Sparse ReRAM Engine: Joint Exploration of Activation and Weight Sparsity in Compressed Neural Networks

Tzu-Hsien Yang*, Hsiang-Yun Cheng§, Chia-Lin Yang*, I-Ching Tseng*,

Han-Wen Hu‡, Hung-Sheng Chang‡, Hsiang-Pang Li‡

*National Taiwan University, §Academia Sinica, ‡Macronix International Co., Ltd.
r06922094@ntu.edu.tw,hycheng@citi.sinica.edu.tw,yangc@csie.ntu.edu.tw,b04902016@ntu.edu.tw,
{paddingtonhu,billchang,sbli}@mxic.com.tw

ABSTRACT

Exploiting model sparsity to reduce ineffectual computation is a commonly used approach to achieve energy efficiency for DNN inference accelerators. However, due to the tightly coupled crossbar structure, exploiting sparsity for ReRAM-based NN accelerator is a less explored area. Existing architectural studies on ReRAM-based NN accelerators assume that an entire crossbar array can be activated in a single cycle. However, due to inference accuracy considerations, matrix-vector computation must be conducted in a smaller granularity in practice, called Operation Unit (OU). An OU-based architecture creates a new opportunity to exploit DNN sparsity. In this paper, we propose the first practical Sparse ReRAM Engine that exploits both weight and activation sparsity. Our evaluation shows that the proposed method is effective in eliminating ineffectual computation, and delivers significant performance improvement and energy savings.

CCS CONCEPTS

• **Computer systems organization** → **Neural networks**; *Special purpose systems*; • **Hardware** → *Memory and dense storage.*

KEYWORDS

Neural network, sparsity, ReRAM, accelerator architecture

1 INTRODUCTION

Deep neural networks (DNNs) have recently emerged as a highly effective solution for many classification and regression tasks, including image classification [26], object detection [39], and speech recognition [4]. Modern DNN models [26, 39, 41] are significantly larger than those used in the 1990s [10], requiring 138MB [41] for filter weight storage, 77MB [2, 41] memory accesses, and thousands to millions of computation operations per input activation. As the size of DNN models continues to grow, along with increasing accuracy and the ability to solve more complex problems, their intensive computing and memory demands introduce performance and energy efficiency challenges to the underlying processing hardware.

Memristor-based neural network accelerators have been shown to be a promising solution to meet the performance and energy efficiency challenges for DNN inference. In contrast to the conventional von Neumann architecture, where computation and data storage are separate, emerging memristor devices such as resistive random access memory (ReRAM) [22] are able to perform arithmetic operations beyond data storage. Recent works have demonstrated that ReRAM crossbar arrays can be used to efficiently perform matrix-vector multiplication in convolution and fully-connected layers of DNNs [9, 23, 40]. By storing filter weights as the conductance of ReRAM cells and converting input feature maps into input voltage signals, we have the dot-product results (output feature maps) at the end of the bitlines in ReRAM crossbar arrays. With such computing-in-memory capability, memristor-based DNN accelerators can reduce data movement and provide significant energy savings compared to CPU and GPU based DNN acceleration platforms [9, 40].

Despite this promising potential, the development of ReRAM-based DNN accelerators is still in its early stage and there remain challenges to overcome. One primary concern is how to efficiently exploit sparsity, which is commonly done in digital CMOS-based accelerators to improve energy efficiency [1, 8, 16, 36, 49]. Many studies have shown that common neural networks have significant redundancy in filter weights and can be pruned dramatically during training without substantially affecting accuracy [18]. In addition to weight sparsity, a massive amount of input activations are zeros in typical neural network models [15, 32], as many neural networks employ the ReLU function which clamps all negative activation values to zero as their nonlinear operator. Furthermore, in ReRAM-based DNN accelerators, there exists a finer-level of sparsity granularity to be exploited, bit-level sparsity, due to the cell bit-density and limited wordline driver resolution. Eliminating zero values in filter weights and input activations are important for both performance and energy. However, due to the tightly coupled crossbar structure, it is difficult to exploit sparsity efficiently in ReRAM-based DNN accelerators.

In a ReRAM crossbar architecture, weights stored in the same wordline need to multiply to the same input, and accumulated currents flowing through the same bitline contribute to the same output. Sparsity can only be exploited when the entire wordline or bitline cells contain zeros. Similarly, the sparsity of feature maps

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322271

can be leveraged when the input bits to the crossbar array are all zeros in the same cycle. A structural pruning algorithm [45] has been proposed to regularize the distribution of zero weights, so that more all-zero rows and columns can be found. SNrram [44] seeks to enable fine-grained column compression at the cost of high output indexing overhead.

Existing sparsity solutions [24, 44] are based on an over-idealized ReRAM crossbar architecture [9, 40]. Most existing ReRAM-based DNN accelerators published in the architectural community overlook the accumulated effect of per-cell current deviation on inference accuracy, as well as the overhead from ADC. Thus they assume that an entire 128×128 or 256×256 crossbar array can be activated in a single cycle. However, in practice, to achieve satisfactory inference accuracy, matrix-vector multiplication in ReRAM-based DNN accelerators must proceed at a smaller granularity, called an Operation Unit (OU) [31]. For example, only nine wordlines and eight bitlines are turned on concurrently within a 512×256 crossbar array in a state-of-the-art ReRAM macro designed for DNN acceleration [6]. Therefore, a practical OU-based ReRAM accelerator is very likely to deliver lower performance compared to an over-idealized design like ISAAC [40] or PRIME [9] since less computation is done in one cycle. However, unlike the over-idealized design, in which the entire crossbar array operates in one cycle, each OU in a crossbar is activated independently. This opens up a new design opportunity for sparsity exploration in a ReRAM-based DNN accelerator.

In this paper, we propose the first practical Sparse ReRAM Engine (SRE) that takes advantage of fine-grained OU-based computations to jointly exploit weight and activation sparsity. Weight compression can be done at the OU level in either the row or column dimension. Row-based compression requires input indexing to fetch the correct inputs for compressed weights, while column-based compression requires output indexing. For activation compression, a naive approach for OU-based ReRAM architecture is to skip an OU computation when inputs to all the wordlines of the OU are zeros. We exploit activation sparsity further with Dynamic OU Formation, a novel method which activates non-contiguous wordlines with non-zero input values in the same cycle to form an OU unit at run-time. This method works perfectly with row-wise weight compression to allow for joint exploration of weight and activation sparsity for ReRAM-based DNN accelerators. Evaluation results show that, for neural networks that use the structural pruning algorithm [45] to regularize weight sparsity during training, SRE provides up to 42.3x performance speedups (average 13.1x) and up to 95.4% energy savings (average 85.3%) over a baseline that does not exploit sparsity.

In summary, this paper offers the following contributions:

• We study the design challenges of sparsity exploration on ReRAM-based DNN accelerators, and show that a practical OU-based design enables new opportunities to effectively exploit DNN sparsity. To our best knowledge, this is the first sparsity exploration work that targets a practical hardware design for ReRAM-based accelerators instead of an over-idealized architecture.

• We propose a sparse ReRAM engine (SRE) to jointly exploit weight and activation sparsity, with only minimal indexing overhead. Our design takes advantage of fine-grained OU-based computations and combines row-wise weight compression with dynamic wordline activation to provide significant performance speedup and energy savings.

• We also compare SRE with the over-idealized ReRAM architecture, which overlooks the inference accuracy loss caused by the accumulated effect of per-cell current deviation. We show that SRE successfully enables a practical ReRAM-based DNN accelerator design , which achieves satisfactory inference accuracy considering the limitation of ReRAM cell reliability while delivering comparable performance and energy efficiency with the over-idealized ReRAM DNN accelerator.

The rest of the paper is organized as follows. Section 2 provides background on the architecture of ReRAM-based DNN accelerator and its challenges in exploiting DNN sparsity. The practical OU-based ReRAM accelerator design is introduced in Section 3, and the new opportunities to exploit weight and activation sparsity in such OU-based design is discussed in Section 4. The proposed Sparse ReRAM Engine is explained in details in Section 5. Section 6 and Section 7 describe evaluation methodologies and results, followed by a summary of related work in Section 8. Finally, we conclude the paper in Section 9.

2 BACKGROUND

2.1 ReRAM-based DNN Accelerator Architecture

Figure 1 shows the generic ReRAM-based DNN accelerator architecture assumed in a few works [24, 40]. The ReRAM-based deep learning accelerator is composed of multiple processing engines (PEs) connected with on-chip interconnects. Each PE consists of multiple computation units (CUs), each of which has multiple crossbar arrays, which are responsible for the acceleration of matrix-vector multiplications in convolution and fully-connected layers. By storing filter weights as the conductance of ReRAM cells and converting input feature maps into input voltage signals, we can obtain the dot-product results (output feature maps) by reading out the accumulated currents on the bitlines. A wordline driver (WLD) such as a digital-to-analog converter (DAC) or an inverter [40] is connected to each wordline of the ReRAM crossbar array to convert the input feature map data to input voltages. The accumulated currents on the bitlines (sum-of-products results) are read out by sample-and-hold (S&H) circuits and fed to the shared analog-to-digital converters (ADCs). In addition to the crossbar arrays, there is one on-chip eDRAM buffer in each PE for temporarily storing input and output feature maps. A non-linear function unit and a pooling unit are also included in the PE to support the implementation of the non-linear function and pooling layer in the neural network.

Figure 2 shows a high-level view of how to map filter weights to a crossbar array for a convolution layer with a 4×4×2 feature map and four 2×2×2 filters. The weights of one filter are mapped to the cells of one bit line. In an ideal ReRAM-based DNN accelerator design [24, 40], all the wordlines in the crossbar array can be activated concurrently in a single cycle. At every cycle, a 2×2×2 input vector of the input feature map is converted to the input voltages of the crossbar array via the WLD. The input sliding window (the red rectangular box in the figure) then shifts right (or down) and the corresponding input vector is fed into the crossbar array in the next cycle. Due to the limited WLD resolution and ReRAM cell

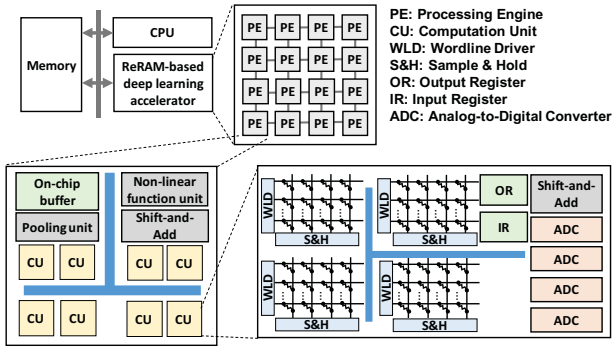

Figure 1: ReRAM-based DNN accelerator architecture.

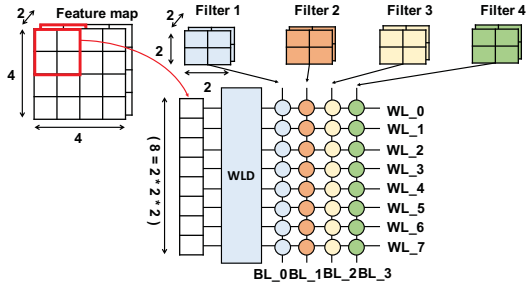

Figure 2: Mapping of filter weights and feqture maps to a crossbar array.

density, in practice, the input vector is decomposed and fed into wordlines using multiple clock cycles, and each data in the original filter weight is also decomposed and mapped onto different bitlines as illustrated in Figure 3. For simplicity, we show the mapping result of the weights only in the first channel of the filters. In this example, suppose that four filters are mapped to the 4×8 crossbar array: the WLD resolution is one-bit, each cell can store two bits, the precision of each feature map data is 2-bit, and the precision of each filter weight is 4-bit. So each 4-bit filter weight is decomposed into two concatenated 2-bit values and thus the weights of each filter span two bit-lines. Similarly, due to limited WLD resolution, the 2-bit input feature map is separated into LSB and MSB groups and fed into wordlines sequentially. In this example, to compute the output neuron for the first input sliding window [1, 2, 3, 1], the data we feed into the input register after decomposition are [1, 0, 1, 1] and [0, 1, 1, 0]; it costs two cycles to get the LSB and MSB part of the output neuron.

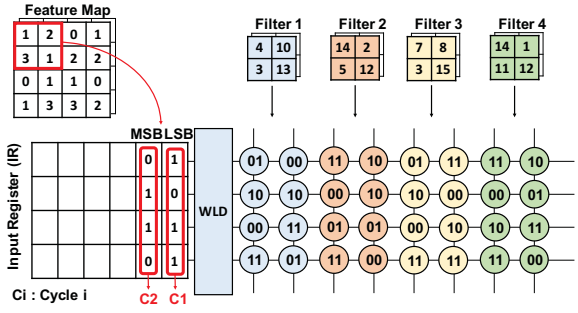

Figure 3: Mapping of filter weights and feature maps to a crossbar array after value decomposition.

2.2 Challenges in Exploiting DNN Sparsity in ReRAM-based DNN Accelerator

Recent studies show that most neural network models have significant amounts of zeros in filter weights and input activations [3]; pruning zero weights and skipping zero activations can help to reduce resource consumption without accuracy loss. In typical deep learning models, about 50% to 70% of input activations are zeros [36], as many neural networks employ the ReLU function, which clamps all negative activation values to zero as their non-linear operator. To further create weight sparsity, algorithmic techniques such as quantization [17], low-rank matrix factorization [12], and ℓ_1-norm regularization [28] have been proposed to prune the network during training. Such high-degree sparsity in input activations and filter weights provides great potential for the underlying deep learning hardware platforms to achieve better performance and energy efficiency. In addition to the sparsity presented in NN models, there exists a finer-level of sparsity granularity to be exploited in ReRAM-based DNN accelerators, bit-level sparsity. As explained in Section 2.1, since a ReRAM cell can store only a limited number of bits, filter weights are decomposed and mapped to multiple bitlines. Similarly, the input vector is decomposed and fed into wordlines using multiple clock cycles, as the WLD has limited resolution. This introduces more opportunities to exploit sparsity for ReRAM-based DNN accelerators. Figure 4(a)(b) shows how the weight (input) sparsity increases as the bits-per-cell (DAC resolution) decreases.

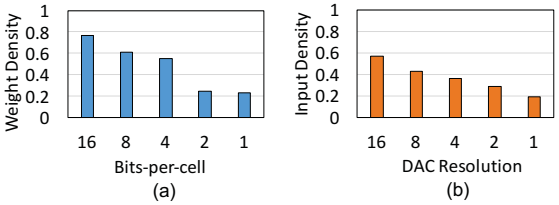

Figure 4: (a) Weight density (fraction of non-zero ReRAM cells) and (b) input density (fraction of non-zero input voltages) of VGG-16 after decomposition.

However, even though there are inherently more ineffectual computations for ReRAM-based DNN accelerators to exploit than digital DNN accelerator designs, the coupled crossbar structure makes it difficult to efficiently exploit sparsity in ReRAM-based DNN accelerators. As we observe from the example in Figure 3, the cell at the 1^{st} wordline and 2^{nd} bitline is zero, but we cannot shift up the rest of the cells of the corresponding bitline (2^{nd} bitline), since the shifted weights would then be multiplied by a wrong input value. Hence for ReRAM-based accelerators, we must find all-zero rows/columns of a crossbar array for compression. To increase the possibility of all-zero rows/columns, Ji et al. [24] propose ReCom, which uses SSL [45] to regularize the distribution of zero weights, so that more all-zero rows and columns can be found. However, with ReCom, there are still many zero weights left in the compressed model. SNrram [44] compresses the model at a finer level, i.e., all-zero filters in structurally compressed neural networks. As the size of a filter is usually smaller than a column, SNrram better exploits sparsity than ReCom [24]. However, this finer granularity comes at the expense of an output indexing module which incurs significant storage overhead.

To exploit activation sparsity, ReRAM-based accelerators also face a new challenge. In contrast to digital accelerators in which the arithmetic unit sequentially computes each individual input value, ReRAM's crossbar array handles the computation of multiple inputs in parallel, as shown in Figure 3. Even though a zero-valued wordline such as the 2^{nd} wordline at the first cycle in Figure 3 does not consume power, a single zero-valued wordline cannot be exploited to reduce execution time. Activation sparsity can be exploited for performance improvement only when all the decomposed input bits fed into the crossbar array in a single cycle are all zeros.

3 A PRACTICAL ReRAM ACCELERATOR ARCHITECTURE

In a practical ReRAM-based DNN accelerator, only a limited number of wordlines and bitlines in a crossbar array can be activated in a single cycle [6, 42, 47]. The maximum number of wordlines that can be turned on concurrently depends on accuracy limitations, while the number of bitlines that can be concurrently activated is constrained by the number of ADCs connected to a crossbar array and the throughput of each ADC. For example, only nine wordlines and eight bitlines are turned on concurrently within a 512×256 crossbar array in a state-of-the-art ReRAM macro designed for DNN acceleration [6].

Turning on a massive number of wordlines concurrently makes it difficult for the ADC to accurately read out the sum-of-product values accumulated on the bitline [31]. As ReRAM cells are non-ideal [5, 31], the per-cell current deviation accumulates on the bitline and leads to overlap with neighboring states (i.e., sum-of-product values) in the accumulated current distribution. The overlap with neighboring states makes it difficult for the ADC with its limited sensing margin to differentiate between different states [6]. Thus, if all of the wordlines in a crossbar array are turned on simultaneously, an incorrect sum-of-products result can be produced, even though none of the cell stores an error value [6, 13, 31]. When too many wordlines are activated concurrently, the sum-of-products errors per bitline degrade the inference accuracy of neural networks.

We use DL-RSIM [31] to analyze the inference accuracy of different neural networks when various numbers of wordlines are activated concurrently[1], as shown in Figure 5. Since oxide-based ReRAM has good electronic properties (high density, low switching energy, and high endurance) [14, 46] and thus is commonly deployed in latest ReRAM-based DNN accelerator studies [6, 42, 47], we use one of the oxide-based ReRAM, WOx ReRAM [22], for our evaluation. We choose the R-ratio and resistance-deviation (σ) of WOx ReRAM [22] as the baseline setting (R_b and σ_b), and analyze the inference accuracy for three different ReRAM cells: the cells with (R-ratio, σ) = (R_b, σ_b), ($2 \times R_b$, $\sigma_b/2$), and ($3 \times R_b$, $\sigma_b/3$). As shown in the figure, the inference accuracy decreases when the number of concurrently activated wordlines increases. Turning on all of the wordlines in a 128×128 crossbar array degrades the inference accuracy to an unacceptable level. Increasing R-ratio and reducing σ can help to improve the inference accuracy by shrinking the overlap with neighboring states in the accumulated current distribution to reduce ADC sensing errors. Nevertheless, even if advances in technology enable the R-ratio/σ to increase/shrink by

[1]The evaluated NN models are described in Section 6.

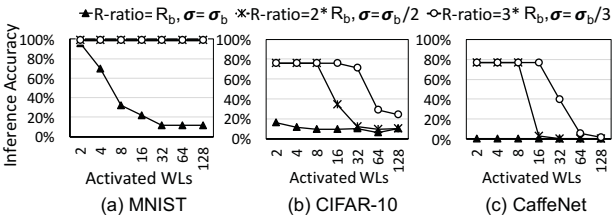

Figure 5: Inference accuracy of (a) MNIST, (b) CIFAR-10, and (c) CaffeNet when various number of wordlines are activated concurrently, with three different types of ReRAM cells. R_b and σ_b are the R-ratio and resistance-deviation of WOx ReRAM [22].

3x of the WOx ReRAM [22], the inference accuracy drops considerably, especially at *CaffeNet*, when more than 16 wordlines are activated concurrently.

In addition to the limitation on the number of concurrently activated wordlines, the number of bitlines that can be turned on simultaneously is constrained by the number of ADCs connected to a crossbar array and ADC's throughput. Since an ADC consumes a significant amount of power and chip area [40], in practical designs multiple bitlines share an ADC [6, 42]. In addition, when considering the area and power constraints of an accelerator chip, it is impractical to deploy higher-frequency ADCs. Thus, in the state-of-the-art ReRAM macro designed for DNN acceleration [6], only a limited number of bitlines can be operated in a single cycle.

Considering the constraints on the number of concurrently activated wordlines and bitlines, a practical ReRAM accelerator can only perform a portion of dot-product computation in a convolution or fully-connected layer within a cycle. We define the maximum amount of dot-product computations that can be performed within a single cycle in a crossbar array as an Operation Unit (OU), with at most S_{WL} wordlines and S_{BL} bitlines activated concurrently. Note that an OU-based architecture does not physically split a large crossbar into smaller ones. It activates a smaller section (OU) of a crossbar array within a single cycle. As shown in Figure 6, an additional wordline activation vector is used to indicate which wordlines should be turned on. Assuming the OU size is 2×2, then only two entries in the wordline activation vector are set to 1. A 2-to-1 multiplexer is connected to each DAC to determine the on/off state of the wordline. When operating the crossbar array in normal memory mode to store synaptic weights in ReRAM cells after offline training, the NN_mode signal is set to 0 and the on/off state of each wordline is specified by the row decoder. During the inference process, the NN_mode signal is set to 1 and the on/off state of each wordline is specified by the wordline activation vector. The multiplexers and the wordline activation vector induce only minimal peripheral overhead.

Figure 7 shows an example of the OU-based dot-product computation for a crossbar array with a 2×2 OU size. Assuming we perform the dot-product computation in the order *OU1*, *OU2*, *OU3*, and *OU4*, the corresponding wordlines are activated in eight different cycles, as marked by *C1* to *C8* in the figure. The dot-product results of different OUs that share the same set of bitlines, such as *OU1* and *OU2*, are added together before the shift-and-add circuit assembles the final result based on the bit position of the input and

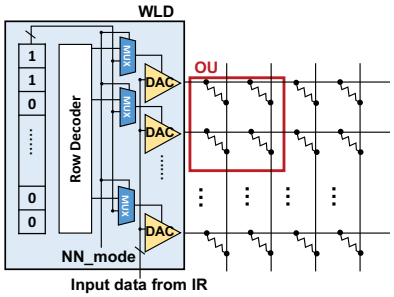

Figure 6: Wordline driver in OU-based ReRAM accelerator.

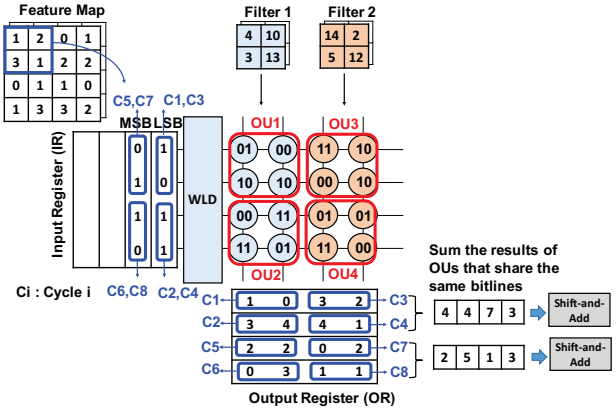

Figure 7: OU-based dot-product computation.

synaptic weight. For example, at the first cycle, the matrix-vector multiplication of *OU1* and the LSB of the first two values in the input sliding window [1, 0] is performed to obtain the output [1, 0]. The matrix-vector multiplication of *OU2* and the LSB of the rest of values in the input sliding window [1, 1] is then performed at the second cycle to yield the output [3, 4]. The output of *OU1* and *OU2* are added together, resulting in the summed output [4, 4] before being assembled by the shift-and-add circuit.

For the example shown in Figure 7, it takes two cycles for an over-idealized design described in Section 2.1 to complete the dot-product computation, while the OU-based design requires eight cycles. However, since fewer wordlines are activated in one cycle, lower-resolution ADCs can be deployed in the OU-based design. Thus, as the sensing speed of an ADC is proportional to the ADC's bit-resolution [38], the ADC sensing time in an OU-based architecture is shorter. Compared to the trivial RC delay (around 10ps for a 100×100 crossbar array [35]), ADC sensing time brings a much larger impact on the operating speed of a ReRAM crossbar array [47]. Hence, the cycle time of a ReRAM-based DNN accelerator is dictated by the slowest ADC sensing stage and the OU-based architecture could achieve shorter cycle time compared with the over-idealized counterpart.

4 NEW OPPORTUNITY FOR EXPLOITING SPARSITY IN OU-BASED ReRAM ACCELERATOR

The OU-based ReRAM accelerator creates new opportunities to exploit DNN sparsity. Unlike the over-idealized design where an entire crossbar array operates in a single cycle, each OU in a crossbar is activated independently. This naturally enables us to exploit a finer granularity of weight and activation compression.

4.1 Weight Compression

There are two ways to exploit weight sparsity: *OU-row compression* and *OU-col compression*. In Figure 8, we use an example with a 4×4 crossbar size and a 2×2 OU size to illustrate these two different types of weight compression, assuming 1-bit WLD resolution and two bits of storage for each ReRAM cell. Figure 8(a) shows the original weight mapping and computation sequence before row compression. The first four weights (marked in green) of Filters 1 and 2 are mapped into the crossbar. The decomposed value of the first four feature map elements (marked in green) in the first input sliding window of the convolution are fetched to the Input Register and sequentially fed into wordlines. It takes four cycles to complete the LSB part of inputs and another four cycles to complete the MSB part, in order to get the dot-product results of the first four output channels (*O1* to *O4*) in this window of convolution.

OU-Row Compression

Row-based compression at the OU level eliminates zero row vectors and shifts the remaining row vectors up. In Figure 8(a), we observe that all the weights of the 2^{nd} row vector in *OU1*, the 1^{st} row vector in *OU2*, the 1^{st} row vector in *OU3*, and the 2^{nd} row vector in *OU4* are zeros. As no crossbar rows are completely composed of zeros, the dot-product computation of these zero weights cannot be skipped in an over-idealized design where the entire crossbar array operates concurrently. With OU-row compression, we can remove these zero rows within each OU and shift the remaining rows up, as shown in Figure 8(b). From this illustration, we see that the input order is different from the original, as some rows of weights are skipped. For *OU1* and *OU2*, the 1^{st}, 4^{th}, 5^{th}, and 6^{th} element (with indexes 0, 3, 4, and 5) of the feature map's input sliding window are to be fetched to the Input Register, while for *OU3* and *OU4*, the 2^{nd}, 3^{rd}, 5^{th}, and 6^{th} element (with indexes 1, 2, 4, and 5) of the feature map's input sliding window are to be fetched. Hence, to support OU-row compression, we need an input index buffer to store input indexes and an input indexing unit to fetch the correct inputs for compressed weights. For every column-wise OU group (e.g., *OU1* and *OU2* belong to the same column-wise OU group), an eDRAM access is required to fetch correct inputs to the Input Register based on the input indexes. Thus, for this example, two eDRAM accesses are required to support OU-row compression, as there are two column-wise OU groups and every column-wise OU group has its own input indexes. Note that as OU-row compression does not change the output channel mapped to each bitline, no output indexing is needed. In this example, the convolution of more feature map elements (6 elements) can be done in 8 cycles when OU-row compression is applied; the skipped computations with zero weights reduce energy consumption.

OU-Col Compression

In a similar fashion, OU-level column-based compression eliminates zero column vectors and shifts the remaining column vectors left. In Figure 8(a), we see that all the weights of the 2^{nd} column vector in *OU1* and the 2^{nd} column vector in *OU4* are zeros. As no crossbar columns are completely composed of zeros, the dot-product computation of these zero weights cannot be skipped in an

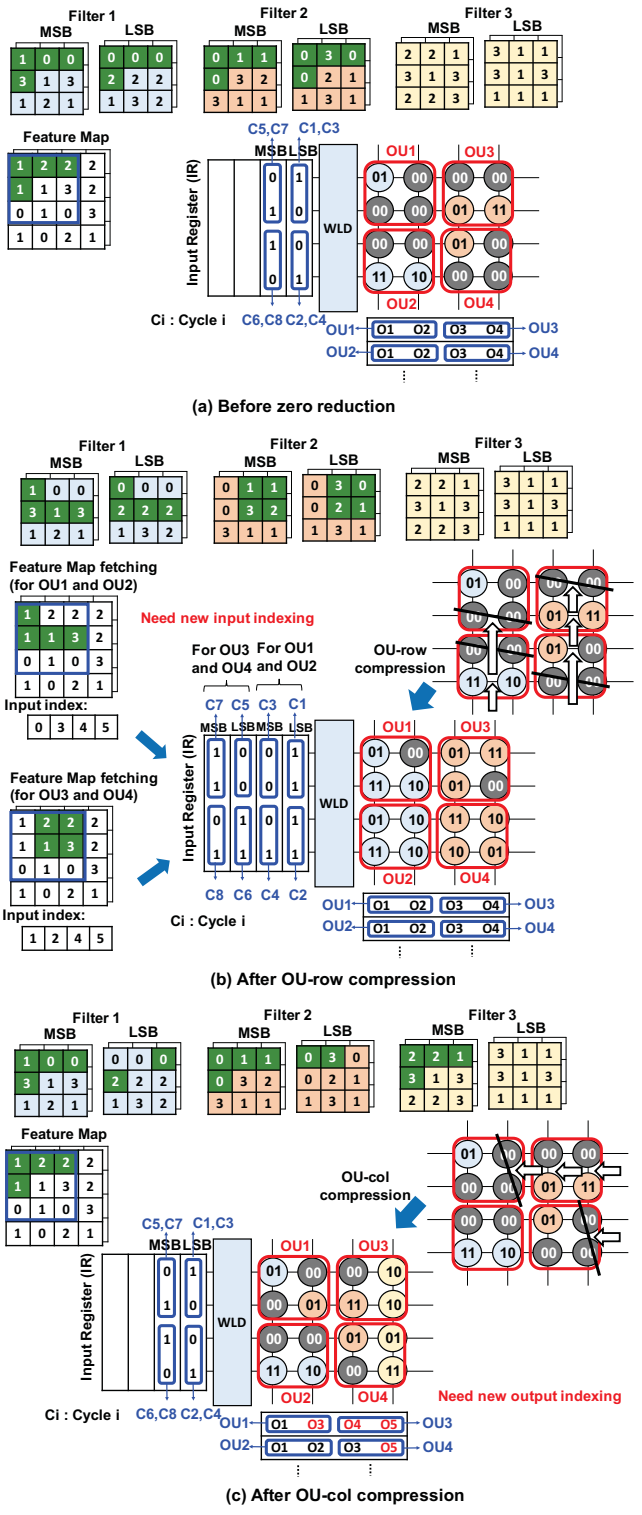

Figure 8: Example of weight compression. (a) Original OU-based computation without zero reduction, (b) OU-row compression, and (c) OU-col compression.

over-idealized design. With OU-col compression, we can remove these zero columns and shift the remaining columns left, as shown in Figure 8(c). From this illustration, we see that the output order is different from the original, as some of the columns of the weights within an OU are skipped. In the original scheme, the mapping between the output current of each bitline and the output channel (*Oi*) is fixed. When OU-col compression is applied, for *OU1*, the output current of the 2^{nd} bitline should be mapped to the 3^{rd} output channel. For *OU3*, the output current of the $1^{st}/2^{nd}$ bitline should be mapped to the $4^{th}/5^{th}$ output channel, while for *OU4*, the output current of the 2^{nd} bitline should be mapped to the 5^{th} output channel. Note that OU-col compression does not change the input order. In this example, in addition to Filters 1 and 2, the convolution with the MSB part of the first four elements of Filter 3 can also be done within 8 cycles when OU-col compression is applied; the skipped computations with zero weights reduce energy consumption.

4.2 Activation Compression

One naive way to exploit activation sparsity for the OU-based ReRAM architecture is to skip an OU computation when the inputs concurrently feed into all the wordlines of the OU are zeros. To exploit activation sparsity further, we propose a method called Dynamic OU Formation (DOF). In the example shown in Figure 8(a), we observe that the input bits feed into the 2^{nd} and 3^{rd} wordlines are zeros when we first compute the LSB part of inputs. The computation associated with these two wordlines cannot be skipped in the aforementioned naive method, as these two wordlines belong to different OUs. The idea of DOF is to activate the 1^{st} and 4^{th} wordlines together in the same cycle for computing the LSB part of inputs, as shown in Figure 9. That is, a virtual OU execution unit is formed dynamically. With DOF, we can skip the 2^{nd} and 3^{rd} wordlines for the computation of LSB part of inputs, and also the 1^{st} and 4^{th} wordlines for the MSB part of inputs. As a result, we need only 4 cycles instead of 8 cycles to complete the convolution in this example; the skipped computations save on energy use. Note that DOF does not change the output channel associated with each bitline; every activated wordline within a dynamically formed OU must follow the same output indexing to guarantee correctness.

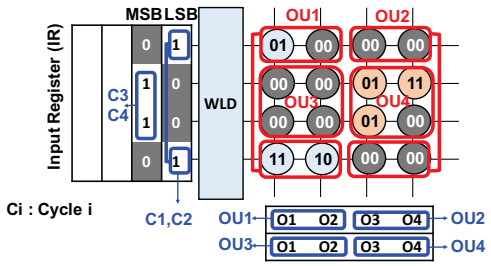

Figure 9: Dynamic OU Formation.

To jointly exploit DOF and weight compression, we must adopt row-wise compression for weights, as row-wise compression does not change the bitline-to-output-channel mapping. With column compression, the same bitline shared by different OU blocks may be mapped to different output channels. For the example shown in Figure 8(c), after column compression, the output current of the 2^{nd}

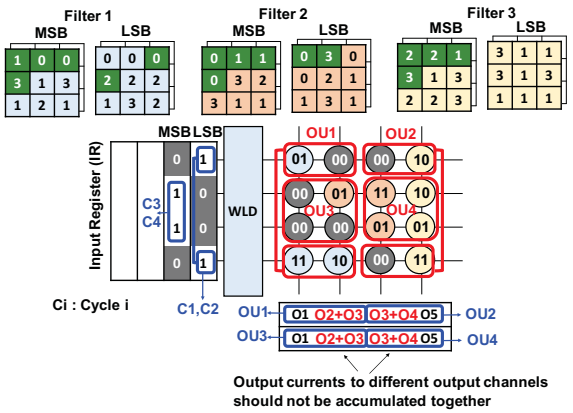

Figure 10: Wrong output when combining DOF with OU-col compression.

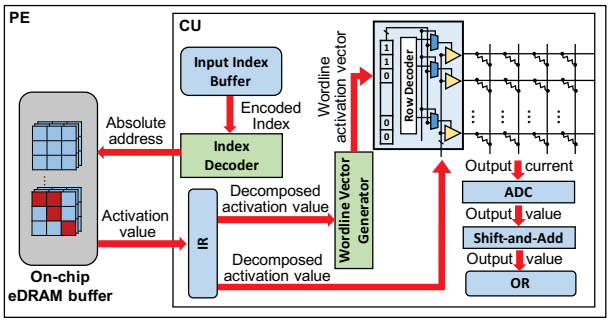

Figure 11: Sparse ReRAM Engine.

bitline in *OU1* and *OU2* is mapped to the 3^{rd} and 2^{nd} output channel respectively, and the output current of the 1^{st} bitline in *OU3* and *OU4* is mapped to the 4^{th} and 3^{rd} output channel respectively. If we attempt to apply DOF, as shown in Figure 10, the partial sum of the dynamically formed OU may accumulate the currents associated with the convolution of different filters and thus the value of the output channel may be wrong. For example, in Figure 10, the 2^{nd} bitline of the dynamically formed *OU1* wrongly accumulates the output currents that should be mapped to the 2^{nd} and 3^{rd} output channels.

5 SPARSE ReRAM ENGINE

In this section, we present the Sparse ReRAM Engine (SRE), in which we use the techniques described in Section 4 (OU-Row Compression + Dynamic OU Formation) for exploiting both weight and activation sparsity. Figure 11 shows the architecture and dataflow of the proposed Sparse ReRAM Engine inside a PE. The weight matrices are first compressed offline using OU-based row compression (ORC) and the corresponding input indexing information is also generated. In each CU, the Input Index Buffer is used to store the input indexing information for the filter weights mapped to this CU. To reduce the storage overhead for input indexes, we store the index difference instead of the absolute values, similar to the approach in [49]. The Wordline Vector Generator produces the wordline activation vector at each cycle to support Dynamic OU Formation. Below we describe the Index Decoder, the Wordline Vector Generator, and the SRE pipeline in details.

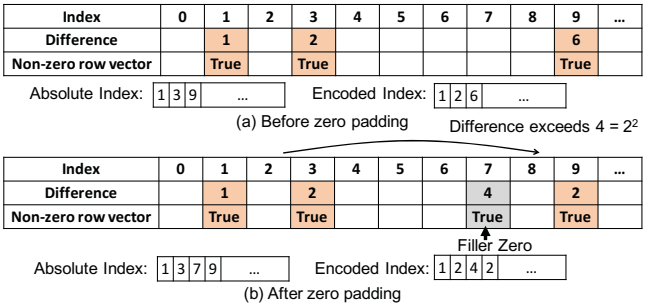

Figure 12: Input index encoding.

5.1 Index Decoder

To reduce the index storage overhead, we store index differences instead of absolute values. Figure 12 shows an example: in (a), we see that the non-zero row vectors correspond to index values 1, 3 and 9, and so on. Thus, instead of storing 1, 3, and 9 in the Input Index Buffer, we store the index differences: 1, 2 and 6. Hence one encoded index of an input address requires $\log_2 max\_dist$ bits, where max_dist is the maximal difference between two non-zero row vectors in the target model. Thus the index storage overhead varies for different models. To bound the storage overhead, we adopt zero-padding [17], in which filler zeros are added if the index difference exceeds the largest unsigned number that can be represented with the target number of index bits. In the example shown in Figure 12, if we limit the number of index bits to 2, the third encoded index value is 6, which exceeds the bound. In this case, as shown in Figure 12(b), we insert a zero row at index 7 in the compressed weight matrices. As doing so reduces the index storage overhead but also affects the weight compression ratio, the index length (target number of index bits) should be chosen carefully, considering this tradeoff for each model.

The decoding procedure for encoded index values uses prefix sum operations, as illustrated in Figure 13. The Index Decoder is an implementation of Hillis and Steele's algorithm [21] for parallel prefix sum, as shown in Figure 14. The decoder width depends on the required decoding throughput. In Section 5.3, we discuss this issue further.

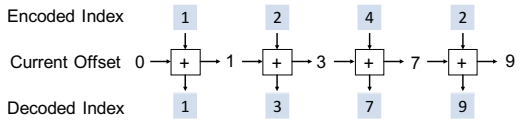

Figure 13: Input index decoding.

5.2 Dynamic OU Formation

Figure 15 shows the required hardware support for Dynamic OU Formation. The Wordline Vector Generator module decides which wordlines should be activated within a cycle to dynamically form a virtual OU execution unit. The example illustrated in Figure 15 assumes 1-bit DAC resolution, 8 wordlines in a crossbar, and a 2×2 OU size. The decomposed input values (1-bit values in this example) are passed to the mask vector to mark non-zero inputs[2]. Then a

[2]If DAC resolution is larger than 1 bit, the decomposed input values are passed through logic circuits that are able to mark non-zero inputs to form the mask vector.

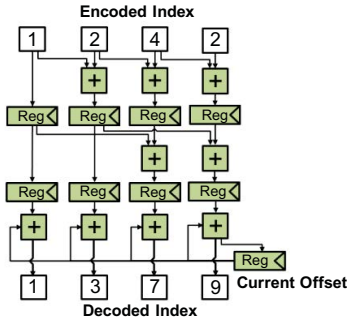

Figure 14: Circuit implementation of parallel prefix sum.

prefix sum operation is performed on this mask vector. Thus the value in the i-th element of the prefix-sum vector indicates the number of non-zero inputs between wordline 0 and wordline i. In this example, we need to find two wordlines with non-zero inputs at every cycle and mark the entries of these two wordlines in the wordline activation vector. To achieve this goal, we use the prefix-sum vector, the mask vector, and a set of comparators to perform a condition check. At cycle c, the wordlines that need to be activated are wordline j, which satisfies

$$(1 + (c-1)S_{WL} \leq Prefix\_sum[j] < 1 + c \cdot S_{WL}) \ \& \ mask\_vector[j],$$

where S_{WL} is the number of wordlines in an OU (S_{WL}=2 in this example). To implement this condition check, counters L and H are set initially to 1 and 1+S_{WL}, and incremented by S_{WL} every cycle.

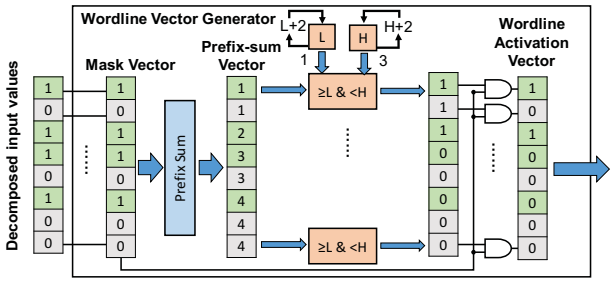

Figure 15: Wordline Vector Generator.

5.3 SRE Pipeline

Figure 16 shows the pipeline diagram of the SRE engine. Each OU computation is completed in one cycle and the outputs of the OU computation are latched in the S&H circuit. In the next cycle, these outputs are fed to the ADC unit. The result of the ADC is then fed to the shift-and-add unit (S+A), where the result is assembled with the data stored in the output register (OR) based on the bit position of the inputs and synaptic weights. These three stages (OU, ADC, and S+A OR Wr) are executed in a pipelined manner to produce the convolution result of each output neuron.

Before each OU computation, we must fetch the associated inputs and activate the corresponding wordlines. To support ORC and DOF, for a crossbar array with n wordlines, we must fetch n inputs (termed a *batch*) at a time from the on-chip eDRAM buffer to the input register (IR), based on the indexes decoded by the Index Decoder. The Wordline Vector Generator (WL Vec Gen) then generates one wordline activation vector from the batch at each cycle to specify which wordlines should be concurrently activated in the

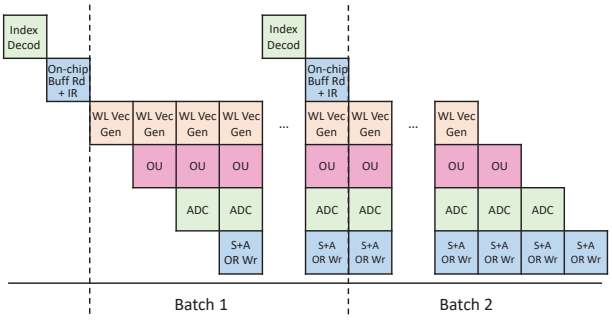

Figure 16: Pipelined execution in SRE.

next cycle to perform the OU computation. Except for the first batch of input data, index decoding (Index Decod) and input data fetching (On-chip Buffer Rd + IR) can be performed concurrently with the OU computation, as shown in Figure 16. The number of cycles required to process a batch of inputs depends on the results of the Dynamic OU Formation. In the extreme case, no OU computation is required for one input batch if all of the input values in the batch are zeros. Thus, to minimize pipeline stalls, index decoding must be completed within one cycle; likewise for input data fetching.

The cycle time of the SRE pipeline is dictated by the slowest stage, which is the sensing of the OU computation result (ADC stage). In the 65nm ReRAM macro [6] with 3-bit sensing resolution, the clock cycle time is 15.6 ns. Assume a 128×128 crossbar with a 16×16 OU, 2-bit ReRAM cells, and 16-bit feature map values: we need a 6-bit ADC to read out the OU computation result, with a cycle time of approximately 30 ns, as the sensing speed of an ADC is proportional to the ADC's bit-resolution [38]. To minimize pipeline stalls, the Index Decoder must decode 128 inputs in 30 ns. Similarly, the Wordline Vector Generator must also generate one wordline activation vector every 30 ns. We implement the Index Decoder and the Wordline Vector Generator in Verilog and use Synopsys Design Compiler to synthesize latency, area, and power. Based on our synthesis results, the width (parallelism degree) of the Index Decoder and the Wordline Vector Generator are both set to eight, to meet throughput requirements while minimizing the area and power consumption. Note that it is also possible to design the eDRAM buffer (8 banks and 512 bits bus width) based on the access latency modeled by CACTI [34] to ensure that fetching 128 16-bit inputs for each crossbar array can be completed in one cycle.

6 EVALUATION METHODOLOGY

We implement a custom cycle-accurate simulator written in Python to evaluate the performance and energy consumption of the Sparse ReRAM Engine. Our simulator models the mapping and execution flow of sparse neural networks on ReRAM crossbar arrays. Table 1 shows the hardware configuration of each PE for our simulation. Except for the OU-related components (ADC and eDRAM buffer), we use the hardware configuration from ISAAC [40], a state-of-the-art over-idealized design. The power consumption of all the memories, including the eDRAM buffer, IR, and OR, is modeled using CACTI [34] at 32nm process assumed in ISAAC [40]. Each PE has a 64KB on-chip eDRAM buffer to store intermediate input and output feature maps. The eDRAM buffer is configured to ensure that fetching a batch of input data could be completed in one cycle. There are 12 CUs within each PE; each CU has 8 crossbar arrays.

Table 1: Hardware configuration.

PE configuration (1.2 GHz, 32nm process, 168 PEs per chip)		
Component	**Spec**	**Power**
eDRAM Buffer	size: 64KB; banks: 2 bus width: 512 bits	29 mW (leakage: 0.38 mW)
eDRAM-to-CU bus	number of wires: 384	7 mW
Router	flit size: 32; number of ports: 8 (shared by 4 PEs)	42 mW
Sigmoid	number: 2	0.52 mW
S+A	number: 1	0.05 mW
MaxPool	number: 1	0.4 mW
OR	size: 3KB	1.68 mW (leakage: 0.21 mW)
CU configuration (12 CUs per PE)		
Component	**Spec**	**Power**
ADC	number: 8; resolution: 6 bits frequency: 1.2GSps	5.14 mW
DAC	number: 8 x 128; resolution: 1 bit	4 mW
S+H	number: 8 x 128	10 μW
Memristor Array	number: 8; size: 128 × 128 bits-per-cell: 2; OU size: 16 × 16	2.4 mW (4.7 μW per OU)
S+A	number: 4	0.2 mW
IR	size: 2KB	1.24 mW (leakage: 0.42 mW)
OR	size: 256B	0.23 mW (leakage: 0.05 mW)

The crossbar array size is set to 128×128, and each ReRAM cell can store two bits. Anticipating future improvements in cell reliability, we set the OU size to 16×16[3]. Since a 6-bit ADC is sufficient for a 16×16 OU, we use the same style of ADC as in ISAAC [40] but follow the equation in [38] to scale the ADC power consumption for the lower bit resolution. For the analysis of indexing overhead, we implement the Index Decoder and Wordline Vector Generator in Verilog and synthesize using the Synopsys Design Compiler under TSMC 28nm process[4]. The obtained power and area are scaled up to 32nm process.

Workloads

We evaluate the proposed Sparse ReRAM Engine on three datasets: MNIST [27], CIFAR-10 [25], and ImageNet [11]. MNIST and CIFAR-10 are small-scale datasets, whereas ImageNet is a large-scale dataset. In our evaluation, we use the ILSVRC 2012, a subset of ImageNet with approximately 1000 categories, each of which includes 1000 images. The neural networks used in our evaluation are LeNet [27] on MNIST, a CNN with three convolution layers and two fully-connected layers on CIFAR-10, and four large-scale CNNs (*CaffeNet* [26], *VGG-16* [41], *GoogLeNet* [43], and *ResNet-50* [19]) on ImageNet. Table 2 lists the network topology of these evaluated NN models. These models are all trained with the SSL pruning algorithm [45]. CaffeNet and VGG-16 are released by [45] and thereby were well trained for structural sparsity. Thus, in the results shown in Section 7, *CaffeNet* and *VGG-16* show higher gains from OU-based row compression than the other models, which are trained by ourselves and are not well tuned for structural sparsity as *CaffeNet* and *VGG-16*. These NN models cover the test cases with a broad range of weight and activation sparsity. Note that the weight sparsity and activation sparsity listed in Table 2 only represents the fraction of zero values in the synaptic weights and feature maps of the target NN model before bit-level decomposition. The amount of sparsity that can be exploited to improve performance and energy efficiency depends on ReRAM bits-per-cell, DAC resolution, and the applied sparsity exploration techniques.

[3]Our motivation experiment in Figure 5 shows that for large-scale NN models, inference accuracy drops if more than 16 wordlines are activated concurrently, even when technology improvements enable the R-ratio/resistance-deviation to increase/shrink by 3x of the WOx ReRAM [22].

[4]We synthesize the circuits under 28nm process as we are only authorized to access TSMC 28nm standard cell library.

Table 2: NN topology of evaluated benchmarks.

Name	**Weight Sparsity**	**Activation Sparsity**	**Structure of networks**
MNIST	42%	28%	conv5x20-pool-conv5x50-pool-500-10
CIFAR-10	34%	22%	conv5x32-pool-conv5x32-pool-conv5x64-pool -64-10
CaffeNet	91%	21%	conv11x96-conv5x256-conv3x384-conv3x384 -conv3x256-4096-4096-1000
VGG-16	95%	41%	conv3x64-conv3x64-pool-conv3x128-conv3x128 -pool-conv3x256-conv3x256-conv3x256-pool -conv3x512-conv3x512-conv3x512-pool-conv3x512 -conv3x512-conv3x512-pool-4096-4096-1000
GoogLeNet	79%	37%	conv7x64-pool-conv3x192-pool-inception(3a) -inception(3b)-pool-inception(4a)-inception(4b) -inception(4c)-inception(4d)-inception(4e) -pool-inception(5a)-inception(5b)-pool-1000
ResNet-50	81%	46%	conv7x64-pool-[conv1x64-conv3x64-conv1x256]x3 -[conv1x128-conv3x128-conv1x512]x4-[conv1x256 -conv3x256-conv1x1024]x6-[conv1x512-conv3x512 -conv1x2048]x3-pool-1000

Comparison Baselines

As the baseline, we use an OU-based ReRAM accelerator that does not exploit any sparsity. In addition, we compare the performance and energy efficiency of the proposed SRE with naive crossbar row-based compression. We also compare it with ReCom [24], a weight matrix row-based compression method designed for neural networks pruned by SSL [45]. When ReCom is applied, if the same pixel of each filter in the same convolution/fully-connected layer is all zeros, the corresponding OU rows are removed to reduce unnecessary computations. Note that we does not compare with SNrram [44], as SNrram uses model-based compression and its crossbar architecture is highly model-dependent. A quantitative performance comparison with SNrram would be difficult because we use a different baseline design from SNrram, and the SNrram paper provides no cycle time information.

For our SRE, we evaluate three different modes: ORC, DOF, and ORC+DOF. ORC adopts only OU-based row compression to exploit weight sparsity. DOF exploits only activation sparsity by dynamically skipping the activation of wordlines with zero input signals. ORC+DOF combines ORC and DOF to jointly exploit both weight and activation sparsity. To bound the storage overhead of input indexing, we adopt zero-padding [17] and choose the minimum number of index bits that ensured less than 10% loss in weight compression ratio compared to when zero-padding is not used. Based on this principle, the length of index bits for MNIST, CIFAR-10, *CaffeNet*, *VGG-16*, *GoogLeNet*, and *ResNet-50* is set to 5, 5, 5, 5, 3, and 3 bits, respectively.

7 EXPERIMENTAL RESULTS

In this section, we first evaluate the performance and energy efficiency of the proposed Sparse ReRAM Engine (SRE for short). We then analyze the indexing overhead of the scheme, followed by sensitivity studies on the OU size and ReRAM bits-per-cell. We also evaluate the performance of SRE when running non-SSL sparse neural networks. Finally, we compare SRE with an over-idealized design [40] to show that jointly exploiting weight and activation sparsity enables a practical ReRAM-based DNN accelerator to achieve comparable performance with substantial energy savings.

7.1 Performance and Energy

Figure 17 shows the performance speedup of different designs against the baseline OU-based ReRAM accelerator that does not

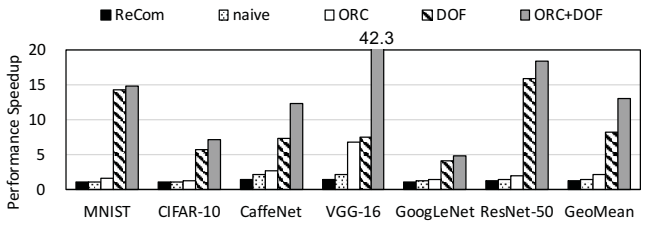

Figure 17: Performance speedup of different sparsity-exploration approaches over the baseline.

exploit any sparsity. With ORC, the performance speedup ranges from 1.3× to 6.8×. As mentioned earlier, *CaffeNet* and *VGG-16* are well tuned for structural sparsity, so they obtain higher benefit from ORC than the other models, 2.6× and 6.8×, respectively. We expect to see similar benefits for the other models if they could be trained well for structural sparsity as *CaffeNet* and *VGG-16*. On the other hand, DOF delivers significant performance gains, ranging from 4.1 × to 16.0×, for all the models. *ResNet-50* obtains the largest gain from DOF as there are many batch norm layers in this model. The joint use of ORC and DOF provides accumulated benefit (average 13.1×). Since zero bits are usually randomly distributed in feature maps, applying ORC rarely degrades the fraction of activation sparsity that can be explored. For example, *VGG-16* sees 6.8×, 7.5× and 42.3× speedup with ORC, DOF and ORC+DOF, respectively. Compared to SRE, both ReCom and naive provide only small performance speedups over the baseline, as ReCom and naive explore weight sparsity only in coarser-grained weight-matrix-row and crossbar-row granularity. The speedup for naive is slightly higher than that for ReCom, since a weight-matrix-row can span multiple crossbar arrays and ReCom cannot remove an all-zero crossbar-row if the row is just part of a non-all-zero weight-matrix-row.

Figure 18 shows the energy consumption of different sparsity-exploration designs normalized to the baseline. ReCom and naive help to conserve a small amount of energy (average 12.5% and 21.3%) by skipping computations with all-zero crossbar-rows. Compared to ReCom and naive, ORC saves more energy by exploiting a finer granularity of weight sparsity to skip computations with all-zero OU rows (50.6% on average). The energy savings come from the reduced number of accesses to peripheral circuits/buffers (ADC, IR, and OR) associated with each OU computation. DOF shows high effectiveness for energy reduction across all NN models (between 70% to 90%). Combining ORC and DOF reduces energy further for *CaffeNet* and *VGG-16* but not for the other four models. We can see from Figure 18, ORC+DOF consumes significantly higher eDRAM energy than DOF, which outweighs the additional computation-related energy savings from ORC. As we discuss in Section 4, since row-wise compression changes the input order, each column-wise OU group must fetch correct inputs from the eDRAM buffer based on its own input indexes. Hence, compared to DOF that does not change the input order, additional eDRAM accesses are required when row-wise compression is applied. As mentioned earlier, *CaffeNet* and *VGG-16* are well trained for structural sparsity so the additional reduction on computation-related energy from ORC can compensate the loss due to extra eDRAM accesses. We expect that ORC+DOF in general should achieve the largest energy savings if NN models could be fine-tuned for structural pruning.

7.2 Indexing Overhead Analysis

To support OU-based row compression in SRE, we must store the input indexes of each column-wise OU group. To bound the storage overhead, we store the index difference between non-zero OU rows instead of the absolute values and adopt zero-padding [17]. Figure 19 shows the storage overhead for input indexing. The overhead varies for different NN models, depending on the size of the model. For large-scale NN models with many convolution layers such as *ResNet-50*, the storage overhead is larger. The OU size also affects the amount of storage overhead. With a smaller OU size, we can remove more all-zero OU rows. Thus, for each column-wise OU group, fewer input indexes need to be stored. Nevertheless, as the OU size decreases, the number of column-wise OU groups increases. As a result, more sets of input indexes must be stored for smaller OUs, leading to greater storage overhead. From our evaluation, we find that the storage overhead increases only slightly when the OU size decreases for most of the evaluated neural networks, except for *ResNet-50*. Even though the storage overhead of *ResNet-50* is higher (778KB) than other NN models, storing the index difference between non-zero OU rows still leads to far lower storage overhead than directly storing the absolute index of each non-zero OU row (about 4MB).

In SRE, to support ORC and DOF, we add the Index Decoder and Wordline Vector Generator components. As the crossbar array has 128 wordlines, the Index Decoder must decode 128 input indexes in one cycle to minimize pipeline stalls. Based on our synthesis, an Index Decoder that can decode 8 indexes at a time in a parallel fashion (width = 8) is required to provide sufficient throughput. The Index Decoder is composed of seven 5-bit adders, six 6-bit adders, four 7-bit adders, eight 13-bit adders, eight 6-bit latches, eight 7-bit latches, eight 8-bit latches, and one 13-bit latch, inducing only a small area (0.001 mm^2) and power (1.24 mW) overhead. Note that the overhead is independent of the OU size, as the Index Decoder must decode the input indexes for all the wordlines in the crossbar array at a time. To support DOF without seriously stalling the pipeline, the Wordline Vector Generator must be able to generate a wordline activation vector within one cycle. Our synthesis result shows that a Wordline Vector Generator that can generate 8 elements of the wordline activation vector at a time in a parallel fashion (width = 8) can provide sufficient throughput. The Wordline Vector Generator is composed of a circuit implementation of parallel prefix sum (with four 1-bit adders, four 2-bit adders, four 3-bit adders, and eight 8-bit adders) and thirty-two 4-bit comparators. The circuit is simple and induces only a 0.001 mm^2 area and 0.86 mW power overhead.

7.3 Sensitivity Studies

We evaluate SRE with different configurations of crossbar architecture, including the OU size and ReRAM bits-per-cell, to analyze the impact of these architectural parameters on the weight compression ratio, performance, and energy consumption.

OU Size

In SRE, the OU size impacts the weight compression ratio and the amount of computation reduction opportunities. Figure 20 shows that as the OU size decreases, the weight compression ratio increases, since it is easier to find all-zero OU rows when the OU is smaller. With a smaller OU, the weight compression ratio is comparable to the ideal case which assumes that all ReRAM cells with

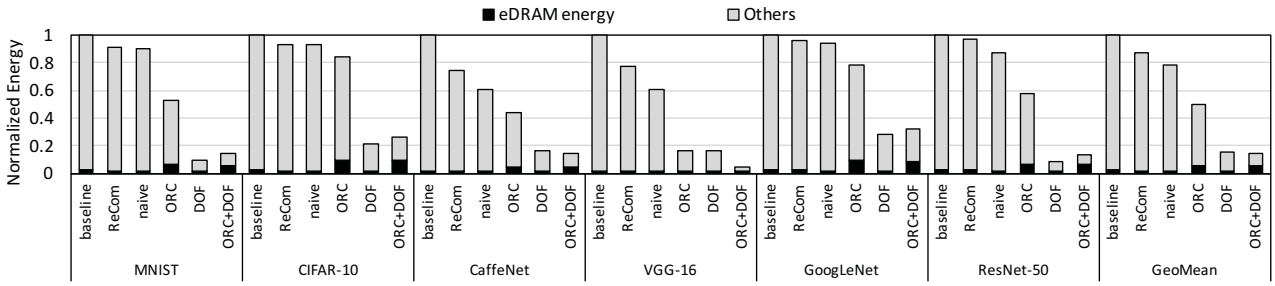

Figure 18: Energy consumption of different sparsity-exploration approaches normalized to the baseline.

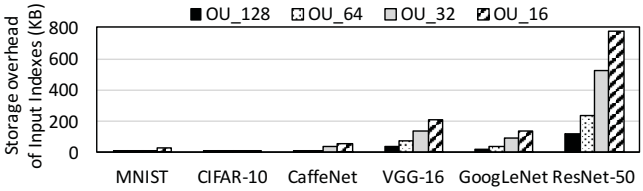

Figure 19: Storage overhead of input indexes for SRE with different OU sizes (from 128×128 to 16×16).

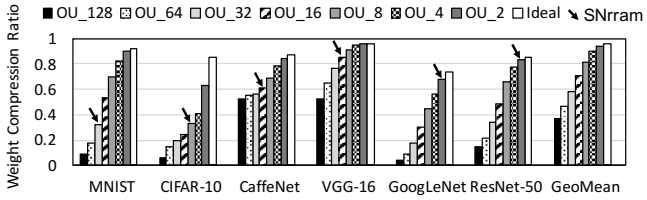

Figure 20: Weight compression ratio of SRE with different OU sizes (from 128×128 to 2×2).

zero weights can be removed. In Figure 20, we also use arrows to indicate the weight compression ratio that can be obtained from SNrram [44]. As SNrram uses model-based compression (i.e., fine-grained column-based compression that removes all-zero column vectors with size equals to filter height × filter width) to exploit weight sparsity, its weight compression ratio varies across different neural network models. Although SNrram yields a high compression ratio for neural network models with small filter sizes such as *GoogLeNet* and *ResNet-50*, it is not a practical design as the output indexing overhead is large and it requires different sizes of crossbar arrays for layers with different filter sizes. With proper OU size settings, the proposed SRE achieves a weight compression ratio similar to that of SNrram.

The energy consumption of SRE also varies for different OU sizes. In the baseline OU-based ReRAM accelerator that does not exploit any sparsity, lower-resolution ADCs with less energy consumption can be utilized when the OU is small. Nevertheless, with a smaller OU, more OU computations are needed to complete the same amount of matrix-vector multiplication, resulting in a higher number of peripheral circuit/buffer (ADC, IR, and OR) accesses. Thus, energy consumption dramatically increases when the OU size decreases in the baseline, as shown in Figure 21(a). After applying ORC+DOF to exploit sparsity, smaller OUs do not necessarily result in higher energy consumption, as shown in Figure 21(b). With a smaller OU size, it is easier to find all-zero OU rows and save unnecessary computations. As a result, for most of the evaluated neural networks, the energy consumption decreases when the OU

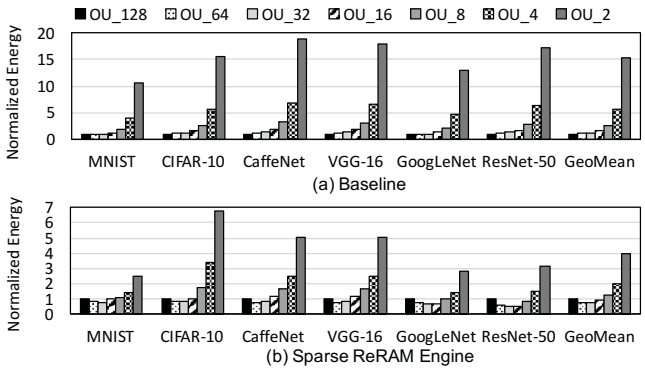

Figure 21: Energy consumption of (a) baseline and (b) SRE for different OU sizes, normalized to 128×128 OU size.

size decreases from 128×128 to 32×32. In this paper, we set the OU size to 16×16 considering the impact on inference accuracy. Results in Figure 21(b) indicates that even if advances in technology make it possible to achieve sustainable accuracy when operating the entire crossbar array in a single cycle, it is more energy-efficient to jointly exploit weight and activation sparsity on OU-based architecture with appropriate OU sizes.

ReRAM Bits-per-cell

In the practical ReRAM crossbar array, filter weights are decomposed and mapped to multiple bitlines, as each ReRAM cell stores only a limited number of bits. When the ReRAM bits-per-cell decreases, more cycles are needed to complete the same amount of matrix-vector multiplication if weight sparsity is not explored. Nevertheless, due to the the weight decomposition process, when ReRAM bits-per-cell decreases, bit-level weight sparsity increases, and it is easier to find all-zero OU rows. As a result, the SRE performance speedup is greater when each ReRAM cell stores a fewer number of bits, as shown in Figure 22. Although advances in technology may enable a ReRAM cell to store higher number of bits with sustainable reliability in the next few years, SRE still provides 11.4× performance speedups on average if the storage capacity of each ReRAM cell is 8 bits.

7.4 Non-SSL Sparse Neural Networks

Even though the structural pruning algorithm (SSL) [45] helps to regularize the distribution of zero weights, the proposed SRE provides performance speedup not only for SSL-trained neural networks but also for non-SSL sparse neural network models, as shown in Figure 23(a). For such non-SSL sparse neural network models, we use the NN models released by SkimCaffe [37]. SkimCaffe adopts

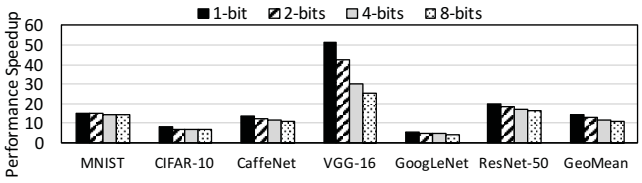

Figure 22: Performance speedup of SRE over baseline for various ReRAM bits-per-cell settings.

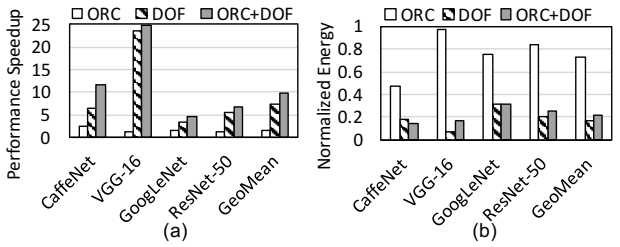

Figure 23: (a) Performance speedup and (b) energy consumption of SRE normalized to the baseline for non-SSL sparse neural networks.

Guided Sparsity Learning (GSL), a generic algorithm that supports different regularization methods and prunes neural networks by adjusting sparsity targets precisely at different layers. When SSL is not applied during training, weight sparsity may decrease and it can be harder to find all-zero OU rows. Thus, ORC yields only small performance speedups. For example, for *VGG-16*, ORC provides a 6.8× performance speedup when SSL is applied (Figure 17) but only a 1.1× speedup when a non-SSL pruning algorithm is applied (Figure 23(a)). In contrast, activation sparsity is less relevant to the weight pruning algorithm. Therefore, DOF still provides significant performance speedups for non-SSL sparse neural network models. By jointly exploiting weight and activation sparsity, the proposed SRE provides a 9.7× performance speedup on average for non-SSL sparse neural networks.

SRE can also provide significant energy savings for non-SSL sparse neural networks, as shown in Figure 23(b). Although ORC provides fewer energy savings for non-SSL sparse neural networks (average 26.7%) compared to SSL neural networks (average 50.6% as shown in Figure 18), DOF still helps to greatly reduce the energy consumption. As fewer weight sparsity could be explored by the non-SSL pruning algorithm, the energy consumption of ORC+DOF is slightly higher than that of DOF for neural networks such as *VGG-16*. For *VGG-16* trained by non-SSL pruning algorithm, ORC+DOF consumes significantly higher eDRAM energy than DOF, outweighing the additional computation-related energy savings from ORC. On average, for non-SLL sparse neural networks, ORC+DOF still yields 78.7% energy savings over the baseline.

7.5 Comparison with Over-Idealized Design

ISAAC [40] is a state-of-the-art but over-idealized design, which overlooks the inference accuracy loss caused by the accumulated effect of per-cell current deviation and assumes that 128 wordlines in a crossbar array can be activated concurrently. As discussed in Section 3, a practical OU-based deign completes less computation in a cycle but has the advantage of shorter cycle time. For the first-order comparison of a 16×16 OU-based design (15-ns cycle time [5]) with ISAAC assuming 128×128 crossbar arrays (100-ns cycle time), the OU-based design is 9.6× slower (64× cycles and 6.6× faster cycle time). However, as what we have presented in this paper, the OU-based design enables a new opportunity to exploit sparsity for further performance and energy improvements. So in this subsection, we demonstrate that with the proposed joint weight and activation sparsity exploration method, the practical OU-based architecture, which achieves satisfactory inference accuracy considering the limitation of ReRAM cell reliability, could actually deliver comparable performance and energy efficiency with the over-idealized design, ISAAC.

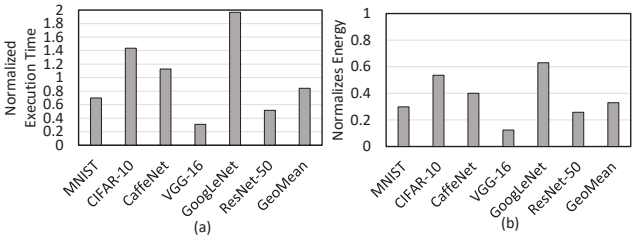

Figure 24: (a) Execution time and (b) energy consumption of SRE normalized to ISAAC.

Figure 24(a) shows the execution time of our SRE normalized to ISAAC's execution time. For fair comparison, we apply ReCom [24] to exploit weight sparsity for ISAAC. We can observe that SRE achieves better performance than the over-idealized ISAAC for 3 out of 6 neural network models. For neural networks with considerable increase in weight compression ratio on OU-based architecture than on ISAAC, SRE delivers higher performance speedup. On average, SRE provides 15.8% performance improvement over ISAAC. In the aspect of energy efficiency, for all of the evaluated neural network models, SRE is more energy efficient than ISAAC, as shown in Figure 24(b). Without exploiting sparsity, the OU-based architecture consumes roughly 2.5× energy than ISAAC due to the combined effect of a higher number of accesses to peripheral circuits/buffers (ADC, IR, and OR) and the lower-resolution (6-bit) ADC (vs. 8-bit ADC). With the proposed scheme, SRE provides 67.0% energy savings over ISAAC on average. The energy savings are considerable especially for neural network models with a large amount of sparsity such as *VGG-16* (87.6%).

8 RELATED WORK

Pruning Algorithms

Various algorithmic techniques have been studied to remove the redundancy inside neural network models. For instance, quantization [17], low-rank matrix factorization [12], and ℓ_1-norm regularization [28] have been proposed to prune networks during training. Han et al. [18] propose another approach to directly remove low-value weights and retrain the network, but most of the computation reduction is attained in fully-connected layers. As the majority of computations are at convolution layers, Molchanov et al. [33] develop a new formulation based on Taylor expansion to iteratively

[5] As described in Section 5, the cycle time of SRE is 30ns at 65nm technology. For an apples-to-apples comparison with ISAAC, the cycle time of SRE is scaled down from 65nm to 32nm, and is approximately 15 ns.

remove the least important parameters in convolution layers. Several studies focus on adapting the sparse network structures to make it hardware-friendly in an algorithmic way [29, 45, 48]. Yu et al. [48] propose a method to customize DNN pruning for different hardware platforms based on each platform's data-parallelism. Wei el al. [45] regularize the structure of DNN and prune weights in a systematic way to derive a hardware-friendly compressed neural network. Liang et al. [29] propose a crossbar-grain pruning algorithm to remove an entire crossbar whose partial sum contributes less to its output. These weight pruning algorithms can be utilized to increase the speedup/energy-savings of DNN inference.

Sparse DNN Acceleration in Digital ASICs and GPUs

Many prior studies have designed different mechanisms to exploit weight sparsity [49], activation sparsity [1, 8], or both [16, 36] in CMOS-based digital accelerators in order to improve the energy efficiency of DNN inference. To exploit weight sparsity, Zhang et al. [49] propose Cambricon-X, which retains only nonzero weights in its internal buffers and uses an indexing module to efficiently fetch needed inputs for computation. However, it still wastes times computing multiplications for zero-valued activations. To exploit activation sparsity, Chen et al. [8] propose Eyeriss to gate the multiplier when an input activation is zero to save energy. Another sparse DNN accelerator, Cnvlutin [1], selects only non-zero activation values for delivery as multiplier operands. Nevertheless, neither Eyeriss nor Cnvlutin skips computations with zero weights. To jointly exploit weight and activation sparsity, EIE [16] uses a compressed representation for both activations and weights, and only delivers non-zero operands to the multipliers. However, EIE is designed only for fully connected layers. SCNN [36] also jointly exploits weight and activation sparsity. It targets the convolution layers, where the majority of computations take place, and uses a sparse planar-tiled input-stationary Cartesian product dataflow to enable efficient storage, delivery, and processing of the sparse weights and activations.

Efficiently executing sparse DNN inferences on GPU is challenging, as the irregular DNN topology and non-contiguous data structure increase the amount of branch divergences and uncoalesced memory accesses. To improve performance, Hill et al. [20] propose a synapse vector elimination technique to drop non-contributing synapses in the neural network and maintain computational regularity when pruning the network model. Their proposed technique reduces under-utilization of GPU resources.

ReRAM-based Sparse DNN Accelerators

The tightly coupled crossbar structure in the ReRAM-based DNN accelerator makes it difficult to effectively skip irregular zero weights and activations in DNNs. Directly applying sparsity exploration techniques designed for CMOS-based digital accelerators and GPUs on ReRAM-based DNN accelerators is not practical, as zero weights/activations are tightly coupled with other non-zero data in the same row/column and cannot be easily skipped. ReCom [24] is the first ReRAM-based DNN accelerator that exploits neural network sparsity. They use SSL [45] to structurally compress neural network models and remove all-zero rows for resource savings. They also exploit activation sparsity to reduce off-chip memory accesses. However, with ReCom's coarse-grain compression, only all-zero rows can be removed; many zero weights still remain in the compressed model. SNrram [44] is another ReRAM-based sparse DNN accelerator that compresses the model at a finer level, i.e., filter-sized all-zero columns in structurally compressed neural networks. As the size of a filter is usually smaller than a row, SNrram better exploits sparsity than ReCom. However, the scheme requires the use of an output indexing module with significant storage overhead. SNrram also exploits activation sparsity, but activation compression is only performed for deconvolution-layers in generative adversarial networks (GANs). Lin et al. [30] propose a different sparse mapping scheme based on k-means clustering, and use a crossbar-grained pruning algorithm to remove crossbars with low utilization. These prior sparsity solutions assume an over-idealized ReRAM crossbar architecture that activates an entire crossbar array in a single cycle. In contrast to these studies, our SRE takes advantage of practical fine-grained OU-based computations to jointly exploit weight and activation sparsity. Chen et al. [7] exploit the output sparsity introduced by the ReLU function to reduce computation. They propose an adaptive estimation method to detect negative output activations and terminate unnecessary bit-level convolutions earlier. Their early-termination technique is orthogonal to our SRE and can be combined with our approach to further improve performance and save energy.

9 CONCLUSION

In this paper, we study the design challenges of sparsity exploration on ReRAM-based DNN accelerators, and demonstrate that a practical OU-based ReRAM accelerator opens up new opportunities to effectively exploit DNN sparsity. We propose the first practical sparse ReRAM engine (SRE) that takes advantage of fine-grained OU-based computations to jointly exploit weight and activation sparsity. Our evaluation across a broad range of neural network models shows that SRE provides significant performance speedups (up to 42.3x) and energy savings (up to 95.4%) over a baseline that does not exploit sparsity. In addition, SRE successfully enables a practical ReRAM-based DNN accelerator design, which achieves satisfactory inference accuracy considering the limitation of ReRAM cell reliability while delivering comparable performance and energy efficiency with the over-idealized counterpart.

ACKNOWLEDGMENTS

We would like to thank Bing-Chen Wu and Tsung-Te Liu for their help on circuit synthesis, and Meng-Fan Chang and Chin-Fu Nien for their suggestions on circuit designs. We also appreciate the anonymous reviewers for their valuable comments and suggestions. This work was supported in part by research grants from the Ministry of Science and Technology of Taiwan (MOST-107-2221-E-002-043-MY2, MOST-107-2218-E-001-004-MY3), National Taiwan University (NTU-108L891903), and sponsored by Macronix Inc., Hsin-chu, Taiwan (107-S-C38).

REFERENCES

[1] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *Proceedings of International Symposium on Computer Architecture (ISCA)*.

[2] M. Alwani, H. Chen, M. Ferdman, and P. Milder. 2016. Fused-layer CNN Accelerators. In *Proceedings of International Symposium on Microarchitecture (MICRO)*.

[3] S. Chang, M. Sandler, and A. Zhmoginov. 2017. The Power of Sparsity in Convolutional Neural Networks. *arXiv* (2017).

[4] G. Chen, C. Parada, and G. Heigold. 2014. Small-footprint Keyword Spotting using Deep Neural Networks. In *Proceedings of International Conference on Acoustics, Speech and Signal Processing (ICASSP)*.

[5] P. Chen, X. Peng, and S. Yu. 2018. NeuroSim: A Circuit-Level Macro Model for Benchmarking Neuro-Inspired Architectures in Online Learning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 12 (2018), 3067–3080.

[6] W. Chen, K. Li, W. Lin, K. Hsu, P. Li, C. Yang, C. Xue, E. Yang, Y. Chen, Y. Chang, T. Hsu, Y. King, C. Lin, R. Liu, C. Hsieh, K. Tang, and M. Chang. 2018. A 65nm 1Mb Nonvolatile Computing-in-memory ReRAM Macro with Sub-16ns Multiply-and-accumulate for Binary DNN AI Edge Processors. In *Proceedings of International Solid-State Circuits Conference (ISSCC)*.

[7] X. Chen, J. Zhu, J. Jiang, and C. Tsui. 2019. CompRRAE: RRAM-based Convolutional Neural Network Accelerator with Reduced Computations Through a Runtime Activation Estimation. In *Proceedings of Asia and South Pacific Design Automation Conference (ASPDAC)*.

[8] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138.

[9] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. 2016. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *Proceedings of International Symposium on Computer Architecture (ISCA)*.

[10] Y. Le Cun, L. D. Jackel, B. Boser, J. S. Denker, H. P. Graf, I. Guyon, D. Henderson, R. E. Howard, and W. Hubbard. 1989. Handwritten Digit Recognition: Applications of Neural Network Chips and Automatic Learning. *IEEE Communications Magazine* 27, 11 (1989).

[11] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and F. Li. 2009. ImageNet: A Large-scale Hierarchical Image Database. In *Proceedings of Computer Vision and Pattern Recognition (CVPR)*.

[12] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. de Freitas. 2013. Predicting Parameters in Deep Learning. In *Proceedings of Neural Information Processing Systems (NIPS)*.

[13] B. Feinberg, S. Wang, and E. Ipek. 2018. Making Memristive Neural Network Accelerators Reliable. In *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*.

[14] D. Garbin, E. Vianello, O. Bichler, Q. Rafhay, C. Gamrat, G. Ghibaudo, B. DeSalvo, and L. Perniola. 2015. HfO2-Based OxRAM Devices as Synapses for Convolutional Neural Networks. *IEEE Transactions on Electron Devices* 62, 8 (2015), 2494–2501.

[15] X. Glorot, A. Bordes, and Y. Bengio. 2011. Deep Sparse Rectifier Neural Networks. In *Proceedings of International Conference on Artificial Intelligence and Statistics (AIStats)*.

[16] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of International Symposium on Computer Architecture (ISCA)*.

[17] S. Han, H. Mao, and W. J. Dally. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *arXiv* (2015).

[18] S. Han, J. Pool, J. Tran, and W. J. Dally. 2015. Learning both Weights and Connections for Efficient Neural Networks. *arXiv* (2015).

[19] K. He, X. Zhang, S. Ren, and J. Sun. 2015. Deep Residual Learning for Image Recognition. *arXiv* (2015).

[20] P. Hill, A. Jain, M. Hill, B. Zamirai, C. Hsu, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars. 2017. DeftNN: Addressing Bottlenecks for DNN Execution on GPUs via Synapse Vector Elimination and Near-compute Data Fission. In *Proceedings of International Symposium on Microarchitecture (MICRO)*.

[21] W. D. Hillis and G. L. Steele. 1986. Data Parallel Algorithms. *Commun. ACM* 29, 12 (1986), 1170–1183.

[22] K.C. Hsu, F.M. Lee, Y.Y. Lin, E.K. Lai, J.Y. Wu, D.Y. Lee, M.H. Lee, H.L. Lung, K.Y. Hsieh, and C.Y. Lu. 2015. A Study of Array Resistance Distribution and a Novel Operation Algorithm for WOx ReRAM Memory. In *Proceedings of International Conference on Solid State Devices and Materials (SSDM)*.

[23] M. Hu, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J. J. Yang, and R. S. Williams. 2016. Dot-product Engine for Neuromorphic Computing: Programming 1T1M Crossbar to Accelerate Matrix-vector Multiplication. In *Proceedings of Design Automation Conference (DAC)*.

[24] H. Ji, L. Song, L. Jiang, H. H. Li, and Y. Chen. 2018. ReCom: An Efficient Resistive Accelerator for Compressed Deep Neural Networks. In *Proceedings of Design, Automation Test in Europe (DATE)*.

[25] A. Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images. In *Tech Report, University of Toronto*.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of Neural Information Processing Systems (NIPS)*.

[27] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based Learning Applied to Document Recognition. *in Proceedings of the IEEE* 86, 11 (1998), 2278–2324.

[28] S. R. Li, J. Park, and P. T. P. Tang. 2017. Enabling Sparse Winograd Convolution by Native Pruning. *arXiv* (2017).

[29] L. Liang, L. Deng, Y. Zeng, X. Hu, Y. Ji, X. Ma, G. Li, and Y. Xie. 2018. Crossbar-aware Neural Network Pruning. *arXiv* (2018).

[30] J. Lin, Z. Zhu, Y. Wang, and Y. Xie. 2019. Learning the Sparsity for ReRAM: Mapping and Pruning Sparse Neural Network for ReRAM Based Accelerator. In *Proceedings of Asia and South Pacific Design Automation Conference (ASPDAC)*.

[31] M. Lin, H. Cheng, W. Lin, T. Yang, I. Tseng, C. Yang, H. Hu, H. Chang, H. Li, and M. Chang. 2018. DL-RSIM: A Simulation Framework to Enable Reliable ReRAM-based Accelerators for Deep Learning. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*.

[32] A. L. Maas, A. Y. Hannun, and A. Y. Ng. 2013. Rectifier Nonlinearities Improve Neural Network Acoustic Models. In *Proceedings of International Conference on Machine Learning (ICML)*.

[33] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. 2016. Pruning Convolutional Neural Networks for Resource Efficient Transfer Learning. *arXiv* (2016).

[34] N. Muralimanohar, R. Balasubramonia, and N. P. Jouppi. 2009. *CACTI 6.0: A Tool to Model Large Caches.* Technical Report. HP Lab.

[35] P. Gu, B. Li, T. Tang, S. Yu, Y. Cao, Y. Wang, and H. Yang. 2015. Technological Exploration of RRAM Crossbar Array for Matrix-vector Multiplication. In *Proceedings of Asia and South Pacific Design Automation Conference (ASPDAC)*.

[36] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of International Symposium on Computer Architecture (ISCA)*.

[37] J. Park, S. R. Li, W. Wen, H. Li, Y. Chen, and P. Dubey. 2016. Holistic SparseCNN: Forging the Trident of Accuracy, Speed, and Size. *arXiv* (2016).

[38] M. Saberi, R. Lotfi, K. Mafinezhad, and W. A. Serdijn. 2011. Analysis of Power Consumption and Linearity in Capacitive Digital-to-Analog Converters Used in Successive Approximation ADCs. *IEEE Transactions on Circuits and Systems I: Regular Papers* 58, 8 (2011), 1736–1748.

[39] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. Lecun. 2014. Overfeat: Integrated Recognition, Localization and Detection using Convolutional Networks. In *Proceedings of International Conference on Learning Representations (ICLR)*.

[40] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *Proceedings of International Symposium on Computer Architecture (ISCA)*.

[41] K. Simonyan and A. Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Proceedings of International Conference on Learning Representations (ICLR)*.

[42] F. Su, W. Chen, L. Xia, C. Lo, T. Tang, Z. Wang, K. Hsu, M. Cheng, J. Li, Y. Xie, Y. Wang, M. Chang, H. Yang, and Y. Liu. 2017. A 462GOPs/J RRAM-based Nonvolatile Intelligent Processor for Energy Harvesting IoE System Featuring Nonvolatile Logics and Processing-in-memory. In *Proceedings of International Symposium on VLSI Technology, Systems and Applications(VLSI-TSA)*.

[43] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. 2015. Going Deeper with Convolutions. In *Proceedings of Computer Vision and Pattern Recognition (CVPR)*.

[44] P. Wang, Y. Ji, C. Hong, Y. Lyu, D. Wang, and Y. Xie. 2018. SNrram: An Efficient Sparse Neural Network Computation Architecture Based on Resistive Random-access Memory. In *Proceedings of Design Automation Conference (DAC)*.

[45] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. 2016. Learning Structured Sparsity in Deep Neural Networks. In *Proceedings of Neural Information Processing Systems (NIPS)*.

[46] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie. 2015. Overcoming the Challenges of Crossbar Resistive Memory Architectures. In *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*.

[47] C. Xue, W. Chen, J. Liu, J. Li, W. Lin, W. Lin, J. Wang, W. Wei, T. Chang, T. Chang, T. Huang, H. Kao, S. Wei, Y. Chiu, C. Lee, C. Lo, Y. King, C. Lin, R. Liu, C. Hsieh, K. Tang, and M. Chang. 2019. A 1Mb Multibit ReRAM Computing-In-Memory Macro with 14.6ns Parallel MAC Computing Time for CNN Based AI Edge Processors. In *Proceedings of International Solid-State Circuits Conference (ISSCC)*.

[48] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke. 2017. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. In *Proceedings of International Symposium on Computer Architecture (ISCA)*.

[49] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. 2016. Cambricon-X: An Accelerator for Sparse Neural Networks. In *Proceedings of International Symposium on Microarchitecture (MICRO)*.