CANNA: Neural Network Acceleration using Configurable Approximation on GPGPU

Mohsen Imani, Max Masich, Daniel Peroni, Pushen Wang, Tajana Rosing
CSE Department, UC San Diego, La Jolla, CA 92093, USA
{moimani, mmasich, dperoni, puw001, tajana}@ucsd.edu

***Abstract*—Neural networks have been successfully used in many applications. Due to their computational complexity,it is difficult to implement them on embedded devices. Neural networks are inherently approximate and thus can be simplified. In this paper, CANNA proposes a gradual training approximation which adaptively sets the level of hardware approximation depending on the neural network's internal error, instead of apply uniform hardware approximation. To accelerate inference, CANNA's layer-based approximation approach selectively relaxes the computation in each layer of neural network, as a function its sensitivity to approximation. For hardware support, we use a configurable floating point unit in Hardware that dynamically identifies inputs which produce the largest approximation error and process them instead in precise mode. We evaluate the accuracy and efficiency of our design by integrating configurable FPUs into AMD's Southern Island GPU architecture. Our experimental evaluation shows that CANNA achieves up to 4.84× (7.13×) energy savings and 3.22× (4.64×) speedup when training four different neural network applications with 0% (2%) quality loss as compared to the implementation on baseline GPU. During the inference phase, our layer-based approach improves the energy efficiency by 4.42× (6.06×) and results in 2.96× (3.98×) speedup while ensuring 0% (2%) quality loss.**

I. Introduction

Internet of Things (IoT) applications typically analyze raw data by running machine learning algorithms [1]. Sending all the data to the cloud for processing is not scalable, cannot guarantee real-time response, and is often not desirable due to privacy and security concerns [2]. Therefore, machine learning algorithms may need to run, at least partially, on mobile and embedded devices to classify, cluster, or process the data at the edge of the Internet.

Neural networks (NNs) are very effective for image processing, video segmentation, detection and retrieval, speech recognition, computer vision, and gaming [3], [4], [5]. NNs exploit learned knowledge to deal with data which they have not previously encountered. Although NNs can outperform many other machine learning models, they require enormous resources to be executed. Many applications require NNs to be executed on embedded devices. On the other hand, many NN applications need to update their model at run-time in order to adapt to the environment or enable a personalization. For instance, in speech recognition, NNs personalize as a function of the user's context or accent [6]. Due to limited processing resources and power budgets, training and testing NNs has not been done on constrained embedded devices.

Most current computing systems deliver only exact solutions at high energy cost, while many algorithms, such as neural networks, do not require exact answers, due to their statistical nature [7], [8], [9]. Slight inaccuracy due to enabled HW approximation in neural networks often results in little to no quality loss. Prior work attempted to accelerate neural network by enabling approximation [10], [11]. These prior designs are application specific, as the hardware could not adapt the level of approximation at run-time. Moreover, these designs enable approximation on all input data regardless of their sensitivity to approximation, potentially yielding less accurate overall results that might be possible otherwise.

In this paper, we propose CANNA, a configurable approximate computing platform which significantly accelerates neural networks in both training and inference phases by exploiting their stochastic behavior. For training, we propose a Gradual Training Approximation (GTA) which significantly accelerates neural network computation, while providing a desirable quality of service. GTA starts training from deep approximation, and gradually reduces the level of approximation as a function of NN internal error, until the accuracy is sufficient. For inference acceleration, we propose a layer-based approximation which selectively relaxes the computation in each layer of the neural network, based on their impact on accuracy. We use a hardware configurable floating point unit (FPU) which can tune the level of approximation at runtime. We evaluate the accuracy and the efficiency of our design by integrating configurable FPUs into AMD's Southern Island GPU architecture. Our experimental evaluation shows that GTA achieves up to 4.84× (7.13×) energy savings and 3.22× (4.64×) speedup when running four different neural network applications with 0% (2%) quality loss as compared to baseline GPU. During the inference phase, our layer-based approach improves the energy efficiency by 4.42× (6.06×) and results in 2.96× (3.98×) speedup while ensuring 0% (2%) quality loss.

We evaluate the accuracy and efficiency of our design by integrating configurable FPUs into AMD's Southern Island GPU architecture. Our experimental evaluation shows that GTA achieves up to 4.84× (7.13×) energy savings and 3.22× (4.64×) speedup when running four different neural network applications with 0% (2%) quality loss as compared to the implementation on baseline GPU. During the inference phase, our layer-based approach improves the energy efficiency by 4.42× (6.06×) and results in 2.96× (3.98×) speedup while ensuring 0% (2%) quality loss.

II. Related Work

Neural networks can be adapted to run on a wide variety of hardware, including: CPU, GPGPU, FPGA, and ASIC

chips [9], [12], [11], [13], [14]. Because they benefit from parallelization, a significant effort has been dedicated to utilizing multiple cores. On GPGPUs, neural networks get up to two orders of magnitude performance improvement as compared to CPU implementations [15].

Prior works attempted to leverage the stochastic properties of neural networks in order to relax the computation accuracy and improve the implementation efficiency [7], [11], [16], [17]. As shown in [16], implementing neural networks in fixed-point quantized numbers improves performance. Similarly, Lin et al.[10] also examined the use of trained binary parameters in order to avoid multiplication altogether. However, not all applications can handle this approach. Modifications of neural networks parameters during training require higher precision and have difficulties with additive quantization noise [18]. Unlike these works, our design allows the use of full floating point precision, giving it more flexibility when needed.

Han *et al.* [19], [20] investigated the use of model compression in NNs. They trained sparse models with shared weights to compress the model*et al.* [19]. The compressed parameters of [19] are used to design ASIC/FPGA accelerators [20]. 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 this previous work, as our design can further reduce power consumption and execution time by enabling gradual and adaptive approximation. In addition, our proposed design uses a general hardware-software platform which accelerates neural network on CPU, GPU, FPGA, and even ASIC, by enabling configurable FPU approximation.

Approximate Computing: There are several approaches to enable approximation in computing: Voltage over scaling (VOS), the use of approximate hardware blocks, and approximate memory units [21], [22], [23]. VOS dynamically reduces the voltage supplied to hardware to save energy, at the expense of accuracy [24], [25], [26], [27]. Error rates for VOS can be modeled to determine the trade-off between energy and accuracy for different applications, allowing voltage to be lowered until an error threshold is reached [28], [29], [25], [30]. However, circuits are sensitive to variations, and if the operating voltage of a circuit is decreased too far, timing errors, which may be too large to correct, begin to appear. Approximate hardware involves redesigning basic component blocks to save energy, at the cost of output accuracy [21], [7], [31], [32]. Liu *et al.* utilizes approximate adders to create an energy efficient approximate multiplier [7]. Hashemi *et al.* designed a multiplier that multiplies with a reduced number of bits to conserve power [21]. Camus *et al.* propose a speculative approximate multiplier that combines gate-level pruning and an inexact speculative adder to lower both the power consumption and FPU area [31]. Approximation is another way to improve the efficiency in-memory computation [33], [34]. Prior designs work at a fixed level of accuracy, whereas we use configurable and adaptive approximate floating point multiplier which can approximately and adaptively process data at run-time.

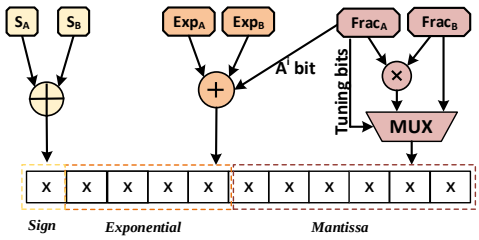


Fig. 1. Configurable approximate multiplication between *A* ad *B* operands.

III. Approximate Floating Point Unit

In a floating point operation, the mantissa multiplication takes the bulk of the processing power and energy consumption. work in [32] improves the characteristics of the FPU by completely removing the costly mantissa multiply. As shown in Figure 1, rather than multiplying the two values together, this work selects one of the original mantissa values and use it directly as the output. Because the range of the mantissa is 1 to 2, the maximum possible error on the output on this approach is 100%. However, we can easily reduce the maximum error to 50% by utilizing the first bit of the discarded mantissa and adding it to the exponent value of the product. When the first bit of the mantissa is 1, the mantissa value ranges between 1.5 and 2. In this case, by incrementing the exponent, the mantissa is effectively halved, with a range from 0.75 to 1. When the first bit of the mantissa is 0, the range is from 1 to 1.5. By utilizing the first bit of the mantissa, the range of the mantissa values goes from 0.75 to 1.5, with a maximum error of 50%.

We also reduce the error by adaptively selecting when multiply is calculated by the FPU at runtime. The threshold value that selects when exact computation is need is defined with a number of bits, n, where each additional bit that is evaluated reduces the maximum approximate error by half. When we used the first mantissa bit for approximation, the error increases the closer the mantissa is to 1.5, which is represented by a 1 followed by all 0s when stored in memory. Therefore, when the first bit is a 0, we examine the next n bits for a 1 and discard the approximate answer if a 1 appears. Likewise, if the first bit is a 1, we examine the next n bits for a 0 and discard the approximate answer should a 0 be found. When a value is discarded, the FPU will run in exact mode and compute the value. By discarding the approximate values with the greatest error, the majority of calculations can be run in approximate mode, while a smaller percentage still runs exactly, resulting in a significantly reduced overall error. This approach minimizes error while still maintaining a large energy improvement. Our design provides drastic improvements to the GPU, because the FPUs are the slowest component of the GPU architecture and consume the most energy during computation.

We integrate the approximate FPU design into the existing AMD Southern Island GPU architecture by modifying its implementations [32]. The GPU architecture consists of 32 computing units, each with four SIMD (single instruction, multiple data). Each SIMD consists of 16 lanes, and has both

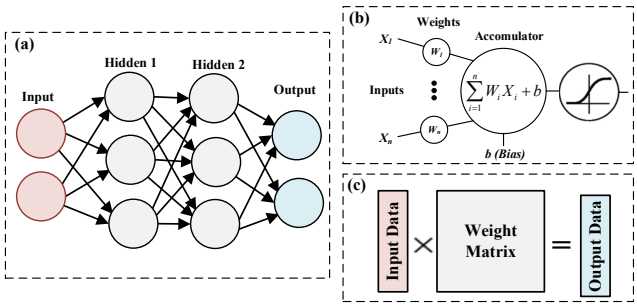


Fig. 2. (a) Neural network structure with two hidden layers, (b) computing model of each neuron and (c) matrix multiplication representation between two NN layers.

integer and floating point units. We replace the multiplication FPU in all GPU cores with our configurable FPUs. Our approximate FPUs run in approximate mode when the error is sufficiently low and otherwise calculated results exactly to maximize the overall accuracy.

IV. Neural Network Acceleration

A. Neural Network in Training & Inference

Figure 2a shows the overall structure of a neural network consisting of input, output and hidden layers. The input data dimension and the number of output classes determines the number of neurons in the input and output layers respectively. The number and size of hidden layers depends on the network topology. As Figure 2b shows, in neural networks, each neuron is a small processing unit with one or more inputs and a single output. Each input has an associated weight determining the strength of the input data. The neuron simply multiplies inputs with their weights and adds them to calculate an output. Finally, the output value passes through an activation function, which is historically a *Sigmoid* function. Neural networks have training and testing phases. During the first training iterations, weights and biases are assigned random values. The training phase finds the best weight values which result in maximum classification accuracy. To find such weights, input data (from the training dataset) passes to the network in a feed forward fashion. Based on the errors measured at the output stage, the network adapts the weights and biases values in back propagation mode. When the network is trained, the trained weights and biases can be used to classify the inputs in he dataset. Two fully connected neural network layers have a huge number of multiplications between them. Figure 2c shows that these operations can be modeled as matrix multiplication, where each row of the matrix represents the weights corresponding to each neuron. The output of each neuron can be computed as:

$$x^i = f(\sum_k W_k^i * x_k^{i-1} + b^i) \tag{1}$$

where multiplications exist between the output of neurons in $i-1^{th}$ layer and the weight matrix in i^{th} layer, W^i, and f is an activation function. Each layer has its own bias vector, b^i. This vector adds to the output signal of each neuron. In back propagation, the i^{th} neural network layer has N inputs and M outputs. The error δ propagates backwards from the output to update the weights and the bias value. Here is the gradient descent equation for updating the weight and bias values:

$$\Delta W = \eta[\delta \odot h^{'}(W*x+b)]x^T \tag{2}$$

$$\Delta b = \eta[\delta \odot h^{'}(W*x+b)] \tag{3}$$

where x is input to the layer, η is learning rate, and $\odot$ shows the element-wise multiplication. After updating the weights and bias value, *delta* also needs to be updated using:

$$\delta = (W^T\delta) \odot h^{'}(W*x+b) \tag{4}$$

In Equation 2, the outer product of $h^{'}(W*x+b)$ and x is the main multiplication cost. The input x has higher potential for bounding rather than $h^{'}(W*x+b)$, since the second term is determined by the cost function and network parameter. The term $(W*x+b)$ is not computed again in back propagation, as this term was previously calculated in the forward pass and can be reused. Similar to other machine learning algorithms, neural networks are stochastic in nature, meaning that they accept a part of inaccuracy in their computation. The goal of this paper is to exploit this scholastic behavior to accelerate both training and inference of a neural network by enabling configurable approximation.

Algorithm 1: Gradual Training Approximation (GTA)

```
inputs: NN Parameters, Training Data, Iter_max, Apx_min
outputs: NN Trained Model
Initialize weights and biases to random values
Initialize Approx-level
for iter = 1 ... iter_max do
    out_iter = feed_forward (input)
    P = error_estimation (out)
    if isConveraged(P) & isApprox(Apx_min) then
        Break
    else
        approx_configurator(P, iter, iter_max, Apx_min)
    end
    back_propagate(ΔW, Δb, δ_l)
end
```

B. CANNA Acceleration During Training

1) Uniform Training Approximation: It is essential to use the precision of floating point units (FPU) for neural network training, as FPUs cover a wide range of numbers appearing during the back propagation. Training neural networks on such approximate hardware accelerates the process while maintaining the desired level of accuracy. The level of approximation is user and application dependent. For example, for an easy image classification task (e.g. MNIST Handwritten digits [35]), training on hardware with deep approximation might provide the same quality of service that hardware with light approximation can provide over more complex datasets (e.g. *ImageNet* [36]). Thus, there is not a fixed optimal approximation level which is acceptable for all applications.

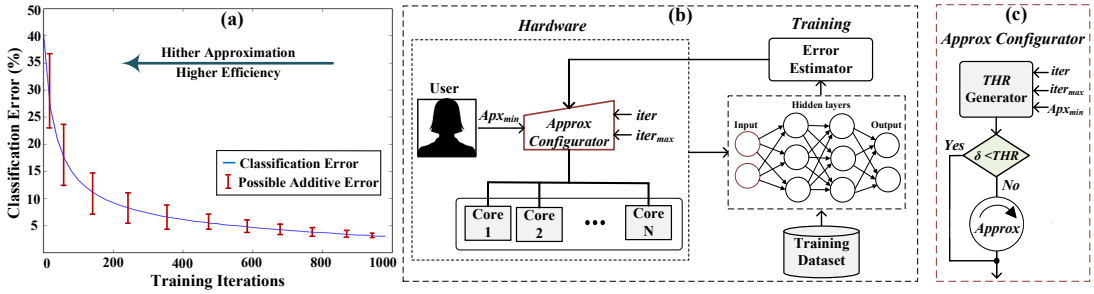


Fig. 3. (a) MNIST classification accuracy in different training iterations (b,c) GTA framework to accelerate neural network training by enabling Adaptive approximation.

TABLE I
QUALITY LOSS, NORMALIZED ENERGY CONSUMPTION AND EXECUTION TIME OF NEURAL NETWORK RUNNING ON GPGPU WITH DIFFERENT LEVEL OF APPROXIMATION (TUNING BITS).

Approximation	Exact	4-bit	3-bit	2-bit	1-bit	0-bit
Quality loss (Δe_{train})	0%	0%	0.9%	1.3%	1.7%	3.2%
Norm.Energy	1	0.31	0.25	0.21	0.18	0.12
Norm. Execution	1	0.45	0.38	0.34	0.31	0.19

In order to generalize the existing core so it accelerates the neural network during the training phase, we use our approximate configurable FPU. Depending on the running application and its accuracy needs, our framework changes the level of hardware approximation to an optimal level.

Table I shows the impact of uniform approximation on the quality loss, energy consumption, and execution time of a neural network when evaluating the MNIST Handwritten digit dataset. This network consists of four fully connected layers with 784, 500, 500, and 10 neurons in each layer, respectively. The application classifies handwritten digitsinto ten different classes, 0–9 . Quality loss is an additive error, defined as the classification error of neural network training on full precision and approximate FPUs:

$$\Delta e_{train} = e_{Approx} - e_{FPU}$$

This result shows that increasing the level of hardware approximation does not automatically imply a degradation in classification accuracy. For example, for this network, our design works with the same accuracy as a full range 32-bit FPU, even when it trains with only four configuration tuning bits. At this level of approximation the FPUs are running 33% of the time in approximate mode (Hit-rate=33%), resulting in a training speedup of 2.2× and energy efficiency improvement of 3.2× as compared to the hardware with full FPU precision. Increasing the level of approximation to no tuning bits further accelerates the training by 5.1× and results in 7.9× energy efficiency improvement with 3.2% quality loss.

2) Gradual Training Approximation: Neural network training accuracy changes during the training phase. During the first training iteration, the weights are assigned randomly, resulting in larger classification error. These weights adapt during the training phase using stochastic gradient decent. Figure 3a shows the error rate of the MNIST dataset over 1000 training iterations. The red line in this graph illustrates

Algorithm 2: Layer-based Inference Approximation

inputs: NN Parameters, Validation Data, Test Data, *QoS*
outputs: NN Layer Configuration
Initialize weights and biases based on trained model
for $i = 1 \ldots N$ **do**
  config(i)=approx$[(l_1 \ldots l_{i-1}, l_{i+1} \ldots l_n)$=Apx$_0$, $l_i = Apx_{max}]$
  out_i= *feed_forward (validation-data, config(i))*
  $\mathscr{C}_i$=*error_estimation* (out_i)
end
for $Apx(j) = Apx_{max} \ldots Apx_{min}$ **do**
  $\mathscr{S}_j$=*selective_approx*$(\mathscr{C}_i, Apx(j))$
  out_j= *feed_forward (test-data,* $\mathscr{S}_j$*)*
  $\mathscr{E}_j$=*error_estimation* (out_j)
  if *(QoS ¿* $\mathscr{E}_j$*)* **then**
    save_config$(\mathscr{S}_j)$
    Break
  end
end

the visual range of approximation that a network can accept during the training. The error reduces significantly during early training iterations, but then saturates. Different error rates while training suggests that uniform approximation is not the best method for approximation because all training iterations do not have the same impact on the final network classification accuracy. During the first iterations, the weights are fairly random, so accepting large approximation should not impact the final classification accuracy. However, during the last iterations, the weights are clsoe to optimal, thus even small hardware approximation may degrade the classification accuracy noticeably.

We propose a gradual training approximation framework, called GTA, which accelerates the neural network training. Our framework, shown in Figure 3b, starts the training from the hardware with maximum level of approximation (zero tuning bits). Then, it updates the level of hardware approximation at each iteration, as the network converges. The convergence controls when the slope of accuracy improvement is less than a threshold value, THR. This THR value adaptively changes in our design to ensure that training completes with acceptable accuracy by the final iteration ($iter_{max}$). As Figure 3c shows,

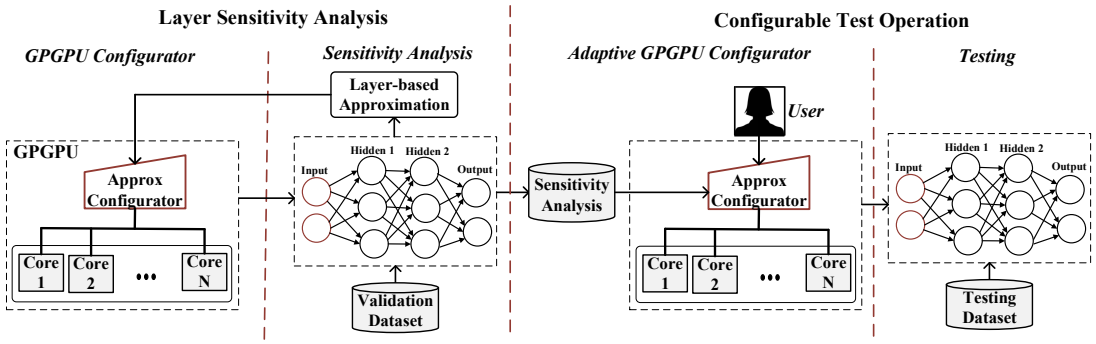


Fig. 4. Framework to accelerate neural network inference by enabling layer-based approximation.

our framework updates the THR value by using four inputs: (i) the current training iteration $iter$, (ii) the maximum number of iterations ($iter_{max}$), (iii) neural network error (δ), and (iv) the maximum hardware precision (Apx_{min}). Based on the updated THR, GTA checks the convergence in each training iteration by measuring the slope of classification accuracy over last 50 iterations. If the slope is smaller than a THR value, our framework reduces the level of approximation by a single step.

Algorithm 1 outlines the details of our gradual training approximation. The first steps assign the weights and biases random values and set the hardware approximation to the maximum level (Apx_{max}). The iterative training starts by feed forward (line 6), where input patterns pass through NN and generate the output class. Our algorithm estimates the network error (line 7) and checks for the convergence (line 8). If training converges and the hardware is at the most precise level (Apx_{min}, defined by user), the algorithm terminates. Otherwise, it updates the level of approximation (line 11) and performs back propagation for the next training iteration (line 13). This iterative procedure terminates when either the network converges during the final hardware approximation, or the iteration reaches the maximum ($iter_{max}$).

C. Inference Acceleration

Similar to the training phase, the inference phase can be accelerated by enabling approximation. However, there is no notion of iterations during testing, as NNs use the same trained model to classify all inputs of the dataset. Uniform approximation is one possible technique to apply inference approximation, which was explored in prior work [7], [11], [16]. This technique performs inference on the hardware with a fixed level of approximation. We select the approximation which provides the desired accuracy during the inference over the validation set. However, this is not the best approach to apply approximation, as it does not consider the impact of individual NN layers. In a neural network, separate layers have different sensitivity to approximation. For example, the first NN layer has higher sensitivity, as it directly works on the input data, thus any approximation can change the network input. Similarly, the NN classification accuracy has high dependency on the output layer, as approximation of this layer can change the output class. The middle neural network layers show much lower sensitivity to approximation.

Based on this observation and configurability of our hardware, CANNA proposes a layer-based inference approximation which selects a different approximation level for each layer depending on their impact on the classification accuracy. Figure 2 shows the inference framework supporting sensitivity analysis and layer-based approximation. This framework first finds the impact of each layer approximation on the final NN accuracy. Algorithm 2 shows the procedure of layer-based inference approximation. The approximation consists of two parts: sensitivity analysis and selective layer-based approximation. During the sensitivity analysis, the program iterates through the network layers and puts one layer on maximum level of approximation (zero tuning bits) at a time, while each other layer is set to exact mode. In this configuration, the feed-forward measures the sensitivity. Analysis continues for all network layers while running the validation dataset. The result of this sensitivity analysis is a vector, $\mathscr{C} = [\mathscr{C}_1 \ldots \mathscr{C}_N]$, where each element represents the sensitivity of one NN layer to approximation. Based on this vector, we enable selective hardware approximation in each NN layer, from the maximum level and then decrease it until the network satisfies quality of service (QoS) defined by user. The last level of approximation obtained is saved as the best network configuration. While changing the neuron approximation, the ratio of layers approximation needs to remain close to the same as that obtained during the sensitivity analysis. However, since the approximation is controlled by a number of tuning bits, our design needs to quantize the sensitivity value to the hardware with the closest approximation level.

Table II shows the sensitivity of different NN layers for the MNIST dataset. This result shows that the first and the last layers have the highest sensitivity to approximation, while the hidden layers could have up to 3× lower sensitivity. The table also shows a few possible configurations that NN layers can take in approximate mode. When the first NN layer works with m-bit level approximation, the second to last NN layers need to be configured with $m-1$, $m-2$ and m tuning bits respectively. With $m = 6$, the network runs 51% of time in approximate mode and provides 4.4× energy savings and 2.9× speedup, with zero quality loss compared to the NN runing on exact hardware. These improvements are respectively 1.6× and 1.8× higher than running NN with uniform approximation.

The layer-based approximation can be easily implemented

TABLE II
LAYER SENSITIVITY AND HARDWARE CONFIGURATION OF NEURAL NETWORK OVER MNIST DATASET

	Layer 1	**Layer 2**	**Layer 3**	**Layer 4**	Δe_{test}
Sensitivity	1	0.52	0.34	0.89	_
Config1	6-bits	5-bits	4-bits	6-bits	0%
Config2	3-bits	2-bits	1-bits	3-bits	0.9%
Config3	2-bits	1-bit	0-bits	2-bits	3.2%

TABLE III
BASELINE NN AND THE TRAINING AND TESTING ERROR OVER FOUR APPLICATIONS

Application	**Network Topology** (l^0, l^1, l^2, l^3)	$e_{train}(\%)$	$e_{test}(\%)$
MNIST	784, 500, 500, 10	0.3	2.4
ISOLET	617, 500, 500, 26	0.7	4.4
HYPER	200, 500, 500, 9	0.9	6.6
HAR	561, 500, 500, 12	0.2	3.4

on multi-core hardware such as GPGPU or FPGA. For a network with N layers, the GPGPU cores need to assign the approximation of cores to at most N different levels. The configuration of all GPGPU cores can be set simultaneously depending what NN layer is assigned to them.

V. EXPERIMENTAL 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 [37] 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 application of proposed design on four general neural network applications:Handwritten Image Recognition (MNIST) [35], Voice Recognition (ISOLET) [38], Hyperspectral Imaging (HYPER) [39], Human Activity Recognition (HAR) [40]. Table III lists the baseline neural network topologies running four applications and their error rates for train and test modes. For each of the four data sets, we compare the baseline accuracy of the train and inference phases with those when using the proposed CANNA framework. We compare the designs in terms of run time and power consumption. Stochastic gradient descent with momentum [41] is used for training. The momentum is set to 0.1, the learning rate is set to 0.001, and a batch size of 10 is used. Dropout [42] with drop rate of 0.5 is applied to hidden layers to avoid over-fitting. The activation functions are set to "Rectified Linear Unit" clamped at 6. A "Softmax" function is applied to the output layer.

B. Training Accuracy-Efficiency

Here we compare the classification accuracy of different NN applications training on GPGPU with uniform and GTA approximation. In uniform mode, the hardware approximation is fixed during the training mode, while GTA changes the approximation adaptively depending on the training error (explained in Section IV-B). Table IV shows the configuration of different applications training on uniform and GTA approximation, providing 0% to 4% quality loss (Δe_{train}). For uniform approximation, the table shows the number of tuning bits in hardware which provides the desired accuracy. For GTA, the level approximation is set by defining the final level of hardware approximation. The table also shows the approximation hit rate, which is the ratio of running FPUs in approximate mode to the total accesses, for each configuration. For instance, for the MNIST application the uniform approximation provides 0% quality loss using 4 tuning bits which results in a roughly 36% approximation hit rate. For the same quality of service, GTA adaptively changes the tuning bits from 0 to 5 bits, resulting in average 70% approximation hit rate.

Figure 5 shows the energy efficiency improvement and the speedup of different applications running on GPGPU with uniform and GTA approximation. The results are normalized to GPGPU using exact 32-bit FPUs. Our experimental results shows that at the same level of accuracy, the GTA always outperforms the efficiency of the uniform approximation. This higher efficiency of comes from GTA ability to put the GPGPU in approximate mode for higher portion of time (as compared to uniform approximation, as shown in Table IV). The result shows that ensuring 0% additive error, GTA (uniform) design can achieve 3.86× (2.26×) energy efficiency improvement and 2.62× (1.62×) speedup as compared to exact mode. For GTA (uniform) design, this improvement increases to 4.84× and 6.11× (2.99% and 4.04%) in energy efficiency and 3.23× and 4.01× (2.07× and 2.37×) in performance accepting 1% and 2% additive errors.

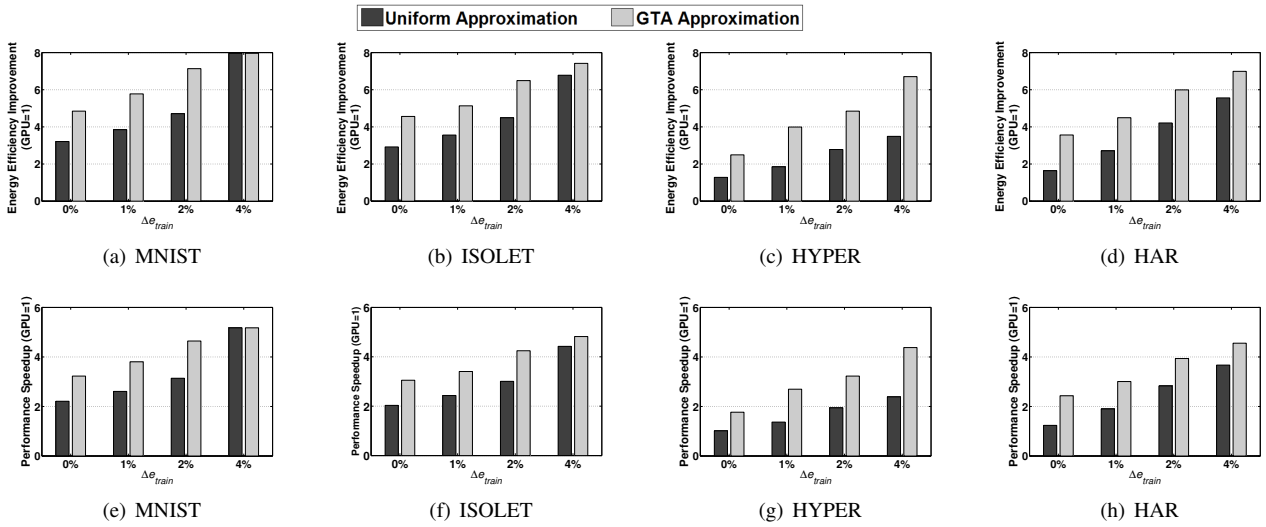
C. Testing Accuracy-Efficiency

Similar to the training, the neural network testing can also be accelerated on approximate hardware. We adjust the threshold value to test the efficiency of our design in different approximation levels. We consider the accuracy that NN can achieve in test mode when the approximation is applied uniformly or with a layer-based approach. In uniform approximation, all layers are approximated at the same accuracy level, while the layer-based approach sets NN layers' approximation based on the results obtained from sensitivity analysis. For each application, Table V shows the sensitivity of each neural network layer to approximation for each application (Note that NNs are trained on exact GPGPU).

Figure 6 shows the energy efficiency improvement and performance speedup of applications running on uniform and layer-based approximation when the network is tested on the hardware configuration listed in Table V. The x-axis shows the maximum acceptable test error (Δe_{test}) for each application. The results are normalized to the energy and performance of traditional GPGPU using 32-bit FPU. The experimental result shows that the layer-based (uniform) approximation could achieve 3.27× (1.86×) higher energy efficiency and 2.25× (1.38×) speedup as compared to GPGPU architecture,

TABLE IV
CONFIGURATION OF DIFFERENT NEURAL NETWORKS RUNNING ON GPGPU WITH UNIFORM AND GTA APPROXIMATION, PROVIDING DIFFERENT QUALITY OF SERVICE.

Applications		**MNIST**				**ISOLET**				**HYPER**				**HAR**			
Training Error (Δe_{train})		**0%**	**1%**	**2%**	**4%**	**0%**	**1%**	**2%**	**4%**	**0%**	**1%**	**2%**	**4%**	**0%**	**1%**	**2%**	**4%**
GTA	*Apx_{min}*	5-bits	4-bits	1-bits	0-bits	7-bits	4-bits	2-bits	1-bit	8-bits	6-bits	5-bits	3-bits	8-bits	6-bits	3-bits	2-bits
	Approx Hit Rate	56%	69%	88%	100%	52%	60%	79%	92%	23%	44%	56%	82%	38%	51%	72%	86%
Uniform	*Tuning bits*	4-bits	3-bits	1-bit	0-bits	5-bits	3-bits	2-bits	1-bit	8-bits	5-bits	4-bits	3-bits	6-bits	4-bits	2-bits	1-bit
	Approx Hit Rate	33%	42%	54%	100%	29%	38%	51%	83%	6%	14%	27%	37%	11%	26%	47%	66%



(a) MNIST (b) ISOLET (c) HYPER (d) HAR

(e) MNIST (f) ISOLET (g) HYPER (h) HAR

Fig. 5. Energy efficiency improvement and speedup of different NN applications training on GPGPU with uniform and GTA approximation.

TABLE V
SENSITIVITY OF EACH NEURAL NETWORK LAYER TO APPROXIMATION USING LAYER-BASED APPROACH.

	Layer 1	**Layer 2**	**Layer 3**	**Layer 4**
MNIST	1	0.52	0.34	0.89
ISOLET	1	0.57	0.65	0.83
HYPER	0.92	0.71	0.59	1
HAR	1	0.48	0.41	0.92

while providing the same accuracy as exact GPGPU. Further relaxing the GPGPU computation significantly improves the energy and performance of GPGPU, specially in layer-based design. For example, accepting 1% (2%) quality loss, layer-based approximation could improve the energy efficiency and speedup of the GPGPU by 4.05× and 2.73× (5.25× and 3.48×) respectively.

D. Scalability and Overhead

CANNA is a general framework which can accelerate several supervised machine learning algorithms which have iterative training procedure or layer-based inference structure. CANNA is a scalable design in terms of the neural network size and supports the approximation on both convolution and fully connected layers. If the number of neurons surpasses the number of GPGPU cores, our design sequentially runs the network and configures the cores at run-time accordingly. Our design is able to reconfigure all cores in a single GPU cycle with negligible impact on the neural network training execution. In inference, CANNA sensitivity analysis and adaptive approximation performs just once at offline over all applications, which results in a negligible overhead. In terms of area, our evaluation shows that the proposed configurable FPU can be designed by adding less than 2.6% area overhead to the existing FPU. This area is negligible considering 7.13× energy savings and 4.64× speedup that CANNA can provide.

VI. CONCLUSION

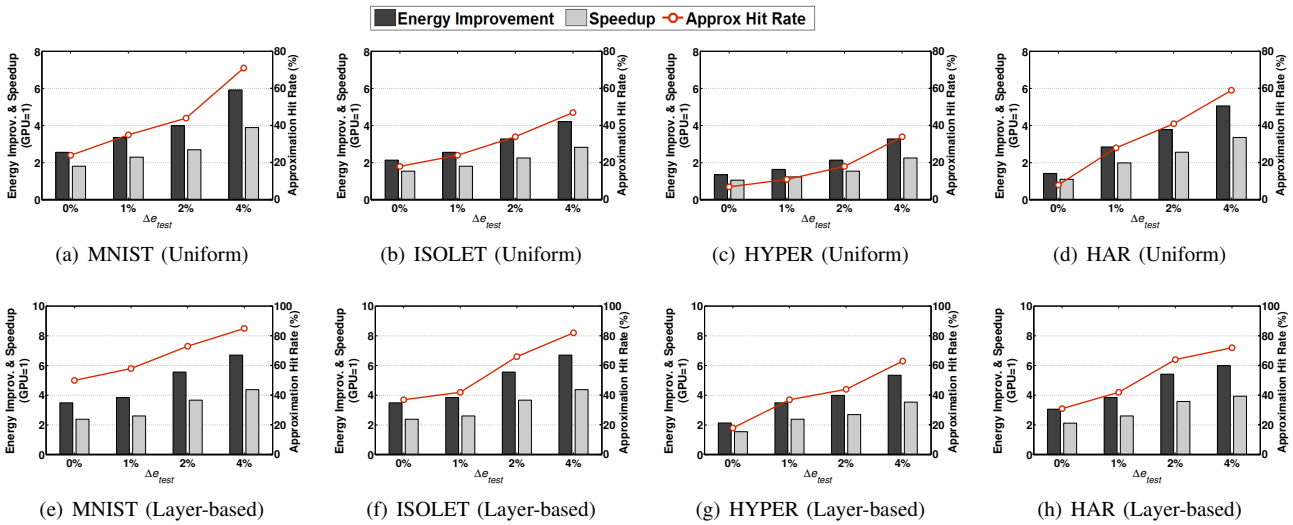
In this paper we propose CANNA, a novel framework to accelerate neural network computation in both training and inference modes by enabling configurable approximation. CANNA supports gradual approximate training which enables the hardware approximation adaptively based on on the network accuracy. Our framework starts the training from deep approximation then changes this level adaptively based on the neural network error rate. For inference, CANNA proposes a layer-based approach which enables approximation on neural network layers based on their sensitivity to approximation. Our experimental evaluation shows that CANNA can achieve up to 7.13× (6.06×) energy savings and 4.64× (3.98×) speedup training (testing) of four different neural network applications with less than 2% quality loss.

VII. ACKNOWLEDGMENT

This work was supported by NSF grants 1730158 and 1527034.

REFERENCES

[1] Y.-K. Chen, "Challenges and opportunities of internet of things," in *ASPDAC*, pp. 383–388, IEEE, 2012.



(a) MNIST (Uniform) (b) ISOLET (Uniform) (c) HYPER (Uniform) (d) HAR (Uniform)

(e) MNIST (Layer-based) (f) ISOLET (Layer-based) (g) HYPER (Layer-based) (h) HAR (Layer-based)

Fig. 6. Energy efficiency improvement, speedup and approximation hit rate of different NN applications testing on GPGPU with uniform and layer-based approximation.

[2] A.-R. Sadeghi *et al.*, "Security and privacy challenges in industrial internet of things," in *IEEE/ACM DAC*, pp. 1–6, IEEE, 2015.

[3] M. Oquab *et al.*, "Learning and transferring mid-level image representations using convolutional neural networks," in *IEEE CVPR*, pp. 1717–1724, 2014.

[4] M. S. Razlighi *et al.*, "Looknn: Neural network with no multiplication," in *IEEE DATE*, pp. 1775–1780, IEEE, 2017.

[5] M. Imani *et al.*, "Efficient neural network acceleration on gpgpu using content addressable memory," in *IEEE DATE*, pp. 1026–1031, IEEE, 2017.

[6] N. Lane *et al.*, "Can deep learning revolutionize mobile sensing?," in *HotMobile*, pp. 117–122, ACM, 2015.

[7] C. Liu *et al.*, "A low-power, high-performance approximate multiplier with configurable partial error recovery," in *DATE*, p. 95, IEEE, 2014.

[8] C.-H. Lin *et al.*, "High accuracy approximate multiplier with error correction," in *IEEE ICCD*, pp. 33–38, IEEE, 2013.

[9] M. Samragh *et al.*, "Customizing neural networks for efficient fpga implementation," in *FCCM*, pp. 85–92, IEEE, 2017.

[10] Z. Lin *et al.*, "Neural networks with few multiplications," *arXiv preprint arXiv:1510.03009*, 2015.

[11] V. Mrazek *et al.*, "Design of power-efficient approximate multipliers for approximate artificial neural networks," in *IEEE/ACM ICCAD*, p. 7, 2016.

[12] C. Zhang *et al.*, "Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks," in *ACM ICCAD*, p. 12, ACM, 2016.

[13] Y. Wang *et al.*, "Deepburning: Automatic generation of fpga-based learning accelerators for the neural network family," in *IEEE/ACM DAC*, IEEE, 2016.

[14] B. D. Rouhani *et al.*, "Curtail: Characterizing and thwarting adversarial deep learning," *arXiv preprint arXiv:1709.02538*, 2017.

[15] D. C. Ciresan *et al.*, "Flexible, high performance convolutional neural networks for image classification," in *IJCAI*, vol. 22, p. 1237, Barcelona, Spain, 2011.

[16] D. Lin *et al.*, "Fixed point quantization of deep convolutional networks," *arXiv preprint arXiv:1511.06393*, 2015.

[17] S. Venkataramani *et al.*, "Axnn: energy-efficient neuromorphic systems using approximate computing," in *ACM/IEEE ISLPED*, pp. 27–32, ACM, 2014.

[18] Lin *et al.*, "Overcoming challenges in fixed point training of deep convolutional networks," *arXiv preprint arXiv:1607.02241*, 2016.

[19] Han *et al.*, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR, abs/1510.00149*, vol. 2, 2015.

[20] S. Han *et al.*, "Eie: efficient inference engine on compressed deep neural network," *arXiv preprint arXiv:1602.01528*, 2016.

[21] S. Hashemi *et al.*, "tldrum: A dynamic range unbiased multiplier for approximate applications," in *ICCAD*, pp. 418–425, IEEE Press, 2015.

[22] M. Imani *et al.*, "Exploring hyperdimensional associative memory," in *IEEE HPCA*, pp. 445–456, IEEE, 2017.

[23] S. M. Seyedzadeh, R. Maddah, A. Jones, and R. Melhem, "Leveraging ecc to mitigate read disturbance, false reads and write faults in stt-ram," in *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*, pp. 215–226, IEEE, 2016.

[24] M. Imani *et al.*, "Resistive configurable associative memory for approximate computing," in *DATE*, pp. 1327–1332, IEEE, 2016.

[25] V. Gupta *et al.*, "Impact: imprecise adders for low-power approximate computing," in *ISLPED*, pp. 409–414, IEEE Press, 2011.

[26] M. Imani *et al.*, "Masc: Ultra-low energy multiple-access single-charge tcam for approximate computing," in *DATE*, pp. 373–378, IEEE, 2016.

[27] M. Imani, S. Patil, and T. Rosing, "Approximate computing using multiple-access single-charge associative memory," *IEEE Transactions on Emerging Topics in Computing*, 2016.

[28] P. K. Krause *et al.*, "Adaptive voltage over-scaling for resilient applications," in *DATE*, pp. 1–6, IEEE, 2011.

[29] M. Imani, A. Rahimi, P. Mercati, and T. Rosing, "Multi-stage tunable approximate search in resistive associative memory," *IEEE Transactions on Multi-Scale Computing Systems*, 2017.

[30] M. Imani *et al.*, "Acam: Approximate computing based on adaptive associative memory with online learning," in *ISLPED*, 2016.

[31] V. Camus *et al.*, "Approximate 32-bit floating-point unit design with 53% power-area product reduction," in *ESSCIRC*, pp. 465–468, IEEE, 2016.

[32] M. Imani *et al.*, "Cfpu: Configurable floating point multiplier for energy-efficient computing," in *IEEE/ACM DAC*, p. 76, ACM, 2017.

[33] Y. Kim *et al.*, "Orchard: Visual object recognition accelerator based on approximate in-memory processing," in *ICCAD*, IEEE, 2017.

[34] M. Imani *et al.*, "Ultra-efficient processing in-memory for data intensive applications," in *IEEE/ACM DAC*, p. 6, ACM, 2017.

[35] Y. LeCun *et al.*, "The mnist database of handwritten digits," 1998.

[36] A. Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," in *NIPS*, pp. 1097–1105, 2012.

[37] R. Ubal *et al.*, "Multi2sim: a simulation framework for cpu-gpu computing," in *PACT*, pp. 335–344, ACM, 2012.

[38] "Uci machine learning repository." http://archive.ics.uci.edu/ml/datasets/ISOLET.

[39] "Hyperspectral remote sensing scenes." http://www.ehu.eus/ccwintco/index.php?title=Hyperspectral_Remote_Sensing_Scenes.

[40] "Uci machine learning repository." http://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones.

[41] I. Sutskever *et al.*, "On the importance of initialization and momentum in deep learning.," *ICML (3)*, vol. 28, pp. 1139–1147, 2013.

[42] N. Srivastava *et al.*, "Dropout: a simple way to prevent neural networks from overfitting.," *JMLR*, vol. 15, no. 1, pp. 1929–1958, 2014.