Energy-Efficient Neural Networks using Approximate Computation Reuse

Xun Jiao[‡], Vahideh Akhlaghi[‡], Yu Jiang[§], and Rajesh K. Gupta[‡]
[‡]Department of Computer Science and Engineering, UC San Diego, CA, USA
[§]School of Software, Tsinghua University, Beijing, China
{xujiao,vakhlagh,gupta}@cs.ucsd.edu, jy1989@mail.tsinghua.edu.cn

***Abstract*—As a problem-solving method, neural networks have shown broad success for medical applications, speech recognition, and natural language processing. Current hardware implementations of neural networks exhibit high energy consumption due to the intensive computing workloads. This paper proposes a methodology to design an energy-efficient neural network that effectively exploits computation reuse opportunities. To do so, we use Bloom filters (BFs) by tightly integrating them with computation units. BFs store and recall frequently occurring input patterns to reuse computations. We expand the opportunities for computation reuse by storing frequent input patterns specific to a given layer and using approximate pattern matching with hashing for limited data precision. This reconfigurable matching is key to achieving a "controllable approximation" for neural networks. To lower the energy consumption of BFs, we also use low-pow memristor arrays to implement BFs. Our experimental results show that for convolutional neural networks, the BFs enable 47.5% energy saving of multiplication operations, while incurring only 1% accuracy drop. While the actual savings will vary depending upon the extent of approximation and reuse, this paper presents a method for reducing computing workloads and improving energy efficiency.**

I. Introduction

Recent advances in neural networks have achieved impressive performance on various application domains such as medical diagnostics [22], image classification [17], speech recognition [13], and natural language processing [6]. The continued success of neural networks has led to their implementation on a variety of hardware platforms [5][12][4]. Energy consumption is an important metric for their implementation in increasingly broad range of computing platforms. Arithmetic operations and memory accesses constitute a significant source of energy consumption. We focus here on reducing the computational workloads in neural networks.

In recent literature, computational workloads have been addressed by using approximations in computations thus creating a tradeoff between accuracy and energy [9][19]. The approximations can be made both in hardware or in software. For instance, approximate computation units have been shown to have better energy efficiency than the exact ones [16]. Neural network computations are dominated by additions and multiplications. Due to their cost and latency, multiplications have been a natural target for optimization in hardware. For instance, in [9], the authors substitute the normal multipliers with inexact multipliers that provide inexact logic but with less hardware cost. Mrazek *et al.* further optimize approximate multiplier design with a uniform structure suitable for hardware implementation [19]. While the adaptability of neural networks in its applications is naturally suited to use approximation, in practice it also requires *retraining* the network to mitigate accuracy loss caused by logic errors from inexact design. Moreover, once the design has been physically implemented in hardware, it is not possible to reconfigure the design to control the approximation level entirely in hardware.

To overcome above-mentioned limitations, we propose using a reconfigurable and controllable approximation technique in neural networks by exploiting the computation reuse opportunities. Computation reuse has been adopted in various applications where *value locality* and *similarity* are observed [20]. To enable computation reuse, we rely on tight integration of Bloom filters (BFs) with the computation units in hardware, a data structure that supports approximate set membership queries with a tunable rate of errors to store frequent computation patterns and return the results without actual execution of energy-intensive float point units (FPUs).

To ensure effectiveness of computation reuse using Bloom Filters, we use a set of techniques. First, we perform approximate pattern matching instead of exact pattern matching in neural networks. This is done in the context of arithmetic operations on floating point numbers. We thus explore matching operations under limited precision of operands. This is done via a reconfigurable BF architecture that can do approximate pattern matching with hashing for data items that feature varying bit width. Second, we perform layer-based pattern matching instead of global pattern matching. That is, we detect and store different set of input patterns for each layer separately. The reason is that in neural networks, each layer has its own set of functions thus may experience different input workloads. Accordingly, we configure BFs for each layer separately. Third, we implement the BFs with resistive memory elements to provide energy efficient storage for saving the frequently used patterns [2].

Based on our implementation and evaluation, this paper makes the following contributions:

- We explore and use computation reuse opportunities in neural networks and enhance them with layer-based approximate pattern matching.
- We design a reconfigurable Bloom filter unit that can perform approximate pattern matching, increasing the

978-3-9819263-0-9/DATE18/©2018 EDAA

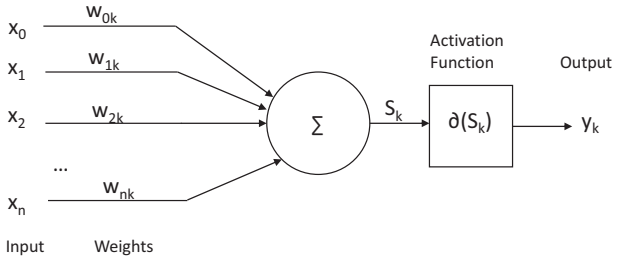

Fig. 1. The computation processes of an artificial neuron.

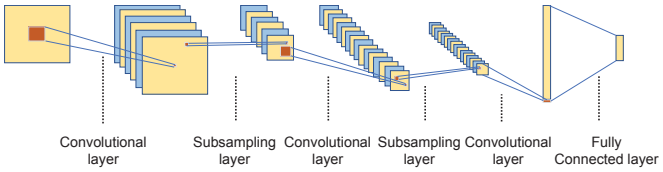

Fig. 2. An illustration of a convolutional neural network.

computation reuse opportunities while leading to a controllable approximation level for neural networks.

- We demonstrate the effectiveness of the approximate BFs by reducing 47.5% energy consumption of multiplication operations in 45nm technology while incurring only 1% accuracy degradation.

II. Neural Networks

Modeled after biological neuronal processing, neural networks are a family of problem-solving methods in machine learning. A neural network is structured as consisting of an input layer, several hidden layers, and one output layer. All layers except the input layer are composed of artificial neurons that perform the basic computations as illustrated in Fig. 1.

A. Neuron Processing

As illustrated in Fig. 1, a typical neuron performs a linear processing part followed by a non-linear processing part. In the linear processing part, inputs are multiplied with corresponding weights, and then all products are accumulated. In the non-linear processing part, an activation function is applied to the weighted sum. Common activation functions include logistic sigmoid, hyperbolic tangent, or rectilinear unit, whose purpose is to enable a neural network to be a universal function approximator [11]. The activation function is usually implemented by a lookup table in hardware [9]. Finally, the output y_k of neuron k is computed as $y_k = \delta(\sum_{j=1}^{n} x_j w_{jk} - \theta)$, where x_j is the j^{th} input, w_{jk} is the synaptic weight connecting j^{th} input and neuron k, θ is the bias, and δ is the activation function. Among the computations in neurons, multiplications are the most energy-consuming part. Thus, we focus on reducing the energy consumption of multiplication in this paper.

B. CNN Architecture

Among different types of neural networks, two of them are most widely used: multi-layer perceptron (MLP) and convolutional neural network (CNN). As one of the simplest neural network model, MLP has one input layer, one output layer, and several hidden layers, where each neuron is directly connected to the outputs of the previous layer. Recently, CNNs have grown in popularity in various applications such as image/video recognition due to its better performance. A CNN applies convolution operations to a restricted part of the input data for each neuron in the convolutional layer. A typical CNN consists of an input and an output layer, as well as multiple hidden layers. The hidden layers can be either convolutional, pooling or fully connected. Fig. 2 depicts a typical CNN architecture that consists of six layers, where the first, third, and fifth layer are convolutional, while the second, fourth are pooling layers, and the sixth layer is a fully connected layer. Consisting of a set of learnable filters, the convolutional layer is the core building block of a CNN. It performs the convolution operations on each filter and a portion of input volume, where a large number of multiplication operations are used, generating a new output image, namely a feature map. Then the pooling layer is used to reduce the size of a feature map by averaging various pixel strengths. This process only preserves the most informational features of input by dropping the unnecessary minor information. We integrate BFs in convolutional and fully connected layers, which account for 98% of multiplications in our experimented neural network [19].

III. Proposed Methodology

In this section, we explore the computation reuse opportunities in neural networks that can avoid the energy overhead due to re-execution, and propose optimization techniques that improve the reuse opportunities. Based on this, we propose a novel architecture that can enable flexible control over computation reuse.

A. Layer-based Pattern Matching

To maximize the energy savings, we need to maximize the computation reuse opportunities. Since we need to store a set of pre-calculated computations, we aim to store most frequent input patterns to maximize the computation reuse opportunities. To do this, we use several steps. First, we profile the input operands of multiplications using some training input. Second, in the profiled input, we look for the most frequent input patterns and calculate their results.

In this process, we use two strategies to look for the most frequent input patterns: *global-based* and *layer-based*. *Global-based* means we look for the most frequent input patterns from all the multiplication operations in neural network inferences, regardless of their locations. *Layer-based* means we look for the frequent input patterns for each layer separately. That is, for each layer, we find the most frequent patterns from the input operands of multiplications profiled from that specific layer. For example, to find the most frequent patterns for the third convolutional layer, we profile all input operands of

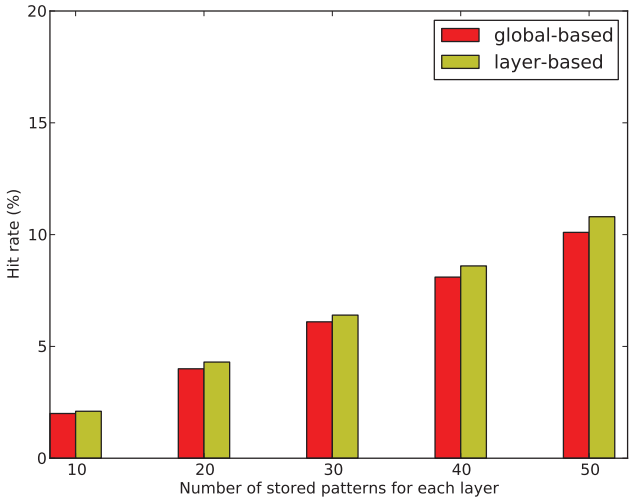

Fig. 3. The hit rate of exact pattern matching.

multiplications in that layer and find the most frequent patterns in this set of input operands.

Third, we then check the hit rate of the chosen frequent patterns using another set of data. We also vary the number of stored input patterns for each layer. Note that, for the sake of simplicity, we always use the same number of stored patterns for each layer. As shown in Fig. 3, we can see that layer-based matching leads to higher hit rate than global-based matching. From now on, we conduct all of our experiments using *layer-based* approach. We also observe that as the number of stored patterns increases, the hit rate also increases. However, the hit rate still remains low, at around 10%, even if we store 50 patterns for each layer. Thus, we improve the hit rate by developing approximation techniques as described in the next section.

B. Approximate Pattern Matching

As shown in Fig. 3, even if we use layer-based pattern matching, the hit rate is still low. Thus, we propose the use of approximate pattern matching for floating point numbers instead of exact matching, i.e., we only match for limited bit width. For example, there are two floating point numbers 0.45 and 0.451, with their IEEE 754 format as 00111110111001100110011001100110 and 00111110111001101110100101111001. If we use exact matching, then 0.451 would not match 0.45. However, if we use 9-bit matching, then 0.451 would match 0.45-because their first 9 bits (sign bit and exponential bits) match. In this case, their first 16 bits (sign bit, exponential bits and 7 mantissa bits) are identical so they will match even under *16-bit matching* mode. We use four different approximate matching modes to measure the hit rate: 9-bit, 10-bit, 11-bit, and exact matching, as illustrated in Fig. 4. We can see that as we increase the approximation level, the hit rate also increases significantly even by 1 bit. For example, by storing 50 patterns (for each layer), a 10-bit approximation can have hit rate at 57.1% while 9-bit approximation can have hit rate at 82.6%, which is 56% higher.

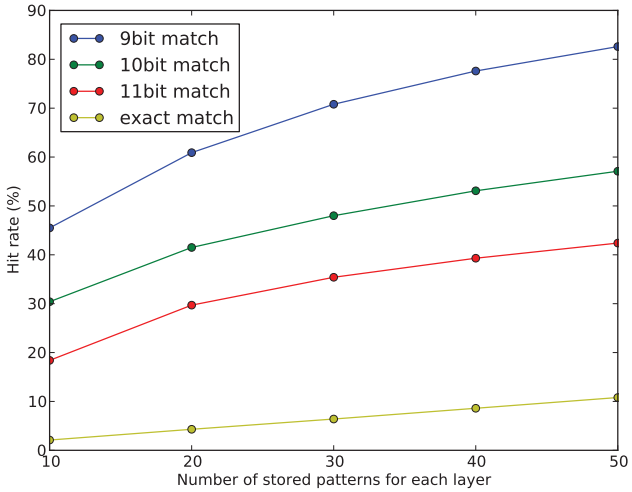

Fig. 4. The hit rate of approximate pattern matching.

But note that the increased hit rate does come with a cost. Rather than returning the exact computation result, the approximate pattern matching will return an inexact result. And as we increase the approximation level, the extent of inaccuracy will also increase. We explore several different approximate matching modes in Section IV-B by varying the matching bit width and number of stored patterns to maximize the energy savings while keeping the accuracy loss minimal.

C. Bloom Filters

To implement approximate pattern matching at the hardware level, we employ BFs. The detected frequent patterns are stored in a set of BFs, and the BFs are integrated to the multiplier. The number of BFs equals to the number of distinct output values generated by the frequent patterns. Each BF stores the patterns corresponding to its assigned output value.

BFs are known as compact storage units that provide an approximate response to the membership queries. A BF consists of a number of hash functions and a Bloom vector (BV). To store a set of inputs in the BF, the hash functions are executed for each input generating addresses to the BV. For each input, the corresponding bits of the BV determined by the hash functions are set to 1. To search for a given input in the BF, the same hash functions generate addresses for an incoming input, and the corresponding bits are checked in the BV. If all the bits are 1, the input is stored in the BF. Otherwise, the input does not exist. The overall architecture of using BF for approximate pattern matching is shown in figure 5. In order to enable the approximate pattern matching, we store approximated input patterns in the BF. We resize each input of the multiplier to *apx_bit* bits by selecting its *apx_bit* most significant bits and concatenate them into a single vector. The obtained vector forms an input to the hash functions which determine bits to be set in the BFs. Similarly, to investigate the approximate pattern matching of incoming inputs to the

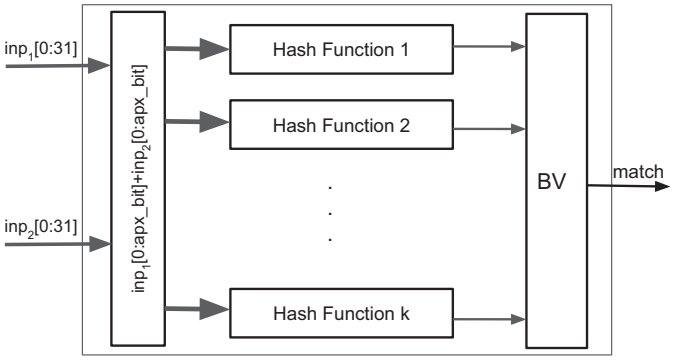

Fig. 5. The implementation of approximate pattern matching.

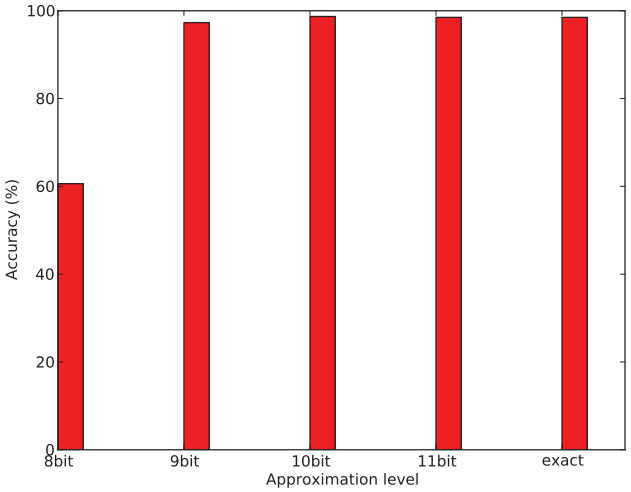

Fig. 6. Neural network accuracy loss due to approximate pattern matching.

multiplier, the inputs are re-sized and concatenated before going to the hash functions. Then, the bits in the BV specified by the hash functions determine whether the incoming pattern of the multiplier is matched or not. In case of matching, the multiplier is clock-gated to avoid the re-execution and the output in the register corresponding to the BF is returned as the output of the multiplier.

Due to the characteristics of hash functions and the limited size of the BV, BF has a false positive (FP) error where BF wrongly confirms the presence of an input. However, the rate of the false positive, which is shown in equation 1, is dependent on several parameters such as the number of input operands stored in the BF (n), the number of hash functions (k) and the size of the BV (m) [8][3]. Therefore, FP can be tuned by properly setting the mentioned parameters.

$$FP = (1 - e^{-\frac{nk}{m}})^k \quad (1)$$

Since most of today's applications such as neural networks demonstrate tolerance to the controlled imprecision in computations, in this work, BFs are adapted to implement approximate pattern matching and recall the computations in neural networks. To further improve the energy consumption of the computations, we employ resistive memory elements to implement Bloom vectors, which exhibit significant energy savings than its CMOS counterparts [2]. Moreover, resistive memory consumes little area overhead as it can be implemented on top of the chip [14].

IV. Experimental Results

A. Experimental Setup

In this work, we use tiny-dnn [1], a header only, dependency free deep learning library written in C++, as our evaluation platform. For CNN, we use LeNet-like architecture as illustrated in Fig. 2. We use MNIST (Mixed National Institute of Standards and Technology) database of handwritten numbers [18] as our dataset to evaluate the accuracy. The dataset is split into a training set and a test set with 60,000 and 10,000 28 $\times$ 28 images. We randomly select 5% of the training input data to profile the frequent input operands. To estimate the energy consumption of the proposed design, we implement the hash functions using Verilog, and we extract the Verilog implementation of a six-stage pipelined floating point multiplier using FloPoCo [7]. Then, the implementations are synthesized using Synopsys Design Compiler, with 45 nm standard CMOS library. The operating voltage is set to 1.0V and the clock period is 1.5 ns. In addition, Bloom vectors (BV) are designed with resistive 1T1R cells using HSPICE, where RON is set to 1K Ω and ROFF to 1M Ω [23]. Bloom filters can be used in different hardware platforms, including CPU [10], FPGA [8], and GPU [2].

B. Accuracy Loss

As described in Section III-B, the BF will return an inexact result due to approximate pattern matching. Thus, we investigate here on how the approximation level impacts the neural network accuracy. We vary the approximate pattern matching mode from 8-bit matching to 11-bit matching and store 10 most frequent patterns for each of the approximation modes with a FP rate of 0.001. Fig. 6 shows the accuracy under each configuration. The baseline accuracy is 98.5% without any approximations. The 8-bit matching introduces aggressive approximation because it does not cover the last bit in the exponent bits, which leads to only 60.6% accuracy. Starting from 9-bit matching, the accuracy loss is insignificant. Note that 9-bit matching covers the sign bit and exponent bits for floating point numbers. This indicates the high error-tolerance of neural networks to data imprecision.

According to Fig. 4, 9-bit matching gives us the highest hit rate among the approximation modes which introduces little drop on neural network accuracy. Thus, using 9-bit matching as our approximation mode, we then investigate how the accuracy will vary with the number of stored frequent patterns. As shown in Table I, various number of stored patterns under 9-bit matching have little impact on neural network accuracy. The lowest accuracy is 97.2% under the (9, 50) configuration, meaning that we use 9-bit approximate pattern matching and store 50 input patterns.

TABLE I
ENERGY SAVINGS AND NEURAL NETWORK ACCURACY ACROSS DIFFERENT BF SETTINGS.

Matching Mode	(BV size, #Hash_Fn, #inp_BF)	Hit Rate	NN Accuracy	E_{save}
(8, 10)	(64, 2, 1)	66.9%	60.6%	58.9%
(9, 5)	(64, 2, 1)	28.9%	97.5%	24.9%
(9, 10)	(64, 2, 1)	45.7%	97.4%	37.9%
(9, 20)	(64, 2, 1)	60.7%	97.9%	45.4%
(9, 30)	(64, 2, 1)	70.6%	97.4%	47.5%
(9, 30)	(32, 3, 1)	70.6%	97.4%	41.6%
(9, 40)	(64, 2, 1)	77.3%	97.3%	47.3%
(9, 50)	(64, 2, 1)	82.3%	97.2%	44.7%
(10, 10)	(64, 2, 1)	30.4%	98.3%	22.3%

C. Energy Savings

We use several different matching modes and BF configuration to compute the energy savings and the resulting neural network accuracy as shown in Table I. The matching mode ($appx\_bit$,$\#inp$) refers to how many bits we use for approximate pattern matching and the number of patterns we store. The BF setting (BV size, #hash_Fn, #inp_BF) refers to BV size in bit length (m), number of hash functions (k) and number of input patterns stored in each BF (n). For example, the BF setting at (64, 2, 1) means that we set the BV size as 64 bits, use 2 hash functions and store 1 input pattern for each BF. To satisfy the FP rate which can lead to acceptable accuracy, we carefully select BF configurations.

Table. I exhibits several important facts. First, 9-bit matching is the optimal matching mode here. By comparing with 8-bit matching and 10-bit matching, we find that 8-bit matching achieves the most energy saving at 58.9%, but its resulted neural network accuracy is only 60.6%, a significant accuracy drop over baseline accuracy of 98.5%. 10-bit matching achieves higher accuracy than 9-bit because it introduces smaller approximation errors into the neural network than 9-bit matching but its resulting energy saving is only at 22.3%, which is less than the one obtained with 9-bit matching mode. Thus, 9-bit matching achieves the better balance between neural network accuracy and energy savings.

Second, after we fix 9-bit matching mode, we then look for the optimal number of patterns to store. We vary the number of stored patterns from 5 to 50. Note that all 9-bit matching modes, regardless of the number of stored patterns, achieve accuracy close to the baseline. Thus, we focus on locating the best energy saving setting. As shown in Fig. 7, we found that the energy saving increases as the number of stored patterns increases from 5 to 30 (we call it the first stage), however the energy saving starts to decrease as the number of stored patterns increases from 30 to 50 (second stage). This is because in the first stage, the hit rate increases as the number of stored patterns increases, which will reduce the use of multipliers. In the second stage, although the hit rate still increases, the energy consumption of BFs increases as the number of stored patterns increases, which dominates the energy consumption. Thus, we find that the optimal matching mode is (9, 30).

Third, we also try different settings of BV size and hash functions. To satisfy the FP error rate of 0.001, we use two realistic BF settings, (64, 2, 1) and (32, 3, 1). The BF setting at (32, 3, 1) consumes more energy than that of (64, 2, 1) because it uses 3 hash functions, which is the main source of energy consumption of BFs. In summary, the optimal configuration is (9, 30) as matching mode and (64, 2, 1) as BF setting. This leads to a neural network accuracy of 97.4% and energy saving of 47.5%.

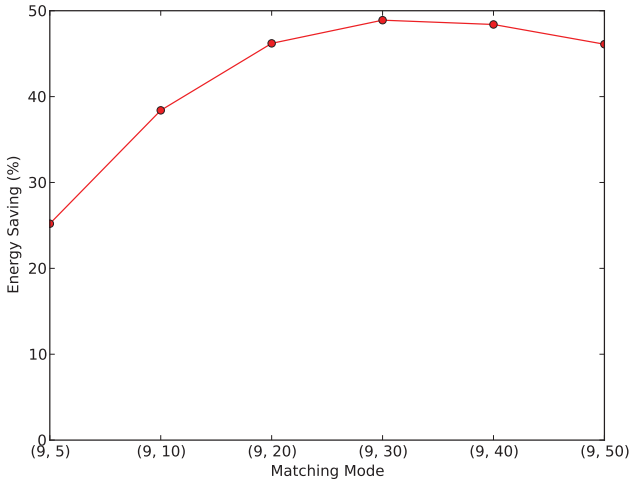

Fig. 7. Energy Savings under different matching mode.

V. RELATED WORK

Thanks to the inherent error resilience of neural networks, various approaches have utilized approximate computing techniques to improve cost and energy efficiency. One typical approach is to use inexact designs to replace normal computation units, mostly multipliers because they account for most energy consumption in computation part. Venkataramani *et al.* evaluates the impact of different neurons on neural network accuracy and selectively approximate the less-critical neurons with dynamically congurable accuracies [21]. A Similar work [24] replaces the less-critical neurons with approximate ones and skip some neuron operations. These two works focus on finding the opportunities for approximate computing without significantly degrade the accuracy. Another two works focus on designing inexact hardware to trade for energy efficiency. Du *et al.* proposed an inexact multiplier design using an inexact logic minimization method, and emphasizes the need to approximate multipliers rather than adders. A hardware optimization approach was proposed in [19] to design multipliers in a uniform way that suits physical VLSI implementation. However, such approaches can result in an accuracy drop and ask for *retraining* to mitigate the accuracy loss. Furthermore, such inexact logic design is not flexible in the sense that it is not reconfigurable once it has been physically implemented, making it less general to different types and architectures of neural networks.

Due to *value locality* and *similarity* presented in various applications, computation reuse has been exploited to improve operational efficiency. Rahimi *et al.* uses content addressable

memory (CAM) to perform computation reuse in GPU and perform voltage scaling on CAM to enable approximate pattern matching within a specified hamming distance [20][15]. However, such approximate pattern match is hard to control because the bit mismatches could also occur in the exponent field. In addition, BFs offer energy efficient storage units because of exploiting small memory unit (i.e., Bloom vector) to represent a set of input patterns. This superiority is acquired at the cost of inaccuracy incurred by FP. CAM consumes relatively large energy, area and manufacturing cost. Therefore, people starting using BFs to enable computation reuse in CPU [10] and GPU [2].

In summary, our work differs from the previous works in two aspects: 1) We propose the first approximate Bloom filters to exploit and enhance the computation reuse opportunities. Such BF design can enable a reconfigurable as well as a controllable approximation by matching patterns with specific bit positions. 2) This approach does not require retraining and the approximation level can be tuned to satisfy the accuracy constraints.

VI. Conclusions

In this work, we exploit the computation reuse opportunities in neural networks and enhance such opportunities by performing approximate pattern matching. We design an approximate BF architecture to physically implement the approximate pattern matching function and tightly integrate it with computation units. By storing the frequent computation patterns, BFs can recall the computation results to avoid the overhead due to redundant executions on computation units. Our experimental results show 47.5% energy reductions of multiplication is obtained with classification accuracy degradation at 1% for convolutional neural networks. Our future works focus on investigating whether the variances of datasets and neural network types and architectures have an impact on the computation reuse opportunities. If so, then we may design a neural network that can maximize the computation reuse opportunities.

References

[1] tiny-dnn: https://github.com/tiny-dnn/tiny-dnn.

[2] Vahideh Akhlaghi, Abbas Rahimi, and Rajesh K Gupta. Resistive bloom filters: from approximate membership to approximate computing with bounded errors. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, 2016.

[3] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrah, Sushil Singh, and George Varghese. Beyond bloom filters: from approximate membership checks to approximate state machines. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, pages 315–326, 2006.

[4] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, volume 49, pages 269–284. ACM, 2014.

[5] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014.

[6] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[7] Florent De Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with flopoco. *IEEE Design & Test of Computers*, 28(4):18–27, 2011.

[8] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John W. Lockwood. Deep packet inspection using parallel bloom filters. In *Proceedings of IEEE Symposium on High Performance Interconnects*, HotI03, pages 44–51, 2003.

[9] Zidong Du, Avinash Lingamneni, Yunji Chen, Krishna V Palem, Olivier Temam, and Chengyong Wu. Leveraging the error resilience of neural networks for designing highly energy efficient accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1223–1235, 2015.

[10] Masayoshi Fujii, Yuuki Sato, Tomoaki Tsumura, and Yasuhiko Nakashima. Exploiting bloom filters for saving power consumption of auto-memoization processor. In *Computing and Networking (CANDAR), 2016 Fourth International Symposium on*, pages 354–360. IEEE, 2016.

[11] G Gybenko. Approximation by superposition of sigmoidal functions. pages 303–314, 1989.

[12] S Himavathi, D Anitha, and A Muthuramalingam. Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization. *IEEE Transactions on Neural Networks*, 18(3):880–888, 2007.

[13] 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. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[14] Mohsen Imani, Shruti Patil, and Tajana Rosing. Approximate computing using multiple-access single-charge associative memory. *IEEE Transactions on Emerging Topics in Computing*, 2016.

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

[16] Honglan Jiang, Cong Liu, Leibo Liu, Fabrizio Lombardi, and Jie Han. A review, classification, and comparative evaluation of approximate arithmetic circuits. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(4):60, 2017.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[18] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998.

[19] Vojtech Mrazek, Syed Shakib Sarwar, Lukas Sekanina, Zdenek Vasicek, and Kaushik Roy. Design of power-efficient approximate multipliers for approximate artificial neural networks. In *2016 International Conference On Computer Aided Design (ICCAD)(prijato)*, page 7, 2016.

[20] Abbas Rahimi, Amirali Ghofrani, Miguel Angel Lastras-Montano, Kwang-Ting Cheng, Luca Benini, and Rajesh K Gupta. Energy-efficient gpgpu architectures via collaborative compilation and memristive memory-based computing. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pages 1–6. IEEE, 2014.

[21] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. Axnn: energy-efficient neuromorphic systems using approximate computing. In *Proceedings of the 2014 international symposium on Low power electronics and design*, pages 27–32. ACM, 2014.

[22] Hongmei Yan, Yingtao Jiang, Jun Zheng, Chenglin Peng, and Qinghui Li. A multilayer perceptron-based medical decision support system for heart disease diagnosis. *Expert Systems with Applications*, 30(2):272–281, 2006.

[23] M. Zangeneh and A. Joshi. Design and optimization of nonvolatile multibit 1t1r resistive ram. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 22(8):1815–1828, Sept 2013.

[24] Qian Zhang, Ting Wang, Ye Tian, Feng Yuan, and Qiang Xu. Approxann: an approximate computing framework for artificial neural network. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 701–706, 2015.