PUMA: A Programmable Ultra-efficient Memristor-based Accelerator for Machine Learning Inference

Aayush Ankit
Purdue University,
Hewlett Packard Enterprise

Izzat El Hajj*
American University of Beirut

Sai Rahul Chalamalasetti
Hewlett Packard Enterprise

Geoffrey Ndu
Hewlett Packard Enterprise

Martin Foltin
Hewlett Packard Enterprise

R. Stanley Williams
Hewlett Packard Enterprise

Paolo Faraboschi
Hewlett Packard Enterprise

Wen-mei Hwu
University of Illinois at Urbana-Champaign

John Paul Strachan
Hewlett Packard Enterprise

Kaushik Roy
Purdue University

Dejan S Milojicic
Hewlett Packard Enterprise

Abstract

Memristor crossbars are circuits capable of performing analog matrix-vector multiplications, overcoming the fundamental energy efficiency limitations of digital logic. They have been shown to be effective in special-purpose accelerators for a limited set of neural network applications.

We present the Programmable Ultra-efficient Memristor-based Accelerator (PUMA) which enhances memristor crossbars with general purpose execution units to enable the acceleration of a wide variety of Machine Learning (ML) inference workloads. PUMA's microarchitecture techniques exposed through a specialized Instruction Set Architecture (ISA) retain the efficiency of in-memory computing and analog circuitry, without compromising programmability.

We also present the PUMA compiler which translates high-level code to PUMA ISA. The compiler partitions the computational graph and optimizes instruction scheduling and register allocation to generate code for large and complex workloads to run on thousands of spatial cores.

We have developed a detailed architecture simulator that incorporates the functionality, timing, and power models of PUMA's components to evaluate performance and energy consumption. A PUMA accelerator running at 1 GHz can reach area and power efficiency of 577 GOPS/s/mm^2 and 837 GOPS/s/W, respectively. Our evaluation of diverse ML applications from image recognition, machine translation, and language modelling (5M-800M synapses) shows that PUMA achieves up to 2,446× energy and 66× latency improvement for inference compared to state-of-the-art GPUs. Compared to an application-specific memristor-based accelerator, PUMA incurs small energy overheads at similar inference latency and added programmability.

Keywords memristors, accelerators, machine learning, neural networks

ACM Reference Format:
Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R. Stanley Williams, Paolo Faraboschi, Wen-mei Hwu, John Paul Strachan, Kaushik Roy, and Dejan S Milojicic. 2019. PUMA: A Programmable Ultra-efficient Memristor-based Accelerator for Machine Learning Inference. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3297858.3304049

1 Introduction

General-purpose computing systems have benefited from scaling for several decades, but are now hitting an energy wall. This trend has led to a growing interest in domain-specific architectures. Machine Learning (ML) workloads in particular have received tremendous attention because of their pervasiveness in many application domains and high performance demands. Several architectures have been proposed, both digital [20, 21, 36, 61, 72, 89] and mixed digital-analog using memristor crossbars [22, 23, 73, 95, 100].

*Work done while at University of Illinois at Urbana-Champaign

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304049

ML workloads tend to be data-intensive and perform a large number of Matrix Vector Multiplication (MVM) operations. Their execution on digital CMOS hardware is typically characterized by high data movement costs relative to compute [49]. To overcome this limitation, memristor crossbars can store a matrix with high storage density and perform MVM operations with very low energy and latency [5, 13, 52, 87, 98, 116]. Each crosspoint in the crossbar stores a multi-bit value in one memristor device, which enables high storage density [112]. Upon applying an input voltage at the crossbar's rows, we get the MVM result as output current at the crossbar's columns based on Kirchhoff's law. A crossbar thus performs MVM in one computational step – including $O(n^2)$ multiplications and additions for an $n \times n$ matrix – which typically takes many steps in digital logic. It also combines compute and storage in a single device to alleviate data movement, thereby providing intrinsic suitability for data-intensive workloads [23, 95].

Memristor crossbars have been used to build special-purpose accelerators for Convolutional Neural Networks (CNN) and Multi Layer Perceptrons (MLP) [23, 73, 95], but these designs lack several important features for supporting general ML workloads. First, each design supports one or two types of neural networks, where layers are encoded as state machines. This approach is not scalable to a larger variety of workloads due to increased decoding overhead and complexity. Second, existing accelerators lack the types of operations needed by general ML workloads. For example, Long Short-Term Memory (LSTM) [51] workloads require multiple vector linear and transcendental functions which cannot be executed on crossbars efficiently and are not supported by existing designs. Third, existing designs do not provide flexible data movement and control operations to capture the variety of access and reuse patterns in different workloads. Since crossbars have high write latency [54], they typically store constant data while variable inputs are routed between them in a spatial architecture. This data movement can amount to a significant portion of the total energy consumption which calls for flexible operations to optimize the data movement.

To address these limitations, we present PUMA, a Programmable Ultra-efficient Memristor-based Accelerator. PUMA is a spatial architecture designed to preserve the storage density of memristor crossbars to enable mapping ML applications using on-chip memory only. It supplements crossbars with an instruction execution pipeline and a specialized ISA that enables compact representation of ML workloads with low decoder complexity. It employs temporal SIMD units and a ROM-Embedded RAM [69] for area-efficient linear and transcendental vector computations. It includes a microarchitecture, ISA, and compiler co-designed to optimize data movement and maximize area and energy efficiency. To the best of our knowledge, PUMA is the first programmable and general-purpose ML inference accelerator built with hybrid CMOS-memristor technology.

A naïve approach to generality is not viable because of the huge disparity in compute and storage density between the two technologies. CMOS digital logic has an order of magnitude higher area requirement than a crossbar for equal output width (~20×, see Table 3). Moreover, a crossbar's storage density (2-bit cells) is $160MB/mm^2$, which is at least an order of magnitude higher than SRAM (6T, 1-bit cell) [95]. A 90mm^2 PUMA node can store ML models with up to 69MB of weight data. Note that the PUMA microarchitecture, ISA, and compiler are equally suitable to crossbars made from emerging technologies other than memristors such as STT-MRAM [94], NOR Flash [46], etc.

We make the following contributions:

- A programmable and highly efficient architecture exposed by a specialized ISA for scalable acceleration of a wide variety of ML applications using memristor crossbars.
- A complete compiler which translates high-level code to PUMA ISA, enabling the execution of complex workloads on thousands of spatial cores.
- A detailed simulator which incorporates functionality, timing, and power models of the architecture.
- An evaluation across ML workloads showing that PUMA can achieve promising performance and energy efficiency compared to state-of-the-art CPUs, GPUs, TPU, and application-specific memristor-based accelerators.

Our simulator and compiler have been open-sourced to enable further research on this class of accelerators.

2 Workload Characterization

This section characterizes different ML inference workloads with a batch size of one. The characteristics are summarized in Table 1. The section's objective is to provide insights on the suitability of memristor crossbars for accelerating ML workloads and highlight implications on the proposed architecture.

2.1 Multi-Layer Perceptron (MLP)

MLPs are neural networks used in common classification tasks such as digit-recognition, web-search, etc. [26, 61]. Each layer is fully-connected and applies a nonlinear function to the weighted-sum of outputs from the previous layer. The weighted-sum is essentially an MVM operation. Equation 1 shows the computations in a typical MLP (act is nonlinear):

$$O[y] = act(B[y] + \sum_{x=0}^{n-1} I[x] \times W[x][y]) \quad (1)$$

MLPs are simple, capturing the features common across the ML workloads we discuss: dominance of MVM operations, high data parallelism, and use of nonlinear operations.

2.1.1 Dominance of MVM

MVM operations are $O(n^2)$ in space and computational complexity, whereas the nonlinear operations are $O(n)$, where n

Table 1. Workload Characterization

Characteristic	MLP	LSTM	CNN
Dominance of MVM	Yes	Yes	Yes
High data parallelism	Yes	Yes	Yes
Nonlinear operations	Yes	Yes	Yes
Linear operations	No	Yes	No
Trancendental operations	No	Yes	Yes
Weight data reuse	No	Yes	Yes
Input data reuse	No	No	Yes
Bounded resource	Memory	Memory	Compute
Sequential access pattern	Yes	Yes	No

is the matrix dimension (layer size). MVMs are therefore the dominant operation in MLPs (and other ML workloads). This property makes memristor crossbars suitable for acceleration since they perform analog MVMs with low energy/latency.

2.1.2 High data parallelism

MLPs (and other ML workloads) have massive amounts of data parallelism. Moreover, practical model sizes are larger than the typical on-chip storage that can be provided by SRAM. For this reason, CMOS implementations suffer from costly DRAM accesses which are particularly taxing due to the absence of data reuse to amortize them. On the other hand, crossbars have extremely high area efficiency which allows deploying many of them on a single chip. Doing so not only captures the high data parallelism in these workloads, but it also provides high storage density to fit models on-chip and bypass costly DRAM accesses.

2.1.3 Nonlinear operations

In addition to MVM operations, MLPs (and other ML workloads) perform nonlinear vector operations (e.g., ReLU). Since these operations cannot be performed in crossbars, an implication on the architecture is the need to provide digital functional units to support them. Such functional units consume significantly more area (~20×) than crossbars for equal output width (see Table 3). The challenge is to size these units appropriately to provide sufficient throughput without offsetting crossbar area/energy efficiency.

2.2 Long Short-Term Memory (LSTM)

LSTMs are the state-of-the-art technique for sequence processing tasks like speech processing, language modelling, etc. [51]. Each layer is fully connected and performs linear and nonlinear operations on the weighted-sum of outputs and the previous state. These operations translate into two MVMs followed by multiple (typically four) vector arithmetic operations and (typically four) nonlinear functions. Equations 2 to 4 show the computations in a typical LSTM:

$$F_t[y] = act(B[f] + \sum_{x=0}^{n-1}(H, I)[x] \times W_f[x][y]) \quad (2)$$

$$C_t[y] = \sum_{x=0}^{n-1}(f_t[y] \times C_{t\text{-}1}[y] + g_t[y] \times Cp_{t\text{-}1}[y]) \quad (3)$$

$$H_t[y] = \sum_{x=0}^{n-1}(h_t[y] \times C_t[y]) \quad (4)$$

To the best of our knowledge, PUMA is the first memristor-based accelerator demonstrated with LSTMs.

2.2.1 Linear and transcendental operations

Unlike MLPs, LSTMs also perform linear vector operations. Moreover, the typical nonlinear vector operations in LSTMs are transcendental (e.g. tanh, sigmoid). Supporting these operations has the same implication on the architecture as discussed in Section 2.1.3 for nonlinear operations. Transcendental operations are particularly challenging due to their high complexity.

2.2.2 Weight reuse

Another key distinction of LSTMs compared to MLPs is data reuse. LSTM inputs consist of a sequence of vectors processed across multiple time-steps with the same weights. This feature benefits CMOS architectures by amortizing DRAM accesses for loading weights, but is not advantageous to memristor crossbars. That said, the scope of weight reuse in LSTMs is only over a few inputs so the workload remains memory-bound. It still suffers in CMOS hardware from insufficient amortization of DRAM accesses.

2.3 Convolutional Neural Network (CNN)

CNNs are widely used for image recognition and classification [67]. They typically include several convolutional, pooling, and response normalization layers. A convolution layer consists of weight kernels strided across the input image in a sliding window fashion. It exhibits a non-sequential memory access pattern since a window of the input consists of parts of the input image from different rows. Equation 5 shows the computations in a typical convolutional layer of a CNN:

$$O[m][x][y] = act(B[m] + \sum_{i=0}^{R-1}\sum_{j=0}^{S-1}\sum_{k=0}^{C-1} I[k][Ux+i][Uy+j] \times W[m][k][i][j]) \quad (5)$$

2.3.1 Input reuse and compute intensity

Convolution layers exhibit both weight and input data reuse. They can be mapped to matrix-matrix multiplications which successively apply weight kernels on different input windows. Matrix-matrix multiplications are compute-bound which makes them well-suited for CMOS hardware since there is enough data reuse to amortize DRAM access cost. However, memristor crossbars can still perform well on matrix-matrix operations by treating them as successive MVMs. An implication on architecture is the opportunity to take advantage of input reuse to minimize data movement within the chip. Another implication is that iterating over inputs creates the need for control flow to represent the workload compactly without code bloat.

2.3.2 Non-sequential access

Unlike MLPs and LSTMs, CNNs exhibit non-sequential accesses due to the way inputs are traversed as well as the behavior of non-convolutional layers such as pooling and

response-normalization. An implication on the architecture is the need to support fine-grain/random access to memory, which is not needed for MLPs and LSTMs where it is sufficient to access data at the granularity of the input/output vectors to each layer.

2.4 Other ML Workloads

Other workloads, both supervised and unsupervised, can be represented using a combination of the patterns in the three applications in this section. *Logistic Regression* [2] and *Linear Regression* [81] compute weighted-sums which are passed to activation functions to generate probabilities and continuous values respectively. *Support Vector Machine (SVM)* [41] and *Recommender Systems* [91] compute weighted-sums followed by nonlinear functions. Their computations are similar to MLP. *Recurrent Neural Networks (RNNs)* [75] used for sequence processing compute weighted-sums on input and previous state. They are similar to LSTMs but without vector operations. *Generative Adversarial Networks (GANs)* are composed of two neural networks (MLP, LSTM, CNN, etc.) which compete to reach equilibrium [44]. *Restricted Boltzmann Machines (RBM)* [102] and *Boltzmann Machines (BM)* [104] are commonly used in unsupervised learning tasks for energy-minimization. While RBM involves weighted-sums of previous state and inputs, BM uses inputs only. Their computations have similarities to MLPs and LSTMs as well.

3 Core Architecture

We propose a programmable architecture and ISA design that leverage memristor crossbars for accelerating ML workloads. PUMA is a spatial architecture organized in three-tiers: cores, tiles, and nodes. Cores consist of analog crossbars, functional units, and an instruction execution pipeline. Tiles consist of multiple cores connected via a shared memory. Nodes consist of multiple tiles connected via an on-chip network. Subsequently, nodes can be connected together via a chip-to-chip interconnect for large-scale execution.

While this hierarchical organization is common in related work [20, 95], our key contributions lie in the core architecture (this section) and tile architecture (Section 4) that bring programmability and generality to memristor crossbars without compromising their energy and area efficiency. An overview of the core architecture is shown in Figure 1. The following subsections discuss the components of the core architecture and the insights behind their design.

3.1 Instruction Execution Pipeline

Existing memristor-based accelerators [23, 73, 95] are limited to one or two ML workloads. They use state machines that can be configured to compose a small set of functional blocks (e.g., convolution block, pooling block, etc.). While this approach works well when the scope of workloads is small, supporting a larger variety of workloads creates high

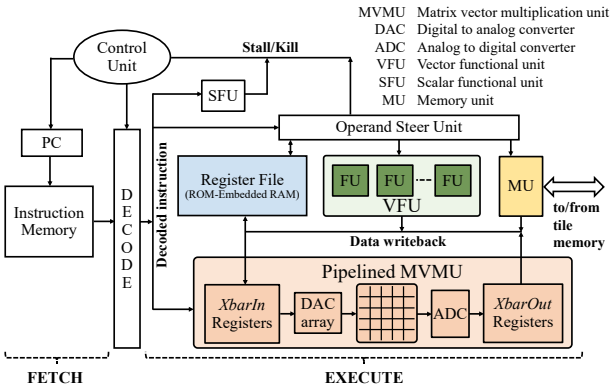

Figure 1. Core Architecture

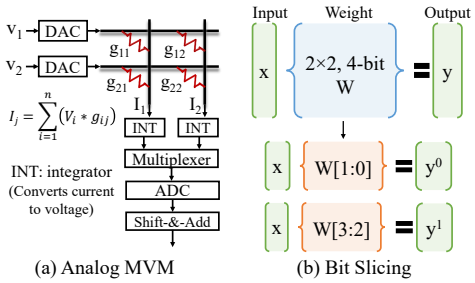

Figure 2. MVM with Crossbars

decoding complexity. For this reason, our core architecture breaks functionality down to finer-grain instructions and supplements memristor crossbars with an instruction execution pipeline. Our approach is based on the observation in Section 2 that despite the large variety of ML workloads, these workloads share many low-level operations.

The instruction execution pipeline is an in-order pipeline with three stages: fetch, decode, and execute. Keeping the pipeline simple saves area to avoid offsetting the crossbars' area efficiency. The ISA executed by the pipeline is summarized in Table 2. Instructions are seven bytes wide. The motivations for wide instructions are discussed in Sections 3.3 and 3.4.3. The ISA instruction usage is shown in Section 3.6. More ISA details are discussed in another paper [7].

The instruction execution pipeline supports control flow instructions (*jmp* and *brn* in Table 2), as motivated in Section 2.3.1. It also includes a *Scalar Functional Unit (SFU)* that performs scalar integer operations (*ALUint* in Table 2) to support the control flow instructions.

3.2 Matrix-Vector Multiplication Unit (MVMU)

The MVMU (illustrated in Figure 1) consists of memristor crossbars that perform analog MVM operations, and peripherals (DAC/ADC arrays) that interface with digital logic via the *XbarIn* and *XbarOut* registers. *XbarIn* registers provide digital inputs to the DACs which feed analog voltages to the crossbar. ADCs convert crossbar output currents to digital values which are stored in the *XbarOut* registers. This crossbar design is similar to ISAAC [95].

Table 2. Instruction Set Architecture Overview

Category	Instruction	Description	Operands
Compute	MVM	Matrix-Vector Multiplication	mvm, mask, -, filter, stride, -, -
	ALU	Vector arithmetic/logical (add, subtract, multiply, divide, shift, and, or, invert) Vector non-linear (relu, sigmoid, tanh, log, exponential) Other (random vector, subsampling, min/max)	alu, aluop, dest, src1, src2, src3, vec-width
	ALUimm	Vector arithmetic immediate (add, subtract, multiply, divide)	alui, aluop, dest, src1, immediate, vec-width
	ALUint	Scalar arithmetic (add, subtract) - Compare (equal, greater than, not equal)	alu-int, aluop, dest, src1, src2, -, -
Intra-Core Data Movement	set	Register initialization	set, -, dest, immediate, -, -
	copy	Data movement between different registers	copy, -, dest, src1, -, ld-width, vec-width
Intra-Tile Data Movement	load	Load data from shared memory	load, -, dest, immediate, -, -
	store	Store data to shared memory	store, -, dest, src1, count, st-width, vec-width
Intra-Node Data Movement	send	Send data to tile	send, memaddr, fifo-id, target, send-width, vec-width
	receive	Receive data from tile	receive, memaddr, fifo-id, count, rec-width, vec-width
Control	jmp	Unconditional jump	jmp, -, -, -, -, pc
	brn	Conditional jump	brn, brnop, -, src1, src2, pc

Figure 2(a) shows how memristor crossbars can be used to perform analog MVM operations. DACs convert the input vector to analog voltages applied at crossbar rows. The matrix is encoded as conductance states (g_{ij}) of the devices that constitute the crossbar. The currents at crossbar columns constitute the MVM result. They are integrated (converted to voltage) then converted to digital values with ADCs.

3.2.1 Precision Considerations

Practically realizable crossbars provide 2-6 bits of precision per device [52]. We conservatively use 2 bits per device, and realize 16-bit MVM operations by combining 8 crossbars via bit-slicing [95], illustrated in Figure 2(b). ADCs are reused across columns to save area. The impact of precision on inference accuracy is evaluated in Section 7.6.

3.2.2 Crossbar Co-location and Input Sharing

Crossbar peripherals have an order of magnitude higher area than the crossbar. Since all eight 2-bit crossbars of a 16-bit MVM operation are used simultaneously on the same input, we co-locate these 2-bit crossbars on the same core in the same MVMU, which allows us to use the same *XbarIn* registers and DAC array to feed them with minimal routing congestion. This co-location and input reuse is provided transparently in the ISA, which exposes a full 16-bit MVM operation in a single instruction (*MVM* in Table 2).

3.2.3 Input Shuffling

As motivated in Section 2.3.1, ML workloads with sliding window computations typically reuse large portions of the input across successive MVM operations (~80% for convolutional layers with filter size 5 and unit stride). However, reused input values may come at different positions in the input vector. To avoid moving data around in *XbarIn*, the MVM instruction provides operands (*filter*/*stride* in Table 2) that re-route XbarIn registers to different DACs, enabling logical input shuffling without physical data movement.

3.2.4 Multiple MVMUs per Core

A core may have multiple MVMUs, in which case it is desirable to activate them in parallel since MVMs are heavy operations. The in-order pipeline does not capture the Instruction-Level Parallelism (ILP) between MVM instructions automatically. Instead, the ISA exposes an operand (*mask* in Table 2) to allow a single MVM instruction to activate multiple MVMUs at once. Compiler optimizations that use this operand are discussed in Section 5.3.2.

3.2.5 Crossbar Writes

PUMA is an inference accelerator, so crossbars are initialized with weights using serial writes at configuration time prior to execution and are not written to throughout execution. In this sense, PUMA is a spatial architecture where data is routed between crossbars, each crossbar storing a different portion of the model. Larger models therefore require more area, and may scale to multiple nodes.

3.3 Vector Functional Unit (VFU)

The VFU executes linear and nonlinear vector operations (*ALU* and *ALUimm* in Table 2), as motivated by Sections 2.1.3 and 2.2.1. An important aspect of designing vector instructions and the VFU is choosing the vector width. Since ML workloads have high data parallelism, they execute wide vector operations, which motivates having wide vector instructions. Wide vector instructions have the benefit of reducing instruction count, and consequently, fetch, decode, and instruction storage overhead. On the other hand, hardware considerations motivate having narrow VFU vector width to avoid offsetting the area efficiency of crossbars as discussed in Section 2.1.3.

To balance the tension between workloads favoring wide vector width and hardware favoring narrow vector width, we propose a VFU design based on *temporal SIMD*. Temporal SIMD uses a narrow width VFU to execute wide vectors over multiple cycles. The vector instruction operands specify the starting address of the vectors in the register file as well as the vector width (*vec-width* in Table 2). The *operand steer unit* holds the decoded instruction and reads the operands from the register file over subsequent cycles to feed the VFU. The additional *vec-width* operand required by temoral SIMD motivates our wide instruction design.

Provisioning the adequate width for VFUs maintains crossbar area efficiency benefits without the VFU becoming a bottleneck and compromising throughput. A narrow VFU is possible because typical ML workloads compute $O(n)$ more operations per MVM instruction than per vector instruction. Section 7.6 evaluates the impact of VFU width on efficiency.

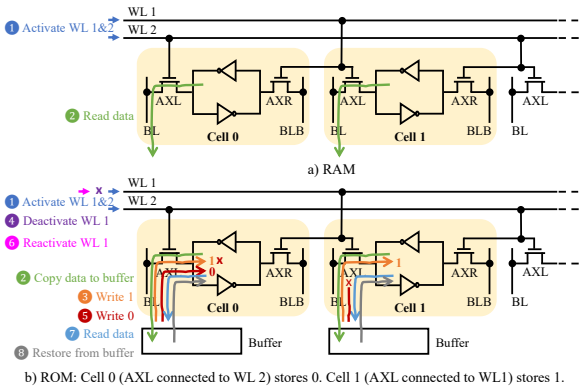

Figure 3. ROM-Embedded RAM

3.4 Register File

We propose a register file design that uses *ROM-Embedded RAM* [69] to accomplish two objectives: (1) harboring general purpose registers, and (2) providing area-efficient transcendental function evaluations as motivated in Section 2.2.1.

3.4.1 Implementing transcendental functions

Area-efficient function evaluations are crucial for preserving crossbar storage density. For this reason, we use a ROM-Embedded RAM structure [69] which adds a wordline per row to embed a ROM that is used to store look-up tables without increasing the array area or RAM latency. Alternative digital implementations to support transcendental functions are prohibitive due to their high area requirements, especially in light of the large number of transcendental function types used in ML. Transcendental function evaluations also use temporal SIMD (Section 3.3) to minimize fetch/decode energy consumption.

Figure 3 details the operation of a ROM-Embedded RAM. In RAM mode, both wordlines (*WL1* and *WL2*) are activated, followed by precharging or driving the bitlines for read or write operations, respectively (similar to typical SRAM). In ROM mode, since a ROM access overwrites the RAM data, the first step is to buffer the RAM data. Subsequently, 1 is written to all cells with both wordlines activated. Next, 0 is written to all cells while keeping the WL1 deactivated, which writes 0 to a cell only if its AXL is connected to WL2. Therefore, cells with AXL connected to WL1 and WL2, will store a ROM value of 1 and 0, respectively. A read with both wordlines activated is done to retrieve the ROM data, followed by restoring the original RAM contents.

3.4.2 Sizing the register file

The register file enables buffering data in general purpose registers to avoid higher-cost access to shared memory. However, if the register file were too large, it would degrade core storage density. A key point to observe is that the majority of ML kernels are such that data is consumed within 1-2 instructions after being produced. This property is preserved via proper instruction scheduling by the compiler to reduce register pressure (Section 5.3.1). Therefore, we provision a per-core register file size of *2*(crossbar dimension)*(# crossbars per core)*. This size retains storage density while addressing the buffering requirements in the common case as shown in Section 7.6. For window-based computations such as pooling layers that have a large number of intervening instructions (due to non-sequential data access across rows), the compiler spills registers to tile memory (Section 5.4).

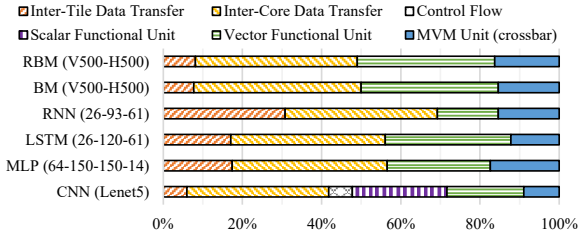

Figure 4. Static instruction usage showing the importance of different execution units.

3.4.3 ISA implications

To accommodate the large register file required to match crossbar size, long operands are used in ISA (*src* and *dest* in Table 2), which is another motivation for the wide instruction design. To accommodate moving data between general purpose registers and *XbarIn*/*XbarOut* registers, a *copy* instruction is included.

3.5 Memory Unit (MU)

The MU interfaces the core with the tile memory via *load* and *store* instructions. These instructions can be executed at 16-bit word granularity to support random access as motivated in Section 2.3.2. However, the instructions also take a *vec-width* operand for wide vector loads caused by sequential access patterns. Vector loads also use temporal SIMD (Section 3.3) to minimize fetch/decode energy consumption.

3.6 Static Instruction Usage

Figure 4 shows the breakdown of the static instruction count for six different ML workloads. The breakdown demonstrates that MVM alone is insufficient to support all types of workloads, and that the ISA and functional units proposed can be used to bridge that gap. The ratio of instructions requiring MVMU versus VFU varies depending on the number of matrix versus vector transformation layers in the network. CNNs additionally use control flow instructions as discussed in Section 2.3.1. Deeper (or wider) versions of the same networks tend to have a similar instruction breakdown, except for data movement instructions which tend to be higher to implement larger matrices spanning multiple cores and tiles.

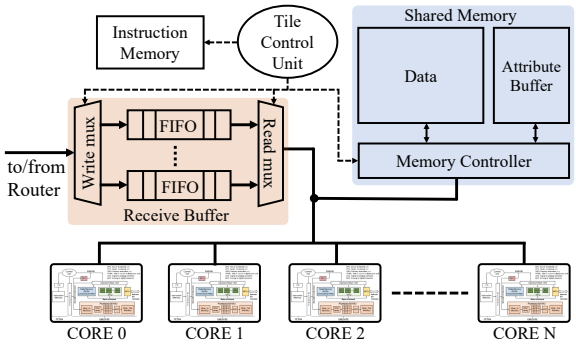

Figure 5. Tile Architecture

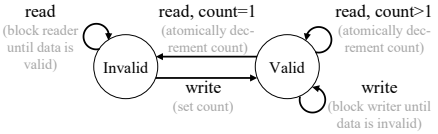

Figure 6. Inter-core synchronization mechanism

3.7 Summary

In summary, the core architecture provides programmability while maintaining crossbar area efficiency. It features an instruction pipeline exposed by an ISA to support a wide variety of ML workloads. The use of temporal SIMD and ROM-Embedded RAM enable linear, nonlinear, and transcendental vector operations. Data movement optimizations are enabled via input shuffling, proper sizing of the register file, and flexible memory access instructions.

4 Tile Architecture

Figure 5 illustrates the architecture of a tile. A tile is comprised of multiple cores connected to a shared memory. The tile instruction memory holds *send* and *receive* instructions that move data between tiles. The shared memory and receive buffer are described in the following subsections.

4.1 Shared Memory

The shared memory facilitates communication across cores and tiles. Our shared memory design follows two key principles: (1) enabling inter-core synchronization, and (2) sizing the shared memory to preserve storage density.

4.1.1 Inter-core synchronization

Synchronization between cores happens when the output of one layer is sent as input to the next layer. It also happens within a layer if a large weight matrix is partitioned across multiple cores and tiles and partial MVM results need to be aggregated together. To enable synchronization, we augment the shared memory with an attribute buffer that has two attributes per data entry: valid and count. The use of valid and count is illustrated in Figure 6. This mechanism enables consumer cores to block until producer cores have written their values, and ensures that producer cores do not overwrite data until it is consumed.

4.1.2 Sizing the shared memory

ML workloads may require buffering large number of inputs to utilize their weight reuse pattern. However, a large shared memory degrades the crossbar storage density. PUMA's spatial architecture enables programming *inter-core/tile pipelines* that exploit the inter-layer parallelism in ML workloads with weight reuse. These pipelines can be used to maintain throughput while keeping the shared memory size small. The pipeline parallelism is based on the observation that we do not require all the outputs of previous layer to start the current layer computation. For example, LSTMs process a sequence of vectors with $S * N$ inputs per layer, where S is the number of vectors per input sequence and N is vector size. A layer can begin its computation as soon as its first N inputs are available. Section 7.5 discusses the sizing requirements for different workloads and the impact on energy consumption.

4.2 Receive Buffer

The receive buffer is an $N \times M$ structure with N FIFOs, each with M entries. FIFOs ensure that data being sent from the same source tile is received in the same order. Having multiple FIFOs enables multiple source tiles to send data concurrently using different FIFOs. It also enables data to be received through the network independently of receive instruction ordering in the program. This independence is important because receive instructions are executed in program order in a blocking manner for hardware simplicity.

Each send and receive instruction has a *fifo-id* operand that specifies the receiving FIFO to be used for incoming data. Using the FIFO ID instead of the sender tile ID provides additional flexibility for the compiler to apply FIFO virtualization, where a FIFO can be used by different sender tiles in different programs or program phases while keeping the number of physical FIFOs small. The key insight is that a typical ML layer will receive inputs from the tiles mapped to the previous layer only. Therefore, using 16 FIFOs (despite the node having 138 tiles) supports workloads with up to *(16 tiles)*(8 cores)*(2 MVMU)*128* previous layer activations, which suffices for large-scale ML workloads.

4.3 Summary

In summary, PUMA tiles enable inter-core synchronization, inter-core/tile pipelines to contain shared memory size, and FIFO virtualization for efficient inter-tile communication.

5 Compiler

PUMA is a spatial architecture (not a data-parallel processor) which means that each core/tile executes a different set of instructions. Writing different code for each core/tile is not scalable as applications grow in size and complexity. A compiler is therefore mandatory for programmer productivity. This section describes key aspects of the compiler, while

```
01  Model m = Model::create("example");
02
03  InVector x = InVector::create(m, M, "x");
04  InVector y = InVector::create(m, M, "y");
05  OutVector z = OutVector::create(m, N, "z");
06
07  ConstMatrix A = ConstMatrix::create(m, M, N, "A");
08  ConstMatrix B = ConstMatrix::create(m, M, N, "B");
09
10  z = tanh(A*x + B*y);
11
12  g.compile();
```

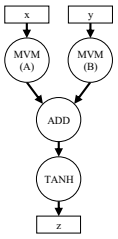

Figure 7. Simple Code Example

other implementation details of the compiler and the rest of the software stack are described in another paper [7].

5.1 Programming Interface

The PUMA compiler is a runtime compiler implemented as a C++ library. A simple code example is shown in Figure 7. The programmer first creates a model (line 01) with input/output vectors (lines 03-05) and constant matrices (lines 07-08). The programmer may also create vector streams which are useful for setting up inter-core/tile pipelines (see Section 4.1.2). The programmer then describes a computation (line 10) which executes at run time to build a graph of the model (Figure 7 on the right). Finally, the model is compiled (line 12) to generate PUMA assembly code for each core and tile. In addition to this native interface, ONNX bindings are also provided for further adoption and interoperability, enabling the compilation of models written in popular DNN frameworks such as Caffe2, PyTorch, Cognitive Toolkit, MXNet, and others.

5.2 Graph Partitioning

The first step in the compilation process is graph partitioning. The compiler divides tensors into 2D tiles, each the size of one MVMU, with appropriate padding, and divides the corresponding vectors and operations in the model accordingly. Next, the graph is hierarchically partitioned, distributing subgraphs to different MVMUs, cores, and tiles as shown in the example in Figure 8. The partitioning scheme used in this paper prioritizes placing MVMUs that feed to the same outputs together on the same core/tile, followed by those that read the same inputs, followed by those that feed each other (i.e., producer-consumer MVMUs). After partitioning the graph, the compiler inserts load/store operations across cores and allocates shared memory accordingly, reusing memory locations when there is pipelining. The compiler also inserts send/receive operations across tiles and assigns FIFO IDs accordingly (see Section 4.2), thereby virtualizing the FIFOs and ensuring there are no conflicts.

5.3 Instruction Scheduling

After the graph is partitioned into a sub-graph for each core/tile, the compiler schedules instructions by linearizing each sub-graph. Instruction scheduling has three main objectives: reducing register pressure, capturing ILP of MVM operations, and avoiding deadlock.

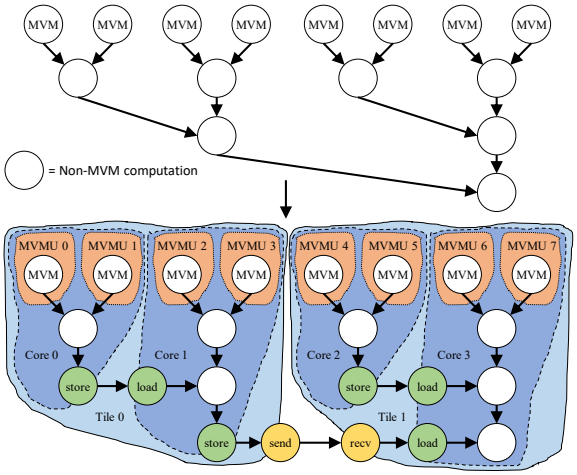

Figure 8. Graph Partitioning Example

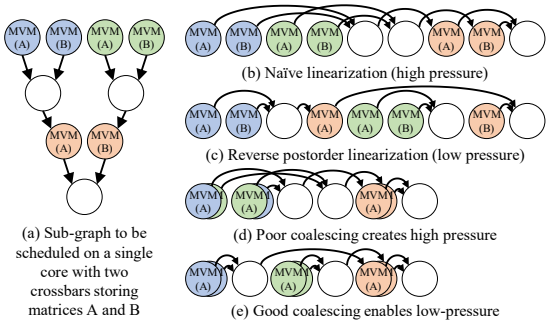

Figure 9. Instruction Scheduling Example

5.3.1 Reducing register pressure

There are many possible ways to linearize the sub-graph, as long as source operations are visited before their destinations to enforce data dependencies. We use a reverse post-order traversal which prioritizes consuming produced values before producing new ones. This ordering reduces the number of live values, hence the register pressure. Figure 9(b) and (c) show two correct linearizations of the sub-graph in Figure 9(a). Reverse postorder in Figure 9(c) results in fewer live values at any point in time.

5.3.2 Capturing ILP of MVM operations

As explained in Section 3.2.4, it is desirable to run different MVMUs in the same core simultaneously. The compiler must therefore fuse independent MVM operations on different MVMUs in the same core. We call this optimization *MVM coalescing*. MVMs can be coalesced when there are no data dependences between them. It is also desirable to coalesce MVMs whose results are consumed soon after one another to reduce register pressure. The compiler's strategy for coalescing is to first look for coalescing candidates among MVMs that are tiles of the same large MVM operation. Once those are exhausted, the remaining MVMs are coalesced by traversing the graph in reverse poster-order (before linearization) and fusing each MVM node with the first eligible

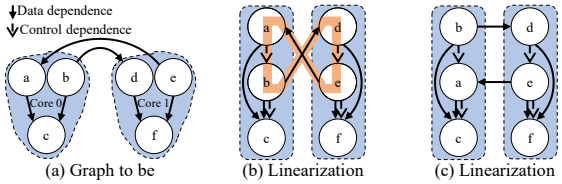

Figure 10. Deadlock Avoidance Example

candidates in the traversal order. The dependence information is updated every time a fusion takes place. Finally, the graph is traversed one last time to perform the linearization. Figure 9(d) and (e) show two example linearizations, with Figure 9(e) following the proposed approach resulting in fewer live values at once.

5.3.3 Avoiding deadlock

Linearizing the sub-graph of each core introduces control edges to the graph. Since communication across cores is blocking (see Section 4.1.1), cycles introduced by improper linearization cause deadlock as shown in the example in Figure 10(b). For this reason, sub-graphs are not linearized independently. The entire graph is linearized at once placing instruction in the corresponding core/tile sequence to ensure a globally consistent linearization order.

5.4 Register Allocation

The final step in the compilation is register allocation. Recall that a core has three sets of registers: *XbarIn*, *XbarOut*, and general purpose. XbarIn (XbarOut) registers can be written (read) by any non-MVM instruction but can only be read (written) by MVM instructions. General purpose registers can be read and written by any non-MVM instructions. Liveness analysis is performed on each of these sets of registers separately. Conflicting live ranges in *XbarIn* and *XbarOut* registers result in spilling to general purpose registers via *copy* instructions. Conflicting live ranges in general purpose registers result in spilling to shared memory via load/store instructions.

5.5 Summary

In summary, the PUMA compiler provides a high-level programming interface and performs graph partitioning, instruction scheduling, and register allocation to generate low-level assembly. Instruction scheduling aims at reducing register pressure, MVM colescing, and avoiding deadlock.

6 Evaluation Methodology

6.1 PUMAsim

We have implemented a detailed architectural simulator (PUMAsim) to evaluate the performance and energy consumption of PUMA. PUMAsim runs applications compiled to the PUMA ISA and provides detailed traces of execution. The simulator incorporates functionality, timing, and

Table 3. PUMA Hardware Characteristics

PUMA Tile at 1GHz on 32nm Technology node				
Component	**Power (mW)**	**Area (mm^2)**	**Parameter**	**Specification**
Control Pipeline	0.25	0.0033	# stages	3
Instruction Memory	1.52	0.0031	capacity	4KB
Register File	0.477	0.00192	capacity	1KB
MVMU	19.09	0.012	# per core dimensions	2 128×128
VFU	1.90	0.004	width	1
SFU	0.055	0.0006	-	-
Core	42.37	0.036	# per tile	8
Tile Control Unit	0.5	0.00145	-	-
Tile Instruction Memory	1.91	0.0054	capacity	8KB
Tile Data Memory	17.66	0.086	capacity technology	64KB eDRAM
Tile Memory Bus	7	0.090	width	384 bits
Tile Attribute Memory	2.77	0.012	# entries technology	32K eDRAM
Tile Receive Buffer	9.14	0.0044	# fifos fifo depth	16 2
Tile	373.8	0.479	# per node	138
On-chip Network	570.63	1.622	flit_size # ports conc	32 4 4
Node	62.5K	90.638	-	-
Off-chip Network (per node)	10.4K	22.88	type link bandwidth	HyperTransport 6.4 GB/sec

Table 4. Benchmarking Platforms

Name	**Platform Characteristics**
Haswell	Intel Xeon E5-2650v3, 10-cores per socket, Dual Socket, 128GB DDR4
Skylake	Intel Xeon 8180, 28-cores per socket, Dual Socket, 64GB DDR4
Kepler	Nvidia Tesla K80, 2496 CUDA Cores, Dual GPUs (only 1 used), 12GB GDDR5
Maxwell	Nvidia Geforce TitanX, 3072 CUDA Cores, 12GB GDDR5
Pascal	Nvidia Tesla P100, 3584 CUDA Cores, 16GB HBM2

power models of all system components. The datapath for the PUMA core and tile was designed at Register-Transfer Level (RTL) in Verilog HDL and mapped to IBM 45nm SOI technology using Synopsys Design Compiler which was used to measure the area and power consumption. Subsequently, these area and power numbers were added to PUMAsim for system-level evaluations of workloads. For fair comparison with other application-specific accelerators, the datapath energy numbers have been scaled to the 32nm technology node. Memory modules are modelled in Cacti 6.0 [76] to estimate the area, power, and latency. The on-chip-network is modelled using the cycle-level Booksim 2.0 interconnection network simulator [59] and Orion 3.0 [63] for energy and area models. We use a chip-to-chip interconnect model similar to DaDianNao's [20] which has also been adopted by other accelerators. The MVMU power and area models are adapted from ISAAC [95]. The memristors have a resistance range of $100k\Omega - 1M\Omega$ and read voltage of 0.5V. The ADC is based on the Successive Approximation Register (SAR) design, and its area and power were obtained from the ADC survey and analysis [77, 107].

In the present study, we do not compromise ML accuracy as we conservatively choose 2-bit memristor crossbar cells. Note that laboratory demonstrations have shown up to 6-bit capabilities [52]. We use 16 bit fixed-point precision that provides very high accuracy in inference applications [20, 95]. Table 3 shows the PUMA configuration used in our analysis and lists the area-energy breakdown of PUMA components.

Table 5. Benchmarks

DNN Type	Application	DNN Name	# FC Layers	# LSTM Layers	# Conv Layers	# Parameters	Non-linear Function	Sequence Size
MLP	Object Detection	MLP_{L4}	4	-	-	5M	Sigmoid	-
		MLP_{L5}	5	-	-	21M	Sigmoid	-
Deep LSTM	Neural Machine Translation	NMT_{L3}	1	6 (3 Enc.,3 Dec., 1024 cells)	-	91M	Signmod, Tanh	50
		NMT_{L5}	1	10 (5 Enc.,5 Dec., 1024 cells)	-	125M	Signmod, Tanh	50
Wide LSTM	Language Modelling	BigLSTM	1	2 (8192 cell, 1024 proj)	-	856M	Signmod, Tanh, LogSoftMax	50
		LSTM-2048	1	1 (8192 cell, 2048 proj)	-	554M	Sigmoid, Tanh, LogSoftMax	50
CNN	Image Recognition	Vgg16	3	-	13	136M	ReLU	-
		Vgg19	3	-	16	141M	ReLU	-

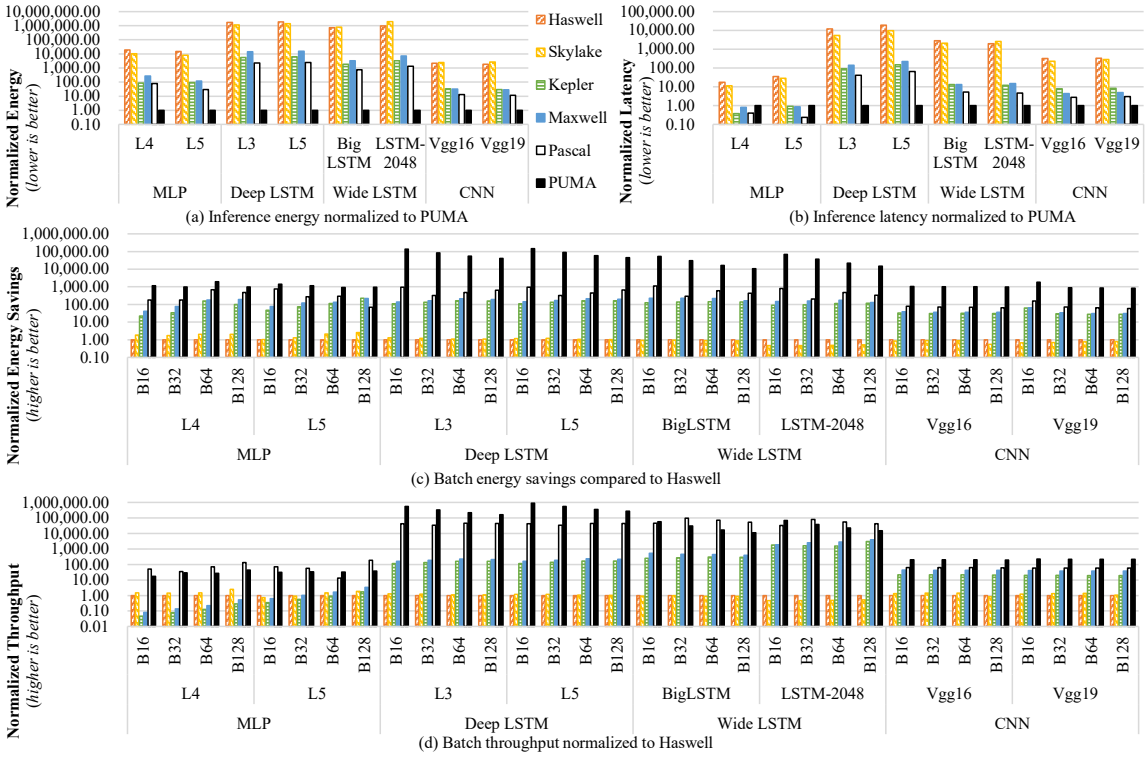

Figure 11. Energy and Latency Results

6.2 System and Workloads

We choose popular server grade CPUs and GPUs (listed in Table 4), Google TPU [61] (CMOS-based ASIC) and ISAAC [95] (application specific memristor-based accelerator) for evaluating PUMA. To measure power consumption of CPUs and GPUs, we used management tools such as board management control (BMC) and nvidia-smi respectively. For GPUs, we do not include full system power, just the board/device power. We run multiple iterations of the benchmarks on the GPU, discarding the longer warmup iterations and reporting results from the faster and more stable iterations.

Torch7 [29] was used to execute the ML models for CPUs and GPUs. The PUMA compiler was used to compile models for PUMA. These models used are listed in Table 5.

7 Results

7.1 Inference Energy

Figure 11(a) shows PUMA inference energy compared to other platforms. PUMA achieves massive energy reduction across all benchmarks for all platforms. Energy improvements come from two sources: lower MVM compute energy from crossbars and lower data movement energy by avoiding weight data movement.

CNNs show the least energy reduction over CMOS architectures (11.7×-13.0× over Pascal). Recall that CNNs have a lot of weight reuse because of the sliding window computation (discussed in Section 2.3.1). Hence, CMOS architectures can amortize DRAM accesses of weights across multiple computations. For this reason, PUMA's energy savings in CNNs come primarily from the use of crossbars for energy efficient MVM computation.

MLPs and LSTMs have little or no weight reuse (discussed in Section 2). Therefore, in addition to efficient MVM computation, PUMA has the added advantage of eliminating weight data movement. For this reason, we see much better energy reductions for MLPs (30.2×-80.1× over Pascal), Deep LSTMs (2,302×-2,446× over Pascal), and Wide LSTMs (758×-1336× over Pascal).

LSTMs (both deep and wide) show better reductions than MLPs because they have much larger model sizes (see # Parameters in Table 5). As model grows in size, weight data grows at $O(n^2)$ and input data grows at $O(n)$. For this reason, we see an increasing disparity between CMOS architectures

which move both weight and input data, and PUMA which only moves input data.

Wide LSTMs have few layers (1-2) with very large matrices, whereas Deep LSTMs have many layers (6-10) with smaller matrices. The large matrices in Wide LSTMs span mutiple PUMA cores/tiles to compute one logical MVM, incurring higher intra-layer data movement overheads. Hence, Deep LSTMs show higher energy benefits than Wide LSTMs.

7.2 Inference Latency

Figure 11(b) shows PUMA inference latency compared to other evaluated platforms. PUMA achieves latency improvements across all platforms except MLPs on some GPUs. Latency improvements come from three sources: lower MVM compute latency from crossbars, no weight data access latency, and spatial architecture pipelining which exploits inter-layer parallelism.

CNNs show the least latency improvement over CMOS architectures (2.73×-2.99× over Pascal). Since CNNs are compute-bound, CMOS architectures can hide the data access latency. Thus, PUMA's primary latency improvements in CNNs come from the use of crossbars for low-latency MVM computation and spatial architecture pipelining.

LSTMs on the other hand are memory-bound. PUMA has the added advantage of eliminating weight data access latency in addition to low-latency MVM computation. For this reason, we see much better latency improvements for Deep LSTMs (41.6×-66.0× over Pascal) and Wide LSTMs (4.70×-5.24× over Pascal) than we see for CNNs. In comparing Deep and Wide LSTMs, Deep LSTMs have more layers than Wide LSTMs, hence more inter-layer parallelism to exploit spatial architecture pipelining (see #LSTM Layers in Table 5). Moreover, Deep LSTMs have less intra-layer communication than Wide LSTMs, hence lower data access latency.

MLPs show slowdown compared to some GPU datapoints (0.24×-0.40× compared to Pascal). The reason is that despite MLPs being memory-bound, the sizes of MLPs are typically small enough. Hence, the memory bandwidth bottleneck is not as pronounced, so they perform fairly well on GPUs. Moreover, MLPs have no inter-layer parallelism so they do not exploit spatial architecture pipelining (Section 4.1.2). Nevertheless, PUMA's order of magnitude energy reduction is still beneficial for MLPs in energy-constrained environments.

7.3 Batch Throughput and Energy

Inference applications are not usually intended for large batch sizes due to real-time application requirements. Nevertheless, CMOS architectures perform well with large batch sizes because of the weight reuse that data-batching exposes. For this reason, we compare PUMA's batch energy and throughput with the other platforms in Figure 11(c) and (d) respectively. Batch sizes of 16, 32, 64, and 128 are used.

Table 6. Comparison with ML Accelerators

Platform	PUMA	TPU [61]	ISAAC [95]
Year	2018	2017	2016
Technology	CMOS(32nm)-Memristive	CMOS(28nm)	CMOS(32nm)-Memristive
Clock (MHz)	1000	700	1200
Precision	16-bit fixed point	16-bit fixed point	16-bit fixed point
Area (mm^2)	90.6	330*	85.4
Power (W)	62.5	45	65.8
Peak Throughput (TOPS/s†)	52.31	23‡	69.53
Peak AE (TOPS/s/mm^2)	**0.58**	**0.07**	**0.82**
Peak PE (TOPS/s/W)	0.84	0.51	1.06
Best AE - MLP	0.58	0.009	-
Best AE - LSTM	0.58	0.003	-
Best AE - CNN	0.58	0.06	0.82
Best PE - MLP	0.84	0.07	-
Best PE - LSTM	0.84	0.02	-
Best PE - CNN	0.84	0.48	1.06

* Less than or equal to half of Haswell's die area [61]
† Tera operations per second (multiply and add are counted as two separate operations)
‡ 92 TOPS for 8-bit arithmetic, scaled by 4 for 16-bit arithmetic [61]

Table 7. Programmability Comparison with ISAAC

Platform	PUMA	ISAAC
Architecture	Instruction execution pipeline, flexible inter-core synchronization	Application specific state machine
	Vector Functional Unit, ROM-Embedded RAM	Sigmoid unit
Programmability	Compiler-generated instructions (per tile & core)	Manually configured state machine (per tile)
Workloads	CNN, MLP, LSTM, RNN, GAN, BM, RBM, SVM, Linear Regression, Logistic Regression	CNN

PUMA continues to have superior energy efficiency across all benchmarks and batch sizes. It also delivers better throughput in most cases, except when compared to Pascal on MLPs and Wide LSTMs. The benefits slightly decrease with larger batches because they expose more weight reuse which benefits CMOS architectures while PUMA's efficiency remains constant across batch sizes. Nevertheless, PUMA continues to perform well even when the batch size is very large.

7.4 Comparison with ML Accelerators

7.4.1 Google TPU

Table 6 compares key technology features and efficiency metrics for TPU [61] and PUMA. PUMA has 8.3× higher peak area-efficiency (TOPS/s/mm^2) than TPU. Since PUMA does not rely on weight reuse to improve throughput like CMOS architectures do, its area-efficiency remains constant across workloads and batch sizes. On the other hand, TPU's peak throughput is almost an order of magnitude lower for applications with low data reuse due to its inability to amortize weight data movement. PUMA has 64.4×, 193×, and 9.7× higher area-efficiency than TPU for MLP, LSTM, and CNN respectively for the best TPU batch size.

PUMA has 1.65× higher peak power-efficiency (TOPS/s/W) than TPU, with similar trends and reasoning as area-efficiency for specific workloads. We expect PUMA's power-efficiency advantage over TPU to grow by over 3×, as silicon processes scale from 32nm to 7nm and 5nm. Thanks to PUMA's higher peak throughput, we can follow the power reduction scaling curve at constant performance. Conversely, to narrow the performance gap, TPU would follow a scaling curve closer to the performance increase curve at constant power. Note that the former scaling curve is much faster (up to ~40% power reduction per silicon node compared with ~20% performance increase). Further, ADC power efficiency has also been following similar and very fast scaling trend with ~2× power reduction per 1.8 years at the same sampling rate [78].

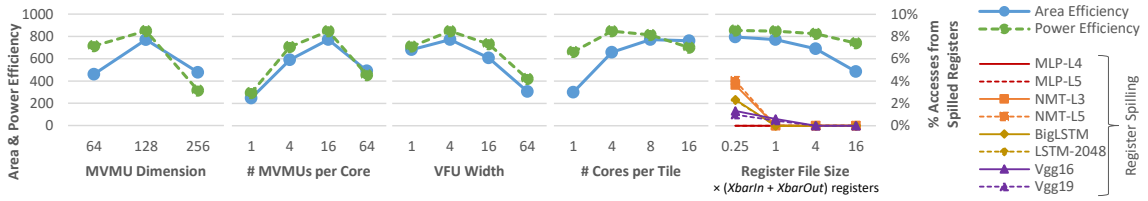

Figure 12. Design Space Exploration: Tile Area Efficiency in GOPS/s/mm^2 and Tile Power Efficiency in GOPS/s/W

Table 8. Evaluation of Optimizations

Workload	Input Shuffling (energy reduction, lower is better)	Shared Memory Sizing (energy reduction, lower is better)	Graph Partitioning (energy reduction, lower is better)	Register Pressure (% accesses from spilled registers)	MVM Coalescing (latency reduction, lower is better)
MLP_{L4}	-	0.70×	0.81×	0%	0.60×
MLP_{L5}	-	0.66×	0.79×	0%	0.66×
NMT_{L3}	-	0.65×	0.65×	0%	0.63×
NMT_{L5}	-	0.63×	0.63×	0%	0.63×
BigLSTM	-	0.58×	0.61×	0%	0.76×
LSTM-2048	-	0.58×	0.62×	0%	0.84×
Vgg16	0.84×	0.75×	0.37×	1.96%	0.69×
Vgg19	0.85×	0.75×	0.43×	1.71%	0.71×

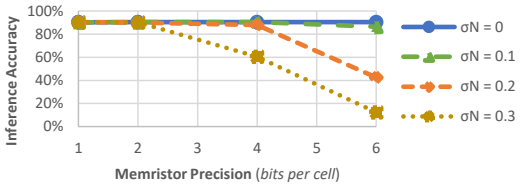

Figure 13. Inference Accuracy

7.4.2 ISAAC

Table 6 compares the peak area and power efficiency of PUMA with ISAAC [95], a memristor-based accelerator customized for CNNs. PUMA has 20.7% lower power efficiency and 29.2% lower area efficiency than ISAAC due to the overhead of programmability. Table 7 compares the programmability of PUMA and ISAAC.

7.4.3 PUMA with Digital MVMUs

To demonstrate the importance of analog computing for MVMU efficiency, we compare PUMA with a hypothetical equivalent that uses digital MVMUs. A memristive 128×128 MVMU performs 16,384 MACs in 2304 ns consuming 43.97 nJ. A digital MVMU would require 8.97× more area to achieve the same latency and would consume 4.17× more energy. Using a digital MVMU would increase the total chip area of the accelerator by 4.93× for the same performance and would consume 6.76× energy (factoring in data movement energy due to increased area).

7.4.4 Tensor Cores

Nvidia V100 GPUs with tensor cores (FP16) can be up to 6× more energy-efficient (architecture, tensor cores, and half-precision) than Pascal GPUs. Therefore, PUMA can still achieve energy gains over GPUs with tensor cores, despite the technology difference (PUMA: 32nm, V100: 12nm).

7.5 Evaluation of Optimizations

Table 8 shows an evaluation of various optimizations described throughout the paper. ***Input shuffling*** reduces energy consumed by data movement within a core by leveraging input reuse in CNNs. ***Shared memory sizing*** keeps the shared memory small by leveraging inter-core/tile pipelining. The baseline here, sizes the shared memory with what would be needed without pipelining, which is 1×, 50.51×, 21.61×, and 15.91× larger for MLPs, Deep LSTMs, Wide LSTMs, and CNNs respectively. Note that MLP does not benefit from inter-tile pipelining because it does not exhibit any weight reuse (Section 2.1). ***Graph partitioning*** (compared to a baseline that partitions the graph randomly) reduces the number of loads, stores, sends, and receives, hence the overall energy. ***Register pressure*** is kept low by the compiler with little or no accesses from spilled registers across benchmarks. ***MVM coalescing*** runs MVMUs in parallel within a core which reduces latency.

7.6 Design Space Exploration

Figure 12 shows a PUMA tile's peak area and power efficiency swept across multiple design space parameters. For each sweep, all other parameters are kept at the sweetspot (PUMA configuration with maximum efficiency). Efficiency is measured using a synthetic benchmark: an MVM operation on each MVMU, followed by a VFU operation, then a ROM-Embedded RAM look-up.

Increasing the ***MVMU dimension*** increases the number of crossbar multiply-add operations quadratically and the number of peripherals linearly resulting in more amortization of overhead from peripherals. However, larger MVMUs also require ADCs with higher resolution and ADC overhead grows non-linearly with resolution, which counterbalances the amortization. Increasing the ***# MVMUs per core*** increases efficiency because of the high efficiency of memristor crossbars relative to CMOS digital components. However, with too many MVMUs, the VFU becomes a bottleneck which degrades efficiency. Increasing the ***VFU width*** degrades efficiency because of the low efficiency of CMOS relative to memristor crossbars. However, a VFU that is too narrow becomes a bottleneck. The sweetspot is found at 4 vector lanes. Increasing the ***# cores per tile*** improves efficiency until shared memory bandwidth becomes the bottleneck. Increasing the ***register file size*** results in lower efficiency, however a register file that is too small results in too many register spills.

Figure 13 shows PUMA's inference accuracy for different memristor bit precision (bits per device) and write noise levels (σ^N). Higher precision can lead to larger accuracy

loss due to the reduction in noise margin. It can be seen that PUMA with 2-bit memristor performs well even at high noise levels. Real CMOS hardware follows the $\sigma^N = 0$ noise level. Further, recent research have explored coding schemes for reliable memristor computation at high precision [38, 92].

8 Related Work

Sze et al. [103] provide a thorough survey of deep learning accelerators. In the digital realm, accelerators can be classified as weight stationary spatial architectures [15, 17, 36, 43, 85, 93], output stationary spatial architectures [33, 47, 86], spatial architectures with no local reuse [19, 20, 117], and row stationary spatial architectures [21]. Many designs also support optimizations and features including weight pruning and exploiting sparsity [3, 4, 25, 32, 48, 84, 89, 119], reducing precision [8, 62], stochastic computing [65, 90], layer fusing [6], meeting QoS/QoR requirements [108], graph tuning [56], and reconfigurable interconnects [68]. Digital accelerators have varied in their degree of flexibility, ranging from custom accelerators specialized for a particular field [14, 79, 101, 114], to accelerators that are fully programmable via an ISA [61, 71, 72, 105, 119]. FPGAs have also been popular targets for building accelerators [37, 45, 74, 86, 96, 97, 110, 117]. All these works remain in the digital domain, while PUMA leverages hybrid digital-analog computing.

Near-memory acceleration for ML has been proposed using DRAM [42, 64, 70] and SRAM [111, 118]. PUMA uses non-volatile memristive crossbars for near-memory acceleration.

Many machine learning accelerators have been proposed that leverage memristor crossbars [9, 10, 12, 22, 23, 53, 58, 66, 73, 88, 95, 100]. These accelerators have been demonstrated on several types of workloads including BSBs [53], MLPs [22, 39, 73, 88], SNNs [9, 66], BMs [12], and CNNs [22, 23, 95, 100, 109]. Some accelerators support inference only [9, 23, 53, 73, 88, 95] while others also support training [12, 22, 66, 100]. Ji et al. [57, 58] transforms neural networks to configure such accelerators. These accelerators vary in flexibility, but even the most flexible rely on state machine configuration and have only been demonstrated on a few types of workloads. PUMA is the first memristor-based accelerator for machine learning inference that is ISA-programmable and general-purpose.

Fujiki et al. [40] propose an ISA-programmable memristor-based accelerator. Their accelerator is a data-parallel accelerator whereas PUMA is a data-flow accelerator with more capability for producer-consumer synchronization. Moreover, their accelerator optimizes crossbars for vector operations in general-purpose workloads whereas PUMA optimizes crossbars for MVM operations prevalent in machine learning and uses digital VFUs for vector operations rather than crossbars.

Chung et al. [24] propose Brainwave, which is a spatial accelerator built with FPGAs. Compared to Brainwave, a PUMA core performs 0.26 million 16-bit ops, equivalent to 1.04 million 8-bit ops, per coalesced MVM instruction. A Brainwave NPU performs 1.3million 8-bit ops per instruction. Therefore, PUMA and Brainwave have comparable control granularity while PUMA has 40.8x higher storage-density (Brainwave Stratix10 estimate).

Memristors have also been proposed for building byte-addressable non-volatile memories. There have been various works centered around system support for non-volatile memory, including file systems [30], memory allocators [11, 83], programming models [16, 27, 106], durable data structures [31, 55, 113], representation of pointers [18, 28, 34, 35], and architecture support [60, 80, 82, 99].

There have been concerns in the community about memristor manufacturability. We distinguish between medium-density embedded memristor applications and high-density storage-class memory (SCM). Memristors in PUMA use 1T1R configuration which have been shown to have good manufacturability [50]. They are very different from SCM, where the selector transistor may be replaced with an in-line two-terminal selector device for higher density which complicates manufacturability. Panasonic formed a joined venture with UMC foundry in 2017 to enable integration of memristors to UMC 40nm CMOS process with first samples planned in 2018 [115]. TSMC also completed development of their own memristor technology that entered risk production in 40nm ULP CMOS process node at the end of 2017 [1].

9 Conclusion

PUMA is the first ISA-programmable accelerator for ML inference that uses hybrid CMOS-memristor technology. It enhances memristor crossbars with general purpose execution units carefully designed to maintain crossbar area/energy efficiency and storage density. Our accelerator design comes with a complete compiler to transform high-level code to PUMA ISA and a detailed simulator for estimating performance and energy consumption. Our evaluations show that PUMA can achieve significant improvements compared to state-of-the-art CPUs, GPUs, and ASICs for ML acceleration.

Acknowledgments

This work is supported by Hewlett Packard Labs and the US Department of Energy (DOE) under Cooperative Agreement DE-SC0012199, the Blackcomb 2 Project. In addition, John Paul Strachan acknowledges support in part from the Intelligence Advanced Research Projects Activity (IARPA) via contract number 2017-17013000002. This work was also supported in part by the Center for Brain-inspired Computing (C-BRIC), one of six centers in JUMP, a DARPA sponsored Semiconductor Research Corporation (SRC) program.

References

[1] 2018. TSMC Annual Report 2017. *TSMC* (Mar 2018).

[2] Alan Agresti. 2002. *Logistic regression.* Wiley Online Library.

[3] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. 2017. Bit-pragmatic Deep Neural Network Computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 382–394.

[4] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: ineffectual-neuron-free deep neural network computing. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 1–13.

[5] Fabien Alibart, Elham Zamanidoost, and Dmitri B Strukov. 2013. Pattern classification by memristive crossbar circuits using ex situ and in situ training. *Nature communications* 4 (2013).

[6] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[7] Joao Ambrosi, Aayush Ankit, Rodrigo Antunes, Sai Rahul Chalamalasetti, Soumitra Chatterjee, Izzat El Hajj, Guilherme Fachini, Paolo Faraboschi, Martin Foltin, Sitao Huang, Wen mei Hwu, Gustavo Knuppe, Sunil Vishwanathpur Lakshminarasimha, Dejan Milojicic, Mohan Parthasarathy, Filipe Ribeiro, Lucas Rosa, Kaushik Roy, Plinio Silveira, and John Paul Strachan. 2018. Hardware-Software Co-Design for an Analog-Digital Accelerator for Machine Learning. In *Rebooting Computing (ICRC), 2018 IEEE International Conference on*. IEEE.

[8] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. 2017. YodaNN: An Architecture for Ultra-Low Power Binary-Weight CNN Acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2017).

[9] Aayush Ankit, Abhronil Sengupta, Priyadarshini Panda, and Kaushik Roy. 2017. RESPARC: A Reconfigurable and Energy-Efficient Architecture with Memristive Crossbars for Deep Spiking Neural Networks. In *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 27.

[10] Aayush Ankit, Abhronil Sengupta, and Kaushik Roy. 2017. Trannsformer: Neural network transformation for memristive crossbar based neuromorphic system design. In *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press, 533–540.

[11] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 2016. Makalu: Fast Recoverable Allocation of Non-volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 677–694.

[12] Mahdi Nazm Bojnordi and Engin Ipek. 2016. Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 1–13.

[13] G. W. Burr, R. M. Shelby, S. Sidler, C. di Nolfo, J. Jang, I. Boybat, R. S. Shenoy, P. Narayanan, K. Virwani, E. U. Giacometti, B. N. Kurdi, and H. Hwang. 2015. Experimental Demonstration and Tolerancing of a Large-Scale Neural Network (165 000 Synapses) Using Phase-Change Memory as the Synaptic Weight Element. *IEEE Transactions on Electron Devices* 62, 11 (Nov 2015), 3498–3507.

[14] Ruizhe Cai, Ao Ren, Ning Liu, Caiwen Ding, Luhao Wang, Xuehai Qian, Massoud Pedram, and Yanzhi Wang. 2018. VIBNN: Hardware Acceleration of Bayesian Neural Networks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 476–488.

[15] Lukas Cavigelli, David Gschwend, Christoph Mayer, Samuel Willi, Beat Muheim, and Luca Benini. 2015. Origami: A Convolutional Network Accelerator. In *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI (GLSVLSI '15)*. ACM, New York, NY, USA, 6.

[16] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 433–452.

[17] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. 2010. A Dynamically Configurable Coprocessor for Convolutional Neural Networks. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 247–257.

[18] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. 2017. Efficient support of position independence on non-volatile memory. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 191–203.

[19] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, 269–284.

[20] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. 2014. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 609–622.

[21] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 367–379.

[22] Ming Cheng, Lixue Xia, Zhenhua Zhu, Yi Cai, Yuan Xie, Yu Wang, and Huazhong Yang. 2017. TIME: A Training-in-memory Architecture for Memristor-based Deep Neural Networks. In *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 26.

[23] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 27–39.

[24] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro* 38, 2 (2018).

[25] Jaeyong Chung and Taehwan Shin. 2016. Simplifying deep neural networks for neuromorphic architectures. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 1–6.

[26] Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. 2010. Deep, big, simple neural nets for handwritten digit recognition. *Neural computation* 22, 12 (2010), 3207–3220.

[27] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 105–118.

[28] Nachshon Cohen, David T. Aksun, and James R. Larus. 2018. Object-oriented Recovery for Non-volatile Memory. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 153 (Oct. 2018), 22 pages.

[29] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*.

[30] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the*

ACM SIGOPS 22nd symposium on Operating systems principles. ACM.

[31] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G Khatib, and Cristian Ungureanu. 2016. Revisiting hash table design for phase change memory. *ACM SIGOPS Operating Systems Review* 49, 2 (2016), 18–26.

[32] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, Xiaolong Ma, Yipeng Zhang, Jian Tang, Qinru Qiu, Xue Lin, and Bo Yuan. 2017. CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-circulant Weight Matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 395–408.

[33] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 92–104.

[34] Izzat El Hajj, Thomas B. Jablin, Dejan Milojicic, and Wen-mei Hwu. 2017. SAVI Objects: Sharing and Virtuality Incorporated. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 45 (2017), 24 pages.

[35] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. 2016. SpaceJMP: Programming with Multiple Virtual Address Spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 353–368.

[36] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello. 2010. Hardware accelerated convolutional neural networks for synthetic vision systems. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. 257–260.

[37] Clément Farabet, Cyril Poulet, Jefferson Y Han, and Yann LeCun. 2009. Cnp: An fpga-based processor for convolutional networks. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 32–37.

[38] Ben Feinberg, Shibo Wang, and Engin Ipek. 2018. Making memristive neural network accelerators reliable. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 52–65.

[39] B. Feinberg, S. Wang, and E. Ipek. 2018. Making Memristive Neural Network Accelerators Reliable. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[40] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2018. In-Memory Data Parallel Processor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 1–14.

[41] Terrence S Furey, Nello Cristianini, Nigel Duffy, David W Bednarski, Michel Schummer, and David Haussler. 2000. Support vector machine classification and validation of cancer tissue samples using microarray expression data. *Bioinformatics* 16, 10 (2000), 906–914.

[42] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 751–764.

[43] Vinayak Gokhale, Jonghoon Jin, Aysegul Dundar, Berin Martini, and Eugenio Culurciello. 2014. A 240 g-ops/s mobile coprocessor for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 682–687.

[44] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems*. 2672–2680.

[45] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. 2017. A Survey of FPGA Based Neural Network Accelerator. *arXiv preprint arXiv:1712.08934* (2017).

[46] Xinjie Guo, F Merrikh Bayat, M Bavandpour, M Klachko, MR Mahmoodi, M Prezioso, KK Likharev, and DB Strukov. 2017. Fast, energy-efficient, robust, and reproducible mixed-signal neuromorphic classifier based on embedded NOR flash memory technology. In *Electron Devices Meeting (IEDM), 2017 IEEE International*. IEEE, 6–5.

[47] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*. 1737–1746.

[48] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 243–254.

[49] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*. 1135–1143.

[50] Yukio Hayakawa, Atsushi Himeno, Ryutaro Yasuhara, W Boullart, E Vecchio, T Vandeweyer, T Witters, D Crotti, M Jurczak, S Fujii, et al. 2015. Highly reliable TaO x ReRAM with centralized filament for 28-nm embedded application. In *VLSI Technology (VLSI Technology), 2015 Symposium on*. IEEE, T14–T15.

[51] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[52] Miao Hu, Catherine Graves, Can Li, Yunning Li, Ning Ge, Eric Montgomery, Noraica Davila, Hao Jiang, R. Stanley Williams, J. Joshua Yang, Qiangfei Xia, and John Paul Strachan. 2018. Memristor-based analog computation and neural network classification with a dot product engine. *Advanced Materials* (2018).

[53] Miao Hu, Hai Li, Qing Wu, and Garrett S. Rose. 2012. Hardware Realization of BSB Recall Function Using Memristor Crossbar Arrays. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*. ACM, New York, NY, USA, 498–503.

[54] Miao Hu, John Paul Strachan, Zhiyong Li, Emmanuelle M Grafals, Noraica Davila, Catherine Graves, Sity Lam, Ning Ge, Jianhua Joshua Yang, and R Stanley Williams. 2016. Dot-product engine for neuromorphic computing: programming 1T1M crossbar to accelerate matrix-vector multiplication. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 1–6.

[55] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*. Springer, 313–327.

[56] Yu Ji, Youhui Zhang, Wenguang Chen, and Yuan Xie. 2018. Bridge the Gap Between Neural Networks and Neuromorphic Hardware with a Neural Network Compiler. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 448–460.

[57] Yu Ji, Youhui Zhang, Wenguang Chen, and Yuan Xie. 2018. Bridge the Gap between Neural Networks and Neuromorphic Hardware with a Neural Network Compiler. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 448–460.

[58] Yu Ji, YouHui Zhang, ShuangChen Li, Ping Chi, CiHang Jiang, Peng Qu, Yuan Xie, and WenGuang Chen. 2016. NEUTRAMS: Neural network transformation and co-design under neuromorphic hardware constraints. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–13.

[59] Nan Jiang, George Michelogiannakis, Daniel Becker, Brian Towles, and William J. Dally. 2013. *BookSim 2.0 User's Guide*.

[60] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. 2017. ATOM: Atomic durability in non-volatile memory through hardware logging. In *High Performance Computer Architecture (HPCA), 2017*

IEEE International Symposium on. IEEE, 361–372.

[61] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 1–12.

[62] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. 2016. Stripes: Bit-serial deep neural network computing. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[63] A. B. Kahng, B. Lin, and S. Nath. 2012. *Comprehensive Modeling Methodologies for NoC Router Estimation*. Technical Report. UCSD.

[64] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 380–392.

[65] Kyounghoon Kim, Jungki Kim, Joonsang Yu, Jungwoo Seo, Jongeun Lee, and Kiyoung Choi. 2016. Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks. In *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 124.

[66] Yongtae Kim, Yong Zhang, and Peng Li. 2015. A Reconfigurable Digital Neuromorphic Processor with Memristive Synaptic Crossbar for Cognitive Computing. *J. Emerg. Technol. Comput. Syst.* 11, 4, Article 38 (April 2015), 25 pages.

[67] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[68] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 461–475.

[69] Dongsoo Lee and Kaushik Roy. 2013. Area efficient ROM-embedded SRAM cache. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21, 9 (2013), 1583–1595.

[70] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. 2017. DRISA: A DRAM-based Reconfigurable In-Situ Accelerator. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 288–301.

[71] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. PuDianNao: A Polyvalent Machine Learning Accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 369–381.

[72] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An instruction set architecture for neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 393–405.

[73] Xiaoxiao Liu, Mengjie Mao, Beiye Liu, Hai Li, Yiran Chen, Boxun Li, Yu Wang, Hao Jiang, Mark Barnell, Qing Wu, et al. 2015. RENO: A high-efficient reconfigurable neuromorphic computing accelerator design. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*. IEEE, 1–6.

[74] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. 2016. Tabla: A unified template-based framework for accelerating statistical machine learning. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 14–26.

[75] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*.

[76] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. *CACTI 6.0: A Tool to Understand Large Caches*. Technical Report. HP Labs, HPL-2009-85.

[77] Boris Murmann. 2011. ADC performance survey 1997-2011. *http://www. stanford. edu/˜ murmann/adcsurvey. html* (2011).

[78] Boris Murmann. 2015. The race for the extra decibel: a brief review of current ADC performance trajectories. *IEEE Solid-State Circuits Magazine* 7, 3 (2015), 58–66.

[79] Sean Murray, William Floyd-Jones, Ying Qi, George Konidaris, and Daniel J Sorin. 2016. The microarchitecture of a real-time robot motion planning accelerator. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[80] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. 2017. An analysis of persistent memory use with WHISPER. In *ACM SIGARCH Computer Architecture News*, Vol. 45. ACM, 135–148.

[81] John Neter, Michael H Kutner, Christopher J Nachtsheim, and William Wasserman. 1996. *Applied linear statistical models*. Vol. 4. Irwin Chicago.

[82] Matheus Almeida Ogleari, Ethan L Miller, and Jishen Zhao. 2018. Steal but no force: Efficient hardware undo+ redo logging for persistent memory systems. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 336–349.

[83] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. 2017. Memory management techniques for large-scale persistent-main-memory systems. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1166–1177.

[84] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 27–40.

[85] Seongwook Park, Kyeongryeol Bong, Dongjoo Shin, Jinmook Lee, Sungpill Choi, and Hoi-Jun Yoo. 2015. 4.6 A1. 93TOPS/W scalable deep learning/inference processor with tetra-parallel MIMD architecture for big-data applications. In *Solid-State Circuits Conference-(ISSCC), 2015 IEEE International*. IEEE, 1–3.

[86] Maurice Peemen, Arnaud AA Setio, Bart Mesman, and Henk Corporaal. 2013. Memory-centric accelerator design for convolutional neural networks. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*. IEEE, 13–19.

[87] Mirko Prezioso, Farnood Merrikh-Bayat, BD Hoskins, GC Adam, Konstantin K Likharev, and Dmitri B Strukov. 2015. Training and operation of an integrated neuromorphic network based on metal-oxide memristors. *Nature* 521, 7550 (2015), 61–64.

[88] Shankar Ganesh Ramasubramanian, Rangharajan Venkatesan, Mrigank Sharad, Kaushik Roy, and Anand Raghunathan. 2014. SPINDLE:

SPINtronic deep learning engine for large-scale neuromorphic computing. In *Proceedings of the 2014 international symposium on Low power electronics and design*. ACM, 15–20.

[89] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 267–278.

[90] Ao Ren, Zhe Li, Caiwen Ding, Qinru Qiu, Yanzhi Wang, Ji Li, Xuehai Qian, and Bo Yuan. 2017. Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 405–418.

[91] Paul Resnick and Hal R Varian. 1997. Recommender systems. *Commun. ACM* 40, 3 (1997), 56–58.

[92] Ron M Roth. 2017. Fault-Tolerant Dot-Product Engines. *arXiv preprint arXiv:1708.06892* (2017).

[93] Murugan Sankaradas, Venkata Jakkula, Srihari Cadambi, Srimat Chakradhar, Igor Durdanovic, Eric Cosatto, and Hans Peter Graf. 2009. A massively parallel coprocessor for convolutional neural networks. In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*. IEEE, 53–60.

[94] Abhronil Sengupta, Yong Shim, and Kaushik Roy. 2016. Proposal for an all-spin artificial neural network: Emulating neural and synaptic functionalities through domain wall motion in ferromagnets. *IEEE transactions on biomedical circuits and systems* 10, 6 (2016), 1152–1160.

[95] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press.

[96] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[97] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN Accelerator Efficiency Through Resource Partitioning. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 535–547.

[98] Patrick M Sheridan, Fuxi Cai, Chao Du, Wen Ma, Zhengya Zhang, and Wei D Lu. 2017. Sparse coding with memristor networks. *Nature nanotechnology* (2017).

[99] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A flexible and fast software supported hardware logging approach for NVM. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM.

[100] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. 2017. PipeLayer: A pipelined ReRAM-based accelerator for deep learning. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 541–552.

[101] M. Song, J. Zhang, H. Chen, and T. Li. 2018. Towards Efficient Microarchitectural Design for Accelerating Unsupervised GAN-Based Deep Learning. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 66–77.

[102] Ilya Sutskever, Geoffrey E Hinton, and Graham W Taylor. 2009. The recurrent temporal restricted boltzmann machine. In *Advances in Neural Information Processing Systems*. 1601–1608.

[103] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *arXiv preprint arXiv:1703.09039* (2017).

[104] Toshiyuki Tanaka. 1998. Mean-field theory of Boltzmann machine learning. *Physical Review E* 58, 2 (1998), 2302.

[105] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. 2017. ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 13–26.

[106] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 91–104.

[107] Qian Wang, Yongtae Kim, and Peng Li. 2016. Neuromorphic processors with memristive synapses: Synaptic interface and architectural exploration. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 12, 4 (2016), 35.

[108] Ying Wang, Huawei Li, and Xiaowei Li. 2017. Real-Time Meets Approximate Computing: An Elastic CNN Inference Accelerator with Adaptive Trade-off between QoS and QoR. In *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 33.

[109] Yandan Wang, Wei Wen, Beiye Liu, Donald Chiarulli, and Hai Helen Li. 2017. Group Scissor: Scaling Neuromorphic Computing Design to Large Neural Networks. In *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 85.

[110] Ying Wang, Jie Xu, Yinhe Han, Huawei Li, and Xiaowei Li. 2016. DeepBurning: automatic generation of FPGA-based learning accelerators for the neural network family. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 1–6.

[111] Zhuo Wang, Robert Schapire, and Naveen Verma. 2014. Error-adaptive classifier boosting (EACB): Exploiting data-driven training for highly fault-tolerant hardware. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE, 3884–3888.

[112] Rainer Waser, Regina Dittmann, Georgi Staikov, and Kristof Szot. 2009. Redox-based resistive switching memories–nanoionic mechanisms, prospects, and challenges. *Advanced materials* 21, 25-26 (2009), 2632–2663.

[113] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems.. In *FAST*, Vol. 15. 167–181.

[114] Reza Yazdani, Albert Segura, Jose-Maria Arnau, and Antonio Gonzalez. 2016. An ultra low-power hardware accelerator for automatic speech recognition. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[115] Richard Yu. 2017. Panasonic and UMC Partner for 40nm ReRAM Process Platform. *UMC Press Release* (Feb 2017).

[116] Shimeng Yu, Zhiwei Li, Pai-Yu Chen, Huaqiang Wu, Bin Gao, Deli Wang, Wei Wu, and He Qian. 2016. Binary neural network with 16 Mb RRAM macro chip for classification and online training. In *Electron Devices Meeting (IEDM), 2016 IEEE International*. IEEE, 16–2.

[117] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15)*. ACM, New York, NY, USA, 161–170.

[118] Jintao Zhang, Zhuo Wang, and Naveen Verma. 2016. A machine-learning classifier implemented in a standard 6T SRAM array. In *VLSI Circuits (VLSI-Circuits), 2016 IEEE Symposium on*. IEEE, 1–2.

[119] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.