A DNN Protection Solution for PIM accelerators with Model Compression

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

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

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

***Abstract*—Deep Neural Network (DNN) is a data-hungry algorithm, which has a large energy cost on moving data between the memory and the computating unit. Recent works have proposed using ReRAM to design Process-In-Memory (PIM) accelerators to perform the computation inside the memory. However, the IP protection of the DNNs deployed on such accelerators is an important topic that has been less addressed. Firstly, due to its non-volatility, ReRAM does not need a continuous power supply to retain data. This makes the accelerator susceptible to new security vulnerabilities, for example, accessibility to the stored model if a device gets stolen. Secondly, because ReRAM's crossbar structure can only compute on cleartext data, encrypting the ReRAM content is no longer a feasible solution in this scenario.**

In this paper, we propose an IP protection solution on ReRAM based DNN accelerators to store DNN weights on crossbars in an encrypted format while still maintaining ReRAM's in-memory computing capability. The proposed solution stores and computes the DNNs in Stochastic Computing (SC) format, which can easily hide its conveyed weight values by scrambling the bit stream segments. However, SC's long bit streams incur a large storage overhead. To tackle this problem, we also propose two techniques to share bits among multiple weights, effectively compressed DNN's model size to reduce storage overhead.

***Index Terms*—ReRAM, security, DNN, compression**

I. Introduction

As non-volatile PIM accelerators could provide higher computation parallelism and avoid massive data movement between the memory and computing unit, ReRAM is a good fit for DNN computing. However, how to protect the DNN IPs after their deployment on ReRAM accelerators remains a less addressed problem. Compared to CMOS accelerators, new challenges arise: i) ReRAM's non-volatility retains data after power is off, making the stored weights easier to be extracted out from the accelerator; ii) ReRAM's crossbar structure can only compute on cleartext data, making encryption on weights impossible.

To address the above problems, we utilize the new 1T4R structure [1] together with Stochastic Computing (SC) to protect DNN weights on ReRAM crossbars. The multiplication is performed in the SC format, i.e., all weights and inputs are represented in the form of bit streams. With the 1T4R structure, the long bits stream of each weight can be truncated into multiple segments. During run time, the segments will be dynamically combined to reconstruct the whole bit stream for computing. Therefore, even if the adversary extracts all the content from the crossbars, the actual reconstructed bit streams are still hidden without knowing the recombine pattern. Only the recombine pattern needs to be stored as confidential information. Using these shuffled segments for protection also avoids the need for conventional encryption, and still maintains the PIM capability of ReRAM.

However, representing a weight in SC format induces an exponential storage overhead. For example, an n-bit binary number needs a bit stream of 2^n bits in SC format to keep the same precision. To solve this problem, we propose two ReRAM-compatible model compression schemes at different granularity levels. Our Bit-Level Sharing (BLS) scheme reuses the same set of segments to reconstruct multiple weight bit streams. Our Crossbar-Level Sharing (CLS) scheme flips the bit matrices of different crossbars to reduce their distance, as a result, we only need to store the bit matrix of one crossbar together with the flipping pattern of the other bit matrices.

II. Background

A. 1T4R Crossbar Structure

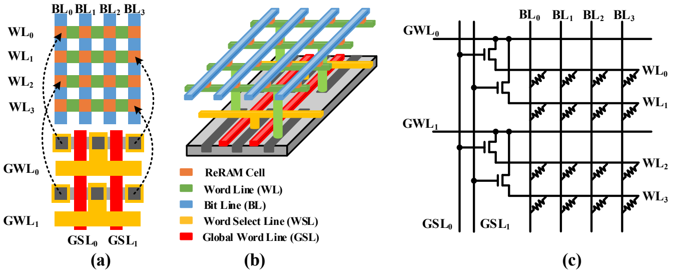
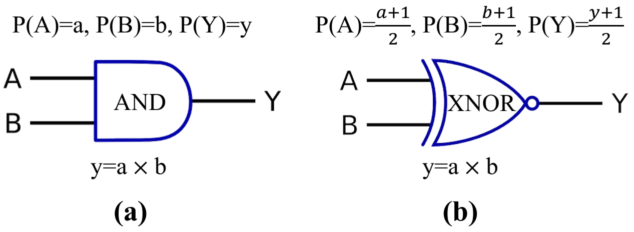


Fig. 1: 1T4R structure.

ReRAM's crossbar structure is a perfect fit for DNN's matrix-vector multiplication (MVM). However, because there are no access transistors to isolate adjacent cells in the crossbar, unactivated word lines or bit lines can induce leakage current (called *sneak current*) during access. Recently, [1] presented a 1T4R ReRAM crossbar structure to address the sneak current problem. In a 1T4R crossbar, each 4×4 sub-array is isolated from other cells in the crossbar by 4 dedicated transistors, which are fabricated underneath the sub-array. Fig. 1(a) and (b) illustrate the 2D and 3D layout, respectively. [1] fabricated a 1T4R test chip to show that the 4 transistors can be fully hidden underneath the area of the 4×4 ReRAM sub-array, thus still achieving the $4F^2$ cell size. Fig. 1(c) shows the schematic diagram of one 4×4 sub-array. Besides bit


P(A)=a, P(B)=b, P(Y)=y
A
B
AND
Y
y=a × b
P(A)=$\frac{a+1}{2}$, P(B)=$\frac{b+1}{2}$, P(Y)=$\frac{y+1}{2}$
A
B
XNOR
Y
y=a × b

(a) **(b)**

Fig. 2: Stochastic Computing Implementation.

Stochastic Computing (SC) represents a number by using a random bit steam, in which the probability of the appearance of 1s indicates the represented value. For example, the bit stream $X = 011001001$ represents 0.4 because the probability $P(X = 1) = 0.4$. Since values are represented by probabilities, the representation for a value is not unique. A multiplication between two SC numbers can be implemented using bit-wise AND operations, as shown in Fig. 2(a). However, only values in the range $[0, 1]$ can be encoded using the above method (called unipolar format). The bipolar format can represent values in the range $[-1, 1]$ by scaling it in $[0, 1]$. For example, the value $s = -0.4 \in [-1, 1]$ is first scaled to $t = (s + 1)/2 = 0.3 \in [0, 1]$, then encoded into $P(T = 1) = 00101010$. The multiplications for bipolar SC numbers are implemented using $XNOR$ gates (Fig. 2(b)).

III. Securing DNN Weights

To protect the DNN, both weights and inputs are computed in SC format. We use the *unipolar format* in SC because the XNOR operation in *bipolar format* is not applicable to crossbar implementation. Fortunately, the ReLU activation in the DNNs filters out all negative values in each layer's outputs (i.e., inputs to the next layer), only weights may have negative values. Therefore, we only store the absolute values of the weights in the *unipolar format* SC bit streams while the sign bits are stored is a separate crossbar (called the sign bit crossbar in this paper). It is possible to expose the sign bits to the adversary, but only knowing the sign bits without the actual absolute weight values can not generate useful inference results.

To hide the bit streams stored in the crossbar from adversary, we slice the bit stream into segments. We adopt the rotated layout of the original 1T4R structure to switch the functionality of the bit lines and word lines. Since the crossbar is a symmetrical structure and the ReRAM cells are acting as resistors during computation, the rotated layout does not need any changes inside the crossbars, only the positions of the word line drivers and the ADCs are exchanged. As shown in Fig. 3, the ADCs are placed at the right of the crossbar while the word line drivers are at the bottom. In this new layout, we rename the original horizontal global word lines ($GWLs$) to global bit lines ($GBLs$), and the original bit lines (BLs) are called word lines (WLs).

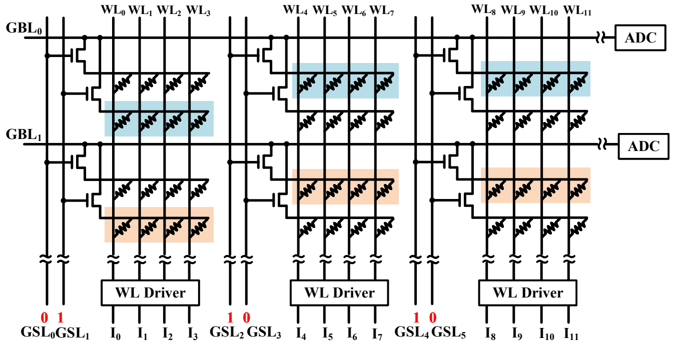


Fig. 3: The rotated layout of the 1T4R structure.

In this new layout, each row in the crossbar is now divided into segments consisting of 4 cells. The $GSLs$ control which segments are connected to the $GBLs$ during computation. For the example shown in Fig. 3, the first pair of $GSLs$ selects the segments on the second and fourth rows to connect to the $GBLs$, the second pair of $GSLs$ selects the segments on the first and third rows to connect to the $GBLs$, etc. As a result, all the blue cells in Fig. 3 are merged to form a logical bit line to compute with the inputs I, and the results are accumulated on GBL_0. Similarly, all the orange cells in Fig. 3 are merged to form another logical bit line to compute with the inputs I, and the results are accumulated on GBL_1. We restrict each pair of $GSLs$ can only be 01 or 10, such that either the segments on the even rows or the segments on the odd rows will be connected to the $GBLs$. Therefore, only one bit in each GSL pair needs to be stored. We use a bit vector to record the first bit in each GSL pair (i.e., GSL_0, GSL_2, ...). This bit vector is referred to as the *GSL vector* in this paper. The uncolored cells in the figure can form another two logical bit lines when all the $GSLs$ negate their inputs. By keeping the GSL vector secret, the actual content of the logical bit lines is stored in the crossbar in an encrypted form.

IV. Bitline-level Sharing

For a 8-bit binary value, we need a 256-bit stream in SC format to preserve the same precision. For a 256×256 crossbar and a GSL vector, there are 256 logical bit lines, each consisting of 256 bits. It is straightforward to map 256 weights onto the crossbar by storing one weight on each logical bit line. However, this incurs a 32× storage overhead compared to the original 8-bit binary format. In this section, we propose a sharing scheme in the bitline level to store more than 256 weights on one crossbar to reduce the storage overhead.

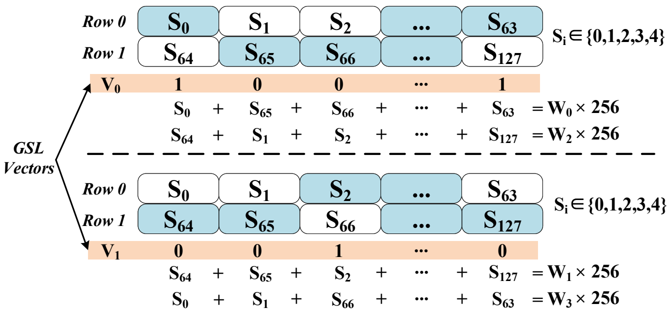


Fig. 4: Mapping 4 weights on two rows in the crossbar.

We use multiple GSL vectors. Each GSL vector maps a different set of 256 weights on the crossbar. For n GSL vectors, a crossbar can store $256n$ weights, with every pair of adjacent physical bit lines constructing $2n$ logical bit line to $2n$ weights. Fig. 4 shows an example of mapping 4 weights on the first two physical bit lines in the crossbar assuming $n = 2$. Each square in the figure represents a segment. S_i indicates the number of 1s in each segment. Since a segment has 4 cells, S_i can only be 0, 1, 2, 3, or 4. V_0 and V_1 are the two GSL vectors. The first weight W_0 is mapped to the logical bit line determined by V_0, i.e, S_0, S_{65}, S_{66}, ..., S_{63}. The second weight W_1 is mapped to the logical bit line determined by V_1, i.e, S_{64}, S_{65}, S_2, ..., S_{127}. By using the negation of V_0 and V_1, we can get another two logical bit lines using the remaining segments to store W_2 and W_3. The left-hand side of the equations in Fig. 4 represents the total number of 1s in each logical bit line. In order to correctly represent W_i, the total number of 1s in the logical bit lines needs to be $W_i \times 256$, which is the right-hand side of the equations. The mapping process is to determine the S_is to satisfy those equations. However, when n increases, it becomes hard to make all the equations to be true. We formulate this problem as a mixed-integer linear programming (MIP) model that minimizes the difference between the left-hand sides and the right-hand sides. In general, the MIP model for mapping $2n$ weights on the two adjacent physical bit lines can be written as follows:

Minimize

$$\overset{2n}{\underset{i=0}{}} D_i$$

Subject to:

$$S_j \in \qquad (0 \quad j \quad 127)$$

$$S_j \quad 0 \qquad (0 \quad j \quad 127)$$

$$S_j \quad 4 \qquad (0 \quad j \quad 127)$$

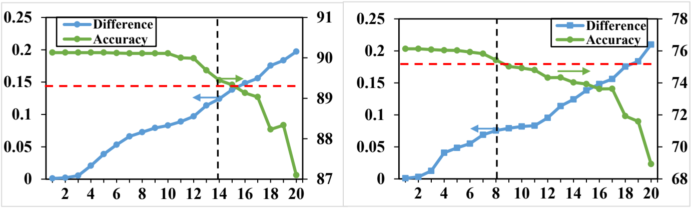
$$D_i \quad (\overset{63}{\underset{j=0}{}} V_{i,j} \quad S_j + \overset{63}{\underset{j=0}{}} \overline{V_{i,j}} \quad S_{j+64}) - W_i \times 256 \qquad (0 \quad i \quad n\text{-}1)$$

$$D_i \quad W_i \times 256 - (\overset{63}{\underset{j=0}{}} V_{i,j} \quad S_j + \overset{63}{\underset{j=0}{}} \overline{V_{i,j}} \quad S_{j+64}) \qquad (0 \quad i \quad n\text{-}1)$$

$$D_i \quad (\overset{63}{\underset{j=0}{}} \overline{V_{i,j}} \quad S_j + \overset{63}{\underset{j=0}{}} V_{i,j} \quad S_{j+64}) - W_i \times 256 \qquad (n \quad i \quad 2n\text{-}1)$$

$$D_i \quad W_i \times 256 - (\overset{63}{\underset{j=0}{}} \overline{V_{i,j}} \quad S_j + \overset{63}{\underset{j=0}{}} V_{i,j} \quad S_{j+64}) \qquad (n \quad i \quad 2n\text{-}1)$$

D_i represents the difference between the two sides of the equations. The first three constraints restrict the number of 1s in each segment to be integers between 0 and 4. Since linear programming can not use absolute values as constraints, two inequalities are used to represent the absolute difference. The next two constraints indicate the difference when mapping weights on logical bit lines determined by V_is. The last two constraints indicate the difference when mapping weights on logical bit lines determined by $\overline{V_i}$s.



(a) `Cifar10` (b) `ResNet50`

Fig. 5: Tolerance of *n* for `Cifar10` and `ResNet50`.

When n becomes larger, it is harder to minimize the target. Fig. 5 shows the mappings of `Cifar10` and `ResNet50` (structure details are listed in Table II) when n increases. The blue lines illustrate the average difference between the left-hand sides and right-hand sides after optimization, while the green lines show the inference accuracy. From the figure, we can see that different DNNs have different tolerance on n. If we set the accuracy loss budget to 1% (marked by the red dashed line), n can be set to 14 in `Cifar10` while n can not exceed 8 for `ResNet50` which is also the worst case in all our tested DNNs. It is possible to use larger ns for different DNNs to achieve more storage savings, however, we conservatively set n to 8 for all DNNs for simplicity in evaluation.

V. Crossbar-level Sharing

Fig. 6 shows the workflow of CLS. The whole workflow is composed of an offline phase to compress the DNN and an online phase to use the compressed DNN for inference. Next, each step will be described in details.

Decomposing. This step takes a pre-trained DNN after BLS, and 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.

Clustering. This 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. We use Kmeans to group similar bit matrices into clusters. 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.

Flipping. This step is to minimize the distances from all the bit matrices in the cluster to the centroid bit matrix. We

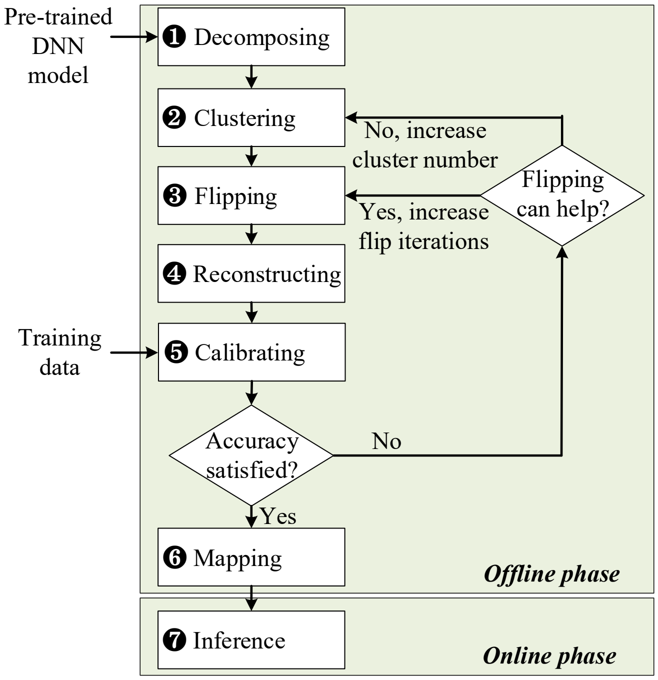


Fig. 6: The overview of crossbar-level sharing workflow.

flip the bits in the granularity of rows and columns. Each row or column only needs one bit to record whether it has been flipped or not. We call the bit vector that records the flip status of each row the *row flip vector (RFV)*, and the other bit vector that records the flip status of each column the *column flip vector (CFV)*. We first calculate the difference between these two bit matrices by XORing them to get the bit matrix B, as shown in Fig. 7. In B, 1 means there is a mismatch between the corresponding bits in A and C. We define a score variable for each row and column in B, indicating the number of mismatched bits. 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.

We use 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 half of the crossbar size (3 in this example), 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 3 now become larger than 3. For example, the score of the forth column in Fig. 7 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 3. Note that even though all the scores are less than 3 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. 7, all the column scores and row scores are already less than 3, 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.

Reconstructing. This step uses the metadata (CFVs and RFVs) produced in the previous step to reconstruct each bit matrix in the cluster from the centroid bit matrix. Fig. 8 shows how 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 consists of all the reconstructed bit matrices.

Calibrating. Because the modified DNN produced in the previous step introduces noises to the weights which impacts the DNN's accuracy. As shown in the third column of Table I, there is an accuracy drop from 76.13% to 71.628% for `ResNet50` if we directly use the modified DNN in the inference phase. We tackle this problem by updating the distribution statistics in batch normalization layers. One thing to note here is that we only update the distribution statistics of batch normalization layers, instead of the trainable parameters (i.e. the gamma weights and beta weights) of the batch normalization layers. Because distribution statistics are collected during the forward propagation, we only need to rerun the forward propagation phase on the training data without the need for backward propagation and weight update. The last column in Table I shows the accuracy after updating the batch normalization statistics.

TABLE I: Batch normalization'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%

Mapping. 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. Fig. 9 shows an example of mapping the centroid bit matrix C and the metadta of A in Fig. 7 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 $\overline{RFV}$.

Inference. In Fig. 9, 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. 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. 9 indicates the different cells between C and A'. If we denote the bits in A' as B_i, the dark cells stores

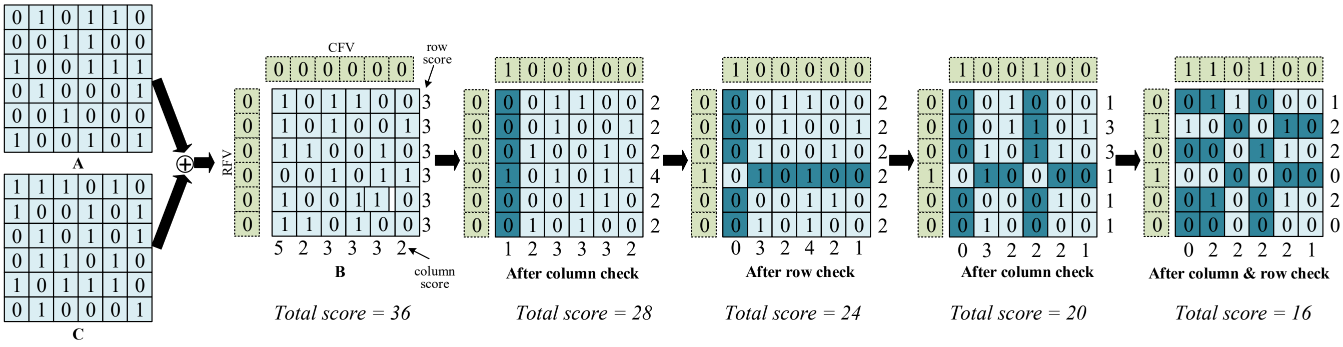


Fig. 7: Flip rows and columns to minimize the distance of two matrices. A is a bit matrix in the cluster. C is the centroid bit matrix of the cluster. B is calculated by XORing A and C.

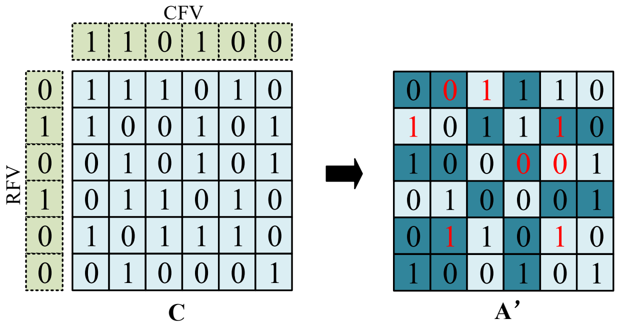


Fig. 8: Reconstruct A' from C.

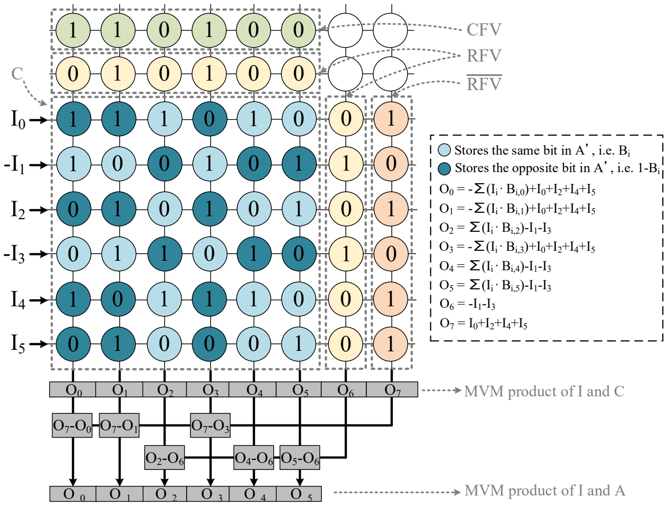

CFV
RFV
$\overline{RFV}$
C
I_0
$-I_1$
I_2
$-I_3$
I_4
I_5
Stores the same bit in A', i.e. B_i
Stores the opposite bit in A', i.e. $1\text{-}B_i$
$O_0 = -\Sigma(I_i \cdot B_{i,0})+I_0+I_2+I_4+I_5$
$O_1 = -\Sigma(I_i \cdot B_{i,1})+I_0+I_2+I_4+I_5$
$O_2 = \Sigma(I_i \cdot B_{i,2})-I_1-I_3$
$O_3 = -\Sigma(I_i \cdot B_{i,3})+I_0+I_2+I_4+I_5$
$O_4 = \Sigma(I_i \cdot B_{i,4})-I_1-I_3$
$O_5 = \Sigma(I_i \cdot B_{i,5})-I_1-I_3$
$O_6 = -I_1-I_3$
$O_7 = I_0+I_2+I_4+I_5$
O_0 O_1 O_2 O_3 O_4 O_5 O_6 O_7
MVM product of I and C
O_7-O_0 O_7-O_1 O_7-O_3 O_2-O_6 O_4-O_6 O_5-O_6
O'_0 O'_1 O'_2 O'_3 O'_4 O'_5
MVM product of I and A

Fig. 9: Map centroid bit matrix and meta data to crossbar.

$1 - B_i$. 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'.

VI. SC Conversion

After each MVM operation in the crossbar, the results are back in binary format. So, there is a need to convert the results to SC format before the next MVM operation. Conventional SC conversion uses linear feedback shift registers (LFSRs) to generate random bit streams with a specific percentage of 1s. However, LFSR-based SC conversion can only generate 1 bit at a time, introducing a long latency for each conversion. For faster SC conversion, We propose to reuse the sign bits of the weights as the source to generate random bit streams.

For each computation between the inputs and the weights, the weights' sign bits also need to be read out from the sign bit crossbar. Since the distribution of 1s in the sign bits of the weights is known beforehand, we can inject 0s or 1s into the sign bits to adjust the ratio of 1s according to the target input value. Fig. 10 shows an example of converting the input 0.85 to SC format, assuming the ratio of 1s in the sign bits are 40% (represent a value of 0.4). In the example, 115 1s are injected into the sign bits consecutively starting from the first bit position. There may be other inject patterns by changing the start bit position of the consecutive 1s.

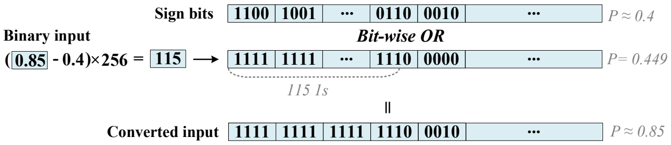


Fig. 10: An example of converting 0.85 from binary format to SC format.

In order to protect the intermediate results, we use a hash table to select a different inject pattern for different inputs. Because 1s are not evenly distributed along the sign bits, using different inject patterns will introduce some random noises into the converted SC values. The hash table is kept confidential in an SRAM buffer on-chip and stored encrypted off-chip. Therefore, the adversary can only observe the binary format intermediate results stored off-chip while the SC format values participated in the computation are dynamically generated on-chip.

VII. Experimental Results

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	ResNet50, ResNeXt50, GoogLeNet, DenseNet

We test our design on four datasets: MNIST [2], SVHN [3], Cifar10 [4] and ImageNet [5]. Table II shows the network structure details. We use PyTorch for fine-tuning and accuracy evaluation. we use the PuLP library to solve the MIP optimizations. For performamce and energy measurements, we compare with ISAAC [6] as it is the most widely used baseline in other ReRAM based DNN accelerator designs. Comparing with ISAAC makes it easy to scale our design's performance and energy consumption numbers to compare with other works.

A. IP Protection Effectiveness

Table III shows the effectiveness of our design's protection on DNN IP. The second column shows the inference accuracy on ISAAC baseline. The third column shows the accuracy on the target device in our design after applying the SC-based protection scheme and our two sharing schemes. The accuracy is very close to the baseline (less than 1% accuracy loss) The fourth column shows the pirate device's accuracy of using the same ReRAM crossbar content from the target device. However, the pirate device does not know the *GSL*s in the target device. There is a significant accuracy loss in the pirate device.

TABLE III: Inference Accuracy

	Baseline	Target Device	Pirate Device
MNIST	99.35%	99.36%	14.98%
SVHN	95.94%	95.23%	67.75%
Cifar10	90.13%	90.11%	39.47%
ResNet50	76.13%	75.43%	14.82%
ResNeXt50	77.61%	77.01%	13.9%
GoogLeNet	69.77%	67.82%	10.44%
DenseNet	74.434%	73.86%	20.44%

B. Performance

Fig. 11 shows the performance of our design compared to ISAAC. Since the *GSL*s only activate half of the crossbar at a time, the read speed is much faster than activating the entire crossbar [6]. Also, because every segment of 4 cells is completely isolated from other cells in the crossbar, the sneak current problem is effectively suppressed, reducing the sensing delay of the ADCs. As a result, the overall latency is better than the baseline. On average, there is a 1.14× speedup.

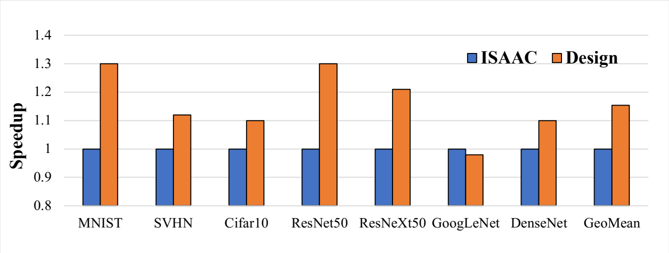


Fig. 11: Speedup normalized to ISAAC.

C. Energy

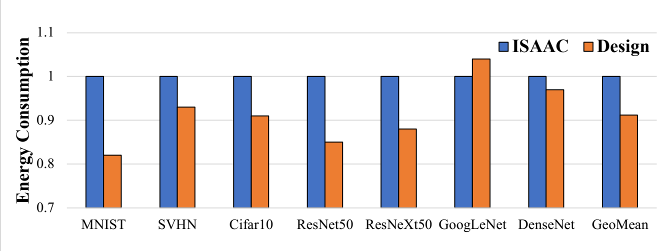


Fig. 12: Energy consumption normalized to ISAAC.

Fig. 12 shows the energy consumption of running the tested benchmarks normalized to ISAAC. Although each computation needs two ReRAM accesses — one for reading the sign bits and the other for MAC operations, the overall energy consumption is still almost the same or slightly better than the baseline. This is because the SC conversion only reads one row out from the crossbar, its energy is much smaller than a full crossbar read. For MAC operation, our design consumes less energy because only half of the cells are activated during computation, and the other half is completely shut off by the transistors of each segment. On average, there is a 9% less energy consumption compared to ISAAC on average.

VIII. Conclusion

This paper proposes a complete solution for DNN IP protection on ReRAM-based accelerators by using SC computing. To tackle the larger storage overhead brought by SC, we proposes two compression schemes at different granularity. Our proposed solution not only provides effective protection on DNN IP, but also achieves performance and energy benefits.

References

[1] C. Yeh *et al.*, "Compact one-transistor-N-RRAM array architecture for advanced CMOS technology," in *IEEE Journal of Solid-State Circuits*, 2015

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

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

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

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

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