A Microcoded Kernel Recursive Least Squares Processor Using FPGA Technology

YEYONG PANG, SHAOJUN WANG, YU PENG, and XIYUAN PENG,
Harbin Institute of Technology
NICHOLAS J. FRASER and PHILIP H. W. LEONG, The University of Sydney

Kernel methods utilize linear methods in a nonlinear feature space and combine the advantages of both. Online kernel methods, such as kernel recursive least squares (KRLS) and kernel normalized least mean squares (KNLMS), perform nonlinear regression in a recursive manner, with similar computational requirements to linear techniques. In this article, an architecture for a microcoded kernel method accelerator is described, and high-performance implementations of sliding-window KRLS, fixed-budget KRLS, and KNLMS are presented. The architecture utilizes pipelining and vectorization for performance, and microcoding for reusability. The design can be scaled to allow tradeoffs between capacity, performance, and area. The design is compared with a central processing unit (CPU), digital signal processor (DSP), and Altera OpenCL implementations. In different configurations on an Altera Arria 10 device, our SW-KRLS implementation delivers floating-point throughput of approximately 16 GFLOPs, latency of $5.5\mu S$, and energy consumption of 10^{-4} J, these being improvements over a CPU by factors of 12, 17, and 24, respectively.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*

General Terms: Processors, Algorithms, Performance

Additional Key Words and Phrases: Kernel methods, vector processors, pipeline, field-programmable gate array (FPGA)

ACM Reference Format:
Yeyong Pang, Shaojun Wang, Yu Peng, Xiyuan Peng, Nicholas J. Fraser, and Philip H. W. Leong. 2016. A microcoded kernel recursive least squares processor using FPGA technology. ACM Trans. Reconfigurable Technol. Syst. 10, 1, Article 5 (September 2016), 22 pages.
DOI: http://dx.doi.org/10.1145/2950061

5

1. INTRODUCTION

In machine learning, traditional linear prediction techniques are well understood, and methods for their efficient solution have been developed. However, many real-world applications are better modeled using nonlinear techniques, which often have very high computational requirements. Mercer kernel techniques utilize linear methods in

This work is supported by the Australian Research Council under the Linkage Project LP110200413 and LP130101034; the Faculty of Engineering and Information Technologies, The University of Sydney, under the Faculty Research Cluster Program; Zomojo Pty Ltd. and Exablaze Pty Ltd.; the National Natural Science Foundation of China under Grant No. 61301205 and 61571160; the Twelfth Government Advanced Research Fund under Grant No. 9140A17050114HT01054; the Fundamental Research Funds for the Central Universities under Grant No. HIT.NSRIF.201615; and the Altera University Program.
Authors' addresses: Y. Pang, S. Wang, Y. Peng, and X. Peng, Automatic Test and Control Department, Harbin Institute of Technology; email: yeyongpang@126.com; N. J. Fraser and P. H. W. Leong, School of Electrical and Information Engineering, The University of Sydney.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2016 ACM 1936-7406/2016/09-ART5 $15.00
DOI: http://dx.doi.org/10.1145/2950061

a nonlinear feature space and combine the advantages of both. Such *kernel methods* are considered one of the major advances in machine-learning research [Friedman 2006; Wu et al. 2007]. Commonly used kernel methods include the support vector machine (SVM), Gaussian processes, and regularization networks [Scholkopf and Smola 2001]. These are batch based, and a global optimization is conducted over all input exemplars to create a model. In contrast, *online methods*, such as the kernel normalized least mean square (KNLMS) [Richard et al. 2009] and kernel recursive least squares (KRLS) algorithm [Engel et al. 2004; Van Vaerenbergh et al. 2006], 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.

We believe that reconfigurable computing, the application of field-programmable gate arrays (FPGAs) to computing problems, will prove to be an enabling technology for applications where minimal response time is critical. In this article, it is demonstrated, for the first time, that a combination of 16 GFLOPs performance with $5.5\mu S$ latency using 10^{-4} J can be achieved in a soft processor. A crucial factor is that FPGAs allow the data converters and/or network interface controller to be tightly integrated with high-speed processing, giving single-chip online implementations an order of magnitude improvement in power, size, throughput, and latency over stored-program technologies. As examples, it can lead to improved speed in predicting events in a wide variety of application domains including algorithmic trading [Lockwood et al. 2012], diagnostic and prognostic monitoring of machines [Jardine et al. 2006], and electricity blackout prevention [Makarov et al. 2005].

Kernel algorithms are based on linear algebra operations, which can be implemented efficiently on all types of computing technology including microprocessors, vector processors, FPGAs, and very large-scale integration (VLSI) technology. In this article, we propose a microcoded architecture for a kernel processor that combines high performance, low latency, and programmability at the microcode level. While the present work focuses on an FPGA implementation with a customizable data path and microcode, the architecture is adaptable to a higher-performance but less flexible VLSI implementation. The contributions of the work include:

—A novel scalable, pipelined vector processor architecture for kernel methods, which has advantages in terms of reusability, extensibility, and performance over a straightforward hardware description language (HDL) design approach.
—The first reported FPGA-based online kernel methods implementation. To the best of our knowledge, at $5.5\mu S$, 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), digital signal processor (DSP), and Altera OpenCL implementations.

Domain-specific processors such as the one described herein enable high performance without the need to be familiar with reconfigurable computing. Compared with high-level synthesis tools, different algorithms can be implemented without a time-consuming resynthesis and implementation step by simply changing the contents of microcode memory. This article is an extended version of Pang et al. [2013], which described a similar processor. Improvements presented here include instruction execution pipelining leading to a greatly improved clock rate, a more thorough description of the processor including design tools, configurable precision, OpenCL comparison, low-latency external peripheral support, implementation results for a more recent FPGA, and extension of the examples beyond sliding-window KRLS.

The remainder of the article is organized in the following manner. In Section 2, the kernel algorithms are described. The proposed architecture and implementation are introduced in Section 3, and an analysis of its performance, power consumption, and latency is given in Section 4. Conclusions are drawn in Section 5. Algorithmic descriptions of fixed-budget KRLS and KNLMS are provided in the appendix.

2. KERNEL-BASED MACHINE LEARNING

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 nonlinear machine-learning problems, the RLS algorithm can be modified to make use of a kernel function. The resulting KRLS algorithm provides a computationally efficient means to apply online learning to nonlinear processes.

In this section, a concise summary of the linear regression, RLS, and SW-KRLS algorithms are given. Fixed-budget KRLS (FB-KRLS) and KNLMS were also implemented and compared for our processor. Their descriptions are given in Appendix A.

2.1. Linear Regression

Let $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]^T \in \mathbb{R}^{N\times M}$ 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 $\mathbf{h} \in \mathbb{R}^M$ that minimized the following cost function: $J(\mathbf{h}) = \|\mathbf{y} - \mathbf{X}\mathbf{h}\|_2^2$. If $(\mathbf{X}^T\mathbf{X})$ is nonsingular, a direct solution can be computed for $\mathbf{h}$ as follows:

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

The dual representation can be obtained by premultiplying Equation (1) by the identity $(\mathbf{X}^T\mathbf{X})(\mathbf{X}^T\mathbf{X})^{-1}$ to obtain $\mathbf{h} = \mathbf{X}^T\boldsymbol{\alpha}$, making $\mathbf{h} = \sum_{i=1}^{N} \alpha_i \mathbf{x}_i$ a linear combination of the training set.

For an online problem, not all training data is known in advance and the solution must be recalculated as new observations are provided. The 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* [Higham 1996], $(\mathbf{A} + \mathbf{x}_n\mathbf{x}_n^T)^{-1} = \mathbf{A}^{-1} - \frac{\mathbf{A}^{-1}\mathbf{x}_n\mathbf{x}_n^T\mathbf{A}^{-1}}{1+\mathbf{x}_n^T\mathbf{A}^{-1}\mathbf{x}_n}$, where $\mathbf{x}_n$ 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 $\mathbf{X}_{n-1} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{n-1}]^T$.

2.2. Kernel Regression

A common way to adapt linear regression to nonlinear problems is to apply a nonlinear mapping, $\Phi(\mathbf{x}) \in \mathbb{R}^M \to \mathbb{F}$, to the input data. Linear regression can then be applied to this new data to find $\tilde{\mathbf{h}}$. The KRLS algorithm, described in Engel et al. [2004], 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, $\kappa(\mathbf{x}_i, \mathbf{x}_j)$. With this new representation, the cost function can be rewritten as $J(\boldsymbol{\alpha}) = \|\mathbf{y} - \mathbf{K}\boldsymbol{\alpha}\|_2^2$, where $\boldsymbol{\alpha}$ is an $N \times 1$ vector of weights.

A common kernel function is the Gaussian kernel, $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|_2^2}{2\sigma^2})$. Other kernel functions include the polynomial kernel, the hyperbolic tangent kernel, and the Laplacian kernel. Anguita et al. [2007] used a modified Laplacian kernel, given by $\kappa(\mathbf{x}_i, \mathbf{x}_j) = 2^{-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|_1}$, since it can be implemented efficiently in hardware. The Gaussian kernel is used throughout this article.

```
Initialize K_0 as (1 + c)I and K_0^{-1} as I/(1 + c).
for n = 1, 2, ... do
   Calculate k_n as [κ(x_{n−N'+1}, x_n), ..., κ(x_{n−1}, x_n)]^T and k_nn = κ(x_n, x_n) + c.
   Calculate K̂_{n−1}^{-1} from Equation (2)
   Calculate K_n^{-1} from Equation (3)
   Calculate α_n using α_n = K_n^{-1} y_n
end for
```

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

The minima of J using the dual representation is given by $\boldsymbol{\alpha} = \mathbf{K}^{-1}\mathbf{y}$. A problem with many kernel-based machine-learning functions is that the computational complexity of these algorithms scales superlinearly with the number of training samples [Engel et al. 2004]. Traditionally this has limited their use for online machine-learning applications. The KRLS algorithm uses a recursive method to compute $\mathbf{K}^{-1}$ efficiently for each new sample with a computational complexity of $O(N^2 + NM)$.

There are several methods for minimizing the computational cost of the KRLS algorithm. Most methods attempt to represent all of the training set using a small subset of training samples, referred to as the *dictionary*. In Engel et al. [2004], a sparsification procedure is employed to prevent any samples from being admitted to the dictionary 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 [Van Vaerenbergh et al. 2006] and the FB-KRLS algorithm [Van Vaerenbergh et al. 2010] set a limit, N', to the number of training samples that can be stored in the dictionary, enforcing a bound on the computation cost.

2.3. Sliding-Window KRLS

The SW-KRLS algorithm removes any training samples that are not in a fixed time window, N'. Given a stream of input/output pairs, $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots\}$, at training sample n, the input matrix becomes $\mathbf{X}_n = [\mathbf{x}_n, \mathbf{x}_{n-1}, \ldots, \mathbf{x}_{n-N'+1}]^T$ and the output vector becomes $\mathbf{y}_n = [y_n, y_{n-1}, \ldots, y_{n-N'+1}]$.

In order to calculate $\boldsymbol{\alpha}_n$, the nth estimate of the weights, the inverse kernel matrix, $\mathbf{K}_n^{-1}$, must be calculated. $\mathbf{K}_n^{-1}$ can be calculated using $\mathbf{K}_{n-1}^{-1}$, $\mathbf{X}_n$, and $\mathbf{x}_{n-N'}$. $\hat{\mathbf{K}}_n^{-1}$ can be calculated using

$$\hat{\mathbf{K}}_n^{-1} = \begin{bmatrix} \mathbf{K}_{n-1}^{-1}(\mathbf{I} + \mathbf{k}_n\mathbf{k}_n^T\mathbf{K}_{n-1}^{-1T}g) & -\mathbf{K}_{n-1}^{-1}\mathbf{k}_n g \\ -(\mathbf{K}_{n-1}^{-1}\mathbf{k}_n)^T g & g \end{bmatrix}, \tag{2}$$

where $\mathbf{k}_n = [\kappa(\mathbf{x}_{n-N'}, \mathbf{x}_n), \ldots, \kappa(\mathbf{x}_{n-1}, \mathbf{x}_n)]^T$; g is given by $g = (k_{nn} - \mathbf{k}_n^T\mathbf{K}_{n-1}^{-1}\mathbf{k}_n)^{-1}$; and $k_{nn} = \kappa(\mathbf{x}_n, \mathbf{x}_n) + c$, where c is a regularization constant. $\mathbf{K}^{-1}$ is then calculated as follows:

$$\mathbf{K}_n^{-1} = \mathbf{G} - \mathbf{f}\mathbf{f}^T/e, \tag{3}$$

where $\mathbf{G}$, e, and $\mathbf{f}$ are given by $\hat{\mathbf{K}}_n^{-1} = [\begin{smallmatrix} e & \mathbf{f}^T \\ \mathbf{f} & \mathbf{G} \end{smallmatrix}]$, and $\mathbf{G}$ is an $N' \times N'$ matrix, $\mathbf{f}$ is an $N' \times 1$ vector, and e is a scalar. $\boldsymbol{\alpha}_n$ is then calculated as $\boldsymbol{\alpha}_n = \mathbf{K}_n^{-1}\mathbf{y}_n$.

The SW-KRLS algorithm removes the oldest training pair from its dictionary when a new sample is encountered, allowing it to track nonstationary processes. Psuedocode for the SW-KRLS algorithm, derived from Van Vaerenbergh et al. [2006] and Van Vaerenbergh [2012], is provided in Figure 1. The corresponding number of arithmetic operations for SW-KRLS with a Gaussian kernel are summarized in Table I. This value is $7.5N^2 + (3M + 31.5) \times N + 3M + 31$ for training and $N^2 + (3M + 21) \times N - 1$ for

Table I. Arithmetic Operations for the SW-KRLS Algorithm

Operation	Add/Sub	Multiply	Divide	Exponential
Calculate $\mathbf{k}_n$ and k_{nn}	$(N+1)\times(2M-1)+1$	$(N+1)\times(M+1)$	0	$N+1$
Calculate $\hat{\mathbf{K}}_{n-1}^{-1}$ from Equation (2)	$2N^2+1$	$2.5N\times(N+1)$	1	0
Calculate $\mathbf{K}_n^{-1}$ from Equation (3)	$0.5(N+1)\times(N+2)$	$0.5N\times(N+1)$	N	0
Calculate $\boldsymbol{\alpha}_n$ using $\boldsymbol{\alpha}_n=\mathbf{K}_n^{-1}\mathbf{y}_n$	$N\times(N-1)$	N^2	0	0

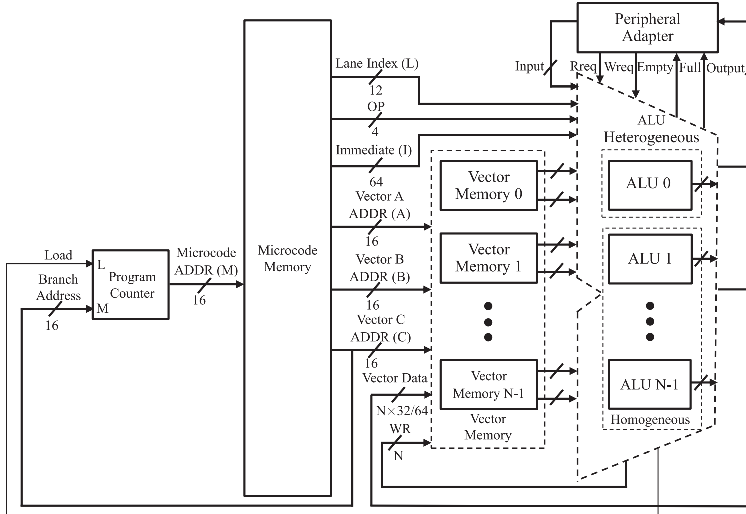


Fig. 2. Vector processor block diagram.

evaluation. Since floating point is used in the processor, the total number of floating-point operations is

$$FLOP = 8.5N^2 + (6M + 52.5)\times N + 3M + 30. \tag{4}$$

The algorithms described in this article, along with previous works [Van Vaerenbergh et al. 2006, 2010], use a regularization parameter to help prevent overfitting. With regularization, the cost function becomes

$$\begin{aligned} J'(\tilde{\mathbf{h}}) &= \|\mathbf{y} - \tilde{\mathbf{X}}\tilde{\mathbf{h}}\|_2^2 + c\|\tilde{\mathbf{h}}\|_2^2 \\ J'(\boldsymbol{\alpha}) &= \|\mathbf{y} - \mathbf{K}\boldsymbol{\alpha}\|_2^2 + c\boldsymbol{\alpha}^T\mathbf{K}\boldsymbol{\alpha}, \end{aligned} \tag{5}$$

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

3. ARCHITECTURE

3.1. System Overview

Our vector processor is a scalable floating-point processor that utilizes single-instruction multiple-data (SIMD)-style execution to operate upon vectors. The number of parallel arithmetic logic units (ALUs) and the precision are customizable. Figure 2 shows a block diagram of the processor architecture, formed from a program counter (PC), microcode memory, vector memory, and ALU and peripheral adapter. Microcode memory is used for storage of the microprogram.

Each independent data path lane includes a vector lane and vector memory. The targeted FPGA device accommodates up to 128 single-precision data path lanes or 32 double-precision data path lanes, which is larger than previous FPGA-based vector

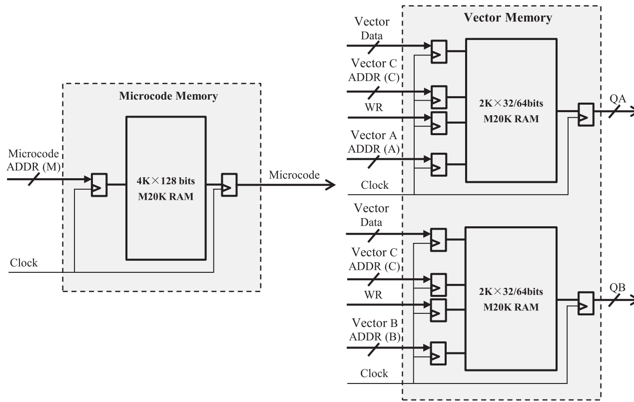


Fig. 3. Microcode and vector memory architecture.

processors [Chou et al. 2011]. Since all vector memory is on-chip, as the number of lanes is increased, the maximum vector memory depth is reduced [Yiannacouras et al. 2012].

Since kernel methods typically require a modest amount of microcode memory, 4K×128-bit is sufficient for all the implementations in this article. Most of the code is completely unrolled so the size of the microcode is very close to the number of machine cycles in Figure 6(a). An external memory interface is not included, and vector memory is directly connected to the ALU. There are two kinds of ALUs in the vector processor. ALU 1 to ALU (N-1) are called homogeneous ALUs and support multiplication, addition, subtraction, comparison, and absolute value. ALU 0 is a heterogeneous superset that additionally supports exponentiation, square root, division, and adder tree operators.

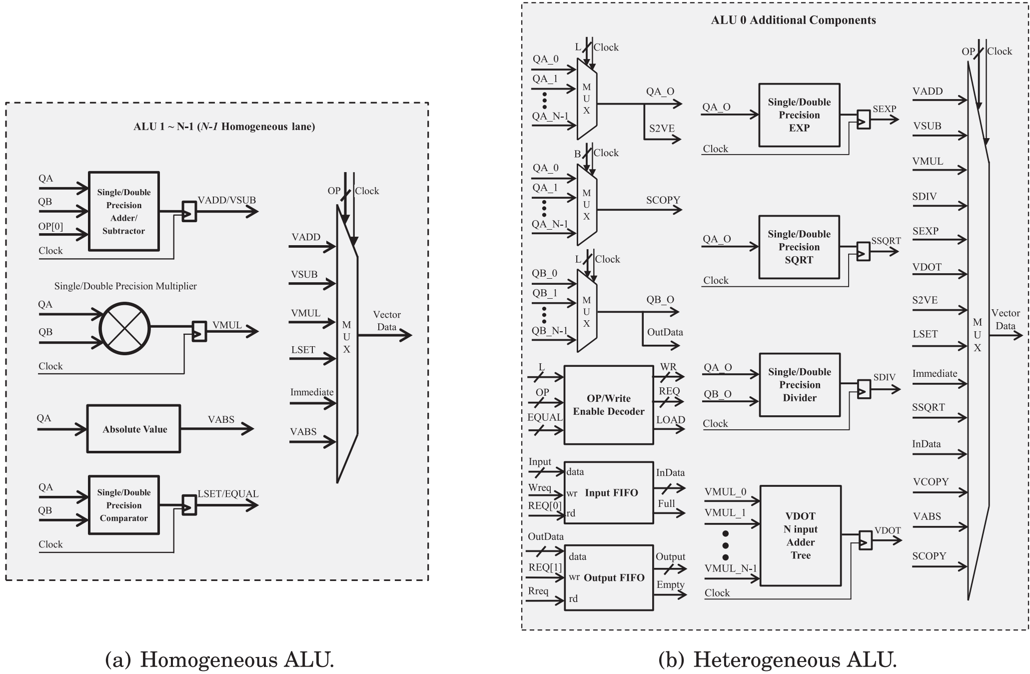
3.2. Memory Interface

A vector scratchpad memory, as introduced in VEGAS [Chou et al. 2011] and VINCE [Severance and Lemieux 2012], is employed to reduce load/store operations from memory. Both input and output ports of the vector memory and microcode memory are registered to get the maximum frequency. The microcode memory directly drives the vector memory to eliminate the need for address registers [Yu et al. 2009]. In the current implementation, Altera Embedded Memory Blocks [Altera 2016c] are used for all memories. As illustrated in Figure 3, microcode memory is configured as a 4K×128-bit single-port ROM memory and utilizes 32 M20K block memories. As the vector processor supports single- or double-precision floating point, the data width can be 32 or 64 bit. The capacity of each vector scratchpad memory is 2,048×32/64 bits, constructed from 4/8×M20K block memories.

The vector memory is used to store all intermediate data and constants. In our implementations of kernel methods, the kernel matrix, its inverse, and other variables are stored in the range 0x0000 to 0x03FF; temporary outputs from the ALU are stored in a region from 0x0400 to 0x7F3; vector memory is initialized in the bitstream; and the remaining 12 words of the memory are used to store constant vectors.

3.3. Vector Lane Architecture

Figure 4(a) illustrates the ALU 1 to ALU (N-1) design, and Figure 4(b) illustrates the additional components that reside in heterogeneous ALU. Each data path lane includes two vector scratchpad memories, a single- or double-precision floating-point adder/subtractor, a single- or double-precision multiplier, a single- or double-precision



(a) Homogeneous ALU. (b) Heterogeneous ALU.

Fig. 4. ALU architectures.

comparator, and a single- or double-precision absolute value operator. Lane widths are limited to powers of 2 to reduce hardware complexity. We chose powers of 2 as they fit neatly into binary numbers, but arbitrary lane widths could be supported with minor modifications. In addition to vector add, subtract, and multiply operations, the architecture supports vector divide, dot product, exponentiation, and square root. All floating-point operations are implemented using the Altera floating-point arithmetic library [Altera 2016a] and advanced DSP blockset.

Division, exponentiation, and square root consume more resources than other floating-point arithmetic modules but are used less frequently; for example, in Stratix V, a single-precision exponentiation module utilizes nine DSP blocks, while a single-precision multiplier requires a single one [Altera 2016a]. If exponentiation was included in each vector lane, resource utilization would increase significantly. For this reason, complex operators are only included in ALU 0. Though the vector exponentiation, division, and square root operations require more cycles to complete, the reduction in resources allows for more lanes to be implemented, increasing overall performance. With minor modifications, other functions such as sin/cos, tan/atan, inverse, and log can be added. Conversely, any unnecessary functional units can be excluded to save resources.

The scalar-to-vector, vector dot product, and adder tree function units are also implemented in ALU0. The scalar-to-vector function unit shares multiplexers with the exponentiation and division modules. The vector dot product instruction utilizes a parallel N input adder tree.

3.4. Peripheral Adaptor

The peripheral adapter implements a low-latency interface to standard bus arbiters such as PCI Express, USB, UART, or any other custom peripheral such as an

Table II. Microcode Format

Function	Vector A Address	Vector B Address	Vector C Address	Lane Index	Operation Code	Immediate Data
Symbol	A	B	C	L	OP	I
Bits	16	16	16	12	4	64

Table III. Microcode Instructions (All *i*, *j*, and *L* Indexes Range from 0 to N-1)

Microcode (Opcode)	Description	Function
VNOP (0000)	No operation	-
VADD (0001)	Vector add	$VM_i[C] = VM_i[A] + VM_i[B]$
VSUB (0010)	Vector subtract	$VM_i[C] = VM_i[A] - VM_i[B]$
VMUL (0011)	Array multiply	$VM_i[C] = VM_i[A] \times VM_i[B]$
SDIV (0100)	Scalar divide	$VM_L[C] = VM_L[A]/VM_L[B]$
SEXP (0101)	Scalar exponentiation	$VM_L[C] = EXP(VM_L[A])$
VDOT (0110)	Vector dot product	$VM_L[C] = \sum_{i=0}^{N-1} VM_i[A] \times VM_i[B]$
S2VE (0111)	Clone a vector N times	$VM_i[C] = VM_L[A]$
LSET (1000)	If less than, then set	If $VM_i[A] < VM_i[B]$, then $VM_i[C] = 1$
Branch If Equal (1001)	Conditional branch	If $VM_i[A] = VM_i[B]$, $M = C$
VCOPY (1010)	Vector copy with alignment	If $i < L$, $VM_i[C] = VM_i[A]$, Else $VM_i[C] = VM_{i+1}[A]$
SLOAD (1011)	Load a scalar	$VM_L[C] = Immedeiate$
SMOV (1100)	Move to/from peripheral	If $A = 0$, $VM_L[C] = InData$, Else $OutData = VM_L[B]$
SSQRT (1101)	Scalar sqrt	$VM_L[C] = sqrt(VM_L[A])$
SCOPY (1110)	Scalar copy	$VM_L[C] = VM_B[A]$
VABS (1111)	Vector absolute	$VM_i[C] = \lvert VM_i[A] \rvert$

analog-to-digital converter. This allows data to be directly transferred from peripherals to vector memory, minimizing latency.

As illustrated in Figure 2, it has separate input and output FIFOs, which we assume are directly connected to other hardware devices. The "Full" and "Empty" signals indicate the FIFO status, according to which the peripheral adapter generates appropriate read and write requests, "Wreq" and "Rreq." Data can be transferred to/from the FIFOs using the SMOV instruction, as they are also memory mapped. In each training iteration, an input vector is read from the input FIFO and processed, and the prediction is written to the output FIFO. Peripheral I/O proceeds in parallel with instruction execution.

3.5. Microcode

The architecture enables the processor to be easily programmed for different kernel algorithms at both the instruction set and data path levels. The microcode is 128 bits wide, with all vector and microcode addresses being 16 bits. MATLAB code from KAFBOX [Van Vaerenbergh 2012] was manually vectorized and converted to microcode.

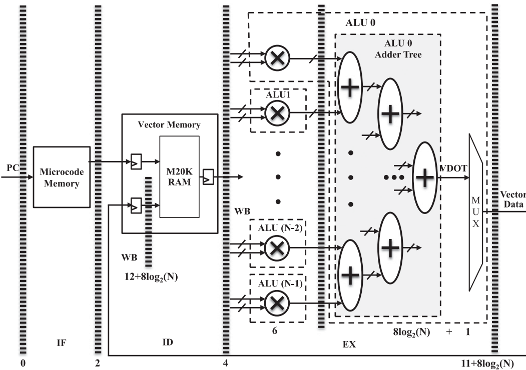
The microcode format is summarized in Table II. "A" and "B" are used to specify their respective input vector address. The vector memory for the ith lane is represented as VM_i. Vector address "C" specifies the destination address. The Lane Index "L" is a 12-bit value that specifies the vector memory lane for storing the result of a VDOT instruction. The opcode is given in the "OP" field and the immediate value field "I" specifies a floating-point constant that can be loaded to vector memory.

As can be seen in Figure 3, both write ports are driven with the same address and data signals (Vector ADDR C and Vector Data), so identical data are written to both ports. Table III describes the function of each opcode.

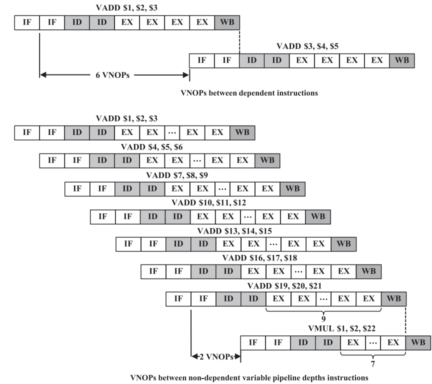
An assembler has been developed to translate a textual description of a program to binary microcode, which can be directly uploaded to the microcode memory.

Table IV. Instruction Pipeline Depths on Stratix V and Arria 10

Device	Stratix V		Arria 10	
Precision	Single	Double	Single	Double
VNOP (0000)	1	1	1	1
VADD (0001)	14	16	9	22
VSUB (0010)	14	16	9	22
VMUL (0011)	12	12	9	16
SDIV (0100)	40	68	31	53
SEXP (0101)	24	32	24	48
VDOT (0110)	$12 + 8\log_2(N)$	$12 + 10\log_2(N)$	$9 + 3\log_2(N)$	$16 + 16\log_2(N)$
S2VE (0111)	7	7	7	7
LSET (1000)	7	7	7	7
Branch If Equal (1001)	6	6	6	6
VCOPY (1010)	6	6	6	6
SLOAD (1011)	4	4	4	4
SMOV (1100)	4	4	4	4
SSQRT (1101)	35	64	18	38
SCOPY (1110)	7	7	7	7
VABS (1111)	7	7	7	7



(a) ALU pipeline organisation.



(b) Pipeline overlap behavior.

Fig. 5. Vector processor pipelining scheme.

Writing software in assembly is straightforward (albeit tedious) since vector and matrix operations are supported by the processor.

3.6. Pipelining

To maximize the clock frequency, we employ pipelined instruction execution, with microcode instructions requiring a different number of cycles to complete. Table IV shows the microcode instruction pipeline depths for both Stratix V and Arria 10 devices.

As can be seen in Figure 5(a), the CPU has a four-stage pipelined data path with instruction fetch (IF), instruction decode (ID), execute (EX), and write-back (WB). Each lane operates on independent data. The pipeline latency for IF, ID, and WB are fixed at two cycles, two cycles, and one cycle, respectively. Latencies for EX are instruction dependent.

Taking the VDOT instruction as an example, the multiplier latency is six cycles, the adder latency is eight cycles, and the adder tree has $8\log_2(N)$ cycle latency. One additional cycle is required to pipeline the vector data multiplexer. The latency of the

execute stage is thus 7 + $8\log_2(N)$ and total latency 12 + $8\log_2(N)$. Performance could be improved using a parallel accumulator with lower latency (e.g., Sun and Zambreno [2009]).

For SLOAD and SMOV instructions, access to vector memory is not required and the ID stage is omitted. One cycle of latency is required by EX so the total latency is 4. For all the other instructions, the total latency equals the execute stage latency plus five cycles (IF, ID, and WB).

Data hazards require VNOPs to be inserted because pipeline interlocks are not implemented. To prevent a read-after-write data hazard, an instruction dependent on the result of a previous instruction must wait until the computed result is written back to the vector memory. The top part of Figure 5(b) illustrates pipeline behavior in this case. The VADD instruction (on the Arria 10) requires six VNOPs to resolve a read-after-write hazard, and the processor must wait for $3 to be written before it can be read in the ID stage of the following instruction. The same sequence on the Stratix V requires 11 VNOPs, demonstrating the advantages of lower latency.

Write-after-write hazards can also occur, as illustrated in the bottom half of Figure 5(b). Since only one port of vector memory can be written per cycle, an instruction with a smaller pipeline depth than previous instructions must wait until the previous write-back stages to finish before moving to its write-back stage. Since VADD requires two more cycles than VMUL, two additional VNOPs are required to resolve this hazard.

4. RESULTS

An implementation of the KRLS vector processor was made using Verilog Hardware Description Language (Verilog HDL) and Altera MegaWizard [Altera 2016b]-generated RTL files. Quartus Prime 16.0 was used for FPGA synthesis, place, and route. Both single- and double-precision versions of the vector processor were implemented, and furthermore, 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 device. We also report synthesis and simulation results for a newer FPGA, the Arria 10 10AX115N3F45E2 device in an Altera DE5a board, this having a new DSP block that supports fixed- and floating-point arithmetic [Langhammer and Pasca 2015].

Altera OpenCL implementations of the same kernel adaptive filtering (KAF) algorithms are used for comparison. The host system is an HP workstation, with an Intel Xeon 5570 CPU running the Windows 7 64-bit operating system. The same Altera DE5 board was connected via a 16-lane Gen2 PCI Express slot. Microsoft Visual Studio 2013 was used to compile the host program and source code pragmas used in OpenCL to maximize task parallelism. OpenCL was chosen as the comparison point as recent studies have shown that it can generate implementations with comparable performance to other techniques [Rashid et al. 2014].

The current implementation assumes online input data are present in the input FIFO, as explained in Section 3.4. Any I/O is supported by using a customized peripheral adapter as described in Section 3.1. The prediction output can be monitored at the “Output” port of the output FIFO, as shown in Figure 2. Performance is limited by processing rather than bandwidth as we can obtain new data every cycle but it takes many cycles to perform a training iteration.

4.1. FPGA Resource Usage

Table V shows resource usage on the Stratix V. The FPGA used features 234K ALMs, 256 DSP blocks (configurable as one 27×27-bit multiplier or two 18×18-bit multipliers), and 2,560 M20K memory blocks. In the DE5, due to the availability of ALMs and 27-bit

Table V. Vector Processor Resource Usage

Device and Precision	Sliding-Window Length	ALM (K) (Used/ % of Total)	M20K Mem (Blocks Used/ % of Total)	DSP Blocks (Used/ % of Total)
Stratix V Single	15	24/10%	153/6%	25/10%
	31	45/19%	281/11%	41/16%
	63	85/36%	537/21%	73/29%
	127	162/69%	1047/41%	137/54%
Stratix V Double	15	43/18%	262/10%	100/39%
	31	82/35%	486/19%	164/64%
Arria 10 Single	15	7/2%	159/6%	60/4%
	31	14/3%	287/11%	108/7%
	63	24/6%	543/20%	204/13%
	127	48/11%	1055/39%	396/26%
Arria 10 Double	15	46/11%	289/11%	116/8%
	31	87/20%	513/19%	180/12%

Table VI. Altera OpenCL KRLS Implementation Resource Usage

Precision	Sliding-Window Length	ALM (K) (Used/ % of Total)	M20K Mem (Blocks Used/ % of Total)	DSP Blocks (27 Bit) (Used/ % of Total)
Single	15	90/38%	947/37%	36/14%
	31	90/38%	998/38%	36/14%
	63	90/38%	1050/41%	36/14%
	127	90/38%	1178/46%	36/14%
Double	15	92/39%	973/38%	54/21%
	31	92/39%	998/39%	54/21%
	63	92/39%	1075/42%	54/21%
	127	92/39%	1223/48%	54/21%

DSP blocks, the maximum number of single-precision lanes that can be supported is 128, and the maximum number of double-precision lanes is 32.

The floating-point cores in the ALUs dominate the area of the vector processor. On the Stratix V for single precision, a single DSP is required for the floating-point multiplier in each of the N lanes, and an additional nine DSPs are required for the exponentiation unit in ALU0. Similarly, memory usage is dominated by the vector memory, which grows linearly with N. ALM usage is mainly that required to implement the floating-point adders in the vector lanes and ALU0 adder tree, plus additional operators and multiplexers in ALU0. Minimal resources are required for control as it is stored in microcode memory. In summary, for single precision on the Stratix V, DSP, memory, and ALM resources scale linearly with the number of vector lanes, N. As would be expected, going to double precision roughly quadruples the number of DSPs and doubles memory.

Table V also shows the resource usage for the Arria 10, which features 427K ALMs, 1,518 hardened single-precision floating-point DSP blocks (one single-precision floating-point DSP block is configurable as two 18×19-bit multipliers), and 2,713 M20K memory blocks. Note that one additional DSP block is used per lane over the Stratix V for the single-precision adder. $N-1$ additional DSP blocks are used for the adder tree, and four additional DSP blocks are used for division and square root units. For double precision, soft logic is used to implement adders, and DSPs are only used in the multiply and exponential, division, and square root operators.

Table VI shows the resource usage of the Altera OpenCL implementation of SW-KRLS, as reported by the Altera OpenCL tool. Increasing sliding-window length

Table VII. Vector Processor Performance Results

Example	Sliding-Window Length	DSP (Single/Double) (μS)@1.0GHz		CPU (Single/Double) (μS)@3.1GHz		OpenCL Kernel (Single/Double) (μS)@261.7MHz		Vector Processor (Single/Double) (μS)@250.0MHz	
SW-KRLS	15	55.6	56.8	1.7	2.0	88.3	103.3	3.2	4.8
	31	199.6	200.8	7.2	7.2	90.6	105.9	4.6	6.9
	63	364.8	366.0	23.8	31.4	93.8	107.2	7.4	-
	127	4,475.6	4,732.4	113.7	143.6	99.7	113.6	12.6	-
FB-KRLS	15	74.0	76.4	2.4	3.0	91.7	109.2	3.4	4.9
	31	249.2	255.6	8.8	9.2	93.5	111.8	5.0	7.2
	63	406.8	409.2	25.8	35.4	97.8	113.0	8.0	-
	127	5,642.4	6,057.2	118.8	152.6	103.9	121.4	13.9	-
KNLMS	15	16.8	17.2	0.8	1.2	38.3	41.6	1.9	2.5
	31	36.6	36.0	1.5	2.3	39.4	43.2	2.6	3.4
	63	45.2	45.6	2.8	4.9	41.2	45.0	3.6	-
	127	75.2	75.6	4.9	8.5	43.0	47.5	5.8	-

consumes more on-chip memory but maintains the same ALMs and DSP blocks. Compared with the vector processor, OpenCL always consumes more block memory. For the largest sliding-window length designs (128 for single and 32 for double), the direct OpenCL implementation consumes fewer resources than the vector processor.

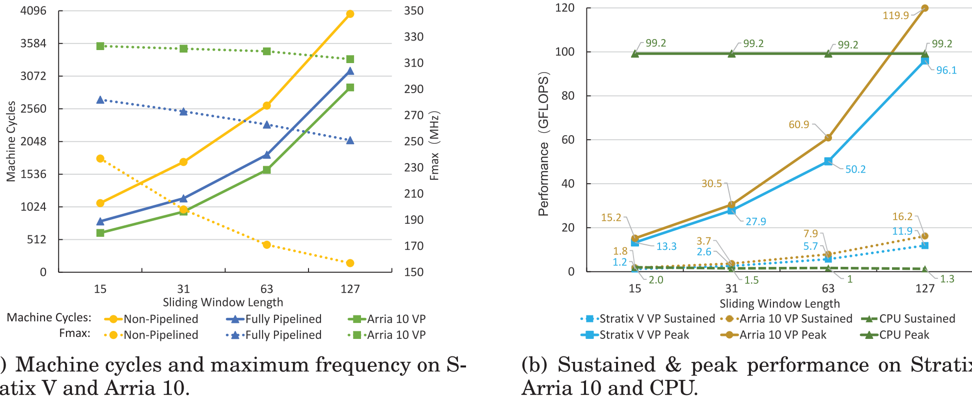
4.2. Performance Analysis

We compare processing performance for all platforms, over all kernel algorithms. 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 13.10 using GCC version 4.8.1. The DSP configuration is a Texas Instruments TMS320C6678 DSP development kit running at 1GHz. The MSGQ APIs provided by the SYS/BIOS system are employed to do multicore communication between the eight-processor core in this KeyStone DSP processor. Some extra efforts are made to make the original code run faster on this multicore DSP processor. The vector processor on the Stratix V device uses 250MHz for the system clock frequency.

Note that an N lane processor is used to implement a sliding-window length of $N-1$. An optimized C code implementation was developed for the CPU and DSP platforms, and the ATLAS [Whaley and Petitet 2005] library was used for linear algebra. Although a multithreaded ATLAS library [Whaley and Petitet 2005] was tested, the highest performance on the CPU was achieved with a single-threaded implementation. We believe this was because our matrix sizes were too small to benefit from multiple cores because in our current implementation, vector lengths cannot exceed the number of lanes.

Table VII shows the vector processor performance on the Stratix V device versus the CPU, DSP, and Altera OpenCL designs for sliding-window lengths between 15 and 127. The input vector size was chosen as $M = 7$. A total of 12 cases were tested for both single and double precision. For small sliding-window sizes, the CPU outperforms all other platforms. The vector processor outperforms the CPU and DSP for $N \geq 32$ on SW-KRLS and FB-KRLS, and for the case $N = 128$, the speedup is 9.

For the KNLMS algorithm, the CPU achieved the highest performance. In contrast to the computations of Equations (2), (3), (8), and (9) for SW-KRLS and FB-KRLS, in the case of KNLMS, the computations of Equations (13) and (14) involve mostly scalar operations and take more than half of the execution time, limiting what can be achieved through vectorization.



(a) Machine cycles and maximum frequency on Stratix V and Arria 10.

(b) Sustained & peak performance on Stratix V, Arria 10 and CPU.

Fig. 6. SW-KRLS single-precision performance comparison between pipelined and nonpipelined processors.

In Table VII, the OpenCL design runtime is the kernel time, with PCI Express transfer time excluded. In the vector processor, the PCI Express bus is not used during training as I/O is done through the Peripheral Adaptor (Section 3.4). Similarly, PCI Express is not used for the other implementations. The vector processor achieves the highest performance of the four implementations when the window length is large.

We compare the performance of the fully pipelined vector processor to our previously published nonpipelined SW-KRLS implementation [Pang et al. 2013] using the same Stratix-V device. As shown in Figure 6(a), the vector processor with fully pipelined architecture uses fewer cycles and achieves higher clock frequency and higher performance. Figure 6(a) provides detailed information on the maximum frequency, F_{max}, for each different lane configuration of the vector processor.

A higher clock frequency is obtained using the Arria 10 device, and the performance is approximately 1.4 times higher than Stratix V. Figure 6(b) shows floating-point operations per second (FLOPs) for the Stratix V, Arria 10, and CPU implementations. Peak performance is calculated as (number of multipliers + number of adders) $\times$ Fmax. Sustained performance is measured as the number of floating-point operations divided by the execution time, where the number of floating-point operations is calculated using Equation (4).

4.3. Precision Analysis

The MG-30 Mackey-Glass [Mackey and Glass 1977] benchmark, modeled by the differential equation $\frac{dx(t)}{dt} = -ax(t) + \frac{bx(t-\tau)}{1+x(t-\tau)^{10}}$ with $(a = 0.1, b = 0.2, \tau = 30)$, was used to test the SW-KRLS implementation. A regularization parameter of $c = 0.01$ and kernel parameter $\sigma = 0.6$ was chosen. The single-precision floating-point vector processor was compared with KAFBOX [Van Vaerenbergh 2012], an open-source double-precision implementation. As shown in Figure 7, the difference in mean squared error (MSE) between KAFBOX and our processor was less than 0.07%, and the maximum relative error was less than 0.5%. We consider this error negligible for our purposes and thus conclude that single-precision floating point is sufficient for the MG-30 problem.

The Lorenz dataset can be created by Runge-Kutta integration of the Lorenz equations and is available in KAFBOX [Van Vaerenbergh 2012].

Figure 8 shows the maximum relative error, MRE, of the SW-KRLS, FB-KRLS, and KNLMS algorithms with a sliding-window length of 127. The error for sample i is calculated by comparing the output of each of the hardware platform simulation results with a double-precision reference generated by KAFBOX over 1,000 training

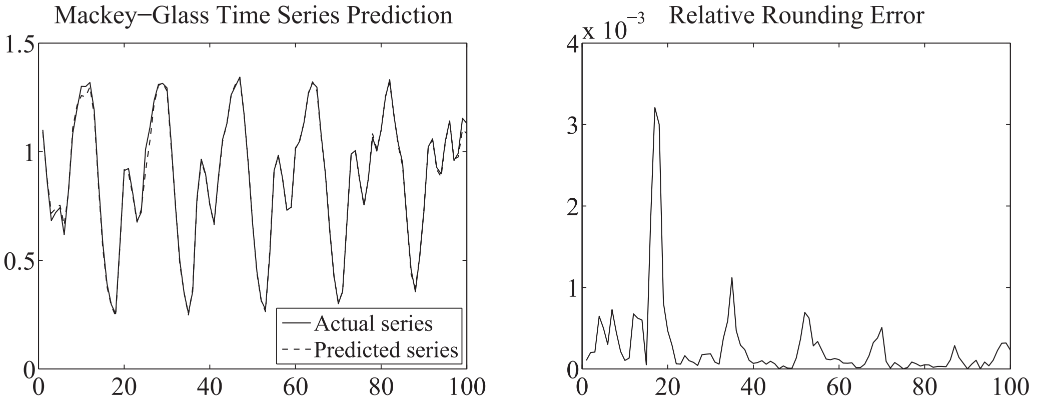
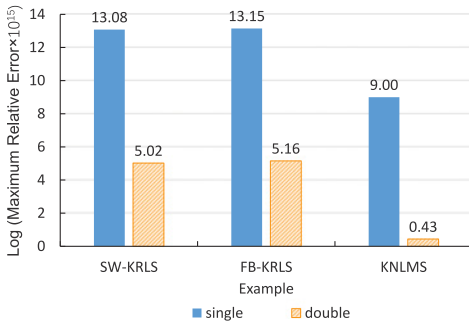
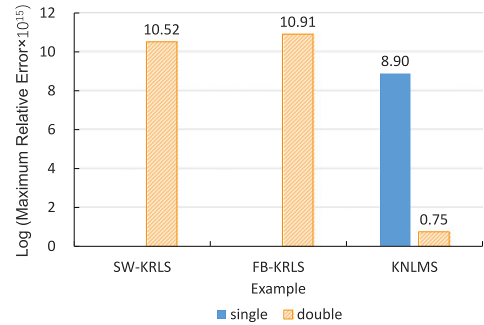


Fig. 7. The left-hand plot shows the KAFBOX output compared with the vector processor. The right-hand plot shows their relative error. Results are for single-precision floating-point and a sliding-window length of 127.



(a) MRE with MG30 dataset.



(b) MRE with Lorenz dataset.

Fig. 8. Error compared to KAFBOX golden reference in single and double precisions with a sliding-window width of 127.

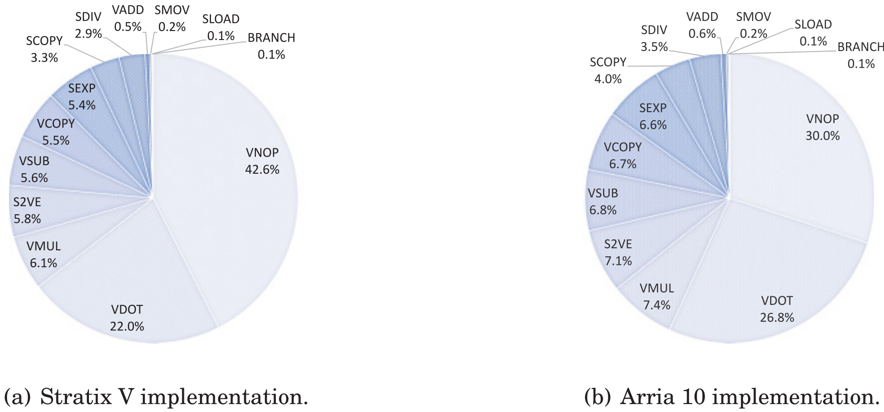
and prediction pairs, that is, $MRE = \max_i(|(x_{iobs} - x_{iref})/x_{iref}|)$, where x_{iref} is the Matlab output and x_{iobs} is the observed prediction output. For the Lorenz problem using SW-KRLS and FB-KRLS, single precision was not sufficient for convergence so only double-precision results are reported.

4.4. Dynamic Instruction Frequency

In this subsection, we provide a dynamic instruction analysis for the SW-KRLS microcode using a lane width of 32. Instruction frequency was measured via simulation and shown in Figure 9. From this figure, we can see that SDIV, SEXP, and SSQRT only take a small portion of the total instruction cycles, justifying a single heterogeneous ALU. Results for the SW-KRLS, FB-KRLS, and KNLMS microcode over different lane widths yield similar results, with the VDOT and VMUL instructions consuming the highest machine cycles for all applications.

4.5. Energy Consumption

This section details the energy consumption of the Vector Processor, Altera OpenCL, DSP, and CPU implementations. The power dissipation of the vector processor on Stratix V and Altera OpenCL implementations was measured using an Agilent U8001A power supply. The Arria 10 power consumption was obtained using the PowerPlay



(a) Stratix V implementation. (b) Arria 10 implementation.

Fig. 9. SW-KRLS microcode dynamic execution analysis, lane width 32.

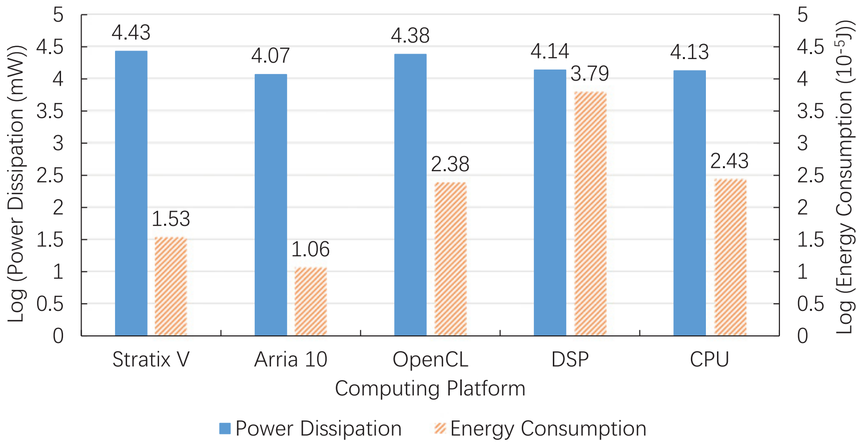


Fig. 10. SW-KRLS energy consumption ($N = 128$).

Power Analyzer tool in Quartus Prime 16.0 at 300MHz. Results will be optimistic as they do not include the consumption of other chips required to make the system function. The DSP power dissipation was measured from the DC input jack using a Fluke 15B 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 optimized for power dissipation, the CPU configuration is an Apple MacBook Air laptop with the following specifications: Intel Core i5 CPU 2467M @ 1.6GHz, 4GB of memory, running Mac OSX 10.9 using Apple LLVM 5.1. Note this machine also uses flash storage as opposed to a hard disk drive.

From Figure 10, it can be seen that the Arria 10 implementations require the least amount of power. When the computational time is also taken into consideration, the vector processor outperforms all other implementations in terms of energy consumption for all sliding-window lengths. For the SW-KRLS algorithm with $N = 128$, energy consumption of the Stratix V FPGA and Arria FPGA is a factor of 8 and 24 lower than that of the CPU. Results for the FB-KRLS algorithm are similar. For KNLMS,

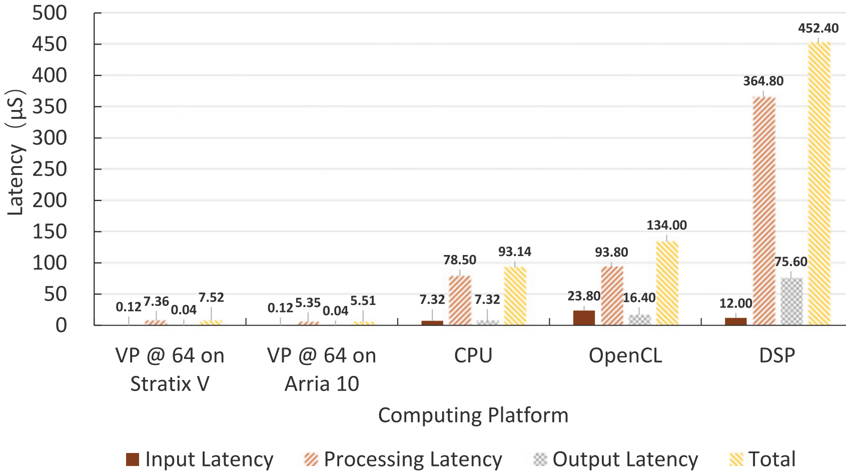


Fig. 11. Latency test results (SW-KRLS, $N = 64$).

significant vectorization is not possible and hence the execution time and energy of the vector processor are similar to the CPU.

4.6. Latency Test

We define latency to be the time taken from the arrival of a new input at a device's input pins (32 bits or 4 bytes for single precision) to when the output prediction changes the device's output pins. On an FPGA, this can be minimized by avoiding any buffering and off-chip transfers. While double buffering can improve throughput by amortizing overheads over many inputs, it does not improve latency. FPGAs have a clear latency advantage as they can avoid off-chip I/O and buffering.

Latency tests for performing online learning and prediction from analog data sources were conducted. For Stratix V and OpenCL, we used a DE5 board equipped with a Terasic High-Speed A/D and D/A Development Kit. The same input/output latencies were used for the Arria 10. The vector processor runs at 250MHz on the Stratix V device and at 300MHz on the, Arria 10 device. For the CPU we used the National Instruments 5781 Baseband Transceiver and NI 7954 FPGA modules, and the DSP latency assumed an LTC2422 ADC and LTC2607 DAC connected via a serial peripheral bus operating at the highest frequency. Results for the SW-KRLS example are shown in Figure 11, with those for FB-KRLS and KNLMS being similar. The vector processor outperforms the other platforms in terms of latency due to its fast processing time and minimal I/O overhead. In comparison, the DSP suffers from both high I/O latency and high processing latency, and the CPU implementation is capable of providing low processing latency but suffers from high I/O latency during data acquisition, due to PCI bus and kernel overheads. The OpenCL implementations used PCI Express DMA data transfer to initialize the kernel and read back data. This results in high input and output latency.

4.7. Related Work

4.7.1. Hardware Implementations of Machine Learning. Module generators, which can create a family of parameterized designs rather than a single one, are common in reconfigurable computing. Anguita et. al. [2011] described an FPGA-based fixed-point core generator for SVMs that allows submodules with different speed, resource, and accuracy tradeoffs to be included. Papadonikolakis and Bouganis [2008] developed a

scalable SVM module generator that allows for the use of different kernel types and uses different numbers of parallel tiles to optimize performance. The design was partitioned into fixed- and floating-point parts to achieve high speed without sacrificing accuracy. Majumdar et. al. [2012] 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, and independent off-chip memory banks were utilized to allow the solution of large problems.

Lin et. al. [2010] developed a deeply pipelined Bayesian computing machine (BCM) using floating-point arithmetic to evaluate probabilistic networks. A system with 16 processing nodes achieved average $80/15\times$ speedups over a CPU/GPU implementation, respectively.

Shan et. al. [2010] developed a map-reduce framework and applied it to the acceleration of the RankBoost algorithm using floating-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 a fixed-point implementation of the Restricted Boltzman Machine (RBM) by Ly [2010]. 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 was introduced. Virtualization through time multiplexing was employed to allow larger problems to be solved.

In this work, a parallel data path is combined with microcoded control logic to produce a soft vector processor. While both Ly [2010] and Majumdar [2012] targeted maximum performance in batch learning tasks, ours is designed for single-FPGA, floating-point embedded applications in which minimizing latency and compactness are the key design goals. Our designs could be considered complementary to some previous work in that module generators, scheduling, map-reduce, MPI, and so forth can be combined with our architecture.

4.7.2. Soft Vector Processors. A soft vector processor is a processor implemented using FPGA resources [Yiannacouras et al. 2012]. Most well-known previous soft vector processors [Yiannacouras et al. 2012; Chou et al. 2011; Severance and Lemieux 2012; Yu et al. 2009] have not supported floating-point operations. The FPVC, or floating-point vector coprocessor, developed by Kathiara and Leeser [2011] adds a floating-point vector unit to the hard Xilinx PowerPC cores, which can exploit SIMD parallelism as well as pipelining. A recent soft vector processor with streaming pipelines [Severance et al. 2014] allows custom vector instructions, which can be floating-point operations.

In terms of data representation, while the performance benefits of fixed-point and reduced-precision floating-point implementations are undeniable, we advocate single- and double-precision floating point. The proposed architecture could be easily modified for fixed-point and mixed fixed-floating-point arithmetic.

We also assume that there is only one data stream to process. This makes parallelism through multithreading difficult as the next data item can only be processed after the state has been updated.

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 optimized 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 [Bittner and Ruf 2012]. Since the round-trip transfer time alone is similar to the FPGA and CPU total latency including processing, we concluded that GPGPUs were not suitable for latency-critical applications and hence were not included in this study.

Similar to previous soft vector processors [Yiannacouras et al. 2012; Chou et al. 2011; Severance and Lemieux 2012; Yu et al. 2009], this vector processor architecture offers scalable performance and area via control of the number of vector lanes. We are not aware of other soft vector processors specifically targeting kernel methods or low-latency machine learning.

5. CONCLUSIONS

In this work, a microcoded vector processor optimized to reduce latency for kernel-based machine-learning algorithms was presented. The architecture allows efficient computation of dot product, matrix-vector multiplication, and kernel evaluation operations, and features simplicity, programmability, and compactness. For SW-KRLS and FB-KRLS, significant improvements in execution time, latency, and energy consumption were observed compared with a CPU and DSP. As limited opportunities are available for vectorization of the KNLMS algorithm, execution time and energy are similar to the CPU, with an order of magnitude reduction in latency.

We showed that the microcoded vector processor can be efficiently implemented on both Stratix V and Arria 10 devices. Floating-point cores on these devices have very different latencies, but this can be handled by adjusting pipeline stage latencies. Newer FPGA architectures will further improve the performance of our processor, and we believe they can be utilized efficiently with minor changes to the pipeline structure and microcode.

The microcoded architecture presented herein is a flexible processor capable of implementing many different algorithms without resynthesis of the design. Future work includes incorporating a parallel accumulator to improve performance of the "VDOT" instruction, and development of tools to allow the architecture to be customized to an even wider range of applications. It would also be interesting to investigate the feasibility of using techniques such as Dekker's algorithm [Dekker 1971] to achieve higher precision using only single-precision operations. This may better utilize the hard single-precision floating-point cores available in Arria 10 and future FPGAs.

APPENDIX

A. DESCRIPTIONS OF THE FB-KRLS AND KNLMS ALGORITHMS

In this appendix, we provide concise descriptions of the FB-KRLS and KNLMS algorithms implemented in the article.

A.1. Fixed-Budget KRLS

The FB-KRLS algorithm has a fixed dictionary size and employs a scheme that attempts to keep the most useful training examples in the dictionary. Using a fixed dictionary size bounds the computational cost of the algorithm. Thus, the computational cost can be determined at design time, a useful property for real-time and high-performance implementations. A comparative study of KAF algorithms [Van Vaerenbergh and Santamaria 2013] suggests that for stationary systems, the FB-KRLS achieves the lowest MSE at the smallest computational cost.

Let $\mathbf{D}_n \in \mathbb{R}^{N' \times M}$ be the dictionary at time n, which contains N' selected training examples from all inputs, $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, such that one input vector is stored in each row; and $\mathbf{y}_n \in \mathbb{R}^{N'}$ be the solution vector, $\{y_1, \ldots, y_n\}$, corresponding to the selected examples of $\mathbf{D}_n$. Using $\mathbf{D}_n$ and $\mathbf{y}_n$, at time n, the least squares weights are $\boldsymbol{\alpha}_n$ and the inverse matrix is $\mathbf{K}_n^{-1}$.

The pseudocode in Figure 12 lists the steps required to update the model. Given a new training example, $(\mathbf{x}_{n+1}, y_{n+1})$, an augmented dictionary $\tilde{\mathbf{D}}_{n+1}$ and augmented solution

---

Initialize $\mathbf{D}_0$ as $\mathbf{0}$, y_0 as $\mathbf{0}$, and $\mathbf{K}_0^{-1}$ as $\mathbf{I}/(1+c)$.
for $n = 1, 2, ...$ **do**
  Add new example as rows to $\mathbf{D}_{n-1}$ and $\mathbf{y}_{n-1}$ to create $\tilde{\mathbf{D}}_n$ and $\tilde{\mathrm{y}}_n$.
  Calculate $\tilde{\mathrm{k}}_n$ using Equation (7).
  Calculate $\hat{\mathrm{K}}_{n-1}^{-1}$ from Equation (8).
  Find j as the index corresponding to the smallest entry of s using Equation (10).
  Remove j^{th} row from $\tilde{\mathbf{D}}_n$ and $\tilde{\mathrm{y}}_n$ to get $\mathbf{D}_n$ and $\mathbf{y}_n$.
  Calculate K_n^{-1} from Equation (9).
  Calculate $\boldsymbol{\alpha}_n = \mathbf{K}_n^{-1}\mathbf{y}_n$.
end for

Fig. 12. Pseudocode of the FB-KRLS algorithm.

vector $\tilde{\mathbf{y}}_{n+1}$ are obtained by appending $\mathbf{x}_{n+1}$ and y_{n+1} to $\mathbf{D}_n$ and $\mathbf{y}_n$, respectively. The kernel vector, $\tilde{\mathbf{k}}_n$, is then given by

$$\mathbf{p} = [\kappa(\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_{N'+1}), \ldots, \kappa(\tilde{\mathbf{d}}_{N'}, \tilde{\mathbf{d}}_{N'+1})]^T \tag{6}$$

$$\tilde{\mathbf{k}}_n = [\mathbf{p}^T, C]^T, \tag{7}$$

where $\tilde{\mathbf{d}}_i$ is the ith entry of $\tilde{\mathbf{D}}_{n+1}$, $C = \kappa(\tilde{\mathbf{d}}_{N'+1}, \tilde{\mathbf{d}}_{N'+1}) + c$, and c is a regularization constant.

$\mathbf{K}^{-1}$ is then calculated by first computing

$$\hat{\mathbf{K}}_{n+1}^{-1} = \begin{bmatrix} \mathbf{K}_n^{-1}(\mathbf{I} + \mathbf{p}\mathbf{p}^T\mathbf{K}_{n-1}^{-1T}g) & -\mathbf{K}_n^{-1}\mathbf{p}g \\ -(\mathbf{K}_n^{-1}\mathbf{p})^T g & g \end{bmatrix}, \tag{8}$$

where g is given by $g = (C - \mathbf{p}^T\mathbf{K}_n^{-1}\mathbf{p})^{-1}$, and then

$$\mathbf{K}_{n+1}^{-1} = \mathbf{G} - \mathbf{f}\mathbf{f}^T/e, \tag{9}$$

where e is the diagonal entry on the jth row/column of the $\hat{\mathbf{K}}_{n+1}^{-1}$, $\mathbf{f}$ vector of the entries in the jth row/column of $\hat{\mathbf{K}}_{n+1}^{-1}$ excluding e, and $\mathbf{G}$ is a matrix made from all remaining entries of $\hat{\mathbf{K}}_{n+1}^{-1}$. Note that $\mathbf{f}$ is defined as the jth row and column since $\hat{\mathbf{K}}_{n+1}^{-1}$ is symmetrical. The value for j is determined by selecting the index corresponding to the smallest entry of $\mathbf{s}$, which is given by

$$\mathbf{s} = \left|\hat{\mathbf{K}}_{n+1}^{-1}\tilde{\mathbf{y}}_{n+1}\right| ./ diag\left(\hat{\mathbf{K}}_{n+1}^{-1}\right), \tag{10}$$

where $diag(\hat{\mathbf{K}}_{n+1}^{-1})$ returns a vector of the diagonal entries of $\hat{\mathbf{K}}_{n+1}^{-1}$ and "./" denotes element-wise division. $\mathbf{D}_{n+1}$ and $\mathbf{y}_{n+1}$ are found by removing the jth row of $\tilde{\mathbf{D}}_{n+1}$ and $\tilde{\mathbf{y}}_{n+1}$, respectively.

A.2. Kernel Normalized Least Mean Squares Algorithm

The KNLMS [Richard et al. 2009] provides a computationally efficient way to approximately calculate the least squares solution. The KNLMS algorithm also provides an effective and computationally inexpensive online method to sparsify the dictionary, known as the coherence criterion. The result is an online approximate kernel regression algorithm with worst-case computational complexity of $O(NM)$ to update the state for each new training example. KNLMS can be derived by considering the instantaneous approximation of the solution to the following affine projection problem at time

```
Initialize the step-size η, and the regularization factor ε
Insert x_1 into the dictionary, D_1, denote it by d_1. α̂_1 = 0, m_d = 1
while n > 1 do
  Get (x_n, y_n)
  if max_{j=1...m_d} |κ(x_n, d_j)| > μ_0 then
    Compute column vector
    k_n = [κ(x_n, d_1), ..., κ(x_n, d_m)]^T.
    Calculate α̂_n using Equation (13).
  else
    m_d = m_d + 1
    Insert x_n into the dictionary, denote it by d_m.
    Compute column vector k_n = [κ(x_n, d_1), ..., κ(x_n, d_m)]^T.
    Calculate α̂_n using Equation (14).
  end if
end while
```

Fig. 13. Pseudocode of the KNLMS algorithm with Coherence Criterion.

step n:

$$\min_{\boldsymbol{\alpha}} \|\boldsymbol{\alpha} - \hat{\boldsymbol{\alpha}}_{n-1}\|_2^2 \quad \text{subject to} \quad y_n = \mathbf{k}_n^T \boldsymbol{\alpha}, \tag{11}$$

where $\mathbf{k}_n$ is the kernel vector given by the kernel function between the latest input, $\mathbf{x}_n$, and each entry currently in the dictionary. Assuming that the current can be adequately represented by the current dictionary, and thus we don't wish to add it the dictionary, Equation (11) can be solved by minimizing the following Lagrangian function:

$$J(\boldsymbol{\alpha}, \lambda) = \|\boldsymbol{\alpha} - \hat{\boldsymbol{\alpha}}_{n-1}\|_2^2 + \lambda\big(y_n - \mathbf{k}_n^T \boldsymbol{\alpha}\big). \tag{12}$$

The solution of $\hat{\boldsymbol{\alpha}}_n$ is found by differentiating Equation (12) with respect to $\boldsymbol{\alpha}$ and λ and setting the derivatives to zero, $2(\hat{\boldsymbol{\alpha}}_n - \hat{\boldsymbol{\alpha}}_{n-1}) = \lambda \mathbf{k}_n$.

By premultiplying each term by $\mathbf{k}_n^T$ and substituting in y_n from the constraint in Equation (11), an expression for λ can be found: $\lambda = 2(\mathbf{k}_n^T \mathbf{k}_n)^{-1}(y_n - \mathbf{k}_n^T \hat{\boldsymbol{\alpha}}_{n-1})$. This is then used to obtain the following recursive update equation:

$$\hat{\boldsymbol{\alpha}}_n = \hat{\boldsymbol{\alpha}}_{n-1} + \frac{\eta}{\epsilon + \|\mathbf{k}_n\|_2^2}\big(y_n - \mathbf{k}_n^T \hat{\boldsymbol{\alpha}}_{n-1}\big)\mathbf{k}_n, \tag{13}$$

where η is a step-size parameter and ϵ is a regularization factor. If the current training example cannot be adequately represented by the current dictionary, then the dictionary is appended with the current input, $\mathbf{x}_n$, and the update equation becomes:

$$\hat{\boldsymbol{\alpha}}_n = \begin{bmatrix} \hat{\boldsymbol{\alpha}}_{n-1} \\ 0 \end{bmatrix} + \frac{\eta}{\epsilon + \|\mathbf{k}_n\|_2^2}\left(y_n - \mathbf{k}_n^T \begin{bmatrix} \hat{\boldsymbol{\alpha}}_{n-1} \\ 0 \end{bmatrix}\right)\mathbf{k}_n. \tag{14}$$

In order to determine whether or not the current dictionary can adequately represent the current input example, the coherence criterion is used, which is given by $\max_{j=1\ldots m_d} |\kappa(\mathbf{x}_n, \mathbf{d}_j)| \leq \mu_0$, where m_d is the current size of the dictionary. If true, the current input example is appended the dictionary; otherwise, the dictionary remains the same. Richard et al. [2009] show that under reasonable conditions, this produces a finite dictionary as n approaches infinity. This also implies that after an initial period, the computational cost for each training step only depends on the final size of the dictionary, M_d. Pseudocode for the KNLMS algorithm, adapted from Richard et al. [2009] and Yukawa [2012], is provided in Figure 13.

REFERENCES

Altera. 2016a. Altera Floating-Point IP Cores User Guide. (2016). http://www.altera.com.

Altera. 2016b. Altera Megawizard User Guide. (2016). http://www.altera.com.

Altera. 2016c. Altera Stratix V Device Handbook. (2016). http://www.altera.com.

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 (2011), 263–282. DOI:http://dx.doi.org/10.1142/S0218126611007244

Davide Anguita, Alessandro Ghio, Stefano Pischiutta, and Scitidro Ridella. 2007. A hardware-friendly support vector machine for embedded automotive applications. In *International Joint Conference on Neural Networks, 2007 (IJCNN'07)*. 1360–1364. DOI:http://dx.doi.org/10.1109/IJCNN.2007.4371156

Ray Bittner and Erik Ruf. 2012. Direct GPU/FPGA communication via PCI express. In *2012 41st International Conference on Parallel Processing Workshops (ICPPW'12)*. 135–139. DOI:http://dx.doi.org/10.1109/ICPPW.2012.20

Christopher H. Chou, Aaron Severance, Alex D. Brant, Zhiduo Liu, Saurabh Sant, and Guy G. F. Lemieux. 2011. VEGAS: Soft vector processor with scratchpad memory. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'11)*. ACM, New York, NY, 15–24. DOI:http://dx.doi.org/10.1145/1950413.1950420

Theodorus J. Dekker. 1971. A floating-point technique for extending the available precision. *Numerical Mathematics* 18, 3 (1971), 224–242. DOI:http://dx.doi.org/10.1007/BF01397083

Yaakov Engel, Shie Mannor, and Ron Meir. 2004. The kernel recursive least-squares algorithm. *IEEE Transactions on Signal Processing* 52, 8 (Aug. 2004), 2275–2285. DOI:http://dx.doi.org/10.1109/TSP.2004.830985

Jerome H. Friedman. 2006. Recent advances in predictive (machine) learning. *Journal of Classification* 23 (2006), 175–197.

Nicholas J. Higham. 1996. *Accuracy and Stability of Numerical Algorithms*. Number 48. Siam.

Andrew K. S. Jardine, Daming Lin, and Dragan Banjevic. 2006. A review on machinery diagnostics and prognostics implementing condition-based maintenance. *Mechanical Systems and Signal Processing* 20, 7 (2006), 1483–1510. DOI:http://dx.doi.org/10.1016/j.ymssp.2005.09.012

Jainik Kathiara and Miriam E. Leeser. 2011. An autonomous vector/scalar floating point coprocessor for FPGAs. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'11)*. 33–36. DOI:http://dx.doi.org/10.1109/FCCM.2011.14

Martin Langhammer and Bogdan Pasca. 2015. Floating-point DSP block architecture for FPGAs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'15)*. ACM, New York, NY, 117–125. DOI:http://dx.doi.org/10.1145/2684746.2689071

Daniel Le Ly and Paul Chow. 2010. High-performance reconfigurable hardware architecture for restricted Boltzmann machines. *IEEE Transactions on Neural Networks* 21, 11 (Nov. 2010), 1780–1792. DOI:http://dx.doi.org/10.1109/TNN.2010.2073481

Mingjie Lin, Ilia Lebedev, and John Wawrzynek. 2010. High-throughput Bayesian computing machine with reconfigurable hardware. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'10)*. ACM, New York, NY, 73–82. DOI:http://dx.doi.org/10.1145/1723112.1723127

John W. Lockwood, Adwait Gupte, Nishit Mehta, Michaela Blott, Tom English, and Kees A. Vissers. 2012. A low-latency library in FPGA hardware for high-frequency trading (HFT). In *2012 IEEE 20th Annual Symposium on High-Performance Interconnects (HOTI'12)*. 9–16. DOI:http://dx.doi.org/10.1109/HOTI.2012.15

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

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 (March 2012), 30 pages. DOI:http://dx.doi.org/10.1145/2133382.2133388

Yuri V. Makarov, Victor I. Reshetov, Vladimir A. Stroev, and Nikolai I. Voropai. 2005. Blackout prevention in the United States, Europe, and Russia. *Proceedings of the IEEE* 93, 11 (2005), 1942–1955. DOI:http://dx.doi.org/10.1109/JPROC.2005.857486

Yeyong Pang, Shaojun Wang, Yu Peng, N. J. Fraser, and P. H. W. Leong. 2013. A low latency kernel recursive least squares processor using FPGA technology. In *2013 International Conference on Field-Programmable Technology (FPT'13)*. 144–151. DOI:http://dx.doi.org/10.1109/FPT.2013.6718345

Markos Papadonikolakis and Christos-Savvas S. Bouganis. 2008. A scalable FPGA architecture for nonlinear SVM training. In *International Conference on ICECE Technology, 2008 (FPT'08).*. 337–340. DOI:http://dx.doi.org/10.1109/FPT.2008.4762412

Rafat Rashid, J. Gregory Steffan, and Vaughn Betz. 2014. Comparing performance, productivity and scalability of the TILT overlay processor to OpenCL HLS. In *2014 International Conference on Field-Programmable Technology (FPT'14)*. 20–27. DOI:http://dx.doi.org/10.1109/FPT.2014.7082748

Cedric Richard, J. C. M. Bermudez, and Paul Honeine. 2009. Online prediction of time series data with kernels. *IEEE Transactions on Signal Processing* 57, 3 (March 2009), 1058–1067. DOI:http://dx.doi.org/10.1109/TSP.2008.2009895

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

Aaron Severance, Joe Edwards, Hossein Omidian, and Guy Lemieux. 2014. Soft vector processors with streaming pipelines. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'14)*. ACM, New York, NY, 117–126. DOI:http://dx.doi.org/10.1145/2554688.2554774

Aaron Severance and Guy Lemieux. 2012. VENICE: A compact vector processor for FPGA applications. In *2012 IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'12)*. 245–245. DOI:http://dx.doi.org/10.1109/FCCM.2012.55

Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. 2010. FPMR: MapReduce framework on FPGA. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'10)*. ACM, New York, NY, 93–102. DOI:http://dx.doi.org/10.1145/1723112.1723129

Song Sun and J. Zambreno. 2009. A floating-point accumulator for FPGA-based high performance computing applications. In *International Conference on Field-Programmable Technology, 2009 (FPT'09)*. 493–499. DOI:http://dx.doi.org/10.1109/FPT.2009.5377624

Steven Van Vaerenbergh. 2012. 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.

Steven Van Vaerenbergh and I. Santamaria. 2013. A comparative study of kernel adaptive filtering algorithms. In *2013 IEEE Digital Signal Processing and Signal Processing Education Meeting (DSP/SPE'13)*. 181–186. DOI:http://dx.doi.org/10.1109/DSP-SPE.2013.6642587

Steven Van Vaerenbergh, I. Santamaria, Weifeng Liu, and J. C. Principe. 2010. Fixed-budget kernel recursive least-squares. In *2010 IEEE International Conference on Acoustics Speech and Signal Processing (ICASSP'10)*. 1882–1885. DOI:http://dx.doi.org/10.1109/ICASSP.2010.5495350

Steven Van Vaerenbergh, Javier Via, and I. Santamaria. 2006. A sliding-window kernel RLS algorithm and its application to nonlinear channel identification. In *Proceedings of the 2006 IEEE International Conference on Acoustics, Speech and Signal Processing, 2006 (ICASSP'06),* Vol. 5. V–V. 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 (Feb. 2005), 101–121. DOI:http://dx.doi.org/10.1002/spe.v35:2

Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. 2007. Top 10 algorithms in data mining. *Knowledge and Information Systems* 14, 1 (Dec. 2007), 1–37. DOI:http://dx.doi.org/10.1007/s10115-007-0114-2

Peter Yiannacouras, J. G. Steffan, and J. Rose. 2012. Portable, flexible, and scalable soft vector processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20, 8 (2012), 1429–1442. DOI:http://dx.doi.org/10.1109/TVLSI.2011.2160463

Jason Yu, Christopher Eagleston, Christopher Han-Yu Chou, Maxime Perreault, and Guy Lemieux. 2009. Vector processing as a soft processor accelerator. *ACM Transactions on Reconfigurable Technology Systems* 2, 2, Article 12 (June 2009), 34 pages. DOI:http://dx.doi.org/10.1145/1534916.1534922

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

Received May 2015; revised May 2016; accepted May 2016