NEON: Enabling Efficient Support for Nonlinear Operations in Resistive RAM-based Neural Network Accelerators

Aditya Manglik†, Minessh Patel‡, Haiyu Mao†, Behzad Salami†, Jisung Park†, Lois Orosa†‡, and Onur Mutlu†
† ETH Zürich ‡ Galicia Supercomputing Center (CESGA)

Abstract—Resistive Random-Access Memory (RRAM) is well-suited to accelerate neural network (NN) workloads as RRAM-based Processing-in-Memory (PIM) architectures natively support highly-parallel multiply-accumulate (MAC) operations that form the backbone of most NN workloads. Unfortunately, NN workloads such as transformers require support for non-MAC operations (e.g., softmax) that RRAM cannot provide natively. Consequently, state-of-the-art works either integrate additional digital logic circuits to support the non-MAC operations or offload the non-MAC operations to CPU/GPU, resulting in significant performance and energy efficiency overheads.

In this work, we propose NEON, a novel compiler optimization to enable the end-to-end execution of the NN workload in RRAM. The key idea of NEON is to transform each non-MAC operation into a lightweight yet highly-accurate neural network. Utilizing neural networks to approximate the non-MAC operations provides two advantages: 1) We can exploit the key strength of RRAM, i.e., highly-parallel MAC operation, to flexibly and efficiently execute non-MAC operations in memory. 2) We can simplify RRAM’s microarchitecture by eliminating the additional digital logic circuits while reducing the data movement overheads. Acceleration of the non-MAC operations in memory enables NEON to achieve a 2.28x speedup compared to an idealized digital logic-based RRAM. We analyze the trade-offs associated with the transformation and demonstrate feasible use cases for NEON across different substrates.

I. INTRODUCTION

Data movement between memory and computation units inhibits the performance of memory-intensive workloads [1][2]. Processing-in-Memory (PIM) offers a potential solution to improve memory-intensive workloads’ performance and energy efficiency by reducing the data movement [3][4]. Among different PIM substrates, Resistive Random-Access Memory (RRAM) is under active investigation for accelerating neural network workloads [5][6]. RRAM’s subarrays are composed of resistive crossbars that offer in-memory Multiply-Accumulate (MAC) computation capability [14][17]. Prior work utilizes this capability to propose RRAM-based neural network accelerators and demonstrate orders of magnitude higher performance and energy efficiency compared to CPU, GPU, and ASICs [18][20].

We survey prior proposals for RRAM-based neural network inference accelerators [18][56] and observe a common design pattern: resistive crossbars execute the MAC operations, and additional computation structures execute the non-MAC operations in the workload. Resistive crossbars cannot execute the non-MAC operations in neural networks, for instance, when they are not multiplied by numbers. Consequently, additional digital logic circuits are integrated into the microarchitecture to support the non-MAC operations. The computation structures are implemented via different methodologies, including digital logic circuits (DLC), lookup tables (LUT), memristor-based logic synthesis (MLS), and operation offloading (OO). Each methodology offers different trade-offs with respect to the ability to support different operations (generalizability) and performance (see Figure 1).

We observe a fundamental restriction of the common design pattern followed by prior proposals: executing different neural network workloads in a single RRAM microarchitecture is difficult and inefficient as the system designer must integrate multiple computation structures in the microarchitecture to support different non-MAC operations in different workloads. Further, integrating memory and high-performance logic in a single chip presents manufacturing difficulties and might result in lower yields [10][58][61].

Our goal in this work is to enable efficient and generalizable support for different non-MAC (nonlinear) operations in RRAM-based neural network accelerators. To achieve this goal, we propose NEON, a novel hardware/software co-design methodology that leverages the resistive crossbars to enable in-memory support for non-linear operations. NEON is based on three key insights: (1) resistive crossbars offer in-memory MAC execution capability, (2) we can leverage the subarray-level parallelism in RRAM to execute multiple neural networks in parallel, and (3) neural networks can accurately emulate different operations via the universal...
function approximation theorem \[62–64\]. NEON leverages these insights to transform the unsupported nonlinear operations into lightweight neural networks and execute them in RRAM.

Our methodology comprises three components: (1) an automated transformation process for replacing the unsupported operations in the execution graph with neural networks referred to as NEON-Nets (Section IV-A), (2) a simplified RRAM microarchitecture supporting MAC operations (via resistive crossbars) and a single nonlinear operation implemented via DLC (Section IV-B), and (3) compiler and run-time modifications to effectively integrate the NEON-Net and the workload neural network, along with system optimizations. These three components enable a single low-cost RRAM microarchitecture to execute a wide range of neural networks flexibly.

We demonstrate the generalizability of NEON by generating and training NEON-Nets for multiple operations (Table III) collected from our evaluation benchmark composed of diverse neural networks (Section VII). Further, we explore the trade-off space between the performance and accuracy of the transformation process (Section VI-D1). We develop an automated tool to implement the transformation process and enable exploration of the NEON-Net design space. We will open source the source code of the transformation tool and the trained NEON-Nets in the final version of the manuscript.

This work makes the following contributions:

1) We propose a novel hardware/software co-design methodology for supporting different nonlinear operations in RRAM. NEON Enables a single RRAM microarchitecture to efficiently support in-memory execution for different neural networks.

2) We explore the trade-off space between high-performance and high-accuracy NEON-Nets. We will open-source the tool’s code to explore the trade-off space and the trained NEON-Nets in the final version of the manuscript.

3) NEON improves the end-to-end system performance by 2.28× and 1.4× over the DLC and LUT methodologies, respectively. NEON incurs 1.42× higher and 1.17× lower area utilization, 2.02× higher and 1.16× lower power dissipation overheads compared to the DLC and LUT methodologies, respectively.

II. BACKGROUND

We briefly introduce neural networks, followed by RRAM’s organization and operation mechanism for accelerating neural network inference. Next, we describe the universal function approximation theorem and prove the generalizability of the NEON methodology based on the theorem.

A. Neural Networks

Neural networks have emerged as an important class of workloads for applications such as self-driving cars \[65\] (using CNNs) and machine translation \[66–67\] (using transformers). The programmer defines the neural network’s structure before compilation, represented as a Directed Acyclic Graph (DAG). In the DAG, the vertices represent the operations, and the edges represent the data flow between the operations. The compiler can optimize the execution graph before deployment on the target hardware \[68–69\].

Convolutional Neural Networks (CNNs) are composed of convolutional and fully connected layers (composed of MAC operations). A convolutional layer comprises several kernels, and the size of each layer is a function of the kernel size and the number of input and output channels. The number of input channels is equal to the depth of the input feature maps (e.g., three for an input layer operating on RGB images). The number of output channels equals the depth of the output feature maps. Convolutional layers are followed by fully-connected layers with a softmax (nonlinear operation) at the output layer (Fig. 2).

Prior works exploit this observation to reduce the number of operations supported in the hardware by offloading softmax operations from the accelerator to the CPU \[48–49\].

Modern neural networks. Figure 3 shows the execution graph (DAG) for a modern neural network, the capsule network CapsNet \[71\]. We observe squash, softmax, and sigmoid operations in the middle of the critical path. It is difficult to offload these operations to the CPU without significant performance and energy consumption overheads stemming from off-chip data movement and frequent synchronization requirements \[6\] \[72\].

B. RRAM

Organization. Figure 4 shows an organizational overview of a reference RRAM microarchitecture designed for accelerating neural network inference. The high-level chip organization comprises multiple memory banks \[18\]. Each memory bank comprises multiple subarrays and sensing circuitry for performing the memory read and write operations. The one-transistor-one-resistor (1T1R) crossbar structure is used for the subarray microarchitecture due to higher cell selectivity and low leakage current \[73–75\]. The subarrays are connected via H-tree interconnects \[21\]. Each subarray can independently
execute different operations, resulting in a PIM substrate with massively parallel computation capabilities [76].

**Operation Mechanism.** We use Figure 4 to illustrate the Processing-in-Memory (PIM) operation mechanism of the resistive crossbars. The objective is to execute a vector-matrix multiplication (VMM) between the vector A and the matrix W. The matrix W is stored in memory (resistive crossbars), and the vector A is input via the wordlines (WL). First, each value \( W_{ij} \) of the matrix W is encoded as the conductance \( G \) (inverse of the cell’s resistance, \( R = \frac{1}{G} \) [77]). Next, the conductance values \( G_{ij} \) are written to the resistive crossbars based on device characteristics [78–80]. This completes the initialization of the crossbars before the execution (step 1 in Figure 4).

During execution, each input value in \( A \) is converted to an analog voltage value \( a_i \) through the digital-to-analog converters (DACs). The DACs drive the voltage on the respective wordline (step 2 in Figure 4). Third, the voltage difference drives a current \( I_i = a_i \cdot G_{ij} \) (inverse Ohm’s law; \( I = V/R \)). The current across the bitline is accumulated to yield the result of the MAC operation between the values stored in the cells on the bitline and the input voltage values (step 3 in Figure 4). Fourth, the accumulated current value is stored in the sample-and-hold circuits (S&H) and digitized via analog-to-digital converters (ADCs) (step 4 in Figure 4). Fifth, the digitized values are forwarded to the additional computation structures for executing the nonlinear operations (step 5 in Figure 4).

**Mapping Neural Network Workloads on RRAM.** To map the workload’s execution graph (DAG) on the resistive crossbars, each layer’s weight matrices are unrolled depth-wise and represented as a vertical column, referred to as a kernel. Every output channel in the layer is considered a single kernel, as shown in Fig. 5. Kernels may be split or duplicated across different RRAM subarrays [81,82] based on the size and the number of input and output channels. Kernel sizes can significantly influence the utilization ratio for resistive crossbars. As an illustrative example, kernel set \( N \) in Figure 5 utilizes only half of the available subarray capacity, resulting in a 0.5 utilization ratio. High utilization ratios (maximum 1.0) are desirable for higher energy efficiency [83].

**Precision.** To execute an n-bit fixed-point MAC operation between the mapped weights and the input value \( A \), 1-bit DACs inject n bits successively over n input cycles [21]. We use separate subarrays for mapping the positive and negative weight values. Multi-bit weight values are distributed across different columns due to the limited precision of a single memristor cell [84–85] (observe multiple columns for each kernel in Fig. 5). The final results are summed across different columns via Shift-and-Add (S&A) units.

**C. Universal Function Approximation (UFA) theorem**

The universal function approximation theorem (UFA) states that a feed-forward neural network with at least one hidden layer and a continuous bounded and non-constant activation function can approximate a function boundary with arbitrary precision [62,87–91]. Prior work has used the UFA theorem to approximate different operations in the C mathematical library with neural networks [92], and replace code regions in general-purpose workloads with neural networks [93].

### III. Trade-offs of Different Methodologies for Supporting Operations in RRAM

This section describes the results of our survey of prior proposals for accelerating neural networks in RRAM, followed by the trade-offs associated with each methodology in the survey.

**Survey.** We survey prior RRAM-based neural network inference accelerators and categorize them based on the methodology for supporting nonlinear operations. Table I reports the survey results for each methodology: Digital Logic Circuits (DLC), Lookup Tables (LUT), memristor-based logic synthesis (MLS), offloading operations (OO) to the host, and no discussion.

| Prior work (Count) | Methodology                          | Nonlinear Operations   |
|--------------------|--------------------------------------|------------------------|
| 18,21,25           | (16) Digital Logic Circuits (DLC)    | Sigmoid, ReLu          |
| 19,20,36,42        | (9) Lookup tables (LUT)              | Sigmoid, ReLu, LeakyReLu|
| 43,46              | (4) Memristor-based Logic Synthesis (MLS) | Softmax, Sigmoid tash |
| 47,48              | (2) Operation Offloading (OO)        | Squash, Softmax, Sigmoid, ReLu|
| 49,56              | (8) Nonlinear Operation support not discussed | -                       |

**TABLE I: Nonlinear operation survey in RRAM**

Next, we discuss the trade-offs for each methodology.

1^ReLu is a notable exception. The function is not continuously differentiable, but the gradient is defined for the discontinuity as a special case [86].
A. Digital Logic Circuits (DLC)

16 out of 39 works in our survey support nonlinear operations via the integration of DLC in the microarchitecture. DLCs offer low latency and low area requirements but suffer from two drawbacks: 1) Fixed-function circuits restrict the microarchitecture to a limited set of nonlinear operations. The operations must be known at design time and cannot be changed after the chip is manufactured. Flexible logic units (e.g., Chebyshev [94] and Taylor series-based function approximation [95,96]) offer marginally higher generalizability but result in significantly worse performance (due to the use of multiplications to calculate the output value) and a larger area requirement compared to fixed-function circuits [35]. Notably, only one prior work [35] in our survey uses flexible logic units. 2) Power dissipation is a key constraint for PIM substrates [97–99]. Static power dissipation restricts the number of DLCs that may be integrated in the microarchitecture. As an example, supporting 1152-dimensional softmax in CapsNet requires 576×EXP and DIV units that incur enormous static power overhead (19.76 W).

B. Lookup Tables (LUTs)

9 out of 39 works in our survey utilize LUTs to support nonlinear operations. LUTs offer flexibility but suffer from two drawbacks: 1) Large memory requirement restricts the scalability of LUTs. Scalability is required to support parallel nonlinear operations on the critical path. For instance, nonlinear operations in CapsNet require 23040 LUTs that consume 2880 MB, 424× larger than the memory requirement of CapsNet’s weights [100]. 2) The size of the RRAM subarray is generally restricted (e.g., 128 or 256-sized crossbar) to improve switching capability and minimize noise effects [75,101,102]. Consequently, large LUTs must be divided across several subarrays, leading to hierarchical memory accesses that incur significant latency penalties [103].

C. Memristor-Based Logic Synthesis (MLS)

MAGIC [104–107] proposes synthesizing primitive logic operations such as XOR and NOR using memristor cells. 4 out of 39 works leverage MLS to support nonlinear operations in RRAM. MLS is unable to offer either performance or generalizability due to the following three drawbacks: 1) Memristors exhibit asymmetric read/write performance, and write operations consume three orders of magnitude higher energy [78,108] than read operations. Each MAGIC-based logic gate requires 2-3 memory write operations per input that incurs significant energy consumption overheads. 2) The system designer must combine primitive operations such as NOR into complex digital operations. For instance, more than 20,000 memristor cells are needed to implement a simple 16-bit multiplier unit that must be further combined into complex operations such as EXP in softmax. 3) The synthesized logic is not reconfigurable and must be fixed at design time, resulting in a fixed-function microarchitecture.

D. Operation Offloading (OO)

Two prior works in our survey rely on the host for supporting nonlinear operations via offloading. Theoretically, OO offers high generalizability as the host is assumed as a general-purpose CPU. However, OO suffers from two drawbacks: 1) Frequent communication and synchronization due to multiple calls for nonlinear operations in the middle of the critical path (e.g., Figure [2] restrict the accelerator’s performance benefits. 2) Excessive off-chip data movement restricts the accelerator’s energy efficiency benefits. For instance, we evaluate the data movement costs between RRAM and CPU using the HyperTransport link used in a prior RRAM accelerator [21]. Data movement stemming from operation offloading requires 64.49× higher latency and 2.38× more energy than the latency and energy required for executing the MAC component of the target workload in RRAM.

IV. NEON Methodology

To enable efficient and generalizable support for nonlinear operations in RRAM-based neural network accelerators, we introduce NEON (NonLinEar emulatioN). NEON is a novel hardware/software co-design methodology to efficiently support different nonlinear operations in RRAM. Next, we describe the key insights, followed by an overview of NEON. Further, we detail each component and its implementation: the transformation process, RRAM microarchitecture, and the compiler support. Finally, we discuss a key feature of NEON-Nets, Operator Scalability.

Key Insights. We base our idea on three key insights:

1) Resistive crossbars offer in-memory MAC execution capabilities.
2) We can leverage the subarray-level parallelism in RRAM to execute multiple neural networks in parallel.
3) Neural networks can accurately emulate nonlinear operations via the universal function approximation theorem [62,64].

Based on these insights, NEON generates and trains neural networks to replace the unsupported nonlinear operations in the target neural network’s execution graph. NEON leverages the inherent strengths of RRAM subarrays (i.e., parallel and energy-efficient MAC execution) to execute the nonlinear operations within the subarray itself. For the rest of the paper, we refer to the target neural network as the workload and the generated neural networks as NEON-Nets. The workload is assumed to be pre-trained and the training dataset is available during the transformation.

Fig. 6: (a) displays the transformation process, and (b) displays the NEON microarchitecture.
Overview. The methodology has three components: (1) an automated transformation process for generating and training NEON-Nets (Section IV-A), (2) a simplified RRAM microarchitecture capable of executing the transformed workload (Section IV-B), and (3) compiler and run-time modifications to effectively integrate the NEON-Net and the workload neural network. Figure 6 shows an overview of the methodology. Next, we describe each component in detail.

A. Transformation Process

Motivation. The UFA theorem guarantees that a neural network can approximate a function boundary with arbitrary precision \[^62\] [\[^87\] [\[^88\]]. However, the theorem does not provide any information about the structure of the neural network or how to design it. The programmer is responsible for determining the NEON-Net’s structure. This is a non-trivial problem and requires significant expertise in designing and training neural networks [\[^110\]]. To overcome this limitation and make NEON generalizable across different nonlinear operations, we develop an automated transformation process to generate NEON-Nets using information from the workload’s execution graph. Figure 6a illustrates the transformation process comprising three steps:

1) Code segment delineation (1 in Fig. 6)
2) NEON-Net training dataset generation (2 in Fig. 6)
3) NEON-Net structure definition (3 in Fig. 6)

Next, we detail each step in the transformation process.

1) Code Segment Delineation

The first step is identifying the nonlinear operation’s code segment for replacement with the NEON-Net. Neural network frameworks such as PyTorch and TensorFlow [\[^111\] [\[^112\]] are widely used in industry and academia for programming and compiling neural networks. These frameworks expose a stable application programming interface (API) for operations used as common blocks across different neural networks. The operations are implemented via subroutines, e.g., softmax subroutine in PyTorch [\[^113\]]. We use the framework’s API to identify the subroutine as the code segment for transformation in the workload’s execution graph. We fix the granularity of replacement to subroutines.

2) NEON-Net Training Dataset Generation

The compiler generates the NEON-Net’s training dataset using the workload and the workload’s training dataset. To generate the dataset, the compiler executes inference over the workload using its training dataset and collects the input and output parameters for the target subroutine (x and y in Listing 1). The input parameter values (x) serve as the input feature values, and the output parameter values (y) serve as the ground truth in the NEON-Net training dataset.

3) NEON-Net Structure Definition

We choose fully-connected (FC) neural networks [\[^114\]] as the base structure for generating NEON-Nets as FC layers are composed of MAC operations that can be directly executed in resistive crossbars. It is possible to consider alternate neural network classes such as CNNs, RNNs, and autoencoders [\[^115\]] as base structures. However, each class requires co-designing the microarchitecture to ensure end-to-end execution capability and further exploration is left for future work.

A NEON-Net is a regression neural network composed of at least three FC layers: input, hidden, and output. The compiler uses information from the workload’s execution graph to determine the input and output layer sizes. Using the softmax subroutine in Listing 1 as an example, the function’s input (x) and output (y) parameter’s dimensions (d) are used to define the NEON-Net’s input and output layer sizes (d nodes in each layer).

```
def softmax(x): # x is a d-dimensional vector
    e_x = np.exp(x) # pointwise exponentiation
    y = e_x / e_x.sum() # pointwise division
    return y # y is a d-dimensional vector
```

Listing 1: Softmax subroutine in Python programming language (implementation based on the NumPy library [\[^116\]])

As the input and output layer sizes are fixed, the number and size of hidden layers primarily influence the NEON-Net’s performance and accuracy. We measure the accuracy via Mean-Square-Error (MSE) [\[^117\]] metric (denoted by $\varepsilon$). We picked MSE based on the highest end-to-end workload output accuracy across different metrics, including cosine similarity, mean absolute error, and MSE.

Determining the number and size of hidden layers. We develop an algorithm to determine the number and size of hidden layers based on the threshold accuracy indicated by the system designer. The algorithm takes the initial NEON-Net structure with input and output layers and iteratively adds hidden layers until the threshold accuracy is achieved. We constrain the maximum possible number of hidden layers to 100 to ensure that the training process completes in a finite period. The number of nodes in the hidden layer is determined based on the target RRAM microarchitecture’s crossbar size (e.g., 128 or 256). This allows NEON to maximize the utilization of the subarrays and encourages over-fitting (desirable as the function boundary is deterministic [\[^118\]])

```
Algorithm 1: NEON-Net hidden layer structure generation algorithm
1: Input: NEON-Net a single hidden layer, generated training dataset, $\varepsilon = 10^{-3}$, max_layers = 100, num_epochs=100, XBar_size=128;
2: initialization: Split dataset into train-validation, counter = 1;
3: while $||f(x) - \varepsilon|| > 0$ and counter $< max_{layers}$ do
4:     Train the NEON-Net for num_epochs using the training dataset;
5:     Determine NEON-Net’s MSE $f(x)$ on validation dataset;
6:     if $||f(x) - \varepsilon|| < 0$ then
7:         return trained NEON-Net structure and weights;
8:     else
9:         Add a hidden layer with XBar parameters;
10:        Re-initialize the NEON-Net weights;
11:        counter += 1;
12: end
B. Microarchitecture Design

NEON replaces the nonlinear operations with a NEON-Net. The NEON-Net is composed of MACs and a single nonlinear operation. The resistive crossbars directly support the MAC operations. However, the microarchitecture must support the nonlinear operation in the NEON-Nets. We integrate a single fixed-function unit to support the NEON-Net’s nonlinear operation. We pick DLC over LUT-based implementation for the implementation as DLCs offer higher performance compared to LUTs for implementing a single function.

Microarchitectural support for the NEON-Net’s nonlinear operation ensures that the NEON-Nets themselves are not recursively transformed. Each nonlinear operation in the execution graph is transformed only once, as successive approximations lead to an infinite recursion problem.

Determining the activation function for NEON-Net. The UFA theorem places restrictions on which functions may be used as activation functions. The function must exhibit the following mathematical properties: nonlinear, continuous, and finite output \([119]\). Based on these constraints, we evaluate different functions as potential candidates: sigmoid, tanh, and ReLu. We perform a grid search by training a fixed NEON-Net with different activation functions: ReLu (MSE: 0.0774), tanh (MSE: 0.0399), and sigmoid (MSE: 0.3036). We pick tanh as it offers the lowest MSE compared to other activation functions. We constrain all layers in the NEON-Net to use tanh as the common activation function. It might be possible to improve NEON-Net’s accuracy with more experiments. However, further exploration is left for future work.

C. Compiler Support

We describe how the compiler identifies subroutines for transformation, fine-tunes the post-transformation workload to recover the accuracy loss, and a run-time optimization to improve system stability.

Transformation Candidates. RRAM-supported subroutines (e.g., convolution) are scheduled directly, and RRAM-unsupported subroutines (e.g., softmax) are marked as transformation candidates. Any kernel that is directly supported on RRAM is not replaced. Each nonlinear operation is replaced with a separate NEON-Net trained on the compilation system.

Fine-tuning the workload after transformation. The transformation process leads to a slight accuracy loss in the workload. We can recover the accuracy loss by fine-tuning the workload as neural networks are inherently tolerant to approximate execution [120,121]. Fine-tuning is performed after replacing all nonlinear operations with NEON-Nets.

Mechanism: We assume the workload is pre-trained before the transformation, and the original training dataset is assumed to be available for fine-tuning. (1) We freeze (mark the layer as untrainable) all layers in the workload after replacing the unsupported operations with NEON-Nets. (2) We unfreeze (mark as trainable) one layer before and after the NEON-Net’s position in the workload. The NEON-Net’s layers are marked as frozen. (3) We resume training for the workload using the original training dataset. (4) In the forward propagation step, the unsupported subroutine’s output is derived from the corresponding NEON-Net’s output for the input values. (5) In the backward propagation step, the unsupported subroutine’s output is derived from the original function implementation (assumed as available in the compilation system, e.g., GPU).

Our fine-tuning mechanism is similar to training methods for low-precision neural networks [122,123].

Run-time. At run-time, the call to the unsupported subroutine is replaced with an equivalent function call to execute inference on the NEON-Net (co-executed in a dedicated subarray along with the workload). The function’s arguments are used as the input values for inference, and the NEON-Net’s output replaces the function’s return parameters on the call stack.

Next, we describe a run-time optimization to improve the system’s stability.

Input domain and output range constraints. Mathematically, a function may have an infinite input domain \((-\infty, \infty)\) that cannot be realized in practice. To overcome this problem, DLCs and LUTs exploit mathematical properties of functions such as the periodicity of trigonometric functions [124,125]. To overcome this problem for NEON, we constrain the function’s input domain and output range by measuring the expected distribution of values from the workload at run-time.

Neural networks often use batch normalization [126,127] after each layer to constrain the values to a normal distribution. For example, Fig. 7a illustrates the input value distribution \(x\) for softmax in CapsNet (trained over the CIFAR-10 dataset). We observe a normal distribution with 99.918% values less than 1.0 and 100% values less than 9.05. We leverage this observation to constrain the NEON-Net’s input domain to \([-1.8, 9.05]\). Similarly, Fig. 7b illustrates softmax’s output value distribution \(y\). We observe that 55% of the values are less than 0.1, and 100% are less than 0.87. Output values are constrained to \((0.0, 0.87]\).

The compiler extracts the expected distribution of input and output values from the NEON-Net training dataset. It parses the input feature and ground truth values to determine the minimum and maximum bounds for the input domain and output range, respectively. At run-time, if any value outside these constraints is encountered, it is rounded to the closest value within the bounds \((\bullet, \bullet)\) in Fig. 3. This is achieved by adding a simple rounding circuit (consisting of a comparator and a multiplexer).

D. Operator Scalability

Digital logic typically accepts unary/binary input values, for example, EXP (unary) and DIV (binary) operations. Increasing the number of input values (operators) incurs a linear increase in the area and power requirements for the digital logic circuits. Similarly, scaling an N-input LUT incurs an exponential increase in the memory requirements \(2^N\). In contrast to DLCs and LUTs, NEON-Nets offer a sublinear increase in the Energy-Delay Product (EDP) when scaling the number of operators. The number of input operators for a NEON-Net equals the number of nodes in the input layer. The number of input nodes is directly proportional to the number of wordline activations in the crossbar. Consequently, increasing the number...
### V. EXPERIMENTAL METHODOLOGY

This section describes the functions transformed by NEON, benchmark workloads, NEON-Net training hyper-parameters, and microarchitecture configurations.

#### A. Functions transformed by NEON

We evaluate the accuracy and performance of NEON-Nets for different nonlinear operations in Table III. The operations are selected based on our survey in Table III: softmax (four versions based on different dimensions), square-root, LeakyReLU, and squash. tanh is directly supported by the DLC in the microarchitecture, and sigmoid (σ) is indirectly supported via the following equation: \( \sigma(z) = \frac{1}{2}(\tanh(z/2) + 1) \). Any operation that is directly or indirectly supported in the microarchitecture is not replaced. We restrict our focus to continuous nonlinear operations as neural networks use continuous functions for activation (the function must be differentiable for using back-propagation to train the network [128]).

#### B. Benchmark Workloads

We detail the different neural networks in our benchmark in Table III along with the corresponding unsupported nonlinear operations transformed by NEON. We present the end-to-end system performance results in Section VII.

#### C. Training Hyper-parameters

**NEON-Net.** We use PyTorch [112] as the programming framework. The NEON-Nets are trained on a single GPU (NVIDIA RTX 2070) using the following hyper-parameters: loss function = Mean Square Error (MSE), loss optimization algorithm = Adam [129], batch size = 1024, learning rate = \( 10^{-4} \), weight decay = 0.0001, training epochs = 100, \( \varepsilon = 10^{-4} \).

Data distributions and pre-processing. NEON-Net training data is collected by executing inference for ten epochs using the workload and the original training dataset. The dataset is not normalized before training the NEON-Net to preserve the input value distribution. We use ten epochs for fine-tuning the workload after replacing all nonlinear operations with NEON-Nets and report the fine-tuning time in Section VI-C.

#### D. Microarchitecture configurations

We consider the following microarchitecture configurations for system evaluations:

- **Digital Logic Circuits (DLC).** The DLC configuration integrates fixed-function digital logic circuits necessary to support different neural networks in the benchmarks. The circuits’ area and power consumption values are taken from the respective manuscripts [130–132].

- **Look-Up Tables (LUT).** The LUT configuration integrates lookup tables necessary to support the nonlinear operations in different workloads. We use the resistive crossbars for storing the LUTs.

**NEON.** The NEON configuration integrates a single digital logic circuit to support the tanh operation. We set the threshold MSE as \( 10^{-4} \) as it is sufficient for 16-bit precision.

**MLS and OO.** Due to fundamental drawbacks associated with MLS and OO (orders of magnitude lower performance and energy efficiency), we do not compare NEON against these methodologies. However, we evaluate NEON’s performance against specialized accelerators that use these methodologies to support a particular workload in our benchmark (Section VII-H).

Area and power consumption values for all components are summarized in Table V. All values have been scaled for the 32 nm process node following the methodology in [133]. As prior work utilizes 16-bit fixed-point precision, all three configurations (DLC, LUT, and NEON) also work at 16-bit fixed-point precision (input, weight, and output values) for an apples-to-apples comparison. Although resistive crossbars support MUL operation, it requires significant modifications to the wordline drivers [134]. Further, multiplying two dynamic values via resistive crossbars requires writing one value to the cells. Write operations incur three orders of magnitude higher energy consumption penalty than read operations. To

### Table II: Benchmark neural networks

| Neural Network | Application Class | Dataset | Input size | Output size | Number of Parameters | Description | Nonlinear Operation (Dim) | Output Accuracy loss | Fine-tuning Time |
|----------------|-------------------|---------|------------|-------------|----------------------|-------------|--------------------------|---------------------|-----------------|
| CNN: VGG-16    | Image Recognition | ImageNet| 224x224x3  | 1000x1      | 138 million          | VGG-16 reference model | softmax (1000)     | -1.36%            | 19 min          |
| GRU-based RNN  | Speech Recognition| LibriSpeech | 128x1      | 128x1       | 594, 432             | Input vector size = 128 Hidden-state size=128 | sigmoid (1), softmax (2040) | 0.1%              | 6.5 min          |
| Capsule Network| Occluded Object Detection | CIFAR-10 | 32x32x3    | 16x10       | 6.81 million         | 3x dynamic routing iterations | squash (3), sigmoid (1) softmax (1152) | -1.8%             | 11 min           |
| Transformer    | Neural Machine Translation | WMT 2016 Translation Task | 128x512xdec | 128x512xdec | 60 million           | Self-Attention Heads = 8 Encoder/Decoder blocks = 6 | SQRT (1), sigmoid (1), softmax (64) | 0.87%             | 15 min           |

![Table II: Benchmark neural networks](image)
TABLE III: Different nonlinear operations with varying input parameter sizes replaced by NEON.

| Nonlinear Operation | Input Domain | Output Range | Accuracy (MSE) | Hidden layers | Training Time | Area (mm²) | Power (mW) |
|---------------------|--------------|--------------|----------------|---------------|---------------|-----------|------------|
| Softmax (64-dim)    | [-3.78, 7.5] | [0.001, 0.87] | $1.5 \times 10^{-8}$ | 1 | 12.5 min | 0.16 mm² | 288.96 mW |
| Softmax (1000-dim)  | [-0.08, 5.45] | [0.01, 0.89] | $2.62 \times 10^{-9}$ | 1 | 11.99 min | 1.66 mm² | 3058.16 mW |
| Softmax (1152-dim)  | [-1.08, 9.05] | [0.01, 0.87] | $2.5 \times 10^{-5}$ | 2 | 15.57 min | 1.99 mm² | 3660.16 mW |
| Softmax (2048-dim)  | [-0.08, 6.78] | [0.01, 0.79] | $3.4 \times 10^{-5}$ | 2 | 18.7 min | 3.45 mm² | 6357.12 mW |
| Square-root (SQRT)  | [-3.2, 4.5]  | [0.0, 2.12]  | $1.4 \times 10^{-4}$ | 2 | 18.64 min | 0.22 mm² | 409.36 mW |
| LeakyReLu ($\alpha = 0.1$) | [-7.8, 8.9]  | [-0.78, 8.9] | $5.5 \times 10^{-7}$ | 1 | 5.59 min | 0.12 mm² | 216.72 mW |
| Squash (8-dim)      | [-1.5, 2.13] | [-1.4, 2.02] | 0.0             | 1 | 5.73 min | 0.12 mm² | 216.72 mW |

TABLE IV: Microarchitectural component power and area values

| Component | Specification | Power (mW) | Area (mm²) |
|-----------|---------------|------------|------------|
| Subarray  | N/A           | 24.08 mW   | 13120 mm²  |
| Bank      | N/A           | 360.79 mW  | 484940 mm² |

Periphery Circuits

| Component | Specification | Power & Area Optimized |
|-----------|---------------|------------------------|
| Exponent  | N/A           | 7.424 mW, 3017 mm²     |
| Division/SQRT | N/A     | 25.88 mW, 23869 mm²    |
| Multiplier | N/A           | 4.7 mW, 236 mm²        |

C. Workload’s End-to-End Accuracy

Table III indicates the end-to-end accuracy loss for different workloads in the benchmark after fine-tuning. The average accuracy loss across the benchmark is -0.54%, indicating higher accuracy than the baseline. We attribute this observation to the tolerance of neural networks to noise injection [121]. Fine-tuning the network after replacement allows it to account for the noise. The slight performance improvement is attributed to the regularizing effects of noise injection during training [123][137].

Fine-tuning time. The average amount of fine-tuning time across the benchmark is 12.88 minutes (Table III). In contrast, training the workloads from scratch requires a few hours on average.

The squash NEON-Net obtains 0.0 MSE, implying perfect emulation of the nonlinear operation. To test the impact of 0.0 MSE NEON-Net on the workload’s (CapsNet) end-to-end accuracy, we replace only the squash operation in the workload and skip the fine-tuning step. We observe a 0% accuracy loss in the workload. We replace other nonlinear operations in CapsNet (softmax) and observe a 0.99% end-to-end accuracy loss. However, the workload recovers the accuracy loss after fine-tuning and improves upon the baseline accuracy by 1.8%.

Accuracy across dimensions. We plot the value distributions for each dimension in the squash NEON-Net and the ground truth in Figure 8 to understand the impact of 0.0 MSE. We observe that the distributions are identical across all eight dimensions. This observation corroborates the 0% accuracy loss observed in the CapsNet workload’s end-to-end accuracy when replacing the squash operation with the corresponding (perfect) NEON-Net.

Fine-tuning Time. The amount of time needed for fine-tuning each workload is as follows: VGG (19 minutes), RNN (6.5 minutes), CapsNet (11 minutes), and transformer (15 minutes). Fine-tuning requires significantly less time than training the workload from scratch (requiring multiple hours or days).

D. Exploring NEON-Net’s Trade-off Space

We describe the trade-offs associated with NEON.

1) Trade-offs between NEON-Net’s Size and Accuracy

We evaluate the performance variation of NEON with an increasing number of neurons per hidden layer. We evaluate the 1000-dimensional softmax obtained from NEON with one hidden layer composed of 128 nodes, with an MSE of $2.62 \times 10^{-9}$. Increasing the number of neurons in each hidden layer from 128 to 1000 results in a 0.97% increase in MSE. Further increasing the number of hidden layers from 1 to 3
Although the accuracy improvements are negligible, the NEON-Net (Section IV-C) insulates NEON against this problem to a certain extent. However, this is not a perfect solution, as a significant data distribution shift is a major problem for neural networks in general, and there is ongoing research in this direction [140].

### 3) Input data distribution shift

It is possible that the input data distribution shifts compared to the original distribution over which the NEON-Net was trained. The input domain and output range bounds (Section IV-D) insulate NEON against this problem to a certain extent. However, this is not a perfect solution, as a significant shift can lead to accuracy loss. In such cases, the NEON-Net must be retrained on the shifted input domain. Note that input data distribution shift is a major problem for neural networks in general, and there is ongoing research in this direction [140].

### VII. SYSTEM EVALUATION

#### A. End-to-end evaluations

We evaluate the end-to-end speedup, power, energy, and resource utilization for the three microarchitecture configurations: DLC, LUT, and NEON.

#### B. Speedup

Figure 10a shows the speedup normalized to the DLC configuration. NEON consistently provides speedup across the entire benchmark, with a geometric mean of 2.28× and 1.4× compared to DLC and LUT configurations, respectively. The performance improvement is attributed to the abstraction of long latency operations such as EXP and DIV (in softmax) with MAC and tanh (in NEON-Net). For instance, VGG obtains a modest speedup (1.06×) compared to the transformer (6.08×). The difference is due to a higher fraction of unsupported operations in transformers compared to VGG.

#### C. Area Utilization

Figure 10b compares the area utilization for all three configurations normalized to the DLC configuration. NEON increases the area utilization by 1.42× compared to the DLC configuration. This increase is attributed to the significantly larger resistive crossbars (0.026 mm²) compared to area-optimized digital logic circuits (0.016 mm² average area).

#### 1. Area Utilization for LUTs. LUTs require 1.17× more area compared to NEON. The difference is attributed to the value retrieval mechanisms: the neural network stores approximate values in the network’s weights (learned via back-propagation). In contrast, LUTs store precise values that require more area.

#### D. Power Dissipation

Figure 10c shows the power dissipation normalized to the DLC configuration. NEON requires 2.02× higher power compared to the DLC configuration and 1.16× lower power compared to the LUT configuration. The difference with respect to DLC is attributed to the difference in power dissipation of fixed-function circuits (10.28 mW on average) compared to resistive crossbars (24.08 mW on average). The power dissipation of resistive crossbars is dominated by the ADCs (16 mW). Lowering ADC power can help improve NEON’s power efficiency [141].

#### E. Energy Consumption

Figure 10d shows the energy consumption normalized to the DLC configuration. Despite a reduction in the execution time, higher area utilization and power dissipation lead to higher energy consumption (15.33× higher than DLC and 1.4× lower than LUTs). Workloads with a higher fraction of transformed operations (Transformer and CapsNet) report significantly higher energy consumption.
F. Operator Scaling

Figure 11 shows the energy-delay product (EDP) for all configurations normalized to DLC as we scale the number of input operators. We observe that digital logic offers lower EDP for a single input operator compared to NEON-Net. This observation is attributed to the higher cost of an entire subarray dedicated to execute the NEON-Net. However, as we increase the number of input operators, we observe that the EDP for digital logic scales linearly due to the fixed cost increments needed to support each new input value. In contrast, NEON-Net yields sub-linear EDP scaling by using more rows in the dedicated subarray. It is worth noting that NEON-Net shows an increase in the slope of the curve from 128 to 256 inputs. This observation is attributed to the addition of an additional subarray upon exceeding the capacity of the first one.

G. NEON-Net Initialization Energy Consumption

NEON-Net subarrays must be initialized via additional memory writes before deploying the system. We consider the initialization cost of NEON-Nets and compare it to a single inference call’s energy requirement for the corresponding workload. NEON-Net initialization consumes on average 3.54% of a single inference call’s energy consumption.

H. Comparison against Relevant Prior Work

ReTransformer [44] proposes a design for accelerating transformers in RRAM. NEON achieves 29.56% speedup over ReTransformer. Long et al. [35] propose a design for accelerating RNNs in RRAM using flexible logic circuits. NEON achieves 11.51× speedup and 14.58× energy reduction over [35]. Zhang et al. [34] propose a CORDIC-based microarchitecture supporting different nonlinear operations in RRAM. NEON obtains 87.09× higher performance over [34].

VIII. RELATED WORK

This section discusses prior efforts to support nonlinear operations in RRAM and neural network-based code approximation.

A. RRAM for Accelerating Neural Networks

Few prior works propose RRAM substrates to support different neural networks and machine learning workloads. Ankit et al. [20] propose PUMA, which relies on lookup tables for supporting non-native operations. We demonstrate that LUTs are area-inefficient compared to NEON. Other works [142, 143] look at designing RRAM-based neural network accelerators in the context of spiking neural networks (SNNs). However, SNNs require different hardware support structures (spike generator and accumulator in contrast to DACs and ADCs). Consequently, these works require significant hardware customization (e.g., FPGA-like interconnects [23]). NEON focuses on supporting different nonlinear operations in ADC-based RRAM accelerators. Zhang et al. [34] propose an RRAM substrate that supports different operations by relying on the CORDIC [144] algorithm for transcendental functions. We demonstrate that NEON is significantly faster than this proposal (Section VII-H). Further, PUMA [20] corroborates our hypothesis that a sufficiently accurate CORDIC unit is infeasible in practice due to a large area requirement and high implementation complexity. Huang et al. [47] propose a 3D-RRAM microarchitecture for accelerating capsule networks. However, their proposal offloads all nonlinear operations to the host. In contrast, NEON supports the nonlinear operations natively in RRAM.

B. Neural network-based code approximation

Esmaeilzadeh et al. [93] replace manually identified code sections in general-purpose workloads with a human-trained
neural network. In contrast, NEON automatically replaces unsupported nonlinear operations in neural network workloads to improve amenability on the target Processing-in-Memory substrate (RRAM). NEON executes the replacement of nonlinear functions into neural networks automatically using a reproducible process. We detail the differences as follows:

**Automatically identifying code segments for transformation.** Prior work \([145, 147]\) relies on the programmer to manually identify and annotate suitable code regions for replacement. NEON overcomes this limitation by leveraging information available at compile-time from the workload’s execution graph.

**Automated neural network definition and training.** Prior works \([148, 149]\) rely on the programmer’s expertise in machine learning to design the replacement neural network structure and train it for high accuracy. NEON overcomes these limitations by automating the network structure definition and training process.

### IX. EXTENSIONS AND FUTURE WORK

This section discusses the extensibility of NEON to different non-volatile memory-based PIM substrates, future directions for NEON, and manufacturing challenges for integrating DLC in memory microarchitectures.

#### A. Applicability of NEON to Different Non-Volatile Memory-based Processing-in-Memory Substrates

Emerging Non-Volatile Memory (NVM) technologies such as Phase Change Memory (PCM) \([150, 157]\) and Spin-Transfer Torque Magnetic RAM (STT-MRAM) \([158, 159]\) have gained attention as novel PIM substrates. These substrates perform operations on data values stored in the memory subarrays, using bitline-based computation mechanisms \([12, 160, 161]\) often organized similar to RRAM microarchitectures. Orthogonal to the underlying NVM technology choice, NEON provides a novel approach to support nonlinear operations in different substrates designed for accelerating neural network workloads. Although NEON is presented and evaluated in the context of RRAM in this paper, we believe that it is easily extensible to other substrates. Evaluating the feasibility and performance of NEON for different substrates is left for future work.

#### B. Future Directions for NEON

**Neural Architecture Search (NAS).** NEON enables generalizable support for nonlinear operations in RRAM while opening a new research problem: how to find high-accuracy and high-performance neural networks for emulating different nonlinear operations? NAS \([162, 168]\) offers one potential research direction towards this goal. NAS transforms the network design process into a search space exploration using a gradient method (e.g., back-propagation or reinforcement learning) \([169, 170]\). A loss function guides the search based on accuracy and performance metrics. NAS optimizes for two orthogonal problems in parallel – designing the network’s structure (including the size, number, and type of layer) and optimizing its trainable parameters (weights) \([171, 174]\). NAS-generated neural networks often significantly outperform manually designed networks in accuracy and performance \([175, 177]\). However, considering the complexity of realizing NAS in practice, this direction is left for future work.

**C. Manufacturing Challenges for Integrating Digital Logic in Memory Microarchitectures**

The manufacturing processes for integrating a large amount of digital logic on RRAM substrates remain an open challenge \([59, 178, 182]\). Memory microarchitectures are optimized for density in contrast with performance-optimized logic process \([60, 61, 183]\). Consequently, integrating general-purpose cores or FPGA units in memory substrates presents significant challenges. Further, programming such systems requires complex instructions that are generally not a part of memory ISAs \([184]\).

### X. CONCLUSION

We propose **NEON**, a novel hardware/software co-design methodology to efficiently support different nonlinear operations in RRAM. NEON enables RRAM to overcome the fundamental restrictions on supported operations by exploiting key strengths of the substrate. Further, it improves the end-to-end system performance compared to the DLC and LUT methodologies across different neural networks. We hope this work opens a new research direction in RRAM microarchitecture design to enable support for different operations without additional computation structures.

**REFERENCES**

[1] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. Processing data where it makes sense: Enabling in-memory computation. *Microprocessors and Microsystems*, 67:28–41, 2019.

[2] Amirali Boroumand, Saugata Ghose, Berkin Akin, Ravi Narayanaswami, Geraldo F. Oliveira, Xiaoyu Ma, Eric Shiu, and Onur Mutlu. Google neural network models for edge devices: Analyzing and mitigating machine learning inference bottlenecks. *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 159–172, 2021.

[3] A. Boroumand, Saugata Ghose, Youngsok Kim, R. Ausavarungnirun, E. Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, A. Knies, P. Ranganathan, and O. Mutlu. Google workloads for consumer devices: Mitigating data movement bottlenecks. *ASPLOS*, 2018.

[4] Amirali Boroumand, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development*, 63(6):3–1, 2019.

[5] Youngeun Kwon and Minsoo Rhu. Beyond the memory wall: A case for memory-centric hpc system for deep learning. *MICRO*, 2018.

[6] H-S Philip Wong, Heng-Yuan Lee, Shiming Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T Chen, and Ming-Jinn Tsai. Metal–oxide ram. *Proceedings of the IEEE*, 100(6):1951–1970, 2012.

[7] H. P. Wong, S. Kim, B. Lee, M. Caldwell, J. Liang, Yi Wu, R. Jeyasingh, and S. Yu. Recent progress of phase change memory (pcm) and resistive switching random access memory (ram). *2010 10th IEEE International Conference on Solid-State and Integrated Circuit Technology*, pages 1055–1060, 2010.

[8] Haifeng Li, Tony F Wu, Subhashis Mitra, and H-S Philip Wong. Resistive ram-centric computing: Design and modeling methodology. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(9):2263–2273, 2017.
IEEE Transactions on very large scale integration (VLSI) systems, 17(2):302–306, 2009.

[136] Pai-Yu Chen, Xiaochen Peng, and Shimeng Yu. Neurosim: A circuit-level macro model for benchmarking neuro-inspired architectures in online learning. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37(12):3067–3080, 2018.

[137] Hyeonwoo Noh, Tackgeun You, Jonghwan Mun, and Bohyung Han. Regularizing deep neural networks by noise: Its interpretation and optimization. Advances in Neural Information Processing Systems, 30, 2017.

[138] William S Noble. What is a support vector machine? Nature biotechnology, 24(12):1565–1567, 2006.

[139] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. R news, 2(3):18–22, 2002.

[140] Ohad Shamir. Distribution-specific hardness of learning neural networks. The Journal of Machine Learning Research, 19(1):1135–1163, 2018.

[141] Boris Murmann. The race for the extra decibel: A brief review of current adc performance trajectories. IEEE Transactions on Electronic Devices, 6(2):146–162, 2016.

[142] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. arXiv preprint arXiv:1611.01578, 2016.

[143] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 8697–8710, 2018.

[144] Bowen Baker, Oktir Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. ArXiv, abs/1611.02167, 2017.

[145] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. ArXiv, abs/1703.01041, 2017.

[146] Lingxi Xie and Alan Loddon Yuille. Genetic cnn. 2017 IEEE International Conference on Computer Vision (ICCV), pages 1388–1397, 2017.

[147] Yingwei Li, Xiaojie Jin, Jierui Mei, Xiaochao Lian, Linjie Yang, Cihang Xie, Qihang Yu, Yuyin Zhou, Song Bai, and Alan L Yuille. Neural architecture search for lightweight non-local networks. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 10297–10306, 2020.

[148] Weiren Jiang, Xinyi Zhang, Edwin H-M Sha, Lei Yang, Qingsong Zhuge, Yiyu Shi, and Jingtong Hu. Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search. In Proceedings of the 56th Annual Design Automation Conference 2019, pages 1–6, 2019.

[149] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search device. DATE, 2015.

[150] H. P. Wong, S. Raoux, S. Kim, J. Liang, J. Reifenberg, B. Rajendran, M. Asheghi, and K. Goodson. Phase change memory. Proceedings of the IEEE, 107(1):108–122, 2019.

[151] Xinyu Gong, Shiyu Chang, Yifan Jiang, and Zhangyang Wang. Autogan: Toward scalable neural architecture search for generative adversarial networks. In Proceedings of the IEEE/CVF International Conference on Computer Vision, pages 3224–3234, 2019.

[152] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. arXiv preprint arXiv:1812.00332, 2018.

[153] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. In International Conference on Machine Learning, pages 7105–7114. PMLR, 2019.

[154] Niv Nayman, Asaf Noy, Tal Ridnik, Itamar Friedman, Rong Jin, and Lihi Zelnik. Xnas: Neural architecture search with expert advice. Advances in neural information processing systems, 32, 2019.

[155] Chenzhi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In Proceedings of the European conference on computer vision (ECCV), pages 19–34, 2018.

[156] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 2820–2828, 2019.
Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yinning Wu, Yuanlong Tian, Peter Vajda, Yangqiu Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 10734–10742, 2019.

Alvin Wan, Xiaoliang Dai, Peizhao Zhang, Zijian He, Yuanlong Tian, Saining Xie, Bichen Wu, Matthew Yu, Tao Xu, Kan Chen, et al. Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 12965–12974, 2020.

H-S Philip Wong and Sayeef Salahuddin. Memory leads the way to better computing. Nature nanotechnology, 2015.

Ogun Turkyilmaz, Santhosh Onkaraiah, Marina Reyboz, Fabien Clermidy, Costin Anghel, Jean-Michel Portal, and Marc Bocquet. Rram-based fpga for “normally off, instantly on” applications. In Proceedings of the 2012 IEEE/ACM International Symposium on Nanoscale Architectures, 2012.

Xifan Tang, Edouard Giacomin, Patsy Cadareanu, Ganesh Gore, and Pierre-Emmanuel Gaillardon. A rram-based fpga for energy-efficient edge computing. In DATE, 2020.

Sansiri Tanachutiwat, Ming Liu, and Wei Wang. Fpga based on integration of cmos and rram. IEEE Transactions on VLSI Systems, 2010.

Meenatchi Jagasivamani, Candace Walden, Devesh Singh, Luyi Kang, Mehdi Asnaashari, Sylvain Dubois, Bruce Jacob, and Donald Yeung. Tileable monolithic rram memory design. In 2020 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS), pages 1–3, 2020.

Jaeduk Lee, Jaehoon Jang, J. Lim, Y. Shin, K. Lee, and E. Jung. A new ruler on the storage market: 3d-nand flash for high-density memory and its technology evolutions and challenges on the future. 2016 IEEE International Electron Devices Meeting (IEDM), pages 11.2.1–11.2.4, 2016.

Alasdair Armstrong, Thomas Bauereiß, B. Campbell, A. Reid, Kathryn E. Gray, Robert M. Norton, P. Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Phur, I. Stark, N. Krishnaswami, and Peter Sewell. Isa semantics for armv8-a, risc-v, and cheri-mips. Proceedings of the ACM on Programming Languages, 3:1 – 31, 2019.