AUTO-PRUNE: Automated DNN Pruning and Mapping for ReRAM-Based Accelerator

Siling Yang
Zhejiang University
slingzjunet@zju.edu.cn

Weijian Chen
Zhejiang University
weijianchen@zju.edu.cn

Xuechen Zhang
Washington State University
Vancouver
xuechen.zhang@wsu.edu

Shuibing He*
Zhejiang University
heshuibing@zju.edu.cn

Yanlong Yin
Zhejiang Lab
yyin@zhejianglab.com

Xian-He Sun
Illinois Institute of Technology
sun@iit.edu

ABSTRACT

Emergent ReRAM-based accelerators support in-memory computation to accelerate deep neural network (DNN) inference. Weight matrix pruning of DNNs is a widely used technique to reduce the size of DNN models, thereby reducing the resource and energy consumption of ReRAM-based accelerators. However, conventional works on weight matrix pruning for ReRAM-based accelerators have three major issues. First, they use heuristics or rules from domain experts to prune the weights, leading to suboptimal pruning policies. Second, they mostly focus on improving compression ratio, thus may not meet accuracy constraints. Third, they ignore direct feedback of hardware.

In this paper, we introduce an automated DNN pruning and mapping framework, named AUTO-PRUNE. It leverages reinforcement learning (RL) to automatically determine the pruning policy considering the constraint of accuracy loss. The reward function of RL agents is designed using hardware's direct feedback (i.e., accuracy and compression rate of occupied crossbars). The function directs the search of the pruning ratio of each layer for a global optimum considering the characteristics of individual layers of DNN models. Then AUTO-PRUNE maps the pruned weight matrices to crossbars to store only nontrivial elements. Finally, to avoid the dislocation problem, we design a new data-path in ReRAM-based accelerators to correctly index and feed input to matrix-vector computation leveraging the mechanism of operation units. Experimental results show that, compared to the state-of-the-art work, AUTO-PRUNE achieves up to 3.3X compression rate, 3.1X area efficiency, and 3.3X energy efficiency with a similar or even higher accuracy.

CCS CONCEPTS

• **Computer systems organization** → **Neural networks**; • **Hardware** → *Analysis and design of emerging devices and systems.*

*Shuibing He is the corresponding author.

ICS '21, June 14–17, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8335-6/21/06…$15.00
https://doi.org/10.1145/3447818.3460366

KEYWORDS

ReRAM-based Accelerator, Pruning, Reinforcement Learning

ACM Reference Format:
Siling Yang, Weijian Chen, Xuechen Zhang, Shuibing He, Yanlong Yin, and Xian-He Sun. 2021. AUTO-PRUNE: Automated DNN Pruning and Mapping for ReRAM-Based Accelerator. In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3447818.3460366

1 INTRODUCTION

Deep neural networks (DNNs) have become the dominant approach to solving a variety of computing problems in computer vision [26], natural language processing [22], robotics[24] among many other fields. Leveraging emerging devices and non-traditional computing systems [1–3, 6, 9, 10, 12, 27, 37] is an ideal approach to accelerate DNN inference. Resistive random access memory (ReRAM) is the main candidate for DNN accelerators because of its superior characteristics of extremely low energy leakage, high-density storage, and in-situ computation.

ReRAM-based accelerators can perform matrix-vector multiplication efficiently in the convolutional (CONV) layers and fully-connected (FC) layers of DNNs [10, 17, 23, 29]. They store weight matrices of DNN filters in crossbar arrays. The weights are represented as the conductances, which conduct dot-product with the voltages converted from the input feature maps. The current in the end of each bitline can be summed up and the dot-product operations are executed simultaneously, thus reducing the massive amount of data movements between memory and arithmetic units required in the von Neumann computer architecture.

Recent researches show that weight sparsity increases as the bits-per-cell decreases [33]. After applying weight sparsifying algorithms (e.g., quantization [7] and low-rank matrix factorization [5]) during training, up to 78% of crossbar cells may store zero weights. As a result, pruning these values before mapping to hardware saves the usage of crossbars and removes unnecessary computations. However, existing pruning algorithms designed on ReRAM-based accelerators have the following three major issues.

First, *these algorithms prune the weights using heuristics or rules.* For example, they typically used heuristics (e.g., patterns and all-zero rows/columns) to direct the pruning process [4, 30, 35]. Because of the nature of heuristics-based algorithms, these schemes may prune some nontrivial weights or preserve some trivial weights. Therefore, it would be very difficult for them to find a pruning policy

on ReRAM-based accelerators that achieves a global optimum for DNN models.

Second, *they do not use the actual accuracy loss of compressed DNN models as direct feedback in pruning*. To reduce the resource and energy consumption of ReRAM-based accelerators, existing algorithms tend to maximize the compression rate of weight matrices of DNNs [4, 33]. They do not consider its impact on the actual accuracy loss of the compressed DNN models, which may not meet the accuracy constraints.

Third, *they ignore direct feedback of hardware in pruning*. The performance of a pruning algorithm cannot be accurately reflected by the compression rate of weight matrices of DNNs. For example, the architecture of ReRAM-based accelerators includes both crossbar arrays and their auxiliary ADC/DAC circuits. The conventional pruning algorithms optimized on the compression rate might not be optimal on the energy consumption for the ReRAM-based accelerator. Furthermore, the algorithms optimized on one hardware may not be optimal on the other with respect to the number of occupied crossbars. We need specialized pruning policies for different architectures of ReRAM-based accelerators.

In this paper, we propose a hardware-aware automated pruning and mapping framework, named Auto-prune, for ReRAM-based accelerators. First, it leverages reinforcement learning to automatically predict the sparsity of DNN layers given the resource constraints and the hardware's feedback. For each layer, the RL agent receives the configuration and characteristic of the layer as observation. Then it outputs the expected pruning ratio of the weight matrix for the layer. After the pruning ratios of all the layers are decided, we leverage the simulator of ReRAM-based accelerators as the environment to obtain direct feedback (i.e., the accuracy loss and/or energy consumption of the compressed DNN models and the compression rate of occupied crossbars). The RL agent then uses the feedback to compute reward of this pruning policy (i.e., a list of pruning ratios for all layers). After multiple epochs of searching both locally and globally, the reward converges and the optimal pruning policy is decided. Second, Auto-prune compresses DNN models in a layer-wise manner given the layer sparsity. Specifically, it prunes the weight matrices in the granularity of column-vectors. We remove the less important column-vectors and shift the remaining vectors left and then map the pruned weight matrices to crossbar arrays. Third, we add a weight indexing structure to the data-path of the architecture of ReRAM-based accelerators. The control unit can feed matching input to the ReRAM-based crossbars to conduct matrix-vector multiplication at the level of the operation unit (OU) [33]. Finally, the control unit also needs to place the output of an OU at its matching position in the output register to solve the dislocation problem.

In summary, this paper offers the following contributions:

- We design an automated pruning and mapping framework for ReRAM-based accelerators, Auto-prune, which searches for a global optimum policy without requiring rule-based heuristics and domain experts.
- We use the actual accuracy loss and direct feedback from the simulator of ReRAM-based accelerators in the pruning policy, which offers a specialized solution given the different configurations of ReRAM-based accelerators.

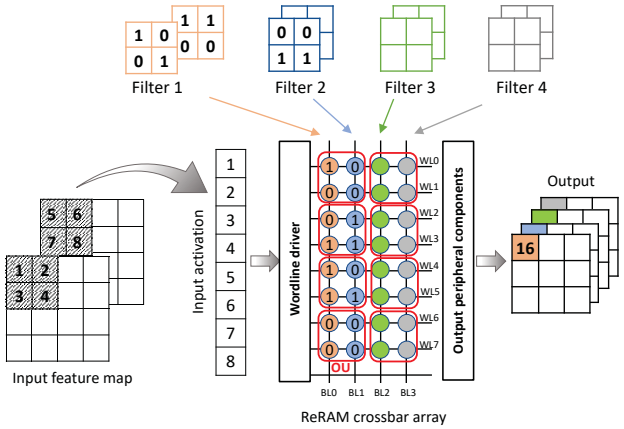


Figure 1: Illustration of mapping filter weights to a crossbar array used in the architecture of ReRAM-based accelerators. BL: bitline; WL: wordline; OU: operation unit.

- We prune and map weight matrices on ReRAM-based accelerator in a finer granularity of column-vector, which makes better trade-offs between compression rate and accuracy compared to conventional coarse-grained methods.
- We design a new data-path to support the column-vector pruning and mapping, which uses the OU mechanism to skip useless inputs, reduce ineffectual computation, and avoid the dislocation problem.

We evaluate Auto-prune with three DNN models including AlexNet [15], VGG16 [13], and Plain20 [8] on two datasets, i.e., CIFAR10 [14] and MNIST [16]. Compared to the state-of-the-art work PIM-Prune [4], Auto-prune can achieve up to 3.3X compression rate, 3.1X area efficiency, and 3.3X energy efficiency with a similar or even higher accuracy.

2 BACKGROUND AND RELATED WORK

2.1 Mapping Filter Weights of DNNs in ReRAM-based Accelerators

The architecture of ReRAM-based accelerators consists of crossbar arrays and input/output peripheral components. In a crossbar array, each bitline is connected to each wordline through a ReRAM cell. Input peripheral circuits (e.g., wordline decoders) convert inputs to the voltage pulses and feed them into the corresponding wordlines. Each ReRAM cell conveys the inner product between the driving voltage and the cell conductance and generates current which reaches the output peripheral circuits, e.g., analog-to-digital converters (ADCs). The accumulated current at the end of each bitline is converted by ADCs to the digital values representing the partial sum of a convolution operation. Because of the overhead from ADC, matrix-vector multiplication in ReRAM-based accelerators must be executed at a smaller granularity, called an operation unit (OU) [33]. Figure 1 shows a simplified example of an 8×4 crossbar array. When OU is enabled, only two wordlines and two bitlines are turned on concurrently within the crossbar array in one cycle.

When a DNN model is mapped onto a ReRAM crossbar array, the synaptic weights of neurons are encoded as the conductances of

Table 1: Comparison of Auto-prune with existing weight matrix pruning schemes for ReRAM-based accelerators.

Pruning technique	*Pattern for pruning*	*Automation*	*Hardware feedback*	*Use OU in data-path*
Lin et al. [21]	Unimportant weight groups	No	No	No
SRE [33]	All-zero row/column vectors	No	No	Yes
PIM-Prune [4]	Unimportant rows and columns	No	No	No
Pattern pruning [35]	Patterns	No	No	Yes
Auto-prune	Unimportant column-vectors	Yes	Yes	Yes

ReRAM cells in crossbars. Figure 1 demonstrates a mapping scheme for ReRAM-based accelerators. It shows the convolution operations between four $2\times2\times2$ filters and one $4\times4\times2$ input feature map in a convolution layer. Each element of the filter is mapped to one bitline of the crossbar array. A set of $2 \times 2 \times 2$ activation values derived from the input feature map is sent to eight wordlines of the crossbar array after being converted to the input voltages. Because only two wordlines and two bitlines in one OU are activated in one cycle, the four convolution operations through the four bitlines are completed in eight cycles. At the end of each bitline, the accumulated currents are converted to the digital value by ADC, which corresponds to an element in one channel of the output feature map. Then, the input sliding window moves right (or down) and the corresponding elements in the input feature maps are fed into the crossbar array in the next cycles. When all the elements of the input feature map complete convolution operations with the four filters in this layer, the output feature map of $3 \times 3 \times 4$ can be stored in buffers and used as the input for the next layer.

2.2 Weight Pruning for ReRAM-based Accelerators

Filter weight matrices of DNN models are sparse because they often store zero or redundant weights, which have a trivial impact on the accuracy of the models [32]. More sparsity can be created by applying weight quantization [7], low-rank matrix factorization [5], and regularization [18, 36]. For example, Denil et al. obtained weight matrices of 95% sparsity after applying low-rank matrix factorization[5]. Therefore, before mapping the weight matrices to crossbars of ReRAM-based accelerators, they need to be pruned to reduce the number of occupied crossbars and their energy consumption. However, all the rows and columns in a crossbar array are coupled together. Even if some cells of the crossbar store zeros, we cannot remove them directly because there are non-zero cells in the same rows or columns. Researchers have designed several weight pruning schemes considering the tightly coupled crossbar structure in ReRAM-based accelerators [4, 21, 33, 35]. They can be used to effectively reduce the number of trivial elements exploiting the sparsity of the weight matrices.

The tightly coupled crossbar structure makes it difficult to exploit the sparsity of neural networks for ReRAM-based accelerators. ReCom is the first to exploit the sparsity of neural networks for ReRAM-based accelerators [11]. It explores the weight sparsity only in the granularity of matrix-row and crossbar-row. SNrram [32] and XCS [19] prune the unimportant columns of the weight matrix to exploit the sparsity. Lin et al. proposed to exchange columns of weight matrices to move non-zero elements together and store them in clusters separated from the clusters of zero elements [21]. Then the clusters of zero elements can be pruned. Yang et al. designed a sparse ReRAM engine that prunes all-zero vectors in weight matrices in either row or column direction for OU-based ReRAM-based accelerators[33]. They exploited the sparsity of weight matrices at the granularity of row/column vectors for a higher compression ratio. For the pruning schemes designed for OU-based ReRAM-based accelerators, additional indexing tables are required in their data-path to active correct OUs in subsequent cycles. PIM-Prune exploits the sparsity at the level of blocks of weight matrices [4]. It prunes the elements in both row and column directions. Most recently, pattern pruning uses patterns that represent irregular vectors of a particular shape to identify more zero elements for pruning [35].

We compare Auto-prune to the major pruning schemes in Table 1. Auto-prune has three major differences from them. First, the existing schemes use the proposed heuristics or rules to find a pruning policy. Because of the nature of heuristics-based algorithms, it would be very difficult to find a global optimum for the DNN model. Auto-prune addresses this issue by searching for an optimal solution globally based on reinforcement learning. Second, they were designed to maximize the compression ratio of weight matrices. They may not meet the accuracy requirements. In contrast, Auto-prune involves direct feedback from the ReRAM-based accelerators in the design loop, which makes better trade-offs between compression rate and accuracy. Third, existing approaches usually prune weight matrices in the granularity of matrix, block, or crossbar row/column. Auto-prune performs the pruning on ReRAM-based accelerator in a finer granularity of column-vector based on the OU mechanism. It delivers a higher compression rate with the accuracy constraint.

2.3 Accelerator Design using Reinforcement Learning

AutoML based on reinforcement learning is designed to release human labor on searching configurations while there are vast search space and limited computational budgets. It is a popular search method with good performance, less assistance from humans, and high computational efficiency [25, 34]. Therefore, AutoML is widely used in neural architecture search. Inspired by the AutoML framework, AMC compresses DNN models automatically [8]. It achieves a higher compression ratio and preserves better accuracy than heuristics-based model compression algorithms. More importantly, it does not require human expertise in the design. Recently, HAQ [31] was designed to decide a hardware-aware quantization policy for DNNs using AutoML. They targeted FPGA and ASCI-based accelerators. Similar to them, Auto-prune uses AutoML to determine the pruning ratio of weight matrices of DNNs before mapping to crossbars. Its reinforcement learning agent is designed

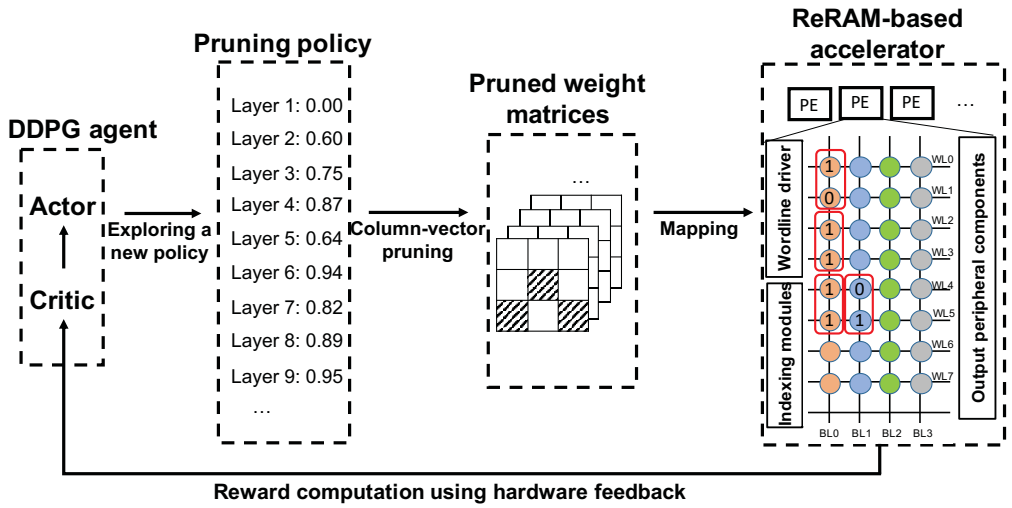


Figure 2: An overview of the AUTO-PRUNE framework.

to consider the characteristics of crossbars in the architecture of ReRAM-based accelerators.

3 DESIGN OF AUTO-PRUNE

We model the pruning task for ReRAM-based accelerators as a reinforcement learning problem. The goal in reinforcement learning is to learn a pruning policy that minimizes the number of occupied crossbars used in ReRAM-based accelerators with accuracy constraints. Figure 2 shows the general architecture of the AUTO-PRUNE framework, which includes the following components.

DDPG agent. It is used to generate actions, i.e., pruning policies for each layer of DNNs. The DDPG agent is a pair of actor-critic network [20]. The *actor* network predicts a new pruning policy for hardware feedback given the input of a state vector consisting of features of DNN models (e.g., the number of input channels) and ReRAM-based accelerators (e.g., the number of crossbars required before pruning). The *critic* network evaluates the importance of the action-state pair using Q-function [28].

Pruning policy. The agent needs to explore a large action space of pruning policies. We use a list of pruning ratios, each corresponds to a layer of DNN models, to denote one pruning policy. In Figure 2, the policy to be evaluated by the hardware assigns the pruning ratio of 0.6 to layer 2, meaning that AUTO-PRUNE will need to prune 60% of the elements in the filter weight matrix of layer 2 before mapping it to crossbars.

Pruned weight matrices. They are a list of filter weight matrices to be mapped to ReRAM-based accelerators after pruning. Different pruning policies may result in different pruned weight matrices, leading to the accuracy variation of the pruned DNN models and the changing resource consumption of hardware.

ReRAM-based accelerator. It is simulated using MNSIM [38]. Because AUTO-PRUNE leverages the OU mechanism to explore fine-grained pruning, we add indexing modules to the data-path of the processing elements (PEs) of ReRAM-based accelerators in the simulator.

Table 2: Symbols used in the DDPG algorithm.

Symbol	Meaning
k	layer index
t	layer type: CONV:1 ; FC: 0
inc	number of channels in the input feature map
$outc$	number of channels produced by the convolution
ks	number of elements of a convolving kernel
h	height of the input feature maps
w	width of the input feature maps
s	stride of the convolution
$xb[k]$	number of crossbars required for mapping layer k
$xb_{saved}[k]$	accumulated number of the crossbar saved from the first layer to layer $k-1$
$xb_{rest}[k]$	number of crossbars required from layer $k+1$ to the last layer
$size_{xb}$	length of the crossbar size
acc_{reram}	accuracy reported by the ReRAM-based accelerator simulator
a_{k-1}	action from the last time step

3.1 DDPG Algorithm for ReRAM-based Accelerators

In this section, we describe the DDPG algorithm to search for an optimal pruning policy given direct hardware feedback. In the design of a DDPG agent, we need to define its *state space*, *action space*, and *reward function*.

State space. For each layer k, we use a 12-dimensional feature vector as our observation. Specifically, the state vector S_k for layer k is defined as

$$(k, t, inc, outc, ks, h, w, s, xb[k], xb_{saved}[k], xb_{rest}[k], a_{k-1}) \quad (1)$$

where all the features are defined in Table 2. It is worth noting that for fully-connected layers their *inc* and *outc* are equal to the

number of input and output neurons respectively. To make the reinforcement learning model effective for both convolutional layers and fully-connected layers, we set both ks and s to 1 for the fully-connected layers even though they do not have such attributes.

Action space. In order to achieve a fine-grained pruning decision, we choose the Actor's action space $a_k \in (0, 1]$ and prune the current layer with a pruning ratio a_k at the granularity of column-vectors. We describe the pruning algorithm in Section 3.2.

Reward function. As direct feedbacks, the simulator passes the number of occupied crossbars and the accuracy of the compressed DNN models to the DDPG agent to compute the reward for reinforcement learning given the current pruning policy. Specifically, we define our reward function to be related to both compression rate and DNN model accuracy in Equation 2. The agent uses the reward function to obtain reward and adjusts its policy based on the reward. The reward function is also designed to balance the impact of the compression rate of crossbars and the accuracy of pruned DNN models in the searching process. Consequently, we can avoid the situation of significant accuracy drop when exploring pruning policies to maximize the compression rate of crossbars.

$$Reward = (1 - \frac{1}{rate^{xb}_{compression}})^{\alpha} \times acc_{reram} \tag{2}$$

In the equation, α is a scaling factor which is set to 2 in our experiments. $rate^{xb}_{compression}$ is a ratio of the number of occupied crossbars for all the layers without pruning (xb_{ori}) to the number of occupied crossbars using the current pruning policy (xb_{cur}) as shown in Equation 3.

$$rate^{xb}_{compression} = \frac{xb_{ori}}{xb_{cur}} \tag{3}$$

$$xb_{ori} = \sum_{k=0}^{L-1}(xb_{ori}[k]) \tag{4}$$

$$xb_{cur} = \sum_{k=0}^{L-1}(xb_{cur}[k]) \tag{5}$$

$xb_{ori}[k]$ and $xb_{cur}[k]$ are the number of occupied crossbars for layer k without pruning and with the current pruning policy respectively. $xb_{cur}[k]$ is obtained from the simulator. $xb_{ori}[k]$ is calculated using the following equations.

For convolutional layer k,

$$xb_{ori}[k] = \lceil \frac{ks \times inc}{size_{xb}} \rceil \times \lceil \frac{outc}{size_{xb}} \rceil \tag{6}$$

For FC layer k,

$$xb_{ori}[k] = \lceil inc/size_{xb} \rceil \times \lceil outc/size_{xb} \rceil \tag{7}$$

3.2 Column-Vector Based Pruning and OU Formation

Given a pruning policy generated by the DDPG actor, AUTO-PRUNE needs to run pruning algorithms to mark nontrivial elements in filter weight matrices based on their weights. It only maps the nontrivial elements to crossbars. We use Algorithm 1 for matrix pruning in AUTO-PRUNE. Specifically, it calculates the number of column-vectors in a column of the weight matrix given the kernel

Algorithm 1 The column-vector based pruning algorithm

Require: g: the granularity of the column-vector;
$ratio_{pruning}$: the pruning ratio generated by DDPG agents;
$outc$: number of output channels of a layer;
inc: number of channels of input feature map of the layer;
ks: kernel size of the layer;
W: 2D-weight matrix of the layer;

1: $num \leftarrow \lfloor ks \times inc/g \rfloor$
2: $tmp\_sum \leftarrow 0$, $dict \leftarrow \{\}$, $list \leftarrow []$
3: **for** $i \leftarrow 0, 1, \ldots, (outc - 1)$ **do**
4: $cnt \leftarrow 0$
5: **for** $j \leftarrow 0, 1, \ldots, (num \times g - 1)$ **do**
6: $tmp\_sum \leftarrow tmp\_sum + abs(W[j][i])$
7: **if** $(j + 1)\%g == 0$ **then**
8: $dict$.append($key : (cnt, i)$, $value : tmp\_sum$)
9: $tmp\_sum \leftarrow 0$
10: $cnt \leftarrow cnt + 1$
11: **end if**
12: **end for**
13: **end for**
14: $sorted\_list \leftarrow ascend\_sort\_by\_value(dict)$
15: **for** $i \leftarrow \lceil ratio_{pruning} \times num \times outc \rceil - 1, \ldots,$ len(sorted_list)-1 **do**
16: $list$.append($sorted\_list[i].key$)
17: **end for**
18: **return** $list$

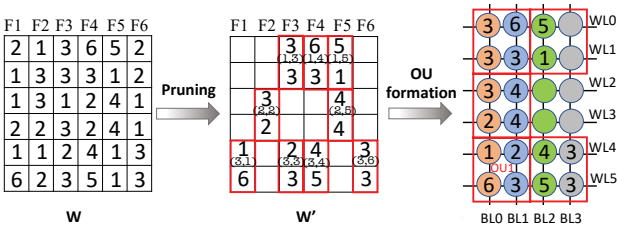


Figure 3: An example of the pruning and OU formation process. The value in each cell represents the weight of the corresponding element in the filter. (x,y) represents the coordinate of a vector in the vector space. W and W' denote the original and pruned weight matrices respectively.

size of the layer, the number of input channels, and the granularity of column-vectors (Line #1). Then for each column in the matrix, it scans through all the column-vectors and records their coordinates and the accumulated weights of the vectors in a dictionary (Line #3-13). Then we sort the vectors based on their weights (Line #14). Finally, based on the pruning ratio, the vectors consisting of nontrivial weight elements are selected and returned (Line #15-18).

We use an example to illustrate the pruning and OU formation process in Figure 3. In the example, we assume the granularity of a column-vector is 2. The weight matrix W can be divided into 18 column-vectors. Each column-vector is indexed using its coordinate in the vector space. For example, the coordinate of the vector $[1, 6]^T$ in filter $F1$ (column 1) is (3, 1), where 3 represents that it is located

Algorithm 2 Vector sequence creation algorithm

Require: h: the number of column-vectors in each OU;
$list[i]$: the list of index pairs from Algorithm 1;

```
1:  Index ← {}, L ← getIndexLen(list)
2:  cnt ← 0
3:  for i =0, 1, ..., (L − 1) do
4:      tag[i] ← 1
5:  end for
6:  for i =0, 1, ..., (L − 1) do
7:      if tag[i] == 1 then
8:          Index.append(list[i])
9:          tag[i] ← 0
10:         cnt ← 1
11:         for j = i + 1, ..., (L − 1) do
12:             if list[i].x == list[j].x and tag[j] == 1 then
13:                 Index.append(list[j])
14:                 tag[j] ← 0
15:                 cnt ← cnt + 1
16:                 if cnt==h then
17:                     break
18:                 end if
19:             end if
20:         end for
21:     end if
22: end for
23: return Index
```

in the third row in the vector space and 1 represents that it is in the first column in the vector space. After running the algorithm for the pruned weight matrix W', nine vectors are pruned to achieve a pruning ratio of 50% because their accumulated weights are smaller than those of the remaining vectors. One observation from the example is that the pruned vectors are randomly located in the weight matrix. In order to exploit the sparsity, Auto-prune accumulates the remaining vectors to remove the holes. Then it leverages the support of OUs in ReRAM-based accelerators to perform matrix-vector multiplication.

For the formation of OUs after pruning, we need to create a list of indexes of column-vectors based on the order of their access in crossbars in the sequence of computation of OUs. Then we will add a corresponding indexing module to the data-path of ReRAM-based accelerators as discussed in Section 3.3. We use Algorithm 2 to form OUs and create the index. Each OU has a fixed number of column-vectors h. In the vector space, we need to find h vectors that are next to each other (Line #11-20). These h vectors are computed together in one OU. The algorithm assigns all vectors in the list output from Algorithm 1 to their corresponding OUs. The output of Algorithm 2 is an indexing list to be used by data-paths of ReRAM-based accelerators. We continue using the example in Figure 3 for illustration. The index output of Algorithm 2 is {(3,1), (3,3), (2,2), (2,5), (1,3), (1,4), (3,4), (3,6), (1,5)}. We assume that the OU size is 2×2 and two column-vectors are assigned to one OU. The data-path hardware in ReRAM-based accelerators will map the weight vectors onto 5 OUs. For example, $[1, 6]^T$ at (3,1) in $F1$ and $[2, 3]^T$ at (3,3) in $F3$ are assigned to OU1.

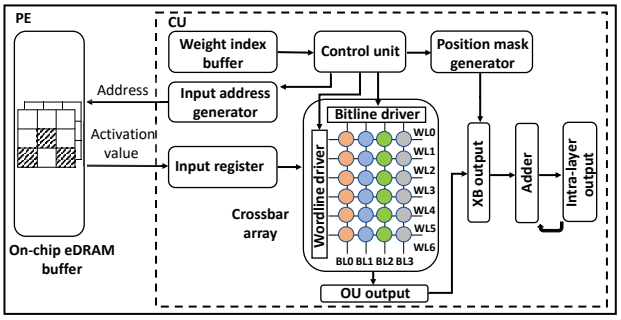


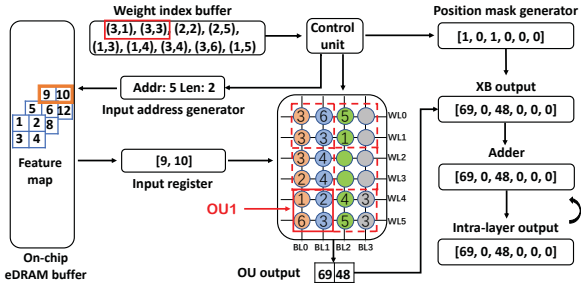
Figure 4: An overview of the data-path for Auto-prune.

3.3 Data-Path Design

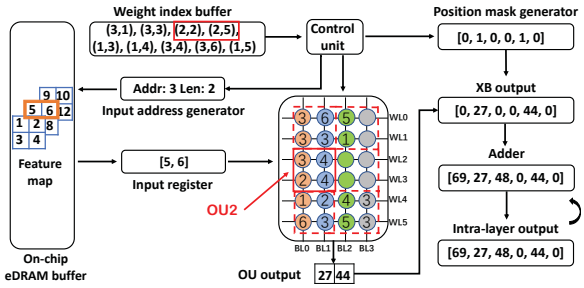
Because we use semi-structural pruning in Auto-prune, vectors of different filters may be placed to the same crossbar-column. As shown in Figure 3, $[3, 3]^T$ of F3, $[3, 2]^T$ of F2, and $[1, 6]^T$ of F1 are placed to the first column in the crossbar, resulting in the dislocation problem [4]. In this section, we introduce a novel data-path to solve the dislocation problem in ReRAM-based accelerators.

Figure 4 shows the data-path designed for the column-vector pruning algorithm used in the Auto-prune framework. Because the crossbar arrays only store nontrivial weights, we only need to fetch the input activations corresponding to the weights. *Weight index buffers* store the index of column-vectors generated by Algorithm 2. In each cycle, control units fetch a few index tuples having the same x coordinate from the weight index buffer. The number of tuples fetched should be smaller than or equal to the number of column vectors in one OU. *Input address generators* generate the address of an activation vector in the input feature map to be accessed by the crossbar given the x coordinate of index tuples from the weight index buffer. Specifically, the buffer address of the activation vector is $g \times (x - 1) + 1$ where g is the granularity of column-vectors. *Input registers* store the input to the crossbar array. To perform computation on OUs, *control units* issue command to active corresponding wordlines and bitlines given the coordinates of column-vectors. Then we need to place the output of crossbars at a matching location in the output. For this purpose, we use *position mask generators* to produce position masks whose length is equal to the number of columns of the original weight matrix. The currents from crossbar arrays are converted by ADCs and then stored in *OU outputs*. Then *XB (crossbar) outputs* recover the final output by padding the value from the OU output with zeros according to the position mask. Then *adders* add the previous output from *intra-layer output* and the new value from XB output. Finally, they store the partial sum to the intra-layer output.

We continue to use the example in Figure 3 for illustration. Given the output of Algorithm 2, The weight index buffer stores {(3,1), (3,3), (2,2), (2,5), (1,3), (1,4), (3,4), (3,6), (1,5)}. In OU1, two vectors $[1, 6]^T$ and $[2, 3]^T$ with indexes (3,1) and (3,3) respectively (marked in the red square in Figure 5(a)) are selected to perform computation. For the selected weights, the input address generator outputs buffer address 5 which is equal to 2×(3-1)+1. Then given the address, [9,10] in the feature map are fetched to the input register. The crossbar array then performs multiplication between [9, 10] and the selected weights in OU1 and then outputs [69, 48]. However, because $[1, 6]^T$



(a) OU1



(b) OU2

Figure 5: States of data-path components after the execution of OU1 and OU2.

and $[2, 3]^T$ are from F1 and F3 respectively as shown in Figure 3. We cannot store it in the intra-layer output directly. Instead, we need to find the corresponding positions of 69 and 48 in the XB output. The positions indicated by the position mask are [1, 0, 1, 0, 0, 0] because the accumulated sum for Filter 1 and Filter 3 should be placed at positions 1 and 3 respectively. As a result, the adder adds [69, 0, 48, 0, 0, 0] from the XB output and [0, 0, 0, 0, 0, 0] from the intra-layer output and then sends the result [69, 0, 48, 0, 0, 0] to the intra-layer output.

In OU2, two vectors $[3, 2]^T$ and $[4, 4]^T$ with indexes (2,2) and (2,5) respectively (marked in the red square in Figure 5(b)) are selected to perform computation. For the selected weights, the input address generator outputs buffer address 3. Then given the address [5,6] in the feature map is fetched to the input register. The crossbar array then performs multiplication between [5, 6] and the selected weights in OU2 and outputs [27, 44]. However, because $[3, 2]^T$ and $[4, 4]^T$ are from F2 and F5 respectively as shown in Figure 3. We need to find the corresponding positions of 27 and 44 in the XB output. The position mask is [0, 1, 0, 0, 1, 0] because the accumulated sum for Filter 2 and Filter 5 should be placed at positions 2 and 5 respectively. As a result, the adder adds [0, 27, 0, 0, 44, 0] from the XB output and [69, 0, 48, 0, 0, 0] from the intra-layer output and then sends the result [69, 27, 48, 0, 44, 0] to the intra-layer output.

4 EVALUATION

4.1 Experimental Setup

Experimental platform. We implement a pruning and mapping framework for ReRAM-based accelerators in Python. We use a ReRAM simulator in the experiments because commercial ReRAM devices are unavailable to us. We use MNSIM [38] as the simulator of ReRAM-based accelerators because of its efficiency, flexibility, and simplicity. To support multi-bit weights, ISAAC [27] and PRIME [3] store the weights in multiple cells in the same row and require additional modules to support shift-and-add operations, thus increasing the overhead of peripheral circuits and lacking in reconfigurability for different bit widths. MNSIM stores multi-bit weights in multiple crossbars, improving the flexibility of computing units while reducing the overhead of peripheral modules.

Table 3: The structure of neural networks. The symbol $a \times Cb\text{-}c$ represents that a convolutional layers with $b \times b$-sized kernels and c output channels. The symbol Fd means a fully-connected layer with d neurons.

Network	Structure
AlexNet	$C3$-64, $C3$-192, $C3$-384, $2 \times C3$-256, F4096, F4096, F10
VGG16	$2 \times C3$-64, $2 \times C3$-128, $3 \times C3$-256,$3 \times C3$-512, $3 \times C3$-512, F4096, F1000, F10
Plain20	$7 \times C3$-16, $6 \times C3$-32, $6 \times C3$-64, F10

We use the hardware model in MNSIM to evaluate the area and energy consumption of crossbars and other modules (including DAC, ADC, IR, OR, and other digital parts). In the setting of the model, each memristor cell stores one bit, and both ADC and DAC are set to be 1 bit. Each weight of the DNN models uses 8-bit quantization, which means 8 crossbars are needed to represent a weight. By default, we set the crossbar size as 128×128 and the granularity of column-vector (g) as 32. The operation unit (OU) size is set to $g \times g$. All other configurations are the same as the default ones used in MNSIM.

Workloads and datasets. We evaluate AUTO-PRUNE with three DNN models, including AlexNet [15], VGG16 [13], and Plain20 [8]. Table 3 shows the structures of these DNN models. We execute the inference of the models on two datasets, i.e., CIFAR10 [14] and MNIST [16]. The CIFAR10 dataset consists of 60, 000 colorful images, each with $32 \times 32 \times 3$ pixels. There are 50, 000 images for training and 10, 000 images for testing. The MNIST dataset consists of 70, 000 grayscale $28 \times 28 \times 1$ images. It includes 60, 000 images for training and 10, 000 images for testing.

Comparison systems. We compare AUTO-PRUNE with the state-of-the-art pruning and mapping frameworks for ReRAM architectures: PIM-Prune [4] and Pattern-Prune [35]. We do not compare AUTO-PRUNE with SRE [33] because SRE has not shown the inference accuracy of the pruned networks in the paper, making the comparison unfair as AUTO-PRUNE provides both low accuracy loss and high compression rate. Neither do we evaluate Lin et al. [21] and XCS [19] because the evaluation results of PIM-Prune [4] have shown PIM-Prune significantly outperforms them, making the comparisons redundant.

As PIM-Prune and Pattern-Prune are not open-source, we re-implement them as faithfully as possible according to the descriptions in their papers respectively. Although PIM-Prune supports three pruning methods (i.e., SC+XRS, SR+XCS, and block-based pruning), we only implement the block-based pruning method because it exploits the fine-grained block-level sparsity in both

Table 4: The compression rate (CR) comparison of different frameworks on CIFAR10.

Network	Method	CR on XBs	Acc5	Acc Drop
AlexNet	Naive	1	99.36%	
	PIM-Prune	4.3	98.81%	0.55%
	Pattern-Prune	1.1	96.48%	2.88%
	Auto-Prune	**14.3**	**99.10%**	**0.26%**
VGG16	Naive	1	99.29%	
	PIM-Prune	6.1	98.62%	0.67%
	Pattern-Prune	2.6	98.43%	0.86%
	Auto-Prune	**11.9**	**98.62%**	**0.67%**
Plain20	Naive	1	98.14%	
	PIM-Prune	7.3	98.19%	-0.05%
	Pattern-Prune	1.2	98.24%	-0.10%
	Auto-Prune	**10.3**	**98.29%**	**-0.15%**

directions (row and column) while the other two methods can only exploit the sparsity in one direction. PIM-Prune prunes the weights in each block within a fixed compression rate, which may lead to sub-optimal pruning performance. Pattern-Prune uses the pattern pruning algorithm [30] to remove unimportant weights and then applies kernel-reordering methods to map the pruned weight matrices to crossbar arrays. Pattern-Prune only focuses on convolutional layers which usually have a large number of patterns. It fails to utilize the sparsity in fully-connected layers. Consequently, the compression rate of the whole network models may be very low using Pattern-Prune. Furthermore, we also compare Auto-prune to the original system (Naive) where DNN models are not pruned and directly mapped to ReRAM crossbar arrays.

4.2 General Results

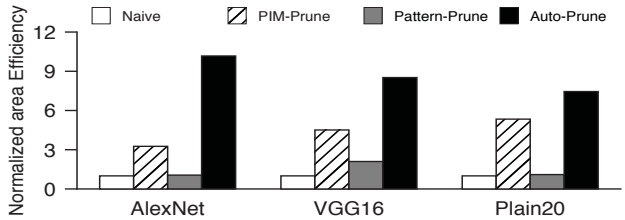
Compression rate. Table 4 shows the compression rate (CR) comparison of different pruning and mapping methods on the CIFAR10 dataset. The CR is defined as the ratio of the number of occupied crossbars (XBs) before pruning to that after pruning. We can see that Auto-prune achieves the highest compression rate among the four methods with the lowest accuracy drops. More specifically, compared to Naive, Auto-prune can get 14.3X, 11.9X, and 10.3X compression rates for AlexNet, VGG16, and Plain20, respectively, with 0.26%, 0.67%, and -0.15% accuracy drops. Its compression rate is up to 2.3X and 12X higher than that of PIM-Prune and Pattern-Prune respectively. Auto-prune has the best pruning performance because of its automated pruning method using reinforcement learning and a fine-grained mapping mechanism.

We also notice that Pattern-Prune has the lowest compression rate. This is because Pattern-Prune is efficient only for convolutional layers but inefficient for fully-connected layers, which have high weight sparsity. In contrast, both Auto-prune and PIM-Prune work effectively for all the layers of the networks. We also observe that AlexNet has the highest compression rate among the networks. The reason is that AlexNet has 5% and 9% higher weight sparsity than VGG16 and Plain20 respectively in our experimental results.

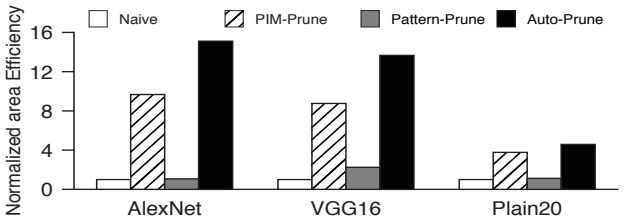
Table 5 shows the compression rates of different pruning methods on the MNIST dataset. For comparison, we use Acc1 to evaluate the accuracy in these tests because all methods achieve similar Acc5 accuracy but distinct Acc1 accuracy on this dataset. Auto-prune also outperforms PIM-Prune and Pattern-Prune. Compared

Table 5: The compression rate comparison of different frameworks on MNIST.

Network	Method	CR on XBs	Acc1	Acc Drop
AlexNet	Naive	1	98.89%	
	PIM-Prune	13.6	98.41%	0.48%
	Pattern-Prune	1.1	97.78%	1.11%
	Auto-Prune	**21.4**	**98.49%**	**0.40%**
VGG16	Naive	1	98.67%	
	PIM-Prune	13.3	98.56%	0.11%
	Pattern-Prune	2.8	98.34%	0.33%
	Auto-Prune	**19.3**	**98.63%**	**0.04%**
Plain20	Naive	1	98.13%	
	PIM-Prune	5.0	97.91%	0.22%
	Pattern-Prune	1.2	97.42%	0.71%
	Auto-Prune	**6.2**	**98%**	**0.13%**



(a) CIFAR10.

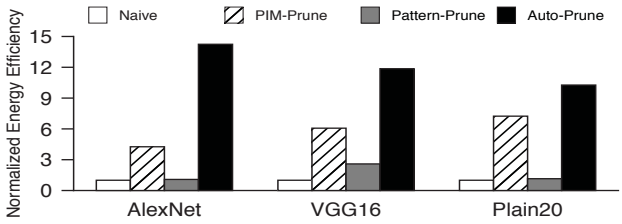


(b) MNIST.

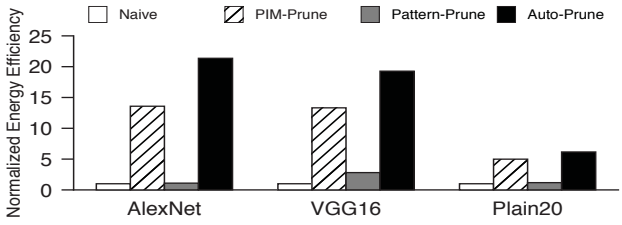
Figure 6: The results of area efficiency on different datasets.

to Naive, Auto-prune achieves 21.4X, 19.3X, and 6.2X compression rates with 0.4%, 0.04%, and 0.13% accuracy drops for the three networks respectively, while the other two methods achieve up to 13.6X, 13.3X, and 5X compression rates with at least 0.48%, 0.11%, and 0.22% accuracy drops. We observe that the compression rates are up to 62% higher on MNIST than those on CIFAR10 for AlexNet and VGG16. This is because the images in the MNIST dataset are easier to be classified than those in the CIFAR10 dataset. The trained networks using the MNIST dataset require a fewer number of parameters to achieve similar accuracy. Consequently, more network weights can be pruned on MNIST. We also notice that the compression rates of Plain20 on MNIST are lower than those on CIFAR10. This is because (1) we choose Acc1 on MNIST which is more sensitive to pruning than Acc5 and (2) the accuracy of Plain20 is more sensitive to pruning than the other two networks.

Area efficiency. Figure 6(a) shows the crossbar area efficiency of different methods on CIFAR10. The area efficiency is normalized to that of the network without pruning (Naive). From Figure 6(a), we can observe that Auto-prune achieves the best area efficiency among the four schemes for all three networks. More specifically, Auto-prune improves the area efficiency by 10.2X, 8.5X, and 7.5X



(a) CIFAR10.



(b) MNIST.

Figure 7: The results of energy efficiency on different datasets.

on AlexNet, VGG16, and Plain20, respectively, while PIM-Prune and Pattern-Prune can improve by up to 3X, 5X, and 6X. These results indicate that our proposed method is efficient to improve crossbar area efficiency.

Figure 6(b) shows the results of ReRAM crossbar area efficiency on MNIST. AUTO-PRUNE improves the area efficiency by 15.1X, 13.7X, and 4.6X for AlexNet, VGG16, and Plain20, respectively. The area efficiency is up to 56% and 13X higher than those of PIM-Prune and Pattern-Prune, respectively. We also observe that the area efficiency of the networks trained on MNIST is up to 60% higher than that on CIFAR10. The main reason is that the pruning ratio with MNIST is higher than that with CIFAR10 for the same network. Therefore, the fewer crossbar arrays make ReRAM-based accelerators more area efficient.

Energy efficiency. Figure 7 shows the energy efficiency comparison among the network models on CIFAR10 and MNIST. The energy efficiency is normalized to that of Naive. We can observe that AUTO-PRUNE achieves up to 2.3X and 12.2X higher energy efficiency compared to PIM-Prune and Pattern-Prune respectively with CIFAR10. For MNIST, AUTO-PRUNE achieves 21.4X, 19.3X, and 6.2X energy efficiency improvement for AlexNet, VGG16, and Plain20 respectively. As AUTO-PRUNE requires up to 95% fewer number of ReRAM crossbar arrays, a smaller number of bitlines, wordlines, ADCs, and DACs are activated, making it more energy efficient than the other pruning schemes.

4.3 Occupied Crossbar Analysis

Table 6 and 7 show the number of occupied crossbars for each layer of the networks on CIFAR10 and MNIST. Because of the space limitation, we only show the results of AlexNet. We have similar observations on other networks. AlexNet has eight layers, where the first five layers are convolutional layers and the last three are fully-connected layers. Because Pattern-Prune may map multiple layers to one crossbar, we cannot compute the number of crossbars for each layer individually. As a result, for Pattern-Prune, we only record the total numbers of occupied crossbars for all convolutional layers and all fully-connected layers respectively. We can observe that different pruning schemes lead to different numbers of occupied crossbars for each layer of the network and AUTO-PRUNE uses the lowest number of occupied crossbars for the whole network. Compared to Naive, AUTO-PRUNE reduces the total number of occupied crossbars by 93%, while PIM-Prune and Pattern-Prune only reduce it by 77% and 7% respectively. Reducing the number of occupied crossbars results in 2.1X and 2.3X higher area efficiency and energy efficiency of ReRAM-based accelerators as shown in Figure 6(a) and Figure 7(a). We also observe that AUTO-PRUNE performs 70% better than PIM-Prune especially for the fully-connected layers (i.e., L_6, L_7, and L_8). This is because the fully-connected layers have higher sparsity (as shown in Figure 9 in Section 4.4) than other layers. The RL agent prunes more unimportant weights in the fully-connected layers with a higher pruning ratio to achieve a global optimum. In contrast, PIM-Prune uses a fixed pruning ratio for all layers without considering the sparsity variation of individual layers, resulting in less amount of pruned crossbar arrays to realize the similar accuracy with AUTO-PRUNE. Furthermore, we find that Pattern-Prune is inefficient for fully-connected layers because it cannot reduce the number of crossbars for such layers.

Table 6: The number of occupied crossbars for each layer of AlexNet on CIFAR10. L_i means layer i of AlexNet.

Method	# of occupied crossbars								
	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	Total
Naive	8	80	336	432	288	2048	8192	256	11640
PIM-Prune	8	24	112	104	72	480	1800	120	2720
Pattern-Prune	conv:280					fc:10496			10776
Auto-Prune	8	40	208	216	120	72	80	72	816

Table 7: The number of occupied crossbars for each layer of AlexNet on MNIST. L_i means layer i of AlexNet.

Method	# of occupied crossbars								
	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	Total
Naive	8	80	336	432	288	2048	8192	256	11640
PIM-Prune	8	16	32	56	40	128	512	64	856
Pattern-Prune	conv:80					fc:10496			10576
Auto-Prune	8	8	24	16	16	96	360	16	544

AUTO-PRUNE has better performance than other pruning schemes with MNIST. Specifically, it reduces the total number of occupied crossbars by 95% compared to Naive, while PIM-Prune and Pattern-Prune only reduce it by 93% and 9% respectively. We also observe that PIM-Prune and AUTO-PRUNE use a fewer number of occupied crossbars with MNIST than that with CIFAR10 for the same network. For example, the total number of occupied crossbars for AUTO-PRUNE reduces from 102 to 68 when the dataset is switched from CIFAR10 to MNIST. This is because MNIST is easier to be classified than the CIFAR10 dataset. AUTO-PRUNE identifies a larger number of trivial elements for pruning in weight matrices with MNIST.

4.4 Sensitivity Studies

Crossbar size. The crossbar size may impact the compression rate of occupied crossbars, the accuracy of networks, the area efficiency, and the energy efficiency of ReRAM-based accelerators. We study

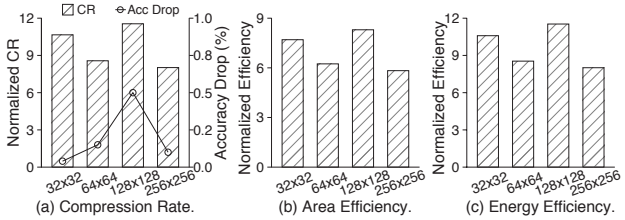


Figure 8: Compression rate, area efficiency and energy efficiency for AlexNet on CIFAR10 with various crossbar sizes.

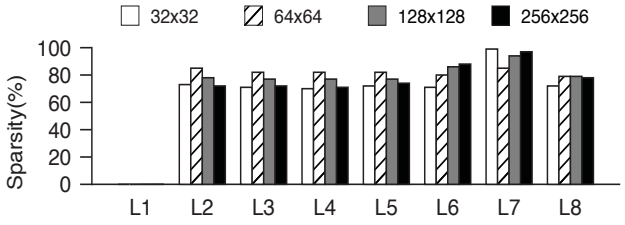


Figure 9: The layer-by-layer sparsity of AlexNet with various crossbar sizes after pruning.

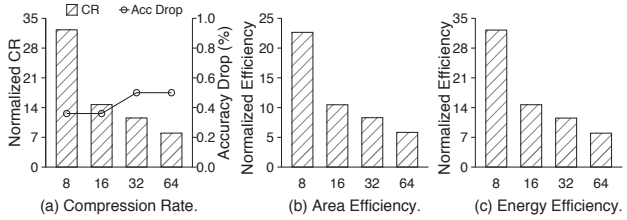


Figure 10: Compression rate, area efficiency, and energy efficiency for AlexNet with various granularities of column-vectors.

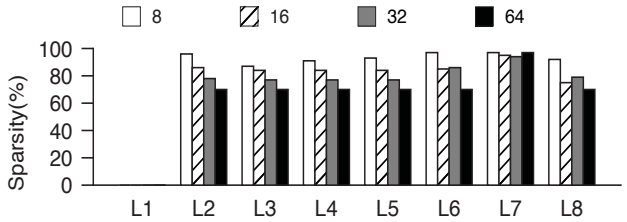


Figure 11: The layer-by-layer sparsity of AlexNet with various granularities of column-vectors after pruning.

the effectiveness of Auto-prune with various crossbar sizes with the CIFAR10 dataset. We only show the results of AlexNet because we observed similar trends for other networks. Figure 8 (a), (b), and (c) show the normalized compression rate, area efficiency, and energy efficiency respectively compared to Naive. We observe that the compression rate of occupied crossbars varies as we increase the crossbar size from 32×32 to 256×256. The crossbar size of 128×128 has the highest compression rate 11.6X. For all the four crossbar sizes, the normalized compression rates are consistently larger than 8 with less than 0.5% accuracy drop. Among the four crossbar sizes, the ReRAM-based accelerator with the crossbar size of 128×128 achieves the highest energy efficiency and area efficiency, as shown in Figure 8(b) and Figure 8(c).

To understand the characteristics of Auto-prune with different crossbar sizes, Figure 9 shows the layer sparsity of AlexNet. For the same crossbar size, we can see that L_6, L_7, and L_8 have higher sparsity on average compared to the other layers, meaning that the fully-connected layers are more sparse and prone to be pruned. For the same layer in AlexNet, the pruning ratios decided by the reinforcement learning algorithm are also varied with different crossbar sizes. To preserve the features in the original input feature map as much as possible, Auto-prune is designed not to prune the first layer. As a result, the sparsity of L_1 equals zero in Figure 9.

The granularity of column-vectors. We study the impact of the granularity of column-vectors on system performance. As Figure 10 shows, as the granularity of column-vectors increases, the compression rate decreases. So do the area efficiency and energy efficiency. When the granularity of column-vectors is 8, Auto-prune achieves the highest compression rate. This is because using finer-grained column-vectors helps exploit the sparsity of weight matrices, improving the chances of pruning unimportant weights. However, the fine granularity makes the hardware architecture more complex in design, incurring higher indexing overhead. In this paper, we choose 32 as the granularity of column-vectors in order to balance performance benefits and hardware overhead.

To understand the characteristics of Auto-prune with various granularities of column-vectors, Figure 11 shows the layer sparsity of AlexNet on CIFAR10 when the granularity increases from 8 to 64. For each layer except L_1 and L_7, as the granularity increases, the layer sparsity decreases. When the granularity of the column-vector is 8, the sparsity of each layer is above 87% and achieves the highest.

4.5 Index Overhead Analysis

To support the data-path of ReRAM-based accelerators in the Auto-prune framework, we need to store the index of the preserved column-vectors of weight matrices in weight index buffers. Assume that the number of elements in weight matrices of the unpruned network is w and the pruning ratio of the network is sp, then the number of preserved elements in the weight matrices after pruning is $\lceil w \times (1-sp) \rceil$. For a given granularity of column-vector g, the number of column-vectors after pruning is $\lceil w \times (1-sp)/g \rceil$. Because the weight index buffer needs to store both x and y coordinates to index each column-vector and we need 5 bits to represent x or y at most for the cases studied in the paper, the total amount of storage space required for the weight index buffer is $2 \times 5 \times \lceil w \times (1-sp)/g \rceil$.

Figure12 shows the storage overhead of the weight index buffer for different networks on CIFAR10 and MNIST. The granularity of column-vector (i.e., g) is 32. The overhead results are derived from w and sp given the specification of networks and the output of Auto-prune with different datasets. For example, the sp is 94% for AlexNet on CIFAR10. Among all the networks, VGG16 with CIFAR 10 uses the largest amount of storage space (162 KB) to store the index structure, leading to 2% storage overhead compared to the original network size.

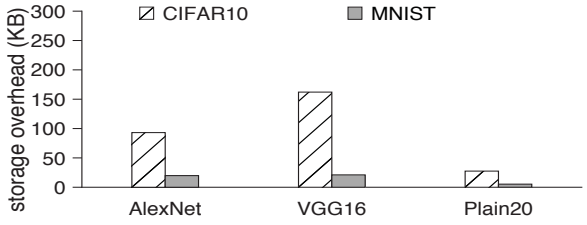


Figure 12: The storage overhead of Weight Index Buffer for different networks on CIFAR10 and MNIST respectively.

5 CONCLUSION

In this paper, we propose a hardware-aware automated DNN pruning and mapping framework, Auto-prune, for ReRAM-based accelerators. It utilizes three techniques to improve the compression rate of ReRAM crossbars. First, it leverages reinforcement learning to automatically determine a global optimum pruning policy given the constraint of accuracy loss. Second, it prunes weight matrices in a finer granularity to exploit the sparsity of weight matrices and only maps the nontrivial weights to crossbars. Third, we devise a new data-path to correctly index and feed input to matrix-vector computation. The data-path skips useless inputs and reduces ineffectual computation. Leveraging the OU mechanism, it solves the dislocation issue. Experimental results show that Auto-prune achieves up to 3.3X compression rate, 3.1X area efficiency, and 3.3X energy efficiency compared to PIM-Prune while maintaining a similar or even higher accuracy. We believe the insights in Auto-prune will inspire the future software and hardware co-design for deep neural networks.

ACKNOWLEDGMENT

This work was supported in part by the Key Scientific Technological Innovation Research Project by Ministry of Education, the Zhejiang Lab Research Project No. 2020KC0AC01, and the National Science Foundation of China No. 62050099.

REFERENCES

[1] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 269–284.

[2] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. 2014. Dadiannao: A Machine-Learning Supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 609–622.

[3] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A Novel Processing-In-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 27–39.

[4] C. Chu, Y. Wang, Y. Zhao, X. Ma, S. Ye, Y. Hong, X. Liang, Y. Han, and L. Jiang. 2020. PIM-Prune: Fine-Grain DCNN Pruning for Crossbar-Based Process-In-Memory Architecture. In *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[5] Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando de Freitas. 2013. Predicting Parameters in Deep Learning. In *Proceedings of the 26th International Conference on Neural Information Processing Systems (NIPS)*. 2148–2156.

[6] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 243–254.

[7] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[8] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. AMC: AutoML for Model Compression and Acceleration on Mobile Devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 784–800.

[9] Yintao He, Ying Wang, Yongchen Wang, Huawei Li, and Xiaowei Li. 2019. An Agile Precision-Tunable CNN Accelerator Based on ReRAM. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–7.

[10] Yintao He, Ying Wang, Xiandong Zhao, Huawei Li, and Xiaowei Li. 2020. Towards State-Aware Computation in ReRAM Neural Networks. In *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[11] Houxiang Ji, Linghao Song, Li Jiang, Hai Li, and Yiran Chen. 2018. ReCom: An Efficient Resistive Accelerator for Compressed Deep Neural Networks. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 237–240.

[12] Yu Ji, Youhui Zhang, Wenguang Chen, and Yuan Xie. 2018. Bridge the Gap Between Neural Networks and Neuromorphic Hardware With a Neural Network Compiler. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 448–460.

[13] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arxiv: 1409.1556* (2014).

[14] Alex Krizhevsky. 2009. *Learning Multiple Layers of Features From Tiny Images*. Technical Report.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification With Deep Convolutional Neural Networks. *Commun. ACM* 60 (2017), 84–90.

[16] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-Based Learning Applied to Document Recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[17] Bing Li, Linghao Song, Fan Chen, Xuehai Qian, Yiran Chen, and Hai Helen Li. 2018. ReRAM-Based Accelerator for Deep Learning. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 815–820.

[18] Sheng Li, Jongsoo Park, and Ping Tak Peter Tang. 2017. Enabling Sparse Winograd Convolution by Native Pruning. *arXiv preprint arXiv:1702.08597* (2017).

[19] Ling Liang, Lei Deng, Yueling Zeng, Xing Hu, Yu Ji, Xin Ma, Guoqi Li, and Yuan Xie. 2018. Crossbar-Aware Neural Network Pruning. *IEEE Access* 6 (2018), 58324–58337.

[20] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous Control With Deep Reinforcement Learning. *arXiv preprint arXiv:1509.02971* (2015).

[21] Jilan Lin, Zhenhua Zhu, Yu Wang, and Yuan Xie. 2019. Learning the Sparsity for ReRAM: Mapping and Pruning Sparse Neural Network for ReRAM Based Accelerator. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. 639–644.

[22] Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. 2019. Multi-Task Deep Neural Networks for Natural Language Understanding. *arXiv preprint arXiv:1901.11504* (2019).

[23] Sparsh Mittal. 2019. A Survey of ReRAM-Based Architectures for Processing-In-Memory and Neural Networks. *Machine learning and Knowledge extraction* 1, 1 (2019), 75–114.

[24] Harry A Pierson and Michael S Gashler. 2017. Deep Learning in Robotics: A Review of Recent Research. *Advanced Robotics* 31, 16 (2017), 821–835.

[25] Songyun Qu, Bing Li, Ying Wang, Dawen Xu, Xiandong Zhao, and Lei Zhang. 2020. RaQu: An Automatic High-Utilization CNN Quantization and Mapping Framework for General-Purpose RRAM Accelerator. In *Proceedings of then 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[26] Christin Seifert, Aisha Aamir, Aparna Balagopalan, Dhruv Jain, Abhinav Sharma, Sebastian Grottel, and Stefan Gumhold. 2017. Visualizations of Deep Neural Networks in Computer Vision: A Survey. In *Transparent Data Mining for Big and Small Data*. 123–144.

[27] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator With In-Situ Analog Arithmetic in Crossbars. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 14–26.

[28] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. Deterministic Policy Gradient Algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning (ICML)*. I–387–I–395.

[29] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. 2017. Pipelayer: A Pipelined ReRAM-Based Accelerator for Deep Learning. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 541–552.

[30] Jingyu Wang, Songming Yu, Jinshan Yue, Zhe Yuan, Zhuqing Yuan, Huazhong Yang, Xueqing Li, and Yongpan Liu. 2020. High PE Utilization CNN Accelerator With Channel Fusion Supporting Pattern-Compressed Sparse Neural Networks. In *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[31] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. HAQ: Hardware-Aware Automated Quantization With Mixed Precision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 8612–8620.

[32] 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 the 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 1–6.

[33] Tzu-Hsien Yang, Hsiang-Yun Cheng, Chia-Lin Yang, I-Ching Tseng, Han-Wen Hu, Hung-Sheng Chang, and Hsiang-Pang Li. 2019. Sparse ReRAM Engine: Joint Exploration of Activation and Weight Sparsity in Compressed Neural Networks. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*. 236–249.

[34] Quanming Yao, Mengshuo Wang, Yuqiang Chen, Wenyuan Dai, Yi-Qi Hu, Yu-Feng Li, Wei-Wei Tu, Yang Qiang, and Yu Yang. 2018. Taking Human Out of Learning Applications: A Survey on Automated Machine Learning. *arXiv preprint arXiv:1810.13306* (2018).

[35] Songming Yu, Yongpan Liu, Lu Zhang, Jingyu Wang, Jinshan Yue, Zhuqing Yuan, Xueqing Li, and Huazhong Yang. 2020. High Area/Energy Efficiency RRAM CNN Accelerator With Kernel-Reordering Weight Mapping Scheme Based on Pattern Pruning. *arXiv preprint arXiv:2010.06156* (2020).

[36] Chengming Zhang, Geng Yuan, Wei Niu, Jiannan Tian, Sian Jin, Donglin Zhuang, Zhe Jiang, Yanzhi Wang, Bin Ren, Shuaiwen Leon Song, et al. 2020. An Efficient End-to-End Deep Learning Training Framework via Fine-Grained Pattern-Based Pruning. *arXiv preprint arXiv:2011.10170* (2020).

[37] Xingyao Zhang, Shuaiwen Leon Song, Chenhao Xie, Jing Wang, Weigong Zhang, and Xin Fu. 2020. Enabling Highly Efficient Capsule Networks Processing Through a PIM-Based Architecture Design. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 542–555.

[38] Zhenhua Zhu, Hanbo Sun, Yujun Lin, Guohao Dai, Lixue Xia, Song Han, Yu Wang, and Huazhong Yang. 2019. A Configurable Multi-Precision CNN Computing Framework Based on Single Bit RRAM. In *Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC)*. 1–6.