

TRAM: Training Approximate Multiplier Structures for Low-Power AI Accelerators

Chang Meng
Eindhoven University of Technology
Eindhoven, The Netherlands
c.meng@tue.nl

Hanyu Wang
University of California, Los Angeles
Los Angeles, USA
hanyuwang@g.ucla.edu

Yuyang Ye
Chinese University of Hong Kong
Hong Kong, China
yuyangye@cuhk.edu.hk

Mingfei Yu
École Polytechnique Fédérale de
Lausanne, Lausanne, Switzerland
mingfei.yu@epfl.ch

Wayne Burleson
University of Massachusetts Amherst
Amherst, USA
burleson@umass.edu

Giovanni De Micheli
École Polytechnique Fédérale de
Lausanne, Lausanne, Switzerland
giovanni.demicheli@epfl.ch

Abstract

Reducing power consumption in AI accelerators is increasingly important. Approximate computing can reduce power consumption while keeping the accuracy loss small. Since multipliers are power-hungry components in AI models, this paper focuses on synthesizing low-power *approximate multipliers* (AxMs). Unlike prior works that design AxMs separately from AI model training, we present *TRAM*, which jointly optimizes the AxM structure and AI model parameters to lower power with small accuracy loss. Experiments show that compared to state-of-the-art AxMs, TRAM achieves up to 25.05% AxM power reduction on CNNs with CIFAR-10, and reduces power by up to 27.09% on vision transformers with ImageNet.

CCS Concepts

• Hardware → Hardware-software codesign.

Keywords

Approximate multiplier, hardware-software co-optimization, low-power, AI accelerator

1 Introduction

The wide deployment of AI accelerators raises concerns about power consumption and creates an urgent need for low-power computing solutions [1]. Approximate computing reduces power consumption by allowing inaccuracies in computations, making it a promising approach to addressing these concerns [2]. Since multipliers are among the most power-consuming components in AI accelerators [3], this paper studies the automatic synthesis of low-power *approximate multipliers* (AxMs).

Many studies have investigated both automatic synthesis and manual design of AxMs [4]. For example, Mrazek *et al.* [5] proposed a genetic programming-based method to synthesize AxMs and later extended it to *convolutional neural networks* (CNNs) [6]. Xiao *et*

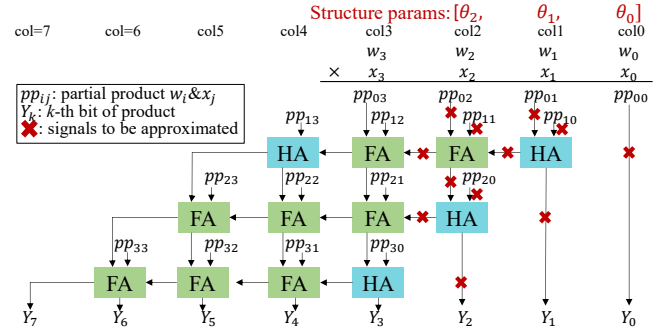


Figure 1: A 4-bit unsigned array multiplier. The red crosses indicate candidate signals that may be approximated. The structure parameter θ_i controls the approximation degree of the i -th accumulation column. We assume that at most $P = 3$ columns can be approximated.

al. [7] formulated AxM synthesis as an integer programming problem and produced low-cost AxMs. Hu *et al.* [8] manually designed AxMs for CNNs using partial product speculation. Furthermore, approximate logic synthesis tools, such as those proposed in [9–11], can synthesize AxMs as well.

However, all the above methods overlook the specific context of AI models, which can lead to suboptimal results when deploying AxMs in accelerators. First, many existing methods do not consider the data distribution used by AI models. For instance, Xiao *et al.* [7] assumed a uniform input distribution, while real data distributions vary across layers. Ignoring this variability can lead to suboptimal designs. Second, most existing works design or synthesize AxMs using local error metrics such as error rate or error distance. However, a small local error does not always translate into a small final accuracy loss in the AI model.

To address these issues, we present *TRAM*, a hardware-software co-optimization framework that Train Approximate Multiplier structures for low-power AI accelerators. TRAM formulates AxM synthesis as a joint optimization problem that updates the AxM structure and AI model parameters together during training. By using real training data, TRAM captures the statistics seen by each multiplier and optimizes both the multiplier structure and model parameters with respect to the final accuracy loss. Our contributions are summarized as follows:

- We introduce a parameterization of the AxM structure, in which each column in the compressor tree is assigned a continuous

This work is supported in part by Synopsys Inc. Corresponding author: Yuyang Ye.



This work is licensed under a Creative Commons Attribution 4.0 International License. DAC '26, Long Beach, CA, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2254-7/2026/07

<https://doi.org/10.1145/3770743.3804298>

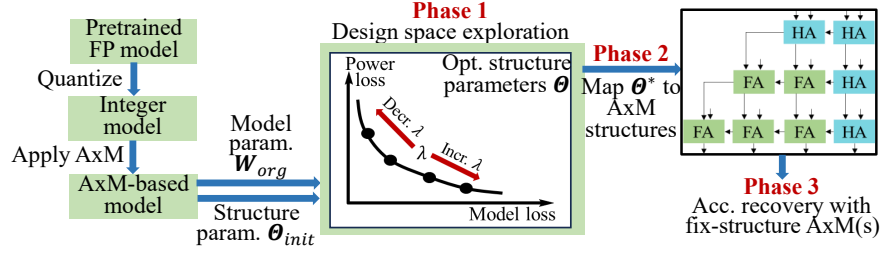


Figure 2: TRAM framework overview.

structure parameter that controls the approximation degree of the column. These parameters are optimized using gradient descent.

- We devise an analytic power model that estimates multiplier power from the structure parameters and provides useful hardware-aware guidance during training.
- We propose an efficient mapping method that converts the optimized structure parameters into concrete AxM designs.

Experimental results show that, compared to state-of-the-art AxM designs, TRAM reduces AxM power by up to 25.05% on CNNs with CIFAR-10 at the same accuracy level, and by 27.09% on vision transformers with ImageNet. Since TRAM allows different structure parameters for different model layers, it naturally supports layer-wise application of different AxMs. Compared to the state-of-the-art layer-wise AxM exploration methods, TRAM reduces AxM energy by 40.86%. Our work is open source and available at <https://github.com/changmg/TRAM>.

The rest of this paper is organized as follows. Section 2 describes preliminaries. Sections 3–5 detail the TRAM framework. Section 6 discusses the experimental results. Section 7 concludes this paper.

2 Integer Multiplier Preliminaries

This paper focuses on *unsigned integer multipliers* that are widely used in AI accelerators [12–15]. Hereafter, we refer to unsigned integer multipliers simply as multipliers. AxMs are usually obtained by modifying *accurate multipliers (AccMuls)*. A B -bit AccMul computes the exact product of two unsigned integer inputs W and X , which are represented in binary as $W = w_{B-1}w_{B-2} \dots w_0$ and $X = x_{B-1}x_{B-2} \dots x_0$.

The multiplier contains $2B$ accumulation columns. The c -th column accumulates the partial products as $S_c = \sum_{i=0}^c pp_{i,c-i}$, where $pp_{i,j} = w_i \cdot x_j$ is the partial product of w_i and x_j , and $0 \leq c \leq 2B-1$ is the column index. The final product is obtained by summing the weighted accumulation results from all columns as $Y = \sum_{c=0}^{2B-1} S_c \cdot 2^c$. For example, Fig. 1 shows a 4-bit array multiplier with 8 accumulation columns. Each column generates partial products and accumulates them using *half adders (HAs)* and *full adders (FAs)*. Approximation can be introduced to the partial products or to the sum and carry-out signals of the half adders and full adders in these columns, as shown by the red crosses in Fig. 1.

To evaluate the accuracy of a B -bit AxM, common error metrics include *error rate (ER)*, *normalized mean error distance (NMED)*, and *maximum error distance (MaxED)* [16], defined as

$$ER = \sum_{1 \leq i \leq 2^{2B}: Y^{(i)} \neq Y_{acc}^{(i)}} p_i, \quad NMED = \sum_{i=1}^{2^{2B}} \frac{|Y^{(i)} - Y_{acc}^{(i)}| \cdot p_i}{2^{2B} - 1},$$

$$MaxED = \max_{1 \leq i \leq 2^{2B}} |Y^{(i)} - Y_{acc}^{(i)}|.$$

where $Y^{(i)}$ and $Y_{acc}^{(i)}$ are the outputs of the AxM and the AccMul under the i -th input combination, p_i is the probability of the i -th input combination, and 2^{2B} is the total number of input combinations.

3 TRAM Overview and Multiplier Structure Parameterization

3.1 TRAM Framework Overview

TRAM aims to generate low-power AxMs for AI accelerators. The overall flow of TRAM is shown in Fig. 2. It starts from a pre-trained floating-point AI model, which is then quantized into an integer model. To further reduce power, AxMs replace the AccMuls in the quantized model. To explore the AxM design space, we represent AxM structures using the *structure parameters* collected in Θ . Changing the AxM structure corresponds to updating Θ . The detailed parameterization of Θ is presented in Section 3.2. Based on this parameterization, we propose a three-phase method to generate low-power AxMs for high-accuracy AI models:

Phase 1. Design space exploration (details in Section 4). This phase explores the AxM design space defined by Θ through model retraining and balances power and accuracy.

Phase 2. AxM structure mapping (details in Section 5). This phase maps the optimized continuous structure parameters in Θ^* from phase 1 to specific AxM structures for each layer of the model.

Phase 3. Accuracy recovery. After mapping the structure parameters to AxM structures in phase 2, we apply these AxMs to the AI model and retrain it to recover the accuracy.

3.2 AxM Structure Parameterization

We parameterize the AxM structure using continuous structure parameters, enabling gradient-based AxM structure optimization through model retraining. Let $\Theta = \{\Theta^{(l)}\} (1 \leq l \leq L)$ denote the collection of structure parameters for all L layers in the model. We assume each layer uses one AxM structure for all multiplications in that layer, described by $\Theta^{(l)}$. $\Theta^{(l)}$ has P parameters: $\Theta^{(l)} = [\theta_0^{(l)}, \theta_1^{(l)}, \dots, \theta_{P-1}^{(l)}]$ (see Fig. 1), where $\theta_c^{(l)}$ ($0 \leq c \leq P-1$) describes the approximation degree of column c . Here, P is a user-defined maximum number of columns that can be approximated.

Each $\theta_c^{(l)} \in [0, 1]$ is a continuous structure parameter that controls the approximation degree of column c in the AxM of layer l . A value of 0 means the column is kept fully accurate, while a value of 1 means that the column is entirely removed. An intermediate value $0 < \theta_c^{(l)} < 1$ represents a partial approximation, where only a subset of the partial products or compressors in column c is removed.

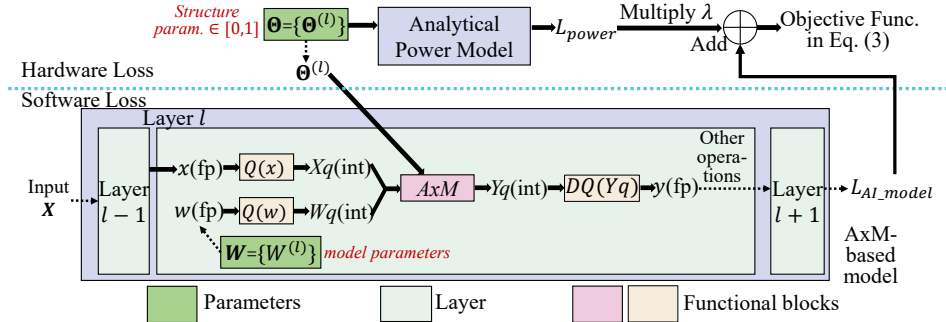


Figure 3: Dataflow for computing the objective function in Eq. (2). The upper part computes the power loss $\mathcal{L}_{power}$ (Section 4.2), and the lower part computes the AI model loss $\mathcal{L}_{AI_model}$ (Section 4.3).

Next, we explain how the structure parameters control the functional behavior of the AxM. For the l -th layer, the approximation error of the c -th accumulation column is defined as $E_c = \theta_c^{(l)} \cdot S_c = \theta_c^{(l)} \cdot \sum_{i=0}^c p p_{i,c-i}$, where S_c is the exact accumulation result of column c . The total approximation error over all P approximated columns is computed by summing the column errors multiplied by their weights 2^c , i.e., $E_{total} = \sum_{c=0}^{P-1} E_c \cdot 2^c$. The AxM output is then obtained by subtracting this error from the exact product:

$$Y = WX - E_{total} = WX - \sum_{c=0}^{P-1} \theta_c^{(l)} \cdot S_c \cdot 2^c, \quad (1)$$

where WX is the exact product of W and X . Eq. (1) enables smooth adjustment of the approximation degree in each column c by varying the structure parameter $\theta_c^{(l)}$ in the range $[0, 1]$. A larger $\theta_c^{(l)}$ leads to a larger approximation error and reduces power consumption, while a smaller $\theta_c^{(l)}$ reduces the error and increases power. This formulation can be extended beyond array multipliers.

4 Phase 1: Design Space Exploration through AI Model Retraining

4.1 Problem Formulation

The structure parameters Θ define the AxM design space. Different choices of Θ lead to AxMs with different power consumption and different AI model accuracy. To balance power consumption and accuracy, we formulate the following optimization problem:

$$\min_{\Theta, W} (\mathcal{L}_{power}(\Theta) \cdot \lambda + \mathcal{L}_{AI_model}(\Theta, W, X)). \quad (2)$$

Eq. (2) consists of two loss terms: the power loss $\mathcal{L}_{power}$ and the AI model loss $\mathcal{L}_{AI_model}$. $\mathcal{L}_{power}$ maps the structure parameters Θ to the total power consumed by all AxMs in the AI accelerator. $\mathcal{L}_{AI_model}$ is the original model loss (e.g., cross-entropy loss for classification) and depends on the structure parameters Θ , the model weights W , and the inputs X . A trade-off parameter λ is introduced to balance these two losses. Increasing λ gives more weight to $\mathcal{L}_{power}$, which lowers power consumption but increases model loss (i.e., lower model accuracy). By tuning λ , we can explore different power-accuracy trade-offs.

We solve the optimization problem in Eq. (2) through model retraining. During retraining, the dataflow for computing the objective in Eq. (2) is shown in Fig. 3. The upper part computes $\mathcal{L}_{power}$ and the lower part computes $\mathcal{L}_{AI_model}$. Sections 4.2 and 4.3 describe how $\mathcal{L}_{power}$ and $\mathcal{L}_{AI_model}$ are computed within this dataflow.

4.2 Computation of Power Loss

The power loss $\mathcal{L}_{power}(\Theta)$ estimates the power consumed by all AxMs in all layers of the AI model as follows:

$$\mathcal{L}_{power}(\Theta) = \sum_{l=1}^L f_{power}(\Theta^{(l)}) \times \frac{\text{\#mults at layer } l}{\text{\#mults in all layers}}, \quad (3)$$

where $\Theta = \{\Theta^{(l)}\} (1 \leq l \leq L)$ is the collection of structure parameters for the L layers. Here, $\Theta^{(l)}$ is the structure parameter vector for layer l , and $f_{power}(\Theta^{(l)})$ denotes the power of the AxM used in that layer. Eq. (3) is a weighted sum of the AxM power over all layers. The weight for layer l is the ratio between its number of multiplication operations and the total multiplication count of the whole model. This weight approximates the fraction of total inference latency spent on layer l . Thus, the weighted sum estimates the time-averaged AxM power during inference.

We propose an analytical method to compute $f_{power}(\Theta^{(l)})$. We first estimate the power of an AccMul as follows:

$$Power_{AccMul} = \sum_{c=0}^{2B-1} Power_c = \sum_{c=0}^{2B-1} \sum_{k=1}^{K_c} cost_{c,k} \cdot N_{c,k}. \quad (4)$$

Here, B is the multiplier bit-width, which results in $2B$ accumulation columns in the AccMul. $Power_c$ is the power of the c -th accumulation column, computed by summing the power of all K_c component types in that column. Example component types include logic AND gates for partial-product generation and various compressors for accumulation. $cost_{c,k}$ is the power of a type- k component in column c , pre-characterized using the standard cell library. $N_{c,k}$ is the number of type- k components in column c of the AccMul. For example, consider the array multiplier in Fig. 1. Column 2 contains 3 AND gates for partial-product generation, 1 HA and 1 FA for accumulation. Thus, column 2 has $K_2 = 3$ component types, labelled as type 1 (AND gate), type 2 (HA), and type 3 (FA). The number of each component type is $N_{2,1} = 3$, $N_{2,2} = 1$, and $N_{2,3} = 1$. Assume the power of each type is $cost_{2,1} = 1$, $cost_{2,2} = 2$, and $cost_{2,3} = 3$. Then, the power consumption of column 2 is $Power_2 = 1 \times 3 + 2 \times 1 + 3 \times 1 = 8$.

Recall that $\Theta^{(l)}$ contains P structure parameters $\theta_c^{(l)} (0 \leq c \leq P-1)$ for layer l . From Eq. (1), $\theta_c^{(l)} \in [0, 1]$ specifies the fraction of logic components removed from the c -th column of layer l . Based on this, we estimate the normalized AxM power for layer l as follows:

$$f_{power}(\Theta^{(l)}) = \frac{Power_{AccMul} - \sum_{c=0}^{P-1} \theta_c^{(l)} \cdot Power_c}{Power_{AccMul}}, \quad (5)$$

where P is the maximum number of approximated columns, and $Power_c$ is the power of the c -th accurate accumulation column. When $\theta_c^{(l)}$ is close to 0, most components in column c are preserved and the power remains high. Scaling $Power_c$ by $\theta_c^{(l)}$ provides an estimate of the power reduction gained from approximating that column. When $\theta_c^{(l)}$ is close to 1, most components in column c are removed and the power becomes low.

4.3 Computation of the AI Model Loss

The AI model loss function $\mathcal{L}_{AI_model}$, such as cross-entropy loss for classification tasks, measures the difference between the model output and the ground truth. It depends on the structure parameters Θ , the model parameters W , and the input data X . For illustration, we present the computation of $\mathcal{L}_{AI_model}$ in the context of a CNN.

As shown in the bottom part of Fig. 3, the forward propagation of $\mathcal{L}_{AI_model}$ processes the layers of the model in order. For CNNs, we replace the accurate multiplications in convolutional layers with AxMs, while for transformer-based models, we replace all linear layers in attention and feed-forward blocks with AxMs. In what follows, we first present how to simulate the AxM behavior. Since the AxM requires quantized inputs, we then present how to simulate the quantization process.

AxM Simulation: As shown in the center of Fig. 3, at the l -th layer, an AxM takes the integer activation X_q , the integer weight W_q , and the structure parameter vector $\Theta^{(l)}$ as inputs. The AxM produces a quantized integer output Y_q . As described in Section 3.2, $\Theta^{(l)}$ is a vector of real values in $[0, 1]$ with length P , where P is the maximum number of approximated columns. Using Eq. (1), we can express the AxM computation in closed form as:

$$Y_q = W_q X_q - \sum_{c=0}^{P-1} \left[2^c \cdot \theta_c^{(l)} \cdot \sum_{i=0}^c (W_q[i] \cdot X_q[c-i]) \right], \quad (6)$$

where $W_q[i]$ is the i -th bit of W_q , and $X_q[j]$ is the j -th bit of X_q . The term $(W_q[i] \cdot X_q[c-i])$ is the partial product of $W_q[i]$ and $X_q[c-i]$. The sum of partial products in column c is scaled by the continuous structure parameter $\theta_c^{(l)} \in [0, 1]$. If $\theta_c^{(l)} = 0$, the c -th accumulation column is kept exactly. If $\theta_c^{(l)} = 1$, the c -th accumulation column is fully removed. If $0 < \theta_c^{(l)} < 1$, logic components in column c are partially removed, and the corresponding accumulation error is estimated by scaling the exact accumulation result using $\theta_c^{(l)}$.

Quantization Simulation: Since the inputs and outputs of an AxM are integers, quantization is required before the AxM operation. We apply the traditional *fake quantization* technique [17] during training. The quantization functions $Q(x)$ and $Q(w)$, and the dequantization function $DQ(Y_q)$ in Fig. 3 follow [18].

5 Phase 2: AxM Structure Mapping

Phase 1 produces a set of continuous structure parameters $\Theta^* = \{\Theta^{*(1)}, \Theta^{*(2)}, \dots, \Theta^{*(L)}\}$, where $\Theta^{*(l)}$ corresponds to the AxM used in layer l . Let $\Theta^{*(l)} = [\theta_0^*, \theta_1^*, \dots, \theta_{P-1}^*]$ (layer index l omitted for brevity), and P is the maximum number of columns that can be approximated. These continuous structure parameters θ_c^* cannot be directly implemented in hardware.

The goal of phase 2 is to map $\Theta^{*(l)}$ into a concrete AxM netlist used in each layer l so that the resulting circuit behaves as closely

as possible to the behavior implied by $\Theta^{*(l)}$. Specifically, during training, $\Theta^{*(l)}$ controls the amount of error added to each column of the AxM. The hardware mapping aims to reproduce this same error behavior. To guide this mapping, we compute the expected AxM output under $\Theta^{*(l)}$ using the closed-form model in Eq. (6). For all input combinations, we compute the reference output:

$$Y_{ref} = W_q X_q - \sum_{c=0}^{P-1} \left[2^c \theta_c^* \sum_{i=0}^c (W_q[i] \cdot X_q[c-i]) \right]. \quad (7)$$

Eq. (7) defines the target behavior that the hardware should match. Thus, given $\Theta^{*(l)}$, our task is to construct an AxM whose output function approximates Y_{ref} as closely as possible.

To obtain such an AxM, we begin with an AccMul. In our implementation, the initial multiplier is an array-based AccMul. For each accumulation column c , the candidates for approximation are the sum and carry outputs of compressors such as HAs and FAs in column c . Each candidate can be tentatively replaced by constant 0. Such replacements act as discrete forms of the continuous error implied by $\theta_c^{*(l)}$.

Inspired by existing approximate logic synthesis methods that assess local replacements using the errors they induce [19–22], we define an error-based metric to evaluate whether a constant replacement of a candidate signal is beneficial. Let Y_{circ}^{curr} be the output of the current circuit and Y_{circ}^{new} the output after a tentative replacement. We evaluate the errors of Y_{circ}^{curr} and Y_{circ}^{new} with respect to Y_{ref} using the same input patterns. A replacement is accepted only if it strictly decreases the *mean squared error (MSE)* from the reference: $MSE(Y_{circ}^{new}, Y_{ref}) < MSE(Y_{circ}^{curr}, Y_{ref})$.

Based on this evaluation metric, we propose a greedy column-wise mapping flow. We traverse the multiplier from the least significant column to the most significant column. For each column, all approximation candidates are considered before moving to the next column. Each candidate is tentatively replaced by constant 0 and evaluated using the MSE criterion. If a tentative substitution reduces the MSE, it is permanently applied to the circuit. Otherwise, the tentative substitution is undone. After processing all columns, we obtain a circuit whose output is close to the reference output Y_{ref} . Finally, the resulting AxM netlist is emitted as Verilog. Note that this procedure always yields a feasible design, since it starts from an AccMul and only applies constant replacements.

6 Experimental Results

We implement the TRAM framework using PyTorch 2.4 [23] and test it on a single NVIDIA A100 GPU. Experiments are conducted with CNN and ViT models on the CIFAR-10 [24] and ImageNet [25] datasets. We evaluate two quantization schemes: *w8a8* (8-bit weights and activations) and *w4a4* (4-bit weights and activations), covering both standard and low-bitwidth regimes commonly used in AI accelerators. The initial *w8a8*-quantized model is obtained through post-training quantization, while the initial *w4a4*-quantized model is prepared using quantization-aware training. Channel-wise quantization is applied to the weights, and layer-wise quantization is applied to the activations.

Unless otherwise specified, all experiments use the following default settings. We use a batch size of 256 and the SGD optimizer with momentum 0.9 and weight decay $5e-4$. The retraining epochs for design space exploration (phase 1) and accuracy recovery (phase

Table 1: Tested 8-bit unsigned multipliers.

Multiplier	Area / μm^2	Delay /ps	Power /mW	ER/%	NMED/%	MaxED
AccMul	27.2	496.5	0.0031	0.0	0.000	0
OPACT_1	18.6	499.6	0.0019	23.0	0.022	516
OPACT_18	14.1	495.1	0.0011	51.0	0.619	7684
Evo_0AB	21.3	499.6	0.0022	97.7	0.057	115
Evo_1DMU	16.9	499.2	0.0015	66.0	0.650	4084
Evo_GJM	12.1	499.5	0.0010	74.9	1.543	9124
AMPPS_S2	17.0	499.9	0.0019	96.3	0.076	417
AMPPS_S3	18.9	497.9	0.0020	95.9	0.073	417
AMPPS_S4	20.8	499.7	0.0022	95.7	0.070	417

3) are both set to 10, which we found sufficient for convergence. Phase 1 uses a fixed learning rate of $5e-4$, while phase 3 uses a cosine annealing schedule that decreases the learning rate from $5e-4$ to 0. The parameter P (see Section 3.2), the maximum number of accumulation columns allowed to be approximated, is set to 8. Initially, the AxM in each layer l removes 4 accumulation columns by setting $\theta_c^{(l)} = 1$ for $c = 0, 1, 2, 3$ and $\theta_c^{(l)} = 0$ for all other columns, serving as a moderate starting point for the optimizer. To evaluate area, delay, and power, we synthesize the AxMs using a commercial logic synthesis tool with the ASAP 7nm standard cell library [26]. Power measurements assume a 100 MHz clock frequency.

We evaluate TRAM under two scenarios: 1) all layers in the AI model share the same AxM type, to compare individual AxM designs against baselines, and 2) different layers can use different AxM types, to evaluate the benefit of layer-wise optimization.

6.1 Experiments with a Single AxM Type

This set of experiments assumes that all layers in the AI model use the same type of AxM. To achieve this in TRAM, we restrict the structure parameters of all layers to be identical, i.e., $\Theta^{(1)} = \Theta^{(2)} = \dots = \Theta^{(L)}$, where L is the number of layers. The baseline AxM designs are “Evo”, “OPACT”, and “AMPPS”, taken from [6], [7], and [8], respectively. The tested 8-bit unsigned multipliers and their errors, areas, and delays are listed in Table 1. The ER, NMED, and MaxED metrics of the AxMs (see Section 2) are measured by enumerating all possible input combinations under a uniform distribution. We use the open-source tool in [27] to obtain the ER and NMED metrics. For a fair comparison, for each Evo and OPACT AxM, we use the same number of epochs of AxM-aware retraining as TRAM. We reimplement this retraining process following the method in [28]. However, the AMPPS AxMs include an encoding process that is not supported by the existing AxM-aware retraining methods. Thus, we do not perform retraining for the AMPPS AxMs, and we directly take the final CNN accuracy reported in [8].

6.1.1 Experiments on CIFAR-10. This set of experiments compares the CNN accuracy and the AxM power consumption on CIFAR-10. We first use the w8a8 quantization scheme and test the ResNet18 and ResNet34 models. Fig. 4 plots the final model accuracy versus the AxM power consumption for the tested AxMs on ResNet18 and ResNet34. Each figure includes the 8-bit AccMul result in the upper right corner for reference. A better AxM achieves higher accuracy and lower power, which appears toward the upper left region of the plot. We vary the trade-off parameter λ in Eq. (2) to generate different AxM designs using TRAM. As λ increases, both

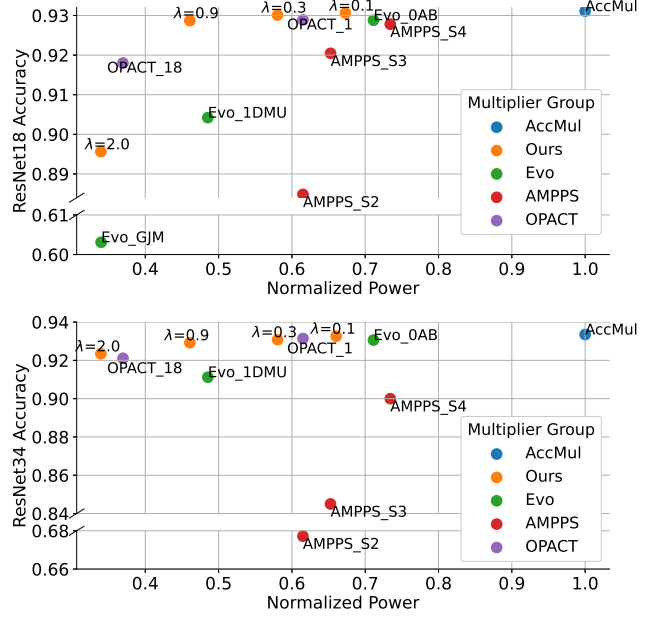


Figure 4: Comparison of final accuracy and AxM power consumption using different 8-bit multipliers on CIFAR-10.

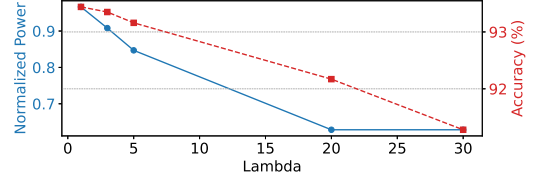


Figure 5: Impact of λ on DenseNet161 accuracy and AxM power under w4a4. Power is normalized to the 4-bit AccMul.

the AxM power consumption and the accuracy decrease. The results show that TRAM achieves a better trade-off between accuracy and power consumption than the Evo, AMPPS, and OPACT AxMs for both ResNet18 and ResNet34 models. For example, compared with OPACT_1 on ResNet18, TRAM with $\lambda = 0.9$ reaches a similar accuracy, while reducing power by 15.44%. On ResNet34, TRAM with $\lambda = 0.9$ reaches a similar accuracy as Evo_0AB and OPACT_1, while reducing power by 25.05% and 15.43%, respectively.

As for runtime, TRAM includes two retraining phases (phases 1 and 3 in Fig. 2) and one AxM structure mapping phase (phase 2). Phase 2 uses a greedy algorithm and takes less than 1 minute to generate an 8-bit AxM. The phase 2 runtime is independent of the CNN model size. With 20 training epochs in total (10 epochs for phase 1 and 10 epochs for phase 3), TRAM takes about 17 and 30 minutes for ResNet18 and ResNet34, respectively.

We also evaluate the impact of λ on the DenseNet161 model under the w4a4 quantization scheme. The maximum number of approximated columns, P , is set to 4. The results are shown in Fig. 5. Similarly to previous results, increasing λ reduces the AxM power and model accuracy. The runtime of TRAM for DenseNet161 under w4a4 is about 1.66 hours for 20 training epochs (phases 1 and 3).

6.1.2 Experiments on ImageNet. This set of experiments evaluates TRAM on the ImageNet dataset [25]. The baseline method is

Table 2: Results on ImageNet with vision transformers.

Model	AxM	Accuracy	Norm. AxM Power
DeiT-S (8-bit AccMul acc. 79.34%)	Ours ($\lambda=1$)	76.78%	82.53%
	Ours ($\lambda=100$)	76.31%	66.08%
	Evo_1KV9	70.16%	93.17%
	Evo_1L2H	67.01%	73.04%
Swin-S (8-bit AccMul acc. 81.83%)	Ours ($\lambda=1$)	79.54%	82.53%
	Ours ($\lambda=100$)	79.15%	66.08%
	Evo_1KV9	79.25%	93.17%
	Evo_1L2H	76.64%	73.04%

TransAxx [29], an AxM-aware retraining method for transformer models. TransAxx tested the power-accuracy trade-off of several 8-bit AxMs from the EvoApprox library [6], and we compare the TRAM-generated AxMs with these designs.

We refer to the TransAxx training settings, which use up to 15 epochs for AxM-aware training. We perform phase 1 (design space exploration) of TRAM for 5 epochs and phase 3 (accuracy recovery) for 10 epochs. Following the TransAxx setup, we sample 100,000 images from the 1.28 million training images for efficient training, and use the full validation set for evaluation. We use the Adam optimizer with a learning rate of $5e-5$ for both phases and set the batch size to 64. Note that TRAM uses channel-wise weight quantization and layer-wise activation quantization, whereas TransAxx uses layer-wise quantization for both weights and activations. Therefore, the comparison with TransAxx should be viewed as a reference rather than a strictly matched apples-to-apples comparison. To save runtime, we set the parameter P , which is the maximum number of approximated columns, to 6.

Table 2 compares TRAM-generated AxMs with the Evo_1KV9 and Evo_1L2H AxMs from the EvoApprox library. We copy the accuracy results of these designs from the TransAxx paper and measure their power using the ASAP 7nm library. Other AxMs tested in TransAxx are excluded due to their impractically large accuracy loss. Table 2 suggests that TRAM provides a favorable accuracy-power trade-off under our setting. For DeiT-S and Swin-S, using $\lambda = 1$ keeps the accuracy acceptable compared to that of the 8-bit quantization model using AccMuls, while reducing AxM power by 17.47%. Increasing λ to 100 provides further power reduction with a small decrease in accuracy. In contrast, Evo_1KV9 consumes much more power, while Evo_1L2H achieves lower power at the cost of a much larger accuracy loss. For Swin-S, comparing our AxM when $\lambda = 100$ with Evo_1KV9, TRAM achieves comparable accuracy and reduces power by an additional 27.09%.

6.2 Experiments with Layer-Wise Different AxM Types

This set of experiments assumes that different layers can use different types of AxMs. To support this in TRAM, we allow structure parameters to differ across layers, i.e., $\Theta^{(1)}, \Theta^{(2)}, \dots, \Theta^{(L)}$ can be different. We compare TRAM with existing mixed-type AxM selection frameworks for CNNs, i.e., MARLIN [30] and ALWANN [31]. Both MARLIN and ALWANN select different AxM types across layers to improve energy efficiency. We use data reported in the MARLIN paper, which also includes results for ALWANN. Although

Table 3: Comparison of ResNet50 accuracy, normalized AxM energy, and runtime using different layer-wise AxM exploration methods on CIFAR-10. “N/A” means not applicable.

Method	ResNet50 Accuracy	Norm. Energy	Runtime /hour
FP32	93.65%	N/A	N/A
AccMul	93.56%	100.00%	N/A
Ours ($\lambda=1$)	93.71%	74.97%	0.69
Ours ($\lambda=10$)	93.65%	55.64%	0.70
Ours ($\lambda=100$)	92.79%	35.90%	0.69
Ours ($\lambda=1000$)	92.97%	35.81%	0.69
MARLIN-1	92.14%	80.67%	111.1
MARLIN-2	91.70%	76.67%	111.1
ALWANN-1	89.08%	78.47%	N/A
ALWANN-2	88.58%	70.02%	N/A

MARLIN is evaluated in a 65 nm technology and our evaluation is based on the ASAP 7nm library, the AxM energy normalized to the AccMul energy still provides a meaningful basis for comparison.

Table 3 compares TRAM with MARLIN and ALWANN on CIFAR-10 using the ResNet50 model and the w8a8 quantization scheme. The AxM energy consumption refers to the energy consumed by all AxMs during inference of one input image. It is estimated by accumulating the energy of approximate multiplications in convolutional layers. The normalized AxM energy is estimated from number of multiplications and AxM power in each layer. Since the tested 8-bit multipliers have nearly identical latency, we assume the same delay for all multipliers, so the delay term cancels out in the normalization. Therefore, the normalized energy is computed from $\sum_l \#mults^{(l)} \times AxMPower^{(l)}$, normalized to the 8-bit AccMul.

From Table 3, our method consistently outperforms both MARLIN and ALWANN in terms of accuracy and energy efficiency. When $\lambda = 1$, our method reaches 93.71% accuracy, which slightly exceeds the baseline 8-bit AccMul accuracy of 93.56%. Meanwhile, AxM energy consumption is reduced by 25.03%. When $\lambda = 1000$, our method achieves 92.97% accuracy with 35.81% normalized energy. Comparing the case of $\lambda = 1000$ with MARLIN-2, TRAM improves accuracy by 1.27% while reducing energy consumption by 40.86%. One reason for the large energy savings is that TRAM explores a larger AxM design space. With the parameterization in Section 3.2 and the mapping in Section 5, TRAM applies constant-0 replacements to HA and FA sum and carry outputs. In contrast, MARLIN explores a more restricted design space based on removing entire columns of partial products. Comparing runtime, TRAM requires 0.7 hours on a single NVIDIA A100 GPU for the 20 training epochs, while MARLIN consumes over 111 hours using 16 threads of a Ryzen 5950X CPU and an NVIDIA RTX A5000 GPU.

7 Conclusion

In conclusion, to achieve a good trade-off between power consumption and accuracy of AxM-based AI accelerators, we propose TRAM to directly explore the AxM design space through model retraining. At the same accuracy level, TRAM significantly reduces AxM power on several CNNs and vision transformers on the CIFAR-10 and ImageNet datasets. In the future, we will extend TRAM to large language models.

References

- [1] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. Green AI. *Communications of the ACM*, 63(12):54–63, 2020.
- [2] Vasileios Leon, Muhammad Abdullah Hanif, Giorgos Armeniakos, Xun Jiao, Muhammad Shafique, Kiamal Pekmestzi, and Dimitrios Soudris. Approximate computing survey, part II: Application-specific & architectural approximation techniques and applications. *ACM Computing Surveys*, 57(7):1–36, 2025.
- [3] Giorgos Armeniakos, Georgios Zervakis, Dimitrios Soudris, and Jörg Henkel. Hardware approximate techniques for deep neural network accelerators: A survey. *ACM Computing Surveys (CSUR)*, 55(4):1–36, 2022.
- [4] Ying Wu, Chuangtao Chen, Weihua Xiao, Xuan Wang, Chenyi Wen, Jie Han, Xunzhao Yin, Weikang Qian, and Cheng Zhuo. A survey on approximate multiplier designs for energy efficiency: From algorithms to circuits. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 29(1):1–37, 2024.
- [5] Vojtech Mrazek, Radek Hrbacek, Zdenek Vasicek, and Lukas Sekanina. EvoApprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 258–261, 2017.
- [6] Vojtech Mrazek, Lukas Sekanina, and Zdenek Vasicek. Libraries of approximate circuits: Automated design and application in CNN accelerators. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 10(4):406–418, 2020.
- [7] Weihua Xiao, Cheng Zhuo, and Weikang Qian. OPACT: Optimization of approximate compressor tree for approximate multiplier. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 178–183, 2022.
- [8] Xiaolu Hu, Ao Liu, Xinkuang Geng, Zizhong Wei, Kai Jiang, and Honglan Jiang. A configurable approximate multiplier for CNNs using partial product speculation. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2024.
- [9] Xuan Wang, Zheyu Yan, Chang Meng, Yiyu Shi, and Weikang Qian. DASALS: Differentiable architecture search-driven approximate logic synthesis. In *International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2023.
- [10] Jingxiao Ma, Soheil Hashemi, and Sherief Reda. Approximate logic synthesis using Boolean matrix factorization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 41(1):15–28, 2021.
- [11] Chang Meng, Weikang Qian, and Giovanni De Micheli. Simulation-guided approximate logic synthesis under the maximum error constraint. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2026.
- [12] William Andrew Simon, Valérien Ray, Alexandre Levisse, Giovanni Ansaloni, Marina Zapater, and David Atienza. Exact neural networks from inexact multipliers via Fibonacci weight encoding. In *Design Automation Conference (DAC)*, pages 805–810, 2021.
- [13] Paras Jain, Safeen Huda, Martin Maas, Joseph E Gonzalez, Ion Stoical, and Azalia Mirhoseini. Learning to design accurate deep learning accelerators with inaccurate multipliers. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 184–189, 2022.
- [14] Chang Meng, Wayne Burleson, Weikang Qian, and Giovanni De Micheli. Gradient approximation of approximate multipliers for high-accuracy deep neural network retraining. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–7, 2025.
- [15] Su Zheng, Zhen Li, Yao Lu, Jingbo Gao, Jide Zhang, and Lingli Wang. HEAM: High-efficiency approximate multiplier optimization for deep neural networks. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 3359–3363, 2022.
- [16] Honglan Jiang, Francisco Javier Hernandez Santiago, Hai Mo, Leibo Liu, and Jie Han. Approximate arithmetic circuits: A survey, characterization, and recent applications. *Proceedings of the IEEE*, 108(12):2108–2135, 2020.
- [17] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2704–2713, 2018.
- [18] Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang, Peng Gao, Yu Qiao, and Ping Luo. OmniQuant: Omnidirectionally calibrated quantization for large language models. *International Conference on Learning Representations (ICLR)*, pages 1–25, 2024.
- [19] Chang Meng, Weikang Qian, and Alan Mishchenko. ALSRAC: Approximate logic synthesis by resubstitution with approximate care set. In *Design Automation Conference (DAC)*, pages 1–6, 2020.
- [20] Chang Meng, Zhuangzhuang Zhou, Yue Yao, Shuyang Huang, Yuhang Chen, and Weikang Qian. HEDALS: Highly efficient delay-driven approximate logic synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 42(11):3491–3504, 2023.
- [21] Soheil Hashemi, Hokchhay Tann, and Sherief Reda. BLASYS: Approximate logic synthesis using boolean matrix factorization. In *Design Automation Conference (DAC)*, pages 1–6, 2018.
- [22] Chang Meng, Alan Mishchenko, Weikang Qian, and Giovanni De Micheli. Efficient resubstitution-based approximate logic synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 44(6):2040–2053, 2025.
- [23] Adam Paszke, Sam Gross, et al. PyTorch: An imperative style, high-performance deep learning library. In *International Conference on Neural Information Processing Systems (NeurIPS)*, pages 8026–8037, 2019.
- [24] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. *Technical Report, University of Toronto*, 2009.
- [25] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009.
- [26] Lawrence T Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandrasekaran Ramamurthy, and Greg Yeric. ASAP7: A 7-nm FinFET predictive process design kit. *Microelectronics Journal*, 53:105–115, 2016.
- [27] Chang Meng, Hanyu Wang, Yuqi Mai, Weikang Qian, and Giovanni De Micheli. VECSEM: Verifying average errors in approximate circuits using simulation-enhanced model counting. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2024.
- [28] Dimitrios Danopoulos, Georgios Zervakis, Kostas Siozios, Dimitrios Soudris, and Jörg Henkel. AdaPT: Fast emulation of approximate DNN accelerators in PyTorch. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 42(6):2074–2078, 2022.
- [29] Dimitrios Danopoulos, Georgios Zervakis, Dimitrios Soudris, and Jörg Henkel. TransAxx: Efficient transformers with approximate computing. *IEEE Transactions on Circuits and Systems for Artificial Intelligence*, 2(4):288–301, 2025.
- [30] Flavia Guella, Emanuele Valpreda, Michele Caon, Guido Masera, and Maurizio Martina. MARLIN: A co-design methodology for approximate reconfigurable inference of neural networks at the edge. *IEEE Transactions on Circuits and Systems I: Regular Papers (TCAS-I)*, 71(5):2105–2118, 2024.
- [31] Vojtech Mrazek et al. ALWANN: Automatic layer-wise approximation of deep neural network accelerators without retraining. In *International Conference on Computer Aided Design (ICCAD)*, pages 1–8, 2019.