ProxSim: GPU-based Simulation Framework for Cross-Layer Approximate DNN Optimization

Cecilia De la Parra
Robert Bosch GmbH
Renningen, Germany
cecilia.delaparra@de.bosch.com

Andre Guntoro
Robert Bosch GmbH
Renningen, Germany
andre.guntoro@de.bosch.com

Akash Kumar
Technological University of Dresden
Dresden, Germany
akash.kumar@tu-dresden.de

***Abstract*—Through cross-layer approximation of Deep Neural Networks (DNN) significant improvements in hardware resources utilization for DNN applications can be achieved. This comes at the cost of accuracy degradation, which can be compensated through different optimization methods. However, DNN optimization is highly time-consuming in existing simulation frameworks for cross-layer DNN approximation, as they are usually implemented for CPU usage only. Specially for large-scale image processing tasks, the need of a more efficient simulation framework is evident. In this paper we present *ProxSim*, a specialized, GPU-accelerated simulation framework for approximate hardware, based on Tensorflow, which supports approximate DNN inference and retraining. Additionally, we propose a novel hardware-aware regularization technique for approximate DNN optimization. By using ProxSim, we report up to 11× savings in execution time, compared to a multi-thread CPU-based framework, and an accuracy recovery of up to 30% for three case studies of image classification with MNIST, CIFAR-10 and ImageNet.**

***Index Terms*—Simulation Framework, embedded Neural Networks, Approximate Computing, Deep Learning**

I. Introduction

Deep Learning (DL) applications for image processing, such as Convolutional Neural Networks (CNNs), can achieve high, near-human accuracy at complex regression and classification tasks. This comes at the cost of large computational requirements, as DNNs perform energy-costly Multiply-and-Accumulate (MAC) operations and memory accesses. Through cross-layer approximate computing, significant savings in computational resources can be achieved by systematically introducing approximations in DNN computations at software and hardware levels. The introduced approximations cause accuracy degradation that may then reach critical levels, from which a recovery is possible only through optimization techniques. To analyze which approximation approaches are adequate for a certain DNN, and to perform the corresponding optimization for accuracy recovery, a specialized DL simulation framework for cross-layer approximation is required. Furthermore, the simulation of approximate, large-scale image processing tasks requires high computing parallelism, which can be achieved only through functional acceleration, for example by using Graphic Processing Units (GPU).

In this work, we present *ProxSim*, a specialized, highly flexible, GPU-accelerated Simulation Framework for testing and optimization of cross-layer DNN approximation. ProxSim allows to retrain DNN parameters given specific approximate hardware, through a combination of existing techniques, and our novel regularization approach for DNN retraining. We sumarize our main contributions as follows:

- ProxSim, a GPU-based, cross-layer approximate simulation framework for DNN optimization.
- Accelerated simulation of approximate hardware in DNN computation through specialized GPU support.
- Formulation of a novel hardware-aware regularization for approximate DNN retraining.
- Exploration and analysis of several techniques for optimization and retraining of approximate DNNs, achieving over 30% of accuracy recovery for the most extreme DNN approximations, compared to the initial accuracy.

We evaluate and validate our contributions through detailed experiments, focused on CNNs for image classification trained with MNIST [1], CIFAR-10 [2] and ImageNet [3].

II. Related work

Highly specialized, GPU-accelerated simulation frameworks for precision-scaling or quantization methods, such as Tensorflow [4], Graffitist [5] or Ristretto [6], have been proposed in the literature, regarding software-oriented approximation techniques. Simulation frameworks for cross-layer approximation of DNNs, on the other hand, are typically oriented towards specialized goals and thus are generally implemented for CPU only, which increments training and inference times, particularly when simulating approximate hardware on large DNNs. Examples of these frameworks are AxDNN [7] and TypeCNN [8]. AxDNN is a pre-RTL simulation framework for design space exploration which combines precision-scaling and pruning methods with simulation of approximate hardware. TypeCNN focuses on variable data types for accurate and approximate arithmetic operations for training and inference of small CNNs that are to run on dedicated hardware.

While algorithmic simulation frameworks for cross-layer DNN approximations such as ApproxANN [9] and AxNN [10] have delivered efficient methods for partial DNN approximation, we focus on evaluating and optimizing full DNN approximation, to maximize energy savings. Recent works, such as AxTrain [11], have demonstrated that it is possible to recover lost accuracy in small, fully approximated DNNs through adapted Stochastic Gradient Descent (SGD) techniques. In

978-3-9819263-4-7/DATE20/©2020 EDAA

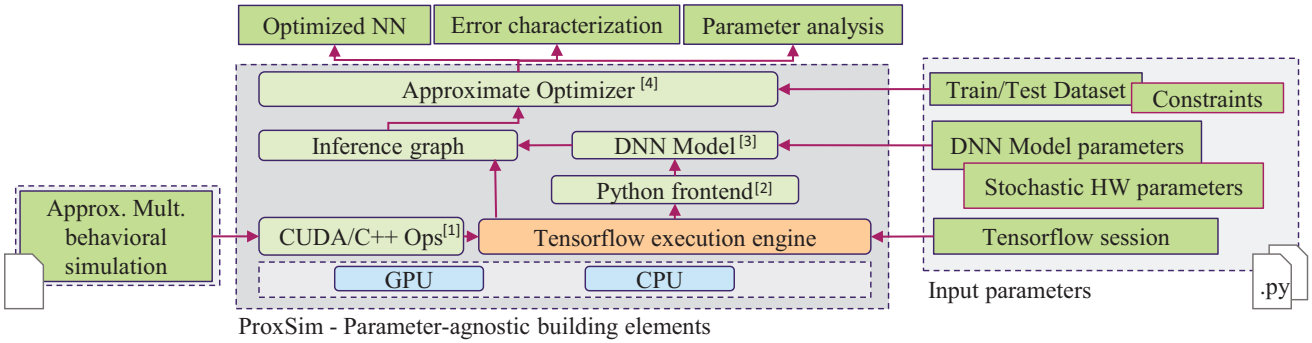

Fig. 1: Architecture of ProxSim

AxTrain, two DNN training techniques, referred to as *passive* and *active* hardware-aware retraining methods, are proposed. Through active retraining, the DNN sensitivity is minimized, making the DNN generally more robust. Alternatively, passive retraining incorporates approximate elements directly into the DNN computation during retraining, resulting in overall better accuracy. Within this context, we complement the state of the art with a novel hardware-aware optimization approach, adequate for retraining large, approximate DNNs dedicated to solve more complex image classification tasks.

III. Simulation Framework

A. Data types

ProxSim supports variable bitwidth for DNN activations and weights, which can be further adapted to be layer-wise parameterizable. In forward passes, the following data types for tensor representation are available:

- 32-bit floating-point (FP) for accurate DNN operations.
- quantization to bitwidths <= 32-bit in int or FP format.
- 8-bit or smaller integer representation for integration of Approximate Arithmetic Units (AUs). To investigate the effects of cross-layer DNN approximation, ProxSim supports reduced bitwidth quantization. A maximum available bitwidth= 8 is supported for AUs, as 8-bit quantization is sufficient for complex DNN classification tasks without significant accuracy loss [5], [12], [13]. For other DNN applications, the available bitwidth may be further increased.

In ProxSim, backward passes are computed using FP32 precision to exploit the accuracy provided by such representation [7] and to allow proper gradient computation in Tensorflow.

B. Architecture

ProxSim is based on a parameter-independent architecture, depicted in Fig. 1, which runs on the Tensorflow execution engine. The main building elements, numbered according to Fig. 1, are described in the following sections.

1) CUDA Operators: Programmed in CUDA [14] for GPU acceleration, these operators build the basic structure of the approximate computational graph. Through a Python wrapper, these operators are integrated in Tensorflow's frontend and incorporated in Tensorflow's execution engine, including forward and backward passes. Implemented specifically for approximate multiplication, these CUDA operators allow a flexible interchange of Approximate Multipliers (AMs) of signed/unsigned 8 bits or smaller. The behavioral simulation of the corresponding AM is hereby given as a Tensor [4] in form of a look-up table of maximum size of 128 kB and the CUDA operators manage an AM-independent execution time.

2) Python frontend: Processing level of ProxSim. The following elements are hereby implemented:

- Quantization module. Inputs, bias and weights can be quantized to parameterizable bitwidths b as in (1):

$$X_q = \text{clip}\left[\left(\frac{X}{\Delta_X}\right), \{-2^{b-1}, 2^{b-1}-1\}\right] \Delta_X = Q(x) \tag{1}$$

  In ProxSim, the quantization step Δ can be calculated through three different methods, as presented in [12]:
  - Maximum Absolute Value:

$$\Delta_X = \frac{max(|X|)}{2^{b-1}} - 1 \tag{2}$$

  - Minimization of Mean Squared Error:

$$\Delta_X = \underset{\Delta_X}{\text{argmin}} \|X - X_q\|_2 \tag{3}$$

  - Minimization of propagated quantization error:

$$\Delta_X = \underset{\Delta_X}{\text{argmin}} \|\widehat{y_k} - y_k\|_2 \quad , \tag{4}$$

    where $\widehat{y_k}$ and y_k correspond to the approximate and accurate output of the k-th layer respectively.

  Using this linear quantization as baseline, we also implement power-of-two transformation [12].
- Stochastic hardware simulation. Stochastic hardware such as fuzzy memory, which are parameterized by their distribution moments and probability density functions.
- Approximate neural layers. Fully-connected and 1D/2D-Convolutional layers. These layers call the corresponding CUDA Operator in case of approximation or the equivalent Tensorflow operation otherwise. For efficient computation, all approximate neural layers are implemented through General Matrix Multiplications (GEMM), where

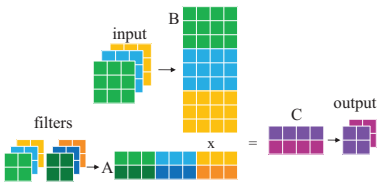

Fig. 2: Graphic representation of tensor transformations to express convolutional layers through GEMM

the input tensors are transformed as in Fig. 2 in case of a convolution.

Through approximate GEMM, denoted as $\widetilde{\phi}(\cdot)$, we can define any quantized and approximated fully-connected or convolutional layer as follows:

$$\widetilde{Y_q} = X_q \widetilde{\times} W_q + B_q = \widetilde{\phi}(X_q, W_q) + B_q \quad , \qquad (5)$$

where X_q, W_q, B_q correspond to the quantized activations, weights and bias respectively. After any neural layer, a non-linearization followed by a pooling operation can be performed. In ProxSim, these non-linearities and pooling operations are not approximated. ProxSim allows to perform layer-wise and neuron-wise DNN approximation. Moreover, this flexibility allows to assign and simulate different approximate hardware in each DNN layer.

3) DNN Model: A class defined to constitute the core of any DNN architecture. This class is built over the Python-frontend and integrates all approximate neural layers to generate a Tensorflow inference graph, given specific DNN model parameters.

4) Approximate Optimizer: In this architectural block, approximate DNN inference and optimization through retraining are performed. IN DNN retraining, we minimize the original cost function used to train the DNN, denoted as $C_\theta(y)$, where θ corresponds to the DNN parameters and y represents the DNN output. The cost function contemplated in this work is the cross-entropy loss, as we consider only DNNs for image classification. This loss is defined as in (6), where y_k is the predicted value, γ_k the label, and n the number of predicted classes.

$$C_\theta(y) = -\sum_{k=1}^{n} \gamma_k \log y_k \qquad (6)$$

During DNN training, $C_\theta(y)$ is minimized through gradient descent algorithms, where (7) is computed through Backpropagation (BP).

$$\frac{\partial C(y)}{\partial W_q} = \frac{\partial C(y)}{\partial y} \frac{\partial y}{\partial W_q} \qquad (7)$$

When integrating cross-layer approximations in the DNN computation, the BP process needs to be altered to deal with gradient noise introduced by approximations. In ProxSim, we apply the approximate BP flow depicted in Fig. 3. Additionally, we implement hardware-aware regularization to the original cost function $C_\theta(y)$, to further improve the DNN optimization process. A thorough explanation of the BP flow and regularization implemented in ProxSim is presented in sub-sections III-C and III-D. Besides the corresponding DNN accuracy, the approximate optimizer manages other instances such as statistical analysis of the DNN parameters and characterization of the approximation error.

C. Approximate Backpropagation

Since approximations introduce undesired gradient noise that negatively affects weight updates, we adopt two methodologies in ProxSim to overcome this problem: Straight-Through-Estimator (STE) [15] and Shadow Weights [16]. For the sake of brevity, we focus on image classification. However, this approximation mechanism is application-agnostic and can be used for other perception tasks such as semantic segmentation or speech recogntion.

1) Straight-Through-Estimator: Approximate DNN retraining is a non-trivial task as the gradient of many approximate functions is not defined. A way around this is to implement an STE as in (8), where the gradient of a non-differentiable function $\widetilde{f}(x)$ is substituted by the gradient of a related, differentiable or sub-differentiable function $f(x)$.

$$\frac{\partial \widetilde{f}(x)}{\partial x} \rightarrow \frac{\partial f(x)}{\partial x} \qquad (8)$$

In ProxSim, the STE is implemented as follows:

- In the quantization module: the gradient of non-differentiable functions such as *clip* and *round* (denoted as $[\cdot]$) is substituted by that of the identity function.
- In the DNN layers: the gradient of the approximate GEMM is substituted by the accurate GEMM gradient.
- For stochastic hardware: The gradient is bypassed with the gradient of the identity function.

2) Shadow weights: When retraining a DNN with quantized values, weights can get stagnated as SGD updates are too small compared to the quantized magnitude to result in an effective weight update. Thus, in ProxSim, we implement *shadow weights* [16], as in Fig. 3. At iteration t, the gradient of the cost function is computed with respect to the quantized weights $W_{q,t}$ and this gradient is then applied to update the unquantized weights w_{t+1}, according to (9). This results in an effective parameter update on each SGD iteration.

$$w_{t+1} = w_t + \Delta w = w_t - \nabla \frac{\partial C(\widetilde{y})}{\partial W_{q,t}} \qquad (9)$$

3) Optimization methods and loss functions: In ProxSim, the available training losses and SGD-based optimization methods correspond to the ones available in the Tensorflow Modules *losses* and *train*, respectively.

D. Approximate Optimization

In ProxSim, we introduce hardware-aware regularization to the original cost function $C(\widetilde{y})$ for approximate DNN retraining. First, we integrate the behavioral simulation of the

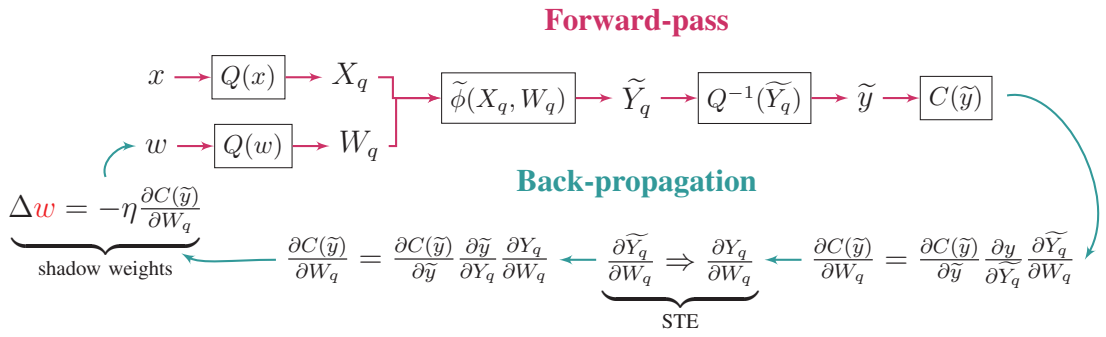

Fig. 3: Gradient flow during backpropagation on ProxSim

approximate hardware elements directly in the computation of DNN forward passes, similarly to the passive training in AxTrain. Then, inspired by (4) [12], we propose to add a regularization term to $C(\widetilde{y})$ which minimizes the propagated approximation error in each DNN layer. To avoid vanishing gradients of this regularization during retraining, we define our additive regularization term as follows:

$$C_\alpha(\widetilde{y}) = C(\widetilde{y}) + \alpha \sum_{k=0}^{n} \|\widetilde{y_k} - y_k\|_1 \quad , \tag{10}$$

where y_k and $\widetilde{y_k}$ correspond to the accurate and approximate outputs of the k-th DNN layer respectively, n denotes the number of approximate layers in the DNN architecture, and α is a scaling value for the hardware-aware regularization. We use the L^1-Norm to ensure robustness against error outliers. We refer to this hardware-aware regularization as *alpha regularization*. The gradient of $C_\alpha(\widetilde{y})$ with respect to the weights at the k-th layer is then computed as follows:

$$\begin{aligned} \frac{\partial C_\alpha(\widetilde{y})}{\partial w_k} &= \frac{\partial C(\widetilde{y})}{\partial w_k} + \alpha \left(\frac{\partial L_n^1}{\partial w_k} + ... + \frac{\partial L_k^1}{\partial w_k} \right) \\ &= \frac{\partial C(\widetilde{y})}{\partial w_k} + \\ &\alpha \left(\frac{\partial L_k^1}{\partial \widetilde{y}_n} \frac{\partial \widetilde{y}_n}{\partial \widetilde{y}_{n-1}} ... \frac{\partial \widetilde{y_k}}{\partial w_k} + ... + \underbrace{\frac{\partial L_k^1}{\partial \widetilde{y_k}} \frac{\partial \widetilde{y_k}}{\partial w_k}}_{\text{k-th gradient}} \right) \end{aligned} \tag{11}$$

By directly including the gradient of the L^1-norm at the k-th layer in our modified cost function $C_\alpha(\widetilde{y})$, we ensure that the propagated approximation error on the k-th layer is directly minimized, thus avoiding vanishing gradients of L^1. As our experiments show, the computational overhead of the alpha regularization is compensated by the improvements achieved through alpha regularization. Note that α needs to be sufficiently small to contribute to $C_\alpha(\widetilde{y})$ without outweighing the cost which optimizes the DNN accuracy. Hence, α is a DNN retraining hyperparameter that can be further optimized.

IV. Evaluation

We dedicate this section to report the results from our evaluation of ProxSim and of our enhaced optimization flow for approximate DNNs. All experiments were performed using an Nvidia GeForce GTX 1080 Ti (11GB GDDR5X,1582MHz), Tensorflow version 1.12.0 and CUDA V9.0.176.

A. CNNs for image classification

We evaluate ProxSim by integrating various CNNs, with a wide range of sizes and number of MAC operations, for image classification at small, medium and large scale. To describe the CNN topologies, we denote convolutional layers with f filters of c size and s stride as Conv$\{c, f, s\}$; similarly, we denote dense or fully-connected layers with x outputs as FC$\{x\}$. All convolutional and dense layers are followed by ReLU activations unless the contrary is stated. The DNNs considered for evaluation are the following:

- Basic-NN. For comparison with the results presented in [8], we implement the same basic DNN for digit recognition with MNIST. The architecture is as follows: Conv$\{5, 8, 1\}$ $\rightarrow$ MaxPool $\rightarrow$ FC$\{10\}$ $\rightarrow$ Sigmoid
- TF-NN. A more complex DNN for robust digit recognition on MNIST. Obtained from [4]. Architecture: Conv$\{3, 28, 1\}$ $\rightarrow$ MaxPool $\rightarrow$ FC$\{128\}$ $\rightarrow$ FC$\{10\}$
- ALL-CNN [17]. CNN for image classification on CIFAR-10. This CNN achieves good accuracy (89.02%) with a relatively small number of parameters and MAC-ops.
- SqueezeNet [18]. Small CNN for large-scale image recognition on ImageNet, with Top-1 acc. of 56.67%.
- ResNet50 [19]. CNN with an architecture based on residual connections, trained for image classification on ImageNet. Top-1 acc. of 72.89%.
- VGG-16 [20]. Large and Robust CNN for image classification trained on ImageNet, with Top-1 acc. of 70.14%.

All CNNs are quantized to int8 through minimization of the propagated quantization error [12].

B. Approximate Multipliers

To evaluate the functionality of ProxSim for approximate inference and retraining, we implement 10 different, arbitrarily chosen approximate multipliers, with a Mean Relative Error (MRE) and Mean Absolute Error (MAE) of up to 16% and 1099 respectively, from the following sources:

TABLE I: Approximate Multipliers integrated in ProxSim

Approx. Multipliers		MRE [%]	MAE	savings[%]
SMApprox	1110	3.96	130	36.04
	3330	6.39	203	36.34
	1111	8.75	642	36.91
	3333	13.63	843	**37.21**
	2222	**16.55**	**1099**	36.92
EvoApprox	118	0.72	5	25.45
	130	1.92	16	31.81
	468	5.43	66	43.76
	230	8.13	86	36.88
	74	**14.35**	**217**	**66.80**

TABLE II: Comparison of training times over 10 epochs with Basic-NN in ProxSim and TypeCNN

Framework \ batch size		1	64	128	final
		training times [s]			acc. [%]
TypeCNN	<FP32>	**538**	-	-	**98.31**
ProxSim	<FP32>	882	31	**20**	97.98
	quantized <8b>	1938	**35**	24	94.57
	quant.+approx. <8b>	1947	167	**49**	93.05

- EvoApprox8b: open source library of approximate multipliers for circuit design and benchmarking of approximation methods [21].
- SMApprox: open source library of FPGA-based Approximate Multipliers [22].

The implemented multipliers and their MRE, MAE and power savings in their corresponding platforms are in Table I.

C. Execution time

1) Comparison with TypeCNN: We fully approximate and retrain the Basic-NN over 10 epochs using the same training method and hyperparameters as in [8], with different batch sizes, to compare the execution times of ProxSim and TypeCNN [8], a highly optimized, CPU-based cross-layer approximate simulation framework. From Table I, we use the approximate multiplier with largest MRE, the SMApprox 2222; this experiment is repeated 10 times. It should be noted, however, that the inference time remains constant for any approximate multiplier. The mean values of our results are presented in Table II. The values reported with TypeCNN were obtained with an Intel Core i7-4720HQ (8GB RAM, 2.60GHz8). We observe that, while TypeCNN is faster than ProxSim with a batch size of 1, the retraining acceleration in ProxSim due to GPU parallelization is evident with larger batch sizes. With a batch size of 128, execution times in ProxSim are 22× faster with quantization and 11× faster with quantization + approximation. Furthermore, the BP process benefits from large batch sizes, and thus from this parallelization, as it leads to a better DNN generalization [23].

As the Basic-NN has few parameters, its sensitivity towards approximations is very high, which is why the accuracy drops around 5% after full quantization and approximation.

2) General evaluation: We now evaluate the execution time of more complex approximate CNNs presented in Section IV-A. For this, we perform DNN inference using the corresponding validation dataset for 50 runs with randomly chosen approximate multipliers from Table I, with a batch size of 64 for all DNNs. The resulting inference times are reported in Table III. We observe proportionallity between the approximate inference time and the amount of DNN parameters/MAC Operations.

D. Approximate optimization

To evaluate the DNN optimization methods implemented in ProxSim, specially our proposed alpha regularization, we retrain selected DNNs from Table III, which have been fully quantized (to 8 bits) and approximated. We compare our results with the passive training method from AxTrain [11].

1) TF-NN: We begin with a small but robust DNN for digit recognition, the TF-NN, with an FP and quantized accuracy of 98.3% and 98.02% respectively. We approximate this DNN using AMs with a large MRE, the SMApprox 2222 and the EvoApprox 74, to assess if an accuracy recovery after extreme approximation is possible. We retrain this DNN through SGD with a learning rate of 1e-2 and a batch size of 256 until the FP accuracy is reached. The obtained results are displayed in Table IV. Although the original FP accuracy can be reached through passive retraining, the alpha regularization achieves similar accuracy recovery in less epochs.

2) ALL-CNN: We now move on towards larger DNNs for more complex tasks. For this analysis, we use ALL-CNN for Cifar10, trained without data augmentation, with FP and quantized accuracy of 89.02%. Since this CNN performs a more challenging task, we define hereby an accuracy tolerance of 0.5%, proposed in [10] for a negligible accuracy loss. We retrain this CNN over 5 epochs using SGD with a learning rate of 1e-3 and a batch size of 256 with passive retraining and with alpha regularization, with the objective of reaching our pre-defined accuracy tolerance. The obtained results with $\alpha = 1e-7$ are shown in Table V. The initial accuracy is denoted as Acc_0, and the accuracies after passive retraining and with alpha regularization are denoted as Acc_p and Acc_α

TABLE III: Execution times of various DNNs in ProxSim

DNN	total val. time [s]		val. time	# Params.	MAC Ops.
	FP	8b approx.	/batch [s]	x 10e3	x 10e6
Basic-NN	2.45	**2.93**	**0.02**	11.7	0.115
TF-NN	2.81	**3.68**	**0.02**	704.2	0.197
ALL-CNN	7.36	**120.38**	**0.77**	1369.7	281.2
SqueezeNet	468.14	**1051.96**	**1.36**	1235.5	349.6
Resnet 50	464.62	**6424.33**	**8.22**	25636.7	3855.9
VGG16	404.85	**9020.91**	**11.55**	138357.5	15448.6

TABLE IV: Retraining of TF-NN in ProxSim

multiplier	Acc_0	Acc_{end}	epochs	α
Smapprox-2222	96.65	98.32	**3**	1e-10
		98.35	7	0
EvoApprox-74	98.09	98.44	**3**	1e-10
		98.42	4	0

TABLE V: Retraining of ALL-CNN on ProxSim

Multiplier		Acc_0	Acc_p	Acc_α	$\Delta_{\alpha,0}$	$\Delta_{\alpha,p}$
SMApprox	1110	88.64	88.79	**88.88**	0.24	0.09
	3330	88.40	88.66	**88.74**	0.34	0.08
	1111	87.38	**87.66**	87.61	0.23	-0.05
	3333	86.55	87.56	**87.63**	1.08	0.07
	2222	85.58	87.26	**87.36**	1.78	0.10
EvoApprox	118	88.59	88.91	**88.95**	0.36	0.04
	130	88.44	88.89	**89.07**	0.63	0.18
	468	86.63	88.68	**88.71**	2.08	0.03
	230	77.30	88.55	**88.77**	11.43	**0.22**
	74	57.49	87.62	**87.76**	**30.27**	0.14

TABLE VI: Retraining of SqueezeNet on ProxSim

Multiplier		Acc_0	Acc_p	Acc_α	$\Delta_{\alpha,0}$	$\Delta_{\alpha,p}$
SMApprox	1110	44.86	47.18	**50.12**	5.26	**2.94**
	3330	37.19	46.70	**46.95**	9.76	0.25
	1111	39.37	**42.71**	42.52	3.15	-0.19
	3333	32.03	36.73	**37.47**	5.14	0.74
	2222	30.10	33.76	**34.06**	3.96	0.30
EvoApprox	118	55.30	55.10	**55.63**	0.33	0.53
	130	54.50	54.52	**55.01**	0.51	0.49
	468	36.43	46.21	**46.54**	10.11	0.33
	230	13.00	**42.62**	42.60	29.60	-0.02
	74	0.63	30.06	30.26	**29.63**	0.20

respectively. $\Delta_{\alpha,0}$ and $\Delta_{\alpha,1}$ are defined as follows:

$$\Delta_{\alpha,0} = Acc_\alpha - Acc_0 \tag{12}$$

$$\Delta_{\alpha,p} = Acc_\alpha - Acc_p \tag{13}$$

The retraining times are of 3734s. and of 3805s. with passive training and alpha regularization respectively. The execution time overhead of the alpha regularization is of 71 seconds (less than 2% of the passive training time). Furthermore, in 90% of the cases, training with alpha regularization is more effective than passive traning, even for few epochs. This leads to an improvement in DNN accuracy from 0.24% to 30.27 % compared to the initial accuracy, and of up to 0.22% compared to passive DNN retraining. Moreover, by using alpha regularization we were able to reach our accuracy tolerance in just 5 epochs in 60% of the cases, using approximate multipliers with an MRE of up to 8.13% (EvoApprox 230).

3) SqueezeNet: We now evaluate ProxSim for optimization of DNNs for large-scale image recognition. For this, we approximate and retrain SqueezeNet, which delivers a Top-1 accuracy of 56.67% with FP precision, and of 55.78% with quantized values. We perform retraining with 20% of the training dataset; the DNN validation is computed with the entire test dataset. We retrain this CNN for 5 epochs with SGD; a learning rate of 1e-6 and a batch size of 32 are hereby used. When retraining with alpha regularization, a value of $\alpha = 1e-10$ is considered for these experiments. The results are reported in Table VI. As expected, alpha regularization, with total training time of 31,061 sec., delivers better results than passive retraining (29,360 sec.) in 80% of the cases, with an excecution time overhead of 5.8%. Furthermore, with only 20% of the training dataset, we achieve an improvement of $\sim 3\%$ with alpha regularization, with respect to passive retraining, and of almost 30% when compared to the initial accuracy.

V. Conclusion and Outlook

The integration of cross-layer approximations in DNNs can lead to significant improvements in computational resources. This integration can benefit from specialized frameworks that allow to analyze the approximations and to recover from the introduced approximation errors. In this work, we presented ProxSim, a GPU-accelerated simulation framework for cross-layer approximate computing in DNNs, where we integrate existing and novel optimization techniques for approximate DNNs. Through numerous experiments, we demonstrate the effectiveness of ProxSim and our proposed optimization approach not only for accelerating approximate DNN simulation, but also for retraining fully approximated DNNs for small, medium and large-scale image classification tasks.

References

[1] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[2] A. Krizhevsky, "Learning multiple layers of features from tiny images," *University of Toronto*, 2009.

[3] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma *et al.*, "Imagenet large scale visual recognition challenge," *IJCV '15*.

[4] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://www.tensorflow.org/

[5] S. R. Jain, A. Gural, M. Wu, and C. Dick, "Trained uniform quantization for accurate and efficient neural network inference on fixed-point hardware," *arXiv preprint arXiv:1903.08066*, 2019.

[6] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," *ICLR '16*.

[7] Y. Fan, X. Wu, J. Dong, and Z. Qi, "Axdnn: Towards the cross-layer design of approximate dnns," in *ASPDAC '19*.

[8] P. Rek and L. Sekanina, "Typecnn: Cnn development framework with flexible data types," in *DATE '19*.

[9] Q. Zhang *et al.*, "ApproxANN: An approximate computing framework for artificial neural network," in *DATE '15*.

[10] S. Venkataramani *et al.*, "AxNN: Energy-efficient neuromorphic systems using approximate computing," *ISLPED '14*.

[11] X. He *et al.*, "AxTrain: Hardware-oriented neural network training for approximate inference," *ISLPED '18*.

[12] S. Vogel, J. Springer, A. Guntoro, and G. Ascheid, "Self-supervised quantization of pre-trained neural networks for multiplierless acceleration," in *DATE '19*.

[13] J. L. McKinstry *et al.*, "Discovering low-precision networks close to full-precision networks for efficient embedded inference," in *ICLR '19*.

[14] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, 2008.

[15] Y. Bengio, N. Léonard, and A. C. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *ArXiv*, 2013.

[16] M. Courbariaux *et al.*, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *NIPS '15*.

[17] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. A. Riedmiller, "Striving for simplicity: The all convolutional net," *ICLR '14*.

[18] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *ICLR '17*.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CVPR '15*.

[20] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *ICLR '15*.

[21] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina, "EvoApprox8B: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods," in *DATE '17*.

[22] S. Ullah, S. Sripadraj Murthy, and A. Kumar, "SMApproxLib: Library of FPGA-based approximate multipliers," in *DAC '18*.

[23] I. Goodfellow *et al.*, *Deep Learning*. MIT Press, 2016.