PROMISE: An End-to-End Design of a Programmable Mixed-Signal Accelerator for Machine-Learning Algorithms

Prakalp Srivastava[†*] Mingu Kang[§*] Sujan K. Gonugondla[†] Sungmin Lim[†]
Jungwook Choi[§] Vikram Adve[†] Nam Sung Kim[†] Naresh Shanbhag[†]
[†]*University of Illinois at Urbana-Champaign* [§]*IBM Thomas J. Watson Research Center*
(psrivas2 gonugon2 sungmin3 vadve nskim shanbhag)@illinois.edu
mingu.kang@ibm.com choij@us.ibm.com

***Abstract*—Analog/mixed-signal machine learning (ML) accelerators exploit the unique computing capability of analog/mixed-signal circuits and inherent error tolerance of ML algorithms to obtain higher energy efficiencies than digital ML accelerators. Unfortunately, these analog/mixed-signal ML accelerators lack programmability, and even instruction set interfaces, to support diverse ML algorithms or to enable essential software control over the energy-vs-accuracy tradeoffs. We propose PROMISE, the first end-to-end design of a PROgrammable MIxed-Signal accElerator from Instruction Set Architecture (ISA) to high-level language compiler for acceleration of diverse ML algorithms. We first identify prevalent operations in widely-used ML algorithms and key constraints in supporting these operations for a programmable mixed-signal accelerator. Second, based on that analysis, we propose an ISA with a PROMISE architecture built with silicon-validated components for mixed-signal operations. Third, we develop a compiler that can take a ML algorithm described in a high-level programming language (Julia) and generate PROMISE code, with an IR design that is both language-neutral and abstracts away unnecessary hardware details. Fourth, we show how the compiler can map an application-level error tolerance specification for neural network applications down to low-level hardware parameters (swing voltages for each application `Task`) to minimize energy consumption. Our experiments show that PROMISE can accelerate diverse ML algorithms with energy efficiency competitive even with fixed-function digital ASICs for specific ML algorithms, and the compiler optimization achieves significant additional energy savings even for only 1% extra errors.**

***Keywords*-deep in-memory computing; analog ISA; programmable machine learning accelerator;**

I. Introduction

Current and emerging applications are increasingly relying on the ability to extract patterns from large data sets to support inference and decision making with Machine Learning (ML) algorithms. Such ML inference algorithms (or simply ML algorithms in this paper) have begun to offer higher performance than humans [1] in cognitive and decision-making tasks [2], but they have demanded more computing capability as they have gotten computationally more complex and required to process larger amounts of data. This coincides with the decline of Moore's Law, and thus it becomes far more challenging to meet such demands than ever. To close the gap between the demand and the offering from traditional general-purpose processors, researchers have begun to explore specialized processors (or accelerators) for ML algorithms [3]–[7]. These ML accelerators can offer orders of magnitude higher energy efficiency than general-purpose processors, but most of them are based on digital circuits and/or implement a specific ML algorithm.

In order to further improve energy efficiency of digital ML accelerators, researchers have proposed analog or mixed-signal accelerators [8]–[17]. These analog and mixed-signal accelerators rely on small-signal computation which is less precise but more energy efficient than traditional large-signal computation in the digital domain. Therefore, they are suitable for acceleration of ML inference algorithms where the application domain itself is tolerant to such imprecision. However, these accelerators lack a programmable architecture, instruction sets, or compiler support necessary for supporting application software. These capabilities are essential to support high-level programming languages like Python [18] and Julia [19], which implement popular ML libraries such as TensorFlow [20], and MXNet [21].

Moreover, the algorithmic error tolerance that allows hardware-level small-signal computations creates energy vs. accuracy tradeoffs that must be controlled from the application level in order to ensure that application-domain accuracy or precision goals are met. Translating these application-level metrics down to suitable hardware-level control knobs for energy and accuracy without requiring application programmers to understand hardware design concepts requires careful hardware, ISA and compiler design.

We propose PROMISE, the first end-to-end design of a programmable mixed-signal accelerator for diverse ML algorithms, which tackles all these challenges. PROMISE can accomplish a high level of programmability without noticeably losing the efficiency of mixed-signal accelerators for specific ML algorithms. PROMISE exposes instruction set mechanisms that allow software control over energy-vs-accuracy tradeoffs, and supports compilation of high-level languages down to the hardware. Specifically, we make

*First two authors contributed equally to the paper

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00015

following key contributions.

PROMISE Architecture and ISA: First, we identify prevalent operations in widely-used ML algorithms and key constraints in supporting these operations for a programmable mixed-signal accelerator. These include (**C1**) intrinsic sequentiality imposed on operations and (**C2**) high variations in delay across different types of operations. (**C1**) limits the number of possible programmable operations and (**C2**) significantly affects performance and energy efficiency of mixed-signal accelerators. Second, we explore PROMISE Instruction Set Architecture (ISA), which can expose these operations and constraints to a compiler, with a programmable mixed-signal accelerator architecture built with silicon-proven components for mixed-signal operations [9]. The hardware design and ISA include mechanisms to control the accuracy-vs-energy tradeoff by varying swing voltages, which can be controlled by compiler-generated code.

Compiler for PROMISE: First, we discuss design goals for a compiler when PROMISE aims to maximize energy efficiency while delivering a desired accuracy for a given ML algorithm. Particularly, the following two aspects of PROMISE explode the solution space to explore when generating code. (**A1**) PROMISE demands software to determine the accuracy of each mixed-signal instruction in given code, while a complex interplay among accuracy of instructions strongly affects energy efficiency and overall final accuracy of the code. (**A2**) PROMISE provides many possible compositions of mixed-signal operations even under the **(C1)** and **(C2)** constraints. Considering the large solution space, we reason that it is inherently impractical for users to manually generate code for a complex ML algorithm. Second, we develop a code generator that translates machine learning kernels down to sequences of mixed-signal operations. Third, for neural network algorithms in particular, we show how to automatically map programmer-specified end-to-end accuracy error tolerances down to hardware-level swing voltage parameter values for individual `Tasks`. A direct mapping is difficult, but we show how to break down the problem into two steps: (i) mapping error tolerance to computational bit precision (using an existing analysis [22]), and (ii) mapping bit precision to voltage swings (using simulation-based profiling). Putting these together, we develop a compiler which can take a broad range of ML algorithms described in a high-level programming language (Julia) and compile it to native PROMISE code. For neural network algorithms, we can translate a specified end-to-end error tolerance and compile it to PROMISE code that satisfies the error tolerance while approximately minimizing voltage swings (within available quantization levels).

Efficacy of PROMISE End-to-End Design: To demonstrate the efficacy of PROMISE, we first build energy and throughput models of PROMISE based on silicon-validated energy, delay and behavioral models of mixed-signal blocks. Second, we take nine popular ML algorithms, describe them in Julia, and generate the code with the PROMISE compiler. Our evaluation shows that PROMISE can offer $3.4-5.5\times$ lower energy and $1.4-3.4\times$ higher throughput than algorithm-specific digital accelerators at comparable inference accuracy. Lastly, the swing voltage optimized code by the PROMISE compiler further provides $4\%-20\%$ (geometric mean: 15%) lower energy than the unoptimized code for complex ML kernels.

Table I: Machine learning (ML) algorithms.

$f(D(W,X))$	Inner loop kernel $D(W,X) = \sum_{i=1}^{N} d(w[i],x[i])$	$f(\,)$
SVM	$\sum_{i=1}^{N} w[i]x[i]$	*sign*
Temp. Match. (L1)	$\sum_{i=1}^{N} \lvert w[i]-x[i]\rvert$	*min*
Temp. Match. (L2)	$\sum_{i=1}^{N} (w[i]-x[i])^2$	*min*
DNN	$\sum_{i=1}^{N} w[i]x[i]$	*sigmoid*
Feature extraction (PCA)	$\sum_{i=1}^{N} w[i]x[i]$	$--$
k-NN (L1)	$\sum_{i=1}^{N} \lvert w[i]-x[i]\rvert$	*majority vote*
k-NN (L2)	$\sum_{i=1}^{N} (w[i]-x[i])^2$	*majority vote*
Matched filter	$\sum_{i=1}^{N} w[i]x[i]$	*min*
Linear Regression	$\sum_{i=1}^{N} w[i]$	*accumulate*
	$\sum_{i=1}^{N} w[i]^2$	*accumulate*
	$\sum_{i=1}^{N} w[i]x[i]$	*accumulate*

II. Background

In this section, we analyze various ML inference algorithms to identify commonalities in their data flow, and then we describe a circuit of a programmable mixed-signal accelerator which is well suited for ML algorithms and we build an ISA (Section III) and a compiler (Section IV) on.

A. ML Algorithms

The ML algorithms involve repeated Vector Distance (VD) computations denoted by $D(W_j,X)$ between N-dimensional input vector X and weight vector W as depicted in [23]. Commonly used VD computations include the dot product, L1 distance (Manhattan distance), L2 distance (Euclidean distance), and Hamming distance for the ML algorithms including the Support Vector Machine (SVM), template matching, Deep Neural Network (DNN), k-Nearest Neighbor (k-NN), and matched filter as listed in Table I.

These ML algorithms have the following three data-flow properties in common. **(P1)** A single VD is obtained by first computing N element-wise Scalar Distances (SDs) $(d(w[j][i],x[i]))$ followed by an aggregation step such as a sum or average generating the final scalar VD $D(W,X) = \sum_{i=1}^{N} d(w[j][i],x[i])$. **(P2)** The VD between a single query vector X and multiple (say N_o) weight vectors W_js $(j=1,2,...N_o)$ needs to be computed. **(P3)** The VD goes through a simple decision function $f()$ such as sigmoid or ReLU to generate the decision y_j. Especially, the VD computation tends to

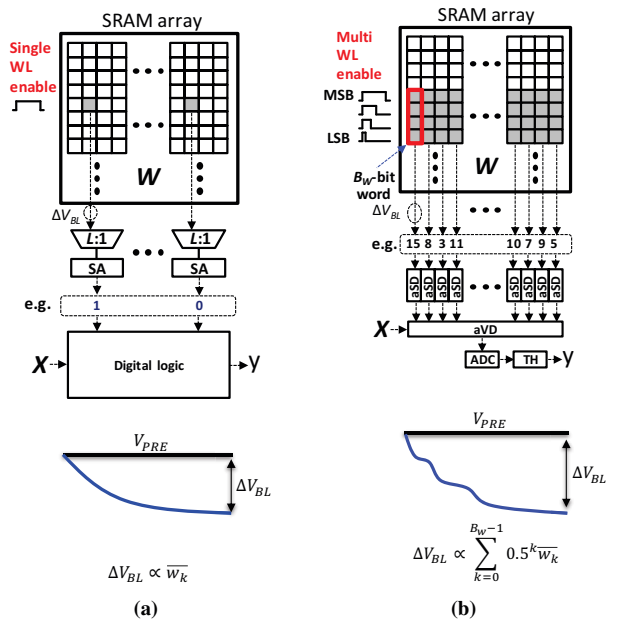


Figure 1: Block diagrams and read operations with bit-line swing (ΔV_{BL}): (a) conventional system, (b) Compute Memory, with bit precision of scalar weight w, $B_w = 4$ and column mux ratio $L = 4$, analog domain in red.

dominate the execution time and energy consumption of ML algorithms for practical problems.

B. Mixed-Signal ML Accelerator

Recently proposed compute memory (CM) [8]–[11], [24]–[26] deeply embeds energy-efficient analog computations into the periphery of conventional bit-cell array. As an innovative feature, CM can offer seamless interface between digital- and anlog-domain computations. More specifically, CM stores a B_w-bit word in a column-major format (i.e., a word is stored in B_w bit-cells connected to the same Bit-Line (BL) across B_w rows), as shown in the red box of Fig. 1(b)) whereas conventional memory does in a row-major format (Fig. 1(a)). After BLs are pre-charged for a read cycle, CM simultaneously asserts B_w Word-Lines (WLs). The durations of these asserted WLs are proportional to the binary weight values of the corresponding bit positions in a given B_w-bit word with a binary Pulse-Width Modulated (PWM) Word-Line (WL) signaling scheme (Fig. 1(b)). Subsequently, each BL develops a voltage drop (ΔV_{BL}) proportional to a binary weighted sum of B_w bits in the corresponding column [8], which constitutes the first processing stage of CM: **(S1)** analog Read (`aREAD`). `aREAD` can not only seamlessly convert digital values stored in memory into analog values for subsequent analog computation stages, but also fetches highly condensed B_w-bit information per BL, significantly improving energy efficiency and throughput.

For ML algorithms, CM can store pre-trained W_j in its bit-cells, then it can serve as a very energy-efficient mixed-signal ML accelerator with the following three subsequent analog processing stages: **(S2)** analog Scalar Distance (`aSD`) implementing scalar distance computations right next to the bit-cell array; **(S3)** analog Vector Distance (`aVD`) performing the aggregation ($\sum_{i=1}^{N}$ in Table I) by simply charge-sharing all the analog outputs from `aSD` blocks in one shot; and **(S4)** Analog-to-Digital Conversion (`ADC`) and ThresHold (`TH`) converting the analog output of `aVD` into a digital word and subsequently generating a final decision from the digital word based on a given decision function $f()$ in Table I. Note that the `aSD` stage can support scalar comparison, multiplication, subtraction, addition, and absolute computation in bit-cell pitch-matched analog circuitry [8], [25], while the `ADC` and `TH` stages consume negligible portion of total energy as they operate infrequently (once after $\geq$128 `aSD` operations).

It has been shown that the CM can offer significantly lower energy consumption and delay than digital ML accelerators, but it supports only limited reconfigurability [9]. Furthermore, the absence of an instruction set limits the use for a short sequence of operations with single computation kernel, single memory bank, and fixed parameters such as a vector length.

III. Instruction Set Architecture for Mixed-Signal Accelerators

In this section, we present PROMISE architecture as a substrate to explore ISA, discuss various challenges in developing ISA, and then propose ISA for a programmable mixed-signal accelerator.

A. PROMISE Architecture

Single Bank Architecture: PROMISE is built on CM as shown in Fig. 2(a), where the standard SRAM read and write functionalities are preserved (at the bottom) for additional flexibility. Along with **(S1)** `aREAD`, **(S2)** `aSD`, **(S3)** `aVD`, and **(S4)** `ADC` and `TH` described in Section II, PROMISE comprises `X-REG` and `CTRL` to transform CM into a programmable mixed-signal accelerator. The more detailed architectural specification of PROMISE is as follows.

A PROMISE bank consists of 256 (= N_{COL}) columns. An 8-bit (= B_w) word is distributed across four consecutive rows constituting a *word row* and two neighboring columns which store 4-bit MSB and 4-bit LSB to enhances linearity through a sub-ranged read technique [9]. That is, `aREAD` can read out a 128-element vector of digital values and seamlessly converts it to that of analog values. Furthermore, `aREAD` can simultaneously perform element-wise addition or subtraction with X, a 128-element vector representing the input operand for inference in Table I. **`aSD`** and **`aVD`** are designed to perform operations on 128 analog values. **`ADC`** consists of eight 8-bit ADCs which operate in parallel and convert $\approx$ 57 million analog values to digital values per second. Note that the `aVD` output of each bank is digitized by these ADCs to

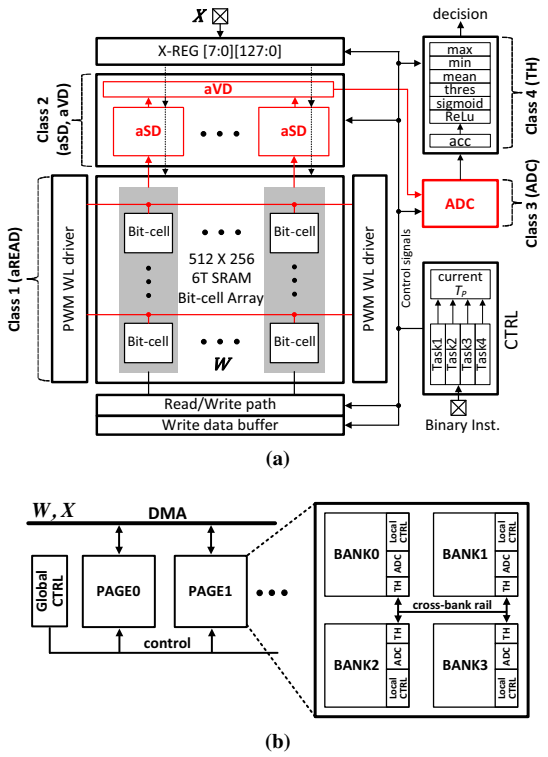


Figure 2: The PROMISE architecture: (a) a single-bank architecture with N_{ROW} = 512, N_{COL} = 256, analog processing marked in red, and (b) a multi-bank architecture.

prevent the noise from analog operations accumulating over the iterations. This digitization also enables a multiple-bank architecture, where reliable data transfers between banks are required. **`TH`** implements non-linear operations such as sigmoid via piece-wise linear approximation [27] not only to compute the decision functions $f()$ in Table I but also to aggregate intermediate computed values when the vector length N is larger than 128.

Lastly, **`X-REG`** is a digital block similar to a vector register file in a SIMD processor, holding eight 128-element vectors representing eight X values. **`CTRL`** is a controller to generate enable signals for the aforementioned components based on a given *instruction* and make CM function as a programmable mixed-signal accelerator.

Multiple bank architecture: PROMISE can be extended to a multi-bank structure (Fig. 2(b)), which has multiple (up to eight in this work) `PAGEs`, each of which includes four banks. Thus, long ($>$ 128) vectors can be distributed across multiple banks for parallel processing. PROMISE does not have the scalability limitation by the analog operations as the partial results from each bank are always converted to digital values by the ADCs. Then, the partial sums can be aggregated across different banks via cross-bank rail, similar to H-Tree in [12]. The data transfer of 8-bit ADC output from each bank to the other through the cross-bank rail

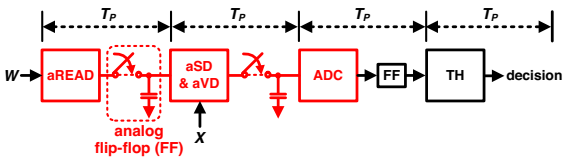


Figure 3: Analog pipeline in PROMISE with operating frequency = $1/T_P$, and red marked area is analog domain.

takes only 0.5 pJ with activity factor of 0.5 from post-layout simulations. This is negligible ($<$ 1%) as a `aREAD` consumes up to 131 pJ/bank as listed in Table III. The control of multi-bank by the instruction is discussed in Section III-C.

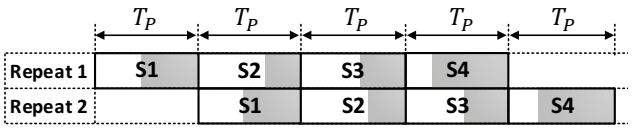
Analog pipeline architecture: To achieve higher throughput than the original CM [9], PROMISE adopts analog pipelining as shown in Fig. 3. This requires analog Flip-Flops (FFs) to hold the output of each stage. The first pipelined stage (`aREAD`) needs N_{COL} analog FFs whereas the second pipelined stage consisting of `aSD` and `aVD` needs only one analog FF to store the aggregated scalar output. Thus, we propose to reuse a pre-existing capacitor, which is a part of the analog multiplier [9] in the `aSD` block, as an analog FF for the `aREAD` stage to minimize the hardware overhead.

B. Challenges in Programmable Mixed-Signal Accelerator

The combination of algorithmic diversity and mixed-signal operations in PROMISE creates many challenges that we should consider when exploring ISA design choices. To support a broad range of ML algorithms, each stage needs to support diverse programmable operations. However, we identify some challenges in supporting (too) many diverse programmable operations which can significantly degrade both throughput and accuracy.

Higher Impact of Delay Variation across Operations: Analog operations often exhibit a wide range of delays depending on operation types [9], while each stage must accommodate the worst case delay out of all the operations for the stage. Furthermore, as all the stages of a pipelined mixed-signal accelerator need to operate at the same clock period T_P, T_P needs to accommodate the worst case delay across all the stages, i.e., $T_P = \max(T_{S1\_max}, T_{S2\_max}, T_{S3\_max}, T_{S4\_max})$, as illustrated in Fig. 4). That is, supporting more programmable operations causes longer idle time for some stages. As a result, we observe up to 2$\times$ throughput degradation when designing for the range of operations supported in this work. Furthermore, such long idle time negatively affects accuracy at the algorithm level because analog values are typically stored on (area-constrained) capacitors and thus degrade over time due to various leakage mechanisms. This problem is especially bad for the S1 stage (`aREAD`) as each BL is subject to the leakage contributions from all the bit-cells in the column (up to 0.6%/ns).

More Limit in Sequence of Operations: We also recognize some other challenges specific to analog processing,



$$T_P = \max(T_{S1\_max}, T_{S2\_max}, T_{S3\_max}, T_{S4\_max})$$

Figure 4: 4-stage processing in PROMISE with operational diversity per stage.

which further limits the number of possible programmable operations. Specifically, a chain of analog processing stages imposes intrinsic sequentiality of operations where two consecutive stages need to be physically closely placed. This is done to avoid substantial degradation in analog voltage from one stage to the next. Furthermore, a large capacitor ratio between consecutive stages (input-output 20:1 to maintain the voltage drop $< 5\%$) is required to transfer the signal via charge-transfer mechanism without an additional analog buffer.

C. PROMISE Instruction Set

We explore an ISA suitable for a programmable mixed-signal accelerator, considering the above constraints.

Instruction Format: We propose a wide-word macro instruction format, which is referred to as `Task`. Akin to a Very-Large Instruction Word (VLIW), a single `Task` consists of multiple operations, *except that the operations are sequential and not parallel like VLIW architectures*. As depicted in Fig. 5(a), the four `Class` fields specify four operations for four pipelined stages of PROMISE, while the three other fields, `OP_PARAM`, `RPT_NUM` and `MULTI_BANK` configure all or specific `Class` operations. More specific descriptions of these seven fields are as follows.

Operating Parameter Field: `OP_PARAM` comprises 33 bits and configures operating parameters of `Class` operations in a given `Task`, facilitating *flexible programmability*. As shown in Fig. 5(b), `W_ADDR` specifies a CM address for a `Class-1` operation. `X_ADDR1` and `X_ADDR2` designate `X-REG` addresses for `Class-1` and `Class-2`, respectively. `SWING` controls BL swing ΔV_{BL}, e.g., `111` allows 30 mV/LSB whereas `001` allows 5 mV/LSB. This parameter is a key knob to control the trade-off between energy and accuracy under software control; Section VI evaluates this accuracy-energy trade-off for further energy savings. Refer to Fig. 5(c) for descriptions of the remaining parameters and their assignments of bit fields.

Class Fields: `Class-1` is composed of 3-bit opcode and defines six possible memory operations. `READ`, `WRITE`, or `aREAD` makes CM perform a digital read, digital write, or analog read operation to a compute-memory address specified by `OP_PARAM` (`W_ADDR`). `aADD` or `aSUB` fuses an analog read and an element-wise analog addition or subtraction into a single operation where two vector operands come

Task

OP_PARAM (28 bits)	RPT_NUM (7 bits)	MULTIBANK (2 bits)	Class-1 (3 bits)	Class-2 (4 bits)	Class-3 (1 bits)	Class-4 (3 bits)

(a)

Contents	Bits	Description
SWING	[27:25]	ΔV_{BL} swing code – 000: min (5 mV/LSB), 111: max (30 mV/LSB)
ACC_NUM	[24:23]	# of operands to be accumulated for `accumulate` opcode in `Class-4`
W_ADDR	[22:14]	Bit-cell array address of W in `Class-1`
X_ADDR1	[13:11]	Bit-cell array address of X in `Class-1`
X_ADDR2	[10:8]	`X-REG` array address of X in `Class-2`
X_PRD	[7:6]	`X_ADDR1` & `2` circulate from 0 to "`X_PRD` - 1"
DES	[5:4]	`Class-4` output destination - 00: `ACC` input, 01: output buffer, 10: `X-REG`, 11:Write data buffer
THRES_VAL	[3:0]	Thresholding reference value for threshold opcode in `Class-4`

(b)

Class	Operation [operand]	Bit length	OP CODE	Option
1	*none*	3 bits (OPCODE)	000	*X_PRD*: *X_ADDR1* circulates from 0 to "*X_PRD*-1"
	write[W_ADDR]		001	
	read[W_ADDR]		010	
	aREAD[W_ADDR]		011	
	aSUBT[W_ADDR, X_ADDR1]		100	
	aADD[W_ADDR, X_ADDR1]		101	
2	*none*	4 bits (OPCODE + aVD)	000	aVD bit: 0: no aggregation 1: aggregation *X_PRD*: *X_ADDR2* circulates from 0 to "*X_PRD*-1"
	compare		001	
	absolute		010	
	square		011	
	sign_mult[X_ADDR2]		100	
	unsign_mult[X_ADDR2]		101	
3	*none*	1 bit (OPCODE)	0	
	ADC		1	
4	*accumulation*	3 bits (OPCODE)	000	*DES, ACC_NUM*
	mean		001	*DES*
	threshold		010	*DES, THRES_VAL*
	max		011	*DES*
	min		100	*DES*
	sigmoid		101	*DES*
	ReLu		111	*DES*

(c)

Figure 5: Instruction set of PROMISE: (a) instruction format, (b) operation parameters (`OP_PARAM`), and (c) operations in each `Class`.

from compute-memory addresses specified by `OP_PARAM` (`W_ADDR` and `X_ADDR1`), respectively.

`Class-2` consists of 4-bit opcode and specifies a composition of one of six possible `aSD` operations with one of two possible `aVD` operations. Specifically, `aSD` operating on a computed value from `Class-1` supports three unary operations: `compare`, `absolute`, and `square` and two binary operations: `sign_mult` and `unsign_mult` where the other operand comes from an `X-REG` address specified by `OP_PARAM` (`X_ADDR2`). `aVD` specifies whether an aggregation should be performed or not after an `aSD` operation.

`Class-3` and `Class-4` comprise 1- and 3-bit opcode and control whether an `ADC` should be performed or not and specify one of seven possible `TH` operations,

respectively. The seven possible TH operations are as follows: accumulation, mean, threshold, max, min, sigmoid, and ReLu.

In summary, Class-1,2,3 define a distance computation, $D(W_j,X)$ in Table I in the analog domain, while Class-4 specifies $f(D(W_j,X))$ in the digital domain.

Loop Control Field: RPT_NUM comprises 7 bits and specifies how many times the Task should be executed to process multiple W_js. The CM and X-REG addresses (W_ADDR, X_ADDR1 and X_ADDR2) are incremented sequentially every iteration. Although unconventional for modern RISC architectures, this is a natural choice for typical ML inference algorithms, which iterate sequentially through data W_js in memory for the computation $D(W_j,X)$.

Multiple Bank Control Field: MULTI_BANK comprises 2 bits and specifies the number of banks used to distribute long (>128) vectors for parallel processing. The intermediate results from banks 1, 2,..., and $2^{MULTI\_BANK}-1$ are transferred to bank 0 through the cross-bank rail in Fig. 2(b) to be aggregated by digital TH accumulation operation. The long vector needs to be distributed to the same row of each bank to support the parallel processing across multiple banks. The instruction is shared by 2^{MULTI_BANK} banks as those banks process the same operation. The output of a Class-4 operation can be transferred to the X-REG of a specific bank in any PAGE by defining *DES_ADD* in OP_PARAM.

Extension to Large Scale Applications: PROMISE is well suited to process 128 dimensional vector processing. Longer vectors (>128) can be processed by repeating the 128 dimensional vector processing sequentially by setting RPT_NUM = (*W*.size() / 128) and other parameters. A word row of CM stores 128 words, and thus two word rows are used to store 256 dimension vector. Two consecutive iterations complete a vector processing. The address W_ADDR and X_ADDR1 (or X_ADDR2) are incremented as the Task iterates to process the next 128 words. However, the X is re-used to compute the distances to many Ws as explained in Section II. Thus, the X_ADDR1 and X_ADDR2 circulates from 0 to X_PRD-1. For example, X_PRD = 2 to process 256 element vector X.

D. Algorithm Mapping and Compiler Need

A ML algorithm sometimes requires several Tasks for different distance metrics. We present an example of template matching with L1 distance kernel to find the closest 512-pixel image out of 127 candidate images (W_j) to input query image (X) by processing across four banks in parallel. The template matching is mathematically defined as:

$$j_{opt} = \arg\min_j \sum_{i=1}^{512} |x[i]w[j,i]| \quad (1)$$

The corresponding Task instruction consists of: RPT_NUM = 127 specifying the number of candidate images; MULTI_BANK = 4 to distribute 512 pixels into four banks (128 pixels per bank) for parallel processing; Class-1 aSUBT to perform element-wise subtraction of X with W_j; Class-2 absolute with aggregation; and Class-3 ADC followed by a digital-domain Class-4 min to compute $f() = \arg\min_j$.

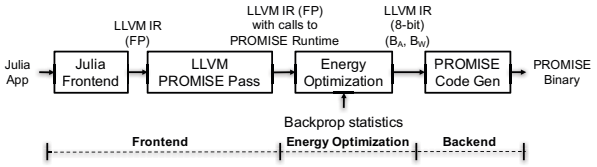


Figure 6: Compiler Pipeline. LLVM(FP) implies that data arrays are in floating point. 8-bit implies fixed point data arrays

Although each Class offers a limited number of operations (6, 12, 2 and 7, respectively), PROMISE can perform more than 1000 compositions of operations for a given X value. Furthermore, we observe that the order of Task instructions in a complex ML inference algorithm and the accuracy setting of each Task through the SWING parameter significantly affect the accuracy at the algorithm level, exploding the solution space for code generation. Hence, it is not efficient and inherently sub-optimal to manually generate code for a complex ML inference algorithm. This in turn gives a compiler an opportunity to generate and optimize code implementing a given ML inference algorithm, which will be explored in the subsequent section.

IV. Compiler

In this section, we discuss the goals of a compiler for PROMISE, describe how PROMISE compiler design meets these goals while translating a given ML algorithm described in a high-level language into the PROMISE ISA, and how the compiler addresses the programmability challenges mentioned in Section III-B.

A. Goals

Hardware Abstraction: The compiler intermediate representation (IR) should abstract away low-level details of the hardware, so that front ends don't have to be concerned about hardware details like Class-1 vs. Class-2 vector operations, or the specifics of the OP_PARAM parameters. At the same time, the IR should enable compilers to perform optimizations, and should capture essential information for generating efficient code on PROMISE. This is similar to how mid-level compiler IRs abstract away details like finite register files and different register classes from front ends, while enabling sophisticated register allocation algorithms to manage these details [28].

Accuracy-Energy Tradeoff: Although the PROMISE ISA allows the compiler to use the SWING parameter as a knob

to tune ΔV_{BL} to exploit the tradeoff between energy and accuracy, it is difficult to go from a high level description of "accuracy" that an application programmer understands in the context of the algorithm to hardware specific parameters such as voltage swings. This is especially true for applications which would have several smaller computations to offload (for example, DNNs can offload each layer computation) to a hardware accelerator. It is even more difficult to reason about how the error in a single computation would affect the overall accuracy of the application. Towards this end, the PROMISE compiler must provide a compiler optimization to determine the optimal `SWING` parameter for each `Task`, starting with some application-level specification of accuracy.

Hardware Specific Optimizations: Furthermore, for applications where data does not fit into PROMISE memory, the compiler needs to find an efficient dataflow pattern based on the size of data arrays.

Easily Extensible to ML Domain Specific Languages: Domain specific languages (DSLs) and libraries for ML are evolving fast: there are already a wide range of popular DSLs such as Torch, Theano, TensorFlow, MXNet, Keras, and others [20], [21], [29]–[31], implemented on top of dynamic programming languages such as Python, Julia, R, Scala, Perl, etc. Thus, a desirable goal of the PROMISE compiler is to be easily extendable to new DSLs. To achieve this, we must provide a language-neutral IR as an interface for the compiler/programmer.

B. AbstractTask and PROMISE Compiler IR

In this section, we first define an `AbstractTask` which is an abstraction of a PROMISE `Task` described in Section III-C. `AbstractTask` is based on our observation that a vector operation can be either a `Class-1` (addition/-subtraction) or a `Class-2` (signed/unsigned multiplication) operation, but that distinction is only relevant for late stage code generation, not front ends and other compiler optimizations. `AbstractTask` is also oblivious to hardware-specific parameters such as the number of elements in a vector (i.e., the length of the bit-cell array), size of the bit-cell array, etc.

An `AbstractTask` has the following ten fields: **(F1)** `W`: address of a 2D data array; **(F2)** `X`: address of a 1D data array; **(F3)** `output`: address of the output 1D data array; **(F4)** `vecOp`: element-wise vector operation between a row of `W` and `X`; **(F5)** `redOp`: reduction operation on the output of `vecOp`; **(F6)** `digitalOp`: unary operation on the output of `redOp`; **(F7)** `vectorLen`: number of elements in `X`; **(F8)** `loopIterations`: number of iterations of the loop executed by the task; **(F9)** `threshold`: threshold value for `Class-4 threshold` operation of `Task`; and **(F10)** `swing`: voltage swing at which this should run on PROMISE. The `swing` field is initialized to the value `0b111` (maximum accuracy) by the frontend, and is fine-tuned later by the energy optimization pass as described in Section IV-D.

The PROMISE compiler IR is a directed acyclic graph (DAG) of such `AbstractTask`s, where each node represents an `AbstractTask`, and a directed edge between two nodes `P` and `C` represents the dataflow from the output of *P* to the input (`W` or `X`) of *C*. For example, for a DNN inference algorithm of fully connected layers, the compiler IR would be a sequential pipeline of `AbstractTask`s. Each `AbstractTask` would represent the computation of the corresponding fully connected layer of DNN.

The IR is acyclic even though a task is always an iterative computation because the loop count is simply represented as the `RPT_NUM` field of a task (Fig. 5(a)). Loops surrounding a sequence of one or more tasks are always executed on the host processor and not on the PROMISE accelerator.

C. Code Generation

Fig. 6 shows the PROMISE compiler pipeline for Julia. There are three parts: (1) a front end to map Julia applications to the IR, (2) energy optimizations on the IR, and (3) a back end to translate the IR to the PROMISE ISA. The IR, energy optimizations and back end are all designed to be independent of the source-level language, to make it easily extendable to other languages or DSLs.

Frontend (Julia program to PROMISE compiler IR): We chose Julia as the source language because such a high-level language enables us to easily identify patterns of computations that can be offloaded to PROMISE. The ML kernels are built using high-level library calls to matrix/vector operations, and so we do not need to use sophisticated compiler analysis to identify these operations. Julia also supports several ML libraries such as MXNet [32], Flux [33], and others and is already used to develop ML applications. (Julia also has a working open-source LLVM front-end. Other choices like TensorFlow didn't have one available at the time we started this work.)

The Julia frontend translates applications to LLVM IR. The PROMISE pass runs over each LLVM function and uses pattern matching to identify computations that can be offloaded to PROMISE, translates it into an `AbstractTask`, and replaces the computations with a call to the PROMISE runtime to pass the parameters of the `AbstractTask`.

The LLVM IR [34] representation is a collection of Single Static Assignment (SSA) [35] dataflow graphs, one graph per function in a program. Each node in an SSA graph corresponds to an LLVM IR instruction. In each function, the PROMISE pass looks for matrix-matrix operations with a reduction component (e.g., matrix-matrix multiplication, which uses the dot product). When it finds one, it also looks for a single basic block "natural loop" [28] enclosing it, using an existing, widely used analysis in LLVM. The loop analysis also identifies the induction variable of the loop. If such a loop is found, the PROMISE pass uses the loop and induction variable information to check if the SSA graph of the basic block matches with the SSA graph pattern shown

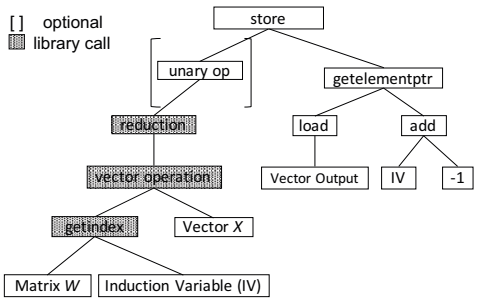


Figure 7: SSA Pattern for single basic block loops. The shaded nodes are calls to Julia library. The part enclosed in square brackets is optional for pattern matching.

in Fig. 7, and explained below. If matched, the single-basic-block loop can be offloaded to PROMISE. Since the pattern matching can be fragile to minor variations in generated LLVM IR, e.g., the loop index variable being incremented instead of decremented in each loop iteration, the compiler converts all single basic block loops into canonical loops before pattern matching.

Starting near the bottom left of the SSA graph, library call `getindex` is used to get the IV_{th} vector of matrix W. Vector X, which must be loop invariant (i.e., it must have no definitions inside the loop), is used to perform an element-wise vector operation with vector W_{IV}. X being loop invariant is essential for the computation to be efficient on PROMISE: X gets stored in the buffer `X_REG` designed to hold a vector with temporal locality and is constant throughout the execution of a `Task`. The output of the vector operation undergoes a `reduction` operation using a Julia library function call. The right child of the `store` SSA node uses the `getelementptr` instruction in LLVM to compute the address where computed value needs to be stored in the vector *Output*.

This pattern captures many widely used ML inference kernels, like template matching, support vector machines, k-nearest neighbor, matched filtering, matrix-vector multiplication, etc., and is used as a canonical form which can be mapped to an `AbstractTask`.

Translation of the SSA graph to an `AbstractTask` is straightforward. The SSA nodes `Matrix W`, `Vector X` and `Vector Output` map to the `W`, `X`, and `Output` fields of a `AbstractTask`, respectively. `loopIterations` can be computed at run-time from the induction variable `IV` and corresponding conditional branch at the end of the basic block. `VectorLen` can be obtained from `X`. The `swing` field is calculated as described in Section IV-D. After mapping to an `AbstractTask`, we replace the loop from the LLVM IR representation with a call to the PROMISE runtime, passing it the fields of *AbstractTask*.

Backend (compiler IR to PROMISE ISA): The backend of the compiler translates the compiler IR to the PROMISE ISA by mapping each `AbstractTask` to an appropriate `Task`. This involves two parts: (1) compile time code generation for `Class 1-4`, and (2) computing the `OP_PARAM`, `RPT_NUM`, `MULTI_BANK` fields at runtime and passing them to the PROMISE run-time, which runs on the host and launches the `Task`.

Code generation for `Class 1-4` is relatively straightforward: most fields have a one-to-one mapping to the corresponding field of the ISA. For example, the digital operation directly maps to the `Class-4` field, and the `Class-3` operation is always `ADC`. Primarily, the backend identifies where the `vecOp` of `AbstractTask` would execute - `Class-1` or `Class-2`. If the `vecOp` is element-wise addition or subtraction, `Class-1` operation is set to `aADD` or `aSUB` , respectively, and `Class-2` performs just the reduction of the resulting values according to `redOp` field of `AbstractTask`. Otherwise, for (un)signed multiplication, `Class-1` performs the `aREAD` and `Class-2` performs the element-wise vector multiplication with `X` operand from `X_REG` and performs the aggregation as well.

The `OP_PARAM`, `RPT_NUM` and `MULTI_BANK` fields require runtime information and are computed on the host before `Task` launch. For vector lengths > 128, `X_PRD` is set to the number of rows required to fit one vector, i.e., `vectorLen/128`. `RPT_NUM` is set to the product of the number of loop iterations and the number of rows per vector, i.e., `X_PRD * loopIterations`. Setting other fields such as `W_ADDR` in `OP_PARAM` is straightforward and we skip those for lack of space.

D. Energy Optimization

In the ML domain, classification accuracy of a model is an important metric. For instance, neural networks for a handwritten digit recognition can achieve classification accuracy of $p_{model} = 98\%$ on the MNIST [36] dataset running on floating point architectures such as GPUs. Moreover, the broader application context that uses the results of the classifier can often tolerate lower accuracy, and that is application-specific. In our work, we allow the source-level programmer to express the additional error they can tolerate when running their model, which we call the mismatch probability (p_m). As deterministic errors can be tolerated easily by re-training the parameters in ML algorithms, we focus on spatial random errors across bit-cells from process variations in this section. Formally, p_m is the upper bound on difference between the classification accuracy of an algorithm running on PROMISE ($p_{PROMISE}$) and the classification accuracy of the ML model (p_{model}), i.e.,

$$p_{model} - p_{PROMISE} \leq p_m \tag{2}$$

The energy optimization in the compiler pipeline in Fig. 6 takes the mismatch probability p_m from the program and determines the `swing` field of each `AbstractTask` in

the application that would ensure that error tolerance is met. Mapping a high-level parameter like p_m directly to a suitable swing voltage is difficult, and is even more challenging for algorithms such as neural networks that have multiple `Tasks`. We solve this problem by breaking it down into two parts, taking advantage of prior work [22] for the first part: (a) determining a minimum bit precision required to achieve the given mismatch probability, using the results of [22] (note that this is an algorithmic property and is hardware-independent); and (b) mapping the required bit precision to the hardware swing voltage, which is a property of PROMISE. These are explained briefly below, second one first.

To achieve B-bit precision in the final output, the error introduced must be less than $1/2^{B+1}$. The major source of error in PROMISE due to lowering the swing voltage is from `aREAD` operations (see Section III). The output of `aREAD` follows a normal distribution $\hat{W} \sim \mathcal{N}(W, \sigma_W^2)$, where $\sigma_W = |W| \cdot f(\texttt{SWING})$ and $f(\texttt{SWING})$ is a function of the `SWING` parameter and ranges from 0.08 ~ 0.75. The `SWING` parameter and $f(\texttt{SWING})$ are inversely proportional, and hence, the σ_W is minimized with higher `SWING` parameter. After the aggregation of N such vector elements through charge-sharing, the standard deviation of the aggregated value (σ_{agg}) of output is $\sigma_W/\sqrt{N}$. Since W is in range [-1, 1], we assume $|W| = 1$ for all values to maximize σ_W and σ_{agg}. In this paper, we choose a confidence level of 99%, which corresponds to $2.6 \times \sigma_{agg}$. To guarantee B-bit precision at the output of aggregation, this yields:

$$2.6\sigma_{agg} = 2.6\frac{f(\texttt{SWING})}{\sqrt{N}} < \frac{1}{2^{B+1}} \tag{3}$$

If we can estimate the required bit precision for a given p_m (part (a), above), we can use (2) to compute the minimum swing voltage. To achieve (a), we leverage prior work [22]. In that work, Sakr et al. analyze the quantization (floating-point to fixed-point conversion) tolerance of neural networks and give a relationship between the accuracy degradation and the bit precisions used to store the activations and weights of a neural network model. Mathematically, it says that given the bit precision of weights and activations (B_W and B_A), they can provide a bound on the mismatch probability $p_{fl} - p_{fp}$, where p_{fl} is the accuracy of the floating-point model, and p_{fp} is the accuracy of the same model quantized to fixed-precision representation for weights and activations. The analysis bounds the mismatch probability by

$$p_m \leq \Delta_A^2 E_A + \Delta_W^2 E_W, \tag{4}$$

where E_A and E_W are statistics of the model obtained while training the model, and $\Delta_A = 2^{-(B_A-1)}, \Delta_W = 2^{-(B_W-1)}$.

Since neural network inference is simply a series of repeated *vector distance* computations described in Section II, computation of activations in each layer can be described as a PROMISE `AbstractTask` with W equal to a weight matrix, and vector X equal to the activations of the previous layer.

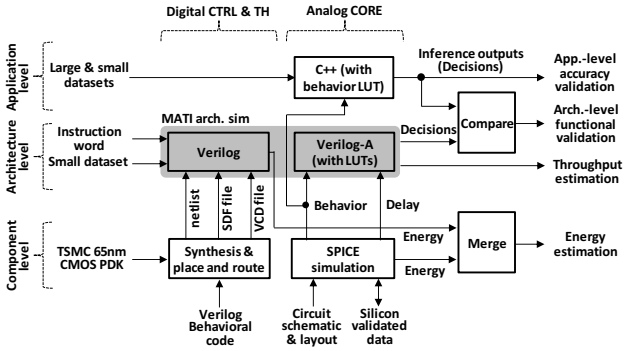


Figure 8: PROMISE validation methodology.

The vector operation is element-wise multiplication and the reduction operation is the sum of all the values in the result. The activation function such as `sigmoid/ReLU/tanh` are supported as `Class-4` operations in PROMISE. Thus, a neural network inference is a sequence of `AbstractTasks`.

In the context of PROMISE, we can use Sakr's model to calculate the bit precision B_X for X in each `AbstractTask` given the mismatch probability p_m for the model (sequence of `AbstractTasks`), weight precision $B_W = 7$ since the PROMISE bit-cell array uses 8-bits to store a value, including one sign bit. Notice that Sakr's model supports multiple layers by assuming equal bit precision for all layers, effectively distributing the error tolerance across the layers.

Putting it all together, we use the back propagation statistics, E_A and E_W of the trained application model, along with the desired p_m, as input to analysis of [22] to estimate B_A and B_W. We then use B_X and `vectorLength` as input to (2) to estimate the minimum swing voltage, which we pass as the `Class-0 SWING` parameter to the PROMISE run-time when launching the `Task`.

The analysis above has focused on neural networks because the results of [22] used in step (a) is limited to neural networks. (The other steps do not have this limitation.) For other combinations of `Class-1` and `Class-2` operations, we would have to extend Sakr's analysis [37]. Doing so is straightforward, but is outside the scope of this paper. Moreover, we can still optimize kernels with only a single `AbstractTask` by using a brute-force sweep through all eight `swing` voltage levels to look for the optimal value.

V. Validation Methodology

This section describes our methodology for validating PROMISE's energy, delay, and accuracy benefits as well as the benefits of the compiler generated code. Fig. 8 summarizes our validation methodology. Specifically, we (a) develop energy, delay, and behavioral models of PROMISE components in TSMC 65 nm GP process including analog non-idealities; (b) incorporate these component-level analog models (in Verilog-A) with Verilog model of the digital CTRL

to ensure correct functionality and estimate accuracy over small data sets; and (c) develop a PROMISE C++ model with component level behavioral models for verifying accuracy over large data sets of compiler generated code.

Component-level Models: The entire mixed-signal chain was post-layout simulated in SPICE to obtain the energy and delay numbers listed in Table III. The total energy and delay for the mixed-signal blocks were compared with the measured results reported in [9] and the differences were found to be within 10% and 9%, respectively. The deterministic errors of the analog components are extracted from measurement in the form of look-up tables (LUTs). On the other hand, the spatial random error across bit-cells due to the process variation was extracted from Monte-Carlo SPICE simulations to obtain statistically sufficient number of samples. Behavioral models incorporating these non-ideal analog effects and delays were then incorporated into component-level Verilog-A models for analog blocks. Verilog models of all the digital components including `CTRL` and `TH` block were developed and synthesized with the same library via Synopsys Design Compiler.

Architecture-level Validation: Verilog and Verilog-A models described in Section V were integrated to obtain a cycle and functionally accurate PROMISE Verilog model. The digital blocks were verified by generating the correct control signals at the right time in the presence of post-layout parasitics when presented with the appropriate PROMISE instruction word for small data sets.

Application-level Validation: A functional PROMISE C++ model incorporating the LUT-based analog behavioral models described in earlier in Section V was also developed. This C++ model was run on large data sets to obtain PROMISE's application-level accuracy. The compiler generated code was verified with the Verilog models of the digital components along with the Verilog-A models to evaluate energy and accuracy of PROMISE.

Benchmarks: The commonly employed ML algorithms listed in Table II were mapped to PROMISE. As PROMISE is programmable, it employs 8-bit data to cover diverse applications and algorithms as shown in many other implementations [1], [7], [38], [39]. For application level energy optimization analysis, we also choose three variants of DNN of different complexity to demonstrate the architecture. These benchmarks give us diversity in complexity and allows us to explore the energy accuracy trade-offs.

Baseline Architectures: It is well known that ASICs are order-of-magnitude more energy and delay-efficient compared to general-purpose processors (CPU/GPU) [47]. Therefore, we choose the following four ASICs as our comparison baselines for the conservative evaluation of PROMISE: (a) **CM**: The programmability overhead is estimated via the comparison with CM. (b) **State-of-arts**: We also compare PROMISE with the recent prior arts with silicon IC prototypes [6], [7], which implement similar algorithms as PROMISE. The DNN is compared with [6] and k-NN and template matching with L1 and L2 distances are compared with [7]. (c) **CONV-8b**: We build the baseline digital architecture (Fig. 9) with the 8-b fixed point computational logic synthesized for the specific algorithm + conventional SRAM. (d) **CONV-OPT**: This is the same as CONV-8b but with minimum bit precision required per benchmark.

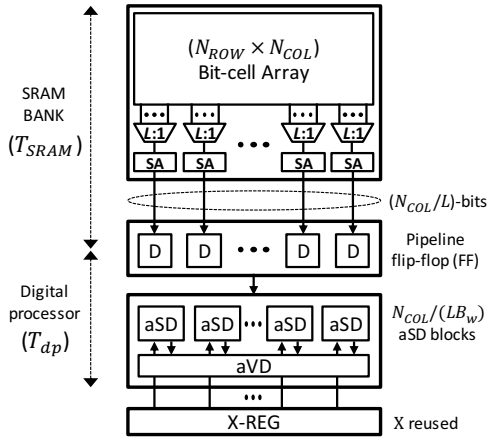


Figure 9: Single bank of CONV-8b with $L = 4$, $N_{ROW} = 512$, $N_{COL} = 256$, where an SRAM communicates with an algorithm-specific synthesized digital processor over a pipelined interface.

Even though prior arts exist for some of benchmarks, many of them do not have a relevant previous ASIC. Furthermore, configurations such as process technology and on-chip memory capacity are not perfectly identical to PROMISE. In order to perform a conservative comparison, the CONV-8b/CONV-OPT (CONV) is chosen to operate at maximum speed while only restricting the number of SRAM banks employed to be same as PROMISE. The SRAM fetches $N_{COL}/(LB_w)$ words per single read access of bank.Therefore, CONV operates with maximum achievable throughput of:

$$f_{CONV} = \left(\frac{N_{COL}/L}{B_w}\right)\left(\frac{1}{T_{SRAM}}\right) \tag{5}$$

The CONV (Fig. 9) consists of computation logic synthesized for each specific benchmark therefore it only incurs the energy costs of that specific benchmark and additional routing, dataflow and control energy are neglected. Additionally, CONV-OPT has the minimum precision for each benchmark thereby making it the most conservative baseline to compare with PROMISE in terms of accuracy, energy and throughput.

VI. Evaluation

In this section we present evaluation of PROMISE. We estimate the energy and delay for each operation using the methodology presented in Section V. We present the gains of the compiler-based energy optimization, compared with the

Table II: Benchmarks for PROMISE Simulations [36], [40], [41].

Algorithm	Application	Database	Data Size (N)	Number of Categories	Problem Size	# of AbstractTasks	AbstractTask	W	X	Comments	Opt. swing ($p_m = 1\%$)
DNN (Multilayer Perceptron)	Hand-written character recognition	MNIST	(8-bit) 22×23	10	10 categories, 60000 training samples, 10000 test samples	4	vecOp: multiplication redOp: sum	Weights	Test samples	5-layer DNN with nodes as follows: 784-512-256-128-10	3,2,3,3
Matched filtering	Event (gun-shot) Detection	Gun-shot Mono sound	(8-bit) 256 512 1024	2	100 test vectors	1	vecOp: multiplication redOp: sum	Filter weights	Test samples		1
Template matching (w/ L1 & L2)	Face Recognition	MIT-CBCL	(8-bit) 16×16 22×23 32×33	64	256 Candidates	1	vecOp: subtraction redOp: L1 – absolute L2 – square	Candidate faces	Test samples	Nearest candidate based on either L1 or L2 distance	2
Linear SVM	Face Detection	MIT-CBCL	(8-bit) 16×16	2	2 categories, 2000 training samples, 858 test samples	1	vecOp: multiplication redOp: sum	Weights	Test samples	Face data converted into a vector, linear SVM applied on it	6
k-NN (w/ L1 & L2)	Hand-written character recognition	MNIST	(8-bit) 16×16 22×23 32×33	10	10 categories, 54210 training samples, 200 test samples	1	vecOp: subtraction redOp: L1 – absolute L2 – square	Training samples	Test samples	Sorting is done in external processor after processing on PROMISE	1
Feature extraction (PCA)	Face Detection	MIT-CBCL	(8-bit) 16×16	–	2000 samples	1	vecOp: multiplication redOp: sum	Weights	Samples	Four features used for face detection based on PCA	–
Linear regression	Modeling linear predictor	Synthetic data	(8-bit) 2 dim.	–	8192 samples	4	vecOp: None redOp: AT1, AT2 – mean AT3 – square AT4 – sign_mult	AT1: U AT2: V AT3: U AT4: U	AT4: V	2-D linear regression : $slope = \frac{\Sigma uv - \Sigma \bar{u}\bar{v}}{\Sigma u^2 - \Sigma \bar{u}^2}$ $y\text{-}intercept = \bar{v} - slope \cdot \bar{u}$	–

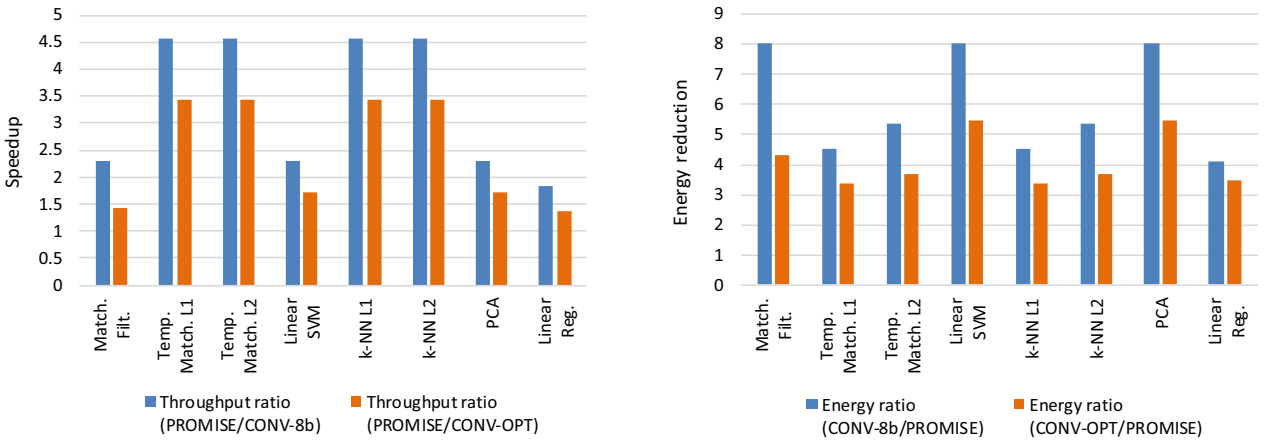


Figure 10: PROMISE (with `SWING` = 111) compared to CONV in terms of: (a) speed-up, and (b) energy savings.

maximum (unoptimized) `swing` voltages, and then compare PROMISE against the baselines described in Section V.

PROMISE executes 128-element vector operation per bank within T_P, i.e., its throughput in terms of "number of OPs per bank per unit time" can be expressed by $f_{PROMISE} = 128/T_P$ per bank. The energy consumption can be divided as:

$$E_{PROMISE} = \sum_{i=1}^{4} E_{\texttt{Class},i} + E_{LEAK} + E_{CTRL}, \quad (6)$$

where $E_{\texttt{Class},i}$ is the energy consumed by `Class`,i instruction, E_{CTRL} and E_{LEAK} are the CTRL block and leakage energies, respectively. Table III shows the energy consumed for each operation at `SWING` = 111.

A. *Effectiveness of Compiler*

Code Generation: The benchmarks for evaluation are listed in Table II. They use a wide variety of combination of operations in different `Classes`. Encoding these diverse algorithms by hand or using a library is not feasible. Coding these algorithms in Julia, and using the compiler to generate PROMISE ISA was both more efficient and error free. Moreover, it is suboptimal to find the `SWING` parameter for benchmarks which use more than one `Tasks`. For example, for DNN benchmark with three hidden layers, the number of `SWING` combinations is $8^4 = 4096$. Inter `Task` compiler analysis is required to find the optimal `SWING` parameter for such algorithms as shown later. Lastly, the compiler also handles different vector sizes for these benchmarks.

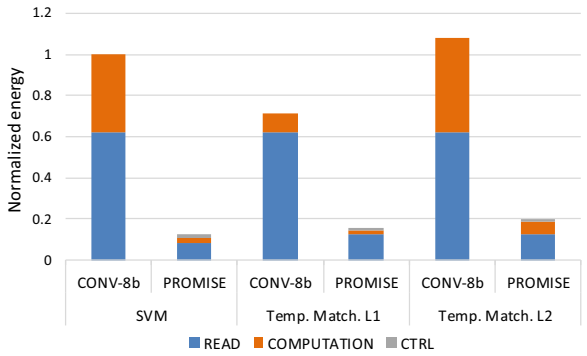


Figure 11: Energy breakdown (normalized to SVM with CONV-8b).

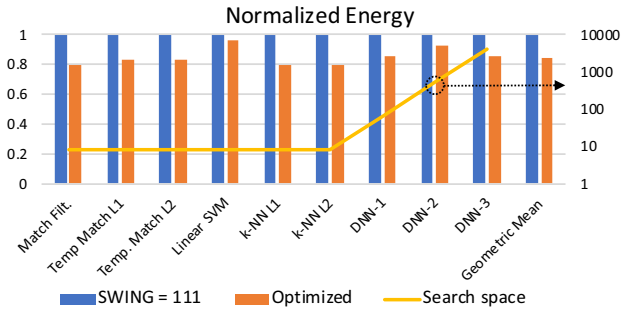


Figure 12: Energy gains by compiler directed energy optimization. DNN-(1, 2, 3) are trained on MNIST dataset. Their structures are: DNN-1(784-128-10), DNN-2(784-256-128-10), and DNN-3(784-512-256-128-10).

Energy Optimization: We evaluate the energy benefits on PROMISE by choosing the optimal `swing` values obtained via compiler directed energy optimization. Fig. 12 shows the energy benefits for energy optimized code generated for PROMISE. The figure contains the energy estimates of PROMISE for the benchmarks under test for two cases: (1) Full Precision - all `Tasks` use maximum `SWING`, and (2) Optimized - `Tasks` use the optimized `SWING` set by the energy optimization pass. We limit the degradation in accuracy to 1% ($p_m = 1\%$). Feature Extraction and Linear Regression are omitted from this evaluation as they are not classification kernels, and mismatch probability is defined for classification algorithms only.

The first six benchmarks (Match Filtering – k-NN L2) in Fig. 12 compile down to a single `AbstractTask` in PROMISE compiler IR, and the optimal `swing` is obtained by doing a sweep over all the eight values of `swing`. The last three benchmarks, DNN-1, DNN-2, and DNN-3 are variants of the multilayer perceptron shown in Table II. The three DNNs have 3, 4, and 5 layers respectively and translate to 2, 3, and 4 `AbstractTasks`. The search space for optimal `swing` increases exponentially with increase of each layer. We use the energy optimization analysis to obtain

Table III: Energy & delay per operation (1 cycle = 1 ns).

Class	Operation	Delay (# of cycles)	Energy/ Bank (pJ)
1	`write`	2	73
	`read`	2	33
	`aREAD`	5	61
	`aSUBT`	7	103
	`aADD`	7	103
2	`compare`	6	5
	`absolute`	6	12
	`square`	8	38
	`sign_mult`	14	16
	`unsign_mult`	14	16
3	`ADC`	138	6
4	`accumulation`	4	≈ 0
	`mean`	3	≈ 0
	`threshold`	2	≈ 0
	`max`	4	≈ 0
	`min`	4	≈ 0
	`sigmoid`	3	≈ 0
	`ReLu`	3	≈ 0
Leakage energy per cycle (1 ns)			0.6
CTRL energy per cycle (1 ns)			5.4

the `swing` values for these DNNs. The optimal `swing` for the `AbstractTasks` in DNN-1 is (3, 6), for DNN-2 is (5, 7, 7), and for DNN-3 is (3, 3, 4, 6). The maximum energy savings come from the lower layers of each DNN, which are wider and also more tolerant to imprecision. Overall the benefits of the optimization range from 4% (Linear SVM) to about 20% (two k-NN versions) with geometric mean of 15%.

B. Performance and Energy

Comparison with State-of-Arts: The k-NN accelerator [7] with L1 and L2-distances is implemented in a 14 nm FinFET process, where 8-bit 128-dimension X is processed with 128 W_js. The k-NN accelerator demonstrates 3.37 (3.84) nJ/decision (processing single input X) with 21.5M (20.3M) decisions/s with L1 (L2) distance. PROMISE achieves 18 (22.9) nJ/decision with 1.12M (0.98M) decisions/s with L1 (L2) distance for the same benchmark with single bank. Though PROMISE achieves lower energy efficiency and throughput, PROMISE is implemented in a 65 nm process (vs. 14 nm FinFET in [7]). If the energy and delay numbers in [7] are scaled to a 65 nm process based on ITRS roadmap [48], PROMISE achieves 4.1× (3.7×) smaller energy and 3.1× (3.4×) lower throughput, achieving 1.3× (1.1×) energy-delay product (EDP) reduction with L1 (L2) distance.

The DNN accelerator [6] was implemented in a 28 nm process for 8-bit 5-layer DNN with a network size of 784-256-256-256-10. The accelerator employs total 1 MB SRAM (to test up to 16-bit case), zero-skipping, and RAZOR technique [49], demonstrating 0.57 μJ/decision and 28K decisions/s. On the other hand, PROMISE enables 8-bit 5-layer DNN with a size of 784-512-256-128-10 in 36 banks (= 576 KB). The network size is not identical, but comparable (PROMISE's network is slightly larger, requiring 69% higher number of

coefficients W_js and MAC operations). PROMISE achieves 0.49 μJ/decision and 558K decisions/s, achieving 1.15× energy saving and 19.9× throughput improvement with 22× EDP reduction though PROMISE was implemented in a 65 nm process (vs. 28 nm in [6]).

Comparison with CONV: Fig. 10(a) shows that PROMISE provides a speed-up of $1.4-3.4\times$ compared to CONV-OPT across the benchmarks. PROMISE's speed-up is the least for linear regression because it needs to re-access the same SRAM data every `Task` because analog data cannot be stored due to leakage whereas CONV stores the data in a local register (pipeline FF in Fig. 9) and reuse it. Fig. 10(b) shows that PROMISE achieves a $3.4-5.5\times$ energy savings compared to CONV-OPT leading to an EDP improvements of $4.7-12.6\times$ compared to CONV-OPT. The key reason for PROMISE's energy efficiency is due to its aREAD (`Class-1`) and aSD/aVD (`Class-2`) executed with low-voltage swing mixed signal computation block (See Fig. 11).

The programmability overhead of PROMISE is minimal, rather our estimates show the concepts introduced in this paper actually can improve over CM. Our results show that PROMISE achieves up to 1.9× speed-up over CM due to the analog pipeline in spite of its operational diversity. In spite of the increased complexity of CTRL to support the programmability, PROMISE was found to achieve 5.5% energy savings over CM due to reduced leakage as PROMISE can go to sleep mode quicker after completing the given `Tasks` due to the throughput gain.

VII. Related Work

ML accelerators: The bandwidth of processor-memory interfaces is a longstanding problem in high-performance computing. Machine learning applications have greatly aggravated the problem, where memory access has begun to dominate the overall energy consumption. There have been proposals for ML accelerators [4], [5], [23] primarily to reduce memory access energy and latency by exploiting the unique data flow of ML algorithms. The Dian-Nao family of deep learning processors was the first to exploit these features and demonstrate approximately 100× improvement in both energy and speed-up compared to a GPU.

Many analog accelerators for ML have also been proposed. RedEye [15] performs the initial convolution layer computations in analog domain before going to digital domain. [50], [51] employed Adaptive Boosting algorithm for image recognition processing raw analog signals from sensors. [52] suggested mixed-signal matrix multiplier via switched-capacitor. Such papers exploit the efficient analog calculation for large data processing at slightly compromised accuracy. However, they do not cover diverse ML algorithms in general.

There have also been efforts to integrate computation near memory with emerging technologies such as resistive RAM (RRAM) [53], and a Memristor crossbar based CNN accelerator [14]. Compute cache [12] also demonstrates the potential benefit of near-memory computing in digital domain for memory-intensive applications. PROMISE proposes an alternative line of integration via the use of compute memory (CM), where the memory and processor integrated in mixed-signal domain by eliminating the standard interface. Prior to PROMISE, CM was used for fixed function architectures [8], [9], [25], [54] to achieve aggressive energy savings.

Accuracy-Energy Tradeoff: There has been a lot of work [55]–[58] in the Approximate Computing field to use high level accuracy specifications by programmer to generate code for unreliable hardware. However, these are built for digital hardware and do not map well to PROMISE which is a novel mixed-signal accelerator targeting applications in a very specific domain (ML).

VIII. Conclusion

This paper presented PROMISE, the first end-to-end design of a programmable mixed-signal accelerator for diverse ML algorithms. PROMISE accomplishes a high level of programmability without losing the efficiency of mixed-signal accelerators for specific ML algorithms. We designed the PROMISE ISA to allow software control over energy-vs-accuracy tradeoffs, and supports compilation of high-level languages like Julia down to the hardware. Our evaluation shows better energy efficiency than digital ASICs, despite much greater programmability, and significant energy savings from small programmer-specified error tolerances.

Acknowledgements

This work was supported by C-FAR and SONIC, two of the six SRC STARnet Centers, sponsored by MARCO and DARPA. It was also supported by NSF (CCF-1302641 and CNS-1705047).

References

[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, 2016.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012.

[3] Y. Chen, T. Chen, Z. Xu, N. Sun, and O. Temam, "DianNao Family: Energy-efficient accelerators for machine-learning," *Communications of the ACM*, 2016.

[4] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *ISCA*, 2016.

[5] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An Instruction Set Architecture for Neural Networks," in *ISCA*, 2016.

[6] P. N. Whatmough, S. K. Lee, H. Lee, S. Rama, D. Brooks, and G.-Y. Wei, "A 28nm SoC with a 1.2 GHz 568nJ/prediction sparse deep-neural-network engine with > 0.1 timing error rate tolerance for IoT applications," in *ISSCC*, 2017.

[7] H. Kaul, M. A. Anders, S. K. Mathew, G. Chen, S. K. Satpathy, S. K. Hsu, A. Agarwal, and R. K. Krishnamurthy, "14.4 A 21.5 M-query-vectors/s 3.37 nJ/vector reconfigurable k-nearest-neighbor accelerator with adaptive precision in 14nm tri-gate CMOS," in *ISSCC*, 2016.

[8] M. Kang, M.-S. Keel, N. R. Shanbhag, S. Eilert, and K. Curewitz, "An energy-efficient VLSI architecture for pattern recognition via deep embedding of computation in SRAM," in *ICASSP*, 2014.

[9] M. Kang, S. K. Gonugondla, A. Patil, and N. R. Shanbhag, "A Multi-Functional In-Memory Inference Processor Using a Standard 6T SRAM Array," *JSSC*, 2018.

[10] M. Kang, S. K. Gonugondla, S. Lim, and N. R. Shanbhag, "A 19.4 nJ/decision, 364K decisions/s, In-memory Random Forest Multi-class Inference Accelerator," *JSSC*, 2018.

[11] J. Zhang, Z. Wang, and N. Verma, "In-Memory Computation of a Machine-Learning Classifier in a Standard 6T SRAM Array," *JSSC*, 2017.

[12] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *HPCA*, 2017.

[13] Y. Huang, N. Guo, M. Seok, Y. Tsividis, K. Mandli, and S. Sethumadhavan, "Hybrid analog-digital solution of nonlinear partial differential equations," in *MICRO*, 2017.

[14] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," in *ISCA*, 2016.

[15] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, "Redeye: analog convnet image sensor architecture for continuous mobile vision," in *ISCA*, 2016.

[16] F. N. Buhler, P. Brown, J. Li, T. Chen, Z. Zhang, and M. P. Flynn, "A 3.43 TOPS/W 48.9 pJ/pixel 50.1 nJ/classification 512 analog neuron sparse coding neural network with on-chip learning and classification in 40nm CMOS," in *Symposium on VLSI Circuits*, 2017.

[17] R. Genov and G. Cauwenberghs, "Kerneltron: support vector "machine" in silicon," *IEEE Transactions on Neural Networks*, 2003.

[18] Python Core Team, *Python: A dynamic, open source programming language*, Python Software Foundation, 2015. [Online]. Available: https://www.python.org

[19] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, 2017.

[20] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *OSDI*, 2016.

[21] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," *CoRR*, vol. abs/1512.01274, 2015.

[22] C. Sakr, Y. Kim, and N. Shanbhag, "Analytical guarantees on numerical precision of deep neural networks," in *ICML*, 2017.

[23] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, Z. Xuehai, and Y. Chen, "PuDianNao : A Polyvalent Machine Learning Accelerator," *ASPLOS*, 2015.

[24] N. Shanbhag, M. Kang, and M.-S. Keel, *Compute Memory*. US Patent 9,697,877 B2, July 04, 2017.

[25] M. Kang, S. K. Gonugondla, M.-S. Keel, and N. R. Shanbhag, "An energy-efficient memory-based high-throughput VLSI architecture for Convolutional Networks," in *ICASSP*, 2015.

[26] S. K. Gonugondla, M. Kang, and N. Shanbhag, "A 42 pJ/decision 3.12TOPS/W robust in-memory machine learning classifier with on-chip training," in *ISSCC*, 2018.

[27] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "CNP: An FPGA-based processor for convolutional networks," in *FPL*, 2009.

[28] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[29] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[30] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermüller, D. Bahdanau, N. Ballas *et al.*, "Theano: A python framework for fast computation of mathematical expressions," *CoRR*, vol. abs/1605.02688, 2016.

[31] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[32] The Apache Software Foundation (ASF), "MXNet.jl," http://dmlc.ml/MXNet.jl/latest, 2015.

[33] M. J. Innes, "Flux: The Julia Machine Learning Library," https://github.com/FluxML/Flux.jl, 2015.

[34] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *CGO*, 2004.

[35] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *TOPLAS*, 1991.

[36] Y. LeCun, C. Cortes, and C. J. Burges, "The MNIST database of handwritten digits," http://yann.lecun.com/exdb/mnist, 1998.

[37] C. Sakr and N. Shanbhag, "An Analytical Method to Determine Minimum Per-layer Precision of Deep Neural Networks," *ICASSP*, 2018.

[38] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *CoRR, abs/1502.02551*, 2015.

[39] B. Murmann, D. Bankman, E. Chai, D. Miyashita, and L. Yang, "Mixed-signal circuits for embedded machine-learning applications," in *Asilomar Conference on Signals, Systems and Computers*, 015.

[40] Production Crate, "Gun Shot Sounds." [Online]. Available: http://soundscrate.com/gun-related/

[41] Center for Biological and Computational Learning, MIT, "CBCL Face Database No. 1," 2001. [Online]. Available: http://cbcl.mit.edu/software-datasets/index.html

[42] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, 2015.

[43] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, 1995.

[44] K. I. Kim, K. Jung, and H. J. Kim, "Face recognition using kernel principal component analysis," *IEEE signal processing letters*, 2002.

[45] R. Brunelli and T. Poggio, "Face recognition: Features versus templates," *IEEE transactions on pattern analysis and machine intelligence*, 1993.

[46] D. K. Mellinger, S. W. Martin, R. P. Morrissey, L. Thomas, and J. J. Yosco, "A method for detecting whistles, moans, and other frequency contour sounds," *The Journal of the Acoustical Society of America*, 2011.

[47] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *ACM SIGARCH Computer Architecture News*, 2010.

[48] ITRS, "ITRS Roadmap." [Online]. Available: http://www.itrs2.net/

[49] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner *et al.*, "RAZOR: A low-power pipeline based on circuit-level timing speculation," in *MICRO*, 2003.

[50] W. Rieutort-Louis, T. Moy, Z. Wang, S. Wagner, J. C. Sturm, and N. Verma, "16.2 A large-area image sensing and detection system based on embedded thin-film classifiers," in *ISSCC*, 2015.

[51] S. Joshi, C. Kim, S. Ha, Y. M. Chi, and G. Cauwenberghs, "21.7 2pJ/MAC 14b 8×8 linear transform mixed-signal spatial filter in 65nm CMOS with 84dB interference suppression," in *ISSCC*, 2017.

[52] E. H. Lee and S. S. Wong, "24.2 A 2.5 GHz 7.7 TOPS/W switched-capacitor matrix multiplier with co-designed local memory in 40nm," in *ISSCC*, 2016.

[53] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory," in *ISCA*, 2016.

[54] M. Kang and N. R. Shanbhag, "In-memory computing architectures for sparse distributed memory," *IEEE Transactions on Biomedical Circuits and Systems*, 2016.

[55] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels," in *OOPSLA*, 2014.

[56] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "ACCEPT: A programmer-guided compiler framework for practical approximate computing," in *U. Washington, Tech. Rep. UW-CSE- 15-01-01*, 2015.

[57] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *PLDI*, 2011.

[58] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "Uncertain<T>: A First-order Type for Uncertain Data," in *ASPLOS*, 2014.