NL-DPE: An Analog In-memory Non-Linear Dot Product Engine for Efficient CNN and LLM Inference

Lei Zhao, Luca Buonanno, Archit Gajjar, *Member, IEEE,*, John Moon, Aishwarya Natarajan, *Member, IEEE,*, Sergey Serebryakov, Ron M. Roth, *Life Fellow, IEEE,*, Xia Sheng, Youtao Zhang, *Member, IEEE,*, Paolo Faraboschi, *Fellow, IEEE,*, Jim Ignowski, *Senior Member, IEEE,*, and Giacomo Pedretti, *Member, IEEE,*

***Abstract*—Resistive Random Access Memory (RRAM) based in-memory computing (IMC) accelerators offer significant performance and energy advantages for deep neural networks (DNNs), but face three major limitations: (1) they support only *static* dot-product operations and cannot accelerate arbitrary non-linear functions or data-dependent multiplications essential to modern LLMs; (2) they demand large, power-hungry analog-to-digital converter (ADC) circuits; and (3) mapping model weights to device conductance introduces errors from cell nonidealities. These challenges hinder scalable and accurate IMC acceleration as models grow.**

We propose NL-DPE, a Non-Linear Dot Product Engine that overcomes these barriers. NL-DPE augments crosspoint arrays with RRAM-based Analog Content Addressable Memory (ACAM) to execute arbitrary non-linear functions and data-dependent matrix multiplications in the analog domain by transforming them into decision trees, fully eliminating ADCs. To address device noise, NL-DPE uses software-based Noise Aware Fine-tuning (NAF), requiring no in-device calibration. Experiments show that NL-DPE delivers 28× energy efficiency and 249× speedup over a GPU baseline, and 22× energy efficiency and 245× speedup over existing IMC accelerators, while maintaining high accuracy.

***Index Terms*—in-memory computing, analog acceleration, neural network, noise, fine-tuning, rram.**

I. Introduction

Artificial Intelligence (AI) models have significantly grown in size and complexity, leading to frequent data movement between memory and processing units. The development of in-memory computing (IMC) accelerators, particularly those based on resistive memories, such as RRAM [1], Phase Change Memories (PCM) [2], and FeFET [3], stand out as one of the most promising solutions due to their potential for high energy efficiency and scalability. IMC accelerators often organize resistive memory cells in a crossbar to construct dot product engines (DPEs), which accelerate vector-matrix multiplications (VMMs) in the analog domain, achieving low energy consumption and high parallelism [4], [5].

However, modern AI models, in particular, the Transformer models that are widely adopted in large language models (LLMs), consist of more than just static VMM operations. That is, they often require data-dependent matrix multiplications (DMMul) and non-linear operations, such as activations, and Softmax. Most IMC accelerators [4]–[6] perform activations and DMMul using on-chip digital circuits or the host processor, which demand the conversion of the analog outputs from DPEs into digital signals via analog-to-digital converters (ADCs) before computation. Unfortunately, ADCs are both energy- and area-inefficient, consuming more than 30% of the chip area and accounting for over 50% of the total power consumption [4], [5]. Recently proposed IMC designs [7], [8] strive to eliminate ADCs but work only if the AI model contains ReLU-like non-linear functions, and thus lack the flexibility to adapt to evolving AI models.

To understand the inefficiency of RRAM-based IMC accelerators for evolving AI models, we analyze the energy consumption breakdown of ISAAC [4] and RAELLA [6]. Section V elaborates on the experiment settings. For non-linear activation and DMMul operations, we optimize using a state-of-the-art Vector Functional Units (VFU) [5], [9] and simulate using CiMLoop [10] [1]. Fig. 1 shows that while ADCs still account for the majority of the energy consumption for CNN models, VFUs dominate the energy consumption in the attention models, i.e., BERT [14]. In summary, it is important to address the non-linear functions and DMMul in modern models.

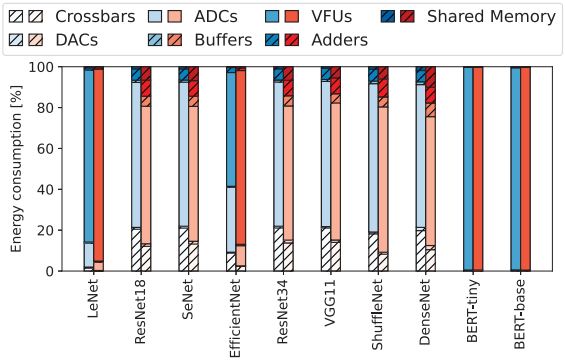


Fig. 1: The energy breakdown of RRAM IMC accelerators, namely ISAAC [4] (blue) and RAELLA [6] (red). We focus on optimizing ADC and VFU, i.e., the bars with solid colors.

Another challenge for IMC-based AI accelerators is that resistive memory cells, such as RRAMs, suffer from conductance noises, i.e., there is a non-negligible difference between the desired cell conductance and the cell conductance that was actually programmed or can be read. While studies have shown that AI models exhibit inherent tolerance to var-

Manuscript received July 28th, 2025.

Lei Zhao, Luca Buonanno, Archit Gajjar, John Moon, Aishwarya Natarajan, Sergey Serebryakov, Xia Sheng, Paolo Faraboschi, Jim Ignowski, and Giacomo Pedretti are with the Artificial Intelligence Research Lab (AIRL), Hewlett Packard Labs, USA (e-mail: lei.zhao@hpe.com, giacomo.pedretti@hpe.com).

Ronny Roth is with Technion - Israel Institute of Technology.

Youtao Zhang is with the University of Pittsburgh, Pittsburgh, PA, USA.

[1]Alternatively, we may optimize to implement DMMul by frequently reprogramming the DPEs [11], [12]. Given the limited endurance of RRAM cells (e.g., $< 10^9$ cycles [13]), this approach is suboptimal and less preferred.

ious errors, including conductance noises, real hardware [1], [15], [16] demonstrate that resistance noises play a crucial role in determining the model's accuracy. Without careful handling of various noise sources, model accuracy may degrade to unacceptable levels [16]. Existing approaches for mitigating conductance noises [1] are limited to VMM operations and thus not usable for non-VMMs in the analog domain.

In this work, we propose NL-DPE, a noise-resilient non-linear dot product engine, to achieve scalable acceleration for modern AI models. In particular, we design a RRAM-based Analog Content Address Memory (ACAM) to act as a decision-tree (DT) engine, enabling the acceleration of arbitrary non-linear functions in the analog domain. Our contributions are as follows.

- We enable the acceleration of both VMM and non-VMM operations in the analog domain. In particular, DMMul and non-linear function operations are conducted in the ACAM engines. By completely eliminating the costly ADC circuits, NL-DPE strives to achieve scalable performance and energy consumption improvements for modern AI models.
- We propose a novel RRAM-based ACAM design that enables the computation of arbitrary non-linear functions in the analog domain, by predicting them with DTs. In particular, the modular per-bit DT architecture provides sufficient accuracy to meet the diverse demands of modern AI models. We elaborate on the decomposition of complicated functions, e.g., Softmax and DMMuls, into a sequence of logarithmic and exponential operations, which are then efficiently implemented in ACAM.
- We achieve error-resilient deployment of large AI models using a software-based, per-DT Noise Aware Fine-tuning (NAF) approach. From a pre-trained model, we extract DMMul and non-linear functions, convert them into DTs, and fine-tune each DT independently with simulated RRAM noise. The resulting model can be directly deployed to NL-DPE arrays without any post-deployment calibration.
  We measure noise from fabricated RRAM devices [15][2], and develop noise models for both crossbar arrays and ACAM units that incorporate various RRAM non-idealities to facilitate NAF. We have open-sourced the NAF framework at: https://github.com/HewlettPackard/nl-dpe.
- We evaluate NL-DPE using the CiMLoop simulator [10] and an NVIDIA H100 GPU. NL-DPE delivers 28× higher energy efficiency and 249× speedup over the GPU baseline at the same quantization bit width, and 22× higher energy efficiency and 245× speedup compared to existing IMC accelerators. A detailed design-space exploration further highlights the advantages of our architecture.

II. Background

A. In-Memory Dot-Product Engine (DPE)

Fig. 2(a) illustrates a VMM operation $\mathbf{y} = \mathbf{xW}$ mapped onto a 2×3 RRAM crossbar. RRAM cells, typically in a 1T1R structure, store weights w_{ij} as conductance values, while

[2]Although the evaluation is based on Ta-Ox RRAM devices, the designs apply to other emerging resistive memory technologies, such as PCM and FeFETs.

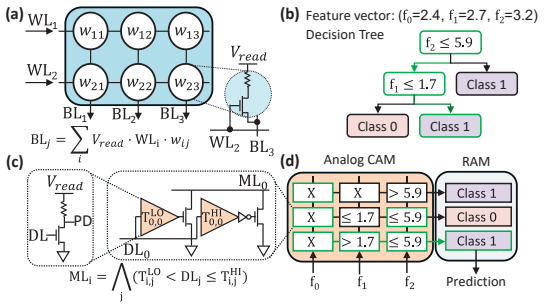


Fig. 2: **(a)** Computing the VMM of in a crossbar array with 1T1R RRAM cells. **(b)** Example of a trained decision tree. **(c)** ACAM cell using 1T1R **(d)** Mapping the decision tree of (a) onto ACAM and performing inference.

inputs x_i are converted into sliced binary voltages on the word lines [4], [6]. By Kirchhoff's and Ohm's laws, the bit-line currents represent the dot products scaled by the read voltage V_{read}, with column-wise accumulation completed via shift-and-add [4]. This enables a full analog VMM in one step. While the figure shows a small 2×3 example, practical designs often use much larger arrays (e.g., 512×512 [6]) for high parallelism.

The analog outputs of the crossbar must be digitized for downstream operations such as activation functions, typically requiring area- and power-hungry ADCs. Although pulse-width modulation can replace ADCs [17], it introduces high latency due to unary encoding and still relies on digital activations. A recent work also proposed a crossbar-based ramp ADC for recurrent neural networks [18].

B. Decision Tree

A decision tree (DT) is a simple yet effective model for classification and regression and can serve as a universal function approximator. It consists of internal nodes that apply conditional branches on trainable feature values and leaf nodes that produce the final prediction (e.g., a class label).

Fig. 2(b) shows a DT trained for binary classification. Each sample has three features (f_0, f_1, f_2) and a label 0 or 1. Internal nodes compare a selected feature to a threshold, and leaf nodes store the predicted label.

To classify an input, traversal starts at the root, following internal-node predicates to the left or right subtree until reaching a leaf containing the predicted label. In Fig. 2(b), the highlighted path shows the input $(f_0 = 2.4, f_1 = 2.7, f_2 = 3.2)$ reaching a leaf that predicts label 1.

C. Analog Content Addressable Memory

RRAM and FeFET have been used to implement ACAMs for accelerating tree-based ML algorithms in the analog domain [19]. Unlike digital CAMs, which compare single bits, an ACAM cell compares an analog input against a stored range. In Fig. 2(c), each cell uses two RRAMs to store a lower bound, upper bound, or a range, programmed as conductances ($T^{lo}ij$ and $T^{hi}ij$). A wildcard cell "X" stores the maximum range for a always-TRUE comparison. Analog comparators use a 1T1R voltage-divider structure, similar to crossbars, driving pull-down logic. ACAM designs [20] store the lower bound with 1T1R and require an inverter for the upper bound.

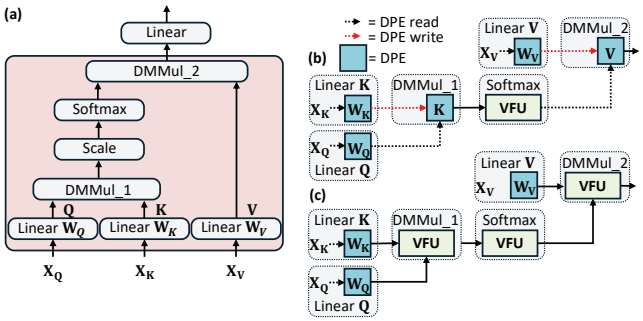


Fig. 3: **(a)** Attention mechanism and its mapping into conventional DPE architectures either by **(b)** reprogramming the DPE arrays or **(c)** using a VFU for DMMul.

Before searching, the match line (ML) is precharged high. If the input DLj is below T_{ij}^{lo} or above T_{ij}^{hi}, the corresponding pull-down or inverter drives ML$_i$ low. Thus, ML$_i$ stays high only when DL$_j$ falls within the stored range.

To map a DT onto an ACAM array, each leaf node is traced back to the root, storing feature thresholds along the path in a row of ACAM cells. In Fig. 2(d), the last row corresponds to the highlighted path in Fig. 2(b). Repeated features along a path are merged into a single range, and wildcard cells ("X") cover features not on the path. During inference, a row's match line is activated if all input features fall within the stored ranges, retrieving the predicted class from adjacent RAM. ACAMs thus accept analog inputs but produce digital match outputs.

D. Attention Mechanism

The attention mechanism is the core operation of Transformer models. As shown in Fig. 3, it takes three token sequences—query ($\mathbf{X_Q}$), key ($\mathbf{X_K}$), and value ($\mathbf{X_V}$))—which are each transformed by linear layers ($\mathbf{W_Q}$, $\mathbf{W_K}$, $\mathbf{W_V}$) into matrices $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$. Attention is then computed as:

$$Attention(\mathbf{K}, \mathbf{Q}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q} \cdot \mathbf{K}^T}{\sqrt{d_k}}\right) \cdot \mathbf{V} \tag{1}$$

where d_k represents the dimensionality of each query and key vector. The Softmax function is defined as:

$$\text{Softmax}(y_i) = \frac{e^{y_i}}{\sum_{j=1}^{L} e^{y_j}} \tag{2}$$

Here, y_i is an element in the input vector $\mathbf{Y} = [y_1, ..., y_L]$ for Softmax, and L denotes the vector length. The output of the attention mechanism is then passed to another Linear layer.

While the Linear layers can be efficiently implemented on RRAM crossbars, there are two design challenges:

- DMMul computation. Given $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ are dynamically generated for each new input, accelerating DMMul with conventional DPE [11], [12] demands programming RRAM crossbars frequently, as shown in Fig. 3(b), which may significantly degrade accelerator's energy efficiency and RRAM cells' endurance. An alternative is to compute DMMuls in digital circuits using VFUs [5], as shown in Fig. 3(c), which becomes the dominant energy consumer as reported in Fig. 1.
- Softmax computation. As shown in Equation 2, Softmax involves exponentiation and division, both of which are difficult to implement using RRAM crossbars. While Softmax computation is typically performed using digital units, it leads to degraded energy efficiency as reported in Fig. 1.

Notably, since both DMMul and Linear layers only contain linear computations, the scaling (i.e., division by $\sqrt{d_k}$) can be fused into the weights (i.e., $\mathbf{W_Q}$, and $\mathbf{W_K}$) during inference.

III. Non-Linear Dot-Product Engine

A. Overview

Fig. 4(a) illustrates our key innovation by comparing the implementations of a snippet of a DNN using a conventional DPE architecture [4] and the proposed NL-DPE, respectively. The snippet consists of three layers: two VMM-based layers (e.g., Convolutional layer) and one non-VMM layer (e.g., activation layer). The conventional DPE performs non-VMM operations digitally, and thus requires ADCs and DACs to convert analog output from the crossbar before non-VMM operations and convert non-VMM output to analog signals before the crossbar of the next layer, respectively. By contrast, the proposed NL-DPE performs non-VMM operations directly in the analog domain using ACAMs, eliminating the need for ADCs after the crossbar, as the ACAMs can process analog inputs directly and output a digital value. While DACs are still necessary before subsequent VMM layers, they consume significantly less energy and area than ADCs [4]. For VMM layers not followed by non-VMM operations, the ACAMs compute an identity function, effectively acting as ADC substitutes and enabling the entire system to operate without ADCs.

We next present an overview of the NL-DPE architecture in Fig. 4(b)-(e). An NL-DPE accelerator chip consists of multiple tiles organized as a 2D mesh with tiles connected using on-chip routers, orchestrated by a controller. This is similar to the popular tile-core accelerator hierarchy [5], which communicates with the host system via a PCIe interface.

Each tile includes a shared memory for storing the input and output of the computations assigned to it, along with a set of adders and associated registers to accumulate partial results from the cores shown in Fig. 4(d). Each core contains an NL-DPE composed of multiple crossbars and ACAM units. One ACAM unit computes the output from the same column of all the crossbars in the same core, while one ACAM unit contains multiple ACAM arrays, with each array computing one bit of the output, as shown in Fig. 4(e). XOR gates are used in the ACAM unit to decode its output for subsequent computations.

B. Computation Modes

Given that NL-DPE computes VMM operations in crossbars and non-linear functions in ACAMs, it can be configured into one of the following three modes:

(1) Dual-compute mode. This is the typical configuration as a DNN accelerator. i.e., the crossbars are configured to compute VMMs while the ACAMs are configured to compute non-linear functions. The crossbars store the weights of the Convolutional or Linear layers and compute the dot-product between the weights and the input, while the ACAMs store the thresholds of the DTs, which are trained to approximate the target non-linear function (Section III-C). A similar mechanism can be used to compute the attention mechanism by

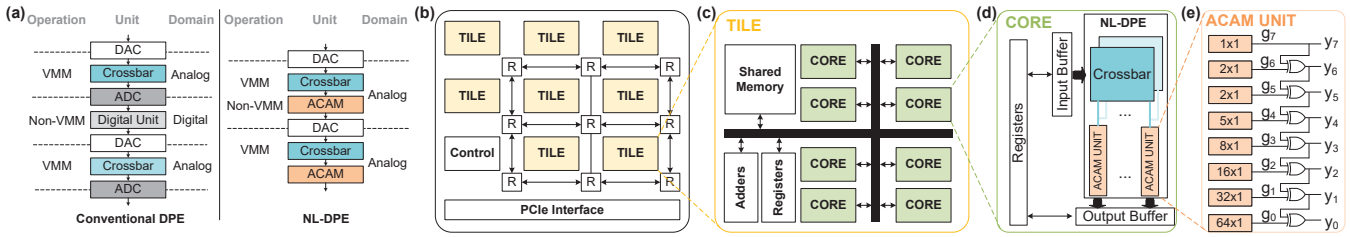


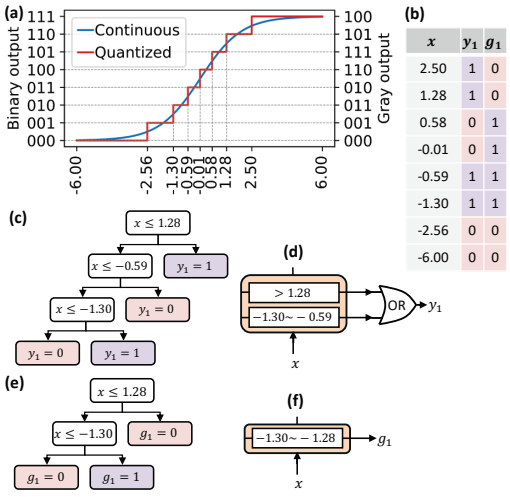
Fig. 4: **(a)** Conceptual representation of conventional DPE (left) and NL-DPE (right), which replaces ADC and digital computing logic. **(b)** Schematic of the accelerator tiled architecture. **(c)** Tile schematic. **(d)** Core schematic. **(e)** ACAM unit schematic.

converting the Linear layer outputs in the logarithmic domain (Section III-D).

(2) Crossbar-only mode. In this configuration, NL-DPE only needs to perform a VMM within the crossbars, e.g., in the case when a Convolutional layer is followed by another Convolutional layer rather than a non-linear activation layer. We then configure ACAMs to compute the identity function, which effectively acts as conventional ADCs, converting the analog output from the crossbars into digital signals.

(3) ACAM-only mode. In this configuration, NL-DPE only exploits the ACAMs to compute non-linear operations. Configuring a subset of NL-DPE cores in this mode enables the computation of more complicated non-linear functions. In this configuration, the crossbars store identity matrices such that their inputs are directly passed to the ACAMs for non-linear operations. That is, NL-DPE functions as a vector ALU, which aims to replace VFUs.

By altering the contents stored in the RRAM cells of crossbars and ACAMs, NL-DPE can flexibly switch all or a subset of cores to different modes. No hardware modification is needed after deployment. Thus, the NL-DPE itself is agnostic to the selected mode.



x	y_1	g_1
2.50	1	0
1.28	1	0
0.58	0	1
-0.01	0	1
-0.59	1	1
-1.30	1	1
-2.56	0	0
-6.00	0	0

Fig. 5: **(a)** The Sigmoid function with output being quantized to 3 bits in unsigned binary (y on the left Y axis) and in Gray code format (g on the right Y axis). **(b)** Training dataset for predicting y_1 and g_1. **(c)** Trained DT to predict y_1. **(d)** Mapping (c) into an ACAM. **(e)** Trained DT to predict g_1. **(f)** Mapping of (e) into an ACAM.

C. Computing simple non-linear functions with ACAMs

We next use Sigmoid as an example to demonstrate how to configure ACAMs to compute a non-linear single-variable function. Fig. 5(a) shows the output of the Sigmoid function when it is quantized to three bits. The blue and red lines represent the Sigmoid function in continuous and quantized versions, respectively. In this example, we assume that 0 and 1 are quantized as 000_b and 111_b, respectively, with the remaining 6 values evenly distributed between 0 and 1, as shown by the left Y-axis in the figure. Note that this simple quantization scheme is only for illustration purposes, an arbitrary quantization scheme can be used.

We convert the computation of each output bit y_i ($0 \leq i \leq 2$) to a binary classification problem and predict its value using one DT. For the example, the y_1 column in the table in Fig. 5(b) tracks its value change at different thresholds, e.g., (x=1.28, y_1=1) and (x=0.58, y_1=0) indicate y_1 being 1 if x>1.28; and 0 if x≤1.28. The complete DT for y_1 is shown in Fig. 5(c). Unlike traditional DTs that predict unseen data, our binary-bit DTs are specifically designed to memorize the exact thresholds in the training dataset, intentionally overfitting the data for precise computation.

Fig. 5(d) shows how the resulting DT is mapped onto an ACAM array. Since the tree has only one feature, the ACAM array logically contains a single column, with the number of rows corresponding to the number of leaf nodes where y_1=1 (two rows in this example). All match lines in the ACAM array are connected to an OR gate, which outputs the computed result for y_1. In practice, multiple ACAM cells can be arranged in a row to perform the OR operation through a pass transistor logic, significantly reducing the logic gate overhead. If the input falls within any of the ranges stored in the two ACAM cells, y_1 is set to 1. Unlike the Look-Up Table (LUT) approach, where an attached memory is needed to retrieve the output (see the RAM after the ACAM array in Fig. 2(d)) [8], the ACAM's output directly represents the bit value, eliminating the need for additional memory access. This method can be applied to compute the other bits (y_0 and y_2), requiring a total of three ACAM arrays to calculate the 3-bit Sigmoid function in this example.

Encoding Optimization. From Fig. 5(b) and (d), we can see that the number of rows in each ACAM array for computing a single output bit is determined by the number of consecutive 1s in the training dataset, i.e., how frequently the output bit toggles between 0 and 1 as the input value increases. Since activation functions are typically monotonic, the less significant output bits tend to toggle more frequently between

0 and 1. For an n-bit activation function, there are 2^n distinct output levels, meaning the worst-case scenario for the number of ACAM rows is 2^{n-1}.

To reduce ACAM array size, we propose an encoding scheme based on Gray code to minimize the bit-toggling rate. Originally designed to minimize error probabilities in analog-to-digital signal conversion, Gray code ensures that only one-bit changes occur at a time between consecutive numbers. For example, considering 3 bits, when the decimal value increases from 3 to 4, there are three-bit flips for binary format ($011_b \rightarrow 100_b$), while the Gray code has only one-bit flip ($010_g \rightarrow 110_g$).

In Fig. 5(a) (right Y-axis), we show the output of the example in Gray code. Instead of computing y_1, we compute g_1 instead. Fig. 5(e) and (f) show the trained DT and its corresponding ACAM mapping. The number of ACAM rows is reduced from 2 to 1.

Table I compares the required ACAM sizes for various 8-bit activation functions predicted with DTs using unsigned binary (B) or Gray coding (G), demonstrating that the latter reduces the ACAM size for all output bits (except the MSB) by approximately half. The last row in the table shows a negligible Mean Squared Error (MSE) compared with digital computation, indicating that the basic function computations incur negligible accuracy loss.

TABLE I: Number of ACAM rows for various 8-bit functions.

	Sigmoid	Tanh	SiLU	GELU	ReLU	Identity	log	exp
Enc	B/G	B/G	B/G	B/G	B/G	B/G	B/G	B/G
bit$_7$	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1
bit$_6$	2/1	2/1	2/1	2/1	2/1	2/1	3/2	2/1
bit$_5$	4/2	4/2	4/2	4/2	4/2	4/2	5/2	4/2
bit$_4$	8/4	8/4	8/4	8/4	8/4	8/4	8/5	8/4
bit$_3$	16/8	16/8	16/8	16/8	16/8	16/8	16/8	16/8
bit$_2$	32/16	32/16	32/16	32/16	32/16	32/16	32/16	32/16
bit$_1$	64/32	64/32	64/32	64/32	64/32	64/32	64/32	64/32
bit$_0$	121/64	113/64	101/64	112/64	121/64	1/64	97/64	108/64
Total	248/128	240/128	228/128	239/128	248/128	128/128	226/130	235/128
MSE	6.15e-08	6.15e-07	7.01e-23	2.46e-06	2.05e-06	1.29e-05	3.03e-06	3.72e-06

Gray code, while reducing the number of bit flips, cannot be directly involved in binary computation. To enable subsequent computations, we convert the Gray code outputs from the ACAMs back to their binary form. This is achieved as follows.

$$y_i = \begin{cases} g_i & i = n-1 \\ XOR(g_{n-1}, g_{n-2}, ..., g_i) & i < n-1 \end{cases}$$

where y_i and g_i are the ith bit in the binary and Gray code format. n is the number of output bits.

ACAM unit implementation. The detailed structure of an ACAM unit is illustrated in Fig. 4(e). We selected a bit-width of 8 after careful design space exploration, as shown in Section VI-A2, thus each ACAM unit consists of 8 ACAM arrays, with each array responsible for computing one output bit. The sizes of these 8 ACAM arrays are determined based on the largest requirement across all functions profiled in our tested AI models. (Table I). Specifically, starting from the most significant output bit, the sizes of the 8 ACAM arrays are 1, 2, 2, 5, 8, 16, 32, and 64. After the ACAM arrays, we use 7 XOR gates for decoding Gray code to binary form.

D. Computing more complicated functions with ACAMs

For more complicated non-linear functions, such as Softmax and DMMul, we observe that deploying one NL-DPE cannot compute their output bits efficiently. We next propose methods to configure NL-DPE adaptively to compute these functions.

Data-Dependent Matrix Multiplication (DMMul). A major difference between Sigmoid and DMMul is that DMMul depends on two inputs, both of which change dynamically. A naive implementation of DMMul in NL-DPE demands writing one input into RRAM cells, which not only introduces prohibitive performance and energy overheads but also degrades RRAM lifetime dramatically.

To address this, we propose to convert each multiplication into a sequence of logarithmic, exponential, and addition operations, i.e.:

$$a \times b = e^{(\log(a)+\log(b))} \tag{3}$$

Both exponentiation and logarithm functions are single-variable operations, which are suitable for computation using ACAMs in the NL-DPE. Fig. 6(a) illustrates the steps of computing the dot-product between two vectors, a fundamental component of DMMul. In Step 1, an element-wise logarithm is computed for both input vectors. With all elements now on the log scale, Step 2 replaces the multiplications with additions. In Step 3, the result is converted back to the linear scale through element-wise exponentiation. Finally, the linear scale values are summed to produce the final output.

In the figure, the logarithm and exponential operations are implemented using the ACAMs, while the add operations use the on-chip adders.

Softmax. Softmax entails exponentiation and division operations, both of which incur high digital hardware costs. By studying the Softmax function in Equation 2, we identify that its expensive exponentiation and division operations can be easily mitigated with ACAM-based computing. In particular, we exploit logarithm operations to convert a division operation as follows [11]:

$$\frac{a}{b} = e^{(\log(a)-\log(b))} \tag{4}$$

Fig. 6(b) illustrates the steps of computing the Softmax function after the conversion. In Step 1, the exponentiation of each element in Softmax's input vector is computed. Next, these exponentiated values are summed to obtain the denominator in Equation 2 (Step 2). In Step 3, this sum is converted to the log scale to use the subtraction in Step 4 and substitute the original division. Finally, Step 5 converts the result back to the linear scale with element-wise exponentiation.

Attention mechanism and its NL-DPE implementation. We next use Fig. 6(c) to illustrate how to configure multiple NL-DPEs to implement complicated non-linear functions. We use the attention mechanism in Fig. 3 as an example.

DMMul and Softmax extensively rely on exponentiation and logarithm operations, both of which can be implemented using the NL-DPE in ACAM-only mode. By examining the entire attention computation flow in Fig. 3, we can combine certain computations to optimize efficiency. Specifically, the inputs to the first DMMul are generated by Linear layers, and the output of Softmax feeds into the second DMMul. Fig. 6(c) illustrates an optimized mapping of the attention mechanism that minimizes redundant computations. Since the two inputs to DMMul_1 and the second input to DMMul_2 are

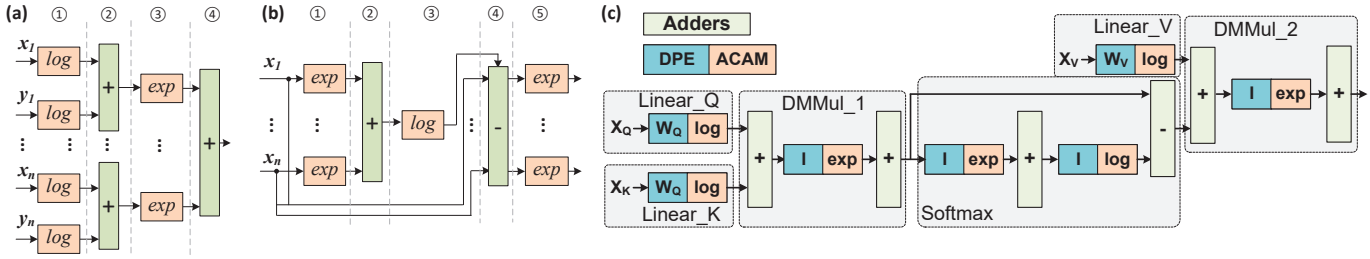


Fig. 6: **(a)** Steps of computing the dot-product between two vectors after replacing multiplication with exponentiation, logarithm, and addition. **(b)** Steps of computing Softmax with exponentiation, logarithm, and addition. **(c)** Mapping the attention mechanism onto NL-DPEs.

produced by Linear layers, we can simplify by using a single NL-DPE to handle both the Linear layers and the logarithm needed for DMMuls, treating the logarithm as an activation following the Linear layers. Additionally, because the final step in Softmax involves an element-wise exponentiation and the first step in DMMul_2 involves a logarithm—operations that are inverses of each other, these steps can be bypassed. As shown in Fig. 6(c), the log-scale output of Softmax is fed directly into DMMul_2, which no longer requires logarithmic processing for its second input. For all other computations, we still use NL-DPEs in the ACAM-only mode by storing an identity matrix in its crossbars.

IV. Addressing Non-Idealities in Analog IMC

In this section, we present our noise-aware fine-tuning (NAF) approach to enhance the noise resilience of NL-DPE. We first present our noise model and then elaborate on the NAF details.

A. Noise Models

It is well-known that RRAM-based IMC accelerators suffer from conductance noises, i.e., there is a non-negligible difference between the desired cell conductance and the cell conductance that was actually programmed or can be read [16] during computing. The noises come from two main sources.

- A RRAM cell may not be precisely programmed to the desired conductance levels. This is due to the stochastic nature of the ionic migration during the programming operation.
- Each read operation from a programmed memory cell may yield slightly varying conductance values. This fluctuation is due to various physical phenomena, such as thermal and random telegraph noise.

RRAM demonstrates minimal conductance drift after programming, which is neglected in this paper.

In this paper, we develop a noise model by fitting it to noise data collected from a fabricated RRAM test chip [15]. We run a program-and-verify algorithm on Ta-Ox RRAM devices using a tolerance of $\pm 0.55\mu$S for $G_{target} > 1\mu$S and tolerance proportional to the conductance levels for $G_{target} \leq 1\mu$S. We then read the conductance values 1000 times to assess the fluctuation noise.

Fig. 7(a) indicates that the maximum standard deviation of programming noise σ_{prog} is approximately 0.4μS and can be modeled as a function of the target conductance G_{target}. Fig. 7(b) indicates that the standard deviation of

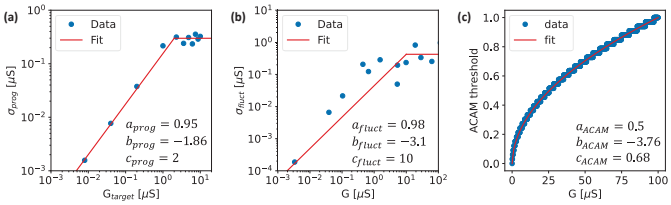


Fig. 7: Experimental data and model fitting of **(a)** standard deviation of programmed conductance σ_{prog} as a function of the target conductance G_{target}, **(b)** standard deviation of fluctuated conductance σ_{fluct} as a function of the mean conductance G, and **(c)** transfer function from the programmed conductance to ACAM threshold.

read fluctuation σ_{fluct} can be modeled as a function of the mean conductance G. These experimental results also match those reported in [16], where the log-scale standard deviation increases approximately linearly with the mean until it saturates at a certain point, which may be related to the threshold between low and high conductance states.

We model the programming and read noises using a normal distribution and compute their standard deviation as follows.

$$\sigma_x = \exp(a_x \log(G.\text{clip}(0, c_x) + b_x) \tag{5}$$

We fit the parameters a_x, b_x, and c_x to the experimental data and report them in Fig. 7(a,b). The programming and read fluctuation errors are then computed as $G_{write} = \sigma_{prog} \cdot \mathcal{N}(0,1)$ and $G_{read} = \sigma_{fluct} \cdot \mathcal{N}(0,1)$, respectively, where $\mathcal{N}(0,1)$ is a normal distribution with mean 0 and standard deviation of 1.

Given a RRAM cell that was programmed to its target conductance $\mathbf{G}_{target}$, we compute the readout conductance G as follows.

$$\mathbf{G} = \mathbf{G}_{target} + \mathbf{G}_{write} + \mathbf{G}_{read}. \tag{6}$$

The conductance model in Eq. 6 applies to RRAM cells in both crossbars and ACAM units. However, ACAM involves a non-linear operation, so the relation between conductance and the ACAM threshold needs to be modeled as well. Fig. 7(c) reports the experimental data on the ACAM threshold values as a function of the programmed conductance [19]. We then model the transfer function as

$$\text{TH} = \exp(a_{ACAM} * \log(G) + b_{ACAM}) + c_{ACAM} \tag{7}$$

with the fitting parameters reported in Fig. 7(c).

Note, we did not consider thermal and PVT variations in the paper. Their impacts on ACAM stability have been

studied in prior works [21]. The corresponding solutions, e.g., opportunely programming the RRAM devices to address the process variations on threshold voltage in the input transistors, are applicable because NL-DPE programs each ACAM individually, without the integration in the full model.

B. Noise-Aware Fine-tuning (NAF)

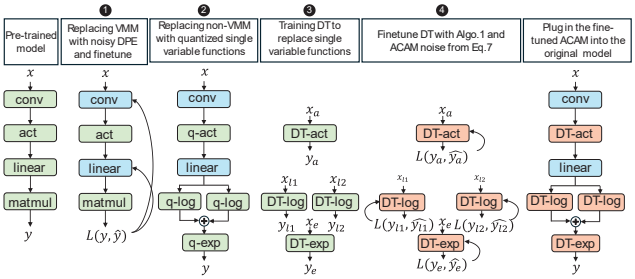


Fig. 8: Steps of mitigating noises in crossbars and ACAMs.

NL-DPE adopts Noise-Aware Fine-tuning (NAF) to achieve noise-resilience. As shown in Fig. 8, NAF is a software-based per-DT approach such that, given a pre-trained AI model, NAF extracts weight parameters and non-VMM operations, which shall be programmed to crossbars and ACAMs, respectively. We convert non-VMM operations to their corresponding DTs. NAF fine-tunes the model in software before deployment by integrating the noise model in both crossbars and ACAMs. While a few end-to-end model fine-tuning rounds are needed for crossbars, each DT is fine-tuned independently, without being integrated in end-to-end model fine-tuning. The fine-tuned model is then programmed to RRAM cell arrays, requiring no post-deployment in-device fine-tuning.

Fig. 8 illustrates the overall process. Given a pre-trained AI model, Step ❶ addresses noises in the crossbars, which is accomplished by a small number of NAF iterations (typically fewer than 10). During each iteration, NAF injects random noise based on our noise model (Eq. 6) into the weights of Convolutional and Linear layers.

Step ❷ converts all non-VMM operations (such as DMMul and Softmax) into single-variable functions (e.g., log, exp, etc.). The output of each single-variable function is quantized so that it is ready for training DTs in Step ❸. Step ❸ trains the DTs for each non-VMM operation that will be eventually deployed on ACAMs.

Step ❹ addresses ACAM noises that may impact DT's accuracy by performing NAF iterations for each DT *independently*. This is different from Step ❶ such that it does not require end-to-end model fine-tuning. After all these steps, the fine-tuned DTs replace the non-VMM operations in the original AI, which is then deployed to device by programming RRAM cells and is ready to use. We next discuss Steps ❶ and ❹ in more details.

Mitigating crossbar noise. Most prior works adopt digital slicing (D-SL) [4] to map model weights onto RRAM cells in crossbars. However, D-SL is suboptimal for mitigating RRAM noise, as it stores *discrete* programmed values in each cell. In contrast, we employ analog slicing (A-SL) [22], which maps model weights to RRAM cells using *continuous* analog values. This enables us to propose a novel A-SL-aware loss function for NAF to effectively reduce the impact of RRAM noise in crossbars.

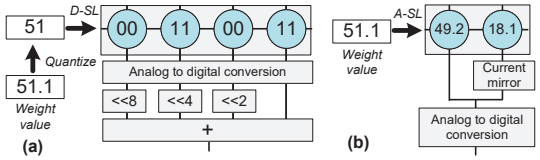


Fig. 9: Mapping a weight value onto RRAM cells in a crossbar with **(a)** digital slicing and **(b)** analog slicing.

Figure 9 illustrates the differences between D-SL and A-SL. In D-SL, a full-precision weight is first quantized to a fixed-point representation (e.g., 8 bits), with each bit segment stored in a separate RRAM cell. During computation, a shift-and-add circuit is used to combine the outputs from individual cells. In contrast, A-SL directly programs each RRAM cell's conductance to reflect a continuous analog value corresponding to the target weight. However, due to limited programming precision, there exists a residual error (denoted as ϵ) between the programmed conductance and the desired value. To compensate for this error, a second RRAM cell is used to store a scaled version of the error (e.g., $10 \times \epsilon$). During inference, an analog current mirror scales down the second cell's output before combining it with the output of the first cell.

Because A-SL stores continuous values in each RRAM cell, we can incorporate the inter-cell error (ϵ) as a regularization term in the NAF loss function:

$$Loss = MSE(y, \hat{y}) + \lambda_1 \|W\|_\infty + \lambda_2 \|\epsilon\|_\infty \tag{8}$$

Here, we use Mean Squared Error (MSE) between the model output and ground truth as the primary loss function. We also add an L_∞ regularization on the weights W to encourage smaller target conductance values, which are known to be less susceptible to noise according to Fig. 7(a) and (b). The coefficients λ_1 and λ_2 control the strength of these regularizations.

Mitigating ACAM noise. To fine-tune the DT thresholds for tolerating analog noises, we incorporate the DT into a training process that includes a forward pass and a backward pass, similar to a neural network. This process requires computing the gradients of the DT thresholds for updates. However, the computations in DTs, such as the comparisons between inputs and thresholds, are inherently non-differentiable, making gradient computation infeasible. To address this challenge, we propose a differentiable approximation of the DT computation based on its implementation in ACAM [21], [23], [24].

Algorithm 1 demonstrates the differentiable method for computing an 8-bit output using 8 DTs mapped to ACAMs. After extracting noisy DT thresholds from the conductance models (Line 2-6), in line 7, we replace the comparison operation between inputs and thresholds with a ReLU function. This ensures that $\mathbf{m}_i$ is positive if the input x lies within the range defined by the lower and upper thresholds. Line 8 uses a sum operation as a differentiable replacement for the OR gate in ACAM. Line 9 employs a division to quantize m_i to 0 or 1, with a small ϵ added to prevent division by zero. After the first loop, each m_i is a floating-point number very close to either 0 or 1, representing the Gray code output bit of an ACAM

Algorithm 1: Differentiable Approximation of ACAM

Input: x: one input value for fine-tuning; $\mathbf{w}_i^L$: lower thresholds of the DT for ith output bit; $\mathbf{w}_i^H$: higher thresholds of the DT for ith output bit;
Output: y: an 8-bit output of the ACAM
Data: g_{ratio}: weight-to-conductance ratio; g_{min}: minimum conductance of RRAM; g_{max}: maximum conductance of RRAM; ϵ: a very small positive constant to avoid division-by-zero.

1 **for** $i = 0$ **to** 7 **do**
2 $\quad \mathbf{g}_i^L = \text{Clip}(\mathbf{w}_i^L \cdot g_{ratio} + g_{min}, [g_{min}, g_{max}])$;
3 $\quad \mathbf{g}_i^H = \text{Clip}(\mathbf{w}_i^H \cdot g_{ratio} + g_{min}, [g_{min}, g_{max}])$;
4 $\quad \tilde{\mathbf{g}}_i^L = \text{Noise}(\mathbf{g}_i^L)$; $\tilde{\mathbf{g}}_i^H = \text{Noise}(\mathbf{g}_i^H)$;
5 $\quad \tilde{\mathbf{w}}_i^L = (\tilde{\mathbf{g}}_i^L - g_{min})/g_{ratio}$;
6 $\quad \tilde{\mathbf{w}}_i^H = (\tilde{\mathbf{g}}_i^H - g_{min})/g_{ratio}$;
7 $\quad \mathbf{m}_i = \text{ReLU}(x - \tilde{\mathbf{w}}_i^L) \cdot \text{ReLU}(\tilde{\mathbf{w}}_i^H - x)$;
8 $\quad m_i = \text{Sum}(\mathbf{m}_i)$;
9 $\quad m_i = m_i/(m_i + \epsilon)$;
10 **end**
11 $y = 0$;
12 **for** $i = 7$ **to** 0 **do**
13 $\quad$ **if** $i == 7$ **then**
14 $\quad\quad b_i = m_i$;
15 $\quad$ **else**
16 $\quad\quad b_i = (m_i - b_{i+1})^2$;
17 $\quad$ **end**
18 $\quad y = y + b_i \cdot 2^i$;
19 **end**

array. The second loop, in lines 12–19, implements a differentiable version of the XOR-based decode logic, replacing it with a combination of subtraction and squaring to maintain differentiability. Specifically, we approximate XOR using the squared difference of the two inputs (line 16). The decoded value is then obtained by summing each bit weighted by its corresponding power of two (line 18). Each bit is represented as a floating-point value, rather than a discrete binary value, to enable smooth gradient propagation.

As a result, all computations in Algorithm 1 are fully differentiable, allowing gradients of the threshold tensors to be computed during backpropagation.

V. Methodology

Key parameters, area, and power of NL-DPE components are summarized in Table II. Most parameters are adopted from ISAAC except crossbars and ACAMs, which are modeled via 16nm SPICE simulations [20]. For an 86×12 ACAM array, energy is measured by integrating voltage and current with all MLs discharging in the worst case. Each ACAM cell occupies $0.72\mu m^2$, consumes $\sim 0.44 fJ$ per search, and completes a search in $\sim 300ps$. Each tile has 8 cores, each with 4 crossbars (two for positive and two for negative weights, size 256×256), with each column connected to an ACAM unit containing 8 ACAM arrays. A total of 130 ACAM cells per unit supports all activation functions (Table I). The RRAM conductance range was set to $6.7k\Omega/100M\Omega$ (i.e., 0.01μS - 150μS) for optimal energy-accuracy trade-off (Section VI-A4). Noise models (Eq. 6) are applied at different NAF stages (Fig. 8), validated with data from a fabricated chip [15].

We compare NL-DPE with three baselines: (1) NVIDIA H100 GPU; (2) CMOS systolic array accelerator Eyeriss [25]; and (3) RRAM-based IMC accelerators ISAAC [4], RAELLA [6], and BRAHMS [7]. ISAAC uses ADCs for all VMM outputs and VFUs for non-VMM operations. RAELLA reduces ADC overhead via adaptive bit slicing but still relies on VFUs, while BRAHMS integrates ACAM for ReLU and hard-Tanh activations, requiring larger ACAM arrays and separate ADC/activation units. To support arbitrary activations, we augment all non-GPU baselines with a piecewise-linear Flex-SFU [9], which is only invoked for activations not originally supported (e.g., GeLU in BERT, SiLU in EfficientNet). Flex-SFU incurs higher energy than optimized digital logic. All accelerators are assumed to have enough tiles for 100M-parameter models, except in the NL-DPE multi-chip scalability analysis (Section VI-F), where the die area is matched to an H100 GPU.

Our GPU baseline is the NVIDIA H100 (16896 CUDA cores, 528 Tensor Cores, 80GB HBM3). All models are quantized to INT8 to match NL-DPE and exported in ONNX format. Quantization and inference use TensorRT, with 200 warm-up iterations followed by 1000 runs for measurement. Latency is reported by TensorRT, average power via NVIDIA-SMI, and accuracy is evaluated using floating-point GPU outputs as a high-quality baseline.

NL-DPE and all baselines are simulated in 32nm using CiMLoop [10]. For ISAAC and RAELLA, we use CiMLoop's implementations [4], [6], which originally support only VMMs. We extend it to non-VMMs by mapping the first operand as input and the second as weight, adding a Flex-SFU [9] and specifying operand mappings for scheduling. Single-operand operations use a one-element weight. DMMul is modeled as a grouped convolutional layer, and element-wise operations as a special convolutional layer, whose input is of size $N \times 1 \times 1 \times 1$ (i.e., N images, each image has only 1 pixel), where N is the element-wise operation size. Memory access and on-chip network overhead are included. For BRAHMS and NL-DPE, ADC and Flex-SFU are replaced with ACAM, implemented as custom CiMLoop plugins.

TABLE II: NL-DPE parameters.

1GHz on 32nm Technology node				
Component	**Spec**	**Params**	**Power** (mW)	**Area** (mm^2)
Core (8 cores per tile)				
DPE	number size	4 256×256	1.31	0.011534
ACAM	number size	256 130×1	43.52	0.041431
Input buffer	size	256B	0.23	0.00077
Output buffer	size	256B	0.23	0.00077
Register	size	128B	0.12	0.000385
DAC	number	4×256	4	0.00017
XOR	number	7×256	0.385	0.000215
Core Total	-	-	49.795	0.055275
Tile				
Register	size	768B	0.69	0.00231
Adders	number	256	12.8	0.0154
Shared memory	data	64KB	20.7	0.083
Core	number	8	398.36	0.4422
Tile Total	-	-	432.55	0.54291

We select a set of popular AI models for evaluation. For the CNNs, we tested CIFAR-10 [26] (on SENet [27], and EfficientNet [28]), and ImageNet [29] (on ResNet-34 [30], VGG11 [31], ShuffleNet-V2 [32] and DenseNet-121 [33]). For LLMs, we use the GLUE [34] dataset and test on the popular BERT model of different configurations (BERT-tiny [35] and BERT-base [14]). We also use LLaMA3.2 [36], a state-of-the-art LLM with billions of parameters, to test NL-DPE's

TABLE III: Accuracy of various stages in the proposed NAF.

Model	SENet	EfficientNet	ResNet-34	VGG11	ShuffleNet-v2	DenseNet121	BERT-tiny	BERT-tiny	BERT-base	BERT-base	BERT-base	BERT-base	BERT-base	BERT-base
Dataset	Cifar10	Cifar10	ImageNet	ImageNet	ImageNet	ImageNet	stsb$^+$	mnli	cola*	mrpc	rte	sst2	qnli	qqp
Baseline (FP32)	95.4	91.17	73.3	70.38	69.35	74.43	0.806	69.9	0.585	84.55	64.98	92.43	91.54	90.9
FP32 + crossbar noise	95.4	91.14	73.17	70.21	68.9	74.15	0.807	69.93	0.585	84.55	65.7	92.43	91.46	90.9
❶	95.4	91.17	73.3	70.22	69.01	74.15	0.81	69.92	0.586	84.73	65.79	92.6	91.98	90.99
❷	94.67	90.81	73.15	69.96	68.54	72.39	0.805	69.86	0.597	83.57	65.34	92.2	89.4	88.89
❸	93.86	89.52	72.92	69.35	68.22	72.42	0.81	68.13	0.57	82.35	64.25	92.31	89.09	89.12
❸ + ACAM noise	61.92	54.06	41.94	50.53	61.69	53.23	0.701	49.00	0.461	78.84	61.73	90.71	86.28	87.41
❹	93.23	89.05	72.06	67.4	68.05	71.34	0.805	68.58	0.58	84.55	64.33	92.43	90.32	89.19

$^+$ = Pearson Correlation Coefficient (PCC), range [-1, 1], higher is better. * = Matthews Correlation Coefficient (MCC), range [-1, 1], higher is better.

scalability in a multi-chip setting. The code used in this work is available at: https://github.com/HewlettPackard/nl-dpe.

VI. Results

A. Design Space Exploration and Trade-offs

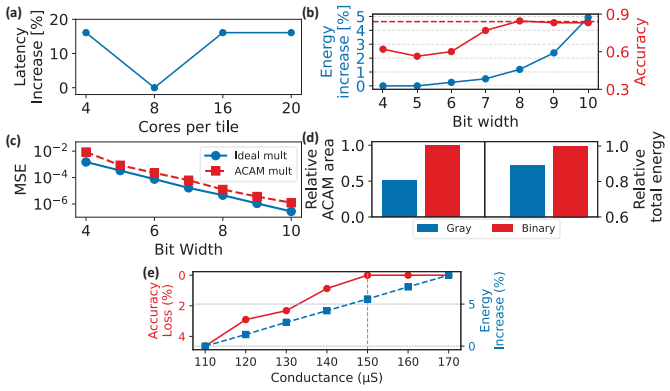


Fig. 10: **(a)**Latency increase of NL-DPE accelerator with varying core counts per tile (normalized to the 8-core configuration).**(b)** Energy and accuracy as a function of bit width. **(c)** Mean Square Error (MSE) performing multiplication as a function of bit width for ideal software multiplier and ACAM-based multiplier.**(d)** Relative ACAM area and total energy for Gray and Binary encoding. **(e)** Tradeoff between accuracy, energy consumption and RRAM's maximum conductance on BERT-base.

We begin by performing design space exploration to determine key parameters and investigate trade-offs in our accelerator architecture. All results are based on BERT-base running the mrpc task, with other benchmarks showing similar trends.

1) Cores in Tile: We evaluate inference latency for different core counts per NL-DPE tile (Fig. 10(a), Table II). Latency is normalized to the 8-core setup, which achieves the best performance. Fewer cores underutilize parallelism, while more cores face resource contention and off-chip memory access. Thus, we adopt 8 cores per tile for all experiments.

2) Bit width exploration: We study the impact of bit width on NL-DPE. Higher bit widths improve accuracy but increase ACAM usage and energy. Below 7 bits, accuracy drops $> 50\%$ even with NAF. At 8 bits, accuracy matches full precision with $< 1\%$ energy increase than 7 bits, and higher widths offer no gain. We adopt 8-bit precision for NL-DPE.

Fig. 10(c) shows the MSE between ACAM DT-based multiplication and ideal software computation across bit widths. An 8-bit ACAM multiplier achieves precision comparable to a 7-bit digital multiplier.

3) Gray encoding: We evaluate the impact of Gray encoding on area and energy, as shown in Fig. 10(d). Gray encoding reduces the ACAM size by 50% at the cost of a few additional XOR gates, whose area and energy overheads are negligible compared to the savings from the smaller ACAM, as shown by the relative energy savings. We evaluated the inference accuracy for binary and Gray encoding, without seeing significant differences (e.g., both reach software equivalent accuracy in the case of BERT-base after NAF.)

4) Conductance range: Fig. 10(e) shows the trade-off between RRAM conductance range, accuracy, and energy. Increasing the maximum conductance improves precision and reduces noise impact, but raises energy. Accuracy saturates beyond 150μS, so we adopt a 0.01–150μS range for all crossbar and ACAM RRAMs.

B. ACAM vs. Digital Function Units

We evaluate ACAM as a replacement for ADC and activation units, comparing it to an 8-bit ADC [37] and FlexSFU [9] at 16nm. Table IV shows 98% lower power and 86× smaller area at a constant latency, enabling each crossbar column to have a dedicated ACAM for maximum performance.

TABLE IV: ACAM vs. ADC+Activation.

	ADC	Activations	Total	ACAM	Improvement
Power $[mW]$	5.73	3.38	9.11	0.17	∼98% better
Area $[\mu m^2]$	457	20867	21324	247	∼ ×86 better

TABLE V: ACAM vs. Digital Multipliers.

	Fixed Point	Floating Point	ACAM
Power $[\mu W]$	539	1582	513
Area $[\mu m^2]$	1059	43610	774

Table V compares our 8-bit ACAM multiplier with conventional 8-bit digital designs, all scaled to 16nm. Compared to the fixed-point design [38], it reduces power by 5% and area by 27%. Unlike digital multipliers, ACAM directly processes continuous analog crossbar outputs without ADCs, supports flexible output formats (e.g., Float8, BFloat8), so we also compare with a floating-point design [39] and achieves 68% lower power and 98% smaller area.

C. Ramp vs Flash non-linear ADC

Table VI reports the comparison with the 16nm ramp ADC presented in [18], which is also capable of computing the activation functions while performing the conversion. When comparing the conversion speed and latency of the two ADCs, the flash topology we propose has a clear edge: the conversion happens in a single search cycle, and the latency does not worsen exponentially with the number of bits. The circuit proposed in [18] shows superior energy-per-conversion, for

the 5-bit case; when normalizing by the number of levels of the conversion (Walden FOM, typically reported in ADC literature), the ramp ADC is 20-24% more energy efficient. We amortize the area of the ramp ADC in [18] (including integrators and comparators) across columns for a fair comparison. Under this normalization, the ramp ADC exhibits a smaller per-column area compared to our flash ADC design. However, this area efficiency is achieved at the cost of sequential conversion and longer latency, which fundamentally limits its ability to meet the high-throughput demands of CIM DPEs.

TABLE VI: Ramp vs Flash non-linear ADC.

	ENOB	Sampl Freq. [MHz]	Energy per conv. [pJ]	Area per Col. [um^2]	Walden FOM [fJ/levels]
This work	8	1,000	5.73	247	22.4
[18] (5-bit)	5	31.2	0.595	14.4	18.6
[18] (8-bit)	8	3.9	4.6	14.4	18.0

D. Accuracy

Table III shows the accuracy of various models at different stages of NAF, except for stsb and cola datasets, which use Pearson Correlation Coefficient (PPC) and Matthews Correlation Coefficient (MCC) as metrics. The ball numbers correspond to the steps depicted in Fig. 8. Only a small error is introduced by the crossbars even in the presence of noise (FP32 + crossbar noise and ❶), thanks to the mitigation effect of A-SL. Using DT-trained piecewise activations (❸) doesn't introduce significant error as well, making the approach potentially generalizable to other accelerators that may efficiently perform inference of tree-based models [19]. However, in the presence of ACAM noise, the DTs experience significant accuracy degradation (as shown by ❸ + ACAM noise), which can only be mitigated through ACAM NAF (❹). Thus, results demonstrate that our DT-trained activations approach is feasible and does not impact accuracy, but NAF is needed for practical utilization with analog accelerators. All numbers reported in Table III are average across 10 runs using different random seeds. We observe the standard deviation is consistently small (below 1%).

E. Performance and Energy

Fig. 11 shows NL-DPE speedup and energy efficiency versus GPU across batch sizes. For single-sample inference ($BS = 1$), GPU suffers from low utilization, while NL-DPE is 112× faster and 28× more energy-efficient. Compared to other IMC accelerators, NL-DPE outperforms by replacing bulky ADCs/VFUs and natively supporting attention, reducing digital post-processing latency.

We evaluate increasing batch sizes. While GPUs benefit from large batches, conventional IMC accelerators see limited gains due to shared ADCs and VFUs, causing high latency and bottlenecks. In contrast, NL-DPE maintains high speedup and energy efficiency across all batches: each crossbar column has a dedicated ACAM for ADC conversion, reducing conversion latency to 3ns, and ACAM units compute multiple DMMul and activations in parallel. Eyeriss suffers from repeated memory accesses. NL-DPE consistently outperforms all baselines on BERT, achieving 249× speedup over GPU for multi-batch inference.

Fig. 12 presents the energy breakdown of on-chip components in NL-DPE for a representative CNN workload (ResNet34) and LLM workload (BERT). Most of the energy is consumed by the compute units (i.e., crossbars, ACAMs, and adders), highlighting the high computational intensity of the NL-DPE architecture.

Importantly, for LLM, NL-DPE is the only architecture improving the performance compared with modern GPUs, particularly for large batch sizes. This is due to the unique capabilities of performing the attention mechanism by a series of highly efficient function approximations with ACAMs, without the need for rewriting the crossbar arrays or using VFUs. Thus, the NL-DPE architecture distinguishes itself as the only IMC solution capable of scalable performance for contemporary, large-scale workloads.

F. Scalability for Larger LLMs

1) Hardware Scalability: As model size grows, a single chip cannot hold the entire model. We adopt a multi-chip setup to avoid RRAM write endurance issues, rather than time-multiplexing, which incurs frequent writes.

BERT-base fits on one chip, so we evaluate scalability with LLaMA models: the 1B-parameter variant (LLaMA3.2-1B) uses 3 chips, and the 3B-parameter variant (LLaMA3.2-3B) uses 8 chips. Chips are connected via a 10 Gbps chip-to-chip (C2C) interface with 30 pJ/bit energy cost, representing a conservative, worst-case scenario.

Even with this overestimated C2C overhead, NL-DPE achieves approximately 100× speedup and energy efficiency improvement over GPUs (Fig. 13(a)). Pie charts show C2C energy rises with chip count (18% for 1B, 35% for 3B), while most energy remains within NL-DPE chips. Off-chip memory contributes negligibly ($< 1.5\%$) to total energy consumption due to IMC's weight-stationary design.

2) NAF Scalability: Scalability is a key concern for in-memory accelerators, especially regarding in-device or end-to-end finetuning. We next evaluate the overhead of our proposed NAF for larger models.

Our results show NL-DPE with NAF is highly scalable. NAF is software-based and applied before deployment, with each DT trained and fine-tuned independently, avoiding full LLM finetuning. To approximate an activation function, each DT uses only 5000 randomly sampled inputs, completing training in seconds via scikit-learn. Fine-tuning uses a small input set for a few epochs; 10 epochs suffice to recover accuracy lost to ACAM noise (Fig. 13(b)). Since DTs are lightweight and independent, multiple DTs can be processed in parallel, ensuring high scalability.

G. Noise Analysis

1) ACAM-Based Functions: We assess ACAM-based computation by plotting the correlation between its outputs and ideal digital results.

- Figure 14(a) shows a strong linear correlation between the 8-bit ACAM multiplier and a conventional 8-bit digital

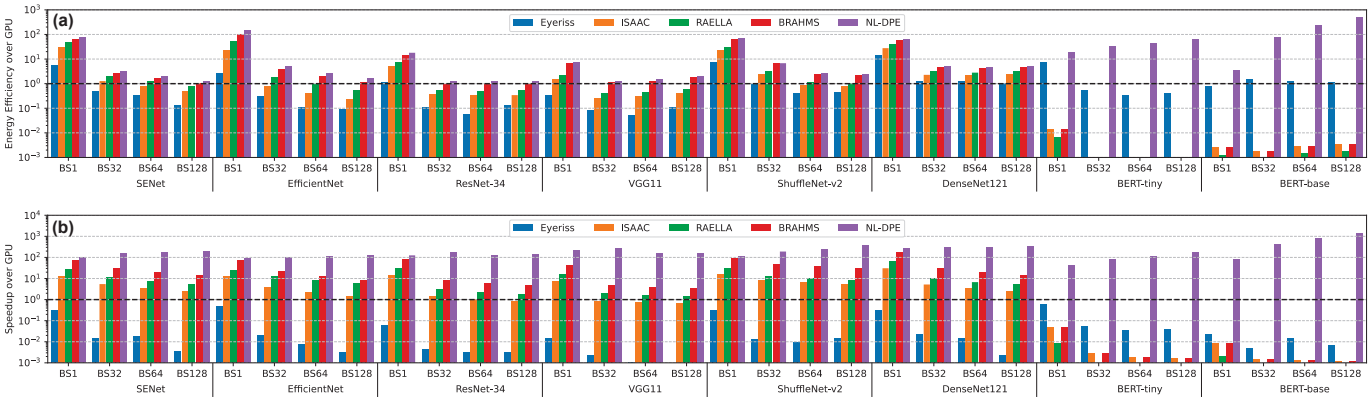


Fig. 11: (a) Normalized energy efficiency and (b) speedup compared to the GPU baseline

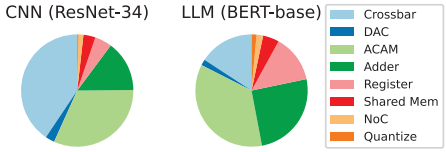


Fig. 12: Energy breakdown for various components on a NL-DPE chip for representative CNN and LLM workloads.

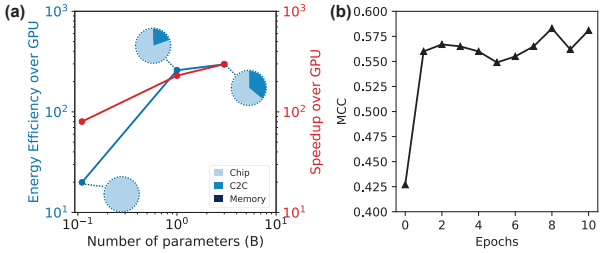


Fig. 13: **(a)** Energy efficiency and speedup over GPU when scaling up NL-DPE for larger LLMs (i.e. LLaMA models). **(b)** MCC of cola on BERT-base as a function of NAF epochs.

multiplier across 500 inputs, with MSE 2.8971×10^{-5} and variance 1.9651×10^{-5}.

- Figure 14(b) shows strong correlation between 256×256 ACAM-based matrix multiplication and the digital baseline, with MSE 8.9037×10^{-4} and variance 4.4807×10^{-3}, reflecting realistic LLM workloads.
- Figure 14(c) further shows the softmax unit's correlation with the digital baseline, exhibiting a mean error of -1.9257×10^{-5} and variance 6.2711×10^{-7}, confirming high numerical fidelity for complex computations.

These results demonstrate that our proposed ACAM-based computation, after NAF, achieves a precision very close to its digital counterpart.

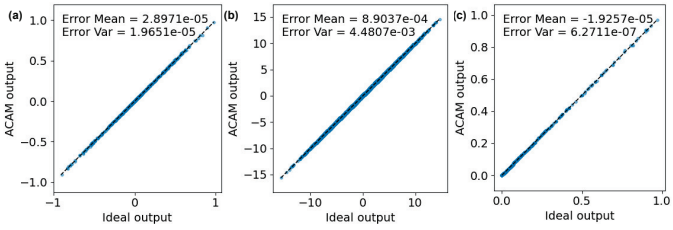


Fig. 14: Correlation between the outputs of the proposed ACAM-based units and the ideal outputs: **(a)** multiplication results, **(b)** matrix multiplication of two 256×256 matrices, **(c)** softmax outputs.

2) Variation of Noise Distribution: To test NAF's robustness to varying noise, we varied the noise standard deviation during inference. Figure 15 plots accuracy loss versus normalized noise standard deviation (0.5×–2.5×, baseline = 1.0).

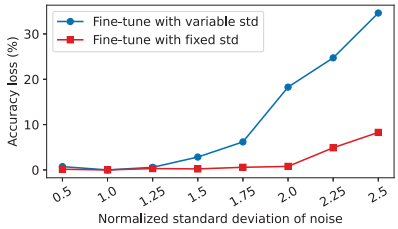


Fig. 15: Accuracy loss of BERT-base on sst2 under different inference-time noise standard deviations. Averaged across 5 runs.

- Scaled-noise training (blue curve): In this case, the training was performed with scaled noise std values corresponding to the inference-time settings. We found that when the noise std is enlarged (e.g., >1.5×), the model experiences noticeable accuracy degradation due to the reduced convergence stability during fine-tuning.
- Fixed-noise training (red curve): During fine-tuning, the model was trained with the hardware-characterized noise model (normalized std = 1.0) as proposed in our noise model, and only the inference-time noise was varied. We observe that the accuracy remains stable even when the inference noise std is increased by up to 2×, demonstrating that our noise-aware fine-tuning method generalizes well to different noise levels.

These results suggest that our hardware-calibrated noise model not only eliminates the need for on-device fine-tuning but also provides sufficient generalization across different device noise levels, without requiring Gaussian noise injection during training.

3) Stuck-At Faults: Figure 16 shows the impact of stuck-at faults (SAFs) on the crossbar and ACAM, and the tolerance of our Noise-Aware Fine-tuning (NAF), using BERT-base on SST-2 from the GLUE dataset.

We first evaluate two crossbar weight mappings under 5–30% SAFs, assuming ideal ACAM: conventional digital mapping (red) and analog slicing (blue). Both maintain near-ideal accuracy up to 5% faults, while analog slicing tolerates

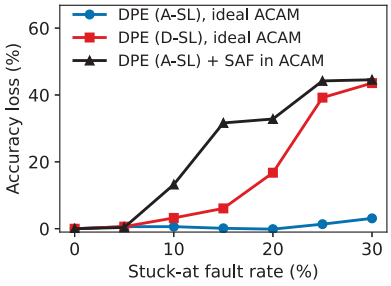


Fig. 16: Accuracy degradation under stuck-at faults (SAFs) in crossbar and ACAM averaged across 5 runs. Red: crossbar SAFs with digital slicing. Blue: crossbar SAFs with analog slicing. Black: SAFs in both ACAM and crossbar (analog slicing).

up to 20% SAFs by distributing each weight across two RRAM cells, allowing fault-free cells to partially compensate for stuck cells.

Then we test SAF on ACAM, which shows higher sensitivity because (1) analog slicing is not applicable to its AND operations, and (2) each output bit is computed independently, so errors in higher bits cause exponentially larger deviations.

During NAF, two strategies are used to mitigate stuck-at faults (SAFs) on ACAM: (1) if a DT does not use all ACAM rows, data mapped to a faulty cell can be reassigned to an unused row, disabling the defective one by skipping its pre-charge during inference; (2) knowing SAF locations in advance, NAF avoids updating or assigning values to these cells during fine-tuning. Figure 16 (black) shows accuracy degradation under these optimizations, tolerating up to 5% SAFs. Additional methods, such as error correction coding (ECC) [40], could further mitigate SAFs, left for future work.

VII. Discussion

NAF for multiple chips. Unlike prior work requiring post-deployment device-in-the-loop fine-tuning, our NAF integrates a per-device, pre-deployment noise model by reading/writing different resistance values. This eliminates fine-tuning on hardware, improves endurance, and minimizes overhead. Noise extraction can be incorporated into fabrication, and software-based fine-tuning requires fewer than 10 epochs (Fig. 13(b)), keeping multi-chip calibration overhead minimal.

Conductance drift. Our experiments show TaOx RRAMs have negligible conductance drift, with only stochastic fluctuations, making NAF highly effective. For other memory types with drift, NAF can incorporate a drift model without changing the training algorithm (Algorithm 1).

Endurance and wear-out analysis. NL-DPE targets inference, with fine-tuned weights programmed once and reused. Device programming during noise profiling requires only $\sim 10s$ cycles [41], negligible compared to RRAM endurance of up to 10^8 cycles [13].

Potential paths for improving the accuracy. NAF fine-tunes each DT individually using synthetic data, keeping overhead low and avoiding original training data. If original training data is available, end-to-end fine-tuning can further improve accuracy at higher cost. DT ensembles (e.g., Random Forests, XGBoost) or error correction codes [40] can also enhance robustness against ACAM errors.

Other operators. Few non-critical operations may need special handling. In LLMs, layer normalization can run on a VFU, though it can be replaced by Dynamic Tanh [42], which ACAM handles efficiently. In CNNs, max pooling is the only exception, using simple comparators.

Sparse models. Sparsity is orthogonal and can be applied on top of our design. For example, SRE [43] improves efficiency by activating crossbars at finer granularity, reducing ADC precision needs. Similarly, when applied to NL-DPE, this can reduce ACAM usage.

VIII. Related Work

DPE-ACAM hybrid architectures. BRAHMS [7] uses ACAM to replace ADCs and project analog signals into non-linear outputs, first routing inputs through pass-gates to produce ReLU-like activations. A/D conversion and projection are separate, adding overhead (Fig. 11). Fuse-and-Mix [8] merges projection and A/D conversion, using ACAM as an address decoder for a memory lookup. In contrast, NL-DPE outputs the result directly from ACAM, reducing peripherals, and supports arbitrary activations beyond ReLU$-\alpha$.

In-memory computing and attention accelerators. ReTransformer [11] and AttCIM [44] avoid programming the intermediate $\mathbf{K}$ matrix into CIM crossbars through matrix decomposition. However, the input matrix $\mathbf{X}$ still needs to be reprogrammed into the array. In contrast, NL-DPE completely eliminates reprogramming of the CIM array. [45] and [46] reduce ADC overhead by computing activation functions in the analog domain. However, [45] supports only ReLU and Tanh, while [46] replaces the widely used Softmax with HardSigmoid. In contrast, NL-DPE supports arbitrary activation functions directly in the analog domain. Transformer performance can be further improved by exploiting attention sparsity [12], [47]. Kernel approximation techniques [48] also reduce computation, but introduce additional overhead from element-wise multiplications, exponentials, and L2-norm operations in FP32. We leave the integration of these optimizations as future work.

IX. Conclusion

We present NL-DPE, an ADC-less analog IMC primitive that unifies crossbars for VMMs and ACAMs for non-VMMs, enabling major CNN and LLM computations entirely in the analog domain. Non-VMMs are approximated with DTs and efficiently processed by ACAMs. A software-based NAF improves accuracy under RRAM noise. Compared to conventional DPEs and GPUs, NL-DPE achieves 28× energy efficiency and 249× speedup on average. NL-DPE offers a new building block for in-memory computing, fully leveraging programmable analog non-linear operations.

X. Acknowledgment

The authors acknowledge the use of AI assistants (ChatGPT, OpenAI) to improve the readability and clarity of portions of the manuscript text and rebuttal responses. The AI tool was used solely for language refinement, and all technical content, ideas, and conclusions are solely the responsibility of the authors.

REFERENCES

[1] W. Wan, R. Kubendran, C. Schaefer, S. B. Eryilmaz, W. Zhang, D. Wu, S. Deiss, P. Raina, H. Qian, B. Gao *et al.*, "A compute-in-memory chip based on resistive random-access memory," *Nature*, vol. 608, no. 7923, pp. 504–512, 2022.

[2] M. Le Gallo, R. Khaddam-Aljameh, M. Stanisavljevic, A. Vasilopoulos, B. Kersting, M. Dazzi, G. Karunaratne, M. Brändli, A. Singh, S. M. Mueller *et al.*, "A 64-core mixed-signal in-memory compute chip based on phase-change memory for deep neural network inference," *Nature Electronics*, vol. 6, no. 9, pp. 680–693, 2023.

[3] M. Jerry, P.-Y. Chen, J. Zhang, P. Sharma, K. Ni, S. Yu, and S. Datta, "Ferroelectric fet analog synapse for acceleration of deep neural network training," in *2017 IEEE international electron devices meeting (IEDM)*. IEEE, 2017, pp. 6–2.

[4] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[5] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy *et al.*, "Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, 2019, pp. 715–731.

[6] T. Andrulis, J. S. Emer, and V. Sze, "Raella: Reforming the arithmetic for efficient, low-resolution, and low-loss analog pim: No retraining required!" in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–16.

[7] T. Song, X. Chen, X. Zhang, and Y. Han, "Brahms: Beyond conventional rram-based neural network accelerators using hybrid analog memory system," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1033–1038.

[8] H. Zhu, K. Zhu, J. Gu, H. Jin, R. T. Chen, J. A. Incorvia, and D. Z. Pan, "Fuse and mix: Macam-enabled analog activation for energy-efficient neural acceleration," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–9.

[9] E. Reggiani, R. Andri, and L. Cavigelli, "Flex-sfu: Accelerating dnn activation functions by non-uniform piecewise approximation," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[10] T. Andrulis, J. S. Emer, and V. Sze, " CiMLoop: A Flexible, Accurate, and Fast Compute-In-Memory Modeling Tool ," in *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2024, pp. 10–23.

[11] X. Yang, B. Yan, H. Li, and Y. Chen, "Retransformer: Reram-based processing-in-memory architecture for transformer acceleration," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[12] A. Yazdanbakhsh, A. Moradifirouzabadi, Z. Li, and M. Kang, "Sparse attention acceleration with synergistic in-memory pruning and on-chip recomputation," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 744–762.

[13] P. Mannocci, M. Farronato, N. Lepri, L. Cattaneo, A. Glukhov, Z. Sun, and D. Ielmini, "In-memory computing with emerging memory devices: Status and outlook," *APL Machine Learning*, vol. 1, no. 1, 2023.

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186.

[15] X. Sheng, C. E. Graves, S. Kumar, X. Li, B. Buchanan, L. Zheng, S. Lam, C. Li, and J. P. Strachan, "Low-conductance and multilevel cmos-integrated nanoscale oxide memristors," *Advanced electronic materials*, vol. 5, no. 9, p. 1800876, 2019.

[16] R. Mao, B. Wen, M. Jiang, J. Chen, and C. Li, "Experimentally-validated crossbar model for defect-aware training of neural networks," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 5, pp. 2468–2472, 2022.

[17] H. Jiang, W. Li, S. Huang, and S. Yu, "A 40nm analog-input adc-free compute-in-memory rram macro with pulse-width modulation between sub-arrays," in *2022 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*. IEEE, 2022, pp. 266–267.

[18] J. Yang, R. Mao, M. Jiang, Y. Cheng, P.-S. V. Sun, S. Dong, G. Pedretti, X. Sheng, J. Ignowski, H. Li *et al.*, "Efficient nonlinear function approximation in analog resistive crossbars for recurrent neural networks," *Nature Communications*, vol. 16, no. 1, p. 1136, 2025.

[19] G. Pedretti, C. E. Graves, S. Serebryakov, R. Mao, X. Sheng, M. Foltin, C. Li, and J. P. Strachan, "Tree-based machine learning performed in-memory with memristive analog cam," *Nature communications*, vol. 12, no. 1, p. 5806, 2021.

[20] C. Li, C. E. Graves, X. Sheng, D. Miller, M. Foltin, G. Pedretti, and J. P. Strachan, "Analog content-addressable memories with memristors," *Nature communications*, vol. 11, no. 1, p. 1638, 2020.

[21] G. Pedretti, C. E. Graves, T. Van Vaerenbergh, S. Serebryakov, M. Foltin, X. Sheng, R. Mao, C. Li, and J. P. Strachan, "Differentiable content addressable memory with memristors," *Advanced electronic materials*, vol. 8, no. 8, p. 2101198, 2022.

[22] G. Pedretti, P. Mannocci, C. Li, Z. Sun, J. P. Strachan, and D. Ielmini, "Redundancy and analog slicing for precise in-memory machine learning—part i: Programming techniques," *IEEE Transactions on Electron Devices*, vol. 68, no. 9, pp. 4373–4378, 2021.

[23] L. Zhao, L. Buonanno, A. Natarajan, J. Ignowski, and G. Pedretti, "Noise aware finetuning for analog non-linear dot product engine," in *NeurIPS 2024 Workshop Machine Learning with new Compute Paradigms*, 2024.

[24] L. Zhao, L. Buonanno, R. M. Roth, S. Serebryakov, A. Gajjar, J. Moon, J. Ignowski, and G. Pedretti, "Race-it: A reconfigurable analog cam-crossbar engine for in-memory transformer acceleration," *The 43rd IEEE International Conference on Computer Design*, 2025.

[25] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 367–379.

[26] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[27] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7132–7141.

[28] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*. PMLR, 2019, pp. 6105–6114.

[29] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[30] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[31] K. Simonyan, "Very deep convolutional networks for large-scale image recognition," *International Conference on Learning Representations (ICLR)*, 2015.

[32] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 116–131.

[33] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[34] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *In the Proceedings of ICLR.*, 2019.

[35] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu, "Tinybert: Distilling bert for natural language understanding," *arXiv preprint arXiv:1909.10351*, 2019.

[36] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[37] P. Wang, F. Li, and Z. Wang, "A single-channel 8-bit 1.6-gs/s alternate-comparator sar adc with dither-based background offset calibration in 28-nm cmos," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2025.

[38] S. Mondal, R. Marimuthu, and S. Ravi, "Approximate 8-bit multipliers and their physical design implementation," *e-Prime-Advances in Electrical Engineering, Electronics and Energy*, vol. 6, p. 100300, 2023.

[39] E.-P. T. Computing, "Fpnew: An open-source multi-format floating-point unit architecture for energy-proportional transprecision computing."

[40] R. M. Roth and G. Pedretti, "On-access error correction in certain types of content-addressable memories," in *2024 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2024, pp. 1161–1166.

[41] E. J. Merced-Grafals, N. Dávila, N. Ge, R. S. Williams, and J. P. Strachan, "Repeatable, accurate, and high speed multi-level programming of memristor 1t1r arrays for power efficient analog computing applications," *Nanotechnology*, vol. 27, no. 36, p. 365202, 2016.

[42] J. Zhu, X. Chen, K. He, Y. LeCun, and Z. Liu, "Transformers without normalization," in *Proceedings of the Computer Vision and Pattern Recognition Conference*, 2025, pp. 14 901–14 911.

[43] T.-H. Yang, H.-Y. Cheng, C.-L. Yang, I.-C. Tseng, H.-W. Hu, H.-S. Chang, and H.-P. Li, "Sparse reram engine: Joint exploration of activation and weight sparsity in compressed neural networks," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 236–249.

[44] R. Guo, Z. Yue, Y. Wang, H. Li, T. Hu, Y. Wang, H. Sun, J.-L. Hsu, Y. Zhang, B. Yan *et al.*, "A 28-nm 28.8-tops/w attention-based nn processor with correlative cim ring architecture and dataflow-reshaped digital-assisted cim array," *IEEE Journal of Solid-State Circuits*, 2024.

[45] H. Wang, R. Liu, R. Dorrance, D. Dasalukunte, N. M. Gowda, and B. Carlton, "Enhancing ai acceleration: A calibration-free, pvt-robust analog compute-in-memory macro with activation functions," *IEEE Solid-State Circuits Letters*, 2024.

[46] N. Leroux, P.-P. Manea, C. Sudarshan, J. Finkbeiner, S. Siegel, J. P. Strachan, and E. Neftci, "Analog in-memory computing attention mechanism for fast and energy-efficient large language models," *Nature Computational Science*, vol. 5, no. 9, pp. 813–824, 2025.

[47] Y. Wang, Y. Qin, D. Deng, J. Wei, Y. Zhou, Y. Fan, T. Chen, H. Sun, L. Liu, S. Wei *et al.*, "An energy-efficient transformer processor exploiting dynamic weak relevances in global attention," *IEEE Journal of Solid-State Circuits*, vol. 58, no. 1, pp. 227–242, 2022.

[48] J. Büchel, G. Camposampiero, A. Vasilopoulos, C. Lammie, M. Le Gallo, A. Rahimi, and A. Sebastian, "Kernel approximation using analogue in-memory computing," *Nature Machine Intelligence*, vol. 6, no. 12, pp. 1605–1615, 2024.