FPGA Implementations of Kernel Normalised Least Mean Squares Processors

NICHOLAS J. FRASER, JUNKYU LEE, DUNCAN J. M. MOSS, JULIAN FARAONE, STEPHEN TRIDGELL, CRAIG T. JIN, and PHILIP H. W. LEONG, School of Electrical and Information Engineering, The University of Sydney

Kernel adaptive filters (KAFs) are online machine learning algorithms which are amenable to highly efficient streaming implementations. They require only a single pass through the data and can act as universal approximators, i.e. approximate any continuous function with arbitrary accuracy. KAFs are members of a family of kernel methods which apply an implicit non-linear mapping of input data to a high dimensional feature space, permitting learning algorithms to be expressed entirely as inner products. Such an approach avoids explicit projection into the feature space, enabling computational efficiency. In this paper, we propose the first fully pipelined implementation of the kernel normalised least mean squares algorithm for regression. Independent training tasks necessary for hyperparameter optimisation fill pipeline stages, so no stall cycles to resolve dependencies are required. Together with other optimisations to reduce resource utilisation and latency, our core achieves 161 GFLOPS on a Virtex 7 XC7VX485T FPGA for a floating point implementation and 211 GOPS for fixed point. Our PCI Express based floating-point system implementation achieves 80% of the core's speed, this being a speedup of 10× over an optimised implementation on a desktop processor and 2.66× over a GPU.

CCS Concepts: • **Computer systems organization → Reconfigurable computing**;

Additional Key Words and Phrases: FPGAs, machine learning, hyperparameter search, pipeline

ACM Reference format:
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 (December 2017), 20 pages.
https://doi.org/10.1145/3106744

1 INTRODUCTION

Machine learning and data mining focus on the development of mathematical ideas and algorithms to learn from data. Interest in these fields has been steadily increasing in recent years as advancements have addressed previously intractable problems such as speech recognition, handwriting recognition, image processing, credit card fraud and automatic fault detection. Kernel methods are an important class of machine-learning algorithms that include support vector

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

Authors' addresses: N. J. Fraser, J. Lee, D. J. M. Moss, S. Tridgell, J. Faraone, C. T. Jin, and P. H. W. Leong, School of Electrical and Information Engineering, The University of Sydney, Australia 2006; emails: nicholas.fraser@sydney.edu.au, eecejk@gmail.com, {duncan.moss, julian.faraone, stephen.tridgell, craig.jin, philip.leong}@sydney.edu.au.

https://doi.org/10.1145/3106744

machines (SVMs) (Scholkopf and Smola 2001), Gaussian processes (GPs) (Rasmussen and Williams 2006), and kernel adaptive filters (KAFs) (Liu et al. 2011).

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

Standard implementations of SVMs (Platt et al. 1998) and GPs (Lawrence et al. 2003) involve *batch-mode* algorithms that perform multiple passes over the training set in order to converge towards an optimal solution/model. These have time complexities that are $O(n_{TST}^2)$ or higher, where n_{TST} is the number of training samples and the resultant models cannot be incrementally updated when new data becomes available. They do not satisfy conditions (1) and (2) because storage of the entire training set is required, the result of one iteration is required before the next iteration can proceed, and many memory accesses are required per data input with processing time increasing with data size. In contrast, KAFs are recursive algorithms that perform a small, fixed amount of computation per data input and meet all the above conditions, making them amenable to efficient FPGA implementations.

Different KAF algorithms have been proposed for classification, regression, and anomaly detection tasks (Liu et al. 2011). In this article, we describe a particularly efficient implementation of the kernel normalised least mean squares (KNLMS) algorithm. KNLMS was chosen because of its simple computational structure and its ability to approximate any continuous function with arbitrary accuracy (Liu et al. 2011). The computational bottleneck for KNLMS is the evaluation of an inner product in the feature space. Our implementation is heavily pipelined, leading to high latency, which is normally undesirable. However, our core is specifically designed to address the problem of algorithm configuration in machine learning, which is known as a *hyperparameter search* (Bergstra and Bengio 2012). For a given machine-learning algorithm with P parameters, each parameter needs to be tuned to suit the dataset at hand. If P parameters are explored, at B being different values, the number of hyperparameter sets is $n_h = B^P$. Training needs to be performed on each hyperparameter set to determine its suitability to the given dataset. We exploit the independence of the parameter search to evaluate L independent parameter settings in parallel, neatly filling the KNLMS pipeline with L cycles of latency. As a result, our implementation achieves very high computational efficiency.

The key contributions of this work are:

— The first fully pipelined datapath for a KAF. Compared with previous vector-processor architectures, much higher performance can be attained because all pipeline stages do useful work and never stall. Pipeline latency is addressed by filling all stages with independent parts of a hyperparameter search.
— A number of optimisations for the KNLMS algorithm: pipelining, memory optimisations, and scheduling are combined to achieve a 575× speedup over a naïve implementation for parameter optimisation for floating point arithmetic.
— A complete PCI Express (PCIe)-based system implementation with a speedup of 10× over a processor, 2.66× over a GPU and a speedup of 660× over a previous microcoded kernel recursive least squares implementation by Pang et al. (2013).

This article is an extended version of our previous conference paper (Fraser et al. 2015). Additional material in article paper includes expanded background and architecture sections; a new fixed-point implementation that uses 35% of LUTs and 17% of DSPs, while achieving 60% lower latency than the previously reported floating-point implementation; addition of a random search, which often finds better hyperparameters than the previously reported grid search (Bergstra and Bengio 2012); and an improved PCIe interface that obviates the transfer of n_h regression values per input vector and achieves 4× higher throughput over the previously reported design. In this work, we have also included the results of a much improved CPU implementation of a parameter search and a GPU implementation as a comparison. Finally, different problem sizes have been considered by experimenting with folding the algorithm on the core, allowing larger problems to be tackled while reducing the cores' throughput.

This article is organised as follows: Section 2 describes the KNLMS algorithm (Richard et al. 2009), hyperparameter search, and literature review; Section 3 describes the proposed architecture; Section 4 shows the performance and accuracy results of the proposed architecture compared with CPU/GPU implementations; and conclusions are drawn in Section 5.

2 BACKGROUND

2.1 Kernel Normalised Least Mean Squares

In this section, the KNLMS algorithm (Richard et al. 2009) is summarised with particular attention to aspects that affect hardware implementations.

In a standard supervised learning problem, training examples are input/output pairs $\{\mathbf{x}_i, y_i\}$, where $\mathbf{x}_i \in \mathbb{R}^M$ is the input vector and $y_i \in \mathbb{R}$ is the output or target. In regression, the goal is to estimate a function, $f(\mathbf{x}_i)$, which maps $\mathbf{x}_i \rightarrow y_i$. Kernel regression attempts to estimate this function by learning a dictionary $\mathcal{D}$, containing a subset of input vectors, and corresponding weights, α. A prediction, $\tilde{y}_i$, is then calculated as follows:

$$\tilde{y}_i = \sum_{n=1}^{N} \alpha_n \kappa(\mathbf{x}_i, \tilde{\mathbf{x}}_n), \tag{1}$$

where $\tilde{\mathbf{x}}_n$ is the n^{th} entry in $\mathcal{D}$, α_n is the n^{th} entry of α, N is the maximum size of $\mathcal{D}$, and $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ is the kernel function, specified at design time. Although different kernels can be accommodated, in this article we focus on the commonly used radial basis function (RBF) kernel, $\kappa(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2}$, where γ is a free parameter chosen to suit the problem at hand.

The KNLMS algorithm is a stochastic gradient descent–based algorithm that learns its model by taking small steps in the direction of the instantaneous gradient to minimise the error in the current training example. Similar to algorithms such as the least mean squares (LMS) algorithm (Widrow and Hoff 1960), it slowly converges to a solution over time.

The coherence criterion (Richard et al. 2009) is used to select the entries in the dictionary. For unit norm kernel functions, the coherence criterion is defined as follows: given a new input example at iteration t, $\mathbf{x}_t$ is added to the dictionary if $\max(|\mathbf{k}_t|) \leq \mu_0$, where $\mathbf{k}_t$ is the kernel vector with the n^{th} element being given by $\kappa(\mathbf{x}_t, \tilde{\mathbf{x}}_n)$ and μ_0 is the coherence parameter chosen at design time. The weights for each iteration are then calculated by solving the instantaneous approximation to the following affine projection problem:

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

where $\hat{\alpha}_{t-1}$ is the set of weights obtained from the previous iteration and $\dagger$ denotes the vector transpose operation. Assuming that the current input vector, $\mathbf{x}_t$, can be adequately represented by the current dictionary and is not added to the dictionary, Equation (2) can be solved by minimising the following Lagrangian function:

$$J(\alpha, \lambda) = \|\alpha - \hat{\alpha}_{t-1}\|^2 + \lambda(y_t - \mathbf{k}_t^{\dagger} \alpha). \tag{3}$$

ALGORITHM 1: KNLMS Algorithm with Coherence Criterion

```
Initialise the step size, $\eta$, the regularization factor, $\epsilon$,
the coherence parameter, $\mu_0$, and select a kernel function, $\kappa$. Insert $\mathbf{x}_1$ into the dictionary, denoting it as $\tilde{\mathbf{x}}_1$.
$\mathbf{k}_1 = \kappa(\mathbf{x}_1, \tilde{\mathbf{x}}_1)$, $\hat{\boldsymbol{\alpha}}_1 = 0$, n = 1.
while $t > 1$ do
    Get $\{\mathbf{x}_t, y_t\}$. Calculate $\mathbf{k}_t = [\kappa(\mathbf{x}_t, \tilde{\mathbf{x}}_1), \ldots, \kappa(\mathbf{x}_t, \tilde{\mathbf{x}}_n)]^\dagger$.
    if max $(|\mathbf{k}_t|) > \mu_0$ then
        Update $\hat{\boldsymbol{\alpha}}_t$ using Equation (5)
    else
        $n = n + 1$. Append $\kappa(\mathbf{x}_t, \mathbf{x}_t)$ to $\mathbf{k}_t$. Insert $\mathbf{x}_t$ into the dictionary, denoting it as $\tilde{\mathbf{x}}_n$. Update $\hat{\boldsymbol{\alpha}}_t$ using
        Equation (6).
    end
end
```

A solution, $\hat{\alpha}_t$, is found by differentiating Equation (3) with respect to α and λ and setting the derivatives to zero, giving:

$$\begin{aligned} 2(\hat{\alpha}_t - \hat{\alpha}_{t-1}) &= \mathbf{k}_t \lambda \\ y_t &= \mathbf{k}_t^\dagger \hat{\alpha}_t. \end{aligned} \tag{4}$$

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

$$\hat{\alpha}_t = \hat{\alpha}_{t-1} + \frac{\eta}{\epsilon + \mathbf{k}_t^\dagger \mathbf{k}_t}\left(y_t - \mathbf{k}_t^\dagger \hat{\alpha}_{t-1}\right)\mathbf{k}_t, \tag{5}$$

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

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

$$\hat{\alpha}_t = \begin{bmatrix} \hat{\alpha}_{t-1} \\ 0 \end{bmatrix} + \frac{\eta}{\epsilon + \mathbf{k}_t^\dagger \mathbf{k}_t}\left(y_t - \mathbf{k}_t^\dagger \begin{bmatrix} \hat{\alpha}_{t-1} \\ 0 \end{bmatrix}\right)\mathbf{k}_t. \tag{6}$$

Pseudocode for the KNLMS algorithm, adapted from Richard et al. (2009) and Yukawa (2012), is shown in Algorithm 1.

A simplified block diagram of a KAF implementation is shown in Figure 1. For KNLMS, the universal approximator block implements Equation (1); the modify weights block implements Equation (6). A high degree of pipelining is necessary for high throughput, but it should be evident that a new input cannot be processed until after the weights are modified. Assuming L cycles of latency from the input x_i to when the new weights have been determined, the next input cannot be processed until at least L cycles later, that is, the initiation interval is L cycles.

2.2 Hyperparameter Search

In general, machine-learning engineers make predictive models of data. When a machine-learning engineer is confronted with a new dataset, they must choose the following: (1) which algorithm they will use to model the data, and (2) for a given algorithm, how to set its *hyperparameters*. In KNLMS, the dictionary, $\mathcal{D}$, and weights, α, are parameters that define the model, not hyperparameters that define the algorithm configuration. The parameters are learned from the training data, while the hyperparameters affect how a particular algorithm learns from that data.

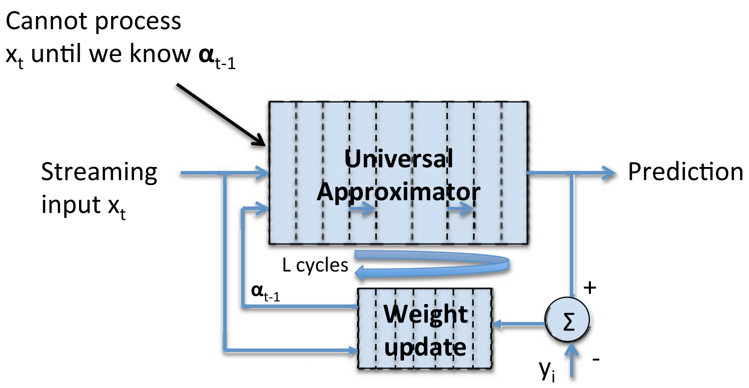

Fig. 1. Pipelined implementation of a kernel adaptive filter.

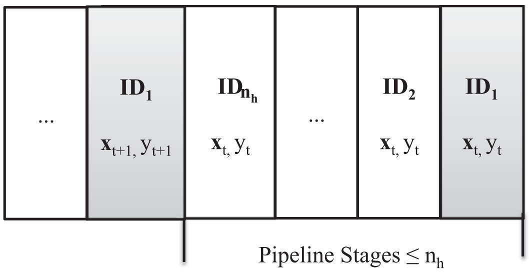

Fig. 2. Required number of independent hyperparameter searches. In IDx, x represents a particular hyperparameter set.

The setting of hyperparameters has been shown to have significant impact on the modelling accuracy of a given algorithm. In particular, Cox and Pinto (2011) show that hyperparameter selection can be the difference between state-of-the-art or chance modelling accuracy. As such, to ensure that high modelling accuracy can be achieved, some sort of hyperparameter space exploration must be performed. In this work, we focus on a hyperparameter search, which seeks an optimised parameter set that minimises a cost function (Claesen and de Moor 2015), E. For regression problems, this cost function is often the least squares error function: $E = \sum_{n=0}^{N}(f(\mathbf{x}_n) - y_n)^2$. For classification problems, this is often the number of misclassified examples in a set. In Algorithm 1, the hyperparameter set is $\{\eta, \gamma, \mu_0, \epsilon\}$, these values being required by Equation (5), Equation (6), and the kernel function κ.

As previously mentioned, dependencies in Algorithm 1 exist on the update of $\hat{\alpha}_t$ and $\mathcal{D}_t$. However, if P parameters are tested at B different values, the number of hyperparameter sets to test is $n_h = B^P$. A pipelined implementation with latency, L, less than or equal to n_h can thus be fully utilized by evaluating hyperparameter settings in different stages, as shown in Figure 2. Put another way, hazards on contiguous inputs with the same ID in Figure 2 can be resolved by evaluating independent hyperparameter settings on successive cycles. For practical problems, this is almost always the case.

An alternative search algorithm uses random rather than evenly spaced candidates (Bergstra and Bengio 2012; Pinto et al. 2009); for example, for 4 hyperparameters, random search generates points randomly in a four-dimensional space. This has the advantage that the experiment can be stopped at any time and the trials form a complete experiment, new trials can be added at any time, each trial is independent, and random search can be more efficient (Bergstra and Bengio 2012).

2.3 Literature Review

Kernel methods are eminently amenable to efficient hardware implementations and several implementations of SVM have been reported. Anguita et al. (2011) described an SVM core generator that allowed for different speed, resource, and accuracy trade-offs and utilised fixed-point arithmetic. Papadonikolakis and Bouganis (2008) described a scalable SVM module generator that supported different kernel types. Their design supported different numbers of parallel computing tiles that allowed for performance/resource trade-offs and was partitioned into fixed-point and floating-point sections to achieve high performance while maintaining high accuracy. The MAPLE architecture, as described by Majumdar et al. (2012), was designed to improve many learning algorithms, including SVM. MAPLE utilised two-dimensional vector-processing elements to accelerate linear algebra routines. The architecture also supported off-chip memory, allowing it to accommodate large learning problems. While much of the computation is similar to KNLMS, as explained in the introduction, SVM is not suitable in online applications in which storage of the entire training set is undesirable or when incremental model updates are required.

Pang et al. (2013) proposed a compact and low-latency microcoded soft vector processor. Parallelism was achieved using up to 128 floating-point vector-processing elements, and kernel evaluations were accelerated through the inclusion of a hardware exponentiation unit. An implementation of the sliding window kernel recursive least squares (SW-KRLS) algorithm (Van Vaerenbergh et al. 2006) was used as an example in the paper. The floating-point implementation of the quantized kernel least mean squares (QKLMS) algorithm (Chen et al. 2012) utilising the survival kernel (Chen et al. 2013) by Ren et al. (2014) is most comparable to this work. Differences include the following: (1) their work was limited to a single-dimensional kernel versus arbitrary dimensions; (2) they employed the survival kernel compared to the much more commonly used Gaussian kernel in this design; and (3) they achieve parallelism through pipelining and 128 parallel processing elements, whereas our efficiency is by virtue of a fully pipelined design.

In this work, we avoid the dependency problem created by the recursive update expressions of KNLMS by filling the pipeline with independent models that are training during the hyperparameter phase of training a machine-learning algorithm. Alternative approaches to avoiding this problem include delayed model adaptation (Long et al. 1989), correction terms (Poltmann 1995), and *braiding* (Tridgell et al. 2015).

As far as the authors are aware, there are no specific FPGA-based coprocessors for GPs, although general-purpose machine-learning accelerators (such as Majumdar et al. 2012) and, in particular, ones that include specific kernel function accelerators (such as Pang et al. 2013) would most likely be able accelerate GPs. Coprocessors may not have been implemented for GPs because of their close relationship to SVMs (Seeger 2000) or perhaps simply because few hardware designers have knowledge of GPs.

Delayed model adaptation has been applied effectively to the LMS algorithm (Widrow and Hoff 1960). The resultant delayed LMS (DLMS) algorithm (Long et al. 1989) has been shown to be stable under certain configurations and it has been shown that high-frequency FPGA implementations can be achieved (Yi et al. 2005). Since introducing a model adaptation delay changes the learning characteristics of an algorithm, it may not be suitable for hyperparameter optimisation for the following reasons: (1) as far as these authors are aware, no stability analysis has been done for a version of LMS that uses a kernel function; (2) the hyperparameters found using a delayed version of KNLMS may differ significantly from KNLMS itself, that is, the learned hyperparameters would need to be deployed on a system that also implements KNLMS with a model adaptation delay; and (3) the computational complexity of KNLMS is higher than LMS, as such the number of pipeline stages (see Section 4) required to achieve high clock frequencies would most likely cause undesirable learning behaviour.

Correction terms are a way of rearranging the equations of a recursive algorithm in order to reduce the critical path of the dependency loop. They have been applied to the LMS algorithm (Poltmann 1995) and utilised effectively in hardware to create high-frequency implementations of LMS (Douglas et al. 1998) that maintain the same learning characteristics of LMS. Adding correction terms usually introduces redundant calculations (when compared with a serial implementation). Also, the amount of extra computations required is proportional to the number of pipeline stages used and, as such, would significantly reduce the maximum dictionary size/feature length that can be realised by our core.

Braiding (Tridgell et al. 2015) was used effectively to overcome the dependencies present in the Naive Online regularised Risk Minimization Algorithm (NORMA) (Kivinen et al. 2004) and endeavours to optimise latency rather than throughput, which is the topic of the present work. It can be seen as an extension of correction terms to kernel-based online learning algorithms when the dictionaries of such algorithms are created from a sliding window. The KNLMS algorithm used in this work does not have this property for the entries of its dictionary; rather, it uses the coherence criterion (Richard et al. 2009) to select its entries. However, it is possible that the technique could be extended to KNLMS, but braiding also suffers from an increase in calculation requirements proportional to the number of pipeline stages in the circuit. Similar to correction terms, this would likely significantly limit the problem sizes that could be tackled by our core.

3 ARCHITECTURE

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

Referring to Algorithm 2, our main insight is to reorder the commonly used search loop on the left to that on the right. This allows the independent hyperparameter evaluations to fill the pipeline and avoid dependencies, as previously explained in Section 2.2. Note that n_h is the number of hyperparameter sets to be evaluated.

3.1 High-Level Description

The idea behind the design is to create a module to accelerate the operations required to update the kernel regression model from timestep $t - 1$ to t, that is, those within the while loop of Figure 1. We call this the *forward path*. A block diagram showing the basic structure of the processor is shown in Figure 3. The submodules are responsible for the following functions: (1) the kernel modules calculate the kernel vector, $\mathbf{k}_t$; (2) the coherence criterion module decides whether to add the latest input example to the dictionary and updates it if necessary; (3) the dot product modules, which produce the *a priori* estimate of y_t, denoted by $\tilde{y}_t$, and the normalisation term, $||\mathbf{k}_t||^2$; and (4) the α update module produces the updated weights, α_t. In order to compute multiple iterations of the KNLMS algorithm, the forward path module is controlled by a scheduler, which is described in Section 3.5.

3.2 Kernel Module

Figure 4 shows the dataflow graph of a kernel module. The kernel module computes the Gaussian kernel, given by $\kappa(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma ||\mathbf{x}_i - \mathbf{x}_j||^2}$. Each kernel module requires $2M - 1$ adders, $M + 1$ multipliers, and 1 exponential unit, where M is the feature length. N kernel modules are required for a

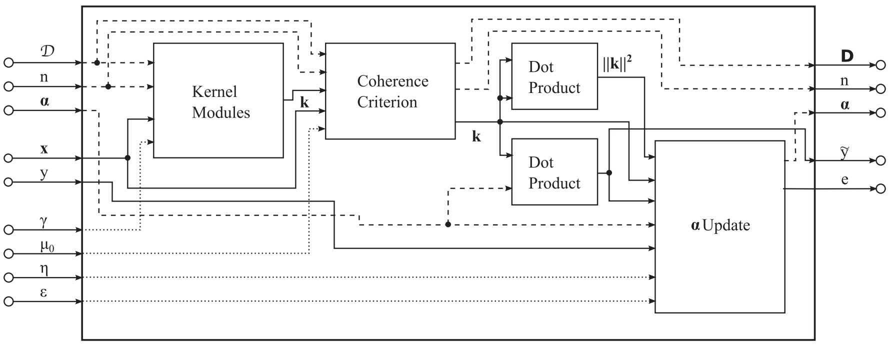

Fig. 3. A block diagram of the KNLMS processor showing the various submodules.

ALGORITHM 2: Parameter Searches

Initialise hyperparameter sets Λ_{1-n_h}, training data $TR_{1-n_{TR}}$, and testing data $TST_{1-n_{TST}}$. $\boldsymbol{\alpha}_{\Lambda_i}$: Weights trained with Λ_i, $\mathbf{k}_{j(\Lambda_i)}$: a kernel vector produced from TST_j and dictionary trained with Λ_i

Normal Grid/Random Search

```
Initialise the sums of errors $E_{1-n_h} = 0$
for $i = 1 : n_h$ do
    for $j = 1 : n_{TR}$ do
        Execute Alg 1 using $\Lambda_i$ with $TR_j$
    end
    for $j = 1 : n_{TST}$ do
        $E_i += (y_j - \mathbf{k}^{\dagger}_{j(\Lambda_i)}\boldsymbol{\alpha}_{\Lambda_i})^2$
    end
end
Pick $\Lambda_\beta$, where $\beta = argmin_\beta E_\beta$
```

Pipelined Search

```
Initialise the sums of errors $E_{1-n_h} = 0$
for $i = 1 : n_{TR}$ do
    for $j = 1 : n_h$ do
        Execute Alg 1 using $\Lambda_j$ with $TR_i$
    end
end
for $i = 1 : n_{TST}$ do
    for $j = 1 : n_h$ do
        $E_j += (y_i - \mathbf{k}^{\dagger}_{i(\Lambda_j)}\boldsymbol{\alpha}_{\Lambda_j})^2$
    end
end
Pick $\Lambda_\beta$, where $\beta = argmin_\beta E_\beta$
```

design supporting a maximum dictionary size of N. The most computationally expensive part of the KNLMS processor is the calculation of the kernel vector.

3.3 Alpha Update Module

The α update module finishes the training step by calculating α_t as shown in Equation (5). The dataflow graph for the α update module is shown in Figure 5. The α update module first calculates the prediction error and normalisation term. This is followed by a scalar vector product and an elementwise vector addition. The α update module operates on vectors of length N and, as such, requires $N + 1$ multipliers, $N + 2$ adders, and 1 divider.

3.4 Coherence and Dot Product Modules

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

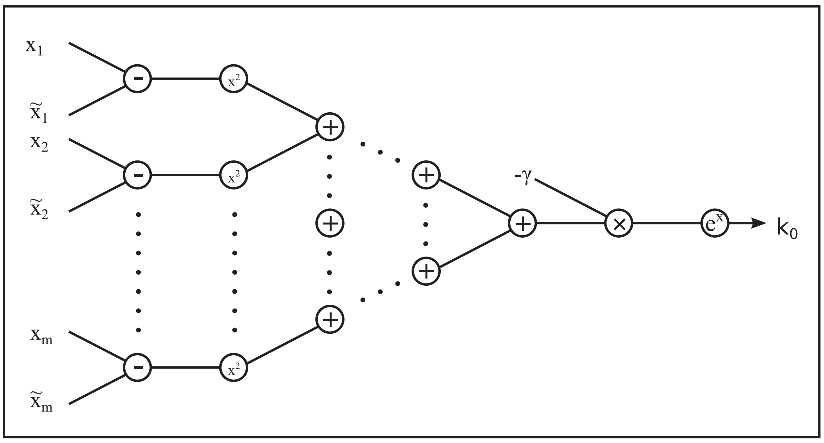

Fig. 4. Dataflow diagram of a kernel module.

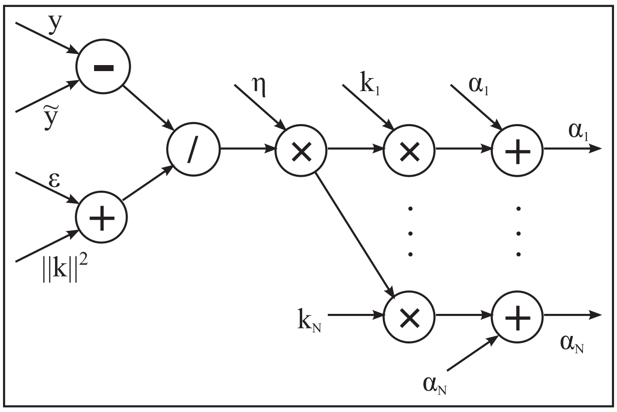

Fig. 5. Dataflow diagram of the α update module.

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

3.5 Optimisations

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

A fully pipelined design cannot be directly synthesised from a C++ description of the algorithm in Figure 1 due to the dependency of the updated dictionary $\mathcal{D}$ and weights $\hat{\alpha}_t$ on the new kernel vector $\mathbf{k}_t$. By omitting the update steps, we can turn the datapath into an acyclic one. The feedback connection is then made by externally connecting outputs to corresponding inputs in Figure 3. This results in the desired design with an initiation interval of 1. Dictionary and weight updates are delayed by L, the latency of the KNLMS core.

A significant bottleneck in creating accurate machine-learning models is parameter optimisation. Even if the kernel function is fixed to be the Gaussian kernel, a search is required over the hyperparameters. This involves performing regression over a test dataset using different parameter settings. Since these are independent problems, they can be executed in parallel as n_h-independent tasks. Each task is executed in a different pipeline slot so that all hardware units in the KNLMS forward path evaluation pipeline can be fully utilised.

Hyperparameter search using KNLMS requires control logic and BRAMs beyond that of Figure 3. In order to perform regression on n_h-independent problems, we require storage of $n_h\times$ dictionaries of size MN and $n_h\times$ length-M weight vectors. This is achieved by using block RAMs and indexing them with a counter $l \in [0, \ldots, n_h)$ so that every n_h cycles, we return to the same dictionary and weight vector. This arrangement removes the need for an $n_h : 1$ multiplexer per dictionary and weight entry. Since writing and reading dictionary and weight vector entries are required each cycle, dual-port BRAMs are used. The architecture produces an error e and updates $\mathcal{D}$, $\boldsymbol{\alpha}$, and n every cycle.

One further improvement over Fraser et al. (2015) is support for random search. A parameter memory (as shown in Figure 7) per hyperparameter allows runtime downloading of parameter choices to be searched. If initialised to grid points, a grid search is realised; if the parameter memory is initialised to random points, a random search will be performed. An alternative approach might be to replace the parameter memory with a state machine that generates grid or random points. In practice, the current architecture is usually preferable since it is more flexible and the number of search points per parameter is modest.

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

3.6 Implementation of the KNLMS Core

For this work, Vivado High-Level Synthesis (HLS) was used to implement the core. Vivado HLS was used because it allows the core designer to explore a large space of possible designs while making only small changes to the code. Also, if there are no data dependencies present within the C++ used to describe the core, Vivado HLS will automatically pipeline the core in an attempt to meet the desired clock constraint.

First, the forward path of the KNLMS algorithm was implemented as a C++ function that takes the hyperparameters, the model parameters ($\mathcal{D}$, α), and a single training example as input. As output, the function returns the updated weights, dictionary, and an *a priori* prediction for the training example. Unlike some other implementations of adaptive filters (such as Douglas et al. (1998) and Yi et al. (2005)) the model parameters are not *stored* within the core itself; rather, they are passed as inputs to the core. As such, there are no dependencies within the core, which means that it may be pipelined arbitrarily without affecting the accuracy of the resultant updated model.

The top-level module (in HLS) sits outside the core. It uses BRAMs to store the different parameter configurations and the corresponding models, that is, the dictionaries $\mathcal{D}$, and weights, α. The module then feeds each training example, parameter configuration, and model into the core while ensuring that the same parameter configuration does not exist within the pipeline of the KNLMS core multiple times.

3.7 System Implementation of Hyperparameter Search

The floating-point KNLMS core can be integrated with a PCIe interface, as illustrated in Figure 6. Data ingress is controlled by a first in, first out (FIFO) input, and an n_h-word parameter memory ($n_h \geq L$) for each of the 4 parameters shown in the bottom left module of Figure 3 was used to

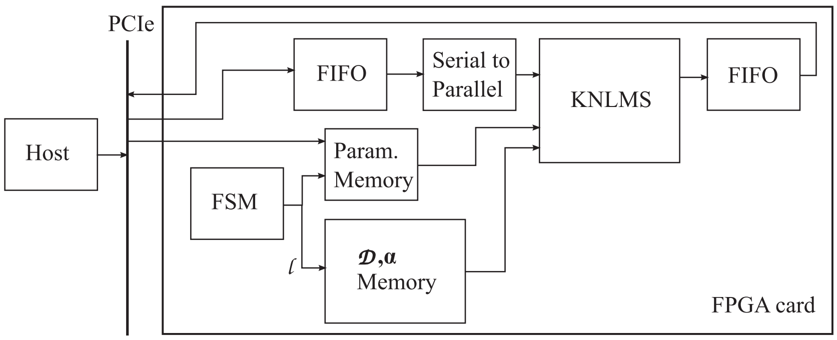

Fig. 6. Block diagram illustrating system integration of the KNLMS processor.

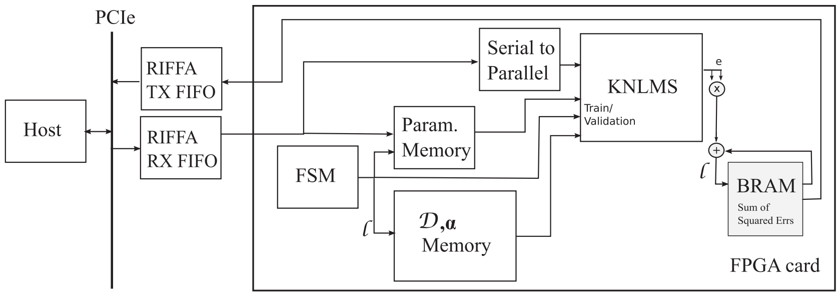

Fig. 7. Block diagram illustrating improved system implementation of the KNLMS processor.

store the parameters to be searched. Separate memories indexed by l (as detailed in Section 3.5) were used to store dictionary and weight values.

When the input FIFO becomes nonempty, the serial to parallel converter converts the data to a vector. A sequence of n_h-independent optimisations with different parameter values is then streamed through the KNLMS processor. In our previous implementation (Fraser et al. 2015), an output FIFO was used to collect n_h output values per input vector. In such an implementation, care must be taken to limit the number of inputs sent before the outputs are read. While this avoids deadlocks, it also limits the burst transfers made on the PCI bus and reduces performance.

An improved system implementation, illustrated in Figure 7, avoids transfers from the KNLMS processor by accumulating the squared error values on chip. The host code simply writes the parameters and training set to the core via the RIFFA2 interface and reads back an expected number of words from the same interface. This allows all of the training input data to be transferred to the core with a single data stream transfer RIFFA API function call for each training-validation set. Therefore, this improved system implementation minimises the overhead from the blocking operations between sending and receiving data for the RIFFA interface. We implemented a 2-fold cross-validation platform with the architecture of Figure 7. A subsequent read of n_h sum of squared error values is required to determine which set achieved the lowest error. The n_h numeric values from the system implementation were verified, comparing to the results from the C implementation.

3.8 Arithmetic

Two implementations were made, one using floating point and the other fixed point. The former has advantages of large dynamic range and ease of comparison with microprocessor-based implementations, whereas the fixed-point implementation offers lower latency and smaller area.

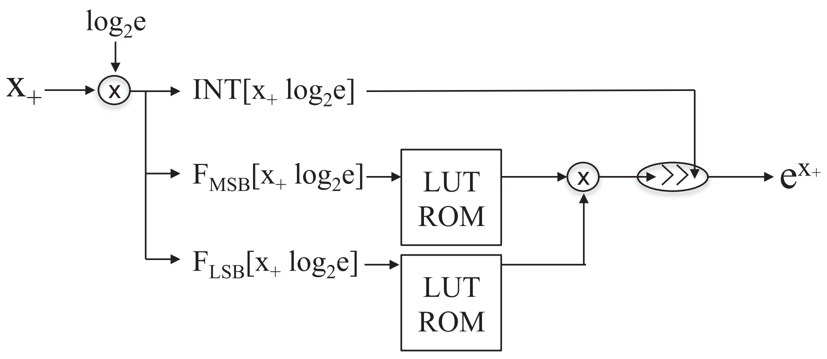

Fig. 8. Dataflow of the exponential function module.

Implementation of the basic operators in fixed-point arithmetic is straightforward. In this article, we chose a 21b two's complement fixed-point representation (5 integer bits and 16 fraction bits) with truncated rounding and saturating arithmetic. As shown later, this is sufficient precision to obtain comparable results with single-precision floating point.

Hardware architectures for evaluating the exponential function have been previously reported (Alachiotis and Stamatakis 2011; Detrey and de Dinechin 2005; Jamro et al. 2007; Pottathuparambil and Sass 2009; Wielgosz et al. 2008). We employed a lookup table(LUT)–based approach presented in Wielgosz et al. (2008), without the Taylor series expansion. The approximation is based on the mathematical property

$$e^x = 2^{x \cdot log_2 e} = 2^{x_I} \cdot 2^{x_F}, \tag{7}$$

where x_I is the integer part and x_F is the fraction part of $x \cdot log_2 e$. Since we only use e^x in computing the Gaussian function, $e^{-\gamma\|\cdot\|^2}$, $x = -\gamma\| \cdot \|^2 \leq 0$. Hence, the input domain for Equation (7) can be expressed using nonnegative numbers x^+, that is, $x = -x^+$. We further represent x^+ in terms of its integer and fractional parts so that $x^+ = x_I^+ .x_F^+$ and break x_F^+ into most significant ($x^+_{F_{MSB}}$) and least significant ($x^+_{F_{LSB}}$) parts to give

$$e^x = 2^{x_I} \cdot 2^{x_F} = 2^{-x_I^+} \cdot 2^{-x^+_{F_{MSB}}} \cdot 2^{-x^+_{F_{LSB}}}. \tag{8}$$

Figure 8 illustrates our implementation using two lookup tables. The width for both $F_{MSB}[x_+ \cdot log_2 e]$ and $F_{LSB}[x_+ \cdot log_2 e]$ was chosen to be 8b. Consequently, the size of each of the two LUTs is 2^8. N exponentiation modules are required in a fully pipelined KNLMS design.

The fixed-point implementation produces an error less than or equal to one half of one unit in the last place (Detrey and de Dinechin 2005; Wilkinson 1994), that is,

$$|e^x - \hat{e}^x| = 2^{-x_I^+} \cdot |2^{-x_F^+} - \hat{2}^{-x_F^+}| \leq \xi, \tag{9}$$

where l is the number of bits in x_F, $\xi = 2^{-(l+1)}$ is a machine epsilon in a fixed-point data format and $\hat{e}^x$ is the exponential estimation of the implementation.

3.9 Area and Latency

We estimated the scalability of the architecture with the key parameters N and M. The number of required operators and estimated latency is shown in Table 1. The operator latency is given in parentheses next to the operator symbol. In order to estimate the latency for a given design, the operator latency is multiplied by the expression in the latency row. In terms of worst-case

Table 1. Formulae for the Number of Floating-Point Operators Required and Latency in Cycles for Float, Fixed: 5b for Integer and 16b for Fraction with Saturating Arithmetic

	+ (11,3)	× (7,6)	/ (30,41)	exp (20,9)	< (4,1)
Operation	$2MN + 2N$	$MN+ 4N + 1$	1	N	$N - 1$
Latency	$\log_2 N+ \log_2 M + 3$	5	1	1	$\log_2 N$

Table 2. Summary of Place and Route Output for Each of the KNLMS Designs

	Naïve (N=16)	Float (N=16)	Fixed (N=16)	Fixed (N=32)
BRAM18K (2060)	8 (0.4%)	145 (7.0%)	138 (6.8%)	274 (31.3%)
DSP48 (2800)	12 (0.4%)	1267 (45.3%)	210 (7.5%)	418 (14.9%)
LUTs (304K)	4,550 (1.5%)	150,494 (49.5%)	53,273 (17.3%)	109,546 (36.2 %)
Latency (cycles)	756	207	121	124
II (cycles)	757	1	1	1
$F_{max}(MHz)$	96.7	314	286	206
GOPS	0.07	161.1	146.7	211.2

scalability, the area of arithmetic operators is $O(MN)$, memory usage is $O(MN)$, and latency $O(\log_2 N + \log_2 M)$.

In the following section, linear regression is used to model area (LUTs and DSPs) and latency (L) using the formulae

$$LUTs = l_1 MN + l_2 N + l_3 \tag{10}$$

$$DSPs = d_1 MN + d_2 N + d_3 \tag{11}$$

$$L = L_1 \log_2 MN + L_2 \log_2 N + L_3 \tag{12}$$

4 RESULTS

This section describes the resource utilisation, performance, and accuracy of the implementation written in C. The design was synthesised and implemented using Xilinx Vivado HLS 2015.3. The target platform was a Xilinx VC707 evaluation board using a Xilinx Virtex 7 XC7VX485TFFG1761-2 FPGA.

4.1 Synthesis and PAR Results

Table 2 shows place and route (PAR) results for the four different implementations of Figure 3: (1) Naïve—an unoptimised Vivado HLS synthesised implementation derived from KAFBOX (Van Vaerenbergh 2012), representing a design without consideration of the resulting hardware datapath; (2) Float—a single precision floating-point design that uses the Xilinx floating-point cores throughout and contains all optimisations described in Section 3.5; and (3) Fixed (5 integer bits and 16 fraction bits)—with all optimisations described in Section 3.8.

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

Table 3. Area Utilisation of Different Designs Obtained from Synthesis

Type	M	N	LUTs (Est)	DSPs (Est)	L (Est)	F_{max}
Float	2	16	77K (77K)	595 (595)	185 (185)	385
	4	16	109K (109K)	819 (819)	196 (196)	385
	16	16	307K (307K)	2163 (2163)	218 (218)	385
	8	2	23K (25K)	161 (161)	162 (162)	385
	8	4	46K (47K)	319 (319)	177 (177)	385
	8	8	95K (90K)	635 (635)	192 (192)	385
	8	16	173K (175K)	1267 (1267)	207 (207)	385
Fixed	2	16	31K (31K)	196 (197)	112 (109)	321
	4	16	47K (47K)	260 (261)	115 (116)	321
	16	16	142K (142K)	644 (645)	133 (130)	321
	8	2	13K (14K)	54 (59)	108 (107)	321
	8	4	23K (24K)	106 (106)	111 (112)	321
	8	8	47K (42K)	210 (201)	116 (118)	321
	8	16	78K (79K)	388 (389)	121 (123)	321

Note: Estimates obtained via linear regression are in parentheses.

reference to Table 1, the number of operations is 513. Note that for the Naïve and Float designs, GOPS is equivalent to GFLOPS.

Table 3 shows the relationship between the design parameters M and N, and the latency and hardware resources. These numbers are different from Table 2 because they are postsynthesis estimates rather than place and route results.

Using the formulae in Equations (10) to (12), linear regression was applied to obtain linear models. The estimates are given in parentheses in Table 3; a maximum percentage error of less than 10% was observed. This confirms that the simple area and latency models in Table 1 are capable of accurate prediction.

As expected, Fixed requires fewer resources and can support a larger dictionary. For $N = 32$ and $M = 8$, Fixed achieved 211 GOPS (1,025 operations at 206MHz) and only used 36% of LUTs, 15% of DSPs, and 31% of BRAMs.

4.2 Learning Accuracy with a Grid/Random Search

We examine out-sample error for our implementations for the chaotic MG-30 Mackey-Glass benchmark modelling the differential equation $dx(t)/dt = ax(t-\tau)/(1+x(t-\tau)^{10}) - bx(t)$ with ($a = 0.2, b = 0.1, \tau = 30$), as implemented in KAFBOX (Van Vaerenbergh 2012). A training set of 3,999 samples was used for hyperparameter search with 3-fold cross-validation. We use 4 hyperparameter candidates for each hyperparameter in a grid search (i.e., $n_h = 4^4$) and 256 random hyperparameter sets in a random search. The ranges for parameter sets for grid and random parameter searches are set as follows: $\gamma = [0.01, 2]$, $\eta = [0.05, 0.3]$, $\epsilon = [0.001, 0.5]$, and $\mu_0 = [0.5, 0.8]$. The result of this execution was two hyperparameter sets.

The quality of the hyperparameters were tested on one hundred 1100 out-sample data points, which we call the testing sets. On each of the 100 testing sets, we perform the following experiment: (1) split the 1100 samples up into 1000 training points and 100 validation points; (2) train a KNLMS model on a single training point; (3) after training on a single point, test the models prediction accuracy on the 100 validation points and calculate the mean squared error (MSE); and (4) repeat steps (2) and (3) until all training points are trained. After calculating the above steps on each of the 100 testing sets for float and fixed point, with the fractional lengths $FL = 8$ and $FL = 16$,

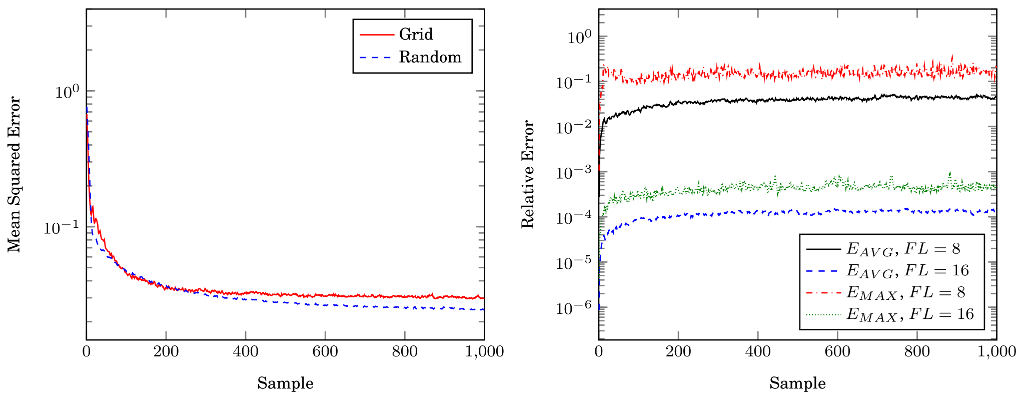

Fig. 9. Accuracy: grid search versus random search using float (left). Average relative and maximum relative error fixed point with different fractional widths compared to float (right).

we then calculate the average MSE at each training iteration, as well as average relative error and maximum relative error with respect to floating point. The left plot of Figure 9 shows the average MSE of our KNLMS configurations using the hyperparameters found by employing the above-mentioned methodology. Explicitly, the parameters $(\sigma, \eta, \epsilon, \mu_0)$ found were $0.85, 0.3, 0.001, 0.8$ for the grid search, and $0.5857, 0.1912, 0.3907, 0.7815$ for the random search. The values on the x-axis represent the current training iteration and the y-axis represents the average MSE over the validation set. On the right, the relative MSE is shown between float and fixed point with $FL = 8$ and $FL = 16$. Bit-accurate Vivado HLS C simulations were used to obtain this figure. When $FL = 16$, it is clear that float and fixed point produce almost identical results with less than 1/1,000 relative error for all data points and, on average, 1/10,000 relative error. While when $FL = 8$, the average relative error becomes over 1/100 and the maximum relative error over 1/10. In our experiments, random search resulted in a lower error than grid search, indicating that a better hyperparameter set was found. This observation is consistent with other studies that found random search to often be superior to grid search (Bergstra and Bengio 2012). Based on Figure 9, 16b are enough for Fixed fraction width to produce almost identical learning accuracies to Float. In this article, we employed a dictionary size of $N = 16$ for our Float design due to resource limitations. Since Fixed uses less area, the same FPGA can support a larger dictionary size of $N = 32$. In some cases, it may be possible that Fixed can achieve improved accuracy by employing a larger dictionary.

We end this section by noting that these results are data dependent and it is not possible to make any generalisations for other datasets. Moreover, insight into how precision requirements, hyperparameter values, and dictionary size affect the quality of predictions is a research problem beyond the scope of this work and, in the opinion of the authors, not currently tractable for nonlinear KAFs.

4.3 KNLMS Processor Performance

A comparison of performance with other KAF implementations is challenging since previous work implemented different algorithms. SW-KRLS (Van Vaerenbergh et al. 2006) requires a matrix inversion per update and has $O(N^2 + NM)$ time complexity. This is in contrast to KNLMS, which is $O(NM)$ and uses stochastic gradient descent techniques (Liu et al. 2011). Moreover, one should be careful in comparing Altera and Xilinx LUTs and DSP blocks, as they are different. Nevertheless, a summary of previous online KAF implementations of which we are aware is presented in

Table 4. Comparison of Online Kernel Method Implementations

Implementation	Algorithm	Device	M	N	DSPs	LUTs	BRAM	Freq MHz	Time ns	Slowdown rel. to Float
Naïve	KNLMS	Xilinx XC7VX485-2	8	16	12	4550	8	96.7	7,829	2,462
Float	KNLMS	Xilinx XC7VX485-2	8	16	1267	150,494	145	314	3.18	1
Fixed	KNLMS	Xilinx XC7VX485-2	8	16	210	53,273	138	286	3.50	1.1
System (Float)	KNLMS	VC707 dev board	8	16	1272	142,900	229	250	4	1.3
System (Fixed)	KNLMS	VC707 dev board	8	16	226	54,689	230	125	8	2.5
CPU (C)	KNLMS	Intel Xeon E5-2670	8	16	-	-	-	2,600	41	12.89
GPU (CUDA)	KNLMS	Nvidia GRID K520	8	16	-	-	-	800	11	3.46
Pang et al. (2013)	SW-KRLS	Altera Stratix V 5SGXEA7C2	7	16	30	41,476	227	237	9,000	2,830

Table 4. Clearly, the different versions of our KNLMS processor have much higher throughput when compared to the other implementations.

The CPU (C) and GPU (CUDA) are versions of KNLMS with a parallelised parameter search implemented using C/CUDA for CPU/GPU, respectively. For the CPU (C) version, many linear algebra libraries were tested, including BLAS (Lawson et al. 1979), ATLAS (Whaley and Petitet 2005) (with and without multithreading), OPENBLAS (Xianyi et al. 2014), and hand-coded routines. The fastest version was chosen to populate Table 4, which, to our surprise, was our hand-coded routines. We suspect that the vector sizes were not large enough to take advantage of cache blocking and the vectorisation methods provided by ATLAS/OPENBLAS, while aggressive compiler optimizations caused the hand-coded routines to be inlined, removing the function call overhead associated with an external library. A parallel parameter search was implemented using OMP on the outer loop described in Algorithm 2 (the normal grid search). Similar to the CPU (C) code, the GPU (CUDA) implements the KNLMS parameter search loop described in Algorithm 2 (the normal grid search). Each GPU thread implements KNLMS training on a different set of parameters in the parameter search. For both CPU and GPU parameter searches, great care was taken to ensure that as many variables as possible were private to each thread. Also, for both CPU and GPU parameter searches, the number of parameters and the length of the training set were scaled to find the best average time per prediction. As stated in Section 2, Pang et al. (2013) implement a different kernel adaptive filtering algorithm using a microcoded vector processor rather than a fully parallel, fully pipelined implementation described herein. The goal of the work from Pang et al. (2013) is to optimise latency rather than throughput and, as such, their performance is much lower than ours. However, their work constitutes the highest performing FPGA implementation of a KAF available in the literature and, therefore, is included in Table 4.

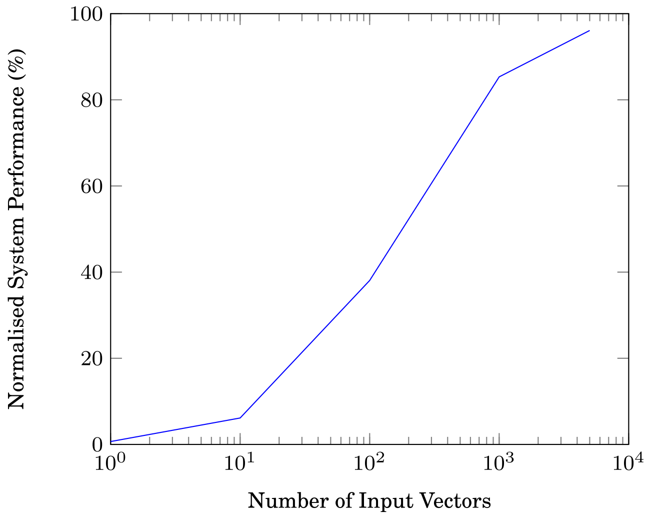

Fig. 10. Plot illustrating system performance of the KNLMS processor versus training set size.

4.4 System Performance

RIFFA 2.2.0 (Jacobsen et al. 2012) was used to provide high-speed data communication between our host computer and the FPGA via a PCIe GEN2 bus. The VC 707 board used supports a maximum bandwidth 4GB/s for unidirectional transfers; RIFFA could reach around 80% of this value. However, high bandwidth can only be achieved if sufficient data is sent to amortise transaction overheads. The optimised interface requires modest I/O bandwidth because each input vector is used $n_h = 256$ times before the next one is required.

Using the optimised system interface, with both the KNLMS Float core and PCI interface operating at 250MHz, the same MG-30 benchmark set samples were trained over a parameter space of $n_h = 256$ values. Figure 10 shows the performance as a percentage of the peak performance, where $f = 250MHz$ is the operating frequency, while varying the training set size.

While the design in Fraser et al. (2015) only achieved 23% of the highest achievable for the core, our new design can achieve full performance for training sets of more than 1,000 input vectors. The main sources of inefficiencies in Fraser et al. (2015) can be attributed to turnaround time of the PCIe bus (i.e., transfers were made in small blocks to avoid filling the FIFO and the result was read into the host before the next block was sent); these were overcome in the present design by calculating the sums of errors in the FPGA, rather than the host. This significantly reduced the amount of data that needed to be sent from the FPGA to the host. The entire dataset could be sent in a single transfer without filling the FIFOs, as detailed in Section 3.6.

Note that the system implementation used a different clock constraint in Vivado HLS in order to meet the timing requirements of the 250MHz clock on the VC707 dev board. When synthesised without the PCI interface, the Fixed and Float were capable of clock frequencies of 314MHz and 286MHz, respectively. However, inclusion of the RIFFA core caused the maximum frequencies to drop to 250MHz and 125MHz, respectively. Separate clocks for the core and interface should support a higher KNLMS core clock rate and a further improvement in throughput.

4.5 Scaling the KNLMS Core

In all previous sections, we considered only a fully parallel, fully spatialised implementation of the KNLMS core. In this section, we consider how the core will scale to larger FPGAs and also larger

Table 5. Scaling the KNLMS Core to Larger FPGAs and Larger Problem Sizes

Implementation	Device	M	N	DSPs	LUTs	Freq MHz	II	Latency	Time ns
Float	Xilinx XC7VX485-2	8	16	126	39,351	283	16	331	3.53
Float	Xilinx XC7VX485-2	8	32	247	77,195	283	16	475	3.53
Float	Xilinx XC7VX485-2	8	64	441	175,031	283	16	772	3.53
Float	Xilinx XC7VX485-2	8	128	882	394,130	283	16	1348	3.53
Float	Xilinx XC7VX485-2	8	256	1769	977,716	283	16	2500	3.53
Float	Xilinx XC7VX485-2	8	32	222	70,924	246	32	483	4.06
Float	Xilinx XC7VX485-2	8	64	439	138,254	246	32	771	4.06
Float	Xilinx XC7VX485-2	8	128	405	291,576	246	32	1367	4.06
Float	Xilinx XC7VX485-2	8	256	808	682,692	246	32	2519	4.06
Float	Xilinx XC7VX485-2	8	64	96	112,487	246	64	994	4.06
Float	Xilinx XC7VX485-2	8	128	216	224,675	240	64	1386	4.17
Float	Xilinx XC7VX485-2	8	256	432	540,213	240	64	2538	4.17
Device	Xilinx XC7VX485-2	-	-	2800	303,600	-	-	-	-

problem sizes. In order to handle larger problem sizes, we fold the design over the maximum dictionary size, N. We use initiation intervals (IIs) of 16, 32, and 64 while scaling the maximum dictionary size. The performance and area estimates shown in Table 5 were created using presynthesis results from Vivado HLS. For the most part, increasing the II by a factor of 2 allows a dictionary that is twice as large to be accommodated for a similar area cost. Surprisingly, for high initiation intervals and dictionary sizes, the DSP usage is lower for the same N/II ratio. In the case of the Xilinx XC7VX485-2 device, the largest problem that can fit based on Table 5 is $N = 128$ at a folding factor of 64.

5 CONCLUSION

A fully pipelined FPGA implementation of hyperparameter search for KNLMS was presented that achieves higher performance than any previously reported design. Pipeline stages are filled with multiple independent tasks corresponding to different machine-learning parameter values, allowing high utilisation of resources. This work demonstrated the feasibility of performing parameter search at nanosecond periods and opens the way for Big Data applications that were previously computationally intractable. Our PCI system achieves a 9.9/2.6× speedup over a CPU/GPU implementation, respectively.

Future work will focus on techniques to further increase parallelism, explore precision trade-offs, reduce the latency of the design, and to further explore how to accelerate larger problem sizes. Another possible use case for this design is to process high-bandwidth, independent streams of data. The system can process 8×32bcontinuously at 250MHz, equating to a throughput of 64Gbps.

REFERENCES

Nikolaos Alachiotis and Alexandros Stamatakis. 2011. FPGA Optimizations for a Pipelined Floating-Point Exponential Unit. In *Proceedings of the 7th International Symposium on Reconfigurable Computing: Architectures, Tools and Applications (ARC 2011), Belfast, UK, March 23-25, 2011.* Springer, Berlin, Heidelberg, 316–327. DOI:http://dx.doi.org/10.1007/978-3-642-19475-7_34

Davide Anguita, Luca Carlino, Alessandro Ghio, and Sandro Ridella. 2011. A FPGA core generator for embedded classification systems. *Journal of Circuits, Systems and Computers* 20, 02, 263–282. DOI: http://dx.doi.org/10.1142/S0218126611007244 arXiv:http://www.worldscientific.com/doi/pdf/10.1142/S0218126611007244

James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13, 281–305.

Badong Chen, Songlin Zhao, Pingping Zhu, and José Carlos Principe. 2012. Quantized kernel least mean square algorithm. *IEEE Transactions on Neural Networks and Learning Systems* 23, 1, 22–32.

Badong Chen, Nanning Zheng, and Jose C. Principe. 2013. Survival kernel with application to kernel adaptive filtering. In *The 2013 International Joint Conference on Neural Networks (IJCNN'13)*. IEEE, 1–6.

Marc Claesen and Bart De Moor. 2015. Hyperparameter search in machine learning. In *The XI Metaheuristics International Conference (MIC'15)*.

David Cox and Nicolas Pinto. 2011. Beyond simple features: A large-scale feature search approach to unconstrained face recognition. In *2011 IEEE International Conference on Automatic Face & Gesture Recognition and Workshops (FG'11)*. IEEE, 8–15.

J. Detrey and F. de Dinechin. 2005. A parameterized floating-point exponential function for FPGAs. In *Proceedings of 2005 IEEE International Conference on Field-Programmable Technology*. 27–34. DOI: http://dx.doi.org/10.1109/FPT.2005.1568520

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

N. J. Fraser, D. J. M. Moss, JunKyu Lee, S. Tridgell, C. T. Jin, and P. H. W. Leong. 2015. A fully pipelined kernel normalised least mean squares processor for accelerated parameter optimisation. In *25th International Conference on Field Programmable Logic and Applications (FPL'15)*. 1–6.

Matthew Jacobsen, Yoav Freund, and Ryan Kastner. 2012. RIFFA: A reusable integration framework for FPGA accelerators.. In *IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'12)*. IEEE, 216–219. http://dblp.uni-trier.de/db/conf/fccm/fccm2012.html/#JacobsenFK12.

E. Jamro, K. Wiatr, and M. Wielgosz. 2007. FPGA implementation of 64-bit exponential function for HPC. In *International Conference on Field Programmable Logic and Applications (FPL'07)*. 718–721. DOI: http://dx.doi.org/10.1109/FPL.2007.4380753

Jyrki Kivinen, Alexander J. Smola, and Robert C. Williamson. 2004. Online learning with kernels. *IEEE Transactions on Signal Processing* 52, 8, 2165–2176.

Neil Lawrence, Matthias Seeger, and Ralf Herbrich. 2003. Fast sparse Gaussian process methods: The informative vector machine. In *Proceedings of the 16th Annual Conference on Neural Information Processing Systems*. 609–616.

C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software* 5, 3, 308–323. DOI: http://dx.doi.org/10.1145/355841.355847

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

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

Abhinandan Majumdar, Srihari Cadambi, Michela Becchi, Srimat T. Chakradhar, and Hans Peter Graf. 2012. A massively parallel, energy efficient programmable accelerator for learning and classification. *ACM Transactions on Architecture and Code Optimization* 9, 1, Article 6, 30 pages. DOI: http://dx.doi.org/10.1145/2133382.2133388

Yeyong Pang, Shaojun Wang, Yu Peng, Nicholas J. Fraser, and Philip H. W. Leong. 2013. A low latency kernel recursive least squares processor using FPGA technology. In *FPT*. 144–151.

M. Papadonikolakis and C. Bouganis. 2008. A scalable FPGA architecture for non-linear SVM training. In *International Conference on ICECE Technology (FPT'08)*. 337–340. DOI: http://dx.doi.org/10.1109/FPT.2008.4762412

Nicolas Pinto, David Doukhan, James J. DiCarlo, and David D. Cox. 2009. A high-throughput screening approach to discovering good forms of biologically inspired visual representation. *PLOS Computational Biology* 5, 11, 1–12. DOI: http://dx.doi.org/10.1371/journal.pcbi.1000579

John Platt and others. 1998. Sequential minimal optimization: A fast algorithm for training support vector machines. https://www.microsoft.com/en-us/research/publication/sequential-minimal-optimization-a-fast-algorithm-for-training-support-vector-machines/.

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

Robin Pottathuparambil and Ron Sass. 2009. A parallel/vectorized double-precision exponential core to accelerate computational science applications. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'09)*. ACM, New York, NY,285–285. DOI: http://dx.doi.org/10.1145/1508128.1508198

Carl E. Rasmussen and Christoper K. I. Williams. 2006. *Gaussian Processes for Machine Learning*. MIT Press, Cambridge, MA.

Xiaowei Ren, Pengju Ren, Badong Chen, Tai Min, and Nanning Zheng. 2014. Hardware implementation of KLMS algorithm using FPGA. In *2014 International Joint Conference on Neural Networks (IJCNN'14)*. IEEE, 2276–2281.

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

Bernhard Scholkopf and Alexander J. Smola. 2001. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond.* MIT Press, Cambridge, MA.

Matthias Seeger. 2000. Relationships between Gaussian processes, support vector machines and smoothing splines. *Machine Learning*).

Stephen Tridgell, Duncan J. M. Moss, Nicholas J. Fraser, and Philip H. W. Leong. 2015. Braiding: A scheme for resolving hazards in NORMA. In *Proceedings of the International Conference on Field Programmable Technology (FPT'15).* 136–143. DOI: http://dx.doi.org/10.1109/FPT.2015.7393140

Steven Van Vaerenbergh. 2012. Kernel Methods Toolbox KAFBOX: a Matlab benchmarking toolbox for kernel adaptive filtering. Retrieved October 1, 2017 at http://sourceforge.net/p/kafbox.

S. Van Vaerenbergh, J. Via, and I. Santamaria. 2006. A sliding-window kernel RLS algorithm and its application to nonlinear channel identification. In *2006 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'06).* Vol. 5. 789–792. DOI: http://dx.doi.org/10.1109/ICASSP.2006.1661394

R. Clint Whaley and Antoine Petitet. 2005. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience* 35, 2, 101–121. http://www.cs.utsa.edu/whaley/papers/spercw04.ps.

B. Widrow and M. E. Hoff Jr. 1960. Adaptive switching circuits. In *IRE WESCON Convention Record.* 96–104.

Maciej Wielgosz, Ernest Jamro, and Kazimierz Wiatr. 2008. Highly efficient structure of 64-bit exponential function implemented in FPGAs. In *Proceedings of the 4th International Workshop, Reconfigurable Computing: Architectures, Tools and Applications (ARC'08), London, UK, March 26-28, 2008.* Roger Woods, Katherine Compton, Christos Bouganis, and Pedro C. Diniz (Eds.). Springer, Berlin, 274–279.

James H. Wilkinson. 1994. *Rounding Errors in Algebraic Processes.* Dover Publications, Incorporated, Mineola, NY.

Zhang Xianyi, Wang Qian, and Zaheer Chothia. 2014. Openblas. Retrieved October 1, 2017 from http://xianyi.github.io/OpenBLAS.

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, 113–131.

M. Yukawa. 2012. Multikernel adaptive filtering. *IEEE Transactions on Signal Processing* 60, 9, 4672–4682. DOI: http://dx.doi.org/10.1109/TSP.2012.2200889

Received April 2016; revised January 2017; accepted June 2017