SNrram: An Efficient Sparse Neural Network Computation Architecture Based on Resistive Random-Access Memory

Peiqi Wang[1,2]*, Yu Ji[1,2], Chi Hong[1], Yongqiang Lyu[1], Dongsheng Wang[1], and Yuan Xie[2]

Tsinghua University[1], University of California, Santa Barbara[2]
wpq14@mails.tsinghua.edu.cn[1], wds@mail.tsinghua.edu.cn[1], yuanxie@ece.ucsb.edu[2]

ABSTRACT

The sparsity in the deep neural networks can be leveraged by methods such as pruning and compression to help the efficient deployment of large-scale deep neural networks onto hardware platforms, such as GPU or FPGA, for better performance and power efficiency. However, for RRAM crossbar-based architectures, the study of efficient methods to consider the network sparsity is still in the early stage. In this study, we propose **SNrram**, an efficient sparse neural network computation architecture using RRAM, by exploiting the sparsity in both weights and activation. SNrram stores nontrivial weights and organizes them to eliminate zero-value multiplications for better resource utilization. Experimental results show that SNrram can save RRAM resources by 69.8%, reduce the power consumption by 35.9%, and speed up by 2.49× on popular deep learning benchmarks, compared to a state-of-the-art RRAM-based neural network accelerator.

Categories and Subject Descriptors

DES2.2 [**Machine Learning/AI**]: Architectures for machine learning and artificial intelligence

Keywords

Neural network, sparsity, resistive random access memory, architecture

1. INTRODUCTION

Deep neural networks (DNNs), which have been creating significant breakthroughs in a range of fields, has a very high computation and memory resource requirements. Accelerators based on conventional hardware platforms [7, 4, 10] still need considerable hardware investment and high power budget for large-scale DNN deployment. Most neural network models have significant redundancy and can be pruned noticeably without accuracy loss [2, 14]. The sparsity of an original neural network can be over 90% [12], which means that the memory requirement of large-scale neural networks can be significantly reduced. Eliminating unimportant weighted connections between layers can effectively reduce computation as well as memory requirements to satisfy resource constraints. Such a high-degree sparsity comes with the great potential of achieving high efficiency on various deep learning platforms, including CPUs, GPUs, FPGAs, and customized hardware accelerators [1, 13, 16].

Novel architectures based on emerging devices such as the resistive random-access memory (RRAM) [17, 20] have been proposed as alternatives for better performance and power efficiency [6], by reducing the data-movement overheads with the concept of near-data computing or processing-in-memory. However, for RRAM-based neural network (NN) accelerator architecture, the study of efficient methods to utilize the sparsity in neural networks is still in its early stage. First of all, many current RRAM-based designs adopt tightly coupled crossbar structure. If mapping a sparse matrix computation to RRAM cells directly, it is difficult to skip zero-value computations. Second, the overhead of peripheral circuits has always been a big challenge. Sparse neural networks need extra phase to decode sparse format of data. Any intention of exploiting the sparsity directly as conventional platforms do [1, 13] will cause significant performance and power overheads on peripheral circuits of RRAMs.

Fortunately, the emergence of structured pruned algorithms in DNNs [23, 24] offers a promising prospect for RRAM-based accelerators. They regularize the structure of DNN and prune weights in a systematic way, leading to a hardware-friendly compact neural network. For example, there will be continuous zeros in one dimension of a tensor matrix after structured pruning. Therefore, we can solve the first challenge by splitting and reassembling such structural sparse matrices into dense ones, then efficiently optimize resource allocations on RRAM. For the second challenge of peripheral circuitry overheads, we utilize both computation and memory capability of RRAM to decode and store sparse data format, avoiding expensive peripheral circuits.

In this paper, we propose an RRAM-based sparse neural network computation architecture (SNrram), which effectively exploits the sparsity in both weight and activation of neural networks in the inference phase. In previous work of other architectures, processing of sparse data during ac-

* This work was supported by Chinese National Key Research and Development Program (Grant No. 2016YFB0200505, 2017YFB0403404), Beijing Innovation Center for Future Chip (Grant No. KYJJ2016005), and a grant from China Scholarship Council. It was also supported in part by NSF 1533933/1719160

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

DOI: https://doi.org/10.1145/3195970.3196116

cessing as well as computing always incurs significant sparse format decoding overheads. In our architecture, this problem can be solved with negligible overheads. Moreover, SNrram transfers the sparsity in activation into the sparsity in weight, and only uses a small area of RRAM crossbar units to store and compute corresponding data in a dense manner. There are multiple processing units running in parallel to increase performance at the same time. In addition, SNrram stores only nontrivial data, organizes them efficiently, and skips the multiplications with zero elements. The main contributions of this work are summarized as follows:

- We propose an RRAM-based sparse neural network computing architecture. To the best of our knowledge, this is the first work to exploit the sparsity both in weight and activation in RRAM-based NN architecture, providing efficient memory and computing schemes for sparse neural networks. This work enables to efficiently deploy large-scale neural networks on RRAM-based NN accelerators.
- We introduce a sparsity transfer algorithm (STA) to normalize the sparsity in both weight and activation into the same format, which greatly reduces the architecture design overhead.
- We design an indexing register unit (IRU) with RRAM crossbars to store sparse indexes and parse the sparse format of data, avoiding controller or peripheral circuit overheads.
- Our experiments verify the efficiency of the proposed architecture. It can achieve 2.49× speedup, 69.8% resource saving, and 35.9% power reduction in comparison with the state-of-the-art.

2. RELATED WORK

Neural network acceleration. Deep neural networks (DNNs) require intensive computation and memory resources, which greatly challenges the deployment of large-scale DNNs to terminal devices with computing, memory, and power constraints. A number of accelerators have been proposed to improve the computing performance of DNNs. CNP [10] exploited the inherent parallelism of convolution networks and takes full advantages of FPGA. DaDianNao [4] introduced a custom multi-chip design of high internal bandwidth and low external communications to enable high-degree parallelism. PRIME [6] outperforms the accelerators based on GPUs, FPGAs, and ASICs over 2000 times by using resistive random access memory (RRAM), a non-volatile memory that stores information by changing cell resistances or conductances. ISAAC [21] is a pipelined RRAM crossbar architecture and defines new data encoding techniques to reduce the power and area overheads of peripheral circuits. These studies to be superior in quality but do not utilize the sparsity in networks, which has emerged to be a key role to boost performance.

Sparse neural networks have been intensively studied on algorithms to provide speedup and save resources. Denil et al. [8] obtained 95% sparsity by using the low-rank matrix factorization. Han et al. [12] combined weight pruning, quantization, and Huffman coding methods together to reduce network size, obtaining 3× to 4× speedup. Traditional hardware architectures need to be revised to exploit the potential of sparse neural networks. Cnvlutin [1] and Eyeriss [5] utilized the sparsity by compression and skipping computation of zero-valued activations in processors. SCNN [16] used Cartesian product operations between nonzero-valued weights and activations in processors.

Architectural support for sparsity. When deploying sparse neural networks subjects to underlying hardware architectures, the irregular pattern of the distribution of trivial weights caused by pruning always degrades the overall performance[24].A range of studies focuses on adapting the sparse network structures to be hardware-friendly in an algorithmic way. Wei el at. [23] learn a compact structure from large DNNs and obtain a hardware-friendly structural sparsity. Yu et al. [24] customize DNN pruning to the underlying hardware by matching the pruned network structure to the data-parallel hardware organization. These studies show potential benefits to traditional platforms with some extra overhead [13], but current RRAM-based accelerators still hardly get further benefits. Work on how to handle sparse neural networks efficiently in RRAM-based architecture is still lacking, which have much potential to be exploited.

It is difficult for RRAM crossbar architectures to process sparse neural networks efficiently because of the coupled structure of crossbars. In the RRAM-based NN accelerator, the matrix is stored by the conductances of the RRAM cells and the vector is represented by the input voltage signals during the matrix-vector multiplication in RRAM crossbar. When voltages are applied to each row, each cell generates current to perform multiplication. The out current of each column accumulates the corresponding currents on each row branch. All columns in the entire crossbar then naturally achieve matrix-vector multiplication. Such a structure can avoid conventional memory access and enable high speed and energy efficiency. However, if some elements are equal to zero, we cannot remove them directly because there are also non-zero data in the same row or column. All rows and columns are coupled together at the physical crossbars and it is impossible to eliminate several elements unless the entire row or column is all zeros. Unfortunately, it is fairly difficult to find completely zero-value rows or columns in weights matrix in sparse neural networks. As the sparse network pruned by an algorithm proposed in prior work[23], the sparsity in one convolution layer achieves 61.3% while only 6.2% channels are pruned completely. Consequently, we can only get a small benefit directly even the overall sparsity is extremely high in pruned networks. Therefore, in this study, we propose a novel RRAM-based sparse neural network processing architecture called SNrram, which is the first work to address this issue.

3. UTILIZING SPARSITY IN RRAM

In this study, we propose SNrram, an RRAM-based sparse neural network computation architecture, which effectively exploits the network sparsity in both weight and activation to improve performance and save resources. This method enables RRAM-based NN accelerator to efficiently deploying large-scale neural networks onto resource-constrained devices.

3.1 Sparsity in Weight

We take convolutional neural network (CNN), which has shown powerful capabilities in a wide range of applications, as an example. Generally, there are mainly three dimensions of the sparsity in weight of CNNs: the intra-filter sparsity

refer to the redundant weights inside convolution filters; the input-channel-wise sparsity refers to pruning weights of one or more input channels that are connected to the next layer; the output-channel-wise sparsity, similarly, refers to pruning the weights of output channels in the current layer.

In traditional RRAM, given a convolution layer with $k \times k$ sized filters, C input channels, and M output channels, it has to fully store the filters as expanded into an $N \times M$ 2D-matrix, where $N = k \times k \times C$ due to the coupled relationship between rows and columns. In order to save memory, we divide the entire matrix into C sub-matrices with respect to the input channels, thus the size of each sub-matrix is $g \times M$, where $g = k \times k$. Therefore, as long as an output channel is pruned in any input channel, we can get a whole empty column in a sub-matrix. Hence, SNrram discards these pruned columns in every sub-matrix and stores the remained nontrivial weights densely into separate crossbar arrays. Fig. 1 gives a simple example with $k = 2, C = 2, M = 4$. For the original 8×4 sparse matrix weights in Fig. 1 (a), we split it into 2 sub-matrices along the row. The two sub-matrices are reorganized into two crossbar arrays and discard columns with all zeros, as shown in Fig. Fig. 1 (b). Because of the unbalance pruning, there's still some zero-value remains in Group 1. We will solve this in future work.

In order to efficiently utilize on-chip resources, we need to optimize resource allocations for these dense sub-matrices. How to effectively allocate the RRAM computation resources to the application is challenging. We follow two essential design philosophies: (1) The computation tasks of different partitions should be balanced, in order to achieve the highest efficiency. (2) Avoid waste of computation resource, either unless computation or null. Following these philosophies, we design the resource allocation scheme as shown in Fig.1 (c). First, we reassemble sub-matrices with the same shape into different parts of one crossbar to higher utilization. Second, we assign different numbers of crossbar arrays to each group according to their shape, in order to balance the computation tasks. Third, we take use of the parallelism in output channel to increase performance. In real situations, the ratio of a sub-matrix is always acute (unbalanced matrices), i.e. the width is far larger than height. A direct mapping of such sub-matrix results in resource waste, because one side of the crossbar to be occupied while the other side is empty. Considering the dependency between columns, in such cases, we split unbalanced sub-matrices again with respect to columns to get a nearly square sub-matrix. Looking at the instance shown in Fig.1 (c), blue cells refer to zero-value and other color cells represent different sub-matrices. We map two independent matrics in the same shape and reused by same input data into the top-right crossbar, which helps increase both performance and resource utilization. We assign two crossbars to the related sub-matrices computation tasks, shown in the top-left, to balance the computation. These subarrays work in parallel on the partial results and an extra unit is needed to sums up the final results.

We introduce an **indexing register unit (IRU)** to accumulate partial results and to parse their sparse format. According to the splitting and reassembling, corresponding sparse indexes are stored in IRU in an expanded way by setting corresponding cells to physically conduct, as shown in Fig. 1 (c). Grey cells mean it's null or turn-off while red cells

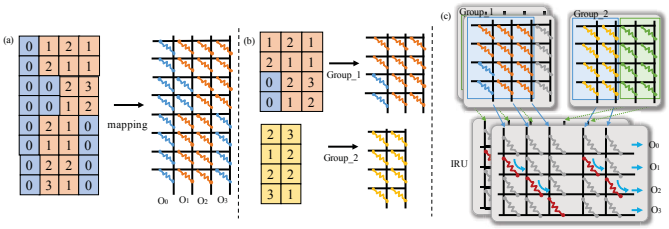


Figure 1: (a) Use RRAM crossbar as a matrix-vector multiplier. (b) SNrram splits one sparse matrix into some dense sub-matrices. (c) Resource allocation scheme in SNrram.

are turned on. If the cells in different columns of the same row are red, the corresponding partial results sent to these columns are accumulated and the results stream out from this row. The output of indexing register unit is directly placed in correct coordinates without extra computation or peripheral circuit controllers.

The IRU also consumes memory, which is negligible compared to the amount of saved RRAM resources by sparse formats. Assuming the sparsity rate of weight matrix is γ, the number of input channel is C, the number of output channel is M, the convolutional filter size is $k \times k$, and the number of remaining weights in each sub-matrix is $n_i (i = 1, 2, ..., C)$. To simplify representation, let $M = \alpha C$, which means the number of output channel is α times of input channel. The ratio between the SNrram's memory footprint and the original memory footprint without utilizing sparsity can be defined as:

$$R = \frac{k^2 \sum_{i=1}^{C} n_i + M \sum_{i=1}^{C} n_i}{k^2 \times C \times M} = (\frac{1}{C} + \frac{\alpha}{k^2})(1 - \gamma) \quad (1)$$

The sparsity rate in weight can be normally over 60% [23], and the relationship $\frac{\alpha}{k^2} < 1$ is valid in most popular CNN modules. Therefore, $0 < R < 1$ is always true, which means SNrram can save memory totally.

For other neural networks without weight sharing, e.g. multilayer perceptron(MLP), they can be processed in the similar method. We still divide the weight matrix into several sub-matrices with size $g \times M$. In order to guarantee weights in one row or line owns the same sparse pattern, we apply the algorithm proposed in prior work [24] to align weights in the same sub-matrix. The sparsity of weight matrix can still achieve 88% without any accuracy loss[24]. Therefore, we can use the same method proposed above to deal with general neural networks.

3.2 Sparsity in Activation

In addition to the sparsity in weight, the sparsity in activations of a sparse neural network can also be utilized to improve the efficiency. There are two types of activation sparsity. The first one is the fixed sparsity caused by the algorithm itself. For example, the widely used transposed convolution operation, which is also called fractionally strided convolution or deconvolution, is widely used in semantic image segmentation and generative adversarial networks [11, 15]. It is often involved in adding many columns and rows of zeros to the input, resulting in huge fixed sparsity in activations. The other is the random sparsity that arises dynamically during computation processing. In this study, we mainly focus on the fixed sparsity in activation.

Fixed sparsity in activation always has regular patterns.

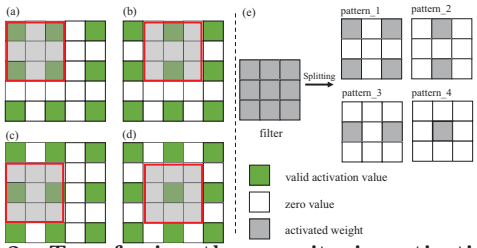


Figure 2: Transferring the sparsity in activation into weight. In this example, $S_w = S_h = 1$ and the filter size is 3×3.

To process structural sparsity, we propose a sparsity transfer algorithm (STA) as follows to transfer it into weight.

Algorithm 1: Sparsity Transfer Algorithm

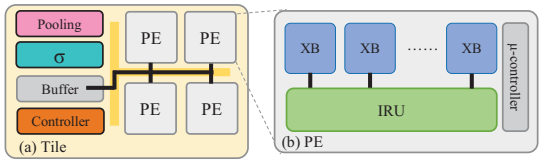
Input: k, S_w, S_h, $valuesofsharedfilter$
Output: $filterpatternsafterdecomposed$
Decompose original transposed convolution filter into several patterns according to structured sparsity in activations;
$l_w = \left\lfloor \frac{k-1}{S_w+1} \right\rfloor + 1$;
$l_h = \left\lfloor \frac{k-1}{S_h+1} \right\rfloor + 1$;
$N = (S_w + 1) \times (S_h + 1)$;
for $n = 0 : N - 1$ **do**
  $offset_w = n \times (S_w + 1)$
  $offset_h = n \times (S_h + 1)$
  $round_w = (l_w + 1) \times (S_w - 1)$
  $round_h = (l_h + 1) \times (S_h - 1)$
  for $j = 0 : l_h$ **do**
    for $i = 0 : l_w$ **do**
      $x = (i \times S_w + 1 + offset_w)\%round_w$;
      $y = (j \times S_h + 1 + offset_h)\%round_h$;
      if $x > k - 1$ *or* $y > k - 1$ **then**
        Continue;
      end
      $Subfilter[n][i][j] = filter[x][y]$
    end
  end
end

In the networks without weight sharing, we can directly hide the weight elements corresponding to the zero-value activations, using the coordinates to calculate the positions. In this case, we can use exactly the same method as the weight sparsity in Section 3.1 to design the architecture. In weight-sharing networks, the situation gets complex. We take the transposed convolution operation as an example here. Assuming the size of filters is $k \times k$, the sparse width between columns is S_w and sparse height between rows is S_h. We can decompose the convolution filter into $(S_w + 1) \times (S_h + 1)$ patterns, each of which only stores the valid elements during the convolution filter sliding. An example is demonstrated in Fig. 2. The sparsity transfer algorithm is shown in Alg. 1, which executes before storing data in the RRAM arrays.

Having collected all the sparse patterns of weight, we store them in a dense matrix and then process the network in a unified method as mentioned in the Section 3.1, i.e., the sparsity in weight and sparsity in activation can be processed together in SNrram.

4. ARCHITECTURAL DESIGN

We show SNrram's architecture design in Fig. 3. It inherits most aspects of standard RRAM-based accelerator design. We elaborate on the design details and map neural networks on SNrram to show its processing flow in the rest of this section.



σ — Sigmoid Unit
XB — RRAM crossbar
PE — Processing Element
IRU — Indexing Register Unit

Figure 3: SNrram hierarchical skeleton.

4.1 SNrram Architecture

At the top view, the SNrram architecture is hierarchical, which contains multiple tiles connected with an on-chip concentrated mesh. Each tile has many processing elements (PEs), pooling unit, sigmoid unit, SRAM buffer, and controller (in Fig. 3(a)). All components are connected by a shared bus. During neural network computation, inputs are fetched from the on-chip buffer into the tile buffer and then distributed into each PE. The matrix-vector product operations involved in networks are performed in crossbar arrays inside PE. Other operations like pooling or activation functions are finished by corresponding units. Because a large-scale neural network will be decomposed to run in several separate PEs, SNrram also needs to temporally hold the intermediate values in the tile buffer.

Fig. 3(b) shows the micro-architecture of a PE, including several RRAM crossbars used for computing, a indexing register unit (IRU) for indexing and accumulation, and other peripheral circuits such as digital to analog (DAC) units and analog to digital (ADC) units necessary to RRAM functions (omitted in the figure). There are also trans-impedance amplifiers [18] between crossbars to support the communication to IRU.

SNrram has three levels of controllers: chip, tile, and processing element-level controllers. The first level controller is essentially on-chip decoder, which supports simultaneous processing for great parallelism. The tile-level controller (in Fig. 3(a)) decode the instructions and allocate resources based on the scheme. The μ-controller in PEs (in Fig. 3(b)) converts the instructions and operations into control signals to the peripheral circuits in RRAM crossbar arrays.

The RRAM crossbars in each PE are used to process the matrix-vector multiplication of neural networks. For the negative value representation, we adopt the same solution as PRIME [6] because the RRAM cells can only represent positive values, i.e., separating weight matrix into two matrices. One stores the positive weights and the other stores the negative weights. Meanwhile, we also use multi-level cells to store more than one bit of information in a single cell by various levels of resistance to improve the RRAM density and computation accuracy. The merging of positive-negative, as well as higher-lower weights, is done in modified ADC units in PEs, which are also the same as PRIME.

4.2 SNrram Deployment

Fig. 4 demonstrates an overview of SNrram processing flow. Step 1, the sparsity in activations will be transferred into weighs according to the sparsity transfer algorithm (STA) proposed in Section 3.2. Step 2, the unified sparse weight matrix is expanded as we illustrate in Section 3.1. Step

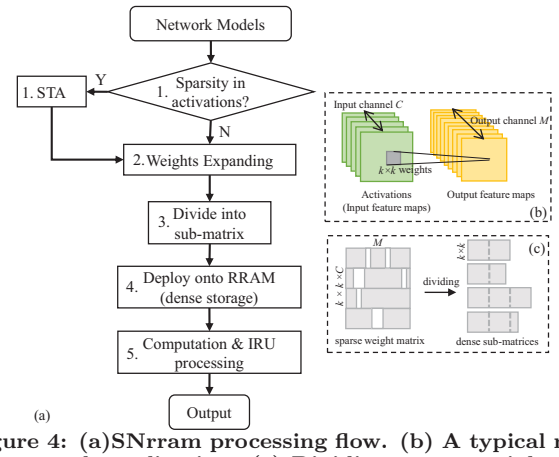


Figure 4: (a)SNrram processing flow. (b) A typical neural network application. (c) Dividing sparse weight matrix into dense weight matrix.

3, the expanded matrix reflecting both weight and activation sparsity will be divided into several sub-matrices and stored densely in each RRAM crossbar array in step 4. By pre-programming the RRAM cells, the input vector (voltage signals) are driven in row-lines, and the output vector (current signals) are accumulated in column-lines. Then, the sparse indexes will be collected into indexing register unit (IRU), to which each sub-matrix is connected, and the output accumulation results are transferred in step 5. The outputs of IRU have sorted automatically in the correct coordinates without extra procedures. The first three steps are achieved in application-level and compiler-level operations and the other steps are in-charged by controllers in SNrram architecture. Furthermore, we can also write bias into RRAM cells, setting the corresponding input as "1" to implement bias addition operation in neural networks.

For different scales of deep neural networks, SNrram can efficiently utilize the saved RRAM resources compared to the original architecture. Small-scale networks can be mapped to a single PE, even just parts of crossbar arrays in a PE. In such a situation, we replicate this network and map them to different independent portions in a PE, and then make all other PEs process copies of the network in parallel. Medium-scale networks can be performed using a single tile with several PEs, in which the network will be split into and processed by different PEs in one tile. In this situation, we store the intermediate values into the tile buffer and use some nearby crossbar arrays in a PE to accumulate the results of every part. Large-scale networks that cannot be mapped to one single tile can be split into different PEs over tiles. Every tile performs parts of the network and some crossbar arrays in a tile can take charge of the output accumulation.

5. EVALUATION

In this section, we first describe the experimental setup. Then, over-all performance, energy, and resource efficiency are presented with comparisons to the state-of-the-art solution.

5.1 Experimental Setup

The benchmarks used in this study include four neural networks, listed in Table 1. The first two are used to recognize the MNIST dataset. The VGG16[22] is a large CNN for the ImageNet dataset. The DCGAN (Deep Convolutional

Table 1: The benchmarks and topologies

Name	Structures of networks
MLP	784-500-250-10
Lenet	conv5x20-pool-conv5x50-pool-500-10
VGG	conv3x64-conv3x64-pool-conv3x128-conv3x128 pool-conv3x256-conv3x256-conv3x256-pool conv3x512-conv3x512-conv3x512-conv3x512 conv3x512-conv3x512-pool-25088-4096-4096-1000
DCGAN	deconv5/2x1024-deconv5/2x512-deconv5/2x256 deconv5/2x128-deconv5/2x3

Table 2: The performance of typical convolution layers in pruned VGG.

Layer	Input # - output #	Pruned connection	Saved resource	Speed up
conv1-2	64-64	56.5%	46.3%	1.23×
conv2-2	128-128	34.5%	12.9%	0.76×
conv3-1	128-256	34.7%	13%	0.71×
conv4-1	256-512	61.3%	22.8%	2.29×
conv5-1	512-512	73.8%	58.5%	2.67×
VGG	–	62.7%	53.2%	1.53×

Generative Adversarial Network) [19] is a typical generative adversarial network for the celebA dataset. We refer to the pruning algorithm in [23] to learn the sparsity in these networks first, and then deploy them on SNrram.

The baseline of our experiments is PRIME [6], the current state-of-the-art RRAM-based neural network accelerator. We evaluate two modes of SNrram: in **SNrram-partial**, we skip the step 1 and step 3 in Fig. 4, deploying sparse neural networks directly onto SNrram architecture; in **SNrram-full**, we implement complete functions in SNrram. Therefore, the difference between baseline and SNrram-partial comes from the sparse neural network itself, while the difference between SNrram-partial and SNrram-full due to all the methods we proposed. In the experiments, our design contains 8 tiles on one chip and each tile owns 4 PEs. For each PE, there are 128 crossbar arrays, each of which owns 128 × 128 RRAM cells. For each RRAM cell, we assume a 2-bit MLC for computation[3]. We model above design configuration using modified NVSim [9] and NeuroSim+ [3] with 65nm TSMC CMOS library.

5.2 Experimental Results

Performance results. By using the same RRAM computation resources as PRIME, the performance of our layer-wise VGG is shown in Table 2 and Fig. 5, and others are summarized in Fig. 6. Overall, SNrram-full achieves only

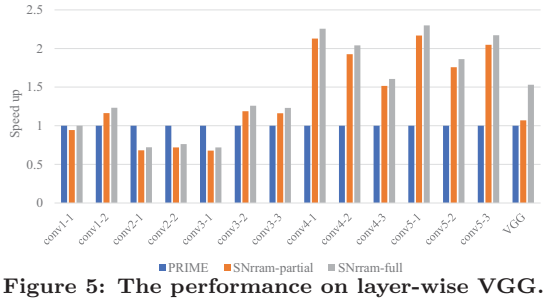


Figure 5: The performance on layer-wise VGG.

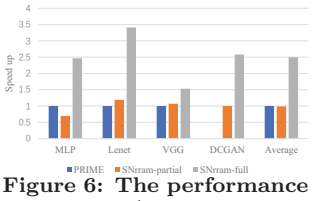


Figure 6: The performance comparison(normalized to PRIME).

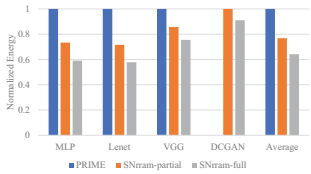
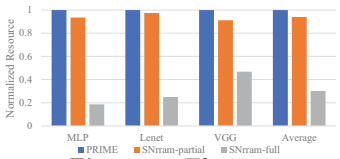


Figure 7: The energy consumption.



Models	Sparsity	Saved resource	
		vs. PRIME	*vs.* SNrram-partial
MLP	82.7%	81.3%	74.7%
Lenet	96.2%	75%	71.4%
VGG	62.7%	53.2%	42.3%
Average	85.7%	69.8%	62.8%

Figure 8: The resource utilization efficiency.

1.53× speedup in VGG, which is mainly caused by the accumulation overhead of unbalanced pruning. As we discussed in Section 3.1, such an unbalanced pruning becomes especially severe in layers having less sparsity, like conv2-2 and conv3-1. While in the high sparsity layers like conv4-1 and conv5-1, SNrram-full can perform well. We will solve the high overhead in unbalanced pruning situations in future work.

Comparing SNrram-partial with the baseline, little benefit is obtained from the sparse neural network itself directly although high sparsity inside. However, SNrram-full to be superior in quality with an average 2.49× speedup over the baseline, which means our proposed methods can exploit and utilize the sparsity feature efficiently. Among all the benchmarks, SNrram-full achieves the highest speedup on Lenet, which is about 3.41× over PRIME. It is obvious that Lenet has the highest sparsity (in Fig. 8). The SNrram-partial performs even worse than PRIME in MLP because of unbalanced pruning and extra accumulation overhead. Nevertheless, SNrram-full can still get benefits from the sparsity on it. The sparsity of DCGAN lies in activation, and we just compare the performance of SNrram-partial and SNrram-full due to the absence of PRIME. We can obtain 2.58× speedup from 79.6% sparsity, as shown in Fig. 6.

Energy results. Fig. 7 presents the energy-saving results, which seem a relatively small improvement because PRIME actually designed a low-power peripheral circuit but SNrram does not. Even so, SNrram (in full) can save about 35.9% power consumption on average over the baseline because we use less computation resource.

Resource efficiency. We measure the resource efficiency through how much computation resource is required in minimum(i.e. the RRAM cells needed for storage and computation). As shown in Fig 8, SNrram (in full) is able to achieve 69.8% computation resource saving on average over PRIME. The saved RRAM recourse comes from the sparsity in weights, thus DCGAN (without weight sparsity) is skipped in this test. The small gap between the baseline and SNrram-partial is because SNrram provides index register unit to achieve the accumulation operation in PEs while PRIME needs extra more crossbar arrays to finish accumulation. The difference between SNrram-partial and SNrram-full is caused by discarding nontrivial weights, which confirms SNrram takes great advantage of sparsity.

6. CONCLUSION

As a promising architecture with the potential of acceleration and power efficiency on resource-constrained devices, RRAM is still not able to utilize the network sparsity, which has severely influenced the deployment of large-scale deep neural networks to resource-constrained devices.This paper proposes the SNrram, an efficient sparse neural network computation architecture based on RRAM, which exploits the sparsity in both weight and activation. SNrram introduces two innovations with the sparsity transfer algorithm and the indexing register unit to process the sparsity efficiently. The experiments show that it can achieve 2.49× speedup, 69.8% resource saving, and 35.9% power reduction in comparison with the state-of-the-art RRAM accelerator.

7. REFERENCES

[1] J. Albericio et al. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *ISCA*, 2016.

[2] S. Changpinyo et al. The power of sparsity in convolutional neural networks. *arXiv:1702.06257*, 2017.

[3] P. Chen et al. Neurosim+: An integrated device-to-algorithm framework for benchmarking synaptic devices and array architectures. In *IEDM*. 2017.

[4] Y. Chen et al. Dadiannao: A machine-learning supercomputer. In *MICRO*, 2014.

[5] Y. H. Chen et al. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *JSSC*, 52(1):127–138, 2017.

[6] P. Chi et al. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ISCA*, 2016.

[7] A. Coates et al. Deep learning with cots hpc systems. In *ICML*, 2013.

[8] M. Denil et al. Predicting parameters in deep learning. In *NIPS*, 2013.

[9] X. Dong et al. Nvsim: A circuit-level performance, energy, and area model for emerging non-volatile memory. In *Emerging Memory Technologies*. 2014.

[10] C. Farabet et al. Cnp: An fpga-based processor for convolutional networks. In *FPL*.

[11] I. Goodfellow et al. Generative adversarial nets. In *NIPS*, 2014.

[12] S. Han et al. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv:1510.00149*, 2015.

[13] S. Han et al. Eie: Efficient inference engine on compressed deep neural network. In *ISCA*, 2016.

[14] B. Liu et al. Sparse convolutional neural networks. In *CVPR*, 2015.

[15] J. Long et al. Fully convolutional networks for semantic segmentation. In *CVPR*, 2015.

[16] A. Parashar et al. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *ISCA*, 2017.

[17] M. Prezioso et al. Training and operation of an integrated neuromorphic network based on metal-oxide memristors. *Nature*, 521(7550):61–64, 2015.

[18] M. S. Qureshi et al. Cmos interface circuits for reading and writing memristor crossbar array. In *ISCAS*, 2011.

[19] A. Radford et al. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv:1511.06434*, 2015.

[20] J. s. Seo et al. A 45nm cmos neuromorphic chip with a scalable architecture for learning in networks of spiking neurons. In *CICC*, 2011.

[21] A. Shafiee et al. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *ISCA*, 2016.

[22] K. Simonyan et al. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*, 2014.

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

[24] J. Yu et al. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *ISCA*, 2017.