Deep Neural Network Acceleration Framework Under Hardware Uncertainty

Mohsen Imani, Pushen Wang, and Tajana Rosing
Computer Science and Engineering, UC San Diego, La Jolla, CA 92093, USA
{moimani, puw001, tajana}@ucsd.edu

***Abstract*—Deep Neural Networks (DNNs) are known as effective model to perform cognitive tasks. However, DNNs are computationally expensive in both train and inference modes as they require the precision of floating point operations. Although, several prior work proposed approximate hardware to accelerate DNNs inference, they have not considered the impact of training on accuracy. In this paper, we propose a general framework called FramNN , which adjusts DNN training model to make it appropriate for underlying hardware. To accelerate training FramNN applies adaptive approximation which dynamically changes the level of hardware approximation depending on the DNN error rate. We test the efficiency of the proposed design over six popular DNN applications. Our evaluation shows that in inference, our design can achieve 1.9× energy efficiency improvement and 1.7× speedup while ensuring less than 1% quality loss. Similarly, in training mode FramNN can achieve 5.0× energy-delay product improvement as compared to baseline AMD GPU.**

I. Introduction

Deep neural networks (DNNs) have been shown a great opportunity to be used in different domains including, natural language processing, computer vision, voice recognition, and health care, and manufacturing [1]–[4]. Although accuracy is an important metric in DNN, in real world the energy efficiency and performance are becoming more important metrics, as many DNN applications want to run on embedded devices with limited resources [4]–[8]. In recent years, growth in the number of smart devices significantly increases the rate of data generation over the world. This increases the demands on machine learning algorithms to preprocess such large data [9], [10]. Learning algorithms usually run on a cloud, since embedded devices do not have enough resources and battery to process costly algorithms. However, it is more efficient, secure and cost effective to run machine learning algorithms on processing edges.

Approximate computing is an efficient way to reduce the cost of computation, by trading efficiency with accuracy. Machine learning applications are stochastic in heart, thus they accept a part of inaccuracy in the computation [11]–[19]. Floating point multipliers are the most computationally expensive units in neural networks [20]–[22]. Prior work used approximate hardware to accelerate DNNs on CPU, GPU or FPGA [23], [24]. The goal of approximate hardware is to provide minimum quality loss while maximizing the efficiency. However, DNNs working on real world data are usually sensitive to approximation. Thus, the hardware cannot benefit from the efficiency that comes from deep approximation.

In this paper, we proposed a novel general framework, called FramNN , which enables DNNs to run on hardware with deep approximation while providing good enough accuracy. FramNN works for any approximate hardware and adapts DNN training models to make it appropriate for underlying hardware. Our design modifies the trained DNN model based on the constraints of an approximate hardware in order to minimize the impact of approximation on DNN accuracy. To accelerate training, we proposed adaptive approximation technique which dynamically changes the level of hardware approximation, depending on the neural network error rate. This technique significantly reduces the average time that hardware work on deep approximation, while providing minimum impact on accuracy. We test the efficiency of the proposed design on six popular DNN applications. Our evaluation shows that in inference, our design can achieve 1.9× energy efficiency improvement and 1.7× speedup while ensuring less than 1% quality loss. Similarly, in training mode, FramNN can achieve 5.0× energy-delay product improvement as compared to precise GPU.

II. Related Work

Neural networks can be adapted to run on a wide variety of hardware, including: CPU, general purpose GPU (GPGPU), FPGA, and ASIC chips [23], [25], [26]. As DNNs benefit from parallelism, significant efforts have been dedicated to utilizing multiple cores. On GPGPUs, neural networks get up to two orders of magnitude performance improvement as compared to CPU implementations.

Prior works attempted to leverage the stochastic properties of DNNs in order to improve efficiency while relaxing the computation [23], [24]. As shown in [24], implementing neural networks in fixed-point quantized numbers improves performance. Similarly, Lin et al. [27] examined the use of trained binary parameters in order to avoid multiplication. Work in [20] also tried to model multiplication in neural neural using a lookup table. However, not all neural network applications can benefit from this approach. Several other designs tried to propose a new computing blocks such as approximate adder, multiplier and multiply-accumulator in order to accelerate the neural network computation. However, all these designs apply approximation on inference while the networks still train on exact hardware. Many applications cannot benefit from this way of approximation, as their accuracy significantly drops over unseen inference data.

Unlike these work, our design adapts the neural network model such that approximation results in minimum loss in quality for any approximate hardware. Our design also uses full floating point precision in training, giving it more flexibility when needed.

Han *et al.* [28], [29] investigated the use of model compression in DNNs. They trained sparse models with shared weights to compress the model*et al.* [28]. The compressed

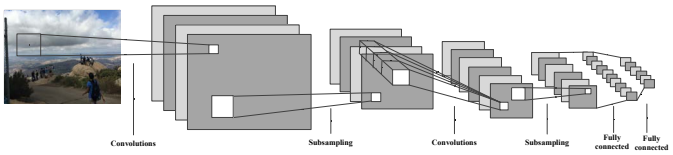

Fig. 1. LeNet-5, a 7 layer convolutional neural network

parameters of [28] are used to design ASIC/FPGA accelerators [29]. Compression fails to improve the implementation in general purpose processors, which require the compressed parameters to be decompressed into the original parameters. Our method is orthogonal to all these previous work, as our design can further reduce power consumption and execution time by enabling adaptive training approximation. In addition, our proposed design uses a general framework to adaptive neural network model to accelerate neural network inference on approximate hardware.

III. Proposed FramNN

A. Neural Network

Figure 1 gives a simple example of Convolutional Neural Network (CNN), called leNet, which is one of the very first CNN design. A typical CNN consists of an input layer, an output layer and multiple hidden layers. CNN may have four main types of layers:

Convolutional layer: is the most basic and important type of layers in CNN, and give the name "Convolutional". It serves as a filter, extracting some features from the original images or from the previous feature maps.

Activation function: is as simple as a nonlinear functions, forcing every pixel to be non-negative, as well as introducing non-linearity.

Pooling layer: The Pooling layer is used as a process of down sampling, the primary purpose is to keep the most important information of the input image and reduce the size at the same time, thus reducing the computational complexity and avoid overfitting.

Fully connected layer: is actually a multi layer Perceptron, and "fully connected" means every neuron in a layer is connected with every neuron in its previous layer as well as its next layer. With the softmax activation function in the output layer as the last one in the fully connected layer, the CNN finally classify the image with the features extracted before.

In the execution process, given input image, the CNN outputs the probability of each class and associates the image with the label with maximum probability. And the training process is the process for the CNN to learn form those input images and adaptively change its parameters in each layer. Training the CNN is actually an optimization problem, and one approach is to iteratively change the weights to minimize the errors with gradient decent. Our goal is to come up with a framework which accelerates neural networks in both training and testing phases, by running them on approximate hardware. Our framework tries to adapt neural network model such that the approximation can have minimum impact on the neural network accuracy.

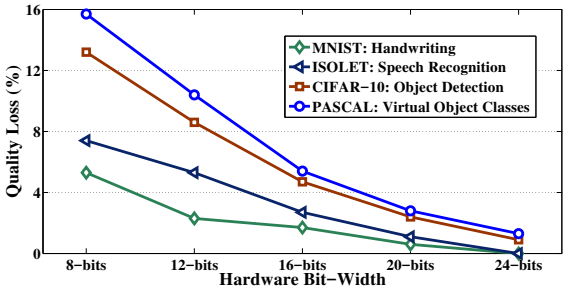

Fig. 2. Quality loss of different DNN applications in different level of hardware approximation.

B. Inference Acceleration on Approximate Hardware

Multiplication is the most expensive neural network computation in inference. There are several designs came up with the new multiplier and multiplier-accumulator to accelerate neural network's inference [23], [30], [31]. However, it seems unreasonable to approximate hardware without modifying the training mode. Although several work tried to show this possibility, our evaluation indicates that the advantage coming from approximation is low.

Let's consider an example of approximate hardware where the multiplications bit-width can change (e.g. using 16-bits multiplier instead of 32-bits). We use such hardware to test the classification accuracy of neural networks running on four popular datasets, MNIST: a handwriting image [32], ISOLET: speech recognition [33], CIFAR-10: object detection [34] and PASCAL: virtual object classes [35]. The configuration of networks running these datasets are listed in Table III. Figure 2 shows the classification accuracy of two networks when the bit-width of multiplier varies from 8-bit to full range (24-bits) in inference. We observed that MNIST can almost achieve maximum accuracy even using 8-bits multiplier. However, working on more complicated datasets such as CIFAR-10 or PASCAL, the network has much higher sensitivity to approximation. For CIFAR-10 and PASCAL datasets, the classification accuracy drops by 13.2% and 15.7% when the bit-width scales to 24 and 16 bits respectively.

Looking closely at DNN functionality, we observed that the network in inference works closely as it is trained for. Therefore, any uncertainty or approximation in hardware in train mode can impact on the inference accuracy. In fact, approximation in inference can significantly decrease the accuracy of DNN, since the network has not been trained for. Considering the above examples, when DNN trains on exact hardware, it is not appropriate to use such trained model to run DNN inference on approximate hardware. This could significantly increase the network inaccuracy since the trained model is not adapted to work under such hardware uncertainty.

For a certain convolutional layer l, the input comes from the output of previous layer $l-1$. Assume the input is an $n \times n$ image, and each pixel denoted by y_{ij}^{l-1}. The outputs are k feature maps, each is the convolution on the input image and a filter (or to say kernel), denoted by matrix $w_{m\times m}^{l}$. The size

of each output feature maps(denoted by x_{ij}^l) will be $n-m+1$.

$$x_{ij}^l = \sum_{a=0}^{m-1}\sum_{b=0}^{m-1} w_{ab}^l y_{i+a,j+b}^{l-1} \quad (1)$$

Just like the parameters in the multi-layer perceptron, let w_{ji}^l denotes the weight from the i^{th} neuron in the l-1^{th} layer to the j^{th} neuron in the l^{th} layer and b^l is the offset parameter. Then, if the input of l^{th} is x_i^l, we have

$$x_j^l = \sigma\left(\sum_i w_{ji} x_i^{l-1} + b_j^l\right) \quad (2)$$

in which $\sigma(\cdot)$ is the activation function, and one of the most common activation function is the *sigmoid* function or the *softmax* function (for output layer). The purpose of back-propagation is to propagate the error from the output layer to the first layer, then update the weights layer by layer to minimize the total classification error.

We propose a novel framework, called FramNN , which adapts DNN training model such that the approximate hardware can have maximum benefit form that. FramNN proposes a general technique that works for any hardware under approximation and generates a trained model which better matches with the underlying hardware. The input of FramNN can be a non-trained DNN or trained DNN model. Our framework modifies that model and generates a new model which perfectly matches with approximate hardware. There are multiple ways for adapt new model for approximate hardware as listed below:

First approach: The first approach is to train DNN on the same approximate hardware which the inference is going to run on. For example, if we plan to use a hardware which has 8-bits multiplier, this approach trains DNN on the hardware with the same approximation. In that case the network has been trained and tested with the same constraints, thus it likely to provide higher accuracy in inference. Although this idea works over a few networks, it has the following disadvantages: (i) it provides very low accuracy over the complicated datasets which require to train using high precision hardware. (ii) It does not allow the hardware to run on deep approximation. To improve the accuracy of this approach, we can train the network on exact mode and then adjust the weights by validating the model over a part of training dataset. This technique provides slightly better accuracy than the first technique, because as we explained, DNNs train based on gradient descent equations which requires full floating point precision in order to work with very small values. In addition, adjusting model on approximate hardware slightly changes the weights and make the model appropriate for approximate hardware.

Second approach: we propose which adapts training model by looking more insight the training procedure. Training consists of two phases: feed-forward and back-propagation. In feed-forward, a training data loads to the network and generates an output based on the current model/weights. Depending on the error in output layer, the network changes the network weights layer by layer in back-propagating using gradient descent equations. The goal of back-propagation is to train the network to minimize fee-forward error at next iterations. Therefore, in order to achieve maximum training accuracy, the hardware uncertainty should be modeled on feed-forward step. Therefore, there should be no approximation on back-propagation step as this step (i) tries to find the best weights for a hardware which runs feed forward. (ii) it works with small values and requires floating point precision for proper training. Thus, we need to model inference approximation on feed-forward step, while back-propagation works in precise hardware. This technique does not compromise the precision that gradient decedent equations require, while provides the model that approximate hardware in inference can enjoy that.

TABLE I
DNN CLASSIFICATION ACCURACY RUNNING ON EXACT AND APPROXIMATE HARDWARE (16-BIT INFERENCE APPROXIMATION).

	MNIST	*ISOLET*	*HAR*	*CIFAR-10*	*CIFAR-100*	*PASCAL*
Exact	98.5%	96.4%	98.3%	85.6%	70.6%	57.7%
Approx.Training	96.8%	93.7%	96.6%	80.9%	65.2%	51.0%
FramNN	98.5%	95.4%	97.8%	84.0%	68.7%	55.5%

Table I shows the classification accuracy of different DNN structures running on exact and approximate hardware over six popular DNN applications. The details of datasets and networks are explained in Section V-A. In approximate mode, the classification accuracy has been calculated using approximate training and proposed FramNN . As we explained, the first approach trains and tests the networks on approximate hardware with 16-bit multiplier. While FramNN trains the networks when the feed forward layer runs on approximate hardware with the same level of approximation. The results indicates that FramNN always outperforms the accuracy of approximate training over all tested datasets. Our evaluation also indicates that FramNN can achieve 2.6% (up to 45%) higher classification accuracy as compared to approximate training over all datasets. In Section V, we will show how this accuracy can significantly improve by adjusting the hardware approximation in training mode.

IV. Optimal Training

There are crucial need to train DNNs on embedded devices. DNNs are trained based on gradient descent equations. We observe that DNNs accuracy changes significantly during training mode. In first training iterations, the DNNs show large error rate. This error reduces during a training such that in a final training iterations the algorithm converges. This training procedure takes significant time and energy of the process. In order to enable approximate training, the approximation needs to be applied on a training portion which has minimum impact on accuracy.

We use an approximate floating point multiplier which can dynamically change the level of approximation. This multiplier adds exponential part of two input value, but instead of multiplying full rang (N-bits) mantissa together, it multiplies a first m-bits of two mantissas. This multiplier can dynamically changes the number of effective mantissa bits. Our design exploits this adaptive hardware to change the level of approximation in training and improve training efficiency while providing the maximum accuracy. Intuitively, at very beginning, we would like to train the model very "roughly", or to say most approximately to cut power consumption and speedup the training procedure. This approximation will have less impact on final DNNs accuracy, since in this mode the network has large error by itself, and the error added by approximation does not significantly impact on DNN accuracy.

TABLE II
EDP IMPROVEMENT AND ACCURACY OF DIFFERENT APPLICATIONS TRAINING ON EXACT AND APPROXIMATE HARDWARE WITH UNIFORM AND LINEAR APPROXIMATION (N=16).

		MNIST	*ISOLET*	*HAR*	*CIFAR-10*	*CIFAR-100*	*PASCAL*
Uniform	*Accuracy*	0.3%	0.7%	0.8%	1.7%	4.7%	4.1%
	EDP	3.4×	2.0×	3.6×	2.7×	2.4×	1.9×
Linear	*Accuracy*	1.4%	1.9%	2.9%	4.4%	13.5%	11.6%
	EDP	6.1×	4.4×	4.8×	4.0×	3.2×	2.7×

At the end of training process, we want more precise training with less bits to mask, at the cost of efficiency, because any error in that iterations, the weight are already adjusted and adding extra error can impact on DNN accuracy. There is a trade-off between the accuracy and power, and the key is to find a way to adaptively change the number of bits to mask to find a balance between them. One approach is to change the level of hardware approximation linearly such that the training starts with maximum approximation and reduce the level of approximation linearly until the training ends in exact mode. Table II lists the accuracy and efficiency of our design when it trains on the exact and approximate hardware. For approximation, we have a hardware which applies uniform and linear approximation during a training. The uniform approximation fixes the level of hardware approximation during training. Our evaluation shows that linear training approximation can result in average 3.9% (up to 7.4%) higher classification accuracy than uniform approximation, while achieving a similar training efficiency.

However, linearly changing the approximation cannot optimally trade accuracy and efficiency. In this paper, we design a novel adaptive strategy which optimally changes the approximation depending on the criterion we define. One of criterion can be minimizing the energy-delay product given the total number of epochs of the training. Let p_i be the power of the system, when i bits mask put on the results of the multiplication, the corresponding number of iterations is denoted as t_i, and let $T = \sum T_i$ as the total number of iterations or epochs. Then the problem can be formulated as the following optimization problem:

$$\begin{aligned} max \quad & \sum p_i t_i^2 \\ s.t. \quad & \sum t_i = T \end{aligned}$$

With Lagrange multiplier, let

$$L(t_i;\lambda) = \sum p_i t_i^2 + \lambda(\sum t_i - T)$$

Then calculate the gradient:

$$\nabla_{t_i} L(t_i;\lambda) = 2p_i t_i + \lambda = 0$$

$$\forall i,\ p_i t_i = constant\ C$$

And then, $t_i = \dfrac{T}{p_i \sum \frac{1}{p_i}}$, that means after training with i bits to mask for t_i iterations, we should switch to $t-i$ bits and training for t_{i-1} iterations until $i = 0$. This algorithm gives us the iteration that the hardware approximation needs to change in order to maximum EDP, while maximizing the accuracy. In Section V, we will show the impact of proposed FramNN on the neural network accuracy and efficiency.

TABLE III
DNN MODELS AND BASELINE ERROR RATES FOR SIX APPLICATIONS.

Dataset	Network Topology	Error
MNIST	784, 512, 512, 10	1.5%
ISOLET	617, 512, 512, 26	3.6%
HAR	561, 512, 512, 19	1.7%
CIFAR-10	$Conv32\times32\times3, Conv32\times3\times3, Pool2\times2,$	14.4%
CIFAR-100	$Conv64\times3\times3, Conv64\times3\times3, 512,\ 10\ (100)$	42.3%
PASCAL	$Conv32\times32\times3, Conv32\times3\times3, Pool2\times2,$ $Conv64\times3\times3, Conv64\times3\times3,\ 512, 1024,\ 1024,\ 20$	29.4%

V. RESULTS

A. Experimental Setup

We integrate configurable FPUs on the AMD Southern Island GPU, Radeon HD 7970 device, a recent GPU architecture with 2048 streaming cores. We use multi2sim, a cycle accurate CPU-GPU simulator for architecture simulation [36] and change the GPU kernel code to enable configurable floating point unit approximation in runtime simulation. We use *Synopsys Design Compiler* to calculate the energy consumption of the balanced FPUs in GPU architecture in 45-nm ASIC flow. We perform circuit level simulations to design configurable FPU using HSPICE simulator in 45-nm TSMC technology. Neural networks are realized using OpenCL, an industry-standard programming model for heterogeneous computing.

We tested the efficiency of the FramNN on six popular machine learning applications. Table III lists the DNN topologies and baseline error rates for the original models. Each DNN model is trained using stochastic gradient descent with momentum [37]. In order to avoid over-fitting, Dropout [38] is applied to fully-connected layers with a drop rate of 0.5. In all the DNN topologies, the activation functions are set to "Rectified Linear Unit" (ReLU), and a "Softmax" function is applied to the output layer.

Handwritten Image Recognition (MNIST): MNIST is a popular machine learning data set including images of handwritten digits [32]. The objective is to classify an input picture as one of the ten digits $\{0 \ldots 9\}$.

Voice Recognition (ISOLET): Many mobile applications require online processing of vocal data. We evaluate lookNN with the Isolet dataset [33] which consists of speech collected from 150 speakers. The goal of this task is to classify the vocal signal to one of the 26 English letters.

Human Activity Recognition (HAR): For this data set, the objective is to recognize human activity based on 3-axial linear acceleration and 3-axial angular velocity that have been captured at a constant rate of 50Hz [39].

CIFAR-10 and CIFAR-100: CIFAR-10 dataset involves classification of different objects [34]. This dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. Dataset is divided into five training batches and test batch contains 1000 randomly-selected images from each class.

PASCAL: Pascal vitual object classes provides standardized image datasets for object class recognition [35]. This dataset contains 20 classes. The train and validation data has 11,530 images containing 27,450 ROI annotated objects and 6,929 segmentations.

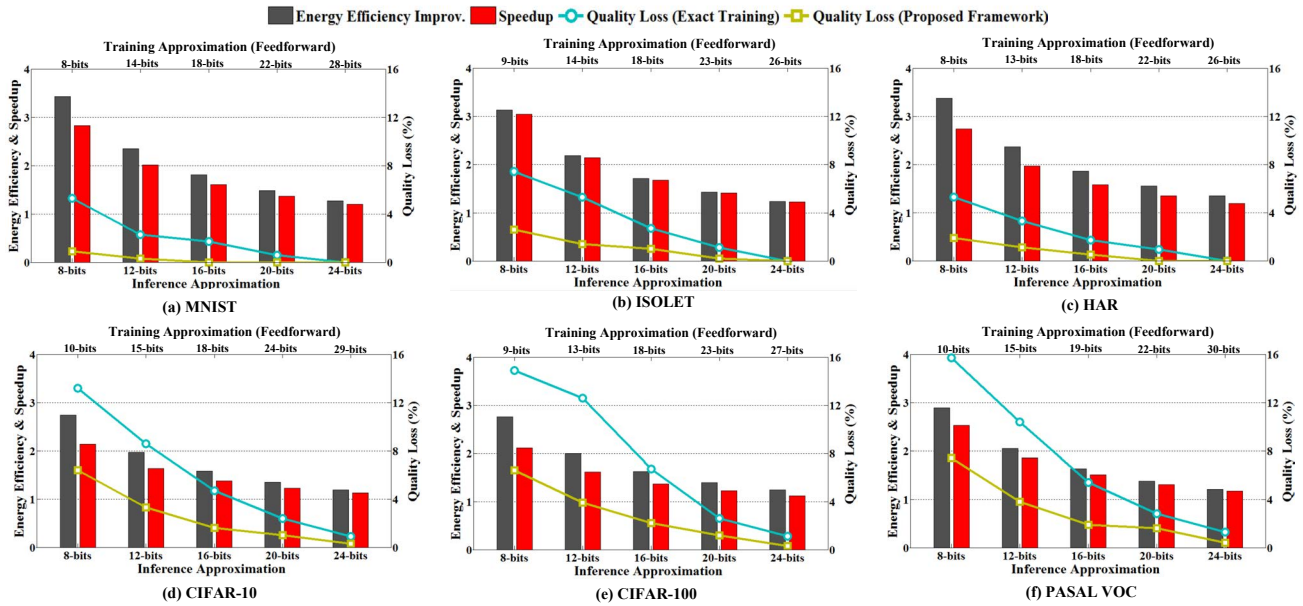

Fig. 3. Energy efficiency improvement, speedup and accuracy of DNN applications in different inference approximation.

B. Inference Efficiency-Accuracy Trade-off

Our goal is to find the best level of feed-forward approximation in training, such that the inference provides maximum accuracy when running on approximate hardware. Our framework starts applying approximation on the feed-forward steps. We observed that in all cases the best level of feed-forward approximation (train mode) is equivalent to the inference approximation. Figure 3 shows the energy efficiency improvement, speedup and classification accuracy of different applications when the inference runs on approximate hardware. We compare our proposed with a conventional technique when the training performs on exact hardware. The bottom and top x-axis in Figure 3 show the level of inference and training approximation (feed-forward) that provide maximum classification accuracy. Our result shows that FramNN provides maximum accuracy when the feed-forward is in slightly deeper approximation than inference. For instance, in order to work with 16-bits hardware precision, MNIST trains on hardware with 18-bits approximation.

The result shows that FramNN accuracy outperforms the accuracy of DNNs training on exact mode. Over MNIST dataset, FramNN can provide the same accuracy as exact hardware when it runs on 16-bits multiplier. Similarly, over CIFAR-100 dataset, our design provides negligible 1.9% quality loss when it runs on 16-bits multipliers. For the same configuration, DNNs trained on exact hardware will have 5.4% error. The result shows that FramNN can achieve on average 1.9× energy efficiency improvement and 1.7× speedup as compare to DNNs in exact mode, while providing less than 1% quality loss. Similarly, accepting 2% quality loss, this energy efficiency and speedup increases to 2.4× and 2.0× respectively.

C. Adaptive Approximate Training

Our design can accelerate training using uniform or adaptive approximate hardware. In linear approximation the hardware precision is fixed during the training, while adaptive approach puts the hardware on deep approximation, then reduces the level of approximation until the final level. This final approximation level has impact on the efficiency of adaptive training. Figure 4 shows energy-delay product (EDP) improvement that adaptive and uniform approximation can provide when the final level of approximation changes. The EDP is normalized to the hardware which trains on exact hardware. In addition, the quality loss defines as an accuracy difference between the network running on precise hardware and proposed approximate hardware. Our evaluation shows that our design can accelerate EDP improvement by 3.6× while still providing the same accuracy as DNNs running on exact hardware. In case of accepting 1% and 2% quality loss, EDP improvement increases to 5.0× (5.8×) respectively. In these configurations, FramNN EDP improvements are 2.1× and 2.3× higher than the results that unifrom approximation provides.

VI. Conclusion

In this paper we propose two novel techniques to accelerate neural network computation in both inference and train modes. Our design exploits the efficiency of hardware in approximate mode to accelerate DNNs computation with minimal impact on accuracy. In inference, our proposed framework adjusts DNN training model to make it appropriate for underlying hardware. In train mode, we proposed adaptive approximation technique which dynamically reduces the level of hardware approximation depending on the neural network error rate. We test the efficiency of the proposed design on six popular DNN applications. Our evaluation shows that in inference, our design can achieve 1.9× energy efficiency improvement and 1.7× speedup while ensuring less than 1% quality loss. Similarly, in training mode, FramNN can achieve 5.0× energy-delay product improvement as compared to precise GPU.

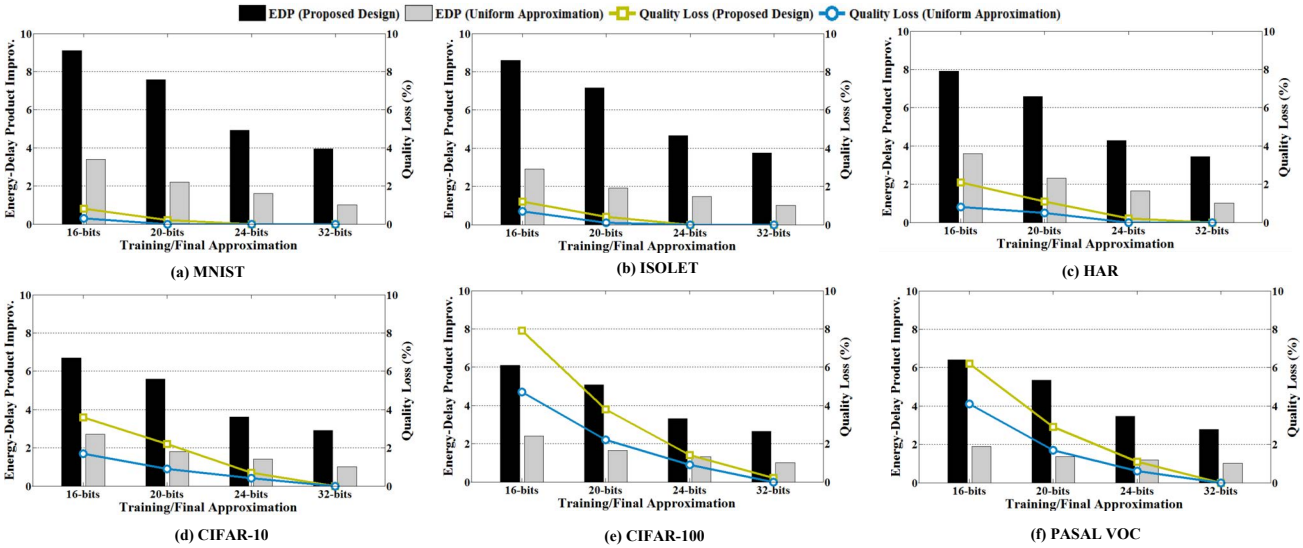

Fig. 4. Energy efficiency improvement, speedup and accuracy of DNN applications in different inference approximation.

VII. Acknowledgment

This work was supported by NSF grants #1730158 and #1527034.

References

[1] B. Yao *et al.*, "Multifractal analysis of image profiles for the characterization and detection of defects in additive manufacturing," *Journal of Manufacturing Science and Engineering*, 2017.
[2] Deng *et al.*, "Recent advances in deep learning for speech research at microsoft," in *ICASSP, 2013*, pp. 8604–8608, IEEE.
[3] Srinivas *et al.*, "Applications of data mining techniques in healthcare and prediction of heart attacks," *IJCSE*, vol. 2, no. 02, pp. 250–255, 2010.
[4] M. Imani *et al.*, "Voicehd: Hyperdimensional computing for efficient speech recognition," in *ICRC*, IEEE.
[5] X. Lei *et al.*, "Accurate and compact large vocabulary speech recognition on mobile devices," in *Interspeech*, vol. 1, 2013.
[6] M. Ghasemzadeh *et al.*, "Resbinnet: Residual binary neural network," *arXiv preprint arXiv:1711.01243*, 2017.
[7] M. Imani *et al.*, "Nngine: Ultra-efficient nearest neighbor accelerator based on in-memory computing," in *ICRC*, IEEE.
[8] Y. Kim *et al.*, "Orchard: Visual object recognition accelerator based on approximate in-memory processing," in *ICCAD*, 2017.
[9] L. L. C. Kasun, H. Zhou, G.-B. Huang, and C. M. Vong, "Representational learning with elms for big data," 2013.
[10] M. Imani *et al.*, "Low-power sparse hyperdimensional encoder for language recognition," *IEEE Design & Test*, vol. 34, no. 6, pp. 94–101, 2017.
[11] M. Imani *et al.*, "Resistive configurable associative memory for approximate computing," in *DATE*, pp. 1327–1332, IEEE, 2016.
[12] M. Imani *et al.*, "Approximate computing using multiple-access single-charge associative memory," *TETC*, 2016.
[13] M. Imani *et al.*, "Multi-stage tunable approximate search in resistive associative memory," *TMSCS*, 2017.
[14] M. Imani *et al.*, "Masc: Ultra-low energy multiple-access single-charge tcam for approximate computing," in *DATE*, pp. 373–378, IEEE, 2016.
[15] M. Imani *et al.*, "Processing acceleration with resistive memory-based computation," in *MEMSYS*, pp. 208–210, ACM, 2016.
[16] M. Imani *et al.*, "Acam: Approximate computing based on adaptive associative memory with online learning," in *ISLPED*, 2016.
[17] J. Sim *et al.*, "Enabling efficient system design using vertical nanowire transistor current mode logic,"
[18] M. Imani *et al.*, "Remam: low energy resistive multi-stage associative memory for energy efficient computing," in *ISQED*, pp. 101–106, IEEE, 2016.
[19] M. Imani *et al.*, "Nvalt: Non-volatile approximate lookup table for gpu acceleration," *Embedded Systems Letters*, 2017.
[20] M. S. Razlighi *et al.*, "Looknn: Neural network with no multiplication," in *DATE*, pp. 1775–1780, IEEE, 2017.
[21] M. Imani *et al.*, "Efficient neural network acceleration on gpgpu using content addressable memory," in *DATE*, pp. 1026–1031, IEEE, 2017.
[22] M. Imani *et al.*, "Canna: Neural network acceleration using configurable approximation on gpgpu," in *ASPDAC*, IEEE, 2018.
[23] V. Mrazek *et al.*, "Design of power-efficient approximate multipliers for approximate artificial neural networks," in *Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on*, pp. 1–7, IEEE, 2016.
[24] D. Lin *et al.*, "Fixed point quantization of deep convolutional networks," *arXiv preprint arXiv:1511.06393*, 2015.
[25] C. Zhang *et al.*, "Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks," in *ACM ICCAD*, p. 12, ACM, 2016.
[26] Y. Wang *et al.*, "Deepburning: Automatic generation of fpga-based learning accelerators for the neural network family," in *IEEE/ACM DAC*, IEEE, 2016.
[27] Z. Lin *et al.*, "Neural networks with few multiplications," *arXiv preprint arXiv:1510.03009*, 2015.
[28] Han *et al.*, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR, abs/1510.00149*, vol. 2, 2015.
[29] S. Han *et al.*, "Eie: efficient inference engine on compressed deep neural network," *arXiv preprint arXiv:1602.01528*, 2016.
[30] D. Kim *et al.*, "A power-aware digital multilayer perceptron accelerator with on-chip training based on approximate computing," *IEEE Transactions on Emerging Topics in Computing*, 2017.
[31] S. S. Sarwar *et al.*, "Multiplier-less artificial neurons exploiting error resiliency for energy-efficient neural computing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pp. 145–150, IEEE, 2016.
[32] Y. LeCun *et al.*, "The mnist database of handwritten digits," 1998.
[33] "Uci machine learning repository." http://archive.ics.uci.edu/ml/datasets/ISOLET.
[34] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.
[35] M. Everingham *et al.*, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, no. 2, pp. 303–338, 2010.
[36] R. Ubal *et al.*, "Multi2sim: a simulation framework for cpu-gpu computing," in *PACT*, pp. 335–344, ACM, 2012.
[37] I. Sutskever *et al.*, "On the importance of initialization and momentum in deep learning.," *ICML (3)*, vol. 28, pp. 1139–1147, 2013.
[38] N. Srivastava *et al.*, "Dropout: a simple way to prevent neural networks from overfitting.," *JMLR*, vol. 15, no. 1, pp. 1929–1958, 2014.
[39] "Uci machine learning repository." http://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones.