Flipping Bits to Share Crossbars in ReRAM-Based DNN Accelerator

Lei Zhao
University of Pittsburgh
Pittsburgh, PA, USA
leizhao@cs.pitt.edu

Youtao Zhang
University of Pittsburgh
Pittsburgh, PA, USA
zhangyt@cs.pitt.edu

Jun Yang
University of Pittsburgh
Pittsburgh, PA, USA
juy9@pitt.edu

***Abstract*—Future deep neural networks (DNNs) tend to grow deeper and contain more trainable weights. Although methods such as pruning and quantization are widely adopted to reduce DNN's model size and computation, they are less applicable in the area of ReRAM-based DNN accelerators. On the one hand, because the cells in crossbars are accessed uniformly, it is difficult to explore fine-grained pruning in ReRAM-based DNN accelerators. On the other hand, aggressive quantization results in poor accuracy coupled with the low precision of ReRAM cells to represent weight values.**

In this paper, we propose BFlip – a novel model size and computation reduction technique – to share crossbars among multiple bit matrices. BFlip clusters similar bit matrices together, and finds a combination of row and column flips for each bit matrix to minimize its distance to the centroid of the cluster. Therefore, only the centroid bit matrix is stored in the crossbar, which is shared by all other bit matrices in that cluster. We also propose a calibration method to improve the accuracy as well as a ReRAM-based DNN accelerator to fully reap the storage and computation benefits of BFlip. Our experiments show that BFlip effectively reduces model size and computation with negligible accuracy impact. The proposed accelerator achieves 2.45× speedup and 85% energy reduction over the ISAAC baseline.

***Index Terms*—neural network, ReRAM, accelerator, crossbar, process-in-memory**

I. Introduction

Deep neural networks (DNNs) have become an effective solution in many classification and regression tasks including computer vision [1], natural language processing [3], speech recognition [2], etc. However, its superior accuracy is achieved by millions of parameters (also known as weights) and computation. The size of DNNs has been keeping growing for almost a decade to constantly achieve better accuracy. For example, AlexNet [1] – the first DNN winner of the ImageNet challenge in 2012 – only contains 60M parameters and 727M operations per input, VGG [4] won this challenge in 2014 with 136M parameters and 15B operations, and most recently one of the state-of-the-art DNNs EfficientNet-L2 [5] has 480M parameters and 612B operations. As DNN's model size and the number of operations continue to grow, there are more demands in new underlying computation hardware that could promise less data movement overhead and more efficient computation.

Process-in-memory (PIM) using emerging memory technologies, such as metal-oxide resistive random access memory (ReRAM) [6], spin-transfer torque magnetic RAM (STT-RAM) [7], and phase change memory (PCM) [8], has shown to be a promising solution to meet the above challenges. Among them, ReRAM is the most widely studied approach. Massive data movement can be eliminated because computation takes place in the memory. The crossbar structure only needs a single operation to perform the matrix-vector multiplication (MVM) based on Kirchhoff's current law, thus achieving significant computation parallelism.

However, although ReRAM is denser than traditional SRAM and DRAM, it still faces the pressure of DNN's growing model size. Because of ReRAM's low endurance and slow write speed, existing ReRAM-based DNN accelerators require weights to be statically mapped on the crossbars to avoid frequently programming the cells. Therefore, more than hundreds of thousands of crossbars are needed to execute state-of-the-art DNNs.

Unfortunately, existing methods that exploit weight sparsity and weight redundancy can not be effectively applied to ReRAM-based DNN accelerators. Pruning [9] removes the values that are close to zero from the weight matrix. However, because ReRAM crossbars rely on regular MVM, exploiting the random and irregular distribution of zero weights requires complex control and peripheral circuits. Quantization [10] is another method that uses shorter bit width to represent weight values. However, quantization below eight bits usually incurs significant accuracy degradation [11], [12].

In this paper, we propose $BFlip$ to flip the bits in crossbars so that multiple bit matrices can share the same crossbar. Because the bits in a crossbar are accessed uniformly, separately flipping each bit induces large metadata and complex control logic. Therefore, we propose to only flip the bits in the granularity of rows and/or columns in a crossbar. We first cluster similar bit matrices together and then flip them to match the cluster's centroid bit matrix. For each cluster, only the centroid bit matrix is stored in the accelerator while all other bit matrices will be reconstructed from the centroid bit matrix during inference. However, since the bits can only be flipped in the granularity of rows and columns, it is very likely that a bit matrix cannot be flipped to perfectly match the centroid bit matrix. We apply a post-flipping calibration method that only updates the distribution statistics of batch normalization (BN) layers to mitigate the precision loss. Finally, We propose a ReRAM-based accelerator that fully reaps the storage and computation benefits of $BFlip$.

In summary, we make the following contributions in this paper:

- We propose a novel bit-flipping scheme to map multiple bit matrices onto the same crossbar to effectively reduce

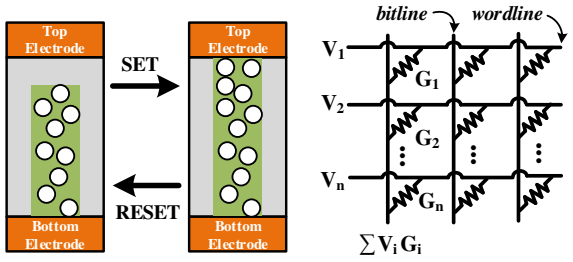


Fig. 1: ReRAM and crossbar structure.

DNN's model size and computation.

- Our proposed post-flipping calibration method only needs to run the forward propagation on training data, without the need for backward propagation and weight update. So our method is much faster than retraining and fine-tuning which are used in conventional pruning and quantization techniques.
- A full-fledged ReRAM-based DNN accelerator is proposed to fully reap the storage and computation benefits of $BFlip$.
- Extensive evaluations on multiple state-of-the-art DNNs are conducted to analyze $BFlip$'s impact on accuracy.

This paper is organized as follows. Section II introduces the necessary background on ReRAM-based in-situ computation. Section III illustrates the overview of $BFlip$'s workflow, whose details are described in Section IV. Section V elaborates the architecture of the proposed accelerator. Experiment methodology and results are presented in Section VI and VII, respectively. Finally, Section VIII summarizes related works and conclusion is made in Section IX.

II. Background

A. ReRAM-Based In-Situ Computing

ReRAM stores information through the resistive switching effects. Fig. 1(a) illustrates the example of a ReRAM cell comprising two electrodes and a resistive switching layer sandwiched in between. When a SET or $RESET$ voltage is applied on the two electrodes, the oxygen vacancy filament in the resistive switching layer will be constructed or destroyed, thus changing the resistance of the cell.

ReRAM cells are organized as a crossbar structure to conduct MVM, as shown in Fig. 1(b). ReRAM cells are located at each cross point of wordlines and bitlines. The resistances of ReRAM cells are programmed according to the values in the matrix, and the wordline voltages are converted from the values of the input vector. For example, the cells on the first column are programmed to resistances R_1, R_2, ..., R_n, and the conductances (i.e. the reciprocal of the resistance) of these cells are G_1, G_2, ..., G_n. If voltages V_1, V_2, ..., V_n are applied on the wordlines, the total current flowing out from the bitline is $\sum_{i=0}^{n} V_i G_i$ based on Kirchoff's Law. When applied to DNN computing, the weight values are stored in the cells, and the input neurons are converted to wordline voltages.

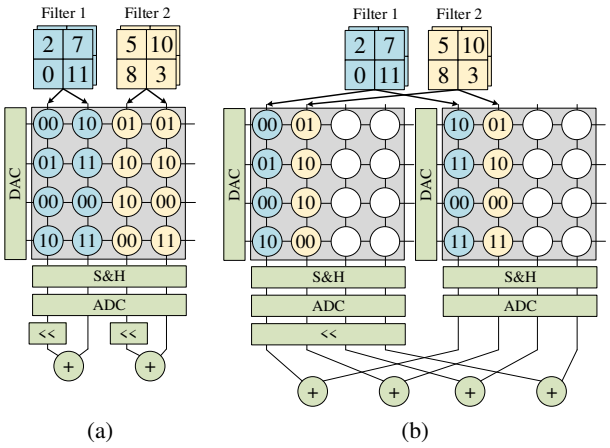


Fig. 2: Map DNN weights to crossbars.

Therefore, The output currents from the bitlines represent the output neurons.

B. Weight Mapping in ReRAM Crossbars

When applying different $WRITE$ voltages across a ReRAM cell, the cell's resistance can be divided into multiple regions. So one cell can store multiple bits. Recent work has demonstrated cells that can store up to seven bits. However, due to the complex control circuit and low precision of the cells, most ReRAM-based DNN accelerators adopt at most 2-bit cells [13], [14] to construct the crossbars. Some works even only use single-bit cells for better precision and inference accuracy [15].

Although there are extensive researches dedicated to quantizing DNN weights to shorter bit width, at least an 8-bit representation for each weight is still required to maintain high accuracy. Therefore, a weight value is spread onto multiple cells. For example, Fig. 2 shows how 4-bit weights are mapped on crossbars composed of 2-bit cells. Each weight is stored on two cells, one stores the two most significant bits while the other stores the two least significant bits. These two cells can reside on two columns of the same crossbar (Fig. 2(a)) or two different crossbars (Fig. 2(b)). $BFlip$ adopts the mapping in Fig. 2(b).

The ReRAM-based MVM is performed using analog signals (i.e. voltages, conductances, and currents). So DACs and ADCs are needed for conversion between the digital and analog domains. Input voltages are generated using DACs. The current signals flowing out from bitlines are first stored in the Sample and Hold (S&H) unit, and then converted to digital values in ADCs. The output from one bitline only represents a partial result, which needs to be combined with results from other bitlines to form the final output. For example in Fig. 2(a), the output of the first bitline needs to be left-shifted by two positions and added with the output of the second bitline to form the final result.

III. Overview

The workflow of $BFlip$ is illustrated in Fig. 3.

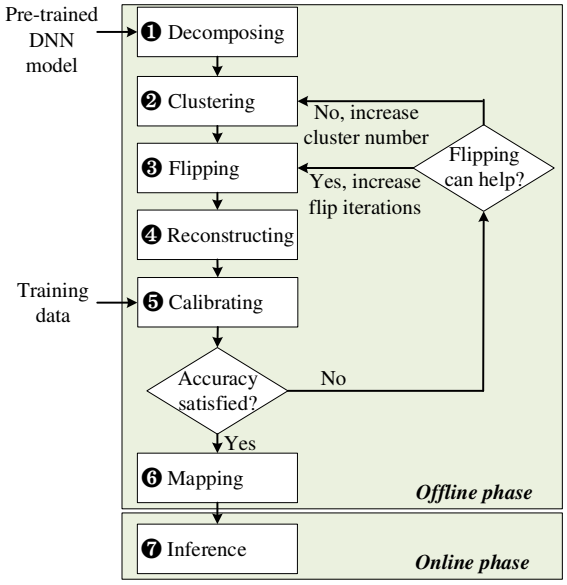


Fig. 3: The overview of $BFlip$'s workflow.

The whole workflow is composed of an offline phase to compress the DNN model and an online phase to use the compressed model for inference. Given a pre-trained DNN model, the $Decomposing$ step decomposes each layer's weight matrix into bit matrices. In the $Clustering$ step, the bit matrices are grouped into clusters. Each cluster has a centroid bit matrix computed by averaging all the bit matrices in the cluster. Within each cluster, the $Flipping$ step finds a combination of row and column flips (referred to as $metadata$) for each bit matrix to make its distance to the centroid bit matrix as close as possible. Next, in the $Reconstructing$ step, we reconstruct all the bit matrices by flipping the centroid bit matrix according to their metadata. In the $Calibrating$ step, we re-run the forward propagation using the reconstructed bit matrices on the training dataset to mitigate the accuracy loss. If the calibration generates a satisfying accuracy, the centroid bit matrices and the metadata will be mapped to physical crossbars in the $Mapping$ step and then enters the online phase for inference. If the accuracy is not satisfying, we first check whether it is still possible to improve the accuracy by applying more iterations in the $Flipping$ step. If so, we re-execute the $Flipping$ step with more iterations. Otherwise, we increase the number of clusters and go through the entire offline phase again until a satisfying accuracy is met.

IV. BFlip Details

In this section, we describe the details of $BFlip$ following the steps introduced in Section III.

A. Decomposing Weight Matrices

Given a pre-trained DNN model, the $Decomposing$ step decomposes each layer's weight matrix into bit matrices. For example, if a weight matrix uses 8-bit fixed-point format to represent its weight values, the weight matrix is decomposed into eight bit matrices. Each bit matrix contains all the bits that have the same significance in the weight values. Then, the bit matrices are partitioned into segments of size $(n-2m)\times(n-2m)$. n is the crossbar size (i.e. n rows and n columns) and m is the maximum number of bit matrices that could share the same crossbar. These parameters are provided by the designer before the offline phase starts. All subsequent operations are performed on the partitioned bit matrices.

B. Clustering Bit Matrices

The $Clustering$ step is to decide which bit matrices could share the same crossbar. We use Hamming Distance as the metric to measure the similarity between bit matrices. A small Hamming Distance between two bit matrices indicates that they are more likely to match each other after a small number of bit flips. We use Kmeans to group similar bit matrices into K clusters. K is a predefined parameter provided by the designer before the offline phase starts, but it may be changed in the following steps. Each cluster has a centroid bit matrix, such that the sum of the squared distances from all the bit matrices in the cluster to the centroid bit matrix is at the minimum. The centroid bit matrix is calculated by taking the average of all bit matrices in that cluster. The bit matrices that are in the same cluster will share the same crossbar. So K is limited by the available crossbars in the accelerator.

C. Flipping Bits in Bit Matrices

After clustering similar bit matrices, the next step is to minimize the distances from all the bit matrices in the cluster to the centroid bit matrix. A naive way is to find all the mismatched bits between each bit matrix and the centroid bit matrix, and only flip those bits to make them identical to the centroid bit matrix. However, this naive way will generate a large amount of metadata to record the positions of the flipped bits. Another problem is that MVM on a crossbar activates all the cells at the same time, flipping each bit individually breaks the regular access pattern and induces complex control overhead.

Therefore, we propose to flip the bits in the granularity of rows and columns. Each row and column only needs one bit to record whether it has been flipped or not. For a $t\times t$ bit matrix, there are only $2t$ bits metadata, which only has an overhead of $2/t$. We call the t-bit vector that records the flip status of each row the *row flip vector (RFV)*, and the other t-bit vector that records the flip status of each column the *column flip vector (CFV)*. To flip the bits in bit matrix A to match the centroid bit matrix C, we first calculate the difference between these two bit matrices by XORing them to get the bit matrix B, as shown in Fig. 4. In B, 1 means there is a mismatch between the corresponding bits in A and C. We define a score variable s for each row and column in B, indicating the number of mismatched bits. When flipping the bits in a row or column, the corresponding score s changes to $t-s$. Finding the combination of row and column flips on A to match C, is equivalent to finding the combination of row and column flips on B to minimize its total score. Unfortunately, finding the optimal flip pattern can be reduced to the *Shortest*

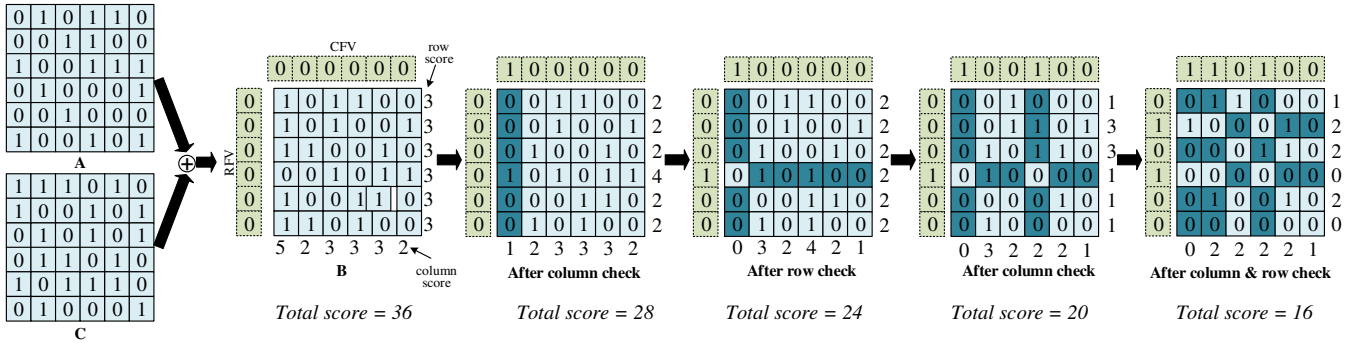


Fig. 4: Flip rows and columns to minimize the distance of two matrices. A is a bit matrix in the cluster. C is the centroid bit matrix of the cluster. B is calculated by XORing A and C.

Vector Problem (SVP) in the lattice, which is known to be NP-hard [16]. Therefore, We propose a greedy approach to find a flip pattern that could make the two bit matrices as close as possible.

We first calculate all the column scores by counting the number of 1s in each column of B, if any of the scores is larger than $t/2$, then we flip this column and flip the bit in CFV to record this column has been flipped. Then we calculate all the row scores and perform the same operation on the rows and RFV as we did for the columns. After the row check is done, it is possible some column scores that are previously less than $t/2$ now become larger than $t/2$. For example, the score of the forth column in Fig. 4 turns from 3 to 4 after the row check. So we iteratively perform the column check and row check until all the scores are less than $t/2$.

Even though all the scores are less than $t/2$ after the column and row checks, the total score can still be reduced if one column and one row are flipped at the same time. For example, after the second column check in Fig. 4, all the column scores and row scores are already less than $t/2$, and the total score is 20. If we flip the second column and the second row simultaneously, the total score can be further reduced to 16. Actually, in addition to flipping one column and one row simultaneously, we could also flip multiple columns and/or multiple rows simultaneously to further reduce the total score. In general, given a set of columns (denoted as C) and a set of rows (denoted as R), flipping all these columns and rows simultaneously could reduce the total score if the following equation is satisfied:

$$\sum_{c \in C} s_c + \sum_{r \in R} s_r > \frac{(|R| + |C|)t - 2t_0 + 2t_1}{2} \quad (1)$$

where s_c and s_r are the column score and row score respectively, $|C|$ is the number of columns in set C, $|R|$ is the number of rows in set R, t is the bit matrix size, t_0 is the number of intersection cells whose value is 0, t_1 is the number of intersection cells whose value is 1. The whole bit flipping algorithm is shown in Algorithm 1.

D. Reconstructing Bit Matrices

The metadata (CFVs and RFVs) produced in the previous step records the information of how to minimize the distance of each bit matrix in the cluster to the centroid bit matrix.

Algorithm 1 Bit Flipping

N_C: maximum number of columns that flip simultaneously
N_R: maximum number of rows that flip simultaneously
t: $t \times t$ bit matrix size

```
Flipped = True
while Flipped do
    Flipped = False
    if Has columns whose score is larger than t/2 then
        Flipped = True
        Flip those columns
    end if
    if Has rows whose score is larger than t/2 then
        Flipped = True
        Flip those rows
    end if
    if Flipped == False then
        for i = 1 to N_C, j = 1 to N_R do
            Choose i columns to form set C
            Choose j rows to form set R
            if C and R satisfy eq. (1) then
                Flipped = True
                Flip these rows & columns simultaneously
            end if
        end for
    end if
end while
```

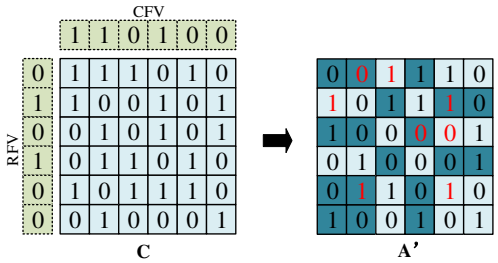


Fig. 5: Reconstruct A' from C.

So, we could either apply the flips on each bit matrix in the cluster to make it close to the centroid bit matrix, or we could apply the flips on the centroid bit matrix to make it more close to each bit matrix in the cluster. The *Reconstructing*

TABLE I: BN's impact on accuracy.

Network	baseline	no BN update	after BN update
ResNet50	76.13%	71.628%	74.508%
VGG16	71.592%	58.214%	-
VGG16-BN	73.360%	68.966%	70.732%

step performs the latter to reconstruct each bit matrix in the cluster from the centroid bit matrix. Fig. 5 shows how to use the metadata generated in the previous step to reconstruct A from C. The reconstructed bit matrix is denoted as A'. The cells with dark background indicate the flipped cells. Because the $Flipping$ step does not guarantee a flip pattern to make A and C identical, the reconstructed bit matrix A' is only an approximation of A. The mismatched cells between A and A' are marked with red numbers. In the same way, we could reconstruct an approximation of every original bit matrix in the cluster from the centroid bit matrix. As a result, we could get a new modified DNN model consists of all the reconstructed bit matrices.

E. Calibrating the Modified DNN

Most recent DNN models contain Batch Normalization (BN) layers to accelerate the training process and improve the accuracy via a regularization effect. BN layers standardize the inputs to DNN layers, i.e. making the each layer's inputs follow a standard distribution. However, calculating the actual mean and standard deviation of each layer's input during inference incurs a large computation overhead. So BN layers use the statistics (mean and standard deviation) collected from the training data to standardize the inputs in the inference phase, based on the assumption that the training data has the same distribution of the whole data seen by the DNN in real applications. Because the modified DNN model produced in the previous step introduces noises to the weights which alters the distribution of the input to the next layer, the above assumption can no longer be held true. As shown in Tab. I, there is an accuracy drop from 76.13% to 71.628% for ResNet50 if we directly use the modified DNN model in the inference phase. We tackle this problem by updating the distribution statistics in BN layers. One thing to note here is that we only update the distribution statistics of BN layers, instead of the trainable parameters (i.e. the gamma weights and beta weights) of the BN layers. Because distribution statistics are collected during the forward propagation, we only need to re-run the forward propagation phase on the training data without the need for backward propagation and weight update. The last column in Tab. I shows the accuracy after updating the BN statistics.

For networks that do not have BN layers, we add BN layers after each convolutional layer and fully connected layer to mitigate the impact of weight noises intruduced in previous steps. Tab. I also shows the accuracy of the VGG16 network with and without BN layers. Adding BN layers not only increases the baseline accuracy, but also effectively mitigates the accuracy loss caused by mismatched bits introduced in previous steps.

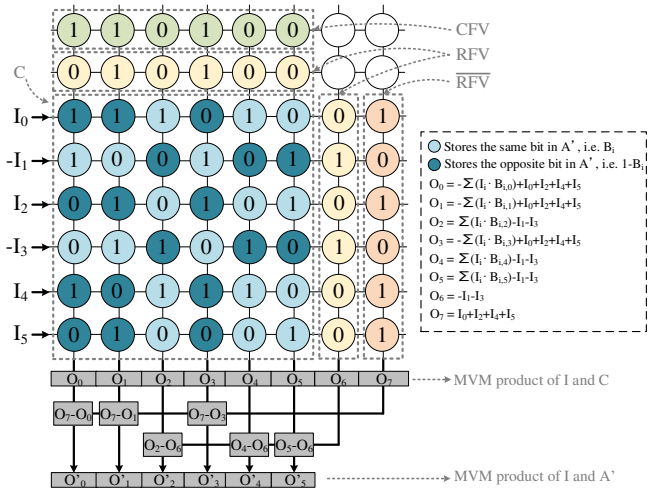


Fig. 6: Map centroid bit matrix and meta data to crossbar.

F. Mapping to Crossbars

All the bit matrices in the same cluster can be constructed from the centroid bit matrix and the metadata. So they can share the same crossbar, and we only need to store the centroid bit matrix and the metadata. If a cluster has more than m matrices (m is the maximum number of matrices that can share the crossbar, defined in the $Decomposing$ step), the cluster is divided into sub-clusters with size not greater than m. Each sub-cluster will map to a separate crossbar. In the $Decomposing$ step, the bit matrices have been cut into sizes of $(n-2m)\times(n-2m)$, so the size of the centroid bit matrix is also $(n-2m)\times(n-2m)$. In addition to storing the centroid bit matrix on the crossbar, the remaining $2m$ rows and $2m$ columns of the crossbar store the metadata of each bit matrix in the sub-cluster. For each bit matrix, we use two rows to store its CFV and RFV, and two columns to store its RFV and the negation of RFV (i.e. $\overline{RFV}$). Fig. 6 shows an example of mapping the centroid bit matrix C and the metadta of A in Fig. 4 to a crossbar. If there are other bit matrices that also share this crossbar, each bit matrix needs two more rows to store its CFV and RFV and two more columns to store its RFV and $\overline{RFV}$.

G. Inference

In Fig. 6, because the crossbar only stores the centroid bit matrix, the output from the crossbar (i.e. O_0, O_1, ..., O_5) is the MVM product between input I and centroid bit matrix C. Note that when performing MVM in the crossbar, the top rows which store the metadata do not participate in the computation by applying a zero voltage on their wordlines. In order to get the MVM product between I and A', additional steps are required to adjust the O_is. The dark cells in Fig. 6 indicates the different cells between C and A'. If we denote the bits in A' as B_i, the dark cells stores $1-B_i$. If the corresponding bit in RFV is 1, we apply the opposite value of the input to on the wordline. As a result, for O_0, O_2, O_4, and O_5, whose bit in CFV is 1, the output is $-\sum(IB)+I_0+I_2+I_4+I_5$. We can adjust these outputs by subtracting them from the output of O_7. And for O_1 and O_3, whose bit in CFV is 0, the output

is $\sum(IB) - I_1 - I_3$. We can adjust these outputs by adding the output of O_6. The adjusted outputs (i.e. O'_0, O'_1, ..., O'_5) are the MVM product between I and A'.

V. Architecture

In this section, we present our $BFlip$ accelerator. The accelerator is integrated into the system via PCIe bus and works as a slave to process DNN tasks received from the CPU. The top-level structure of the $BFlip$ accelerator follows the generic ReRAM-based DNN accelerator design, which consists of multiple tiles as shown in Fig. 7. The tiles are connected using an on-chip concentrated mesh. The right of Fig. 7 shows the details of one such tile. Each tile has an eDRAM buffer to store inputs of DNN layers, an output buffer for output aggregation, a Shift and Add unit to form the full production results from bit slices, a Pooling unit for pooling layers, and an Activation unit that implements non-linear activation functions. The computation is distributed to multiple Processing Engines (PEs). Each PE has a set of ReRAM crossbars to perform the MVM computation. Each crossbar is partitioned into four areas – the rows at the top store the CFVs (shown in green) and RFVs (shown in yellow), the right columns store the RFVs (shown in yellow) and $\overline{RFV}$s (shown in pink), and the left bottom part to store the centroid bit matrix (shown in blue). Both the inputs and the weights are in two's complement format.

A MVM computation for one bit matrix consists of three steps. In the first step, the RFV vector and the CFV vector are read from the top rows from the crossbar. The CFV vector is stored in $Buffer_1$ of the Reconstruct unit. The RFV vector is loaded into the RFV buffer in the wordline driver. In the second step, the inputs are converted to their opposite values according to the bits in the RFV buffer. Then, the inputs compute with the centroid bit matrix and the right-most columns which store the RFVs and $\overline{RFV}$s. The results of the centroid bit matrix are stored in $Buffer_2$ of the Reconstruct unit. The results of the RFVs and $\overline{RFV}$s are stored in $Buffer_3$ of the Reconstruct unit. In the third step, the ALU adjusts the results in the three buffers to get the MVM product between the input and the reconstructed bit matrix.

VI. Methodology

A. Accuracy Models

We evaluate four datasets: MNIST [19], SVHN [17], Cifar10 [18] and ImageNet [20]. We construct custom neural network structures for SVHN and Cifar10. The details of these structures are listed in Tab. II. For MNIST, we run it on LeNet-5 [21]. For ImageNet, we test it on five different networks – VGG16 [4], ResNet50 [22], ResNeXt50 [23], GoogLeNet [24] and DenseNet [25]. All these five networks are popular ones in the machine learning community.

B. Hardware Models

We implement a custom cycle-accurate simulator for the $BFlip$ accelerator. The hardware parameters are listed in Tab. III. Most of the parameter values are adopted from $ISAAC$ [13]. For components that are new in the $BFlip$ accelerator, we use CACTI [26] and NVSIM [27] to model SRAM buffers and ReRAM crossbars, respectively. Both SRAM and logic use 32nm process. Crossbar arrays use single-bit ReRAM cells, which need 4.4ns for read and 52.2ns for write. The DAC's resolution is one bit and the ADC's resolution is seven bits. We use the same ADC design as in $ISAAC$, and scale the power consumption reported in $ISAAC$ to seven bits using the equation in [28].

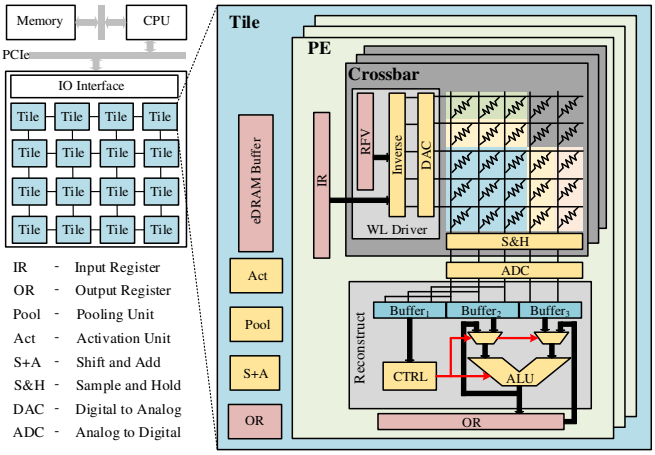


Fig. 7: $BFlip$ accelerator architecture.

TABLE II: Benchmarks.

Dataset	Network
MNIST	LeNet-5
SVHN	conv3x32-conv3x32-pool-conv3x64-conv3x64-pool-conv3x128-conv3x128-pool-1024-512-10
Cifar10	conv3x128-conv3x128-conv3x128-pool-conv3x256-conv3x256-conv3x256-pool-conv3x512-conv3x512-conv3x512-pool-1024-10
ImageNet	VGG16, ResNet50, ResNeXt50, GoogLeNet, DenseNet

We compare $BFlip$'s performance and energy consumption with two existing works – $ISAAC$ [13] and SRE [14]. $ISAAC$ is the most widely used baseline in ReRAM-based DNN accelerator designs. The performance and energy comparison results with $ISAAC$ can be conveniently scaled to compare with other ReRAM-based DNN accelerators. SRE is one of the state-of-the-arts that exploit model sparsity in ReRAM-based DNN accelerators. Both $ISAAC$ and SRE use two-bit cells. We modify these two baselines to use single-cells for a fair comparison with $BFlip$.

VII. Results

In this section, we divide the evaluation of $BFlip$ into two parts. In the first part, we first evaluate $BFlip$'s impact on accuracy with different ms (i.e., the number of bit matrices that share the same crossbar). In terms of compression ratio, changing m in $BFlip$ has the same effect as using different bit widths in quantization. Tab. IV shows the equivalent quantized bit width for different ms, assuming crossbar size

TABLE III: BFlip accelerator parameters.

Unit	Spec	Power
Resconstruct Unit		
Buffer	size: 48B	0.07mW
ALU	num: 1	0.1mW
PE		
ADC	num: 8; resolution: 7bits	9mW
DAC	num: 8×128; resolution: 1bit	4mW
S+H	num: 8×128	10uW
Xbar array	num: 8; size: 128×128; cell bits: 1	1.8mW
RFV buffer	size: 16B	0.028mW
Reverse	num: 1	0.01mW
Reconstruct	num: 4	0.64mW
IR	size: 2KB	1.24mW
OR	size: 256B	0.23mW
PE Total	num: 1	17
Tile		
PE	num: 12	204mW
eDRAM	size: 64KB; banks: 2; width: 256	20.7mW
Bus	wires: 384	7mW
Router	flit size: 32; ports: 8	42mW
Activation	num: 2	0.52mW
S+A	num: 1	0.05mW
Maxpool	num: 1	0.4mW
OR	size: 3KB	1.68mW
Tile Total	num: 1	276mW
Chip		
Tile	num: 168	46.3W
Hyper Tr	links: 4; freq: 1.6GHz	10.4W
Chip Total	num: 1	56.7W

is 128×128. In the second part, we analyze $BFlip$'s benefits of performance improvement and energy reduction.

A. Accuracy

Fig. 8 shows the accuracy results of $BFlip$, the full-precision baseline, and the conventional quantization method. The full-precision baseline is obtained from the PyTorch torchvision package. For quantization, eight bits are sufficient for all benchmarks to keep the accuracy the same as the full-precision baseline. However, further reducing the bit width will significantly degrade the accuracy. Small datasets can sustain a more aggressive quantization. For example, there is an accuracy loss of only 0.09%, 0.16%, and 0.3% for MNIST, SVHN, and Cifar10 when their weights are quantized to four bits. However, DNN models for ImageNet need at least seven bits to prevent a massive loss of accuracy. On the other hand, we observe little accuracy loss for $BFlip$ even when $m = 9$ (equivalent to 1-bit quantization). Binarized neural networks (BNNs) have gained a lot of attention as each weight value only needs one bit. BNNs are different from convention 1-bit quantization in that many optimizations are made in either the quantization method or the training process. The accuracy gap between BNNs and the full-precision model is getting smaller in recent years. Fig. 8 also shows the accuracy of recent state-of-the-art results of BNNs for each model. For ImageNet, there is still a 6.26% to 22.6% accuracy gap between BNN and the full-precision baseline. However, the accuracy loss of $BFlip$ with $m = 9$ (equivalent to 1-bit quantization) is still negligible (2.18% on average).

TABLE IV: Equivalent quantized bit width for different m.

Quant. bit width	4	3	2	1
m for BFlip	2	3	5	9

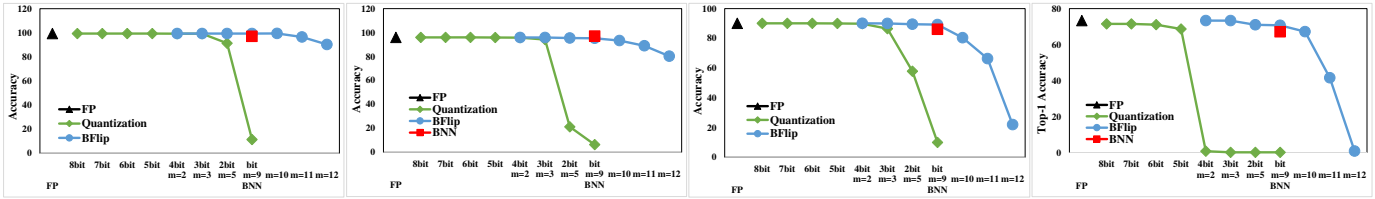
B. Performance and Energy

We select $m = 2$ and 9 for $BFlip$ as examples to compare the performance speedup with the baselines. We select these two configurations because $m = 2$ is equivalent to quantizing weight bit width to four bits and the accuracy loss is negligible for small datasets, while $m = 9$ is equivalent to BNNs which have been wildly studied in previous works. Fig. 9 shows the performance speedup of $BFlip$ over the baselines. All the results are normalized to $ISAAC$. $BFlip$'s performance speedup ranges from 1.26× to 2.15× for $m = 2$ and 1.6× to 3.8× for $m = 9$, and the average speedup is 1.58× and 2.45×, respectively. The performance speedup comes from reusing the crossbar outputs, as the crossbar-based MVM is the most time-consuming operation. Using larger m also helps to reduce the computation bottleneck, because only a smaller region inside the crossbar needs to be accessed simultaneously for computation, which needs significantly less access time than activating the whole crossbar. For example, when $m = 9$, one crossbar computation result can be used to generate the MVM product of nine bit matrices, and only a 110×128 sub-region inside the crossbar needs to participate in the MVM computation. Because the computation saving in SRE depends on the sparsity of the pruned model, its performance is worse than $ISAAC$ for dense models (Cifar10 and GoogLeNet).

Fig. 10 shows the energy consumption of $BFlip$ when $m = 2$ and 9 normalized to $ISAAC$. SRE reduces energy consumption by skipping all-zero rows in OUs (Operation Unit) – a fine-grain sub-region inside a crossbar. However, because the inputs need to be reordered due to the irregular access pattern in the crossbar, it demands additional eDRAM accesses which incur a large energy overhead. So, it can save more energy consumption on models with structural sparsity (for example, 90% energy save for VGG16). $BFlip$ only needs to convert the inputs into their opposite values, the order is not changed, so no additional eDRAM access is required. The energy saving of $BFlip$ does not depend on sparsity structure after pruning, it saves 75% and 85% energy on average for $m = 2$ and 9, respectively.

VIII. Related Work

ReRAM has been extensively studied to design PIM accelerators [13], [15]. As model compression techniques (such as pruning and quantization) are widely adopted in ASIC accelerators, more recent works start to focus on designing compression-friendly accelerators based on ReRAM. [29] proposes a structural pruning algorithm to find more all-zero rows and columns by regularizing the distribution of zero weights. [30] uses a fine-grained column compression to exploit the



(a) MNIST (b) SVHN (c) Cifar10 (d) VGG16

(e) ResNet50 (f) ResNeXt50 (g) GoogLeNet (h) DenseNet

Fig. 8: Accuracy of the full-precision baseline, conventional quantization, BNN and $BFlip$.

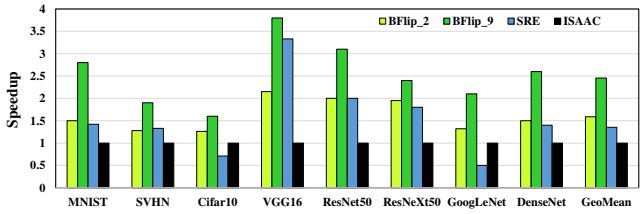


Fig. 9: Speedup of $BFlip$ over baselines.

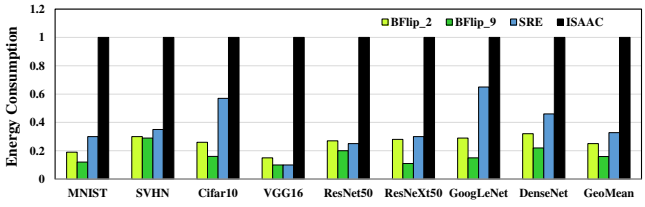


Fig. 10: Energy consumption of $BFlip$ over baselines.

sparsity in ReRAM crossbars. However, it suffers from a high overhead of output indexing. [14] exploits the sparsity by activating a sub-region of a crossbar, such that more all-zero rows or columns can be found in the sub-regions. It suffers from a high overhead of reordering the inputs.

IX. Conclusion

We propose $BFlip$ to share a crossbar by multiple bit matrices. $BFlip$ provides a way to balance between accuracy and model size. Our evaluation shows that $BFlip$ achieves significantly higher accuracy compared to the conventional quantization method and binarized neural networks.

References

[1] A. Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.

[2] A. Senior *et al.*, "Deep neural networks for acoustic modeling in speech recognition," in *IEEE Signal Processing Magazine*, 2012

[3] B. Hu *et al.*, "Convolutional neural network architectures for matching natural language sentences," in *NIPS*, 2014.

[4] K. Simonyan *et al.*, "Very deep convolutional networks for large-scale image recognition," in *ICLR*, 2015

[5] Q. Xie *et al.*, "Self-training with noisy student improves imagenet classification," in *CVPR*, 2020

[6] H. Wong *et al.*, "Metal–oxide RRAM," in *Proceedings of the IEEE*, 2012.

[7] A. Vincent *et al.*, "Spin-transfer torque magnetic memory as a stochastic memristive synapse," in *ISCAS*, 2014.

[8] G. Burr *et al.*, "Experimental demonstration and tolerancing of a large-scale neural network (165 000 synapses) using phase-change memory as the synaptic weight element,", in *T-ED*, 2015.

[9] P. Molchanov *et al.*, "Pruning convolutional neural networks for resource efficient inference," in *ICLR*, 2017.

[10] I. Hubara *et al.*, "Quantized neural networks: Training neural networks with low precision weights and activations," in *JMLR*, 2017.

[11] B. Jacob *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *CVPR*, 2018.

[12] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *arXiv:1806.08342*, 2018.

[13] A. Shafiee *et al.*, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *ISCA*, 2016.

[14] T. Yang *et al.*, "Sparse reram engine: Joint exploration of activation and weight sparsity in compressed neural networks," in *ISCA*, 2019.

[15] M. Bojnordi *et al.*, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *HPCA*, 2016.

[16] M. Ajtai, "The shortest vector problem in L2 is NP-hard for randomized reductions," in *STOC*, 1998.

[17] Y. Netzer *et al.*, "Reading digits in natural images with unsupervised feature learning," in *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[18] A. Krizhevsky *et al.*, "Learning multiple layers of features from tiny images," *technical report*, 2009.

[19] Y. LeCun *et al.*, "The mnist database of handwritten digits," http://yann.lecun.com/exdb/mnist/, 2011.

[20] J. Deng *et al.*, "Imagenet: A large-scale hierarchical image database," in *CVPR*, 2009.

[21] Y. LeCun *et al.*, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, 1998.

[22] K. He *et al.*, "Deep residual learning for image recognition," in *CVPR*, 2016.

[23] S. Xie *et al.*, "Aggregated residual transformations for deep neural networks," in *CVPR*, 2017.

[24] C. Szegedy *et al.*, "Going deeper with convolutions," in *CVPR*, 2015.

[25] G. Huang *et al.*, "Densely connected convolutional networks," in *CVPR*, 2017.

[26] N. Muralimanohar *et al.*, "CACTI 6.0: A tool to model large caches," *Technical Report*, 2009.

[27] X. Dong *et al.*, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *TCAD*, 2012.

[28] M. Saberi *et al.*, "Analysis of power consumption and linearity in capacitive digital-to-analog converters used in successive approximation ADCs," *TCAS1*, 2011.

[29] W. Wen *et al.*, "Learning structured sparsity in deep neural networks," in *NIPS*, 2016.

[30] P. Wang *et al.*, "Snrram: an efficient sparse neural network computation architecture based on resistive random-access memory," in *DAC*, 2018.