Hardware-Software Co-design to Accelerate Neural Network Applications

MOHSEN IMANI, RICARDO GARCIA, SARANSH GUPTA, and TAJANA ROSING,
University of California San Diego

Many applications, such as machine learning and data sensing, are statistical in nature and can tolerate some level of inaccuracy in their computation. A variety of designs have been put forward exploiting the statistical nature of machine learning through approximate computing. With approximate multipliers being the main focus due to their high usage in machine-learning designs. In this article, we propose a novel approximate floating point multiplier, called CMUL, which significantly reduces energy and improves performance of multiplication while allowing for a controllable amount of error. Our design approximately models multiplication by replacing the most costly step of the operation with a lower energy alternative. To tune the level of approximation, CMUL dynamically identifies the inputs that produces the largest approximation error and processes them in precise mode. To use CMUL for deep neural network (DNN) acceleration, we propose a framework that modifies the trained DNN model to make it suitable for approximate hardware. Our framework adjusts the DNN weights to a set of "*potential weights*" that are suitable for approximate hardware. Then, it compensates the possible quality loss by iteratively retraining the network. Our evaluation with four DNN applications shows that, CMUL can achieve 60.3% energy efficiency improvement and 3.2× energy-delay product (EDP) improvement as compared to the baseline GPU, while ensuring less than 0.2% quality loss. These results are 38.7% and 2.0× higher than energy efficiency and EDP improvement of the CMUL without using the proposed framework.

CCS Concepts: • **Computer systems organization** → **Architectures**; • **Computing methodologies** → *Machine learning*;

Additional Key Words and Phrases: Approximate computing, neural network, floating point unit, energy efficiency

ACM Reference format:
Mohsen Imani, Ricardo Garcia, Saransh Gupta, and Tajana Rosing. 2019. Hardware-Software Co-design to Accelerate Neural Network Applications. *J. Emerg. Technol. Comput. Syst.* 15, 2, Article 21 (April 2019), 18 pages.
https://doi.org/10.1145/3304086

This work was partially supported by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, and also NSF grants #1730158 and #1527034.
Authors' addresses: M. Imani, R. Garcia, S. Gupta, and T. Rosing, University of California San Diego, 9500 Gilman Dr, La Jolla, CA 92093, USA; emails: {moimani, rag023}@ucsd.edu, sgupta@eng.ucsd.edu, tajana@ucsd.edu.

1550-4832/2019/04-ART21 $15.00
https://doi.org/10.1145/3304086

1 INTRODUCTION

In 2015, the number of smart devices around the world exceeded 25 billion. This number is expected to double by 2020 (Atzori et al. 2010; Gantz and Reinsel 2011). Many of these devices have batteries with strict power constraints, so the need for systems that can efficiently handle the computing requirements of data-intensive workloads is undeniable (Ji et al. 2012; Khoshavi et al. 2016). Deep neural networks (DNNs) have been effectively used for diverse classification problems, such as image processing, video segmentation, speech recognition, computer vision, health-care, and manufacturing (Hinton et al. 2012; Imani et al. 2018b, 2018c; LeCun et al. 2010; Oquab et al. 2014; Salamat et al. 2018). Running DNNs on the general purpose processors is slow, energy hungry, and prohibitively expensive (Krizhevsky et al. 2012). Machine-learning applications are stochastic at heart, thus they do not need highly accurate computation. So by accepting slight inaccuracy, instead of doing all computation precisely, we can get significant energy and performance improvements (Han and Orshansky 2013; Imani et al. 2016d). Therefore, many traditional and state-of-the-art computing systems use floating point units (FPUs) (Courbariaux et al. 2014; Razlighi et al. 2017). For such algorithms of high energy and performance high power is required. To cover the same dynamic range, the fixed point unit must be 5 times larger and 40% slower than a corresponding floating point (Liang et al. 2003). Similarly, many DNN applications require floating-point precision due to the fact that the iterative training algorithm often update the parameters using gradients whose values are too small to sustain the additive quantization noise (Lin and Talathi 2016).

Multiplication is one of the most common and costly FP operations, slowing down the computation in many applications such as signal processing, neural networks, and streaming processes (Imani et al. 2016c; Suhre et al. 2013). Multiplication cost can be reduced by designing an approximate multiplication unit. Most of prior work attempted to reduce the bit-size of multiplication to enable approximation (Hashemi et al. 2015; Narayanamoorthy et al. 2015). However, either the lack of accuracy tuning or the large area requirements of the tuned designs significantly reduce the advantages provided by such approximation.

In this article, we instead propose a configurable floating point multiplication, called CMUL, which significantly improves the multiplication energy consumption by trading off accuracy. CMUL avoids the costly multiplication when calculating the fractional part of a floating point number by adding the input mantissas, instead of multiplying them. To tune the level of accuracy, our design checks the number of consecutive 0's and 1's on the first N bits of both input mantissas. The larger sequence of continuous bit, the higher accuracy CMUL multiplication can achieve. To use CMUL for DNN acceleration, we propose a framework that modifies the trained DNN model to make it suitable for approximate hardware. Our framework adjusts the DNN weights to a set of "*potential weights*" that are suitable for approximate hardware. Then it compensates the possible quality loss by iterative retraining the network based on the existing constraints. We evaluate the efficiency of the proposed approach on AMD GPU architecture by replacing the conventional FPUs with the proposed CMUL. Our evaluations on four DNN applications show that CMUL can achieve on average 60.3% energy efficiency and 3.2× energy-delay product (EDP) improvement as compared to the baseline GPU, while ensuring less than 0.2% quality loss. These results are 38.7% and 2.0× higher energy efficiency and EDP improvement of the CMUL without using the proposed framework.

The rest of article is organized as follow: Section 2 and Section 3 review the related work and background. Section 4 describes the proposed approximate multiplications. Section 6 describes the supported framework to accelerate neural network applications on approximate hardware. The hardware integration has been described in Section 5. The experimental results are presented in Section 7. Finally, Section 8 concludes the article.

2 RELATED WORK

2.1 Approximate Computing

There are several commonly examined approaches to approximate computing: voltage over scaling (VOS), use of approximate hardware blocks, and use of approximate memory units. VOS involves dynamically reducing the voltage supplied to a hardware component to save energy, but at the expense of accuracy. Error rates for VOS can be modeled to determine the tradeoff between energy and accuracy for applications, allowing voltage to be lowered until an error threshold is reached (Imani et al. 2017c; Krause and Polian 2011). However, the circuit is sensitive to any variations, and if the operating voltage of a circuit is decreased too far, timing errors begin to appear.

Another strategy is the application of Non-volatile memories (NVM) to create approximate memory units for energy efficient storage and computing purposes (Gnawali et al. 2018; Imani et al. 2016d; Kim et al. 2015). In computing, the goal of this approach is to store common inputs and their corresponding outputs. This style of associative memory can retrieve the closest output for given inputs to reduce power consumption (Imani et al. 2016a, 2016b; Peroni et al. 2019). This approach does not work well in applications without a large number of the redundant calculations. Associative memory can be integrated into FPUs reduce these redundancies.

Approximate hardware involves redesigning basic component blocks to save energy, at the cost of accurate output (Camus et al. 2016; Hashemi et al. 2015; Lin and Lin 2013; Liu et al. 2014). Liu et al. utilize approximate adders to create an energy efficient approximate multiplier (Liu et al. 2014). Hashemi et al. designed a multiplier that selects a reduced number of bits used in the multiplication to conserve power (Hashemi et al. 2015). Camus et al. propose a speculative approximate multiplier combines gate-level pruning and an inexact speculative adder to lower power consumption and shrink FPU area (Camus et al. 2016).

All the methods adopt to operation accuracy needed at runtime. They only have one level of approximation that is independent of the inputs. In contrast to previous work, we design a configurable approximate floating point multiplier that approximately processes data using an input mantissa directly in the output. In addition, we propose a framework that fixes one of the multiplication operands in neural network to significantly reduce the error of approximate hardware.

2.2 Neural Network

Modern neural network algorithms are executed on diverse types of processors such as GPU, FPGAs, and ASIC chips (Ciresan et al. 2011; Han et al. 2016; Iandola et al. 2016; Imani et al. 2017b; Nazemi et al. 2018; Razlighi et al. 2017). Prior work tried to use fixed-point quantized numerals to improve the efficiency of DNN (Lin et al. 2016). Work in Lin et al. (2015) exploited trained binary parameters to avoid multiplication. However, many applications require floating-point precision, since the iterative DNN training algorithm often update the parameters using gradients whose values are too small to sustain the additive quantization noise (Lin and Talathi 2016). In contrast, our proposed design uses floating-point precision rather than confining the parameters to binary numerals.

Other efficient way to improve the DNN efficiency is model compression. For example, work in Han et al. (2015) trained sparse models with shared weights to compress the model. The compressed parameters of Han et al. (2015) can be used to realize ASIC/FPGA accelerators (Han et al. 2016). However, compression does not help with execution on general purpose processors, in which case the compressed parameters should be decompressed into the original parameters. Dimensionality reduction is investigated for efficient execution of DNNs (Imani et al. 2018a). These methods are orthogonal to our proposed CMUL, since CMUL only reduces the cost of hardware computation with minimal impact of quality loss.

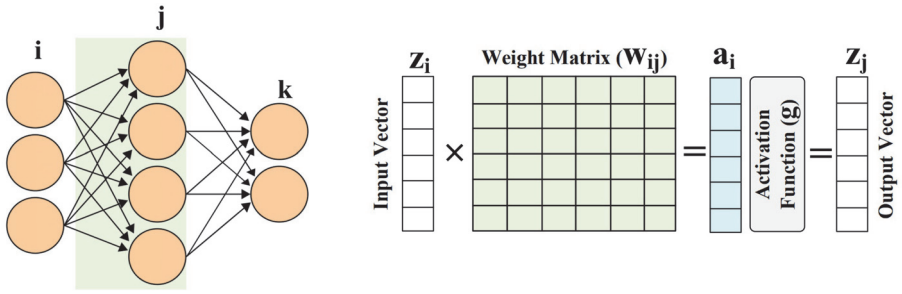

Fig. 1. General structure of DNN in fully connected layer.

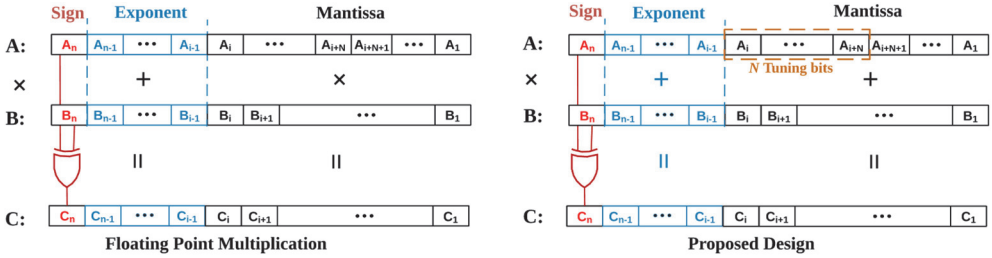

Fig. 2. Approximate multiplication of proposed CMUL between A ad B operands.

3 DEEP NEURAL NETWORKS

A DNN model consists of multiple layers that have multiple neurons. These layers are stacked on top of each other in a hierarchical formation; that is, the output of each layer is forwarded to the next layer. The output of the last layer is used for inference. Figure 1 shows the structure of a fully connected layer in a neural network. The computation in a single layer of neural network can be modeled as a vector-matrix multiplication, which involves large amount of multiplications. However, floating point operations are costly and energy hungry. Multiplication is the most commonly used floating point operation in both learning and multimedia applications (Han and Orshansky 2013; Hashemi et al. 2015). For example, looking at image filters such as the *Sobel filter* and the *Robert filter*, we observed that about 85% of floating point arithmetic involve multiplication. The neuron takes a vector of neuron values from the preceding layer $\mathbf{X} = \langle X_0, \ldots, X_n \rangle$ and then computes its output as follows:

$$Z_j = \varphi\left(\sum_{i=1}^{n} W_{ij}\dot{X}_i + b\right),$$

where W_i and X_i correspond to a weight and an input, respectively; b is a bias parameter; and φ is a nonlinear activation function.

3.1 IEEE 754 Standard

In floating point notation, a number consists of three parts: a sign bit, an exponent, and a fractional value. In *IEEE 754* floating point representation, the sign bit is the most significant bit, where bits 31 to 24 hold the exponent value, and the remaining bits contain the fractional value, also known as the mantissa. The exponent bits represent a power of two ranging from −127 to 128. The mantissa bits store a value between 1 and 2, which is multiplied by 2^{exp} to give the decimal value.

FPU multiply follows the steps shown in Figure 2. First, the sign bit of $A \times B = C$ is calculated by XORing the sign bit of the A and B operands. Second, the effective value of the exponential terms

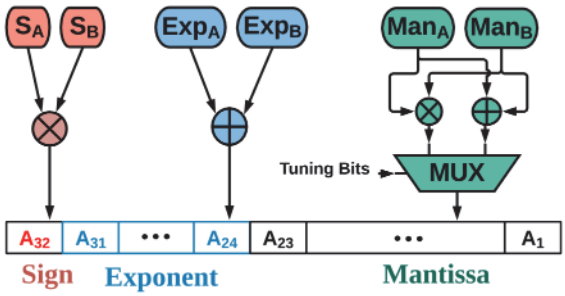

Fig. 3. CMUL Integration with N tuning bits.

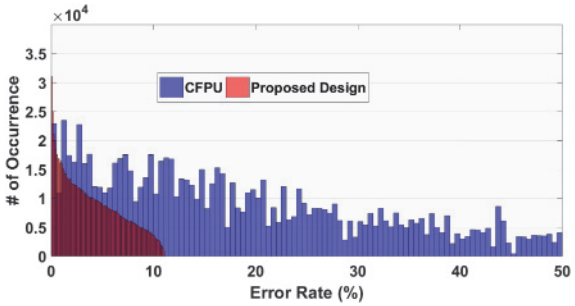

Fig. 4. Histogram of error distribution for proposed design and CFPU.

are added together. Finally, the two mantissa values are multiplied to provide the result's mantissa. Because the mantissa ranges from 1 to 2, the output of the multiplication always fall between 1 and 4. If the output mantissa is greater than 2, then it is normalized by dividing by 2 and increasing the exponent by 1.

3.2 Limitations

Recently, work in Imani et al. (2017a) proposed a configurable floating point multiplier (CFPU), which adaptively multiplies the input operands. CFPU decides to run the multiplication in exact or approximate mode depending on the input mantissas. However, CFPU relies on one input to approximate, which results in errors that range from 0% to 50% works in approximate mode only when one of the input mantissa has N leading one or zero bits (N is a tuning bits). This reduces the number of inputs that CFPU can process in approximate mode. In Section 4, we explain the functionality of the proposed approximate multiplier, then in Section 6, we explain the framework this approximate multiplier to accelerate DNN on GPU architecture.

4 PROPOSED APPROXIMATE MULTIPLIER

In floating point multiplication, the mantissa multiplication is the most costly operation that takes about 80% of the total multiplication energy (Imani et al. 2017a). Here we propose CMUL to accelerate floating point multiplication by eliminating the costly mantissa multiplication. Our design XORs two input sign bits to get the output sign bit. The two input exponents are added to calculate the output exponent. Finally, instead of multiplying the two mantissas, we add two mantissas and used the result as the mantissa for the output (as shown in Figure 3). The result shows that when we replaced the mantissa multiplication with addition, the error rate is less than or equal to 11.1%. Figure 4 shows the error distribution of 1 million random approximations executed by CMUL and

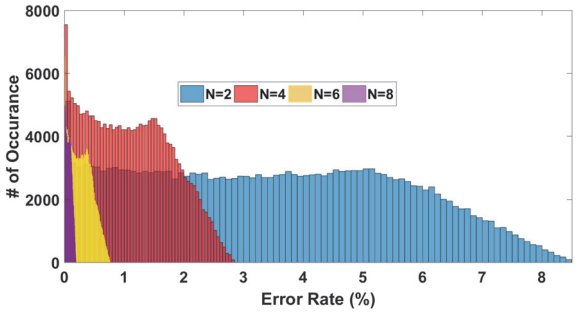

Fig. 5. Accuracy distribution of proposed design as the # of consecutive 1's or 0's changes.

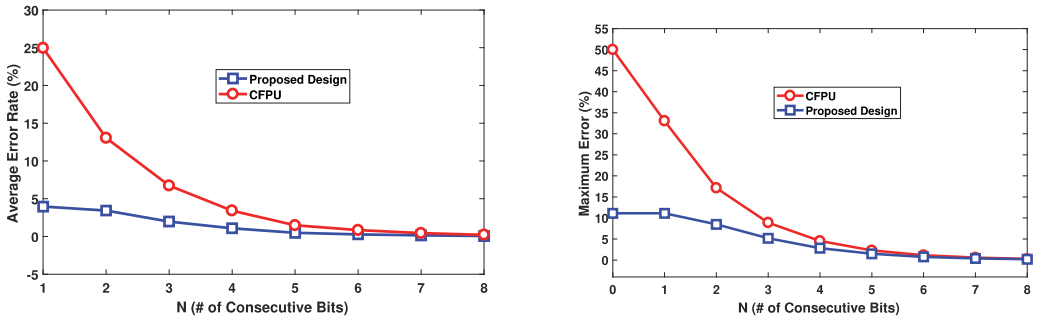

Fig. 6. (a) Average Error Rate and (b) maximum Error of CMUL using different N tuning bits.

CFPU. The result shows that CMUL has higher accuracy than the CFPU with no tuning. Due to a lower error rate without tuning, our design is able to approximate a large amount of numbers, resulting in speedup and energy efficiency improvement.

4.1 Tuning Accuracy

Although proposed approximate multiplication provides high energy savings, the accuracy of computation is heavily affected depending on the application. For some applications, with quantized inputs, e.g., the *Sharpen filter*, the proposed design can work precisely with no quality loss. In addition, many recognition algorithms, such as motion tracking and plate detection applications, only need to quantify changes in the input data. Therefore, the approximate multiplication can be nearly exact for such applications.

To ensure the desired accuracy is achieved we design a tuning method that allows the design to operate only when the approximation is at the desired error rate. The tuning process consists of checking the N number of consecutive 0's or 1's in both of the input mantissas if one of the inputs has the minimum required N value the design will operate in approximation mode. Figure 5 shows the error distribution of random approximations as the value of N changes. The data show that as N increases exponentially the error rate decreases as the number of consecutive 1's or 0's found in one of the input mantissa.

To show the level of accuracy that can be achieved with the proposed design, random inputs with different N values were generated and input into the CFPU and CMUL to compare maximum and average error rates. Figure 6 shows that as N increases the error rate goes to zero for both maximum error and average error rate for both designs. The data also show that the proposed

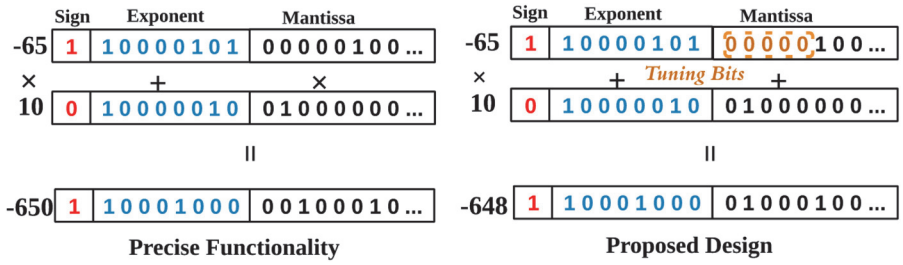

Fig. 7. An example of 32-bit multiplication in conventional precise FPU and proposed CMUL using $N = 5$ tuning bits.

design is far better in both average error rate and maximum error. Comparing both designs, the CFPU has a larger maximum and average error rate for low N values, whereas the proposed design has a significantly lower maximum and average error rate for low N values. This is significant, since the lower the N value the more inputs the approximation design can approximate. Thus the proposed design can handle a greater number of inputs than the CFPU with higher accuracy that will result in less energy and higher speeds, since more multiplications would be able to be approximated.

An example of CMUL multiplication is shown in Figure 7 for two 32-bit floating point numbers in precise FPU and proposed CMUL with $N = 5$. The conventional FPU finds the product of $A = -65$ and $B = 10$ by adding the exponents and multiplying the two mantissa, while XORing the sign bit to find three parts of the output data. In contrast, our design first checks both of the input mantissas for N consecutive 0's or 1's if one of the mantissas contains the desired or exceeds the desired number of consecutive 0's or 1's. In this example, since the input operand mantissa of A has a leading zero in the mantissa the N number of consecutive 0's is checked. In this example, $N = 5$ and the mantissa of A also has five consecutive 0's the design will proceed with the approximation. However, if the set value of N was larger then 5, then the design would run the exact mode instead of approximation. In application, N is selected based on the maximum error rate the application can tolerate, with accuracy increasing with higher N value; however, if a lower or higher error rate is required, then N can be changed accordingly. When one of the input operands meets the tuning condition, the multiplication processes in approximate mode. In the example shown in Figure 7, the approximation results is -648, while the exact multiplication gets -650. If a higher accuracy is desired, then increasing the value of N would allow the design to only approximate values that are under a certain threshold.

5 CMUL INTEGRATION

5.1 AMD GPU Architecture

We integrated the proposed CMUL in a GPU southern Island architecture Radeon HD 7970 device. The architecture of GPU has been shown in Figure 8. This GPU has 32 compute units, where each contains a scheduler and a set of 4 SIMD execution units. Each SIMD execution unit has 16 cores, which gives a total number of 64 cores per compute unit. Each streaming core consists of both integer and floating point units. We replace all multipliers in floating point multiplier (MUL), multiply-accumulator (MAC), and multiplier-addition (MAD) units with the proposed CMUL. Every time an application launches, all GPU cores are configured as approximation level. CMUL is a modified version of the standard floating point multiplier in GPUs that uses hardware modification to support approximation.

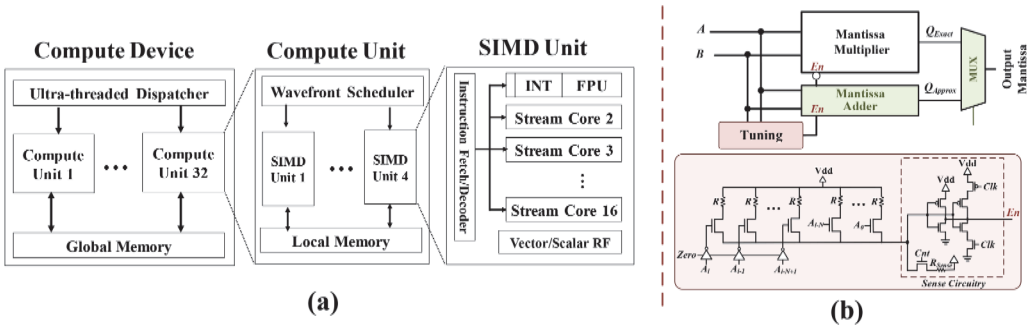

Fig. 8. (a) The architecture of AMD Radeon HD 7970 GPU. (b) Circuitry to support tuning the level of approximation in CMUL.

5.2 CMUL Hardware Support

Figure 8(b) shows the circuitry to support CMUL accuracy tuning. Our design looks at the first N mantissa bits of both input operands to check the tuning condition. If the tuning condition is satisfied in either input mantissas, then our design adds the mantissa of the input operands to generate the mantissa of the multiplication output. Otherwise, similarly to conventional FPUs, the multiplication of the input operand mantissas generates the output multiplication mantissa. Similarly, to tune the level of approximation, our design uses N bits (after the first mantissa bit) of the selected mantissa to decide when to perform mantissa multiplication or approximate it. The number of tuning bits sets the level of approximation, with each additional bit reducing the maximum error by half. The goal is to check the value of the $A_{i-1}, \ldots, A_{i-N}$ to make sure they are the same. As Figure 8(b) shows, the tuning circuitry is a simple transistor-resistor circuitry that samples the match-line (ML) voltage to detect the $A_{i-1}, A_{i-2}, \ldots, A_0$ input operand. In case of any 1-bit in a mantissa, the sense amplifier will detect changes in the ML voltage ($ML = 1$). The circuitry also needs to select the inverted values of the tuning bits for the circuitry to search. To detect the 1 bit on $A^{th}_{i-1}, \ldots, A^{th}_0$ indices on CMUL, the sense amplifier Clk needs to be set to 250ps. Based on the results, we can dynamically change the sampling time to balance the ratio of the running input workload on the approximate CMUL core. For each application, this sampling time can individually set to provide target accuracy.

For DNN application, CMUL hardware support do not need to use tuning circuitry, since the software framework always ensures that the DNN weights satisfy the tuning condition. Therefore, CMUL always works in approximate mode and adds the mantissa of the input operands to generate the mantissa of the multiplication output. The conventional 32-bit floating point multiplier takes 7,690μm^2 area. To enable CMUL functionality, the conventional multiplier needs to use extra 23-bit fixed-point adder and a tuning circuit. Our evaluation using *Synopsys Design Compiler* shows that the adder and the tuning logic consumes 101.5μm^2 and 28.3μm^2 area, respectively. Thus, the CMUL has a 1.68% larger area as compared to the conventional floating point units. This area overhead is negligible considering the flexibility and efficiency that the CMUL can provide.

We propose an automated framework to fine-tune the level of approximation and satisfy required required accuracy while providing the maximum energy savings. Figure 9 shows the proposed framework consisting of the accuracy tuning and accuracy measurement blocks. The framework starts by putting CMUL in the maximum level of approximation. Then, based on the user accuracy requirement, it dynamically decreases the level of approximation until computation accuracy satisfies the user desired quality. For each application, this framework returns the optimal number of CMUL tuning, which provides maximum energy and performance efficiency.

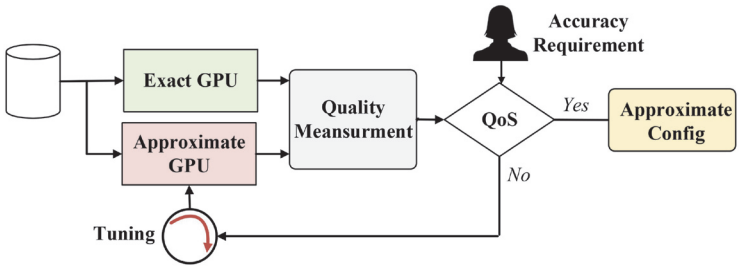

Fig. 9. Framework to support tunable approximation.

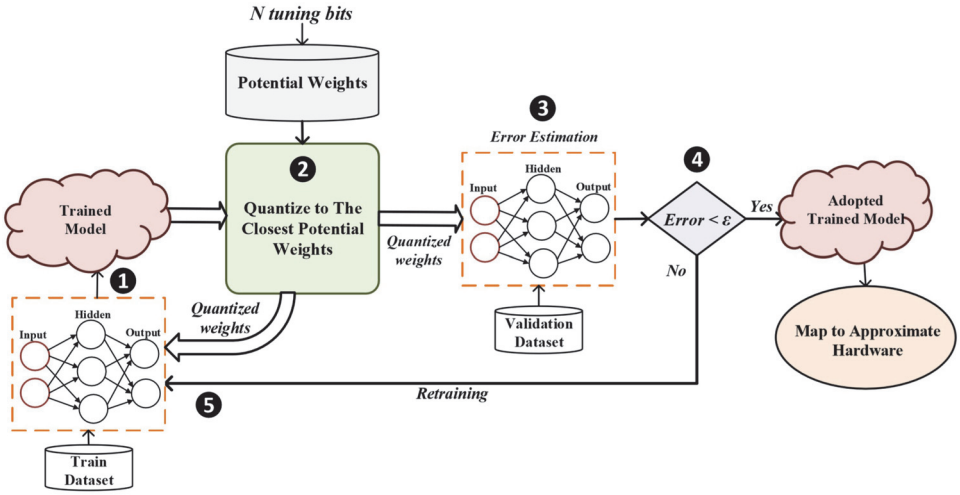

Fig. 10. The overview of the proposed framework in adjusting the DNN weights to a set that is suitable for approximate hardware.

6 DNN ACCELERATION FRAMEWORK

6.1 Overview

In this section, we describe a novel framework to accelerate DNN applications on the approximate GPU architecture. As we explained, the enhanced GPU is configurable, and thus it can be used in a similar way as other applications to accelerate DNN. However, we observe that using this method, there are a few numbers to satisfy the tuning condition. DNNs during inference use a set of fixed weight values. Our framework ensures that DNNs use a weight representation that is suitable for our new approximate hardware. Adjusting the DNN weights ensures that the proposed CMUL has minimum error rate when multiplying input and weights. Figure 10 shows the overview of the proposed framework. In the first step, we get the DNN model by training the network (❶). Our framework adjusts the weights by quantizing them to a closest value that satisfies the tuning condition (❷). The adjusted model inputs the DNN and the accuracy of the new network checks over the validation dataset (❸). This accuracy is compared with the baseline trained model ($\Delta e = e_{Adjusted} - e_{Baseline}$). If the quality loss due to model adjustment is less than ϵ, then we use the adjusted model for the rest of classification (❺); otherwise, we retrain the network using the adjusted weights (❻). This iterative process continues until the error condition is met or the algorithm runs for a pre-specified number of epochs. Note that the retraining approach is general,

and it improves the classification accuracy regardless of approximate hardware. In other words, the retraining framework adapts network to work with the existing constraints. In the following, we explain the details of the proposed framework.

6.2 Weight Modification

The DNN computation involves many of multiplications between the input vector and weight matrix. These multiplications can be accelerated by processing on approximate hardware. However, the error of the approximate hardware, described in Section 4, depends on the input operand values. As we showed in Section 4, the multiplication of input and weight elements has low error rate when one of the input operands have a specific representation. In particular, when one of the mantissas starts with a continues sequence of 0s or 1s, our approximate multiplication results in much lower error rate. The upper bound of the multiplication error rate can be controlled depending on the length of the sequence.

Here, we use the idea of weight modification to adjust the DNN weights such that they become suitable for underlying approximate hardware. The DNN training gives us weights that do not usually have our desired pattern. We modify the trained DNN model to force the weights to follow a particular pattern. Our framework first generates a list of all "*potential weights*" that are suitable for approximate hardware. These numbers are all floating point values that have N consecutive 0s or 1s in the start of their mantissa. Our framework looks at each trained weight in neural network and assigns it to a closet value in *potential weights* list (❷). In case, if the potential weights include large number of values (small N), then there will be very small change in each DNN weight, so DNN model may work with the same accuracy as original DNN model. However, weights with small N run on approximate hardware with larger error. Using a *potential weight* with large N, the approximate hardware will have significantly low error rate. However, the modified weights will be far from the original DNN weights, so it will result in larger change in DNN accuracy. In fact, there is a tradeoff between hardware and software in enabling approximation. Our framework enables software approximation by limiting the values that DNN weights can take. The more limitation on the weights to get patterns with large N, it results in higher software approximation. However, this reduces the level of approximation in hardware, as each multiplication can perform with lower error. To compensate for the software approximation error, a retraining of the network is done and then the optimal N value is selected such that the total hardware+software approximation error is minimized.

6.3 Error Compensation

Limiting the weight is often accompanied by some degree of additive error, $\Delta e = e_{Adjusted}$-$e_{Baseline}$(❸). This error is a difference of the DNN accuracy using baseline and adjusted model. After each model adjustment iteration, our framework compares the Δe with the ϵ value (❹). If the condition is not satisfied, then our framework retrains the neural network to find a new model adopted with the current constraints. After each retraining iteration, all DNN weights again map to a closest value in "*potential weight*" list. This process continues for several iterations until Δe is less than ϵ or the number of iteration passes the maximum epochs (❺). Figure 11 shows an example of speech recognition (Dheeru and Karra Taniskidou 2017) accuracy during retraining iterations when the mantissa of the DNN weights are forced to start with $N = 4$ consecutive 0s or 1s. Our result shows that in the first iterations, the weight limitation has significantly impact on the classification accuracy. However, our framework can completely compensate the possible quality loss by retraining the network for several iterations ($\Delta e = 0\%$).

Table 1 shows the error of different DNN applications, when we limit the *potential weights* to values that satisfy required approximation, specified by user. We tested the impact of our framework

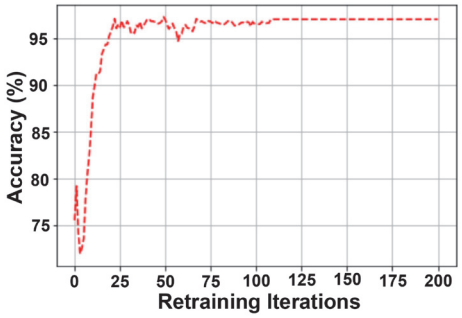

Fig. 11. An example of speech recognition accuracy during different retraining iterations ($N = 5$).

Table 1. Error Loss of Different Applications When the Weight Are Adjusted to a List with a Defined N Tuning Condition

N	1	2	3	4	5	6	7
MNIST	0	0	0	0	0.40%	0.89%	1.93%
ISOLET	0	0	~0%	0.06%	0.53%	1.71%	2.86%
UCIHAR	0	0	0.07%	0.33%	0.42%	1.16%	2.49%
CIFAR-10	0	~0%	0.10%	0.45%	0.97%	2.33%	3.21%

on four different DNN applications including: handwritten digits recognition (MNIST) (LeCun et al. 1998), speech recognition (ISOLET) (Dheeru and Karra Taniskidou 2017), activity recognition (UCIHAR) (Anguita et al. 2013), and object recognition (CIFAR-10) (Krizhevsky and Hinton 2009). Our evaluation shows that for application such as MNIST and ISOLET, our framework can compensate the quality loss when using weights with N equal or less than 4. However, for applications such as CIFAR, 0% quality loss can be achieved using weight with $N = 2$.

7 RESULTS

7.1 Experimental Setup

We integrated the proposed approximate CMUL on the floating point units of an AMD Southern Island GPU, Radeon HD 7970 device. We modified Multi2sim, a cycle accurate CPU-GPU simulator (Ubal et al. 2012) to model the CMUL functionality in three main floating point units in GPU architecture: multiplier, MAC, and MAD. We evaluated power of conventional FPUs using Synopsys Design Compiler and optimized for power using Synopsys Prime Time for 1ns delay in 45nm ASIC flow (Compiler 2000). The circuit level simulation of CMUL has been performed using HSPICE simulator in 45nm TSMC technology. We test the efficiency of enhanced GPU on 11 general Open*CL* applications: *Sobel, Robert, Mean, Laplacian, Sharpen, Prewit, QuasiRandom, FFT, Mersenne, DwHaar1D*, and *Blur*. In these applications, roughly 85% of the floating point operations involve multiplication.

7.2 Benchmarks and DNN Models

Table 2 lists the baseline neural network topologies running four applications and their error rates for train and test modes. For all four datasets, we compare the baseline accuracy of the train and inference phases with those when using the proposed CMUL framework. We compare the designs in terms of runtime and power consumption. Stochastic gradient descent with momentum (Sutskever et al. 2013) 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 (Srivastava et al. 2014) with drop rate of 0.5 is

Table 2. DNN Models and Baseline Error Rates for Four Applications (Input Layer, *IN*; Fully Connected Layer, *FC*; Convolution Layer, *C*; and Pooling Layer, *PL*)

Dataset	Network Topology	Error
MNIST	$IN: 784,\ FC: 512,\ FC: 512,\ FC: 10$	1.5%
ISOLET	$IN: 617,\ FC: 512,\ FC: 512,\ FC: 26$	3.6%
UCIHAR	$IN: 561,\ FC: 512,\ FC: 512,\ FC: 19$	1.7%
CIFAR-10	$IN: 32\times32\times3, CV: 32\times3\times3, PL: 2\times2,$ $CV: 64\times3\times3, CV: 64\times3\times3, FC: 512,\ FC: 10$ 100	14.4%

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.

Handwritten Image Recognition (MNIST): MNIST is a popular machine-learning dataset including images of handwritten digits (LeCun et al. 1998). The objective is to classify an input picture as 1 of the 10 digits {0 ... 9}.

Voice Recognition (ISOLET): Many mobile applications require online processing of vocal data. We evaluate lookNN with the Isolet dataset (Dheeru and Karra Taniskidou 2017), 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 (UCIHAR): For this dataset, 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 (Anguita et al. 2013).

Object Recognition (CIFAR): CIFAR-10 (Krizhevsky and Hinton 2009) are two datasets each of which includes 50,000 training and 10,000 testing images belonging to 10 classes, respectively. The goal is to classify an input image to the correct category, e.g., animals, airplane, automobile, ship, truck, and so on. For the two datasets, we exploit similar topologies based on convolution layers (CV), but they have different numbers of neurons in the last FC layer according to the number of classes.

7.3 Approximate Multipliers

To understand the advantage of proposed design, we compare the energy consumption and delay of the proposed CMUL with the state-of-the-art approximate multipliers proposed in Imani et al. (2017a), Hashemi et al. (2015), Narayanamoorthy et al. (2015), and Kulkarni et al. (2011). The application of previous designs limits to a small range of robust and error-tolerant applications, as they are not able to tune the level of accuracy in runtime. In contrast, the proposed CMUL dynamically finds the inaccurate data and processes them in precise mode. CMUL tunes the level of accuracy at runtime based on the user accuracy requirement. This makes the application of CMUL general. It should be noted that the proposed framework is general and can work properly on other large scale datasets. For example, as prior work in Imani et al. (2018c) showed, CMUL can get minimal quality loss even for larger datasets such as ImageNet.

Table 3 lists the power consumption, critical path delay, and energy-delay product of CMUL alongside previous work in Imani et al. (2017a), Hashemi et al. (2015), Narayanamoorthy et al. (2015), and Kulkarni et al. (2011) in their best configurations. Our evaluation shows that at the same level of accuracy, the proposed design can achieve 2.4× EDP improvement compared to the state-of-the-art approximate multipliers.

7.4 Tunable CMUL

We show the efficiency of the CFPU by running different multimedia and general streaming applications on the enhanced GPU architecture. We consider 10% average relative error as an acceptable

Table 3. Comparing the Energy and Performance of the CMUL and Previous Designs

	Power (mW)	Delay (ns)	EDP (pJs)	Tuned Error	Tunable	No Tuning Error
CMUL 3	0.15	1.1	0.18	6.3%	***Yes***	11.1%
CFPU3 (Imani et al. 2017a)	0.17	1.6	0.44	6.3%	***Yes***	50%
DRUM6 (Hashemi et al. 2015)	0.29	1.9	1.04	6.3%	*No*	NA
ESSM8 (Narayanamoorthy et al. 2015)	0.28	2.1	1.2	11.1%	*No*	NA
Kulkarni (Kulkarni et al. 2011)	0.82	3.5	10.0	22.2%	*No*	NA

Table 4. Normalized EDP and Quality Loss (QL) of the GPU Enhanced with CFPU in Different Tuning Mode

Tuning bits	Sobel		Robert		Mean		Laplacian		FFT		Mersenne		DwtHaar1D		Blur	
	EDP	*QL*	*EDP*	*QL*	*EDP*	*QL*	*EDP*	*QL*	*EDP*	*QL*	*EDP*	*QL*	*EDP*	*QL*	*EDP*	*QL*
1 bit	0.11	2.43%	0.13	0.45%	0.15	0.27%	0.17	0.37%	0.11	9.18%	0.15	4.29%	0.14	11.09%	0.17	6.24%
2 bit	0.14	1.13%	0.15	0.17%	0.16	0.14%	0.18	0.21%	0.28	5.19%	0.23	2.37%	0.17	8.2%	0.26	2.93%
3 bits	0.16	0.21%	0.16	0.06%	0.19	0.03%	0.19	0.02%	0.37	3.1%	0.31	1.9%	0.25	4.1%	0.37	0.03%
4 bits	0.17	0.01%	0.17	~0%	0.23	~0%	0.20	0.01%	0.41	1.07%	0.36	0.62%	0.29	1.98%	0.42	0.09%
5 bits	0.18	~0%	0.17	~0%	0.25	~0%	0.21	~0%	0.46	0.43%	0.44	0.11%	0.36	0.30%	0.51	0.02%

Table 5. Normalized EDP and QL of the GPU Enhanced with CMUL in Different Tuning Mode

Tuning bits	Sobel		Robert		Mean		Laplacian		FFT		Mersenne		DwtHaar1D		Blur	
	EDP	*QL*	*EDP*	*QL*	*EDP*	*QL*	*EDP*	*QL*	*EDP*	*QL*	*EDP*	*QL*	*EDP*	*QL*	*EDP*	*QL*
1 bit	0.08	2.09%	0.11	0.35%	0.14	0.09%	0.09	0.37%	0.10	7.26%	0.12	3.02%	0.10	8.42%	0.11	4.36%
2 bit	0.12	0.94%	0.13	0.07%	0.13	0.06%	0.14	0.09%	0.22	3.56%	0.18	1.33%	0.13	5.77%	0.20	1.05%
3 bits	0.13	0.35%	0.14	0.01%	0.16	0.02%	0.15	0.01%	0.30	1.17%	0.24	0.63%	0.21	1.8%	0.33	0.01%
4 bits	0.13	0.02%	0.14	~0%	0.17	~0%	0.16	0%	0.36	0.41%	0.32	0.12%	0.25	0.24%	0.27	~0%
5 bits	0.15	~0%	0.15	~0%	0.21	~0%	0.19	~0%	0.42	0.14%	0.39	0.03%	0.31	0.12%	0.42	~0%

accuracy metric for all applications, verified by Esmaeilzadeh et al. (2012). We tune the level of approximation by checking the N bits of mantissa in the input operands. If all N tuning bits in one of the input mantissa is 0 or 1, then the multiplication runs in approximate mode; otherwise, it runs precisely by multiplying the mantissa of input operands. For each application, Table 4 and Table 5 show the normalized EDP and quality loss of different applications running on approximate GPU enhanced by CPU and CMUL, respectively. For both designs, we change the number of tuning bit from 1 (none) to 5 bits. The results are normalized to the EDP of the GPU using conventional FPUs. Increasing the number of tuning bits improves the computation accuracy by processing the far and inaccurate multiplications in precise mode. However, more number of tuning bits slows down the computation, because a larger portion of data is processed on precise. Our experimental evaluation shows that running applications on proposed CFPU provides 3.1× EDP improvement as compared to a GPU using conventional FPUs, while ensuring less than 1% quality of loss. Our results in Table 5 shows that CMUL can achieve 2.7× higher EDP improvement as compared to CFPU design while providing the same quality of computation.

7.5 CMUL and DNN Acceleration

To provide large efficiency, we design a framework that adapts DNN to run on approximate hardware. Using our framework, the DNN quality loss may be happened by both software and

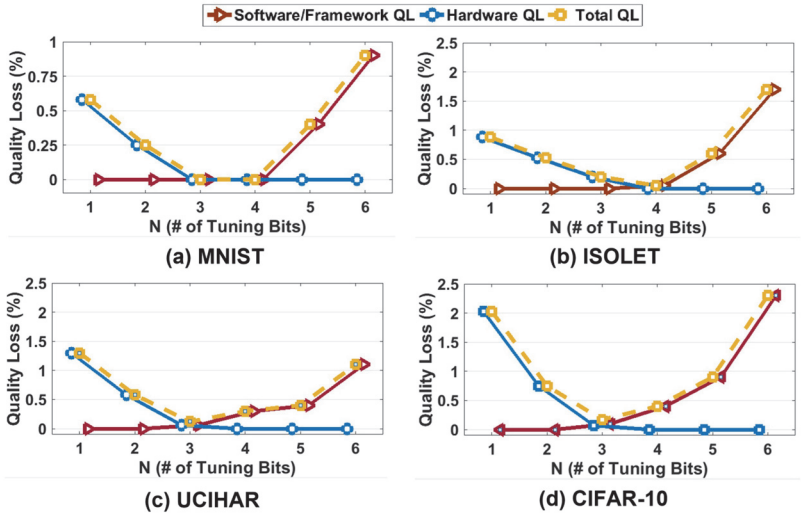

Fig. 12. Quality loss of different DNN applications due to software/framework and hardware approximation using different tuning bits (N).

hardware. Figure 12 shows the quality loss different DNN applications running on proposed approximate hardware. The x-axis in figure shows the N, the sequence of the 0s and 1s at the mantissa of the weight. For example, $N = 4$ ensures that all DNN weights have four consecutive 0s or 1s in the first N bits their mantissas. The lines in the figure show the breakdown of quality loss coming from software framework and proposed approximate hardware. The results show that increasing the N parameter from 1 to 6, the software framework approximation starts increasing due to weight constraint applied by the framework. Using large N, the DNN do not have good flexibility to assign proper weights to DNN, thus it results in large quality loss. However, the quality loss due to hardware approximation has reverse relation to the N value. Using large N, the approximate multiplier can achieve lower error. Figure 12 also shows the total DNN quality loss due to both software and hardware approximation. Our result shows that applications provide optimum quality loss using different N values. For example, MNIST can achieve to minimum 0% quality loss using $N = 3$ and 4, while ISOLET can achieve to 0.05% quality loss using $N = 4$.

One major advantage of our proposed framework is that CMUL does not need to check the N for each DNN application. Regardless of the N value, CMUL always takes the same time/energy to run a DNN application. Figure 13 and Figure 14 report the energy consumption and EDP of the DNN applications running on the enhanced-GPU with and without supported framework. All results are relative to the energy and EDP of the conventional GPU using exact FPUs. Our evaluation shows that using the framework our design can achieve to the same energy efficiency and EDP improvement, regardless of the value of N. In fact, our framework can approach full hardware approximation (CMUL always runs in approximate mode). However, the approximate GPU with no supported framework runs partially in approximate mode. The results show that this hit rate and energy efficiency is usually too small when the N becomes large. In addition, for every input, CMUL with no framework needs to pay the overhead of checking the tuning condition. Our result shows that CMUL with supported framework can achieve 60.3% and energy efficiency and 3.2× EDP improvement as compare to the baseline GPU, while they ensure less than 0.2% quality loss for tested applications. At the same level of accuracy, these results are 38.7% and 2.0× higher than energy efficiency and EDP improvement of the CMUL with no supported framework.

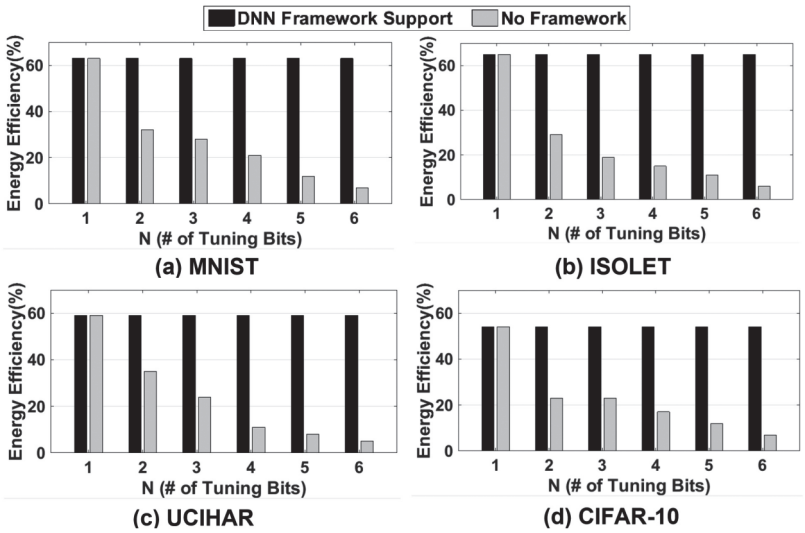

Fig. 13. Energy efficiency improvement of the enhanced GPU with and without DNN framework support.

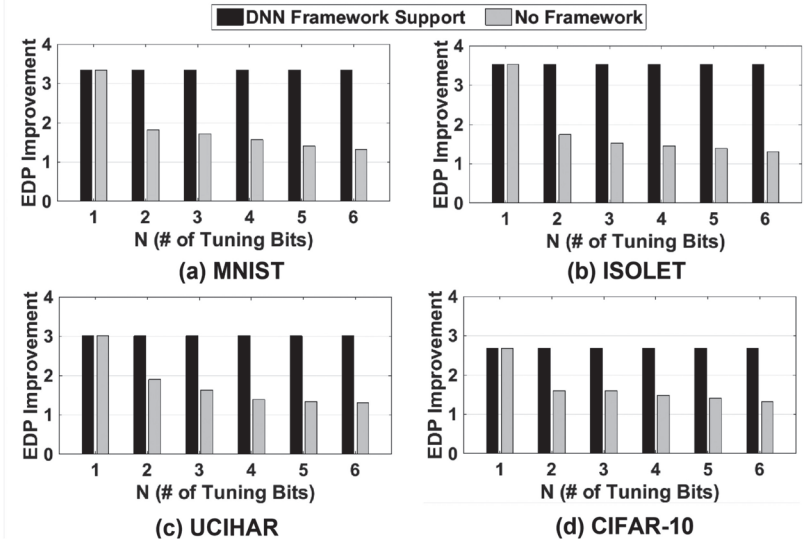

Fig. 14. Energy-delay product of enhanced GPU with and without DNN framework support.

Table 6 compares the EDP improvement of the CMUL using the proposed DNN framework with CMUL and CFPU (Imani et al. 2017a) with no framework support. Our evaluation shows that CMUL using framework has less than 0.5% quality loss over all applications. To provide the same quality of classification, CMUL and CFPU with no framework support need to run the computation in a configuration very close to the precise mode, and thus they do not provide much advantage as compared to conventional GPU. In addition, CMUL using framework ensures that all multiplications can run in approximate mode, and this results in significantly performance improvement. In contrast, in CMUL and CFPU with no framework support, the slowest thread with the least number of multiplications in approximate mode, bounds the GPU performance. To further

Table 6. Comparing the EDP Improvement of CMUL with and without DNN Framework with CFPU Design

	QL=0.5%			QL=2%		QL=4%	
	CMUL	*CMUL No Framework*	*CFPU*	*CMUL No Framework*	*CFPU*	*CMUL No Framework*	*CFPU*
MNIST	3.34×	1.06×	1.02×	1.25×	1.12×	1.56×	1.40×
ISOLET	3.53×	1.15×	1.12×	1.18×	1.06×	1.45×	1.30×
UCIHAR	3.01×	1.07×	1.03×	1.29×	1.17×	1.62×	1.46×
CIFAR-10	2.68×	1.08×	1.04×	1.28×	1.15×	1.60×	1.44×

improve the EDP of the CMUL and CFPU, one can put the multiplications in deeper approximate mode. However, this results in significantly quality loss. The results in Table 6 shows that even with 2% (4%) quality loss, CMUL and CFPU without framework support provide 2.5× and 2.8× (1.9× and 2.2×) lower EDP improvement as compared to CMUL, which ensures less than 0.5% quality loss.

8 CONCLUSION

In this article, we propose a configurable floating point multiplier that can approximately perform the computation with significantly lower energy and performance cost. The proposed approximate multiplication has tuning capability by adaptively process each new piece of data precisely. We also proposed a framework to accelerate DNN applications with our approximate FPU. Our framework modifies the training of the DNN to make it suitable for underlying approximate hardware. Our evaluations on four DNN applications show that CMUL can achieve 60.3% and energy efficiency and 3.2× EDP improvement as compared to the baseline GPU, while they ensure less than 0.2% quality loss as compared to precise hardware. These results are 38.7% and 2.0× higher than energy efficiency and EDP improvement of the CMUL without using the proposed framework. Another main advantage of the proposed framework is its generality, as it can be applied on any approximate multiplier.

REFERENCES

Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge Luis Reyes-Ortiz. 2013. A public domain dataset for human activity recognition using smartphones. In *Proceedings of the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN'13)*.

Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The internet of things: A survey. *Comput. Netw.* 54, 15 (2010), 2787–2805.

Vincent Camus, Jeremy Schlachter, Christian Enz, Michael Gautschi, and Frank K Gurkaynak. 2016. Approximate 32-bit floating-point unit design with 53% power-area product reduction. In *Proceedings of the 42nd Annual European Solid-State Circuits Conference (ESSCIRC'16)*. IEEE, 465–468.

Dan C. Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. 2011. Flexible, high performance convolutional neural networks for image classification. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Vol. 22. 1237.

Design Compiler. 2000. Synopsys Inc. https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html.

Matthieu Courbariaux, Jean-Pierre David, and Yoshua Bengio. 2014. Low precision storage for deep learning. *arXiv preprint Arxiv:1412.7024* (2014).

Dua Dheeru and Efi Karra Taniskidou. 2017. UCI Machine Learning Repository. Retrieved from http://archive.ics.uci.edu/ml

Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 449–460.

John Gantz and David Reinsel. 2011. Extracting value from chaos. *IDC Iview* 4, 12 (2011), 1–12.

Self Prasad Gnawali, Seyed Nima Mozaffari, and Spyros Tragoudas. 2018. Low power spintronic ternary content addressable memory. *IEEE Transactions on Nanotechnology* 16, 6 (2018), 1206–1216.

Jie Han and Michael Orshansky. 2013. Approximate computing: An emerging paradigm for energy-efficient design. In *Proceedings of the 2013 18th IEEE European Test Symposium (ETS'13)*. IEEE, 1–6.

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. In *Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. IEEE, 243–254.

Song Han, Huizi Mao, and William J. Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

Soheil Hashemi, R. Bahar, and Sherief Reda. 2015. DRUM: A dynamic range unbiased multiplier for approximate applications. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 418–425.

Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, et al. 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Sign. Process. Mag.* 29, 6 (2012), 82–97.

Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, and Kurt Keutzer. 2016. Firecaffe: Near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2592–2600.

Farhad Imani, Changqing Cheng, Ruimin Chen, and Hui Yang. 2018a. Nested gaussian process modeling for high-dimensional data imputation in healthcare systems. In *Proceedings of the Institute of Industrial and Systems Engineers 2018 Conference & Expo (IISE'18)*. 19–22.

Mahdi Imani, Seyede Fatemeh Ghoreishi, and Ulisses M Braga-Neto. 2018b. Bayesian control of large MDPs with unknown dynamics in data-poor environments. In *Advances in Neural Information Processing Systems*.

Mohsen Imani, Yeseong Kim, Abbas Rahimi, and Tajana Rosing. 2016a. Acam: Approximate computing based on adaptive associative memory with online learning. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. ACM, 162–167.

Mohsen Imani, Pietro Mercati, and Tajana Rosing. 2016b. ReMAM: Low energy resistive multi-stage associative memory for energy efficient computing. In *Proceedings of the 2016 17th International Symposium on Quality Electronic Design (ISQED'16)*. IEEE, 101–106.

Mohsen Imani, Shruti Patil, and Tajana S Rosing. 2016c. MASC: Ultra-low energy multiple-access single-charge TCAM for approximate computing. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. EDA Consortium, 373–378.

Mohsen Imani, Daniel Peroni, Yeseong Kim, Abbas Rahimi, and Tajana Rosing. 2017b. Efficient neural network acceleration on gpgpu using content addressable memory. In *Proceedings of the 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE'17)*. IEEE, 1026–1031.

Mohsen Imani, Daniel Peroni, and Tajana Rosing. 2017a. CFPU: Configurable floating point multiplier for energy-efficient computing. In *Proceedings of the 54th ACM/EDAC/IEEE Design Automation Conference (DAC'17)*. IEEE, 1–6.

Mohsen Imani, Abbas Rahimi, Pietro Mercati, and Tajana Rosing. 2017c. Multi-stage tunable approximate search in resistive associative memory. *IEEE Trans. Multi-Scale Comput. Syst.* 4, 1 (2017), 17–29.

Mohsen Imani, Abbas Rahimi, and Tajana S. Rosing. 2016d. Resistive configurable associative memory for approximate computing. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'16)*. IEEE, 1327–1332.

Mohsen Imani, Mohammad Samragh, Yeseong Kim, Saransh Gupta, Farinaz Koushanfar, and Tajana Rosing. 2018c. RAPIDNN: In-memory deep neural network acceleration framework. *arXiv preprint arXiv:1806.05794* (2018).

Changqing Ji, Yu Li, Wenming Qiu, Uchechukwu Awada, and Keqiu Li. 2012. Big data processing in cloud computing environments. In *Proceedings of the 12th International Symposium on Pervasive Systems, Algorithms and Networks (ISPAN'12)*. IEEE, 17–23.

Navid Khoshavi, Xunchao Chen, Jun Wang, and Ronald F. DeMara. 2016. Read-tuned STT-RAM and eDRAM cache hierarchies for throughput and energy enhancement. *arXiv preprint arXiv:1607.08086* (2016).

Yeseong Kim et al. 2015. CAUSE: Critical application usage-aware memory system using non-volatile memory for mobile devices. In *Proceedings of the 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'15)*. IEEE, 690–696.

Philipp Klaus Krause and Ilia Polian. 2011. Adaptive voltage over-scaling for resilient applications. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'11)*. IEEE, 1–6.

Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images, (Volume. 1, issue. 4, page. 1–7). Technical report, University of Toronto.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. 1097–1105.

Parag Kulkarni, Puneet Gupta, and Milos Ercegovac. 2011. Trading accuracy for power with an underdesigned multiplier architecture. In *Proceedings of the 2011 24th International Conference on VLSI Design (VLSI Design'11)*. IEEE, 346–351.

Yann LeCun, Corinna Cortes, and Christopher J. C. Burges. 1998. The MNIST database of handwritten digits 10 (1998), 34. http://yann.lecun.com/exdb/mnist.

Yann LeCun, Koray Kavukcuoglu, and Clément Farabet. 2010. Convolutional networks and applications in vision. In *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS'10)*. IEEE, 253–256.

Jian Liang, Russell Tessier, and Oskar Mencer. 2003. Floating point unit generation and evaluation for FPGAs. In *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03)*. IEEE, 185–194.

Chia-Hao Lin and Chao Lin. 2013. High accuracy approximate multiplier with error correction. In *Proceedings of the 2013 IEEE 31st International Conference on Computer Design (ICCD'13)*. IEEE, 33–38.

Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. 2016. Fixed point quantization of deep convolutional networks. In *Proceedings of the International Conference on Machine Learning*. 2849–2858.

Darryl D. Lin and Sachin S. Talathi. 2016. Overcoming challenges in fixed point training of deep convolutional networks. *arXiv preprint arXiv:1607.02241* (2016).

Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. 2015. Neural networks with few multiplications. *arXiv preprint arXiv:1510.03009* (2015).

Cong Liu, Jie Han, and Fabrizio Lombardi. 2014. A low-power, high-performance approximate multiplier with configurable partial error recovery. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'14)*. IEEE, 1–4.

Srinivasan Narayanamoorthy, Hadi Asghari Moghaddam, Zhenhong Liu, Taejoon Park, and Nam Sung Kim. 2015. Energy-efficient approximate multiplication for digital signal processing and classification applications. *IEEE Trans. VLSI Syst.* 23, 6 (2015), 1180–1184.

Mahdi Nazemi, Amir Erfan Eshratifar, and Massoud Pedram. 2018. A hardware-friendly algorithm for scalable training and deployment of dimensionality reduction models on FPGA. *arXiv preprint arXiv:1801.04014* (2018).

Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. 2014. Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'14)*. IEEE, 1717–1724.

Daniel Peroni et al. 2019. ALook: Adaptive lookup for GPGPU acceleration. In *Proceedings of the IEEE Asia and South Pacific Design Automation Conference (ASP-DAC'19)*. IEEE, 1–7.

Mohammad Samragh Razlighi, Mohsen Imani, Farinaz Koushanfar, and Tajana Rosing. 2017. Looknn: Neural network with no multiplication. In *Proceedings of the 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE'17)*. IEEE, 1775–1780.

Sahand Salamat et al. 2018. RNSnet: In-memory neural network acceleration using residue number system. In *Proceedings of the 2018 IEEE International Conference on Rebooting Computing (ICRC'18)*. IEEE, 1–10.

Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* 15, 1 (2014), 1929–1958.

Alexander Suhre, Furkan Keskin, Tulin Ersahin, Rengul Cetin-Atalay, Rashid Ansari, and A. Enis Cetin. 2013. A multiplication-free framework for signal processing and applications in biomedical image analysis. In *Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'13)*. IEEE, 1123–1127.

Ilya Sutskever, James Martens, George E. Dahl, and Geoffrey E. Hinton. 2013. On the importance of initialization and momentum in deep learning. In *International Conference on Machine Learning*. 1139–1147.

Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: A simulation framework for CPU-GPU computing. In *Proceedings of the 2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. IEEE, 335–344.

Received July 2018; revised November 2018; accepted December 2018