A Low Latency Kernel Recursive Least Squares Processor using FPGA Technology

Yeyong Pang*, Shaojun Wang*, Yu Peng*, Nicholas J. Fraser† and Philip H.W. Leong†
* Harbin Institute of Technology
NO.92 West Da-zhi Street Harbin, 150001, China
† School of Electrical and Information Engineering
Building J03, The University Of Sydney, 2006, Australia

***Abstract*—The kernel recursive least squares (KRLS) algorithm performs non-linear regression in an online manner, with similar computational requirements to linear techniques. In this paper, an implementation of the KRLS algorithm utilising pipelining and vectorisation for performance; and microcoding for reusability is described. The design can be scaled to allow tradeoffs between capacity, performance and area. Compared with a central processing unit (CPU) and digital signal processor (DSP), the processor improves on execution time, latency and energy consumption by factors of 5, 5 and 12 respectively.**

I. Introduction

In machine learning traditional linear prediction techniques are well understood, and methods for their efficient solution have been developed. Many real-world applications are better modelled using non-linear techniques, which often have high computational requirements. Mercer kernel techniques utilise linear methods in a non-linear feature space and combine the advantages of both. Such *kernel methods* are considered one of the major advances in machine learning research [1], [2].

Commonly used kernel methods include the support vector machine (SVM), Gaussian processes and regularization networks [3]. These are batch-based, and a global optimisation is conducted over all input exemplars to create a model. In contrast, *online methods*, such as the kernel recursive least squares (KRLS) algorithm [4], [5], update the state in a recursive and incremental fashion upon receiving a new exemplar. Although not as extensively studied as batch-methods, online approaches are advantageous when latency is critical. Reconfigurable computing can be used as another, orthogonal means to reduce latency. We show that these two techniques can be combined in a harmonious fashion to perform prediction with exceptionally low latency.

In this paper, we describe a microcoded implementation of the KRLS algorithm. Although not explored in this paper, microcode enables the processor to be easily optimised for other algorithms at both the instruction set and datapath levels. The contributions of the work include:

- A novel, scalable architecture for kernel methods which has advantages in terms of reusability, extensibility and performance over a straightforward hardware description language (HDL) approach.
- The first reported FPGA-based online KRLS implementation. To the best of our knowledge, this implementation has the lowest latency of any reported kernel method to date.
- Results describing the performance, latency, power and area of the design in terms of prediction accuracy and performance, and a detailed performance comparison with central processing unit (CPU) and digital signal processor (DSP) approaches.

We believe that reconfigurable computing will prove to be an enabling technology for applications where minimal response time is critical. One important factor is that FPGAs allow the data converters and/or network interface controller to be tightly integrated with high speed processing. As examples, it can lead to improved speed in predicting events in a wide variety of application domains including algorithmic trading [6]; diagnostic and prognostic monitoring of machines [7]; and electricity blackout prevention [8].

The remainder of the paper is organised in the following manner. In Section II, previous work in related areas is reviewed. In Section III, the KRLS algorithm is described. The proposed architecture of the KRLS implementation is introduced in Section IV, and an analysis of its performance and power consumption are given in Section V along with comparisons with CPU and DSP based approaches. Finally, conclusions are drawn in Section VI.

II. Background

In this section, a review of previous implementations of relevant machine learning algorithms on FPGAs is given. Soft vector processors are briefly reviewed in Section IV.

Module generators are common in reconfigurable computing. These allow a family of parameterised designs to be produced rather than a single one. Anguita et. al. [9] described an FPGA-based fixed-point core generator for SVMs which allows sub-modules with different speed, resource and accuracy tradeoffs to be included. Papadonikolakis and Bouganis [10] developed a scalable SVM module generator which allows for the use of different kernel types, and uses different numbers of parallel tiles to optimise performance. The design was partitioned into fixed and floating-point parts to achieve high speed without sacrificing accuracy. Majumdar et. al. [11] described the many-core MAPLE architecture which was designed to accelerate a number of learning and classification problems, including SVMs. Vector processing elements in a two dimensional grid were used to perform linear algebra. On-chip memory used

978-1-4799-2198-0/13/$31.00 ©2013 IEEE

for in-processor processing was combined with independent off-chip memory banks.

Lin et. al. [12] developed a Baysian computing machine (BCM) using floating-point arithmetic to evaluate probabilistic networks. Deep pipelines were used to achieve high throughput, and a novel scheme was devised to efficiently schedule memory accesses and processing nodes. A system with 16 processing nodes achieved an average $80/15\times$ speedups over a CPU/GPU implementation respectively. This system can be applied to Bayesian problems of different size and graph topology without changing the FPGA implementation of the BCM.

Shan et. al. [13] developed a map-reduce framework and applied it to the acceleration of the RankBoost algorithm using float-point arithmetic. They also described how their work could be applied to the PageRank and SVM algorithms.

Perhaps most similar architecturally to this work, is an implementation of the Restricted Boltzman Machine (RBM) by Ly [14]. A microprogrammed controller was employed, and a new technique to implement transcendental functions in a highly pipelined manner by combining lookup tables with linear interpolators introduced. The system used 32-bit fixed point arithmetic and could be scaled by allowing multiple FPGAs to be connected via MPI. Virtualisation through time multiplexing was employed to allow larger problems to be solved.

In this work, a parallel datapath is combined with microcoded control logic to produce a vector processor. While both Ly [14] and Majumdar [11] targetted maximum performance in batch learning tasks, ours is designed for single-FPGA, floating-point embedded applications in which minimising latency and compactness are the key design goals. Similar architectures have been applied to the acceleration of linear algebra problems, utilising both spatial parallelism and pipelining to achieve high performance. To better accelerate kernel methods, hardware support for kernel evaluation is included. Our designs could be considered complementary to previous work in that module generators, scheduling, map-reduce, MPI, etc can be combined with our architecture.

In terms of data representation, while the performance benefits of fixed-point and reduced precision floating-point implementations are undeniable, we advocate single precision floating-point since it facilitates verification and is more robust in the presence of ill-conditioned data. The proposed architecture could be easily modified for fixed-point and mixed fixed-floating point arithmetic.

General-purpose graphics processing unit (GPGPUs) are an alternative acceleration technology to FPGAs, CPUs and DSPs. They offer massive amounts of computing power, but are optimised for throughput rather than latency. For a 4-byte transfer, CPU-to-GPGPU and GPGPU-to-CPU latency has been measured as $40.4\mu s$ and $41.9\mu s$ respectively [15]. Since the round trip transfer time alone is similar to the FPGA and GPGPU total latency including processing, we concluded that GPGPUs were not suitable for latency-critical applications and hence were not included in this study.

III. The Kernel Recursive Least Squares Algorithm

Linear regression is commonly used in many machine learning problems. In the case of online learning, the recursive least squares (RLS) algorithm provides a computationally efficient way to perform linear regression. For non-linear machine learning problems, the RLS algorithm can be modified to make use of a non-linear kernel function. The resulting kernel recursive least squares (KRLS) algorithm, provides a computationally efficient means to apply online learning to non-linear processes.

In this section, brief summaries of linear regression, the RLS algorithm and the KRLS algorithm are provided. A variation on the KRLS, the sliding window KRLS (SW-KRLS), is also described as it provides tracking capability to the KRLS algorithm along with explicit bounds on the computational cost of the algorithm.

A. Linear Regression

Let $\mathbf{X} = \{x_1, x_2, ..., x_n\} \in \mathbb{R}^{N\times M}$ be the input training set of N observations of dimension M, and the target be $\mathbf{y} \in \mathbb{R}^N$. Linear regression attempts to find the optimal vector $\mathbf{h} \in \mathbb{R}^{\mathbf{M}}$ which satisfies $J = \min ||\mathbf{y} - \mathbf{Xh}||^2$.

If $(\mathbf{X}^T\mathbf{X})$ is not singular, a direct solution can be computed for $\mathbf{h}$ as follows:

$$\mathbf{h} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}\mathbf{y} \quad . \tag{1}$$

The dual representation can be obtained by pre-multiplying Eq. (1) by the identity $(\mathbf{X}^T\mathbf{X})(\mathbf{X}^T\mathbf{X})^{-1}$ to obtain $\mathbf{h} = \mathbf{X}^T\alpha$ making $\mathbf{h} = \sum_{i=1}^{N} \alpha_i x_i$ a linear combination of the training set.

For an online problem, not all training data is known and the solution must be recalculated as new observations are provided. The recursive least squares (RLS) algorithm provides a direct and computationally efficient, $O(M^2)$, solution for $\mathbf{h}$ for every new sample, without needing to recalculate the matrix inverse. It uses the *Matrix Inversion Lemma* [16], $(\mathbf{A} + x_n x_n^T)^{-1} = \mathbf{A}^{-1} - \frac{\mathbf{A}^{-1}x_n x_n^T\mathbf{A}^{-1}}{1+x_n^T\mathbf{A}^{-1}x_n}$, where x_n is the latest vector of observations and $\mathbf{A} = \mathbf{X}_{n-1}^T\mathbf{X}_{n-1}$ where $\mathbf{X}_{n-1}$ is given by $\mathbf{X}_{n-1} = \{x_1, x_2, ..., x_{n-1}\}$.

B. Kernel Methods

A common way to adapt linear regression to non-linear problems is to apply a non-linear mapping, $\Phi(x) \in \mathbb{R}^M \rightarrow \mathbb{F}$, to the input data. Linear regression can then applied to this new data to find $\tilde{\mathbf{h}}$. The kernel recursive least squares (KRLS) algorithm, described in [4], uses the "kernel trick" to compute an inner product in the feature space without explicitly computing the feature vectors.

Let $\mathbf{K} = \tilde{\mathbf{X}}\tilde{\mathbf{X}}^T$ be the kernel matrix where $K_{i,j}$, the entry corresponding to the ith row and j^{th} column of the kernel matrix, is given by the kernel function, $\kappa(x_i, x_j)$. With this new representation, the cost function can be updated to: $J' = \min ||\mathbf{y} - \mathbf{K}\alpha||^2$, where α is a $N \times 1$ vector of weights.

A common kernel function is the Gaussian kernel $\kappa(x_i, x_j) = \exp(-\frac{||x_i - x_j||^2}{2\sigma^2})$. Other kernel functions include the polynomial kernel, the hyperbolic tangent kernel and the Laplacian kernel. The Laplacian kernel, $\kappa(x_i, x_j) = 2^{-\gamma||x_i - x_j||}$, was used in [17] since it can be implemented in hardware efficiently.

The minima of J' using the dual representation is given by $\alpha = \mathbf{K}^{-1}\mathbf{y}$. A problem with many kernel based machine learning functions is that the computational complexity of these algorithms scale superlinearly with the number of training samples [4]. Traditionally this has limited their use for online machine learning applications. KRLS uses a recursive method to compute $\mathbf{K}^{-1}$ efficiently for each new sample with a computational complexity of $O(N^2)$.

C. Bounding the Computational Cost per Sample

There are several methods for minimising the computational cost of the KRLS algorithm. In [4], a sparsification procedure is employed to prevent any samples from being admitted to the training set if they can be represented by linear combinations of the previous samples. This helps to reduce the computational complexity but does not bound the computational requirements. The SW-KRLS algorithm [5] and the fixed budget KRLS (FB-KRLS) algorithm [18] set a limit, N', to the number of training samples which can be stored in the dictionary, enforcing a bound on the computation cost.

The SW-KRLS algorithm removes any training samples that are not in a fixed time window, N'. Given a stream of input/output pairs, $\{(x_1, y_1), (x_2, y_2)...\}$, at training sample n, the input matrix becomes $\mathbf{X}_n = [x_n, x_{n-1}, ..., x_{n-N'+1}]$ and the output vector becomes $\mathbf{y}_n = [y_n, y_{n-1}, ..., y_{n-N'+1}]$.

In order to calculate α_n, the nth estimate of the weights, $\mathbf{K}_n^{-1}$, the inverse kernel matrix can be calculated using $\mathbf{K}_{n-1}^{-1}$ and $\mathbf{K}_n$, the nth kernel matrix. $\mathbf{K}_n$ can be calculated as follows:

$$\hat{\mathbf{K}}_n = \begin{bmatrix} \mathbf{K}_{n-1} & k_n(x_n) \\ k_n(x_n)^T & k_{nn} + c \end{bmatrix} \tag{2}$$

where $k_n(x_n) = [\kappa(x_{n-N'+1}, x_n), ..., \kappa(x_{n-1}, x_n)]^T$, $k_{nn} = \kappa(x_n, x_n)$, c is a regularization constant and $\mathbf{K}_{n-1}$ is the kernel matrix calculated from the previous training sample. $\mathbf{K}_n$ is defined as follows:

$$\mathbf{K}_n = \begin{bmatrix} k_{n-N,n-N} + c & p^T \\ p & \hat{\mathbf{K}}_{n-1} \end{bmatrix} \tag{3}$$

where $p = [\kappa(x_{n-N'}, x_{n-N'+1}), ..., \kappa(x_{n-N'}, x_n)]^T$ and $k_{n-N',n-N'} = \kappa(x_{n-N'}, x_{n-N'})$. $\hat{\mathbf{K}}_n^{-1}$ can then be calculated using:

$$\hat{\mathbf{K}}_n^{-1} = \begin{bmatrix} \mathbf{K}_{n-1}^{-1}(\mathbf{I} + \mathbf{b}\mathbf{b}^T\mathbf{K}_{n-1}^{-1T}g) & -\mathbf{K}_{n-1}^{-1}\mathbf{b}g \\ -(\mathbf{K}_{n-1}^{-1}\mathbf{b})^T g & g \end{bmatrix} \tag{4}$$

where $\mathbf{b}$ is given by $k_n = [\mathbf{b}\ d]^T$ and g is given by $g = (d - \mathbf{b}^T\mathbf{K}_{n-1}^{-1}\mathbf{b})^{-1}$.

$\mathbf{K}^{-1}$ is then calculated using Eq. (5)

```
Initialise K_0 as (1 + c)I and K_0^{-1} as I/(1 + c).
for n = 1, 2, ... do
    Get K̂_n from K_{n-1} with Eq. (2)
    Calculate K̂_{n-1}^{-1} from Eq. (4)
    Get K_n from Eq. (3)
    Calculate K_n^{-1} from Eq. (5)
    Calculate α_n using α = K_n^{-1} Y_n
end for
```

Fig. 1. Pseudocode: a training step for the SW-KRLS algorithm.

$$\mathbf{K}_n^{-1} = \mathbf{G} - \mathbf{f}\mathbf{f}^T/e \tag{5}$$

where $\mathbf{G}$, e and $\mathbf{f}$ are given by:

$$\hat{\mathbf{K}}_n^{-1} = \begin{bmatrix} e & \mathbf{f}^T \\ \mathbf{f} & \mathbf{G} \end{bmatrix} \quad . \tag{6}$$

and $\mathbf{G}$ is a $(N'-1) \times (N'-1)$ matrix, $\mathbf{f}$ is a $(N'-1) \times 1$ vector and e is a scalar.

Other pruning techniques also been suggested including approximate linear dependency (ALD) [4], the surprise criterion [19] and error minimisation [20].

The SW-KRLS algorithm removes the oldest training pair from its dictionary when a new sample is encountered allowing it to track non-stationary processes. Psuedocode for the SW-KRLS algorithm, derived from [5] and [21] is provided in Figure 1.

Note that the algorithm described in this paper, along with previous works [5] and [18], uses a regularization parameter to help prevent overfitting. With regularization, the cost function becomes:

$$J'' = \min_{\tilde{h}} ||\mathbf{y} - \tilde{\mathbf{X}}\tilde{\mathbf{h}}||^2 + c||\tilde{\mathbf{h}}||^2 = \min_{\alpha} ||\mathbf{y} - \mathbf{K}\alpha||^2 + c\alpha^T\mathbf{K}\alpha \tag{7}$$

and the ideal solution of weights becomes $\alpha = (\mathbf{K} + c\mathbf{I})^{-1}\mathbf{y}$ where $\mathbf{I}$ is the identity matrix.

IV. Architecture

A. System Overview

Figure 2 shows a block diagram of the processor architecture. It is formed from a program counter (PC), microcode memory, vector memory, arithmetic logic unit (ALU) and data control unit. The program counter is a simple counter which automatically increments every cycle, and its output forms the microcode memory address. The branch address can be loaded into the program counter via the Branch instruction (Table II). Microcode memory is used for storage of the microprogram.

With some exceptions, such as [22], most well-known previous soft vector processors [23], [24], [25], [26], have not supported floating-point operations. Apart from those reviewed earlier, none have been optimised for machine learning applications. Similar to previous soft vector processors [23], [24], [25], [26], the KRLS processor architecture offers scalable

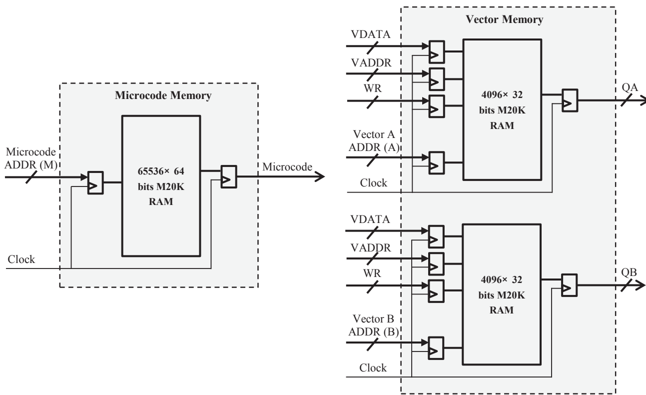

Fig. 3. Microcode and Vector memory architecture.

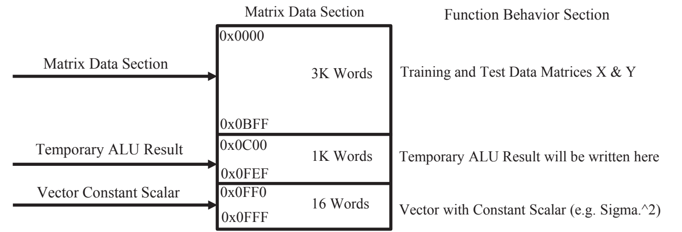

Fig. 4. Vector memory map.

performance and area via control of the number of vector lanes. Each independent datapath lane includes a vector lane and vector memory. The targeted FPGA accommodates up to 128 datapath lanes, which is larger than previous vector processors [23]. Since all vector memory is on-chip, as the number of lanes are increased, the maximum vector memory depth is reduced [24].

Since online machine learning problems typically require a modest amount of memory, an external memory interface is not included, and vector memory is directly connected to the arithmetic logic unit (ALU). A control unit is implemented as a finite state machine. It generates enable signals and write addresses to the vector memory with proper latency according to the Microcode, and can route data across vector lanes. It is also responsible for generating the CON signal which controls the function of the heterogeneous ALU, discussed in Section IV-C. When a branch instruction is issued, data control unit generates the Load signal and transfers vector address C to the branch address.

B. Memory Interface

A vector scratchpad memory is employed as introduced in VEGAS [23] and VINCE [25], to reduce load/store operations from memory. Furthermore, the microcode memory output directly drives the vector memory to eliminate the need for address registers [26].

Stratix V Embedded Memory Blocks are used for all memories. As illustrated in Figure 3, Microcode memory is configured as a 64K×64-bit single port ROM memory, and occupies 256 M20K block memories. As the vector processor uses single precision floating point, the data width is fixed to be 32 bit, vector memory is built from 512×32 bit RAM primitives. For a 128 lane vector processor, the maximum capacity of each vector scratchpad memory is 4096×32 bits, constructed from 7×M20K block RAMs.

The vector memory is used to store all intermediate data and constants. It is divided into 3 sections as illustrated in Figure 4. Training and test data is stored in the memory within the range 0x0000 and 0x0BFF, and temporary outputs from the ALU are stored in a region with starting address 0x0C00. Vector memory is initialised in the bitstream, the last 16 words of the memory being used to store constant vectors. Peripherals are memory mapped to the higher addresses of the vector memory of the heterogeneous lane (i.e. Vector memory 1).

C. Vector Lane Architecture

Figure 5 illustrates the ALU architecture. Each datapath lane includes two vector scratchpad memories, a single precision floating point adder/subtractor and a single precision multiplier. Lane widths are limited to powers of two to reduce hardware complexity. In addition to vector add, subtract and multiply operations, the architecture supports vector divide and exponentiation. All floating point operations are implemented using the Altera floating point library.

Division and exponentiation consume more resources than other floating point arithmetic modules, but are used less frequently. For a Stratix V device, a single precision exponential module utilizes $9\times$ 27-bit DSP blocks, a low latency single precision division utilizes $5\times$ 27-bit DSP blocks, while a single precision multiplier only utilizes a single 27-bit DSP block [27]. Thus if a divider and exponentiation module were included in each vector lane, resource utilization would increase significantly, and as such, only a 17 lane vector processor could be implemented, reducing overall performance.

Hence, divider and exponentiation modules are included only in the first ALU lane, which we call a heterogeneous ALU. Though the vector exponential and division operations require more cycles to complete, the reduction in resources allow for more lanes to be implemented, increasing overall performance. The S2VE and PVDOT function units are also implemented in the heterogeneous ALU lane. The S2VE function unit shares multiplexers with the exponential and division modules, and the PVDOT function module is a $\log_2(N)$ level fully pipelined single precision adder tree which accepts N parallel inputs.

With minor modifications, other functions such as sqrt, sin/cos, tan/atan, inverse, log and compare can be added. It is therefore easy to port other machine learning algorithms to our vector processor, this being one of our future objectives. Conversely, any unnecessary functional units can be excluded to save resources.

D. Microcode

The microcode format is summarised in Table I. The microcode is 64-bits wide, with all vector and microcode addresses being 16-bits. The 'BAK' bits are reserved for future use. The SW-KRLS MATLAB code from KAFBOX [21] was manually vectorised and converted to microcode.

Table II, describes the function of each opcode. The fetch & decode column represents how many clock cycles it takes

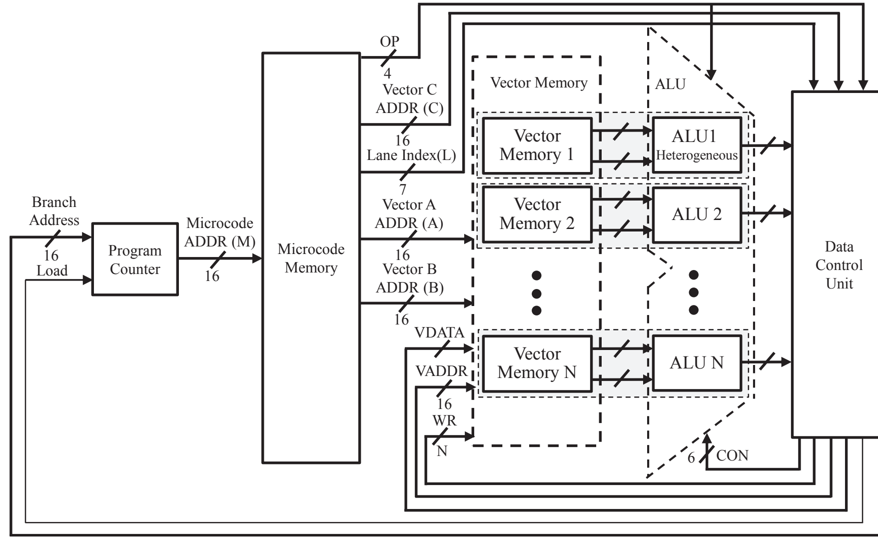

Fig. 2. Vector processor block diagram.

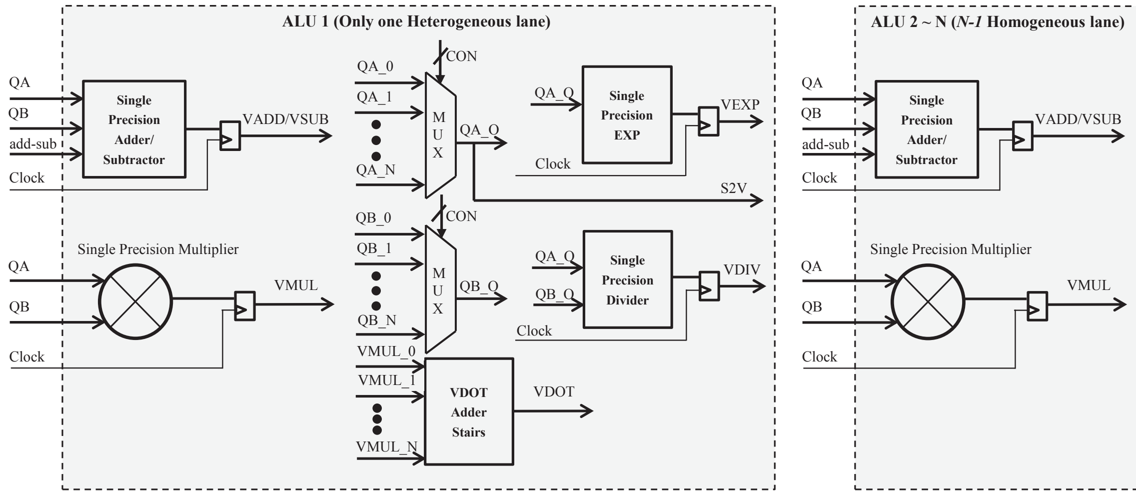

Fig. 5. Vector lane architecture.

to load a microinstruction and the input vectors from memory. The execute column represents the number of clock cycles used to calculate the result. The write back column represents how many clock cycles are used store the result. In the current implementation, no parallelism in the execution of adjacent instructions is exploited. The Lane Index, L, is a 7 bit value that specifies which lane of the vector memory is used to store the result for the PVDOT instruction. The vector memory for the ith lane is represented as VM_i, and two separate addresses (A and B) are used. Address C specifies the destination address. As can be seen in Figure 3, both write ports are driven with the same address and data signals (VADDR and VDATA), so identical data are written to both ports.

Microinstructions are either simple or parallel. Simple instructions include vector operations such as array multiply (VMUL) and vector addition (VADD). Parallel instructions (PVxxx) are functionally equivalent to N vector operations but take advantage of the pipelined ALU for improved performance, particularly for matrix operations. For example, as illustrated in Figure 6, PVADD overlaps N consecutive vector additions achieving a 14N/(N + 13) speedup over N VADD calls.

V. Results

An implementation of the KRLS vector processor was made using VHSIC Hardware Description Language (VHDL). Quartus II 13.0 was used for FPGA synthesis, place and route. The design was written in such a way that key parameters such as the memory sizes and number of vector lanes can be configured at compile time. The target platform was an Altera DE5 board populated with a medium size Stratix V 5SGXEA7C2. Although the results reported were for the DE5 board, it can be easily ported to different FPGA devices, vendors and floating-point formats. The current implementation assumes input and output data are present in vector memory as explained in Section IV-B. I/O is not currently supported, but can be implemented as a memory mapped device.

This section describes the KRLS processor performance, la-

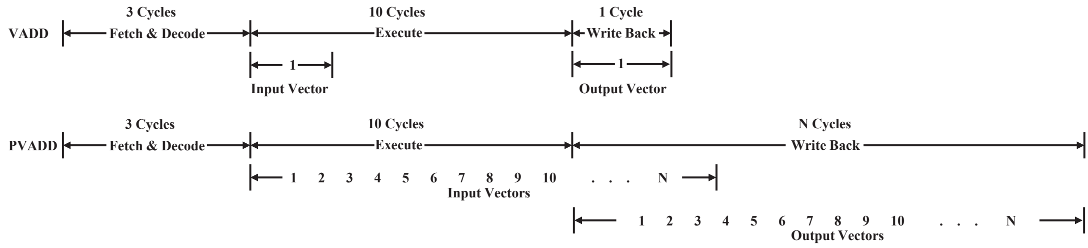

Fig. 6. Timing diagrams of VADD and PVADD showing the difference between simple and chained instructions.

TABLE I. MICROCODE FORMAT.

Function	Vector A address	Vector B address	Vector C address	Lane Index	BAK	Operation Code
Symbol	A	B	C	L	BAK	OP
Bits	16	16	16	7	5	4

TABLE II. MICROCODE INSTRUCTIONS (ALL i, jAND L INDEXES RANGE FROM 1 TO N).

Microcode (Opcode)	Description	Function	Fetch & Decode	Execute	Write Back	Total Cycles
NOP (0000)	No operation		1	-	-	1
BRANCH (0111)	Branch	$M = C$	3	1	-	4
VADD (0001)	Vector add	$VM_i[C] = VM_i[A] + VM_i[B]$	3	10	1	14
VSUB (0010)	Vector subtract	$VM_i[C] = VM_i[A] - VM_i[B]$	3	10	1	14
VMUL (0011)	Array multiply	$VM_i[C] = VM_i[A] \times VM_i[B]$	3	6	1	10
VDIV (0100)	Vector divide	$VM_i[C] = VM_i[A]/VM_i[B]$	4	14	N	N+18
VEXP (0110)	Vector exponentiation	$VM_i[C] = EXP(VM_i[A])$	4	17	N	N+21
S2VE (1000)	Clone a vector N times	$VM_i[C+j] = VM_j[A]$	4	0	N	N+4
PVADD (1001)	N× Vector add	$VM_i[C..C+N-1] =$ $VM_i[A..A+N-1] + VM_i[B..B+N-1]$	3	10	N	N+13
PVSUB (1010)	N× Vector subtract	$VM_i[C..C+N-1] =$ $VM_i[A..A+N-1] - VM_i[B..B+N-1]$	3	10	N	N+13
PVMUL (1011)	N× Vector multiply	$VM_i[C..C+N-1] =$ $VM_i[A..A+N-1] \times VM_i[B..B+N-1]$	3	6	N	N+9
PVDOT (0101)	N× Vector dot product	$VM_L[C..C+N-1] =$ $\sum_{i=0}^{N-1} VM_i[A..A+N-1] \times VM_i[B..B+N-1]$	3	6+ $10\log_2(\mathrm{N})$	N	$\mathrm{N}+9+$ $10\log_2(\mathrm{N})$

tency, power consumption and resource usage information with different dictionary sizes. For all experiments described in this section, the MG-30 Mackey-Glass [28] benchmark modelled by the differential equation $\frac{dx(t)}{dt} = -ax(t) + \frac{bx(t-\tau)}{1+x(t-\tau)^{10}}$ with ($a = 0.1, b = 0.2, \tau = 30$) is used. The SW-KRLS algorithm is implemented using a regularization parameter of $c = 0.01$ and the kernel parameter is $\sigma = 0.6$.

The KRLS processor uses single precision floating point arithmetic. This was compared with an open source double precision implementation, KAFBOX [21]. The difference in mean squared error (MSE) between KAFBOX and our processor was less than 0.07%. For a single prediction, the maximum relative error was less than 0.5%. We consider this negligible for our purposes and thus we chose single precision floating point for our implementation.

Note that an N lane processor is used to implement a sliding window length of $N-1$. An optimised C implementation was developed for the CPU, DSP and Nios II platforms. Although both single and multithreaded ATLAS [29] were tested for the linear algebra routines, the highest performance was achieved with a serial implementation. We believe this was because our matrix sizes were too small to benefit from multiple cores.

A. KRLS FPGA Resource Usage

Table III shows the resource usage of the vector processor and the Nios II processor on an Altera DE5 development board

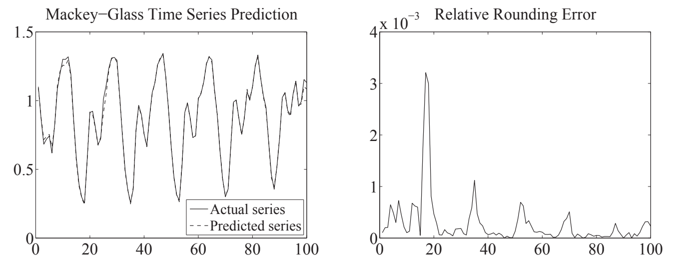

Fig. 7. SW-KRLS output using single precision floating point and a sliding window length of 127.

featuring an 5SGXEA7N2F45C2 FPGA. Due to the availability of M20K blocks, the maximum number of lanes that can be supported is 128. Table III also provides detailed information on the theoretical maximum frequency, F_{max}, for each different lane configuration of the vector processor. Obviously, a higher clock frequency can be attained using a faster part such as Stratix V C1 (fastest speed grade) device [24].

B. KRLS Vector Processor Performance

This test compares batch processing performance between all of our platforms. The CPU configuration is a desktop PC with the following specifications: an Intel Core i5-2400 CPU at 3.10GHz, 4GB of memory running Linux Ubuntu 12.10 LTS

TABLE III. VECTOR PROCESSOR RESOURCE USAGE

Lane Configuration (N)	M20K	27-bit DSP blocks	ALMs	F_{max}(MHz)
DE5 Available	2560	256	234720	-
VP @ 16	227	30	20738	236.96
VP @ 32	451	46	40705	197.68
VP @ 64	899	78	80981	170.58
VP @ 128	2058	142	155687	157.22
Nios II	320	3	4179	253.9

TABLE IV. KRLS VECTOR PROCESSOR PERFORMANCE

Algorithm	SW-KRLS			
Lane Configuration	16	32	64	128
KRLS Vector Processor(μs)	9	12	18	28
Intel CPU(μs)	2	9	33	141
DSP Processor(μs)	147	462	2127	54926
Nios II Soft Processor(μs)	1406	4275	13930	58428
Speedup VP/CPU	0.3	0.8	1.9	5.1
Speedup VP/DSP	17.0	39.3	122.8	2002.4
Speedup VP/Nios	162.9	363.8	804.3	2130.1

using GCC version 4.7.2. The DSP configuration is a Texas Instruments TMS320C6748 DSP development kit running at 300MHz. The soft processor is a Nios II configured for high performance and running at 250MHz. The Nios II processor is also configured to use a floating point co-processor for accelerating floating point operations. The vector processor uses 150MHz for the system clock frequency, f_s.

Table IV shows the vector processor performance against a CPU, a DSP and a Nios II for sliding window lengths between 15 and 127. For small sliding window sizes, the CPU outperforms all other platforms. The vector processor outperforms the CPU and DSP for $N \geq 64$, and for $N = 128$, the speedup is 5.

C. Energy Consumption

This section details the energy consumption of the Vector Processor, Nios II, DSP and CPU implementations. The power dissipation of the Nios II and vector processor implementations was measured using an Agilent U8001A power supply. The DSP power dissipation was measured using voltage and current measurements at the DC input jack on the board using a Fluke 15B digital multimeter. The CPU power dissipation was estimated using OS reported battery usage over a 15 minute test on a laptop using battery power with the LCD screen switched off. We also provide energy usage per training/prediction pair using $E = Pt$ where E is energy usage, P is power usage and t is the prediction time.

In this section, in order to select a system optimised for power dissipation, the CPU configuration is an HP Pavilion dv6 2120TX laptop with the following specifications: Intel Core i5 CPU M 430 @ 2.27GHz, 4GB of memory running Linux Mint Debian using GCC version 4.6.3.

From Table V it can be seen that the Nios II and DSP implementations require the least amount of power to operate. However, when the computational time is taken into consideration, the vector processor outperforms all other implementations in terms of energy consumption for sliding window lengths greater than 31. For $N = 128$, energy consumption of the FPGA is a factor of 12 lower than that of the CPU.

TABLE V. POWER CONSUMPTION

Computing Platform	Sliding Window Width	Power Dissipation (mW)	Execution Time (μS)	Energy Consumption (10^{-5}J)
KRLS Vector Processor	15	17480	9	16
	31	19080	12	23
	63	22320	18	39
	127	26880	28	75
Nios II	15	15120	1407	2127
	31		4275	6464
	63		13930	21063
	127		58428	88344
DSP	15	1975	147	29
	31	1975	462	91
	63	1975	2127	420
	127	2025	54926	11123
CPU	15	36818	4	13
	31		13	49
	63		46	170
	127		238	876

D. Latency Test

The previous tests in this paper have excluded I/O. We explore its effect on latency in this section. The vector processor test used the Terasic DE5 development board and the Terasic High-Speed A/D and D/A Development Kit, featuring Analog Devices AD9248 and AD9767 converters. The processing time can be acquired from Table IV. The input and output latency was estimated by examining the electrical characteristics in the datasheet for AD9248 and AD9767. In order to achieve minimum latency the ADCs and DACs were set to their maximum sample rates, 62.5MHz and 125MHz respectively. The $InputLatency_{I/O} = 0.12\mu s$, and the $OutoputLatency_{I/O} = 0.04\mu s$. For all tests described in the subsection, $N = 64$ was used.

The CPU test utilised the National Instruments (NI) 5781 Baseband Transceiver and 7954 FPGA modules which is capable of data acquisition at 100 MS/s. To minimise latency, single cycle memory read/writes were used instead of DMA transfers. The SW-KRLS C library was compiled as a virtual instrument module in NI LabVIEW. The latency was measured using an Agilent Oscilloscope.

The DSP latency was estimated by examining the electrical characteristics timing diagrams for the serial data interface in the datasheet for the TLV320AIC31, the ADC/DAC chip provided on the TMS320C6748 development board. The minimum data clock period was calculated by summing the data clock high period, low period, rise time and fall time. The minimum clock period was then multiplied by the lowest bitrate supported by the TLV320AIC31 to determine the I/O latency. The result is $Latency_{I/O} = (35ns + 35ns + 4ns + 4ns) * 16 = 1.248\mu s$. Using this method is likely to yield an optimistically low estimation of the I/O latency. Even so, using the highest audio sample rate (96kHz) supported on the development board, the resulting I/O latency of $10.42\mu s$ is still less than 1% of the processing time for a sliding window length of 63.

The results of the latency tests are shown in Table VI. The vector processor outperforms the other platforms in terms of latency due to its fast processing time and minimal I/O overhead. The DSP has low I/O latency but suffers from high processing latency. The CPU implementation is capable of providing low processing latency but suffers from high I/O latency during data acquisition due to PCIe bus and kernel overheads.

TABLE VI. LATENCY TEST RESULTS ($N = 64$)

Computing Platform	Input Latency (μS)	Processing Latency (μS)	Output Latency (μS)	Total (μS)
VP @ 64	0.12	17.62	0.04	17.78
CPU	7.32	78.50	7.32	93.14
DSP	1.25	2127	1.25	2129.50

VI. CONCLUSION

In this work, a microcoded vector processor for the acceleration of kernel based machine learning algorithms was presented. The architecture is optimised for dot product, matrix-vector multiplication and kernel evaluation, and features simplicity, programmability and compactness. Comparing an implementation of the SW-KRLS algorithm on the vector processor with $N \geq 64$ with a CPU and DSP, it was found that it outperforms the others in terms of execution time, latency, and energy consumption for $N \geq 64$.

The microcoded architecture presented herein is a flexible processor capable of implementing many algorithms. Future work includes the development of tools to assist in the creation of microcode, further optimising the processor to enhance parallelism, and its application to other machine learning problems.

ACKNOWLEDGMENT

The authors would like to thank the Altera University Programme for their support of our research and the donation of a Terasic DE5 board. This work was supported by the Australian Research Council under the Linkage Project LP110200413.

REFERENCES

[1] J. H. Friedman, "Recent advances in predictive (machine) learning," *Journal of Classification*, vol. 23, pp. 175–197, 2006.

[2] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg, "Top 10 algorithms in data mining," *Knowl. Inf. Syst.*, vol. 14, no. 1, pp. 1–37, Dec. 2007.

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

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

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

[6] J. W. Lockwood, A. Gupte, N. Mehta, M. Blott, T. English, and K. A. Vissers, "A low-latency library in FPGA hardware for high-frequency trading (HFT)," in *Hot Interconnects*. IEEE Computer Society, 2012, pp. 9–16.

[7] A. K. Jardine, D. Lin, and D. Banjevic, "A review on machinery diagnostics and prognostics implementing condition-based maintenance," *Mechanical Systems and Signal Processing*, vol. 20, no. 7, pp. 1483 – 1510, 2006.

[8] Y. Makarov, V. Reshetov, A. Stroev, and N. Voropai, "Blackout prevention in the United States, Europe, and Russia," *Proceedings of the IEEE*, vol. 93, no. 11, pp. 1942–1955, 2005.

[9] D. Anguita, L. Carlino, A. Ghio, and S. Ridella, "A FPGA core generator for embedded classification systems," *Journal of Circuits, Systems and Computers*, vol. 20, no. 02, pp. 263–282, 2011. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0218126611007244

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

[11] A. Majumdar, S. Cadambi, M. Becchi, S. T. Chakradhar, and H. P. Graf, "A massively parallel, energy efficient programmable accelerator for learning and classification," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, pp. 6:1–6:30, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2133382.2133388

[12] M. Lin, I. Lebedev, and J. Wawrzynek, "High-throughput bayesian computing machine with reconfigurable hardware," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '10. New York, NY, USA: ACM, 2010, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/1723112.1723127

[13] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "FPMR: Mapreduce framework on FPGA," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '10. New York, NY, USA: ACM, 2010, pp. 93–102. [Online]. Available: http://doi.acm.org/10.1145/1723112.1723129

[14] D. Ly, *A High-Performance, Reconfigurable Architecture for Restricted Boltzmann Machines*. Master of Applied Science Thesis: University of Toronto, 2009.

[15] R. Bittner and E. Ruf, "Direct GPU/FPGA communication via PCI express," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*. IEEE, 2012, pp. 135–139.

[16] N. Higham, *Accuracy and stability of numerical algorithms*. Siam, 1996, no. 48.

[17] D. Anguita, A. Ghio, S. Pischiutta, and S. Ridella, "A hardware-friendly support vector machine for embedded automotive applications," in *Neural Networks, 2007. IJCNN 2007. International Joint Conference on*. IEEE, 2007, pp. 1360–1364.

[18] S. Van Vaerenbergh, I. Santamaria, W. Liu, and J. Príncipe, "Fixed-budget kernel recursive least-squares," in *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1882–1885.

[19] W. Liu, I. Park, and J. C. Príncipe, "An information theoretic approach of designing sparse kernel adaptive filters," *Neural Networks, IEEE Transactions on*, vol. 20, no. 12, pp. 1950–1961, 2009.

[20] B. J. De Kruif and T. J. De Vries, "Pruning error minimization in least squares support vector machines," *Neural Networks, IEEE Transactions on*, vol. 14, no. 3, pp. 696–702, 2003.

[21] S. Van Vaerenbergh, "Kernel methods toolbox KAFBOX: a Matlab benchmarking toolbox for kernel adaptive filtering," Grupo de Tratamiento Avanzado de Señal, Departamento de Ingeniería de Comunicaciones, Universidad de Cantabria, Spain, 2012, software available at http://sourceforge.net/p/kafbox.

[22] J. Kathiara and M. Leeser, "An autonomous vector/scalar floating point coprocessor for fpgas," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, 2011, pp. 33–36.

[23] C. H.-Y. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. G. Lemieux, "Vegas: soft vector processor with scratchpad memory," in *FPGA*, J. Wawrzynek and K. Compton, Eds. ACM, 2011, pp. 15–24.

[24] P. Yiannacouras, J. Steffan, and J. Rose, "Portable, flexible, and scalable soft vector processors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 20, no. 8, pp. 1429–1442, 2012.

[25] A. Severance and G. Lemieux, "Venice: A compact vector processor for FPGA applications," in *FPT*. IEEE, 2012, pp. 261–268.

[26] J. Yu, C. Eagleston, C. H.-Y. Chou, M. Perreault, and G. Lemieux, "Vector processing as a soft processor accelerator," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 2, pp. 12:1–12:34, Jun. 2009. [Online]. Available: http://doi.acm.org/10.1145/1534916.1534922

[27] "Floating-point megafunctions user guide," Altera Corp, 101 Innovation Drive, San Jose, CA 95134.

[28] M. C. Mackey, L. Glass *et al.*, "Oscillation and chaos in physiological control systems," *Science*, vol. 197, no. 4300, pp. 287–289, 1977.

[29] R. C. Whaley and A. Petitet, "Minimizing development and maintenance costs in supporting persistently optimized BLAS," *Software: Practice and Experience*, vol. 35, no. 2, pp. 101–121, February 2005, http://www.cs.utsa.edu/~whaley/papers/spercw04.ps.