in Section 4. Figure 24 shows that the DSP usage is entirely predictable across the different DKNLMS variants. The DSP usage corresponds *exactly* to the amount of multiplies specified in Table 1 until the number of

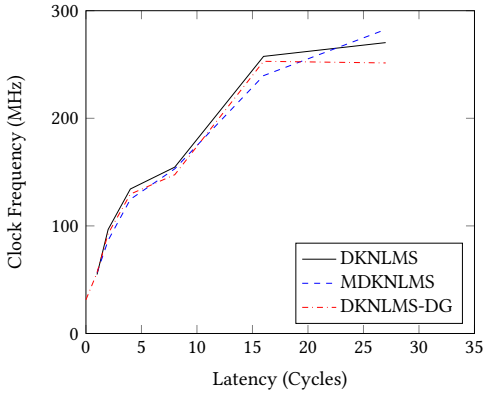

Fig. 22. Throughput versus latency (clock cycles) for $\tilde{N} = 64$ and $M = 8$ for all DKNLMS variants

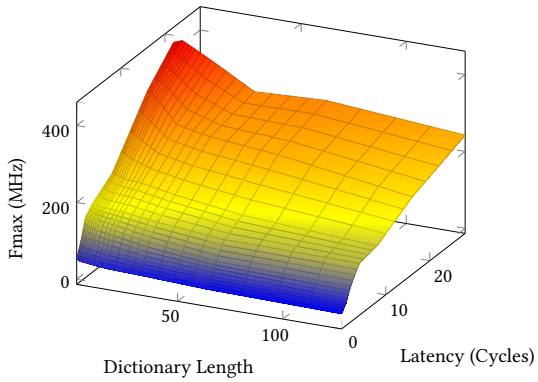

Fig. 23. Throughput versus latency (clock cycles) for $M = 8$ for MDKNLMS

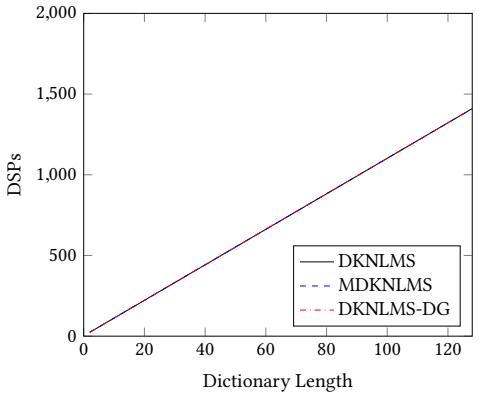

Fig. 24. DSP usage when $M = 8$ for all DKNLMS variants

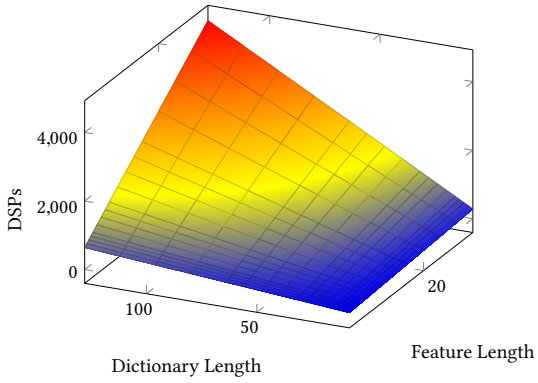

Fig. 25. DSP usage while scaling $\tilde{N}$ and M for MDKNLMS

DSPs increases beyond what is available in the target device, in which case the extra multipliers are implemented in LUTs.

Figure 26 shows LUT usage as a function of M and $\tilde{N}$. Note that in this figure the resource of DKNLMS with all optional registers enabled is shown. The other designs very closely match the resource usage shown in this figure, within 20%. Interestingly, the resource usage is higher when $d = 4$ than when $d = 8$ we suspect this is due to the routing tool replicating parts of the design to try to shorten the critical path. Clearly, the LUT usage is significantly less than the DSP usage (when one considers the availability of such resources on an FPGA device), meaning that we'll be DSP bound as we scale.

6.4 Implementation and System Performance

In this section, we consider post-synthesis results along with system level performance, latency and performance measurements. For comparison, we use two CPU-based platforms: (1) a laptop PC running Ubuntu Linux 14.04 with an Intel Core i7-4500 CPU running at 1.8GHz and GCC 4.8; and (2) an Ultra96 board from Agilent, which features an ARM Cortex-A53 CPU running at 1.2GHz

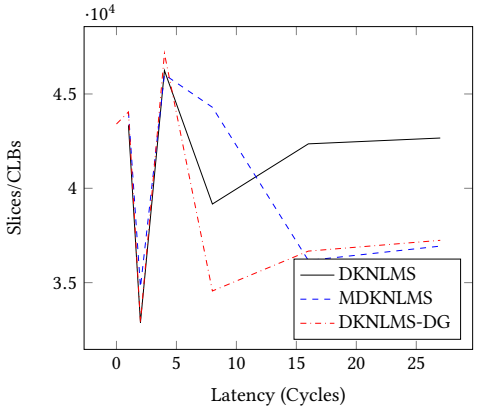

Fig. 26. Resource usage versus latency (clock cycles) for $\tilde{N} = 64$ and $M = 8$ for all DKNLMS variants

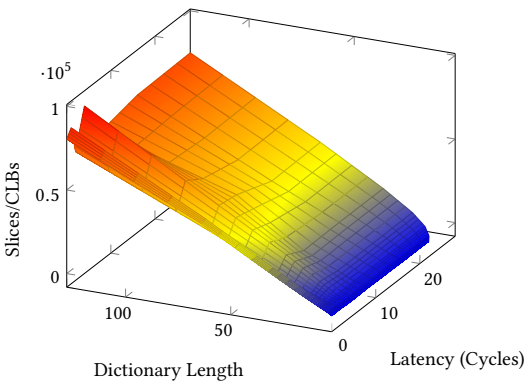

Fig. 27. Resource usage versus latency (clock cycles) for $M = 8$ for MDKNLMS

and GCC 6.2. For the CPU implementations, a C library was created to implement KNLMS. During testing, ATLAS [37], OpenBLAS [36] (both with and without multi-threading) and a single-threaded hand-coded library were used to provide the linear algebra routines. For each test, the fastest implementation is reported, which was always the hand-coded library. Although the hand-coded library was written in pure C, care was taken to ensure that GCC's auto-vectorisation capabilities could effectively optimise the code. This includes implementing loops with compile-time static loop bounds on many loops and avoid irregular memory access patterns. We examined the assembly code of several routines (including the dot product) an ensured they were significantly unrolled and made use of the processor's SIMD instructions. Our C implementation was compiled using -O3, -ffast-math and several other optimisation flags to improve performance. Surprisingly, using highly tuned linear algebra libraries and multi-threading did not provide any performance benefit, we suspect the small vector sizes used throughout meant the overheads of multi-threading outweighed the benefits. Furthermore, the fine grained parallelism available in the KNLMS algorithm make it difficult to parallelise on CPUs, for a single model. Although we were unable to extract any extra performance by going to multiple cores/threads, there is still a theoretical benefit, as such all CPU results in this section could possibly be improved by a factor c, the number of cores available in the CPU, particularly if training of multiple parallel models is required. Also, CPU implementations of DKNLMS (and the other variants) in C did not produce higher performance, therefore we only provide the performance results of KNLMS when running on the CPU. In terms of fabrication process, the i7-4500 CPU is 22nm, while the FPGA devices, an XC7Z020 and an XCZU3EG, are 28nm and 16nm respectively.

The FPGA implementations in this section fall into two categories:

(1) post-synthesis estimates
(2) system implementations

The post synthesis estimates are post-place-and-route results of the MDKNLMS cores only. The system implementations utilise AXI master to stream, AXI stream to master and generated drivers from fpga-tidbits[2] to move data between the DDR memory of the ARM host and the accelerator core and to manage the core. The system implementation also contains an extra FIFO of length d to allow

[2]https://github.com/maltanar/fpga-tidbits

Table 4. Comparison Between Different System Implementations of Various Kernel Adaptive Filters

Reference	Algorithm	Device	Dictionary Size	Feature Length	Datatype	Ops / Update	Latency (μs)	Throughput (MHz)	Board Power (W)	Performance (GOps/s)	Energy (μJ/Update)
Post-synthesis Estimates											
Tridgell et al. [32]	NORMA	XC7VX485T	32	8	Fixed18	-	0.080	137.8	-	-	-
Tridgell et al. [32]	NORMA	XC7VX485T	64	8	Fixed18	-	0.087	137.4	-	-	-
Fox et al. [16]	Fastfood	XCKU035	16,384	1,024	Fixed18	-	4.4	0.42	-	-	-
Ours	MDKNLMS(d = 16)	XC7Z020	30	7	Fixed18	872	0.26	66.7	-	58.16	-
Ours	MDKNLMS(d = 16)	XCZU3EG	46	7	Fixed18	1336	0.10	187.5	-	250.5	-
System Implementations											
-	KNLMS	ARM Cortex-A53	30	7	Float32	872	10.5	0.095	4.1	0.083	43.16
-	KNLMS	ARM Cortex-A53	46	7	Float32	1336	15.8	0.063	4.1	0.084	65.08
-	KNLMS	Intel i7-4500U	30	7	Float32	872	1.3	0.80	21.1	0.694	26.38
-	KNLMS	Intel i7-4500U	46	7	Float32	1336	2.0	0.52	21.1	0.700	40.58
Ren et al. [29]	KLMS	XC7V2000T	53†	1	Float32	-	-	2.4	-	-	
Pang et al. [27]	KNLMS	DE5	32	7	Float32	930	2.6	0.38	-	0.353	
Pang et al. [27]	KNLMS	DE5	64	7	Float32	1858	3.6	0.28	22.32‡	0.520	79.71
Fox et al. [17]	NORMA	ExaNIC X4	14	8	Fixed24	-	-	104.2	-	-	
Fraser et al. [18]	KNLMS	VC707	16	8	Float32	514	-	0.98	-	0.504	
Ours	MDKNLMS(d = 16)	Pynq-Z1	30	7	Fixed18	872	3.4	66.7	3.22	58.16	0.048
Ours	MDKNLMS(d = 16)	Ultra96	46	7	Fixed18	1336	1.5	187.4	8.38	250.4	0.045

† the dictionary size is estimated by reimplementing the work
‡ power measurement taken for the SW-KRLS algorithm

the stream to AXI controller to apply backpressure to the core. For the system implementation, the time series data starts and finishes on the host and this transfer time is included in the timing measurements for system performance. Table 4 shows our results for MDKNLMS ($d = 16$), along with key results of several previous works. In particular, note that throughput refers the sample rate for a *single model* for which the implementation is able to operate at. For our designs, this sample rate also corresponds to the clock rate of the FPGA design. As with many machine learning algorithm families, KAFs come in various different forms and target different application domains. As such, readers should be wary when directly comparing results as the specific targets of each work may not be reflected in Table 4. Given this, where possible the results from previous works are shown which closely match the algorithm and parameters used in this work. The throughput / latency numbers for the post-synthesis results refer to the sample rate and pipeline depth of the core. For the system implementations, the throughput / latency numbers refers to measured numbers from the host code (C++ code running on the ARM core). Finally, the performance (GOps/s) is calculated is the number of Ops / Update for each new sample of the adaptive filter, multiplied by the number of samples per second that can be processed by the hardware, i.e., the throughput.

For power measurements, for the Ultra96 platform board power is measured using device reported board power measurements[3]. KNLMS and MDKNLMS were run in a loop for approximately 10 minutes, with power measured every 10 seconds. The reported power is the average of the power measurements over this period. For the Pynq-Z1 platform, board power is measured using a inline USB power meter. Again, average power reported over a 10 minute test. For the PC platform, energy dissipation was measured using OS reported battery charge over a 10 minute test with the display switched off and the CPU governor set to "performance", to prevent throttling of the CPU clock.

Looking at the system implementations in Table 4, The CPU implementations achieve 0.66 / 0.084 GOps/s for the Intel i7 / ARM Cortex-A53 respectively. The architectures proposed by Ren et al. [29] and Fraser et al. [18] were designed for multi-channel systems, or hyperparameter optimization, and therefore perform poorly when constrained to accelerating a single channel. For Fraser et al. [18], this translates into 0.50 GOps/s of floating-point performance. Although the hardware by Pang et al. [27] is designed accelerate single-channel models, the authors target flexibility over performance. This means that the most comparable design ($\tilde{N} = 64$, $M = 7$) achieves

[3]Board power measured in uW by polling /sys/class/hwmon/hwmon0/power1_inputcat

Table 5. Resource Usage of System Implementations

	Algorithm	Device	Dictionary Size	Feature Length	Datatype	DSPs (%)	LUTs (%)	BRAM (18k) (%)	Fmax (MHz)
	MDKNLMS(d = 16)	Pynq-Z1	30	7	Fixed18	220 (100%)	42473 (79.84%)	16 (5.71%)	66.7
Device Total	-	XC7Z020	-	-	Fixed18	220	53200	280	-
	MDKNLMS(d = 16)	Ultra96	46	7	Fixed18	360 (100%)	54437 (77.15%)	37 (8.56%)	187.5
Device Total	-	XCZU3EG	-	-	Fixed18	360	70560	432	-

relatively low utilisation: a floating-point performance of 0.52 GOps/s. It should be noted, that the design proposed by Pang et al. [27] can be scaled to larger problems than this work. Furthermore, the design achieves better performance for RLS-style kernel methods, such as FB-KRLS [34] and SW-KRLS [35].

When comparing similar sized problem sizes, starting with $\tilde{N} = 30$ and $M = 7$, the Pynq-Z1 implementation achieves throughput 702 / 83× higher than the ARM / i7 implementations respectively. Similarly, it outperforms the implementation by Pang et al. [27], achieving 165× higher peak performance in GOps/s. When power is taken into consideration, the Pynq-Z1 design consumes the least board power of all designs 1.27 / 6.55× less than the ARM / i7 implementations respectively. Finally, the energy consumed per update for the Pynq-Z1 design is 899 / 550× less than the ARM / i7 implementations respectively.

Moving on to the larger problem size, $\tilde{N} = 46$ and $M = 7$, the Ultra96 implementation achieves throughput 2975 / 360× higher the equivalent ARM / i7 implementations respectively. For power, the ARM achieves 2.04× lower power consumption than the Ultra96 platform. Conversely, the i7 consumes 2.52× more power. In terms of energy, the Ultra96 is the most efficient design of all, achieving 1446 / 902× better efficient than the equivalent ARM / i7 implementations.

For architectures which are more similar, the design proposed in Fox et al. [17] achieves very high throughput, 104.2 MHz. In comparison, our Ultra96 implementation achieves 187.4 MHz for a larger model, at 250 GOps/s at a slightly lower precision, Fixed24 → Fixed18. This corresponds to an increased data rate of 1.80×. The Pynq-Z1 also achieves respectable performance at 58 GOps/s. Note, we consider all operations in Table 1, so exponential and division counts as a single operation, even though they are lightweight table lookups in practice. Finally, the measured latency of our designs are significantly higher than the theoretical latency of the cores themselves. This is due to a combination of DRAM latency and batching that occurs in our system implementation, These overheads would not occur in implementations which are directly coupled to sensors.

6.4.1 Resource Usage. The resource usage of our system implementations is given in Table 5. These numbers reflect the system designs reported in Table 4. Our designs utilise all of the available DSPs on each device, afterwhich multipliers are mapped to LUTs and quickly fill the available LUT resources. For both designs, we scale the core to fill almost all of the available device resources. Since the design is latency (specifically, clock cycle latency) sensitive, the overall Fmax is not affected too much by scaling the design. While the performance numbers are respectable for both the Pynq-Z1 and Ultra96 boards, the Fmax for all designs is quite low. Even HLS-based designs can usually achieve 100 / 300 MHz on the XC7Z020 / XCZU3EG respectively. This again, is due to the sensitivity of the design to latency. Each design is a core containing 16 pipeline stages, which utilises over 75% of the available LUTs on each device. For comparison, Table 5 also contains the total resources available on each device for the Pynq-Z1, Ultra96 platforms respectively.

7 CONCLUSION

In conclusion, in this work a new technique (delayed model adaptation) has been proposed for modifying KAFs. The technique significantly reduces the dependency problem of KAFs (due to their recursive nature) allowing for high throughput hardware implementations. The delayed model adaptation technique is demonstrated by modifying the KNLMS algorithm. In doing so, several new variants of KNLMS are proposed which do not suffer from the same dependency problem. A core generator is described which is able to generate all variants of KNLMS. For similarly sized problems, the DKNLMS algorithms are able to achieve speedups of 360× over a CPU and peak performance 165× higher than a previous FPGA implementation. Our most performance design can operate at data rates of 187.4 MHz and achieves a peak performance of 250 GOps/s. Furthermore, the design also achieves a 1.80× speedup over a prior FPGA implementation of NORMA. This work demonstrates that high throughput implementations of KAFs are achievable on current FPGA hardware, and hope this enables the use of KAFs in many more applications, with tight throughput requirements.

Finally, in comparison with other FPGA-based KAF designs, this work shows that one-size-fits-all solutions might not be an effective way of addressing the computational demands of machine learning algorithms. This is because machine learning algorithms are used in a vast number of application domains, with widely varying problem sizes and constraints. In particular, we show how very few prior works have addressed the high performant, single-channel online models as would be required for certain applications, such as channel equalisation. Furthermore, if those requirements are addressed specifically, and algorithmic modifications are made, significant gains can be attained.

REFERENCES

[1] Davide Anguita, Alessandro Ghio, Stefano Pischiutta, and Sandro Ridella. 2007. A hardware-friendly support vector machine for embedded automotive applications. In *Neural Networks, 2007. IJCNN 2007. International Joint Conference on*. IEEE, 1360–1364.

[2] Nachman Aronszajn. 1950. Theory of Reproducing Kernels. *Transactions of the American mathematical society* 68, 3 (1950), 337–404.

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, 1216–1225.

[4] Chaim Baskin, Natan Liss, Yoav Chai, Evgenii Zheltonozhskii, Eli Schwartz, Raja Girayes, Avi Mendelson, and Alexander M Bronstein. 2018. Nice: Noise injection and clamping estimation for neural network quantization. *arXiv preprint arXiv:1810.00162* (2018).

[5] Chaim Baskin, Eli Schwartz, Evgenii Zheltonozhskii, Natan Liss, Raja Giryes, Alex M Bronstein, and Avi Mendelson. 2018. UNIQ: Uniform Noise Injection for Non-Uniform Quantization of Neural Networks. *arXiv preprint arXiv:1804.10969* (2018).

[6] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.* 13 (2012), 281–305.

[7] Chris M Bishop. 1995. Training with noise is equivalent to Tikhonov regularization. *Neural computation* 7, 1 (1995), 108–116.

[8] Badong Chen, Nanning Zheng, and Jose C Principe. 2013. Survival kernel with application to kernel adaptive filtering. In *Neural Networks (IJCNN), The 2013 International Joint Conference on*. IEEE, 1–6.

[9] Corinna Cortes and Vladimir Vapnik. 1995. Support-Vector Networks. *Mach. Learn.* 20, 3 (Sept. 1995), 273–297. https://doi.org/10.1023/A:1022627411411

[10] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*. 3123–3131.

[11] Feng Ding. 2013. Decomposition based fast least squares algorithm for output error systems. *Signal Processing* 93, 5 (2013), 1235–1242.

[12] Feng Ding, Ling Xu, Fuad E Alsaadi, and Tasawar Hayat. 2018. Iterative parameter identification for pseudo-linear systems with ARMA noise using the filtering technique. *IET Control Theory & Applications* 12, 7 (2018), 892–899.

[13] Scott C Douglas, Quanhong Zhu, and Kent F Smith. 1998. A pipelined LMS adaptive FIR filter architecture without adaptation delay. *Signal Processing, IEEE Transactions on* 46, 3 (1998), 775–779.

[14] Yaakov Engel, Shie Mannor, and Ron Meir. 2003. The Kernel Recursive Least Squares Algorithm. *IEEE Transactions on Signal Processing* 52 (2003), 2275–2285.

[15] Gerald Estrin. 1960. Organization of computer systems: the fixed plus variable structure computer. In *Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference*. ACM, 33–40.

[16] Sean Fox, David Boland, and Philip Leong. 2018. FPGA Fastfood - A High Speed Systolic Implementation of a Large Scale Online Kernel Method. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18)*. ACM, New York, NY, USA, 279–284. https://doi.org/10.1145/3174243.3174271

[17] Sean Fox, Stephen Tridgell, Craig T. Jin, and Philip H. W. Leong. 2016. Random projections for scaling machine learning on FPGAs. In *Proc. International Conference on Field Programmable Technology (FPT)*. 85–92. https://doi.org/10.1109/FPT.2016.7929193

[18] Nicholas J. Fraser, Junkyu Lee, Duncan J. M. Moss, Julian Faraone, Stephen Tridgell, Craig T. Jin, and Philip H. W. Leong. 2017. FPGA Implementations of Kernel Normalised Least Mean Squares Processors. *ACM Trans. Reconfigurable Technol. Syst.* 10, 4, Article 26 (Dec. 2017), 20 pages. https://doi.org/10.1145/3106744

[19] Nicholas J Fraser, Duncan JM Moss, JunKyu Lee, Stephen Tridgell, Craig T Jin, and Philip HW Leong. 2015. A fully pipelined kernel normalised least mean squares processor for accelerated parameter optimisation. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*. IEEE, 1–6.

[20] Barbara Hammer and Kai Gersmann. 2003. A note on the universal approximation capability of support vector machines. *Neural Processing Letters* 17, 1 (2003), 43–53.

[21] Simon S Haykin. 2005. *Adaptive filter theory*. Pearson Education India.

[22] Jyrki Kivinen, Alexander J Smola, and Robert C Williamson. 2004. Online learning with kernels. *IEEE transactions on signal processing* 52, 8 (2004), 2165–2176.

[23] Quoc Le, Tamás Sarlós, and Alex Smola. 2013. Fastfood-approximating kernel expansions in loglinear time. In *Proceedings of the international conference on machine learning*, Vol. 85.

[24] Weifeng Liu, José C Príncipe, and Simon Haykin. 2011. *Kernel Adaptive Filtering: A Comprehensive Introduction*. Vol. 57. John Wiley & Sons.

[25] Guoz-hu Long, Fuyun Ling, and John G Proakis. 1989. The LMS algorithm with delayed coefficient adaptation. *Acoustics, Speech and Signal Processing, IEEE Transactions on* 37, 9 (1989), 1397–1405.

[26] Michael C Mackey, Leon Glass, et al. 1977. Oscillation and chaos in physiological control systems. *Science* 197, 4300 (1977), 287–289.

[27] Yeyong Pang, Shaojun Wang, Yu Peng, Xiyuan Peng, Nicholas J. Fraser, and Philip H. W. Leong. 2016. A Microcoded Kernel Recursive Least Squares Processor Using FPGA Technology. *ACM Transactions on Reconfigurable Technology and Systems* 10, 1, Article 5 (Sept. 2016), 22 pages. https://doi.org/10.1145/2950061

[28] Rainer D Poltmann. 1995. Conversion of the delayed LMS algorithm into the LMS algorithm. *Signal Processing Letters, IEEE* 2, 12 (1995), 223.

[29] Xiaowei Ren, Pengju Ren, Badong Chen, Tai Min, and Nanning Zheng. 2014. Hardware Implementation of KLMS Algorithm using FPGA. In *Neural Networks (IJCNN), 2014 International Joint Conference on*. IEEE, 2276–2281.

[30] Cédric Richard, José Carlos M Bermudez, and Paul Honeine. 2009. Online prediction of time series data with kernels. *Signal Processing, IEEE Transactions on* 57, 3 (2009), 1058–1067.

[31] Sherif A Tawfik. 2005. Mini-max Approximation and Remez Algorithm. http://www.math.unipd.it/~alvise/CS_2008/APPROSSIMAZIONE_2009/MFILES/Remez.pdf

[32] Stephen Tridgell, Duncan J.M. Moss, Nicholas J. Fraser, and Philip H.W. Leong. 2015. Braiding: a Scheme for Resolving Hazards in Kernel Adaptive Filters. In *Proc. International Conference on Field Programmable Technology (FPT)*. 136–143. https://doi.org/10.1109/FPT.2015.7393140

[33] Steven Van Vaerenbergh. 2012. Kernel Methods Toolbox KAFBOX: a Matlab benchmarking toolbox for kernel adaptive filtering. Software available at http://sourceforge.net/p/kafbox.

[34] Steven Van Vaerenbergh, I Santamaria, Weifeng Liu, and JC Príncipe. 2010. Fixed-budget kernel recursive least-squares. In *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*. IEEE, 1882–1885.

[35] S. Van Vaerenbergh, J. Via, and I. Santamaria. 2006. 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. V–V. https://doi.org/10.1109/ICASSP.2006.1661394

[36] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*. IEEE, 1–12.

[37] R. Clint Whaley and Antoine Petitet. 2005. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience* 35, 2 (February 2005), 101–121.

[38] B. Widrow and M.E. Jr. Hoff. 1960. Adaptive Switching Circuits. In *IRE WESCON Convention Record*. 96–104.
[39] Ying Yi, Roger Woods, Lok-Kee Ting, and CFN Cowan. 2005. High speed FPGA-based implementations of delayed-LMS filters. *Journal of VLSI signal processing systems for signal, image and video technology* 39, 1-2 (2005), 113–131.