Gradient Approximation of Approximate Multipliers for High-Accuracy Deep Neural Network Retraining

Chang Meng[1], Wayne Burleson[2], Weikang Qian[3,4], and Giovanni De Micheli[1]
[1]Integrated Systems Laboratory, EPFL, Lausanne, Switzerland, [2]University of Massachusetts, Amherst, USA
[3]UM-SJTU Joint Institute and [4]MoE Key Lab of AI, Shanghai Jiao Tong University, Shanghai, China
Emails: chang.meng@epfl.ch, burleson@umass.edu, qianwk@sjtu.edu.cn, giovanni.demicheli@epfl.ch

***Abstract*—Approximate multipliers (AppMults) are widely employed in deep neural network (DNN) accelerators to reduce the area, delay, and power consumption. However, the inaccuracies of AppMults degrade DNN accuracy, necessitating a retraining process to recover accuracy. A critical step in retraining is computing the gradient of the AppMult, *i.e.*, the partial derivative of the approximate product with respect to each input operand. Conventional methods approximate this gradient using that of the accurate multiplier (AccMult), often leading to suboptimal retraining results, especially for AppMults with relatively large errors. To address this issue, we propose a difference-based gradient approximation of AppMults to improve retraining accuracy. Experimental results show that compared to the state-of-the-art methods, our method improves the DNN accuracy after retraining by 4.10% and 2.93% on average for the VGG and ResNet models, respectively. Moreover, after retraining a ResNet18 model using a 7-bit AppMult, the final DNN accuracy does not degrade compared to the quantized model using the 7-bit AccMult, while the power consumption is reduced by 51%.**

***Index Terms*—Gradient, approximate multiplier, deep neural network, retraining**

I. Introduction

Modern *artificial intelligence* (*AI*) technologies excel in a wide range of areas such as natural language processing and computer vision, driving widespread adoption of *deep neural network* (*DNN*) accelerators in edge devices, cloud computing systems, *etc*. However, this rapid growth raises serious concerns about power consumption [1].

To achieve energy-efficient DNN accelerators, researchers have adopted an emerging design paradigm called *approximate computing*, which reduces power consumption at the cost of small errors [2], [3]. Approximate computing is particularly suitable for DNN accelerators, since DNNs are inherently resilient to errors and noise. By carefully introducing errors into a DNN accelerator, the final output quality is almost unaffected, while the area, delay, and power consumption of the accelerator can be significantly reduced [4]. Generally speaking, traditional DNN compression techniques with low-precision data representations, such as int4 [5] and float8 [6], can also be viewed as approximate computing techniques.

Among various approximation techniques for DNN accelerators, designs based on *approximate multiplier (AppMult)* are among the most popular [4], [7]. Fig. 1 shows a typical design flow for AppMult-based DNN accelerators. It begins with a pre-trained floating-point DNN model, followed by quantization to convert the model into a version with integer weights and activations. Since integer multipliers are the most power-consuming components in a quantized DNN, they are approximated to reduce power consumption. However, the inaccuracies of AppMults degrade the DNN accelerator's accuracy, necessitating AppMult-aware retraining to recover the accuracy [8]–[13].

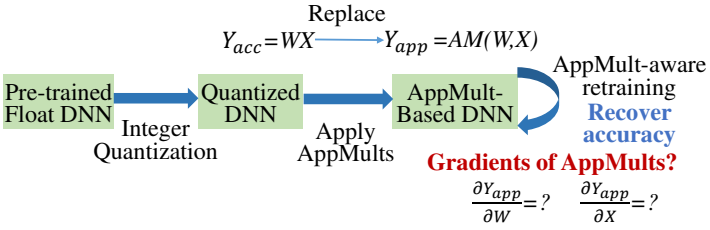


Fig. 1: Design flow of AppMult-based DNN accelerators.

A basic question in AppMult-aware retraining is how to compute the gradient of an AppMult, *i.e.*, the partial derivative of the approximate product Y_{app} *with respect to (w.r.t.)* each input operand, *i.e.*, W or X. Previous works proposed a *straight through estimator (STE)*, which approximates the gradient of an AppMult using that of an *accurate multiplier (AccMult)* [8]–[13]. It is effective for small-error AppMults, as their approximation errors are small, resulting in gradients that closely match those of AccMults. However, in scenarios requiring aggressive power reduction, AppMults with relatively large errors are often preferred. For these AppMults, their gradients can deviate significantly from those of AccMults. In such cases, the STE fails to provide accurate gradients, and retraining with STE-based gradients may lead to suboptimal results. To address this issue, we propose a more precise gradient approximation of AppMults, aiming to improve retraining accuracy, especially for AppMults with relatively large errors. Our main contributions are as follows:

- We propose a difference-based gradient approximation for AppMults to improve the retraining accuracy.
- We develop an AppMult-aware DNN retraining framework that incorporates our proposed gradient approximation.
- Our method improves the retraining accuracy by 4.10% for VGG models and 2.93% for ResNet models, alleviating ac-

This work is supported by the Swiss National Science Foundation Grant "Supercool: Design methods and tools for superconducting electronics" with funding number 200021_1920981 and a grant from Synopsys Inc. Corresponding author: Chang Meng.

curacy degradation from AppMults. For a ResNet18 model with a 7-bit AppMult, it reduces multiplier power consumption by 51% without accuracy loss, showing its effectiveness for designing energy-efficient DNN accelerators.

Our AppMult-aware retraining framework is available at https://github.com/changmg/AppMult-Aware-Retraining.

The remainder of the paper is organized as follows. Section II introduces the preliminaries of AppMult-aware retraining. Sections III and IV present our gradient approximation of AppMults for high-accuracy DNN retraining. Section V shows the experimental results. Section VI concludes the paper.

II. Preliminaries

A. Approximate Multipliers (AppMults)

This paper focuses on *integer AppMults*, which are commonly used in DNN accelerators [14]–[16]. In what follows, we refer to them as AppMults for short. A general AppMult with input operands W and X and output Y implements the following function:

$$Y = AM(W, X) = WX + \epsilon(W, X), \tag{1}$$

where WX is the exact product, AM is the AppMult function, and $\epsilon(W, X)$ is the approximation error. For example, Fig. 2 shows a 7-bit unsigned AppMult, which removes the rightmost 6 columns of partial products. Its approximation error is $\epsilon(W, X) = -\sum_{i=0}^{5}\sum_{j=0}^{5-i}\left(2^{i+j}pp_{ij}\right)$, where pp_{ij} is the partial product of w_i (i-th bit of W) and x_j (j-th bit of X).

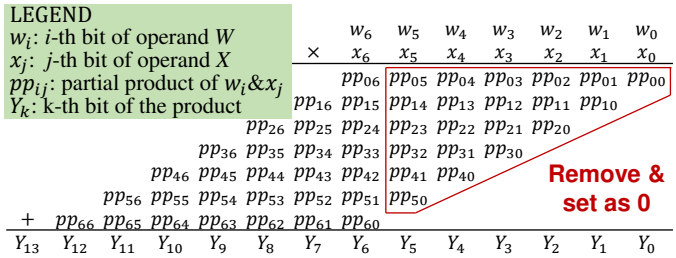


Fig. 2: A simple 7-bit unsigned AppMult, where the rightmost 6 columns of partial products are removed [17].

To evaluate the accuracy of a B-bit AppMult, common error metrics include *error rate (ER)*, *normalized mean error distance (NMED)*, and *maximum error distance (MaxED)* [7], defined as follows:

$$\begin{aligned} ER &= \sum_{1\le i\le 2^{2B}: Y^{(i)}\neq Y_{acc}^{(i)}} p_i, \\ NMED &= \sum_{i=1}^{2^{2B}} \frac{|Y^{(i)} - Y_{acc}^{(i)}|\cdot p_i}{2^{2B}-1}, \\ MaxED &= \max_{1\le i\le 2^{2B}} |Y^{(i)} - Y_{acc}^{(i)}|, \end{aligned} \tag{2}$$

where $Y^{(i)}$ and $Y_{acc}^{(i)}$ are the outputs of the AppMult and the *accurate multiplier (AccMult)*, respectively, 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.

B. AppMult-Aware DNN Retraining

As shown in Fig. 1, AppMult-aware DNN retraining is used to recover DNN accuracy after applying AppMults. Mainstream AppMult-aware retraining techniques rely on gradient descent and consist of two key steps: forward propagation and backward propagation.

During the forward propagation, the input data is processed through the DNN to compute the output. In this step, AppMults are simulated to perform approximate multiplications, typically through *lookup table (LUT)*-based methods (*e.g.*, [9]–[11]) or behavioral-level simulations (*e.g.*, [12]).

During the backward propagation, the gradients of the loss function *w.r.t.* the DNN parameters are computed, and the parameters are updated by descending along the negative gradient direction. This step involves computing the gradients of AppMults, and to the best of our knowledge, all existing AppMult-aware DNN retraining frameworks utilize the *straight-through estimator (STE)* to estimate the gradient of AppMults [8]–[13]. Specifically, these frameworks approximate the gradient of an AppMult using that of an AccMult. For a general AppMult in Eq. (1), the STE method estimates its gradient as follows:

$$\frac{\partial AM}{\partial W} \approx X, \frac{\partial AM}{\partial X} \approx W. \tag{3}$$

In other words, STE assumes that the gradients of the approximation error, $\frac{\partial \epsilon}{\partial W}$ and $\frac{\partial \epsilon}{\partial X}$, are 0. The approach is effective when an AppMult has a small error (*i.e.*, ϵ is close to 0). However, when an AppMult exhibits a relatively large error, its gradient can deviate significantly from that of the AccMult. In this case, STE-based retraining may yield suboptimal results due to inaccurate gradients. To address this, our work proposes a more precise gradient approximation of AppMults to improve retraining accuracy.

III. Difference-Based Gradient Approximation of Approximate Multipliers

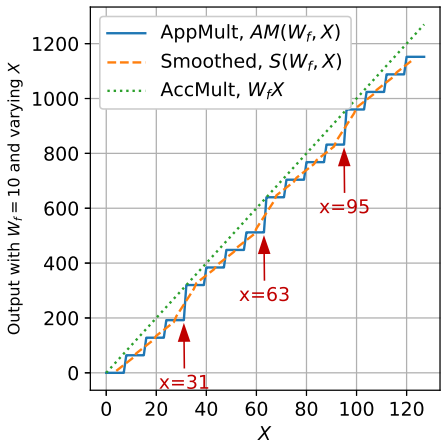
This section presents our proposed gradient approximation of AppMults for high-accuracy DNN retraining. For a general AppMult with a function $AM(W, X)$, we propose to approximate its gradients $\frac{\partial AM}{\partial W}$ and $\frac{\partial AM}{\partial X}$ by the following two steps:

1) Smooth the AppMult function, detailed in Section III-A.
2) Compute the difference-based gradient, detailed in Section III-B.

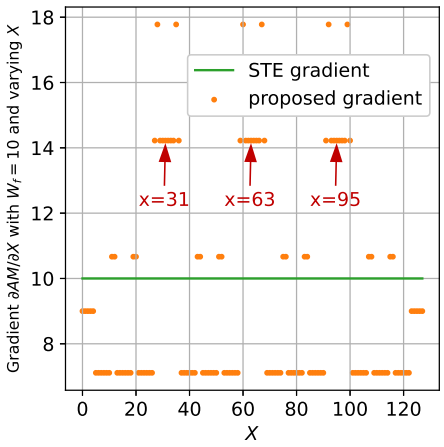
For clarity, we assume that the AppMult is unsigned in this section, although our method can be easily extended to signed AppMults. Additionally, we will only focus on approximating $\frac{\partial AM}{\partial X}$ in the following discussion, noting that the method for $\frac{\partial AM}{\partial W}$ is similar. To compute $\frac{\partial AM}{\partial X}$, we analyze the function $AM(W_f, X)$, where W_f represents a fixed value of W.

A. Smoothing AppMult Function

Since the least significant part of AppMults is often approximated or even removed, $AM(W_f, X)$ may exhibit a stair-like behavior *w.r.t.* X (see the blue curve in Fig. 3(a)). This makes $\frac{\partial AM}{\partial X}$ zero for most X and produces significant large $\frac{\partial AM}{\partial X}$ at the stair edges. This behavior is not ideal for gradient



(a) The AppMult function, the smoothed AppMult function using a half window size of 4, and the AccMult function when $W_f = 10$.



(b) The proposed difference-based gradient and the STE-based gradient of the AppMult when $W_f = 10$.

Fig. 3: Smoothing of a 7-bit unsigned AppMult function $AM(W_f = 10, X)$ and its gradient approximation. The AppMult corresponds to the one in Fig. 2, which removes the 6 rightmost columns of partial products. The red arrows show three relatively large changes in the AppMult function, corresponding to the large values in the difference-based gradient.

descent, as zero gradients for most X can prohibit the DNN parameters from being updated, while the large gradients at the stair edges can destabilize the gradient descent process. As a result, directly using the real gradient of $AM(W_f, X)$ in retraining can lead to suboptimal results. To address this issue, we propose to smooth $AM(W_f, X)$ into a new value $S(W_f, X)$ using moving average as follows:

$$S(W_f, X) = \frac{1}{2HWS+1} \sum_{\Delta x=-HWS}^{HWS} AM(W_f, X+\Delta x), \quad \text{for } HWS \le X \le 2^B - 1 - HWS, \tag{4}$$

where B is the bitwidth and HWS is a positive integer called *half window size*. For each input pair (W_f, X), Eq. (4) considers its neighbor points within a window of size $(2HWS+1)$, and calculates the average of the AppMult outputs in this window to produce the smoothed output $S(W_f, X)$. For example, the orange curve in Fig. 3(a) shows the smoothed function $S(W_f = 10, X)$ for the AppMult function $AM(W_f = 10, X)$ (shown in blue) with $HWS = 4$. After smoothing, $S(W_f = 10, X)$ has no zero gradients and no large gradients, making it more suitable for gradient descent. Note that Eq. (4) is only applied to $HWS \le X \le 2^B - 1 - HWS$. This is because only $S(W_f, X)$ for $HWS \le X \le 2^B - 1 - HWS$ is used in our gradient approximation, which will be shown in Section III-B.

B. Difference-Based Gradient Computation

After smoothing the function $AM(W_f, X)$ to $S(W_f, X)$, we propose to approximate the gradient of $S(W_f, X)$ *w.r.t.* X using the following difference-based method:

$$\frac{AM(W_f, X)}{\partial X} \approx \frac{S(W_f, X)}{\partial X} \approx \frac{S(W_f, X+1) - S(W_f, X-1)}{2}, \quad \text{for } HWS < X < 2^B - 1 - HWS. \tag{5}$$

Eq. (5) considers the two neighboring points of (W_f, X), *i.e.*, $(W_f, X+1)$ and $(W_f, X-1)$, and uses the slope between them to approximate the gradient.

Note that Eq. (5) is only applied to $HWS < X < 2^B - 1 - HWS$. For the other values of X, we estimate the gradient as:

$$\frac{AM(W_f, X)}{\partial X} \approx \frac{\max_X AM(W_f, X) - \min_X AM(W_f, X)}{2^B}, \quad \text{for } 0 \le X \le HWS \text{ and } 2^B - 1 - HWS \le X < 2^B. \tag{6}$$

Eq. (6) computes the maximum and minimum values of $AM(W_f, X)$ for $X \in [0, 2^B - 1]$, which is the total change of $AM(W_f, X)$ for all possible values of X under the fixed W_f. This total change is then divided by 2^B to obtain the average change per unit X as the gradient approximation.

For instance, using Eqs. (5) and (6) with $W_f = 10$ and $HWS = 4$, for the AppMult function $AM(W_f = 10, X)$ (blue curve in Fig. 3(a)), its difference-based gradient is shown in Fig. 3(b) in orange. For comparison, the STE-based gradient is plotted in green in Fig. 3(b), showing a constant value of 10 for all values of X, as $W_f = 10$. The magnitude of the difference-based gradient reflects the rate of change of the AppMult function $AM(W_f, X)$ *w.r.t.* X, where larger gradient values indicate greater changes of $AM(W_f, X)$ per unit X. For example, $AM(W_f = 10, X)$ has three relatively large changes at $X = 31$, 63, and 95, indicated by the red arrows in Fig. 3(a). Correspondingly, the difference-based gradient in Fig. 3(b) exhibits large values around these points. In contrast, the STE-based gradient is always 10 for all values of X, failing to capture the variations in the changing rate of $AM(W_f, X)$. Therefore, the difference-based gradient offers better guidance for the gradient descent process, potentially improving retraining accuracy.

IV. AppMult-Aware Retraining Framework Using the Proposed Gradient Approximation

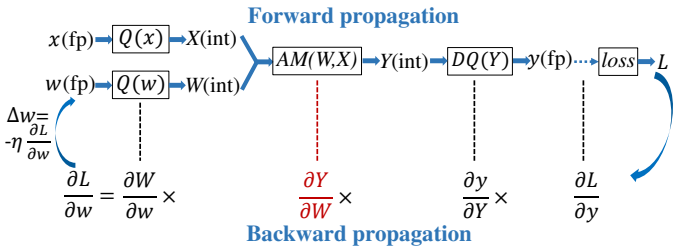


Fig. 4: Forward and backward propagation in the AppMult-aware DNN retraining framework. The term "fp" denotes "floating point". The functions Q and DQ are the quantization and dequantization functions, defined in Eqs. (7) and (8), respectively. The red part indicates where the proposed gradient approximation is applied. The computation of $\frac{\partial L}{\partial x}$ follows a similar approach to that of $\frac{\partial L}{\partial w}$.

We develop an AppMult-aware DNN retraining framework to incorporate the proposed difference-based gradient approximation for AppMults. As shown in Fig. 4, the framework involves a forward propagation and a backward propagation.

During the forward propagation, since the DNN operates with integer AppMults, we simulate both quantization and AppMult behaviors using the method in [18]. For quantization simulation, we apply the traditional *fake quantization* technique [19]. As shown in the top of Fig. 4, the floating-point weight w and activation x are quantized into integers W and X using a quantization function Q. For example, a simple uniform quantization is applied in our framework (other quantization methods can also be used), which is defined as:

$$W = Q(w) = \text{round}(\frac{w}{s_w} + Z_w), X = Q(x) = \text{round}(\frac{x}{s_x} + Z_x), \tag{7}$$

where s_w and s_x are the floating-point quantization scales for the weights and activations, and Z_w and Z_x are their respective integer zero points. Once the AppMult function $Y = AM(W, X)$ is computed, the dequantization function DQ converts the integer Y back into floating-point value y, defined as follows:

$$y = DQ(Y) = s_w s_x (Y - Z_x W - Z_w X + Z_w Z_x). \tag{8}$$

To simulate the behavior of AppMults, *i.e.*, the function $AM(W, X)$ as shown in the top middle of Fig. 4, we utilize a LUT-based method similar to those used in other frameworks [9]–[11], [18]. Specifically, we precompute $AM(W, X)$ for all possible input combinations of W and X, and store the results in LUTs. Then, we implement CUDA [20] kernels that compute the AppMults by referencing the precomputed LUTs. In modern DNN accelerators, state-of-the-art solutions utilize a bit-width no more than 8 [21], ensuring LUT sizes remain manageable and can be stored in the GPU memory. For example, a 7-bit AppMult LUT has 2^{14} entries, and storing these using 16-bit integers requires 32MB memory, which can be put in the fast shared memory in modern GPUs.

During the backward propagation, the gradients of the loss function L *w.r.t.* the weight w and activation x are computed. The bottom part of Fig. 4 illustrates the computation of $\frac{\partial L}{\partial w}$ using the following formula:

$$\frac{\partial L}{\partial w} = \frac{\partial W}{\partial w}\frac{\partial Y}{\partial W}\frac{\partial y}{\partial Y}\frac{\partial L}{\partial y} = Q'(w)\frac{\partial AM}{\partial W}DQ'(Y)\frac{\partial L}{\partial y}. \tag{9}$$

The computations of $Q'(w)$, $DQ'(Y)$, and $\frac{\partial L}{\partial y}$ follow the same approach in [9]–[11]. However, the computation of $\frac{\partial AM}{\partial W}$ is different, as it utilizes our proposed difference-based gradient approximation. To compute $\frac{\partial AM}{\partial W}$, we propose a LUT-based method. Specifically, we precompute the difference-based gradients of AppMults using Eqs. (5) and (6) for all possible combinations of W and X, and store them in LUTs. Then, we implement CUDA kernels that compute the gradients by looking up these LUTs. Note that our framework can also accommodate other user-defined gradients of AppMults, enabling the exploration of the effects of various gradient approximations on the retraining process.

V. Experimental Results

A. Experimental Setup

We implemented our AppMult-aware DNN retraining framework in PyTorch 2.4 [22] and CUDA 12.4. The framework supports arbitrary user-defined gradients for AppMults, and in our experiments, we compared two methods: 1) the proposed difference-based gradient approximation and 2) the baseline STE method in previous works [8]–[13], which approximates the gradient of the AppMult using that of the AccMult. For a fair comparison, we did not use the previous AppMult-aware retraining frameworks in [8]–[13]. Instead, both methods were implemented in our retraining framework. The experiments were conducted using 4 NVIDIA GeForce RTX 3090 GPUs, with each experiment running on a single GPU.

We utilized Synopsys Design Compiler [23] to measure the area, delay, and power of the AppMults using the ASAP 7nm standard cell library [24] and measured the power using a 1GHz clock under a uniform input distribution. The ER, NMED, MaxED error metrics of AppMults were measured by enumerating all possible input combinations under a uniform distribution, according to Eq. (2). To retrain the AppMult-based DNN models, a default setting based on the other AppMult-aware DNN retraining works [8]–[13] was applied unless otherwise specified: batch size 64, number of epochs 30, Adam optimizer, and learning rate 0.001 in epochs 1–10, 0.0005 in epochs 11–20, and 0.00025 in epochs 21–30.

We tested our framework on the CIFAR-10 and CIFAR-100 [25] datasets using the VGG19 [26] and ResNet [27] DNN models. As the most computation-intensive parts of these DNNs are the convolutional layers, we replaced all accurate multipliers in these layers with the same type of AppMults to reduce the hardware cost, following the approach used in previous works [13], [16].

The tested multipliers and their errors (see Eq. (2)), areas, delays, and powers are listed in Table I. These unsigned multipliers have bit widths of 8, 7, and 6, belonging to

TABLE I: Characteristics of tested unsigned multipliers, including area, delay, power, and error metrics. *HWS* refers to the selected half window size for the difference-based gradient approximation based on experiments. The term "_rm$k$" indicates the removal of the rightmost k columns of partial products. The term "_syn" means the AppMult is generated by the approximate logic synthesis tool [28]. "N/A" stands for not applicable.

Multiplier	Area /μm^2	Delay /ps	Power /μW	ER /%	NMED /%	MaxED	HWS
mul8u_acc	25.6	730.1	22.93	0.0	0.0	0	N/A
mul8u_syn1	13.0	582.2	9.68	99.1	0.28	1937	16
mul8u_syn2	12.3	577.7	9.29	99.5	0.30	2057	16
mul8u_2NDH	10.0	512.6	6.48	98.7	0.44	2709	32
mul8u_17C8	7.7	624.4	5.01	99.0	0.56	1577	16
mul8u_1DMU	15.6	837.6	11.09	66.0	0.65	4084	32
mul8u_17R6	6.9	743.3	4.60	99.0	0.67	1925	32
mul8u_rm8	11.6	655.0	9.19	98.0	0.68	1793	16
mul7u_acc	19.0	695.0	15.72	0.0	0.00	0	N/A
mul7u_06Q	10.6	861.9	7.90	95.4	0.24	162	4
mul7u_073	11.0	889.8	8.61	95.2	0.27	154	2
mul7u_rm6	11.4	599.0	9.00	96.1	0.28	273	2
mul7u_syn1	11.5	561.3	9.06	97.6	0.28	457	8
mul7u_syn2	10.9	532.4	7.98	98.8	0.39	713	8
mul7u_081	10.7	673.6	7.67	97.3	0.45	314	16
mul7u_08E	8.9	612.5	6.15	97.5	0.46	317	4
mul6u_acc	14.1	680.1	10.47	0.0	0.00	0	N/A
mul6u_rm4	10.3	563.9	7.06	81.3	0.3	49	2

three categories: 1) Simple AppMults by removing the partial products in the least significant parts like the one in Fig. 2 (marked with "_rm$k$", donoting the removal of the rightmost k columns of partial products); 2) AppMults from the approximate arithmetic library EvoApproxLib [29]; 3) AppMults generated by the approximate logic synthesis tool [28] (marked with "_syn"). Table I also reports the accurate multiplier information for reference (marked with "_acc").

Since different AppMult functions have different degrees of smoothness, we selected different half window sizes (*HWS*, see Section III-A) for each of them. Specifically, we tried different *HWS* = 1, 2, 4, 8, 16, 32, and 64 for each AppMult, and for each *HWS*, we trained a small LeNet [30] model on the CIFAR-10 dataset for 5 epochs. Then, we chose the best *HWS* that achieves the smallest training loss. The selected *HWS* is listed in the last column of Table I.

B. Experiments on the CIFAR-10 Dataset

1) Comparison Using the VGG19 Model: This experiment compares our gradient approximation method with the STE method on the CIFAR-10 dataset using the VGG19 model. The top part of Table II presents the accuracy after retraining with various 7-bit and 8-bit AppMults, comparing the STE-based gradient with our difference-based gradient approximation. The accuracies of the quantized DNNs using the AccMults after quantization-aware training [19], the initial accuracies using AppMults before AppMult-aware retraining, and the power consumption, delay, and NMED of the multipliers are also reported. Note that the power consumption and delay are normalized to those of the 8-bit AccMult (mul8u_acc).

From the top part of Table II, we can see that our method consistently outperforms STE for all tested 8-bit and 7-bit AppMults, improving the accuracy after retraining by an average of 4.10%. Compared to the initial accuracy before AppMult-aware retraining, our method recovers the accuracy from 23.06% to 86.16% on average. Notably, for mul8u_1DMU and mul8u_rm8, our method improves the accuracy by 10.28% and 15.31%, respectively, compared to STE. Moreover, for mul7u_073, our method recovers the accuracy from 83.59% to 91.89%, and the final accuracy is very **close to the reference accuracy of 92.10%** with 7-bit AccMult. Meanwhile, mul7u_073 (normalized power=0.38) **reduces power consumption by 45%** compared to the 7-bit AccMult (normalized power=0.69), offering an attractive trade-off between power consumption and accuracy.

As for runtime, for example, our method takes about 1.4 hours to retrain the VGG19 model with a 7-bit AppMult using a single NVIDIA RTX 3090 GPU, which is about 1.4 times of the STE-based method. The runtime overhead is primarily due to the additional computation of the difference-based gradient

TABLE II: Retraining results with the STE-based gradient and our difference-based gradient on the CIFAR-10 dataset. Power consumption and delay are normalized to those of the 8-bit AccMult (mul8u_acc). **Bold** entries denote that our method outperforms the STE-based method.

DNN	Multiplier	Inital acc. /%	Acc./% after retrain STE	Ours	Improve	Multiplier information Norm. power	Norm. delay	NMED /%
VGG19	mul8u_acc	Reference accuracy: 92.48%				1.00	1.00	0.00
	mul8u_syn1	7.39	80.05	**85.65**	**5.60**	0.42	0.80	0.28
	mul8u_syn2	8.19	86.21	**89.85**	**3.64**	0.41	0.79	0.30
	mul8u_2NDH	9.36	91.07	**91.21**	**0.14**	0.28	0.70	0.44
	mul8u_17C8	9.21	86.96	**88.35**	**1.39**	0.22	0.86	0.56
	mul8u_1DMU	8.84	69.13	**79.41**	**10.28**	0.48	1.15	0.65
	mul8u_17R6	9.75	84.23	**86.19**	**1.96**	0.20	1.02	0.67
	mul8u_rm8	10.37	57.72	**73.03**	**15.31**	0.40	0.90	0.68
	mul7u_acc	Reference accuracy: 92.10%				0.69	0.95	0.00
	mul7u_06Q	67.07	91.79	**91.95**	**0.16**	0.34	1.18	0.24
	mul7u_073	83.59	91.50	**91.89**	**0.39**	0.38	1.22	0.27
	mul7u_rm6	8.17	76.85	**82.79**	**5.94**	0.39	0.82	0.28
	mul7u_syn1	8.24	83.19	**90.11**	**6.92**	0.40	0.77	0.28
	mul7u_syn2	10.98	73.42	**76.90**	**3.48**	0.35	0.73	0.39
	mul7u_081	9.84	86.63	**88.56**	**1.93**	0.33	0.92	0.45
	mul7u_08E	71.86	90.00	**90.28**	**0.28**	0.27	0.84	0.46
	VGG19 mean over 7&8-bit AppMults	23.06	82.05	**86.16**	**4.10**			
ResNet18	mul8u_acc	Reference accuracy: 93.73%				1.00	1.00	0.00
	mul8u_syn1	9.99	87.44	**91.64**	**4.20**	0.42	0.80	0.28
	mul8u_syn2	10.19	91.58	**92.07**	**0.49**	0.41	0.79	0.30
	mul8u_2NDH	27.47	93.37	**93.43**	**0.06**	0.28	0.70	0.44
	mul8u_17C8	12.48	91.80	**92.29**	**0.49**	0.22	0.86	0.56
	mul8u_1DMU	9.94	82.07	**92.10**	**10.03**	0.48	1.15	0.65
	mul8u_17R6	12.24	90.95	**92.07**	**1.12**	0.20	1.02	0.67
	mul8u_rm8	9.92	81.16	**90.96**	**9.80**	0.40	0.90	0.68
	mul7u_acc	Reference accuracy: 93.61%				0.69	0.95	0.00
	mul7u_06Q	91.30	93.39	**93.66**	**0.27**	0.34	1.18	0.24
	mul7u_073	90.77	93.42	**93.50**	**0.08**	0.38	1.22	0.27
	mul7u_rm6	9.54	89.79	**93.27**	**3.48**	0.39	0.82	0.28
	mul7u_syn1	10.15	92.71	**93.00**	**0.29**	0.40	0.77	0.28
	mul7u_syn2	10.00	80.17	**90.15**	**9.98**	0.35	0.73	0.39
	mul7u_081	12.62	92.60	**92.80**	**0.20**	0.33	0.92	0.45
	mul7u_08E	86.81	92.14	**92.62**	**0.48**	0.27	0.84	0.46
	ResNet mean over 7&8-bit AppMults	28.82	89.47	**92.40**	**2.93**			

during the backward propagation. However, it is acceptable given the significant accuracy improvement.

2) Comparison With the ResNet18 Model: This experiment compares our gradient approximation method with the STE method on the CIFAR-10 dataset using the ResNet18 model. The bottom part of Table II presents the accuracy after retraining using various 7-bit and 8-bit AppMults, comparing the STE method with ours.

From the bottom part of Table II, we observe that our method consistently outperforms STE for all tested AppMults, with an average accuracy improvement of 2.93%. Compared to the accuracy before AppMult-aware retraining, our method recovers the accuracy from 28.82% to 92.40% on average. Notably, for mul8u_1DMU and mul7u_syn2, our method improves the accuracies by 10.03% and 9.98%, respectively, compared to STE, and the final accuracies after retraining are 92.10% and 90.15%, respectively. Although these final accuracies are slightly lower than the reference accuracies, there is a significant hardware cost reduction. Compared to the 8-bit AccMult, mul8u_1DMU **saves 52% power**. Compared to the 7-bit AccMult, mul7u_syn2 **reduces power by 49% and delay by 23%**. This demonstrates a promising trade-off between accuracy and hardware cost. Moreover, for mul7u_06Q, our method achieves an accuracy of 93.66%, which is even higher than the reference accuracy of 93.61% for 7-bit AccMult, while reducing the power by 51%. This shows that in some cases, AppMults can achieve both hardware cost reduction and accuracy improvement.

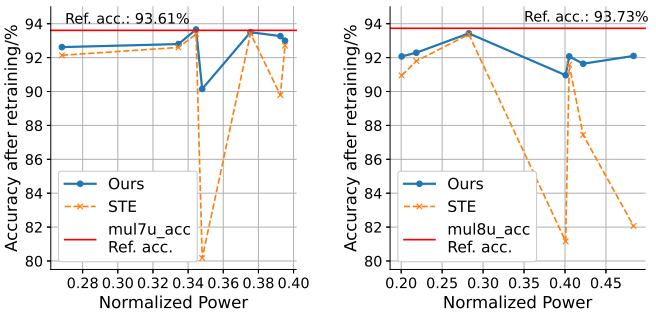
Using the data from Table II, we plot the ResNet18 accuracy after retraining versus power consumption for 7-bit and 8-bit AppMults in Fig. 5(a) and Fig. 5(b), respectively. In both figures, at the same normalized power level, our method consistently outperforms the STE method in accuracy. Reference accuracies for the 7-bit and 8-bit AccMults are indicated by the red horizontal lines. Our method achieves better accuracy-power trade-offs, with acceptable drops in accuracy compared to the reference accuracies, whereas the STE method exhibits significant fluctuations, sometimes reducing the accuracy by over 10% compared to the reference accuracies.

As for runtime, for example, our method takes about 2.4 hours to retrain the ResNet18 model with a 7-bit AppMult using a single NVIDIA RTX 3090 GPU, which is about 2.6 times the runtime of the STE method. However, we believe that it is acceptable given the large accuracy improvement.

C. Experiments on the CIFAR-100 Dataset

This experiment compares our gradient approximation method with the STE method on the CIFAR-100 dataset using the ResNet34 and ResNet50 models. To show the wide applicability of our framework, instead of using 8-bit and 7-bit AppMults, we test the 6-bit unsigned AppMult in Table I (mult6u_rm4), which removes the rightmost 4 columns of partial products. Compared to the 6-bit AccMult, mul6u_rm4 saves 27% area, 17% delay, and 33% power consumption.

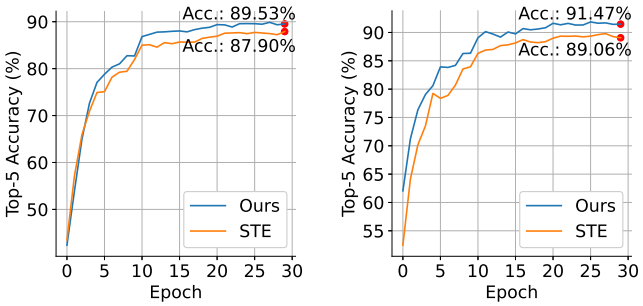
Fig. 6 illustrates the Top-5 testing accuracy curves for two models, ResNet34 and ResNet50, trained on the CIFAR-100 dataset over 30 epochs. Fig. 6(a) shows the accuracy comparison for ResNet34, while Fig. 6(b) presents the results for ResNet50. For ResNet34, our method achieves a higher final accuracy of 89.53% compared to STE's 87.90%. For ResNet50, our method also outperforms STE, achieving a final accuacy of 91.47% compared to STE's 89.06%. Notably, for ResNet34, our method shows better performance after 4 epochs, while for ResNet50, our method consistently outperforms STE for all epochs. Therefore, our method demonstrates a faster convergence rate than STE.



(a) 7-bit AppMults. The normalized power of mul7u_acc is 0.69.

(b) 8-bit AppMults. The normalized power of mul8u_acc is 1.00.

Fig. 5: Trade-off between ResNet18 accuracy and power consumption using 7-bit and 8-bit AppMults on the CIFAR-10 dataset. Power is normalized to that of the 8-bit AccMult (mul8u_acc).



(a) ResNet34.

(b) ResNet50.

Fig. 6: Top-5 testing accuracies of ResNet34 and ResNet50 models versus epochs, using the 6-bit AppMult mul6u_rm4 on the CIFAR-100 dataset.

VI. Conclusion and Future Work

This paper proposes a difference-based gradient approximation technique for AppMults to enhance the accuracy of AppMult-aware retraining. Our method improves the retraining accuracy by an average of 4.10% for the VGG model and 2.93% for the ResNet18 model, compared to the STE method. When retraining a ResNet18 model with the 7-bit AppMult mul7u_06Q using our method, the final DNN accuracy does not degrade compared to the quantized model using the 7-bit AccMult, while the power consumption is reduced by 51%. In the future, we will extend our method to other AI models, including 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] Jie Han and Michael Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *European Test Symposium (ETS)*, pages 1–6, 2013.

[3] Sparsh Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)*, 48(4):1–33, 2016.

[4] 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.

[5] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. Low-bit quantization of neural networks for efficient inference. In *International Conference on Computer Vision Workshop (ICCVW)*, pages 3009–3018, 2019.

[6] Jeongwoo Park, Sunwoo Lee, and Dongsuk Jeon. A neural network training processor with 8-bit shared exponent bias floating point and multiple-way fused multiply-add trees. *IEEE Journal of Solid-State Circuits (JSSC)*, 57(3):965–977, 2021.

[7] 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.

[8] Xin He, Liu Ke, Wenyan Lu, Guihai Yan, and Xuan Zhang. AxTrain: Hardware-oriented neural network training for approximate inference. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6, 2018.

[9] Cecilia De la Parra, Andre Guntoro, and Akash Kumar. ProxSim: GPU-based simulation framework for cross-layer approximate DNN optimization. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1193–1198, 2020.

[10] 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.

[11] Jing Gong, Hassaan Saadat, Hasindu Gamaarachchi, Haris Javaid, Xiaobo Sharon Hu, and Sri Parameswaran. ApproxTrain: Fast simulation of approximate multipliers for DNN training and inference. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 42(11):3505–3518, 2023.

[12] Rodion Novkin, Florian Klemme, and Hussam Amrouch. Approximation-and quantization-aware training for graph neural networks. *IEEE Transactions on Computers (TC)*, 2023.

[13] Tianyang Yu, Bi Wu, Ke Chen, Chenggang Yan, and Weiqiang Liu. Toward efficient retraining: A large-scale approximate neural network framework with cross-layer optimization. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 2024.

[14] William Andrew Simon, Valérian 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.

[15] 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.

[16] 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.

[17] Muhammad Abdullah Hanif, Faiq Khalid, and Muhammad Shafique. CANN: Curable approximations for high-performance deep neural network accelerators. In *Design Automation Conference (DAC)*, pages 1–6, 2019.

[18] Filip Vaverka, Vojtech Mrazek, Zdenek Vasicek, and Lukas Sekanina. TFApprox: Towards a fast emulation of DNN approximate hardware accelerators on GPU. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 294–297, 2020.

[19] 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.

[20] NVIDIA Corporation. CUDA toolkit. https://developer.nvidia.com/cuda-toolkit, 2024.

[21] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing*, 461:370–403, 2021.

[22] 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.

[23] Synopsys, Inc. Synopsys softwares. http://www.synopsys.com, 2024.

[24] Lawrence T Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. ASAP7: A 7-nm FinFET predictive process design kit. *Microelectronics Journal*, 53:105–115, 2016.

[25] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. *Technical Report, University of Toronto*, 2009.

[26] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[28] 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.

[29] 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.

[30] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.