Invocation-driven Neural Approximate Computing with a Multiclass-Classifier and Multiple Approximators

Haiyue Song*, Chengwen Xu*, Qiang Xu**, Zhuoran Song*, Naifeng Jing*, Xiaoyao Liang*, and Li Jiang*

*MoE Key Lab of Artificial Intelligence, AI Institute
*School of Electronic, Information and Electrical Engineering, Shanghai Jiao Tong University, China
**Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong

***Abstract*—Neural approximate computing gains enormous energy-efficiency at the cost of tolerable quality-loss. A neural approximator can map the input data to output while a classifier determines whether the input data are safe to approximate with quality guarantee. However, existing works cannot maximize the invocation of the approximator, resulting in limited speedup and energy saving. By exploring the mapping space of those target functions, in this paper, we observe a nonuniform distribution of the approximation error incurred by the same approximator. We thus propose a novel approximate computing architecture with a Multiclass-Classifier and Multiple Approximators (MCMA). These approximators have identica network topologies, and thus can share the same hardware resource in an neural processing unit(NPU) clip. In the runtime, MCMA can swap in the invoked approximator by merely shipping the synapse weights from the on-chip memory to the buffers near MAC within a cycle. We also propose efficient co-training methods for such MCMA architecture. Experimental results show a more substantial invocation of MCMA as well as the gain of energy-efficiency.**

I. Introduction

Approximate computing is a promising technique to gain energy efficiency with tolerable losses of computation quality for specific error-tolerant applications, such as recognition, mining, and search (RMS) applications. Existing works strive to reduce the computation effort by precision scaling [1], loop perforation [2], memorization [3]; others try to minimize the performance hurdle by skipping memory references [4]. Approximate computing can also gain more parallelism using strategies like lowering the precision of intermediate computations [5], memorization [6], etc. The reported power saving is usually within $5\% - 40\%$ [7], depending on the application, the approximation technique, and the acceptable error.

Neural networks (NNs) are suitable for general approximate computing. First, NNs are inherently utilized in many error-tolerant applications, such as real-world pattern recognition [8]. Second, NNs are theoretically universal approximators to fit any continues function [9]. Previous works mimic the functions of approximable code regions [10] using NNs. Third, NNs expose enormous parallelism for performance/energy benefits, in favor of the booming development of hardware accelerators [11]. The same accelerator can approximate various functions by merely changing the NN topologies and weights. Above advantages make NNs a principal candidate in approximate computing, namely *approximators*.

Quality control is essential in approximate computing to identify those safe-to-approximate work loads [12]. On one hand, various techniques employ high-level, application specific light-weight checks(LWCs) to estimate the quality of the approximation output [13]. These works then advocate a roll-back or online tuning mechanism to guarantee the computation reliability. On the other hand, a proactive strategy is to predict whether the load is safe-to-approximate or not using statistical and learning models [12], [14]–[17].

Approximate computing framework that deploys two neural networks can further optimize the approximation quality and invocation [18], [19]. In these works, one neural network is trained as a *classifier*, to differentiate safe-to-approximate input from others, while the other as an *approximator*. It is challenging to train two neural networks because of the correlation between them, but we advocate this approximate computing framework in this paper due to the superiority of neural networks.

Existing neural approximate computing with quality control strives to optimize two objectives simultaneously: minimizing the approximation error and maximizing the invocation of approximator (the probability of invoking the approximator to gain energy efficiency). In this work, we argue that these two goals may contradict each other. Based on the principle of

This research was partially supported by National Natural Science Foundation of China (Grant No. 61602300, 61432017), Shanghai Science and Technology Committee (Grant No. 18ZR1421400), Shanghai Jiao Tong University Biomedical Engineering Research Foundation (No. YG2015MS17), and Shanghai clinical ability construction of The three grade hospital (No. SHDC12015904). The Corresponding author is Li Jiang.

ICCAD '18, November 58, 2018, San Diego, CA, USA

ACM ISBN 978-1-4503-5950-4/18/11...$15.00
https://doi.org/10.1145/3240765.3240819

approximate computing, it is not necessary to further reduce the approximation error when the error is lower than the error bound. Instead, we should maximize the invocation of approximator. Because more input invokes the accelerator—ASIC [10], FPGA [20] or NVM [21], [22] etc.—rather than CPU, more energy efficiency we gain. Therefore, we advocate the invocation-driven design for approximate computing. In this paper, we propose a novel approximate computing architecture *to maximize the invocation of approximate accelerators. The key idea is to orchestrate multiple approximators which fit more input data.* The contribution of this work is summarized as below:

- We investigate two compound structures of approximate computing: i) Multiple cascaded classifiers and approximators (MCCA) architecture that naturally extends from the existing methods; ii) Multiclass-Classifier and Multiple Approximators architectures (MCMA) that processes much faster than MCCA.
- We investigate two co-training methods, i.e. complementary and competitive training, for multiple approximators in the compound structure. Each approximator can fit a distribution of the input data—a partition in input space—and the whole compound structure can theoretically fit all samples in the input space thus maximize the invocation of approximate accelerators.
- We implement the compound approximate computing structures in the existing neural processing unit(NPU) design. Based on the investigation, the proposed approximate computing architecture can achieve superior speedup without hardware or performance overhead.

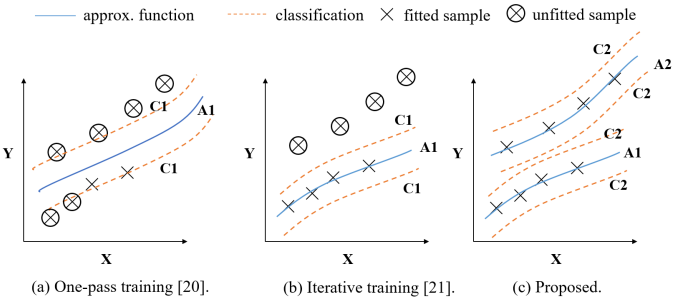
The rest of the paper is organized as follows: section II shows related works and motivation. Proposed approximate computing architectures are described in section III. Section IV shows the experiments, and section V concludes this paper.

II. Related works and Motivation

A. Model based quality control for Approximate Computing

Various approximate computing methods use statistical or learning models to predict the approximation errors. Rumba et al. [12] propose decision trees and linear models for predicting the error in the runtime. Wang et al. [15] propose a reinforcement learning method to determine when and how to rollback for occasional large errors with a minimum cost. However, due to the bias of input feature space, some approximate outputs cannot satisfy the quality requirement. These works give up screening the input data on the accelerator call site, resulting in unnecessary rollbacks.

Proactive quality control mechanisms use predictors to determine the input shifted to the accelerator and invoke the accelerator. In [16], a Bayesian network learns the cost and error models of an optimization problem offline and determines the quality control knob by solving this optimization problem. Rahimi et al. [17] selectively reduce and dynamically tune the precision, subjecting to a statistical quality knob; consequently, an approximator within the fixed area budget can accommodate more parallel approximate kernels. These works strive to maximize the invocation of a single approximation scheme. *The upper limit of a given approximation scheme, e.g., an 8-bit precision approximate-multiplier, undoubtedly circumscribes the potential of safe-to-approximate input in the input space, preventing us from exploring more energy efficiency.*



(a) One-pass training [20]. (b) Iterative training [21]. (c) Proposed.

Fig. 1. A conceptual view of fitting the data samples with different neural approximate computing architectures.

B. Approximate Computing with two neural networks

Mahajan et al. [18] present the first approximate computing framework that employs two neural networks as the classifier and the approximator, respectively. They make the best effort to train an approximator using the input and output data of the target function for approximation. Then, they test the approximator using the same input data and generate the approximation output. Comparing the approximation error with the error bound can tell us whether the input data is safe-to-approximate or not, which serves as labels to train the classifier. Their results show the NN-based predictor is better than the table-based predictor regarding prediction accuracy. However, this work ignores the correlation between the two neural networks. The capability of the classifier is constrained by the approximator because the approximator produces the "label" in the training sample of the classifier.

Consequently, Xu et al. [19] propose an iterative training method for the classifier and approximator. The fundamental idea is to repeat the training process in [18], retrain the approximator using only the safe-to-approximate input identified by the classifier, and then retrain the classifier again using the new "label" produced by the approximator. As the iterative training goes on, the approximation error of the approximator and the prediction error of the classifier keep decreasing. However, this method is weak in enhancing the invocation of the approximator because the amount of safe-to-approximate data shrinks after several iterations.

C. Motivation

Without the loss of generality, we use a simple but motivational example to point out the remaining problem of the existing neural approximate computing framework involving two neural networks. In [18], the approximator (*A1*) and the classifier (*C1*) are trained separately in one pass, as shown in Fig. 1(a). The optimization targets of *A1* and *C1* may be mismatched. The *A1* strives to fit "all" possible input samples, which makes it challenging for *C1* to differentiate

the safe-to-approximate data from others. By the iterative retraining of *A1* using the safe-to-approximate input samples recognized by *C1*, and vice versa, as shown in Fig. 1(b), *A1* and *C1* can coordinate their optimization objectives [19]. *A1* evolves to provide more accurate approximation output, while *C1* becomes more robust to discriminate those safe-to-approximate data from others. However, the approximator after sufficient training may overfit one cluster/distribution of input sample. Such bias also pushes away other input samples from the approximation function. The input sample pushed out of the "error bound" in the view of the classifier results in a degraded invocation of the approximator.

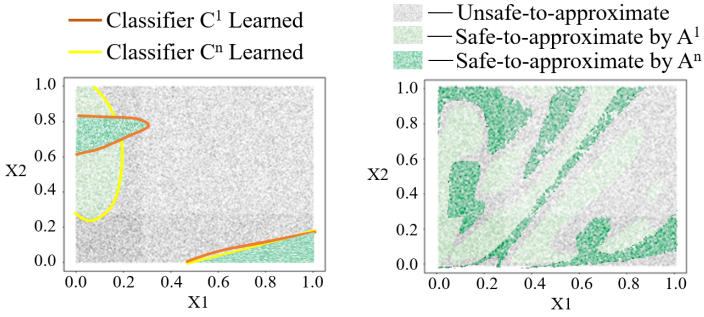
Motivated by this, the primary target of this work is to maximize the invocation of the approximator (salvage the abandoned safe-to-approximate data), by initiating multiple approximators. As shown in Fig. 1(c), each approximator (e.g., *A1* and *A2*) concentrates on fitting a reasonable amount of data samples that are easily recognized by the classifiers (e.g., *C1*, *C2*). The compound structure can cover much more data in the input space that maximizes the overall invocation. The rest of this paper focuses on designing a compound structure of neural networks and their training methods.

III. Proposed Approximate Computing Architecture

In this section, we analyze and interpret different data selection strategies based on iterative training method [19]. The observation inspires us to develop two compound structures of neural networks for our approximate computing architecture. At last, we deploy our architectures in hardware design based on a typical Neural Processing Unit design [10].

A. Clustering of safe-to-approximate input sample

It is challenging to optimize the invocation of the approximator because there is no place to put the invocation into the loss functions of both neural-networks. Two neural networks have two viewpoints upon the input samples. We can select training data predicted as safe-to-approximate by the classifier in the current iteration, denoted as category *C*, or choose samples whose approximation error–resulting from the approximator–is within the error bound, denoted as category *A*. Such difference is due to the mismatch between the viewpoints of the two neural networks. The previous work [19] simply chooses the safe-to-approximate samples, on which the two neural networks agree (denoted as "AC"). Alternatively, we discover the clustering effect of the safe-to-approximate samples that can help us increase the invocation of the approximator.



(a) Training with *category C*. (b) Training with *category A*.

Fig. 2. Distribution of training data for approximator in iterative training. C^1 and C^n mean classifier in the first and the n_{th} iteration. The same for A^1 and A^n.

The approach to such discovery is described as follows. Given a typical RMS application, we apply iterative training multiple times, and plot the input samples: safe-to-approximate samples are in green color. We further track the change of the plots in the final iteration and color the safe-to-approximate input samples in light green. As depicted in Fig. 2 (a) and (b), the safe-to-approximate input samples are clustered when selecting the training data using *C* in the iterative training process; the sphere of these clusters gradually expands in the later iterations of training. When choosing *A* in each iteration of training, on the contrary, the safe-to-approximate input samples are scattered into the input space as small pieces. It is much easier for the classifier to discriminate the clustered safe-to-approximate samples. It is also easier for approximator to fit the clustered safe-to-approximate samples.

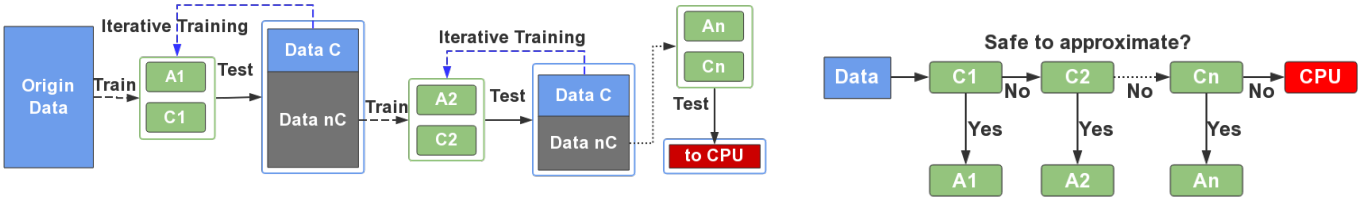
For those samples out of the clusters, we can remedy those abandoned samples using additional approximators. Therefore, we orchestrate multiple approximators in the approximate computing architecture and develop the training methods to incline each approximator to a different cluster of safe-to-approximate samples, as described in the next section.

B. Multiple Cascaded Classifiers and Approximators (MCCA)

The key idea to increase the invocation of approximator is to consecutively employ additional approximator for the remedy of the input samples abandoned in previous iterations. Meanwhile, a question emerges: how many approximators are enough to cover the majority of the input space?

To answer this question, we propose the Multiple Cascaded Classifiers and Approximators (MCCA) structure by cascading multiple pairs of the approximator-classifier structure proposed in the iterative training method. Fig. 3(a) illustrates an example MCCA structure and the training process. MCCA structure is composed of cascaded pairs of classifier and approximator, each of which sequentially divides input data space into safe-to-approximate and unsafe-to-approximate clusters. In the beginning, the original input samples are used to train the first pair of the classifier-approximator structure, denoted as A_1 and C_1. The training process is similar to the iterative training [19], except that we select the training samples using category *C* to train the C_1-A_1 pair *in the second iteration.* After the above training process converges, we feed the remaining input samples not yet to be recognized by C_1 (Data nC) to the second pair C_2-A_2. Such process continues until a specific pair of C_n and A_n cannot converge. The remaining unsafe-to-approximation data should enter CPU for precise computation.

At run-time, the approximate computing architecture with MCCA structure also consumes the input loads in a cascaded



(a) Training. (b) Execution.

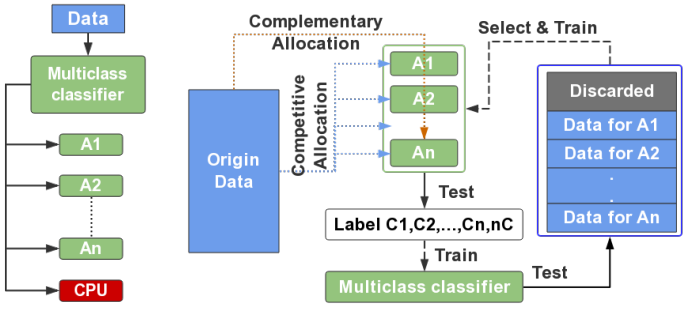
Fig. 3. Proposed MCCA architecture and its training process.

manner. The execution process is shown in Fig. 3(b). A_1 approximates the input data based on the prediction of C_1. If C_1 disapproves, the input data are sent to the next pair C_2-A_2. The input data are finally executed in CPU if rejected by all the classifiers. Obviously, MCCA is too time consuming that may offset the speedup resulting from the approximate accelerator. We solve this problem by proposing a more efficient structure in the next section.

C. Multiclass-classifier and Multiple Approximators (MCMA)

We propose a "parallel" compound structure with a multiclass-classifier and multiple approximators (denoted as MCMA). Given the input samples at the runtime, as shown in Fig. 4(a), the multiclass-classifier predicts which approximator can generate a safe-to-approximate result. The approximator with the highest confidence in the prediction consumes the input sample. A critical question is how to train such a paralled structure.

We propose two data allocation mechanisms to train the MCMA structure. For *Complementary* mechanism, input samples are fed into all the approximators from A_1 to A_n serially. The latter approximator is trained using the input samples that previous approximators fail to approximate. After the initialization, the training process of MCMA structure adopts the iterative training method. In each iteration afterward, we generate the label for training the multiclass-classifier using the *complementary* mechanism: A_1 tests all the input samples and produces the label C_1 for any input sample that A_1 can safely approximate. Otherwise, no label is assigned to that input sample. Subsequently, A_2 tests the remaining input samples without any label and produces label C_2 for the sample that is safe-to-approximate for A_2. This procedure continues until all the approximators have finished the testing. The remaining input samples without any label are labeled as nC. We then train the multiclass-classifier using all the input samples with labels. In the next iteration, we test the derived classifier using all the input samples. The resulting prediction, e.g., Ci, with the highest confidence indicates the most suitable approximator, e.g., Ai, to approximate the input sample. The multiclass-classifier distributes each input samples to its most suitable approximate and eventually partitions the input space into $n+1$ territories. Each of the n approximators takes the samples in its territory for the next iteration of training. The basic idea of the complementary allocation mechanism is similar to **AdaBoost** algorithm [23]. However, the execution process of the MCMA structure is more efficient than that of an AdaBoost algorithm: The MCMA structure derives the final approximation from one approximator, while an AdaBoost algorithm derives the result by voting from all the approximators.



(a) Execution. (b) Training process.

Fig. 4. Proposed MCMA architecture and its training process.

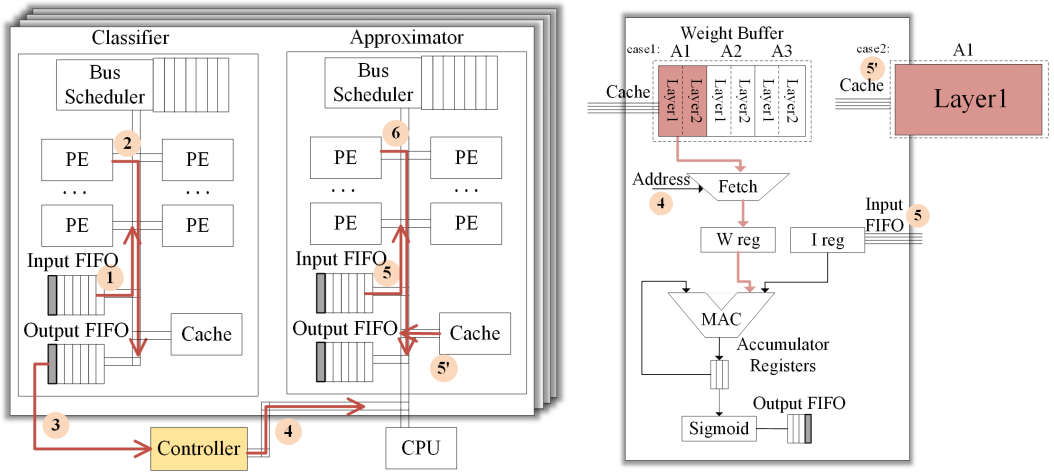
For the *Competitive* mechanism, in the first iteration, we assign all the input samples to all the approximators in parallel. All the approximators compete with each other to fit as many input samples as possible. Each approximator may bias different distribution of the input samples due to the randomness in the training samples.

Furthermore, we can vary the hyper-parameters in the neural networks such that each approximator can reach different local minima. After the initialization, we competitively train the multiclass-classifier: each approximator tests each input sample and generates the approximation error. We generate the label for this sample according to the approximator deriving the lowest approximation error and use this label to train the multiclass-classifier. Then, we test the multiclass-classifier and assign the samples to the approximator that the classifier trusts the most. The same procedure continues iteratively.

Note that multiple approximators may have the similar confidence to approximate the same input sample. Their territories may thus overlap to each other. After some iterations, the bias of each approximator is reinforced, and the multiclass-classifier can easily distinguish the territories of different approximators.

D. Hardware design of MCMA structure

The *MCMA* architecture consists of more than one approximators. Thus, we propose an NPU design that provides instant



(a) The NPU structure and the data flow of MCMA. (b) Structure of a PE.

Fig. 5. Proposed NPU Architecture.

switch among different approximators and enlarges the degree of parallelism. As shown in Fig. 5(a), the NPU consists of identical tiles of computing resource based on [10]. Each tile includes multiple Processing Elements (PEs), Input/output FIFOs, and a Cache, connected by an internal bus. PE calculates the output of one neuron at a time in the inference task of the neural network. The bus scheduler schedules the data from input FIFO to PEs and from PEs to output FIFO through the data bus. It also schedules the weights of the neural network from the cache to each PE.

The structure of the PE, as shown in Fig. 5(b), includes a weight buffer, a fetch unit that reads the weights from the specific addresses of the weight buffer and sends the weights to W register. The I register stores the input sample transferred from Input FIFO. When both values arrive in the registers, a Multiply Add Accumulator unit carries out the arithmetic computation. The result is activated by the activation unit (e.g., Sigmoid) and sent to Output Buffer. We can dynamically allocate the classifier and approximators to the tiles according to the scale of resource and neural networks. Considering the multiple approximators, we deploy a controller which receives the prediction result from the classifier and sends the control signal to the approximator.

Fig. 5 shows the data flow. The weights of the classifier are loaded in the initialization stage. In stage 1, data are transferred from the input FIFO to each PE of the classifier. In stage 2, each PE calculates the output of one neuron and put the output in the output FIFO. In stage 3, the result is sent to the controller from the output FIFO. In stage 4, the controller invokes the approximator if the data are safe-to-approximate, otherwise invokes the CPU. Then in stage 5, the input data are transferred to PEs in the approximator for computation. In stage 6, the results are sent back to the output FIFO.

Our architecture needs to support Weight switch among different approximators, as shown in Fig. 5(b). Three scenarios may occur according to the size of the neural networks. Case 1: the weight buffer can store all weights of the approximators. No weights need to be loaded from the cache. Case 2: the weight buffer is not large enough to store the weights of one approximator. In this case, either the *MCMA* method or any other NPU needs to load the weights layer by layer. So there is no extra overhead compared with previous methods. Case 3: the weight buffer is large enough for storing one approximator but not enough for storing all approximators. In this case, when the prediction of the i_{th} sample is different from the $i-1_{th}$ sample, the approximator loads the weights from cache to the weight buffer.

In summary, the MCMA architecture can adapt to any existing NPU design such as [10] by adding a simple controller for switching the approximations. The data movement between the NPU and other components, e.g., CPU or DMA, remains the same as the original NPU design.

IV. Experiments and Results

A. Experimental setup

In the experiment, we compare the proposed MCCA and MCMA architectures, to the conventional classifier-approximator architecture (denoted as *Iterative* [19] and *one-pass* [18]). For all classifiers and approximators, we use multilayer perceptrons with backpropagation algorithm. *MCCA*, *MCMA* and *Iterative* are trained with five iterations. And *one-pass* with one iteration. In the training, we use RMSprop optimizer and set epoch as 1500.

We select eight benchmark applications from [10] and GNU GSL scientific computing library as shown in Fig 6. The first seven benchmarks are identical from the previous paper [10]. These benchmarks are for comparison among different methods. The Bessel benchmark has two-dimensional input, thus we use images to show the data distribution. In the table, the topology of the neural network is described by the number of neurons in each layer. For example, $6 \rightarrow 8 \rightarrow 1$ means there are 6 neurons in the input layer, one hidden layer with 8 neurons and an output layer with one neuron. In every benchamrk, the approximators and classifiers in all methods are of the same topology, except the last layer of the classifiers in *MCMA* method, due to the multi-classification task. The

#	Benchmark	Domain	Train Data	Test Data	Approximator Topology	Classifier Topology
1	Black-Scholes	Financial Analysis	70K options	30K options	6->8->1	6->8->2(4)
2	FFT	Signal Processing	8K fp numbers	3K fp numbers	1->2->2->2	1->2->2(4)
3	Inversek2j	Robotics	70K (x,y) pairs	30K (x,y) pairs	2->8->2	2->8->2(4)
4	Jmeint	3D gaming	70K traingles	30K traingles	18->32->16->2	18->16->2(4)
5	JPEG encoder	Compression	512*512 pixel color image	512*512 pixel color image	64->16->64	64->16->2(4)
6	K-means	Machine Learning	100K pairs of (r,g,b) points	50K pairs of (r,g,b) points	6->8->4->1	6->8->4->2(4)
7	Sobel	Image Processing	512*512 pixel color image	512*512 pixel color image	9->8->1	9->8->2(4)
8	Bessel	Scientific Computing	70K fp pairs	30K fp pairs	2->4->4->1	2->4->2(4)

Fig. 6. Benchmark description

network topologies are selected by balancing between the performance and the structure size.

We use invocation of classifiers and root-mean-square error(RMSE) of the data approximated by the approximator(simply we call it *error*) to measure our models. The error bound means the quality requirements for the output. The lower the error bound, the higher the quality requirement. We vary the error bounds and show the results considering the various quality requirements among different applications. We also visualize the statistics of the data distribution.

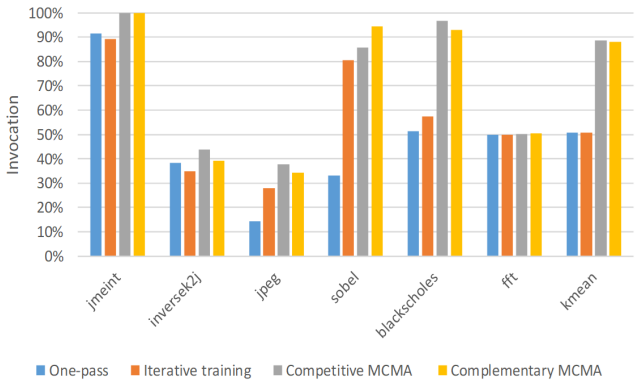
B. Results and analysis

Compared with previous methods, our architecture has i) higher invocation of the approximators while keeping the *error* under the threshold; ii) higher speedup and energy efficiency.

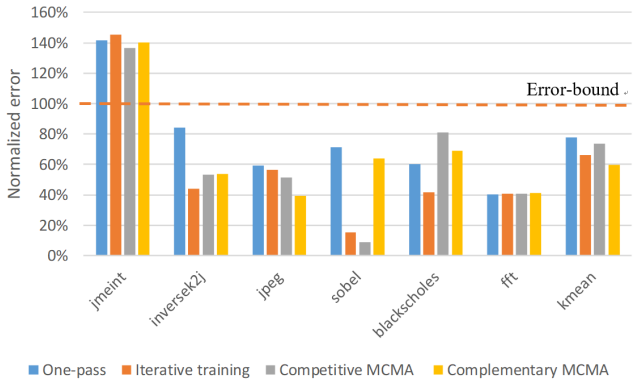
Fig. 7(a) and Fig. 7(b) show the "invocation" and "approximating error" in different benchmarks among *one-pass*, *iterative* and two types of *MCMA* architectures. We normalized the approximation error with respect to the error bound. In average, the *MCMA* methods have greater invocation than the *one-pass* method by 27% and less "approximation error" by 10%. In kmeans and Black-Scholes, our methods outperform the previous methods in invocation by 40%(from 50% to 90%) while the "approximation error" remains unchanged. In most benchmarks, the approximation error is below the error bound which means the quality control is good except the jmeint benchmark. The FFT bench is regarded as "not suitable for approximation" and thus all the methods show no difference.

Fig. 7(c) shows the detailed results on the Black-Sholes benchmark by varying error bound. Invocation of all the methods increases as the error bound arises. When the user requires a tighter error bound, compared with other methods, the drop of invocation of our proposed architecture is the smallest. In other words, the proposed architecture is more desired for those approximate critical applications. In most cases, the *MCMA* architecture has a higher invocation of the approximator than the *MCCA*. Also since the *MCCA* architecture is cascaded which is not time and energy efficient, the rest of the experiment will focus on the *MCMA* architecture.

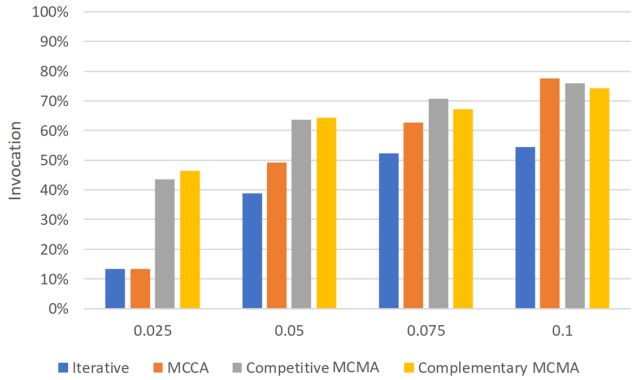
Fig. 8 shows the "speedup" and "energy reduction" in different benchmarks corresponding to the invocation and error in Fig. 7(a) and Fig. 7(b). The results are normalized respect to the *one-pass* method. Due to the space limit, we estimate the performance of MCMA by scaling the performance of NPU in [10] based on the invocation of NPU. This is a valid estimation as the proposed MCMA architecture is merely the same as original NPU design. The average speedup is about 1.23× and the energy reduction is about 1.15× in both *competitive MCMA* and *complementary MCMA*. The



(a) Comparisons on the invocation across different benchmark functions.

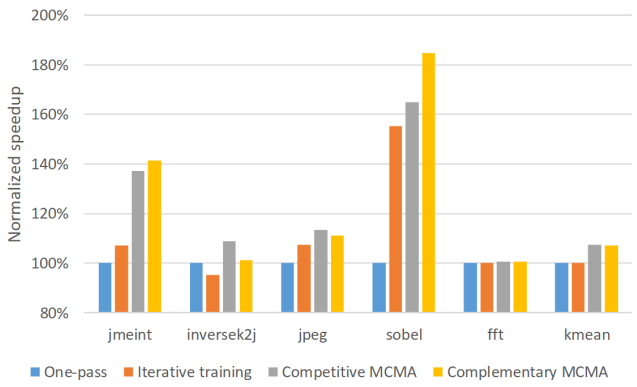


(b) Comparisons on the approximation error normalized to the error bound.

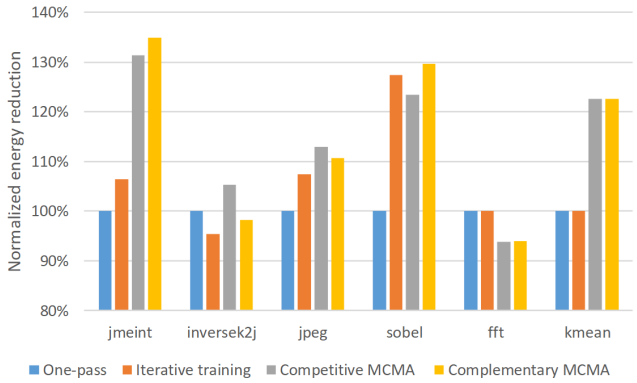


(c) Comparisons on the invocation varying the error bound in Black-Sholes.

Fig. 7. Results of invocation and error.



(a) Comparison on speedup normalized to one-pass method.



(b) Comparisons on energy reduction normalized to one-pass method.

Fig. 8. Comparisons on speedup and energy reduction.

computation gap between CPU and NPU is large, so the speedup is largely determined by the invocation for those computation-bound applications. For those communication-bound applications, neural approximate computing may not be suitable. In the sobel benchmark, the invocation of *MCMA* method is much larger than the *one-pass* method, leading to significant speedup and energy reduction.

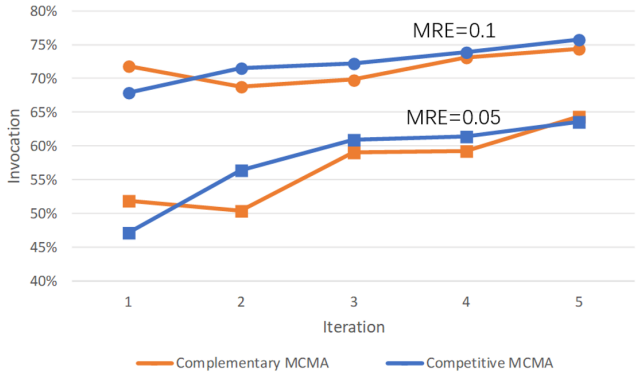
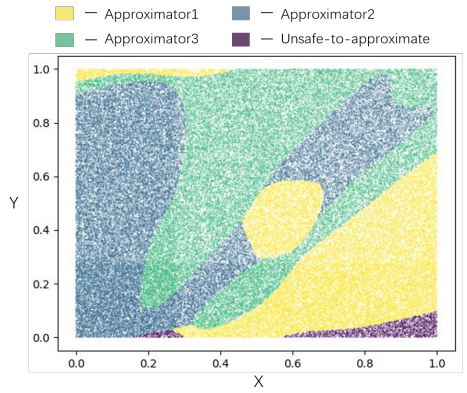
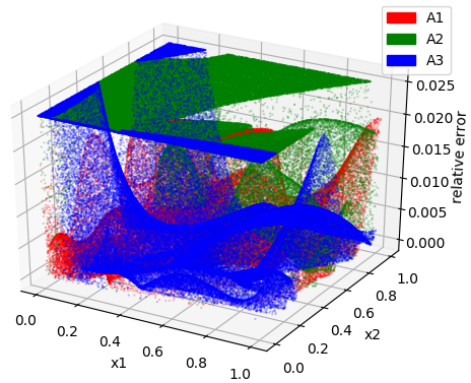


Fig. 9. Comparisons on Invocation between *Complementary* and *Competitive* allocation schemes in MCMA.

Fig. 9 illustrates the invocation rate for two allocation schemes in the iterative training process of the MCMA architecture using Bessel bench. Competitive scheme involves



(a) Distribution of data samples of three approximators.



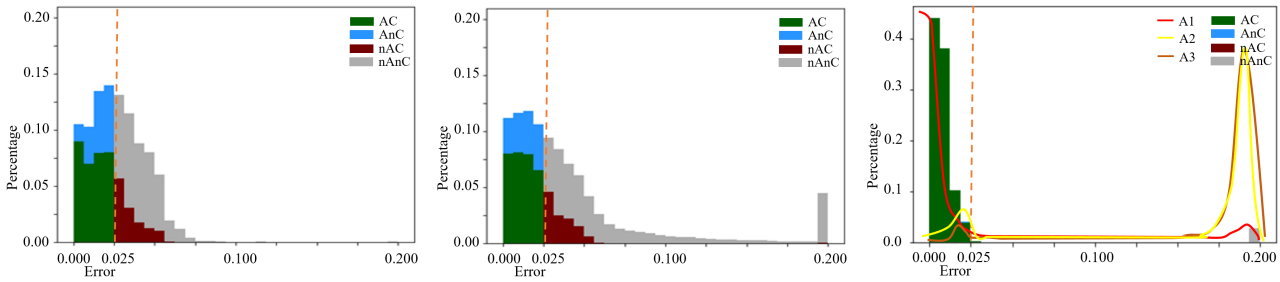
(b) Relative error derived by the three approximators.

Fig. 10. Data distribution of the Bessel bench using the MCMA architecture.

less invocation in the beginning, but it keeps progressing in the following iterations and outperforms the complementary scheme. We also observe that the invocation of approximator using complementary allocation scheme drops in the second iteration. A possible explanation is that the multi-class classifier begins to work until the second iteration. The classifier shuffles the partition of the training data dramatically and redistributes them to all the approximators. The approximators strive to adapt to this drastic change of training data.

Because Bessel bench has two-dimensional input, it is easy to show the data distribution with graphs. Without loss of generality, we studied the data distribution of Bessel bench by plotting the data samples of our three approximators. We first map these samples to a 2-dimension feature space with two input dimension of Bessel function as the X- and Y- axis, as shown in Fig. 10 (a). Each approximator in the MCMA architecture has its own specialty in fitting a cluster of samples. To plot the errors (Z-axis) of the above data samples, we build a 3-dimension distribution of the samples, as shown in Fig. 10 (b). We see that each approximator generates results with a large error in some area. However, with the cooperation of the multi-class classifier, all the approximators approximate a large portion of data samples under the error bound.

Fig. 11 shows the distribution of data samples on approximation error in the *one-pass*, *iterative*, and *MCMA* method. In the legend, A means safe-to-approximate data, C means



(a) one-pass method (b) Iterative method (c) MCMA method

Fig. 11. Data distribution of the Bessel bench along the approximation error

data predicted acceptable by the classifier. So AC means true positive, nAnC for true negative, AnC means false negative, and nAC for false positive. In Fig. 11 (a), after using *one-pass*, many data samples stick around the error bound(dashed line). This phenomenon indicates the hardness to classify the safe-to-approximate data from the others. With *iterative* method, as shown in Fig. 11 (b), the approximator evolves to provide a more accurate output of those accepted data (the green portion); while the classifier becomes more robust to discriminate those unsafe-to-approximate data(fewer samples around the error bound). In Fig. 11 (c), A_i means error distribution curve of the i_{th} approximator. We can see the first approximator covers most of the data and the other two approximators cover the rest part. Our architecture dramatically increases the true invocation rate (true positive with label *AC*). The MCMA architecture almost recognizes all the safe-to-approximate samples (low false negative rate), resulting in a high recall of the classifier. In fact, if we draw a classification hyperplane, the hyperplane is almost vertical and approaches to the error bound, indicating a near-optimal classifier.

V. Conclusion

Neural approximate computing with fine-grain quality control is promising to gain energy-efficiency and performance by the trade-off the tolerable errors in RMS applications. In this paper, we propose new methods to improve the invocation of approximator. Based on the analysis of data distribution, we propose novel approximate computing architectures by orchestrating multiple classifiers and approximators; the corresponding training methods and hardware design are also described. The proposed architectures can dramatically improve the invocation of the approximate accelerator with quality guarantee for larger gain of energy-efficiency.

References

[1] T. Ye et al. Approxma:approximate memory access for dynamic precision scaling. In *Design Automation and Test in Europe Conference.*, pages 337–342, 2015.

[2] S. Douskos et al. Managing performance vs. accuracy trade-offs with loop perforation. In *ACM Sigsoft Symposium on the Foundations of Software Engineering*, pages 124–134, 2011.

[3] Abbas Rahimi, Luca Benini, and Rajesh K. Gupta. Spatial memoization: Concurrent instruction reuse to correct timing errors in simd architectures. *IEEE Transactions on Circuits and Systems II Analog and Digital Signal Processing*, 60(12):847–851, 2013.

[4] M. Samadi et al. Paraprox: Pattern-based approximation for data parallel applications. *SIGARCH Computer Architecture News*, 42:35–50, 2014.

[5] R. Antonio et al. More flops or more precision? accuracy parameterizable linear equation solvers for model predictive control. In *IEEE Symposium on Field Programmable Custom Computing Machines*, pages 209–216, 2009.

[6] S. Sinha and W. Zhang. Low-power fpga design using memoization-based approximate computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(8):2665–2678, Aug 2016.

[7] H. Esmaeilzadeh et al. Architecture support for disciplined approximate programming. *Architectural Support for Programming Languages and Operating Systems*, 47(4):301–312, 2012.

[8] H. Song et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *FPGA*, pages 75–84, 2017.

[9] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[10] H. Esmaeilzadeh et al. Neural acceleration for general-purpose approximate programs. In *IEEE/ACM International Symposium on Microarchitecture*, pages 449–460. IEEE Computer Society, 2012.

[11] S. Liu et al. Cambricon: An instruction set architecture for neural networks. In *International Symposium on Computer Architecture*, pages 393–405. IEEE Press, 2016.

[12] K. Daya S et al. Rumba: an online quality management system for approximate computing. In *IEEE/ACM International Symposium on Computer Architecture*, volume 43, pages 554–566. ACM, 2015.

[13] Grigorian et al. Brainiac: Bringing reliable accuracy into neurally-implemented approximate computing. *High Performance Computer Architecture (HPCA)*, 2015.

[14] S. Adrian et al. Expressing and verifying probabilistic assertions. *Programming Language Design and Implementation*, 49(6):112–122, 2014.

[15] T. Wang et al. On effective and efficient quality management for approximate computing. In *International Symposium on Low Power Electronics and Design*, ISLPED '16, pages 156–161, 2016.

[16] S. Xin et al. Proactive control of approximate programs. *Architectural Support for Programming Languages and Operating Systems*, 51(4):607–621, 2016.

[17] Abbas Rahimi, Luca Benini, and Rajesh K. Gupta. An approximation workflow for exploiting data-level parallelism in fpga acceleration. In *Design, Automation and Test in Europe*, pages 1279–1284, 2016.

[18] M. Divya et al. Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration. *International Symposium on Computer Architecture*, 44(3):66–77, 2016.

[19] C. Xu et al. On quality trade-off control for approximate computing using iterative training. In *Design Automation Conference*, page 52, 2017.

[20] M. Thierry et al. Snnap: Approximate computing on programmable socs via neural acceleration. In *IEEE International Symposium on High PERFORMANCE Computer Architecture*, pages 603–614, 2015.

[21] B. Li et al. Rram-based analog approximate computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(12):1905–1917, 2015.

[22] C. Ping et al. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *International Symposium on Computer Architecture*, pages 27–39. IEEE Press, 2016.

[23] Schapire R E. Freund Y. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 1997.