AtomLayer: A Universal ReRAM-Based CNN Accelerator with Atomic Layer Computation

Ximing Qiao, Xiong Cao, Huanrui Yang, Linghao Song, and Hai Li

Duke University, Durham, North Carolina, USA

{ximing.qiao,xiong.cao,huanrui.yang,linghao.song,hai.li}@duke.edu

ABSTRACT

Although ReRAM-based convolutional neural network (CNN) accelerators have been widely studied, state-of-the-art solutions suffer from either incapability of training (e.g., ISSAC [1]) or inefficiency of inference (e.g., PipeLayer [2]) due to the pipeline design. In this work, we propose AtomLayer—a universal ReRAM-based accelerator to support both efficient CNN training and inference. AtomLayer uses the *atomic layer computation* which processes only one network layer each time to eliminate the pipeline related issues such as long latency, pipeline bubbles and large on-chip buffer overhead. For further optimization, we use a unique filter mapping and a data reuse system to minimize the cost of layer switching and DRAM access. Our experimental results show that AtomLayer can achieve higher power efficiency than ISSAC in inference (1.1×) and PipeLayer in training (1.6×), respectively, meanwhile reducing the footprint by 15×.

1 INTRODUCTION

Deep neural networks (DNNs) have been widely used in many industry sectors [3][4]. Conventional GPUs are hard to attain the power efficiency and energy consumption requirements in large-scale data processing [5] and edge/mobile computing [6]. Custom DNN accelerators such as Google TPU [7] and Eyeriss [8] have been proposed. As DNN executions are associated with heavy data movement between the accelerator and memory, the design of these DNN accelerators are generally constrained by the limited memory bandwidth [9].

Processing-in-memory (PIM) recently emerges as a promising candidate for DNN accelerator design, in which acceleration units are placed adjacent or within the memory [10]. One important PIM practice is the design implemented with emerging resistive random access memory (ReRAM), where the accelerator directly stores neural network filters in ReRAM cells to perform the computation. Previous ReRAM-based PIM accelerators, e.g., PRIME [11], ISAAC [1], PipeLayer [2], etc. have achieved significant improvement in both computational and power efficiencies compared to the CMOS-based DNN accelerators.

ReRAM arrays provide very high density in data storage but require long latency and high energy in write operations [12]. Thus

previous designs store the entire neural network on chip and compute all network layers in a pipeline (we refer to it as *parallel layer computation*). As such they only need to write the ReRAM cells for once as far as the neural network parameters do not change. However, since the depth of the pipeline is determined by the depth of neural networks, a very deep pipeline can induce problems such as long latency, pipeline bubbles and large on-chip buffer overhead. Different pipeline designs have been proposed to tradeoff among these issues. For example, ISAAC reduces the on-chip buffer size which results in incapability of neural network training [1]. PipeLayer supports training by sacrificing the inference efficiency and suffering from potential pipeline bubbles [2].

In this work, we investigate the possibility of eliminating the deep pipeline by executing one neural network layer each time (which is referred to as *atomic layer computation*). Although this design requires a large DRAM main memory to store the intermediate data and intense communication between the main memory and the accelerator, it can greatly benefit from low on-chip buffer overhead, low latency and high hardware utilization. Furthermore, we demonstrate that for convolutional neural network accelerations, the required communication can be significantly reduced. Comparing to the previous works, our main contributions include:

- We propose AtomLayer—the first ReRAM-based CNN accelerator that supports the atomic layer computation to solve the pipeline related issues in previous designs including long latency, pipeline bubbles and large on-chip buffer overhead;
- We propose a rotating crossbars structure that provides low-cost switching among the computation of different network layers;
- We propose a row-disjoint filter mapping and a data reuse system that can effectively reduce the DRAM accesses associated with the atomic layer computation;
- We demonstrate that AtomLayer achieves high power efficiency in both inference (1.1× than ISAAC) and training (1.6× than PipeLayer) procedures under a 15× smaller footprint.

The remainder of this paper is organized as follows: Section 2 gives the background about convolutional neural networks and ReRAM-based DNN accelerators; Section 3 introduces the design and working mechanism of AtomLayer; Section 4 describes the evaluation setup; Section 5 presents the experimental results; at last, Section 6 concludes this work.

2 BACKGROUND

2.1 Convolutional Neural Network Basics

2.1.1 Data reuse pattern in convolution layers. The computation of 2D convolution is shown in Figure 1. A convolution kernel (*filter*) starts from the top-left corner of the input feature map, slides to the right end, and then jumps back to the next row. This process can be

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC '18, June 24–29, 2018, San Francisco, CA, USA

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5700-5/18/06...\$15.00

<https://doi.org/10.1145/3195970.3195998>

Figure 1: Two levels of sliding windows in 2D convolution.

represented in a two-level hierarchy: the first level is a window of several rows that slides downward to provide *inter-row* data reuse, and the second level is a window of several elements that slides rightward to provide *intra-row* data reuse.

In CNNs, the convolution is extended to multiple channels and batches, which introduces *inter-channel* data reuse when the input data is shared among different output channels. This extended *4D convolution* defined on *4D tensors* can be written as:

$$\text{output}_l(B, Oc, Oh, Ow) = \text{input}_l(B, Ic, Ih, Iw) * \text{filter}_l(Oc, Ic, Fh, Fw), \quad (1)$$

where l indicates the l^{th} convolution layer. B is the batch size. Ic, Ih and Iw are the number of channels, feature map height, and feature map width of the input. Oc, Oh and Ow are the corresponding parameters of the output. Fh and Fw are the height and width of the filters. In Equation (1), we omit the bias for simplicity of expression. The computation of the 4D convolution then is:

$$\begin{aligned} \text{output}_l[b][oc][x][y] &= \sum_{ic=0}^{Ic-1} \sum_{i=0}^{Fh-1} \sum_{j=0}^{Fw-1} \text{input}_l[b][ic][x+i][y+j] \\ &\quad \times \text{filter}_l[oc][ic][i][j], \\ 0 \leq b < B, 0 \leq oc < Oc, 0 \leq x < Oh, 0 \leq y < Ow. \end{aligned} \quad (2)$$

Note that although fully connected layers in a CNN do not involve any data reuse, they can be represented in the form of 4D convolutions by setting the feature map and filter sizes to one.

2.1.2 CNN training. The CNN training includes three stages, each of which involves convolutions with different input and filter tensors. The first stage is the forward stage, in which the convolutions are the same as those in the CNN inference. The second stage is a backward procedure to back-propagate errors. It rotates the filter tensors and uses the errors as the input tensors, such as: $\text{error}_l(B, Oc, Oh, Ow) * \text{rot}_{180}(\text{filter}_l(Oc, Ic, Fh, Fw))$. The third one is the gradient computing stage, which uses the input tensors in the forward pass as filters and the rotated errors as the input tensors. In addition, the positions of batches and channels are swapped. The representation of the gradient is $\text{rot}_{180}(\text{error}_l(Oc, B, Oh, Ow)) * \text{input}_l(Ic, B, Ih, Iw)$.

2.2 ReRAM-Based DNN Accelerators

Figure 2 shows the basic functional unit of a ReRAM-based DNN accelerator called *crossbar*. Elements of a matrix operand are represented as the conductances of every ReRAM cell located at each cross-point of the crossbar, and the vector operand is represented as the voltage input of the wordlines of the crossbar. In this form, the matrix-vector multiplication product can be obtained by sensing the current output from the bitlines of the crossbar. Since the ReRAM crossbar naturally supports vector-matrix multiplication, a naive

Figure 2: Vector-matrix multiplication performed on a ReRAM crossbar with peripheral devices.

method to perform convolution on ReRAM crossbars is to turn the convolution into a matrix multiplication by adding redundant data. This method is used in ISAAC, PipeLayer, and the baseline model of this work.

The precision of data stored in each cell is limited by the number of programmable resistance levels of ReRAM devices. A high precision can be achieved by combining multiple ReRAM cells that have low data precision on each. For example, the bitline currents are first sampled by sample-and-hold (S&H) units and then converted into digital values by a shared ADC; finally, these digital values are accumulated using a shift-and-add (S&A) unit [1].

2.3 State-of-the-art Designs and Their Limitations

ISAAC [1] uses ReRAM crossbars for filter storage and matrix multiplication. Besides the ReRAM crossbars, eDRAM buffers are used to cache inputs and outputs. A region based pipeline allows layer $i + 1$ to initiate as soon as layer i produces enough outputs to cover a filter. Therefore, only part of the feature map needs to be buffered. This design effectively reduces the requirement of eDRAM buffer capacity but also makes ISAAC incapable to perform training, as neural network training requires all intermediate data to be stored. For communication between pipeline stages, ISAAC uses routers and H-trees to connect the computing units, which contribute to 1/5 of the total power and a half of the total area.

PipeLayer [2] stores all filters, inputs, and outputs in morphable ReRAM crossbars, which can be used for both caching and computing. This design supports neural network training. However, the inference power efficiency is significantly lowered as many unnecessary ReRAM writes are introduced. PipeLayer uses a batch based pipeline to reduce the buffer size which potentially introduces pipeline bubbles when the neural network is deep.

3 ATOMLAYER ARCHITECTURE

3.1 System Overview

Figure 3 depicts the top-level architecture of the proposed Atom-Layer accelerator. The design shares a DRAM main memory with a CPU or a GPU. This main memory stores the initial neural network filters, the first layer inputs, and all intermediate data generated in the neural network inference or training.

AtomLayer contains an array of processing elements (PEs), an on-chip network, an ALU tree, and a global output buffer (GOB). The PEs load all the filters from the DRAM and compute all convolutions. The on-chip network has three parts for different functions: 1) an input network (INet) fetches inputs and filters from the DRAM

Figure 3: Top-level AtomLayer architecture.

and broadcasts them to one row of PEs; 2) a local network (LNet) manages the inter-PE communication between vertically adjacent PEs; and 3) an output network (ONet) gathers the PE outputs and passes them to the ALU tree. The ALU tree consists of multiple column-ALUs (CALU) and one global ALU (GALU). It performs partial sum accumulation, pooling, and non-linear activations. GOB stores intermediate results (i.e., partial sums) when the final output cannot be generated in one cycle.

Figure 4 presents the PE architecture which includes three main components: rotating crossbars, peripheral devices, and four buffers. The rotating crossbars are used for storing filters and computing convolution. The peripheral devices include a set of write DACs to write new filters as well as a set of read DACs, S&H, ADCs, and S&A to assist the crossbar computation. The filter buffer (FB) connects the INet to the write DACs and updates every cycle when writing new filters. The output buffer (OB) receives the outputs from the S&A and enables the ONet to move them to ALUs. The input buffer (IB) and register ladder (RL) together with the INet and LNet form the data reuse system. In this work, we use the same configurations of the ReRAM crossbars and the peripheral devices as those in ISAAC [1].

3.2 Rotating Crossbars

The rotating crossbars are the key components to perform atomic layer computation. Our idea behind this is to include the filters of all the neural network layers in every PE by leveraging the abundant ReRAM crossbars, while within each PE, not all the layers are processed simultaneously. Due to the high density and the zero idle power of ReRAM crossbars, the peripheral devices (i.e., read DAC, S&H, ADC, and S&A) will be the main constraint. Accordingly, we define *crossbar set* as the basic unit of the rotating crossbars. Each crossbar set contains a certain number of ReRAM crossbars so that its peak computing performance exactly matches the performance of the peripheral devices. In every cycle, one crossbar set is connected to the read DACs and S&H for computation, and another crossbar set is linked to the write DACs for filter writing (which are indicated by the green and red lines in Figure 4, respectively). The number of crossbar sets shall be no less than the number of the neural network layers, in order to guarantee that the filters from different layers are stored in different crossbar sets.

In an inference task, we load all the filters to the ReRAM crossbars before the computation starts. The accelerator reads inputs from the

Figure 4: Architecture of a processing element (PE).

DRAM, processes one convolution layer, one optional pooling layer, one optional non-linear layer, and then writes the final outputs back to the DRAM. This process is repeated through all the layers by changing the read DAC and S&H connections according to the layers' order.

The training procedure is more complex than the inference process due to the update of parameters and the involved ReRAM writes. Accordingly, we design the following procedure to hide the write latency. The forward stage is the same as in the inference task though the inputs are processed in batches, i.e. the first layer of the entire batch is computed first, then followed by the execution of the next layer of the same batch, and so on. Such a batch based processing allows us to overwrite the filters of the finished layers with the new one required in the backward stage. After the forward stage of a batch finishes, the backward stage for the same batch starts instantly as all its filters are ready. Again, the used filters are replaced by the inputs values in the forward stage to compute the gradients. In the gradient computing stage, when one layer is being processed, the input values of another layer in the forward stage are written concurrently. Consequentially, no stall will occur during the entire training process.

3.3 Filter Mapping and Data Reuse System

AtomLayer uses a data reuse system to reduce the required DRAM bandwidth and access energy. The key of the proposed technique is to use a *row-disjoint* filter mapping to distribute the data reuse hierarchy levels into multiple hardware levels, and design the corresponding hardware structure for each level. Specifically, the intra-row data reuse is restricted in each PE and realized by a *register*

Figure 5: The row-disjoint filter mapping in PE block (left) and PE array (right).

Figure 6: The register ladder (RL). Each square represents a multiplexer-register pair. Red arrows indicate the source of the data. Numbers 1-5 denote the five inputs in the first row, and 1'-3' represent the first three inputs in the second row.

ladder structure, the inter-row data reuse is restricted among several adjacent PEs and realized by a *buffer ladder* structure, and the inter-channel data reuse is restricted to each PE row, which can be directly supported by the input network (INet) broadcast.

In this section, the crossbar size, filter size, and input feature map size are assumed to be 128×128 , 3×3 and 5×5 for demonstration.

3.3.1 Row-disjoint filter mapping. The row-disjoint filter mapping maps the same filter row to the same PE, and adjacent filter rows to adjacent PEs. As illustrated in Figure 5, the mapping contains three steps: First, we map the filters of every 42 (crossbar height divided by filter width) input channels and 128 output channels to every three vertically adjacent PEs, which comprise a *PE block*. Second, within each PE block, the three rows of the 42×128 filters are mapped to the three PEs according to their row numbers. At last, the 42×128 filter rows in each PE are reshaped to fit the 128×128 crossbar: rows belonging to the same output channel are concatenated in ascending order of the input channel number, and rotated into a 126-height column. Then the 128 columns from different output channels are merged into a matrix and mapped to the crossbar. An accurate description is shown as follows: we assume n and m denote the row index and column index of a PE in the PE array, while y and x denote the row index and column index of a ReRAM cell in the crossbar, then the 4D index of the filter in the cell is:

$$\text{filter}(128m + x, \lfloor 128/3 \rfloor \lfloor n/3 \rfloor, n\%3, y\%3), \quad (3)$$

in which $A\%B$ denotes the remainder of A divided by B .

3.3.2 Register ladder. The register ladder is located in each PE to support the intra-row data reuse. The goal of this structure is to simulate a group of FIFOs to read the inputs. So for each input channel, we can have a simulated FIFO with its first three entries connected to the DACs to provide the inputs.

Each register ladder consists of 256 16-bit multiplexer-register pairs, which are arranged in two columns as shown in Figure 6. The register ladder has two function modes: *buffer mode* and *FIFO mode*. In buffer mode, the left column fetches 128 inputs from the input buffer (IB), and the right column passes the current values in the left column to the DACs. In FIFO mode, registers are configured into several U-shapes, each of which functions as a FIFO (as depicted in Figure 6, cycle 3). The working procedure of RL is as follows: In the first two cycles, both columns work in buffer mode. All five elements of an input row are read and the first three of them are

Figure 7: The buffer ladder (BL). Each input buffer has two write ports connected to the INet and LNet, and one read port connected to the left column of RL.

passed to the DACs. In cycle 3, the top six multiplexer-register pairs form a FIFO. The formed FIFO pops out the first element and passes the second to the fourth elements to the DACs. In cycle 4, the right column still works in FIFO mode while the left column switches to buffer mode and reads three inputs of the next row (indicated by the apostrophes). By repeating the same procedure, the register ladder can pass five inputs from the input buffer to the DACs every three cycles. Note that the input buffer access is always in the form of regular 128-element blocks.

3.3.3 Buffer ladder. The input buffers in one PE column form a buffer ladder, which supports the inter-row data reuse in PE blocks. As depicted in Figure 7, we name the PEs in a PE block as PE1-PE3. According to the filter mapping, PE1 contains the first filter row and PE2 contains the second filter row. As indicated by the inter-row data reuse, for example, the second input row should be first used in PE2 and then reused in PE1. The buffer ladder connects input buffers in adjacent PEs via the LNet and assigns two write ports and one read port to each input buffer, in order to support the simultaneous movement of new and old data.

Figure 7 uses the same notations as those in Figure 6 to show how BL cooperates with RL. In the first two cycles, PE1 reads the first input row (indicated by numbers 1-5) and PE2 reads the second input row (indicated by 1'-5'), both from the INet. When a block of inputs is sent to the RL in PE2, it overwrites the corresponding block in PE1 via the LNet. By following this procedure, when PE1 finishes computing the first input row, the second input row is always ready.

3.4 Support for Different Layer Shapes

The above filter mapping and data reuse procedure assume that the size of PE array and capacity of IB fit the layer's shape perfectly. Considering a general solution for various possible network structures and layer shapes, the following adaptations are necessary.

First, the dimension of the physical PE array could be incompatible with the filter mapping:

- (1) If the mapping requires more PE rows, the extra rows will overflow to the next physical PE column.

Table 1: Design parameters used in our evaluations.

Component	Parameters	Number	Power (mW)	Area (mm ²)
ADC	8bit, 1.2GSps	8	16	0.0096
DAC	1bit	8×128	4	0.00017
S&H		8×128	0.01	0.00004
ReRAM	8×128×128,	Total 16,		
Crossbar	2bit/cell	1 active	2.4	0.0032
S&A		4	0.2	0.00024
Input Buffer	2KB	1	1.24	0.0021
Reg. Ladder	512B	1	0.39	0.00162
Filter Buffer	256B	1	0.23	0.00077
Output Buffer	256B	1	0.23	0.00077
1 PE Total			24.70	0.01851

- (2) If the mapping requires more PE columns, only part of the output channels will be processed at one time. The filter weights will be distributed to multiple crossbar sets.
- (3) If one column of the mapping requires more PEs than the whole physical PE array has, only part of the input channels will be processed at one time, and GOB will store the partial sums.
- (4) If the required PE number is too small, we replicate the mapping to cover the entire physical PE array.

Second, input buffers might not be able to cache an entire input row when the feature map size is large. Then the input row will be split into multiple segments and processed in multiple passes.

Third, the crossbar height can be less than the length of a filter row, which usually happens in the gradient computing stage. If so, multiple register ladders will be connected through the buffer ladder to form longer FIFOs.

4 EVALUATION MODELS

4.1 Energy and Area Models

Our evaluation adopted the same configurations of the ReRAM crossbars and the peripheral devices (ADC, DAC, S&H, and S&A) as those used in ISAAC [1]. The models of SRAM buffers (IB, FB and OB) and register files (RL) are extracted from CACTI [13]. The access energy of the off-chip DRAM is set to 20pJ/bit according to EIE [14]. The write energy of ReRAM cells is obtained from PipeLayer [2], or 3.91nJ per cell. The detailed parameters and power/area values of the main components are summarized in Table 1.

In each benchmark, the number of crossbar sets in each PE is selected to fit the network’s structure, from 16 for VGG-19 (w/o FC layers) to over 150 for ResNet-152. The precisions of inputs, outputs and filters are all set to 16-bit according to previous works [1][2], so that each crossbar set contains eight ReRAM crossbars.

4.2 Performance Model

In our experiments, we assume that the ReRAM crossbars always run at their peak performance, which performs 128×128 multiplications and additions every 16 cycles, and the period of each cycle is 100ns [1]. We also assume that the on-chip network and the ALUs do not form any bottleneck. Computational efficiency (GOPs/s/mm²) and power efficiency (GOPs/W) are the two metrics to compare our work to ISAAC and PipeLayer.

Figure 8: (a) Energy breakdown on different data reuse levels. (b) Power efficiency on different DRAM energy choices.

4.3 Benchmark Networks

We use VGG-19 [15], a widely used CNN architecture with 16 convolution layers and 3 fully connected layers as the benchmark in our design space exploration and performance evaluation. In order to fully investigate the advantages of AtomLayer, we also use ResNet-152 [16] and DCGAN [17] as the benchmarks to evaluate inference latency and training efficiency. ResNet-152 and DCGAN are representative networks of those with complex data dependencies: one has a huge layer number, and the other has alternate access to two different networks. Among these benchmarks, the 16-bit precision is enough for inference [18] but requires some special techniques in training to achieve a high accuracy [19].

5 RESULTS

5.1 Analysis

5.1.1 Effect of rotating crossbars. Rotating crossbars can significantly reduce the ReRAM write energy in inference tasks. For VGG-19, the energy required to update all neural network weights is at least 0.63W for convolution layers, and 3.87W for FC layers, while processing them only needs 57mW and 0.7mW per image, respectively. Rotating crossbars eliminate all ReRAM writes in computation. Without it, the inference power efficiency will be lowered by 38% even if the inputs are processed in 128-image batches.

The main disadvantage of rotating crossbars is the area overhead of the extra crossbars, which can be heavily affected by the number of PEs and the size of FC layers. For VGG-19, when the FC layers are not considered, a 160PE system requires 16 crossbar sets in each PE (corresponds to the 16 convolution layers), and a smaller 40PE system needs 44 (layer conv3-1 to conv5-4 requires multiple crossbar sets). From Table 1, we can see that the 16 crossbar sets cover only 1.7% of the PE area. However, when the FC layers are considered, the 160PE system requires 64 crossbars sets in each PE, and the required set number of a 40PE system increases to 233, taking over 75% of the total chip area. Only when the PE number is increased to 2,016, which is the same as in ISAAC, the crossbar set number drops back to 21. Despite of this, for other networks like the ResNet family, the size of FC layers is much smaller.

5.1.2 Effect of data reuse system. The data reuse system minimizes the communication between the accelerator and off-chip DRAM. Without the data reuse, as demonstrated by the baseline model in Figure 8(a), the DRAM access can consume 43% of the total power on average and up to 70% in some layers. In our experiments, the communication reduction rate is affected by the IB size: a small IB can disrupt the intra-row reuse flow. The results show

Table 2: Inference performance results.

	GOPs/s/mm ²	GOPs/W	mm ²	W
160PE (w/o FC)	615.7	682.5	5.32	4.80
160PE (w/ FC)	475.6	682.5	6.89	4.80
ISAAC	478.9	627.5	85.4	65.8
PipeLayer	1485	142.9	82.6	168.6

Table 3: Inference latencies.

	VGG-19 (ms)	ResNet-152 (ms)
160PE	6.92	4.01
ISAAC	8.00	43.46
PipeLayer	2.60	14.13

that a 2KB IB in each PE provides the best tradeoff between the low buffer overhead and high communication reduction. This configuration lowers the number of DRAM read accesses by 7.3× and total access by 4.6×, and consequently, reduces the DRAM bandwidth requirement from 18.5GB/s to 4.0GB/s.

The communication reduction also decreases the DRAM energy cost. As shown in Figure 8(b), AtomLayer shows significantly higher power efficiency than the baseline model without the data reuse system and remains a comparable power efficiency to ISAAC when the DRAM access energy is up to 40pJ/bit.

5.2 Comparison to State-of-the-art Techniques

5.2.1 Inference. As shown in Table 2, our 160PE system outperforms ISAAC by 1.1× on inference power efficiency and improves the computational efficiency by 1.3× if FC layers are not considered. The main reason is that AtomLayer has much smaller on-chip buffer size and simpler on-chip communication. Comparing to PipeLayer, the power efficiency improvement achieves 4.7× due to the avoidance of frequent ReRAM writes, while the computational efficiency is lower because of the difference in peripheral devices.

The biggest benefit of atomic layer computation compared to previous pipeline designs is its ability to combine low power, small footprint, and low latency. With the total power less than 5W and 12× to 15× (affected by FC layers) reduction in footprint, AtomLayer achieves a comparable inference latency to ISAAC and PipeLayer on VGG-19 (Table 3). For ResNet-152, AtomLayer has even lower latency due to the less computation, while the pipeline designs require 5× longer time due to the 8 times deeper pipeline.

5.2.2 Training. Table 4 shows the energy breakdown in the training procedures of VGG-19 and ResNet-152 using a batch size of 128. As can be seen that the frequent ReRAM writes involved in neural network training consumes the majority of the energy. These extra costs reduce the power efficiency of AtomLayer to 232.9GOPs/W on VGG-19 training and even lower on ResNet-152 training. This result is still 1.6× higher than the one of PipeLayer (142.9GOPs/W), but this divergence might be mainly caused by the

Table 4: Training energy breakdown.

	VGG-19 w/o FC	VGG-19	ResNet-152
DRAM access	3.8%	3.4%	4.9%
ReRAM write	71.9%	74.3%	83.5%
Others	24.4%	22.2%	11.6%

different choice of the peripheral devices, since both designs have the same amount of ReRAM writes.

Another advantage of AtomLayer is that it prevents pipeline bubbles. In the training of ResNet-152, PipeLayer can only achieve 46% of its peak performance, because a 304-stage pipeline needs to be filled and cleaned in each training batch. For DCGAN, in 50% of the training time, half of the network needs to stop and wait, resulting in 68% hardware utilization, even if the network structure is not very deep. On the other hand, the atomic layer computation allows AtomLayer to work at a full speed regardless of the network structure complexity.

6 CONCLUSIONS

The previous research studies, i.e., ISAAC and PipeLayer, have demonstrated promising performance of ReRAM-based CNN accelerators. However, the parallel layer computation scheme adopted by these techniques introduces some design drawbacks such as the incapability of ISSAC to perform neural network training and the low power efficiency of PipeLayer in inference. In this work, we propose AtomLayer—the first ReRAM-based DNN accelerator that supports both efficient inference and training. AtomLayer improves power efficiency by 1.1× over ISSAC in inference and 1.6× over PipeLayer in training, and reduce the accelerator footprint by 15×.

ACKNOWLEDGMENT

This research is supported by NSF CCF-1744082, NSF CNS-1744111 and DOE SC0017030.

REFERENCES

- [1] Ali Shafiee et al. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *ISCA*, 2016.
- [2] Linghao Song et al. PipeLayer: A pipelined ReRAM-based accelerator for deep learning. In *HPCA*, 2017.
- [3] Alex Krizhevsky et al. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.
- [4] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.
- [5] Dominik Scherer et al. Accelerating large-scale convolutional neural networks with parallel graphics multiprocessors. *ICANN*, 2010.
- [6] Yong-Deok Kim et al. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.
- [7] Norman P Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017.
- [8] Yu-Hsin Chen et al. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *JSSC*, 52(1), 2017.
- [9] Yunji Chen et al. Dadiannao: A machine-learning supercomputer. In *MICRO*, 2014.
- [10] Boxun Li et al. Memristor-based approximated computation. In *ISLPED*, 2013.
- [11] Ping Chi et al. Prime: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *ISCA*, 2016.
- [12] Dimin Niu et al. Design trade-offs for high density cross-point resistive memory. In *ISLPED*, 2012.
- [13] Naveen Muralimanohar et al. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *MICRO*, 2007.
- [14] Song Han et al. EIE: efficient inference engine on compressed deep neural network. In *ISCA*, 2016.
- [15] Karen Simonyan et al. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [16] Kaiming He et al. Deep residual learning for image recognition. In *CVPR*, 2016.
- [17] Alec Radford et al. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.
- [18] Darryl Lin et al. Fixed point quantization of deep convolutional networks. In *ICML*, 2016.
- [19] Darryl Lin et al. Overcoming challenges in fixed point training of deep convolutional networks. *arXiv preprint arXiv:1607.02241*, 2016.