Accelerate Inference of CNN Models on CPU via Column Combining Based on Simulated Annealing

Chien-Hung Lin*, Ding-Yong Hong†, Pangfeng Liu*‡§, Jan-Jan Wu†
* Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan
Email: {r09922143, pangfeng}@csie.ntu.edu.tw
† Institute of Information Science, Academia Sinica, Taipei, Taiwan, Email: {dyhong, wuj}@iis.sinica.edu.tw
‡ The Graduate Institute of Networking and Multimedia, National Taiwan University, Taipei, Taiwan
§ The Graduate School of Advanced Technology, National Taiwan University, Taipei, Taiwan

***Abstract*—Convolutional Neural Networks (CNNs) have been successful in various computer vision tasks. However, the size of state-of-the-art CNN models tends to be tremendous, which results in very long inference times and high memory usage. Model compression technology such as unstructured pruning can prune a significant proportion of parameters without affecting accuracy, but efficient utilization of sparsity remains a challenge. Column combining compress unstructurally-pruned CNN models by combining multiple sparse columns in a convolutional filter matrix into a single dense column. In addition, pruning all but the largest magnitude weight in each row of the combined column further compresses the matrix effectively. However, previous work did not address the details of partitioning sparse columns to minimize the negative impact of additional pruning on the performance of the model. In this work, we first prove that the column partition problem is an NP-Complete problem. Next, we propose a column combining scheme based on simulated annealing and global unstructured pruning to minimize the adverse effects of additional pruning on model performance. We implement the acceleration of column-combined CNN models using the TVM AI compiler without special hardware support. The proposed scheme achieves more efficient model compression, leading to a 0.65% improvement in accuracy and a 1.24× faster inference time on VGG19 under 88% sparsity with the TinyImageNet dataset.**

***Index Terms*—deep learning, model compression, pruning, column combining, simulated annealing, TVM**

I. Introduction

Convolutional Neural Networks (CNNs) are an important technology in deep learning. CNNs have achieved impressive results in many computer vision problems, such as image classification [1]–[4], image segmentation [5], [6], and object detection [7], [8]. However, state-of-the-art CNN models become larger for better performance. The huge number of parameters in the model leads to a long inference time and a large amount of memory. For instance, CoCa [9], a state-of-the-art CNN model with 91% top-1 accuracy for the ImageNet dataset, contains 2100 million parameters.

Model compression techniques such as quantization [10] and neural network pruning are common techniques to accelerate the speed of CNN models while maintaining high accuracy. The model pruning methods include *structured pruning* [11]–[13], and *unstructured pruning* [14], [15]. Unstructured pruning prunes at the granularity of a single parameter. Compared with filter-wise or channel-wise structured pruning, such fine granularity allows unstructured pruning to prune many parameters without loss of accuracy.

We cannot accelerate CNN models by setting the weights to zeros only. Even if there are many zeros in the filter tensor, the computation time of the convolution is the same because the size of the tensor and the number of arithmetic operations remain the same. Therefore, it is essential to skip the multiply-accumulate computations (MACs) of the pruned weights to achieve acceleration. However, unstructured pruning prunes weights in an irregular manner, so we cannot directly modify the network's architecture to skip the computations of these weights.

The convolution in a CNN model can be transformed into a matrix multiplication between the feature map matrix and the convolutional filter matrix. Kung et al. [16] described a novel unstructured pruning, called *column combining*, that combines multiple sparse columns in a convolutional filter matrix into a single dense column. The column combining only keeps the weight with the largest magnitude for a row and removes the elements of the same row from the rest of the columns. As a result, the column combining compresses each convolutional filter matrix in CNN models into a dense format, which greatly improves the utilization of hardware resources.

Previous column combining [16] did not explain how to partition sparse columns in order to minimize the adverse effects of column-combine pruning on the performance of the model. In addition, the heuristic column partition in previous work [16] has certain limitations. For instance, it can only deal with one group of columns at a time instead of considering all columns globally. Despite the fact that it results in a good performance for those groups partitioned first since there are more column choices available for finding the best fit, the performance deteriorates for the later partitioned groups due to a lack of choices on available columns. Hence, the algorithm may not find a good solution with such a limited search space.

In this work, we first prove the column partition problem is an NP-Complete problem. Following that, we propose a *simulated annealing* method to combine columns after *global unstructured pruning*. We adopt global unstructured pruning because it leverages sparsity information for each convolutional layer, allowing for the compression of a CNN model without the requirement of a large number of predetermined

2379-1896/23/$31.00 ©2023 IEEE
DOI 10.1109/CANDAR60563.2023.00011

hyperparameters. We use simulated annealing because it can explore a wide range of column partitions to minimize the adverse effects of additional pruning on model performance. Compared to the previous method, our scheme achieves higher model compression, improves model accuracy, and runs faster.

Furthermore, previous work [16] proposed special hardware (i.e., systolic array [17]) to accelerate the matrix multiplication between the feature map and the compressed filter. In this work, we implement the matrix multiplication for the compressed filter on the CPU. We design the optimization of tiling, which partitions the matrices into tiles and computes them separately, and row rearrangement, which reorders the rows of the feature map matrix in order for a sequential access pattern. As a result, the cache utilization becomes more effective and has a better data locality. We leverage the auto-tuner of the TVM AI compiler [18] to search for an optimization schedule of the matrix multiplication and accelerate column-combined CNN computation without special hardware support.

In summary, the contributions of this paper are as follows:

- We prove the column partition problem is an NP-Complete problem.
- We propose a column combining scheme based on simulated annealing and global unstructured pruning.
- We implement and test the optimizations on the CPU for CNNs after column combining with the TVM AI compiler.

We organize the remainder of the paper as follows. Section II introduces the background knowledge of neural network pruning and column combining. Section III describes that the column partition problem is an NP-Complete problem. Section IV describes our column combining scheme in detail. Section V describes the experimental settings and evaluates the improvement of our scheme. Section VI concludes the paper.

II. Background

This section provides an overview of the model pruning techniques in this work.

A. Unstructured Pruning

As deep learning models become increasingly larger and have more parameters, model compression has become a critical technique to reduce model complexity and inference time. One of the model compression methods is unstructured pruning, which involves identifying and removing unimportant weights to reduce the number of parameters. Han et al. [15] proposed a scheme that evaluates the importance of weights globally based on their magnitudes. Weights with smaller magnitudes in the entire model are considered less important on model performance and thus can be pruned. They prune over 90% of the weights from VGG-16 and AlexNet models on the ImageNet dataset without any loss of accuracy [15]. However, unstructured pruning results in irregularly sparse matrices, which require special formats such as CSC (Compressed Sparse Columns) or CSR (Compressed Sparse Rows) to accelerate the inference time by skipping zeros in the matrices. Matrix operations on sparse matrices (such as matrix multiplication) require specialized hardware and library support to obtain effective acceleration.

B. Structured Pruning

Structured pruning aims to address the limitations of unstructured pruning by reducing the number of parameters in a *structured* manner. That is, structured pruning removes weight in *logical units*. For example, structured pruning may remove filters [19], channels [11], or stripes [12] from the model to reduce the number of weights. Hence, structured pruning reduces model size and inference time without the need for specialized hardware or library support. However, the coarse-grained granularity of structured pruning impairs model performance. Under the same sparsity, structurally pruned models usually have lower accuracy than unstructurally pruned models [20].

One of the most commonly used techniques for structured pruning is filter pruning [19]. As depicted in Figure 1, filter pruning not only removes filters considered unimportant but also eliminates the feature maps that are the outputs of pruned filters.

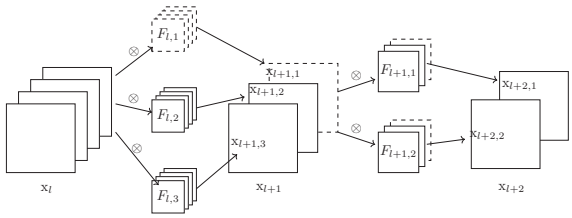

Fig. 1: An illustration of filter removal. Let $x_{l,i}$ be the i-th channel of the *input feature map* in the l-th convolution layer, $F_{l,i}$ be the filter which is convoluted with $x_{l,i}$, and x_{l+1} be the *output feature map* in the l-th convolution layer. Once $F_{l,1}$ has been pruned, $x_{l+1,1}$ will no longer exist, and filter pruning will also remove the first channel from all filters $F_{l+1,j}$ in the subsequent layer.

C. Column Combining

Kung et al. [16] proposed a novel column-combining method for compressing an unstructurally pruned CNN model. This approach first prunes the model in an unstructured manner to increase its sparsity. Then, it partitions the sparse columns in every filter matrix into *column groups*. Finally, it combines columns in a column group into a single dense column, resulting in a compressed model that utilizes hardware resources more efficiently.

A filter matrix typically has tens to hundreds of rows, so it is common for multiple columns to have non-zero weights at the same row, which is called *conflicts* in column combining. It is almost impossible to combine a large number of columns without any conflict. As a result, if a conflict occurs in a row during column combining, we only keep the weight with the largest magnitude on that row and ignore the rest, and we do this for all rows. Compared with unstructured pruning, column combining is more likely to prune weights with larger magnitudes and hurt the performance of the model. Figure 2 depicts an example of column combining. In Figure 2a, Columns of

the same color are partitioned into the same group. There are conflicts that occur in the first row of the red column group. Then Figure 2b shows the result after column combining. In the combined red column, weight 4 is reserved because of the largest magnitude, and the nonzero weights 3 and -1 are removed.

3	-2	4	-1	-3	7
0	1	-1	2	5	6
1	0	0	-5	-4	-8
2	4	-3	0	2	9

(a) Before column combining

4	-3	7
2	5	6
-5	-4	-8
-3	4	9

(b) After column combining

Fig. 2: An illustration of column combining

Kung et al. [16] introduced a dense-column-first combining heuristic inspired by the bin packing problem. The dense-column-first algorithm first sets the maximum number of columns per group for each filter matrix and determines the allowed number of conflicts based on the number of filters in each matrix. Next, the algorithm prunes each filter matrix individually at various levels based on the maximum number of columns per group to increase its sparsity. The dense-column-first policy processes one column group at a time. It first tries to add every column that has not been assigned to the current column group and evaluate the number of nonzero values and conflicts if we assign this column to the current group. Then it selects the column that has the most nonzero values in the combined column while the number of conflicts does not exceed a threshold on the number of conflicts. The selected column then joins the current column group. The selection repeats until the number of columns in this group reaches the maximum number of columns per group, or if the current column group adds any column that has not been assigned a group, will cause the number of conflicts to exceed the maximum number of conflicts allowed. After the algorithm completes the current column group it goes on to build the next column group. By repeating the operations above, the dense-column-first combining policy could partition columns into column groups while limiting the number of conflicts.

D. GEMM-based Convolution

Convolution is a critical operation in CNN models. Convolutional layers account for the majority of model computation. Optimizing convolution operations is therefore essential for model acceleration. Direct convolution slides a kernel map through a feature map by a stride size and multiplies the corresponding elements in both maps and sums them up to obtain the convolution. Since the elements to multiply and sum in the feature maps (i.e., a block) are not continuous in memory, memory access efficiency degrades during convolution computation.

GEMM-based convolution is a technique to improve the memory access efficiency of convolution. It employs im2col [21] to expand each block in feature maps into continuous vectors and reconstruct feature maps into a matrix to improve spatial locality. In addition, it converts the kernel map into a two-dimensional *filter matrix*. The convolution now becomes a matrix multiplication, which can take advantage of many existing matrix multiplication optimizations for speedup. For example, we implement the tiling optimization, which partitions the matrices into small tiles and multiplies them separately. As a result, the cache utilization becomes more effective and has a better data locality, which accelerates the convolution.

III. Proof of Column Partition Problem

This section formally establishes the hardness of the column combining columns. We consider a matrix M of r rows and c columns. Each element in the matrix is either 0 or 1. Two columns are *conflicted* if they both have a 1 in the same row. Now given a constant k, could we partition the columns of the matrix into k groups $\{g_1, \ldots, g_k\}$, so that any pair of columns in a group are *not* conflicted? We will refer to this problem as the *column partition problem*.

Before we establish the hardness of the column partition problem, let us consider a *clique cover* problem. We are given an integer Q and graph $G = (V, E)$. Is it possible to partition V into Q subsets, and all subsets are cliques? It is well-known that the clique cover problem is NP-complete.

Now we show that the column partition problem is NP-complete by reducing the clique cover problem to it. We have a graph $G = (V, E)$ and an integer Q as a clique cover problem instance. We first construct a *complement* graph $H = (V, F)$, where $e \in F$ if and only if $e \notin E$. That means G and H have the same set of nodes, and the edges of H are those missing in G.

We then construct a matrix M of $r = |F|$ rows and $c = |V|$ columns. Each row of the matrix represents an edge in F, and each column of the matrix represents a node in V. We place two 1's into two columns of each row, where the two columns represent the two nodes this edge connects. The rest of the matrix is 0's. Finally, we set the number of groups to Q, as in the clique cover problem. We now consider M and Q as an instance of the column partition problem.

Theorem 1. *The column partition problem is NP-complete.*

Proof. It is easy to see that if we can find a clique cover of at most Q cliques in G, we can partition V in H into Q subset, and all pairs of nodes are disconnected in H. That is, the columns they present are not conflicted in the column partition problem. As a result, we have asolution to the instance of the column partition problem. We can also argue that if we have a solution for the column partition problem instance, we can find a clique cover with Q cliques. The theorem follows. □

We conclude from theorem 1 that it is unlikely that we can partition a set of columns in such a way that no conflict occurs in any column partition. As a result, we will use a simple heuristic, like simulated annealing, for this combinatorial optimization problem.

IV. Scheme

In this section, we propose a column-combining scheme based on simulated annealing and global unstructured pruning. From the experimental results, we observe that the simulated annealing method partitions the columns of the sparse filter matrix and mitigates the impact of conflicts on the model performance.

A. Column Combining based on Simulated Annealing

Our algorithm uses both global unstructured pruning and simulated annealing for column pruning. First, the user specifies the final sparsity percentage s and a α percentage of prune weight for the global unstructured pruning to prune among all pruned weights. That is, the algorithm will prune $s\%$ of the total weight, among which the global unstructured pruning accounts for $s\alpha\%$ within the total pruned weights and the simulated annealing accounts for $s(1-\alpha)\%$. For example, if we want to prune 80% of the total weights and α is 0.6, then the global unstructured pruning will prune $80\% \times 0.6 = 48\%$ of the total weights, and the column combining will prune $80\% \times (1 - 0.6) = 32\%$ of the total weights.

Since our column combining algorithm performs both global unstructured pruning and column combining with conflicts, it is critical to determine how much each of these two prunings should prune. That is, we need to set α carefully. For example, it is obvious that most of the weights pruned by column combining have larger magnitudes than those of the weights pruned by unstructured pruning. When α is small and the column combining needs to prune more weights, more conflicts will occur and the number of column groups in the combined filter matrix will reduce. This results in a more aggressive model compression, which reduces the computation and inference time. However, a denser model compression also reduces model accuracy. Therefore, setting α to balance the amount of pruned weights between the global unstructured pruning and the simulated annealing column combining is crucial.

Even after we set the correct α, it is still not clear how to prune weights among all layers between the global unstructured pruning and the simulated annealing column combining. The reason is that this α pruning percentage may not be suitable uniformly for all layers. Therefore we derive a simple method to determine where this $s\alpha$ and $s(1-\alpha)$ pruned weights should be.

We will prune the model twice. First we start from the dense model, prune away a fixed percentage of remaining weights from the model, and retrain the model to retain the accuracy. This is, instead of pruning away $s\%$ of the weights in one shot, we use iterative pruning [15] to prune and retrain the model until we have pruned $s\%$ of the weights. Note that in the last round of the first pruning we do not retrain the model because we only want to know the number of weights removed in every layer. Next we also start from the dense model and stop when the pruning percentage reaches $s\alpha$. Now we can compare the pruned weight in every layer and determine the number of weights the simulated annealing needs to prune away so that the total pruning percentage is $s\%$.

After knowing the number of weights to prune in every layer, we remove them in two steps. First we use a simulated annealing-based column partition algorithm to partition columns of each convolutional filter matrix into column groups. The goal is to minimize the sum of the magnitudes of the pruned weights due to conflicts for the target layer sparsity in each convolutional layer. Then we prune the second model again by combining multiple columns within a column group, possibly with conflicts, into a single column. This compressing compact the convolutional filter matrix into a dense format.

Now we summarize the column combining algorithm into three stages.

1) We prune the models twice – once to remove $s\%$ of the weights to determine the number of weights to remove in every layer for the next step, and once to remove $s\alpha\%$.
2) We use a simulated annealing-based column partition algorithm to partition columns of each convolutional filter matrix into column groups. Then we prune the second model again by combining multiple columns within a column group into a single column so that we prune away exactly $s(1-\alpha)\%$ of the weights.
3) We retrain the model to recover accuracy lost due to column-combining pruning.

B. Simulated Annealing-based Column Partition

In Section III we show that the column partition problem is an NP-Complete problem. With hundreds to thousands of columns in a convolutional filter matrix, it is computationally impractical to brute-force search the optimal partition of columns so that the number of conflicts is minimal. In this work, we use simulated annealing (SA) [22] to efficiently find a good solution. Simulated annealing is a heuristic algorithm often used to solve optimization problems, such as the traveling salesman problem [23] or bin packing problem [24].

The pseudo-code of the simulated annealing-based column partition algorithm is in Algorithm 1. The conflict count target in the input is the number of weights we want to prune in this layer. The algorithm starts from a random initial state, in which all columns are randomly partitioned into groups. At each iteration, the algorithm generates the neighbor of a

state by randomly exchanging two columns from two different groups. We also randomly move a column to a different group as the neighbor of a state to expand the search space. Then we evaluate the neighbor of a state. The difference of energy is the difference of the sum of magnitudes pruned due to conflicts, and we use it to guide the simulated annealing.

Algorithm 1 SA-based Column Partition Algorithm

Input: Random initial column partition state (s_0); Initial and final temperatures (T_{init}, T_{end}); Cooling factor (f); Number of iterations at each temperature ($iter$); Number of column groups ($group$); Number of conflicts ($conflict$), The conflict count target($target$)

Output: A column partition state after optimization (s_k)

$state \leftarrow s_0$
$temp \leftarrow T_{init}$
$loops \leftarrow 0$
while $temp > T_{end}$ **do**
  $state' \leftarrow neighbor - state\ (state)$
  $\Delta E \leftarrow delta - energy\ (state, state')$
  if $random(0,1) < e^{-\frac{\Delta E}{temp}}$ **then**
    $state \leftarrow state'$
  end if
  $loops \leftarrow loops + 1$
  if $loops = iter$ **then**
    $temp \leftarrow temp \times (1 - f)$
    $loops \leftarrow 0$
    if $conflict > 1.005 \times target$ **then**
      increase $group$ by one
    else if $conflict < target$ **then**
      decrease $group$ by one
    end if
  end if
end while
$s_k \leftarrow state$
return s_k

Differing from the previous method [16], our partition algorithm aims to minimize the *sum of magnitudes* that are pruned due to conflicts, rather than just focusing on reducing the *number* of conflicts. We believe the importance of conflicts varies depending on their magnitudes. If we only consider the number of conflicts but not their magnitudes, we may lose valuable information about the relevance of the conflicts. If the new state prunes a less sum of magnitudes, the new state performs better and will always be accepted. Otherwise, there is still $e^{-\frac{\Delta E}{temp}}$ possibility to accept the new state. Simulated annealing sometimes accepts worse states, making the solution less likely to stick to a local optimum.

The simulated annealing starts from a high temperature T_{init} and gradually cools down to T_{end}. All new states are likely to be accepted at high temperatures, giving the optimizer a wide search space. As the temperature slowly decreases, the state gradually converges. After every $iter$ iterations, the temperature is multiplied by $(1 - f)$ to simulate a cooling process. In this work, T_{init}, T_{end}, f, $iter$ are equal to 1, 10^{-5}, 0.01, 10^6, respectively.

We adjust the number of column groups dynamically from one temperature to the next. First we verify if the current number of conflicts is close to our conflict count target. If the number of conflicts exceeds our conflict count target, then the current number of column groups is too small, so we add an empty column group so that the number of conflicts can be effectively reduced at the next temperature. On the other hand, if the number of conflicts is below our target, it means that the current number of column groups is too large and should be reduced. To do so, we disband the column group with the highest number of conflicts, and individually reassign every column in this group to the group that causes the least conflicts, to minimize the impact of disbanding this group. Also note that we only add or remove one group at a time. This minor modification has minimal impact on the current partition state. In this way, we can dynamically adjust the number of column groups during the simulated annealing process, and prune the number of weights we want in this layer.

C. Matrix Multiplication for Combined Matrix

Our column combining complicates the filter matrix multiplication. In the matrix multiplication, we compute the inner product of the rows of the filter matrix and the columns in the feature map. Within the inner product, we multiply the weight in the i-th column of the filter and the data in the i-th row of the feature map. However, a column after combining consists of weights from various columns. We need to multiply these weights with the data in the corresponding row of the feature map. Hence we use an index matrix to store the column index of each weight for each convolutional layer. That is, the matrix multiplication for a filter matrix after column combining first loads the weight and its column index from the index matrix, then multiplies the corresponding row in the feature map according to the column index. Formally, let A be an m by n filter matrix after column combining, B be an o by p feature map, and D be the m by n index matrix. The output matrix C (multiplication of A and B) is in Equation 1.

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \times b_{(d_{i,k}),j} \tag{1}$$

Figure 3 illustrates an example of multiplication after column combining. The first combined column consists of the first, the eighth, the tenth, and the fifteenth columns. The first weight in the combined column is from the first column, so we multiply it with the first row in the feature map. The second weight is from the tenth column, so we multiply it with the tenth row in the feature map, and so on.

The drawback of the column combining is that the rows that multiply with the same column are not contiguous in the feature map, and the access order of the rows is different from that without column combining. It leads to an irregular access pattern and poor data locality. To improve data locality, we rearrange the rows required for each combined column

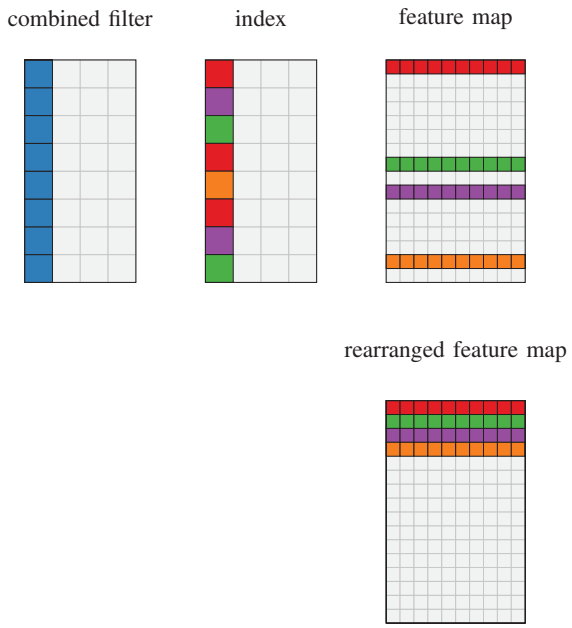

Fig. 3: An illustration of row rearrangement

together, and rearrange the rows according to the access order of the column groups. That is, we place the required rows of the first column group first (the rearranged feature map in Figure 3), then the required rows of the second column group, and so on. As a result, the matrix multiplication becomes more cache friendly and has better performance.

V. Evaluation

In this section, we describe the performance results with our column combining scheme and compare it with the dense-column-first combining in previous work [16].

A. Experiment Settings

1) Models: The experiments use VGG19 [2] and ResNet-50 [3] as our test models. The VGG19 model consists of sixteen convolutional layers with batch normalization and one fully connected layer. The ResNet-50 model consists of fifty-three convolutional layers with batch normalization and one fully connected layer. ResNet-50 implements residual learning, allowing the network to learn residual functions instead of directly learning the underlying mapping.

2) Benchmarks: We evaluate our column combining scheme on two popular datasets, Imagenet [25] and TinyImagenet [26]. Imagenet is one of the largest and most widely used image classification datasets. It consists of almost 1.3 million 224×224 color images in 1000 classes. TinyImagenet is a smaller version of the original ImageNet dataset. It consists of 200 classes, with each class having 500 training images and 50 validation images. All of these are 64×64 color images.

3) Implementation: In this study, we first use the PyTorch framework [27] to train and fine-tune the CNN models. Then, we use the auto-tuner in the TVM AI compiler to optimize the matrix multiplication on the CPU, including the dense and sparse versions. For the dense-column-first combining method, we follow [16] to set the hyperparameters to compress the convolutional layers. Please refer to the experimental parameters in appendix A. We conduct all the experiments on a server with a 16-core, 2.9 GHz Intel Xeon Gold 6226R CPU and 188GB RAM. All models are single-precision. We measure the inference time of each model by averaging 100 inference runs. Note that our method runs global unstructured pruning twice. The first pruning, which determines the sparsity of each layer, requires 20-50 epochs (a large value for higher sparsity). The second pruning and the retrain after column-combine pruning require about 100 epochs, which is typical for training VGG and ResNet on ImageNet.

B. Performance of Different α Percentage

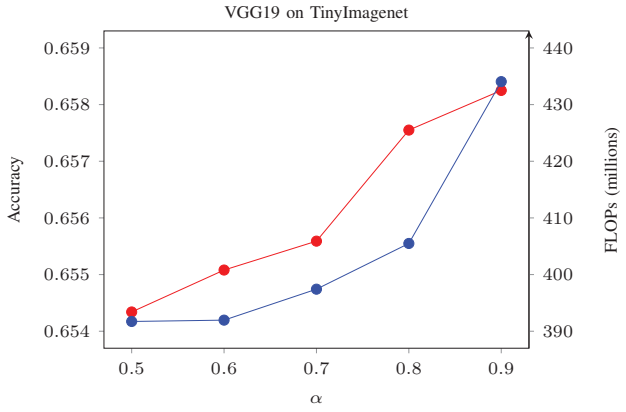

Fig. 4: Accuracy (red line) and FLOPs (blue line) of VGG19 with various α under 90% sparsity on the TinyImageNet dataset.

We first evaluate the performance of the models with different α's under the same sparsity. Figure 4 illustrates the accuracy and the FLOPs (floating-point operations) with various α's under 90% sparsity on the TinyImageNet dataset.

We observe that the model accuracy is lower with a smaller α. When α is small, more weights are pruned by the column-combine pruning due to conflicts. It results in fewer column groups and FLOPs, but also reduces the model accuracy. On the contrary, higher accuracy and FLOPs are achieved with a larger α. By using different α's, we can adjust the model to be more computationally efficient or more accurate.

Models with low sparsity have a slight drop in accuracy after pruning because there are still a large number of weights in the model. In this case, we choose a smaller α to increase model compression. On the other hand, models with high sparsity have a significant drop in accuracy after pruning because there are few weights left in the model, so we should choose a larger α to maintain model accuracy.

In the following experiments, we use the following α values for various sparsity ranges: 0.5 for sparsity less than 60%, 0.6

for 60-70% sparsity, 0.7 for 70-80% sparsity, 0.75 for 80-90% sparsity, and 0.8 for sparsity exceeding 90%.

C. Performance of Column Combining Methods

We describe the accuracy result with our column combining scheme and compare it with the dense-column-first combining in previous work [16].

Figure 5 and Figure 6 respectively show the accuracy of VGG19 and ResNet-50 on the TinyImageNet dataset.

We observe that our column combining scheme outperforms dense-column-first combining on the TinyImageNet dataset. Our column combining achieves 0.65% accuracy improvement on VGG19 with 88% sparsity and 1.51% accuracy improvement on ResNet-50 with 90% sparsity. The reason is that our SA-based column combining can achieve better group partition than the dense-column-first policy.

Figure 7 and Figure 8 present the result on the ImageNet dataset. Our method outperforms dense-column-first combining slightly on VGG19 for most levels of sparsity. As for ResNet-50, it shows a similar accuracy to that of dense-column-first combining.

D. Acceleration of Column Combining Methods

Figure 9 depicts the FLOPs on VGG19 with TinyImageNet. Figure 10 depicts the inference time to process a batch of 32 images in TinyImageNet on VGG19. We observe that our method has significantly less amount of computation than dense-column-first combining at low sparsity, but the FLOPs of both combining methods are similar at high sparsity.

We find that our method prunes the first few convolutional layers, except for the first layer, of the model more aggressively at low sparsity. Although these convolutional layers have a small number of weights, they contain a high proportion of weights with small magnitudes. Global unstructured pruning tends to prune more weights in these layers. Therefore, we compress these layers more aggressively based on the information obtained from global unstructured pruning. As a result, our method achieves less computation and more acceleration on inference at low sparsity.

As the sparsity increases, the sparsity of each layer becomes closer to that of the dense-column-first combining. Moreover, we observe that when the model is further compressed, fewer zero weights remain in the combined filter matrix. That means it becomes less likely we can reduce conflicts through a better partitioning. This is the reason for similar computation and inference time between the two methods when the sparsity is high. Figure 11 and Figure 12 show the results with the ImageNet dataset, which are similar to the results on TinyImageNet.

Figure 13 depicts the FLOPs on ResNet-50 with TinyImageNet. Figure 14 depicts the inference time to process a batch of 32 images in TinyImageNet on ResNet-50. We observe that our method has slightly less computation than dense-column-first combining at low sparsity, and at higher sparsity our method can even have more computation. This is different from the case of VGG19 that our method prunes the first

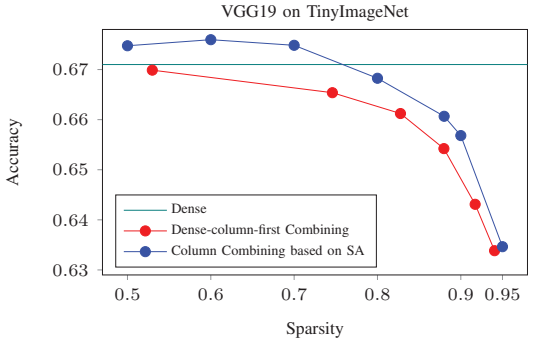

Fig. 5: Accuracy on VGG19 with TinyImageNet dataset.

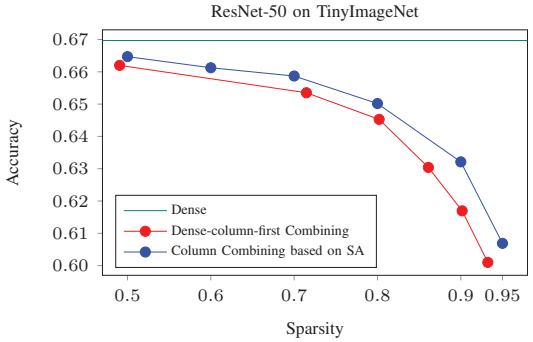

Fig. 6: Accuracy on ResNet-50 with TinyImageNet dataset.

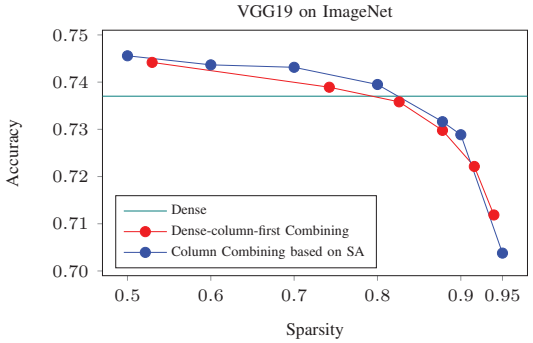

Fig. 7: Accuracy on VGG19 with ImageNet dataset.

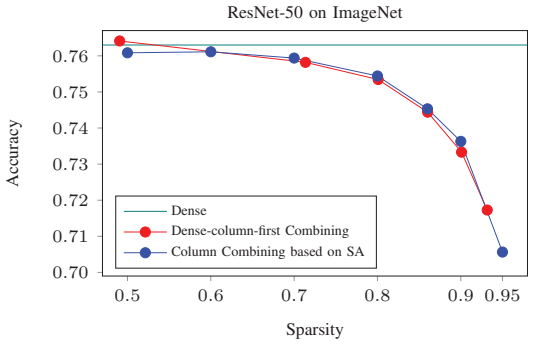

Fig. 8: Accuracy on ResNet-50 with ImageNet dataset.

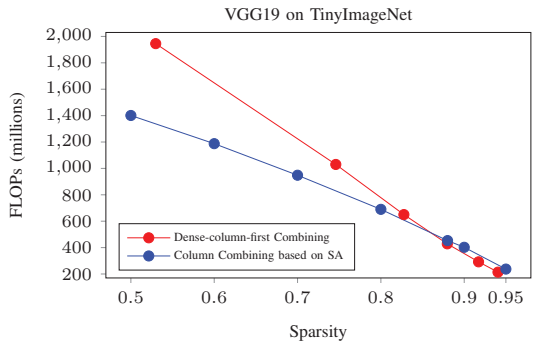

Fig. 9: FLOPs on VGG19 with TinyImageNet dataset.

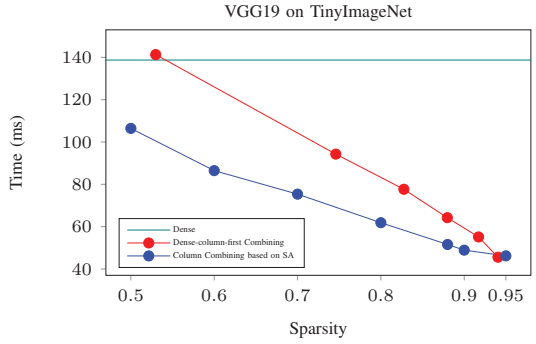

Fig. 10: Inference time required to process a batch of 32 images in the TinyImageNet on VGG19.

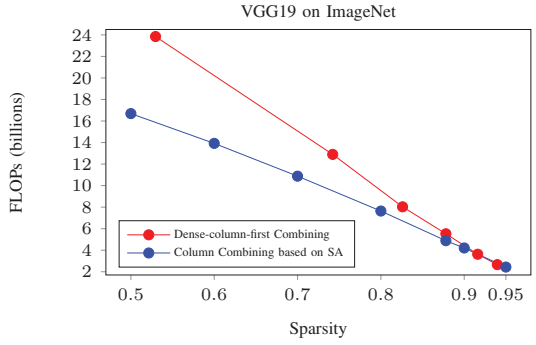

Fig. 11: FLOPs on VGG19 with ImageNet dataset.

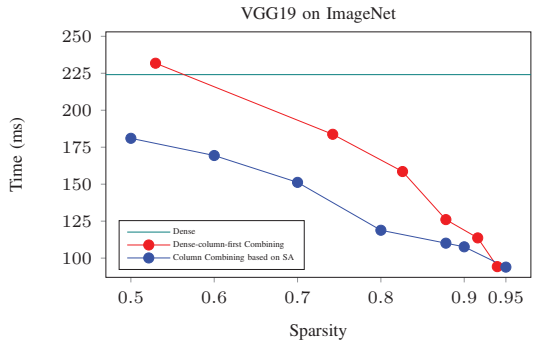

Fig. 12: Inference time required to process a batch of 4 images in the ImageNet on VGG19.

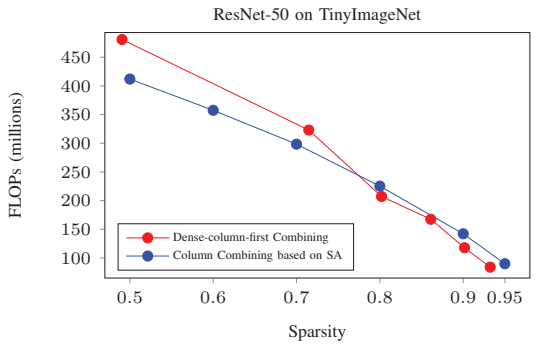

Fig. 13: FLOPs on ResNet-50 with TinyImageNet dataset.

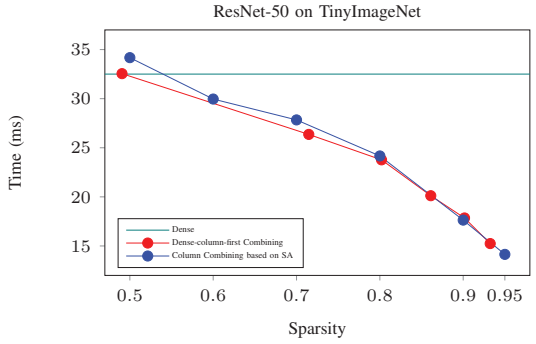

Fig. 14: Inference time required to process a batch of 32 images in the TinyImageNet on ResNet-50.

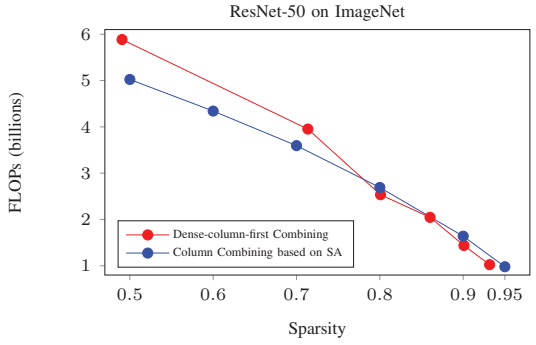

Fig. 15: FLOPs on ResNet-50 with ImageNet dataset.

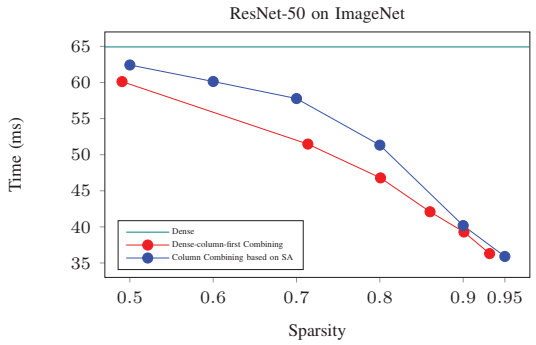

Fig. 16: Inference time required to process a batch of 4 images in the ImageNet on ResNet-50.

few convolutional layers of ResNet-50 more conservatively at low sparsity. These layers contain a high proportion of weights with huge magnitudes, which are usually considered important, and few weights are pruned. Because these layers are only slightly compressed, the reduced computation time from column combining cannot compensate the additional latency caused by memory access. Therefore, we skip the compression of the early layers when performing inference at low sparsity, the same as the dense-column-first combining method. For example, compared to compressing all layers which require 43ms at 50% sparsity, the inference time is reduced to 34ms by skipping those early layers. However, it is important to note that if the compression of the early layers is skipped, we are unable to accelerate the model through computation reduction in those layers. This results in our method having less computation but higher inference time at low sparsity.

Figure 15 and Figure 16 show the results with the ImageNet dataset, which are similar to the results on TinyImageNet.

E. Performance of Row Rearrangement

Fig. 17: The speedup of each convolutional layer of VGG19 with and without row rearrangement

To study the effects of the row rearrangement, we conduct the experiments on VGG19 under 88% sparsity with the TinyImageNet dataset. Figure 17 shows the speedup of row rearrangement for the convolutional layers in VGG19, compared to that without row rearrangement.

The result indicates that the deeper convolutional layers (i.e., the layers close to the output layer) achieve significant performance gain from row rearrangement, but no improvement is observed in the earlier layers. For deeper convolutional layers, the feature maps have more input channels, which lead to more rows. Therefore, the required rows by the combined column group are more likely to be distant from each other, since our column combining scheme can select any column in the filter. As a result, a significant speedup is achieved with improved cache locality by rearranging the rows together.

We also observe that the performance of the second, third, and fourth convolutional layers degrades slightly. The reason is that these layers might not exhibit poor cache locality because they have fewer rows compared with the deeper layers. In addition, the row rearrangement incurs runtime overhead. As a result, the execution time of these layers is increased.

VI. Conclusion

This paper proposes a column combining scheme based on global unstructured pruning and simulated annealing to compress a CNN model. Our method applies global unstructured pruning to leverage model sparsity information and uses simulated annealing search for better partition to minimize the adverse effects of conflict pruning. We introduce a parameter α for users to balance model accuracy and computation efficiency. Furthermore, we implement the matrix multiplication for the combined matrix and optimize it with row rearrangement. We leverage the auto-tuner of the TVM AI compiler to search optimization schedules and accelerate the inference of column-combined CNN on the CPU. Experimental results show that our method achieves a 0.65% improvement in accuracy and a 1.24× faster inference time on VGG19 under 88% sparsity with the TinyImageNet dataset, compared to the dense-column-first method.

There are some possible future works for our column combining. For example, we may prune weights with other metrics instead of magnitudes. There have been several other metrics to determine the importance of weights, such as gradient [28], saliency [29], or entropy [30]. These metrics may identify the importance of weights more precisely or determine the number of weights to be pruned in each layer more accurately. By pruning with these new metrics, we may reduce accuracy loss or achieve more acceleration from the column combining.

VII. Acknowledgement

We would like to thank all colleagues and students who contributed to this study. This research was supported by the National Science and Technology Council of Taiwan, project number MOST110-2221-E-002-072-MY2, NSTC112-2221-E-001-014 and MOST110-2221-E-001-007-MY2. We thank to National Center for High-performance Computing (NCHC) for providing computational and storage resources. We would also like to thank our excellent anonymous reviewers for their invaluable comments.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[4] S. H. Hasanpour, M. Rouhani, M. Fayyaz, and M. Sabokrou, "Lets keep it simple, using simple architectures to outperform deeper and more complex architectures," *arXiv preprint arXiv:1608.06037*, 2016.

[5] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*. Springer, 2015, pp. 234–241.

[6] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.

[7] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7263–7271.

[8] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *Advances in neural information processing systems*, vol. 28, 2015.

[9] J. Yu, Z. Wang, V. Vasudevan, L. Yeung, M. Seyedhosseini, and Y. Wu, "Coca: Contrastive captioners are image-text foundation models," *arXiv preprint arXiv:2205.01917*, 2022.

[10] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.

[11] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 1389–1397.

[12] F. Meng, H. Cheng, K. Li, H. Luo, X. Guo, G. Lu, and X. Sun, "Pruning filter in filter," *Advances in Neural Information Processing Systems*, vol. 33, pp. 17 629–17 640, 2020.

[13] C. Guo, B. Y. Hsueh, J. Leng, Y. Qiu, Y. Guan, Z. Wang, X. Jia, X. Li, M. Guo, and Y. Zhu, "Accelerating sparse dnn models without hardware-support via tile-wise sparsity," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[14] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[15] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," *Advances in neural information processing systems*, vol. 28, 2015.

[16] H. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 821–834.

[17] H. T. Kung and C. E. Leiserson, "Systolic arrays (for vlsi)," in *Sparse Matrix Proceedings 1978*, vol. 1. Society for industrial and applied mathematics Philadelphia, PA, USA, 1979, pp. 256–282.

[18] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, 2018, p. 579–594.

[19] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.

[20] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," *arXiv preprint arXiv:1810.05270*, 2018.

[21] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[22] P. J. Van Laarhoven, E. H. Aarts, P. J. van Laarhoven, and E. H. Aarts, *Simulated annealing*. Springer, 1987.

[23] Z. Wang, X. Geng, and Z. Shao, "An effective simulated annealing algorithm for solving the traveling salesman problem," *Journal of Computational and Theoretical Nanoscience*, vol. 6, no. 7, pp. 1680–1686, 2009.

[24] R. Rao and S. Iyengar, "Bin-packing by simulated annealing," *Computers & Mathematics with Applications*, vol. 27, no. 5, pp. 71–82, 1994.

[25] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, pp. 211–252, 2015.

[26] Y. Le and X. Yang, "Tiny imagenet visual recognition challenge," *CS 231N*, vol. 7, no. 7, p. 3, 2015.

[27] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[28] C. Wang, G. Zhang, and R. Grosse, "Picking winning tickets before training by preserving gradient flow," *arXiv preprint arXiv:2002.07376*, 2020.

[29] N. Lee, T. Ajanthan, and P. H. Torr, "Snip: Single-shot network pruning based on connection sensitivity," *arXiv preprint arXiv:1810.02340*, 2018.

[30] T.-W. Chen, P. Liu, and J.-J. Wu, "Exploiting data entropy for neural network compression," in *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2020, pp. 5007–5016.

APPENDIX A
THE EXPERIMENTAL PARAMETERS FOR DENSE-COLUMN-FIRST COMBINING

VGG19						
Max pruning rate	0.5	0.6	0.7	0.8	0.85	0.9
layers	5, 11	5, 4, 7	5, 4, 7	5, 4, 7	5, 4, 7	5, 4, 7
groups	1, 2	2, 3, 4	3, 4, 6	4, 6, 8	6, 9, 12	8, 12, 16
gamma	0.9	1.4	1.6	1.75	1.85	2
ResNet-50						
Max pruning rate	0.5	0.6	0.7	0.8	0.85	0.9
layers	24, 29	11, 13, 19, 10	1, 10, 13, 19, 10	1, 10, 13, 19, 10	1, 10, 13, 19, 10	1, 10, 13, 19, 10
groups	1, 2	1, 2, 3, 4	1, 2, 3, 4, 6	1, 2, 4, 6, 8	2, 3, 5, 8, 12	2, 4, 8, 12, 16
gamma	0.9	1.4	1.6	1.75	1.85	2