HRBP: HARDWARE-FRIENDLY REGROUPING TOWARDS BLOCK-WISE PRUNING FOR SPARSE TRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

Recently, pruning at initialization and training a sparse network from scratch (sparse training) become increasingly popular. However, most sparse training literature addresses only the unstructured sparsity, which in practice brings little benefit to the training acceleration on GPU due to the irregularity of non-zero weights. In this paper, we work on sparse training with fine-grained structured sparsity, by extracting a few dense blocks from unstructured sparse weights. For Convolutional Neural networks (CNN), however, the extracted dense blocks will be broken in backpropagation due to the shape transformation of convolution filters implemented by GEMM. Thus, previous block-wise pruning methods can only be used to accelerate the forward pass of sparse CNN training. To this end, we propose the Hardware-friendly Regrouping towards Block-based Pruning (HRBP), where the grouping is conducted on the kernel-wise mask. With HRBP, extracted dense blocks are preserved in backpropagation. We further propose HRBP++ to reduce zero kernels by extracting common sparse kernel patterns on all kernels within one block. Extensive experiments on CIFAR-10, CIFAR-100, and ImageNet demonstrate that HRBP (HRBP++) can almost match the accuracy of unstructured sparse training methods while achieving a huge acceleration on hardware.

1 INTRODUCTION

Convolutional Neural Networks (CNN) have accomplished enormous progress on many computer vision tasks, such as classification, detection, and segmentation. However, most successful models are overparameterized and computationally extensive. The excessive computation usually requires tedious training and makes it difficult to deploy cumbersome models into real-world applications. Network pruning (LeCun et al., 1990; Han et al., 2015a;b; Li et al., 2016), which removes unnecessary weights from the heavy dense model, stands as one of the most effective methods to compress a heavy model into a lightweight counterpart while maintaining its accuracy.

Traditionally, network pruning follows a three-step paradigm: 1) training a dense network to convergence; 2) identifying a subset of weights (sparse network) by pruning unnecessary connections; 3) retraining or finetuning the sparse network to recover accuracy. However, dense training is still inevitable in this paradigm. The recent Lottery Ticket Hypothesis (LTH) (Frankle & Carbin, 2019) suggests that sparse network can be trained from scratch (sparse training) to the same accuracy as its original dense model. Consequently, the tedious dense training is unnecessary. During the training process, The sparse structure (sparse mask) can either be static (Lee et al., 2019; Wang et al., 2020; Tanaka et al., 2020) or dynamic (Mocanu et al., 2018; Evci et al., 2020; Liu et al., 2021).

Most sparse training literatures (Lee et al., 2019; Wang et al., 2020; Tanaka et al., 2020; Mocanu et al., 2018; Evci et al., 2020; Liu et al., 2021) explore unstructured sparsity only, where zero weights distribute irregularly. Although unstructured sparsity can maintain the accuracy at a high sparsity ratio, it brings little training time reduction on modern hardware because the irregular mask leads to poor data locality and low parallelism (He et al., 2017; Mao et al., 2017; Wen et al., 2016). An alternative approach, structured sparsity (He et al., 2017; Liu et al., 2017), where the entire filter or channel is pruned, is more hardware-friendly and computationally efficient. However, it usually leads to more accuracy drop compared to unstructured pruning.

Recently, the fine-grained structured pruning becomes popular, which is a trade-off between structured pruning and unstructured pruning. On the one hand, the N:M sparsity (Zhou et al., 2021; Sun et al., 2021) defines blocks to meet requirements that only N weights are non-zero for every continuous M weights, which allows acceleration in the inference phase on modern hardware. The N: M transposable mask (Hubara et al., 2021) further ensures that both the weight matrix W and its transpose W^T follow the same sparsity pattern. Thus it can accelerate both forward pass and backward pass. However, these methods require specialized hardware, *i.e.*, the sparse tensor cores (Zhu et al., 2019). Moreover, the transpose matrix W^T does not describe the backward of CNN accurately. As shown in Fig. 1, the convolution operation is usually implemented with general matrix multiplication (GEMM) on hardware. In this case, calculating the gradient w.r.t inputs requires rotating each kernel first, then conducting kernel-wise transpose, rather than a simple transpose operation (See Sec. 2.1 for more detail). Thus, the transposable masks may not always achieve the expected acceleration on the backward pass of CNN. On the other hand, the regrouping algorithm (block-wise pruning) (Rumi et al., 2020; Yuan et al., 2021; Chen et al., 2022) finds dense blocks by grouping unstructured sparse weights, which can accelerate sparse training on general hardware. However, as shown in Fig. 2, the extracted blocks in forward pass usually cannot be maintained in backward pass. Thus, these methods cannot accelerate the backpropagation as well.

In this paper, we propose the Hardware-friendly Regrouping towards Block-wise Pruning (HRBP) for sparse CNN training. HRBP performs the regrouping algorithm on the kernel-wise mask. Thus, it has the ability to maintain the same dense blocks at both forward and backward pass. Meanwhile, all blocks extracted by HRBP have the same shape, which can alleviate unbalanced workload issues in many-core graphics processing units (GPUs) (Chen et al., 2010). Furthermore, we propose HRBP++ to reduce the number of zero kernels, where all kernels in one group share the same sparse pattern. Specifically, sparse training with fixed HRBP++ can almost match the accuracy of unstructured pruning methods such as SNIP (Lee et al., 2019) and GraSP (Wang et al., 2020), but brings 1.4x and 1.6x overall training acceleration with 90% and 95% sparsity for ResNet. Our main contributions are summarized as follows:

- We detailed analyze the implementation of CNN's forward and backward pass with GEMM, and find that current fine-grained structured pruning methods cannot guarantee the backward acceleration.
- We propose a novel Hardware-friendly Regrouping Block-wise Pruning (HRBP/HRBP++) algorithm that extracts dense blocks from the non-zero weights, while maintaining the spatial regularity of the blocks from the weight transformation of the backward propagation, therefore accelerating both forward and backward of CNN training.
- We conduct extensive experiments on CIFAR-10/100 and ImageNet-1K and demonstrate that sparse training with HRBP can achieve a better trade-off between accuracy and hardware acceleration.

2 Preliminaries

2.1 Convolution Operation and Its Implementation

The weights of a 2D convolutional layer can be defined by $\mathbf{K} \in \mathbb{R}^{C_O \times C_I \times K_h \times K_w}$, where C_O, C_I, K_h and K_w are the number of output channels, the number of input channels, kernel height and kernel weight, respectively. In convolution operation, each filter $\mathbf{K}_c$ slides over the input feature map $\mathbf{I} \in \mathbb{R}^{C_I \times H_I \times W_I}$ and computes a weighted sum of the mapped input values at a time, which generates one activation map $\mathbf{O}_c \in \mathbb{R}^{H_O \times W_O}$. Thus, all C_O kernels conduct C_O times of convolution operations and produce the output map $\mathbf{O} \in \mathbb{R}^{C_O \times H_O \times W_O}$.

Forward pass with GEMM. On hardware, the convolution operation is usually implemented with general matrix-matrix multiplication (GEMM) (Chetlur et al., 2014), where tensor is laid out in the memory in the `NCHW` or the `NHWC` format (See Appendix A for more details). We take the `NCHW` format as an example . As shown in Fig. 1(a), for the input $\mathbf{I}$, the `im2col` operation flattens each convolution window of the input and stacks them as columns in a matrix. Thus, the 2D input feature map $\mathbf{I}$ is unrolled into a matrix $\mathbf{X} = \texttt{im2col}(\mathbf{I}) \in \mathbb{R}^{(C_I K_h K_w) \times (H_O W_O)}$. Meanwhile, $\mathbf{K}$ is reshaped and stored in different rows of another matrix $\mathbf{W} = \texttt{reshape}(\mathbf{K}) \in \mathbb{R}^{C_O \times (C_I K_h K_w)}$.

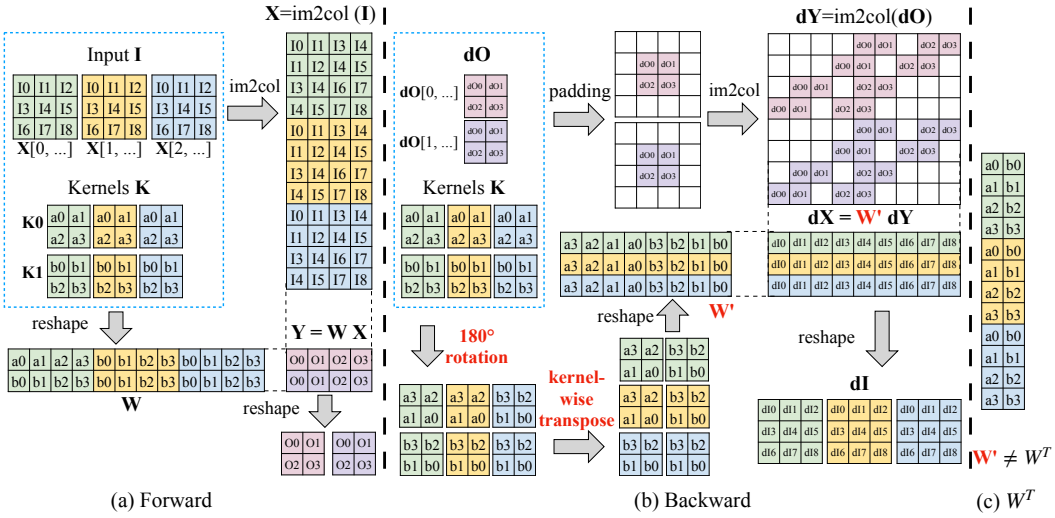


Figure 1: Implementation of forward and backward pass of convolution operation with GEMM in `NCHW` layout. Different color represents different channels. In forward pass, the kernels $\mathbf{K}$ is reshaped to matrix $\mathbf{W}$. In backward pass, each filter is rotated 180° firstly, then the kernel-wise transpose is conducted to obtain the new kernel layout $\mathbf{K}'$. Then $\mathbf{K}'$ is reshaped to matrix $\mathbf{W}'$, which is different from the transpose matrix $\mathbf{W}^T$.

To this end, the forward pass is calculated by $\mathbf{Y} = \mathbf{W}\mathbf{X} \in \mathbb{R}^{C_o \times (H_O W_O)}$. Then the 2D output map is obtained by $\mathbf{O} = \texttt{reshape}(\mathbf{Y})$.

Backward pass with GEMM. Given the gradients of the 2D output map $\mathbf{dO} \in \mathbb{R}^{C_O \times H_O \times W_O}$, the backpropagation involves two matrix multiplications. 1) Calculate the gradients w.r.t. to the kernel $\mathbf{dK}$, which is implemented by $\mathbf{dW} = \mathbf{dY} \cdot \mathbf{X}^T$ following $\mathbf{dK} = \texttt{reshape}(\mathbf{dW})$. 2) Calculate the gradients w.r.t to the input $\mathbf{dI}$, which can be obtained by a full convolution between the kernel $\mathbf{K}$ and $\mathbf{dO}$ (LeCun et al., 1989). In detail, as in Fig. 1(b), we conduct padding and `im2col` operation on $\mathbf{dO}$, and obtain $\mathbf{dY} = \texttt{im2col}(\mathbf{dO}) \in \mathbb{R}^{(C_O K_h K_w) \times (H_I W_I)}$. Meanwhile, we flip each kernel first vertically and then horizontally (*i.e.*, 180° rotation) and perform the kernel-wise transpose to get the new kernel layout $\mathbf{K}'$. Then we reshape $\mathbf{K}'$ to matrix $\mathbf{W}' \in \mathbb{R}^{C_I \times (C_O K_h K_w)}$. Thus, the gradient is calculated by $\mathbf{dX} = \mathbf{W}'\mathbf{dY}$. Finally, we have $\mathbf{dI} = \texttt{reshape}(\mathbf{dX})$.

Discussion. Many literatures denote the forward pass to $Y = WX$ and the backward pass to $dX = W^T dY$ for simplicity, where $W \in \mathbb{R}^{C_O \times C_I}$ and $X \in \mathbb{R}^{C_I \times B}$. Here B is the number of input vector. However, this simple form cannot describe CNN accurately. As in Fig. 1(c), $W^T \in \mathbb{R}^{C'_I \times C_O}$ is different from the matrix $\mathbf{W}'$. If and only if $K_h = K_w = 1$, they are the same, which makes the CNN a linear layer instead. Thus, the backward pass of CNN cannot be linearly simplified to $dX = W^T dY$. Consequently, sparse pattern based on W^T (Hubara et al., 2021) may not always obtain the expected acceleration on arbitrary CNN backpropagation. In practice, this algorithm has additional requirements on the shape of CNN and the layout in the memory. See Appendix B for more details.

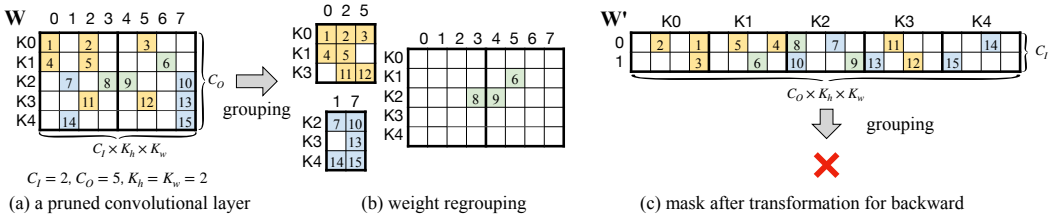


Figure 2: Example of weight regrouping on convolutional layer. Indexed cells are non-zeros. Different color represents different blocks. The extracted dense blocks cannot be kept in the backward pass .

2.2 WEIGHT REGROUPING ON UNSTRUCTURED SPARSITY

Given a CNN $f_\theta(\cdot)$ with weights θ, a sparse subnetwork is defined by $f_{\theta \odot m}(\cdot)$ with a binary pruning mask $m \in \{0,1\}^{|\theta|}$, where $\odot$ is the element-wise product. In unstructured pruning, zeros distribute irregularly in mask m. To this end, the convolution operation still needs to maintain the same size kernel matrix as dense network. Thus, it is difficult to reduce the computation of unstructured sparse weights on hardware. Even with the help of the dedicated sparse matrix representation technique such as CSR format (Buluç et al., 2009), the unstructured sparsity still expects to have over 85% sparsity ratio to acquire limited acceleration since the irregular weight distribution causes great computation overhead due to poor data locality (Yuan et al., 2021). The recent weight regrouping (reorganization) (Rumi et al., 2020) can accelerate the unstructured sparse weights on hardware by extracting multiple smaller dense blocks, which can improve the throughput with GEMM.

Implementation. Given a sparse kernel matrix $\mathbf{W} \odot m$, weight regrouping finds similar rows and columns and bring them together. In detail, it firstly clusters the C_O rows into several groups based on the Jaccard similarity among non-zeros columns. For each group, it then picks out columns with most non-zeros weights from all $C_I K_h K_w$ columns and generates the dense block. For example, in Fig. 2(a), we can group kernel K0, K1, and K3 together, and select columns with at least two non-zero (*i.e.*, 0th, 2th, and 5th columns), which leads to the orange dense block in Fig. 2(b). The remaining sparse block (cells with green color) are usually discarded (Chen et al., 2022).

Limitations. However, the regrouping in (Rumi et al., 2020) cannot be applied to the training directly. 1) As shown in Fig. 2(c), the dense blocks are fragmentary in backward pass due to the transformation of kernel matrix, making the backward acceleration unfeasible. For example, the orange dense block found in $\mathbf{W}$ are scattered irregularly on $\mathbf{W}'$. One naive solution is extracting different blocks in forward and backward separately. However, it is possible that specific weights in the blocks of forward are removed in the blocks of backward, which obstructs backpropagation of the weight. 2) Besides, the shape of dense blocks are arbitrary, which introduces imbalance memory access and data locality. Thus, it makes the GPU suffer great workload imbalance.

3 METHODOLOGY

3.1 HARDWARE-FRIENDLY REGROUPING FOR BLOCK-WISE PRUNING (HRBP)

Motivation. The reason for fragmentary blocks in the backward of CNN is that all $C_I K_h K_w$ columns are considered independently. Thus, one dense block can select arbitrary locations on the kernel of different input channels. To address this issue, the grouping algorithm should locate identical elements for different input channels within one block. When $K_h = K_w = 1$, the CNN reduces to a linear layer and the fragmentary blocks issue can be solved naturally. Inspired by this, we propose HRBP to extract dense blocks that can also be kept in CNN backpropagation.

HRBP. Different from (Rumi et al., 2020) that conducts regrouping on the mask of kernel matrix $m \in \mathbb{R}^{C_O \times (C_I K_h K_w)}$, HRBP conducts regrouping on the kernel-wise mask $m' \in \mathbb{R}^{C_O \times C_I}$ of weights, which is obtained by counting Thus, the whole weights of one kernel with size $K_h K_w$ are kept or discarded based on m'. As in Fig. 3(a), given an unstructured mask m with dense ratio r, we firstly generate its corresponding kernel-wise mask m' by counting non-zeros cells in each kernel and keeping $rC_I C_O$ kernels with most non-zero cells. We then cluster C_o rows into t groups with equal size. For each group, we select $r \cdot C_I$ input channels with the most non-zero weights to generate one dense block. Lastly, we discard the remaining elements and refill zero slots in dense blocks. To this end, we have t dense blocks with identical shape $\frac{C_o}{t} \times (rC_I K_h K_w)$. As in Fig. 3(a), the same groups of weights are preserved in backpropagation, making the sparse training with dense block applicable. See Appendix C for the pseudocode of HRBP.

HRBP++. The vanilla HRBP inevitably introduces too many zero kernels, which may result in a heavy accuracy drop like channel-wise pruning. To this end, we further propose HRBP++, aiming to keep more non-zero kernels. Similar to HRBP, HRBP++ also conducts regrouping on m'. For each row group, we selects $r/s \cdot C_I$ input channels with the most non-zero values, where s is the dense ratio of each kernel with size $K_h K_w$. In practice, s needs to satisfy $s \geq r$ and $s \in \{i/(K_h * K_w)\}_{i=1}^{K_h * K_w}$. We then extract a common pattern from kernels within one group based on the unstructured mask. In detail, for each group, we count the number of non-zero weights in each cell of the kernel, and choose $sK_h K_w$ cells with most non-zero weights as a pattern for the

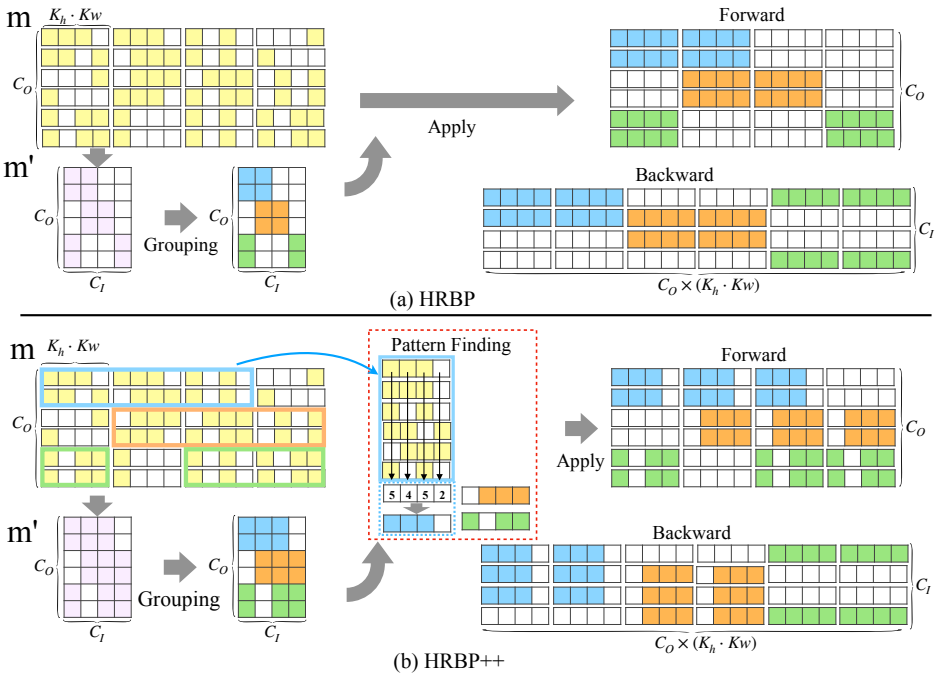


Figure 3: Illustration of HRBP and HRBP++. White cell means zero weights. Given the unstructured mask (matrix with yellow cell), HRBP(++) firstly extracts the kernel-wise mask (matrix with lavender cell) by counting non-zero weights of each kernel. We then group the kernel-wise mask into several equal shape blocks (we have three groups marked with blue, orange, and green). In HRBP, the entire weights of a kernel are kept or discarded. In HRBP++, we extract common patterns of each block and apply it to all kernels within the group.

dense block. For example, in Fig. 3(b), one kernel has 4 cells. Given the 6 kernels from the blue group, the number of non-zero weights is $5, 4, 5, 2$ for the 4 cells, respectively. Thus, we select the $1, 2, 3$-th cell as a common pattern and apply it to all kernels within the blue group. To this end, HRBP++ allows for zero values within kernels, resulting in fewer entirely discarded kernels for the same overall sparsity ratio.

3.2 Sparse Training with HRBP (++)

HRBP(++)-based Random Mask. With HRBP(++), we can extract dense blocks from sparse weights and accelerate both forward and backward of sparse training. For simplicity, we apply HRBP(++) to a random pruning mask, which randomly choose connections to satisfy the sparsity ratio requirements. To this end, we design a strategy to create random pruning mask in HRBP (++) style, named *HRBP(++)-based Random Mask*. For the sparse ratio of each layer, we follow the Erdős–Rényi-Kernel (ERK) (Evci et al., 2020) where larger layers are allocated with higher sparsity than smaller layers within a network. For HRBP, we randomly split the C_O rows into t equal groups. For each group, we randomly keep rC_I channels and zero out the rest. For HRBP++, we start from a random unstructured mask m and apply the proposed HRBP++ algorithms to slightly modify m. Thus, the HRBP(++)-based random mask can be implemented with several dense blocks on GPU and accelerate the training process.

Block-wise updating for Dynamic Sparse Training (DST) DST starts from a random sparse network. After optimizing several iterations, it prunes a portion of weights based on the pruning criterion, and grows new connections according to the grow criterion. Then the new sparse network is trained until the next update. Typically, any location of the entire weight can be pruned or activated (*i.e.*, grown). However, this default setting is not aligned to HRBP, since updating connections at arbitrary locations destroys the extracted blocks. To this end, we propose the block-wise updating for DST with HRBP++, which follows the prune-grow-regroup scheme. As shown in Fig. 4, we maintain the cluster of rows (filters). For each row, we prune $d\%$ weights based on magnitude and

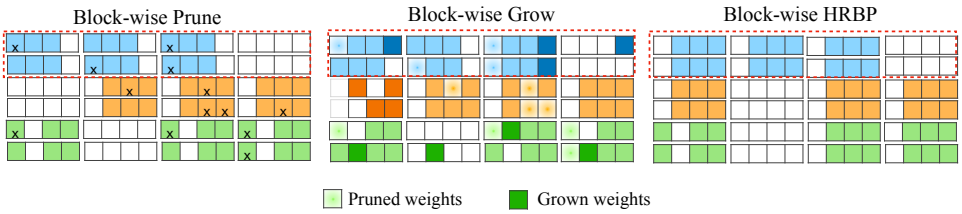


Figure 4: Illustration of Block-wise updating for Dynamic Sparse Training.

grow back new $d\%$ weights based on its gradient. Then we perform the HRBP++ on the new weights to group the new mask into several dense blocks.

3.3 HRBP Hardware Implementation

The extracted dense blocks from HRBP(++) have the same shape. To this end, in our kernel design, a thread block only processes the output channels of the same group which enables us to have better reuse of the input data (e.g., the input data can be loaded to shared memory which can be access by all the threads inside a thread block). Inside a thread block, each thread is responsible for one portion of the output on (X, Y) dimension which further helps us to achieve good data reuse of the kernel weight (kernel weight can also be put into shared memory).

We also applied tiling across both height and width dimension of the input channel to increase the number of launched thread blocks, which further increases the parallelism level. The tiling size plays a significant role of the performance. A small tiling size can increase the parallelism level, while a big tiling size will provide better reuse of the weight data. Thus, finding a good tile size for a given problem is non-trivial. A simple but powerful method is brute force search, it guarantees to find out the best tiling size for specific hardware. In our design, input data are put into shared memory and can be accessed by all thread block to improve the data reuse. In the meantime, the cost by using brute force search is very low because most of the undesired tiling size candidates are discarded in order to satisfy the usage requirement of the shared memory.

4 Experiments

4.1 Experimental Setups

Dataset & Networks. We follow the experimental settings in (Wang et al., 2020; Yuan et al., 2021) and conduct experiments on CIFAR-10, CIFAR-100 and ImageNet-1K (Deng et al., 2009). As the common setting in sparse training, we apply the WideResNet-32-2 and VGG-19 (Simonyan & Zisserman, 2014) for CIFAR-10/100 and ResNet-50 (He et al., 2016) for ImageNet.

Training. For CIFAR-10/100, we use a batch size of 128 and train networks with SGD optimizer for 160 epochs by default. The learning rate is set to 0.1 initially and is decayed by a factor of 0.1 at the 80th and 120th. Moreover, we run each experiment 3 times and report the mean value and standard derivation. For ImageNet, we adopt the Pytorch official implementation and train the networks for 100 epoch as (Wang et al., 2020). The learning rate is 0.1 initially and is decayed at 30-th, 60-th, and 90-th epoch with factor 0.1. For sparse training, we set the number of dense blocks t to 8 (Rumi et al., 2020) and the dense ratio of kernel s in HRBP++ to $\frac{4}{9}$ for all 3×3 kernels. The minimum size of a block B_1 is set to 8 (Rumi et al., 2020).

Hardware Our evaluation platform is NVIDIA Ampere A100 (108 SMs, 40GB) machine running Ubuntu 18.04.4 LTS. The versions of CUDA and cuDNN are 11.0.0 and 8.0.4, respectively. For baseline implementations of convolution operations for full sparse weights, we adopt the widely used GEMM-based convolution, i.e., `CUDNN_CONVOLUTION_FWD_ALGO_GEMM` (Chetlur et al., 2014). We report the end-to-end relative training acceleration compared with the dense baselines, which includes both forward and backward calculation on hardware.

4.2 Sparse Training with HRBP(++)-based Random Mask

Settings. We first investigate HRBP/HRBP++ on sparse training with fixed masks. We compare it against random unstructured pruning (RU) and random channel-wise pruning (RC) with ERK ratio.

Table 1: Test accuracy and overall training acceleration of pruned ResNet-32 and VGG-19 on CIFAR-10 and CIFAR-100 datasets. All methods apply a fixed mask during the training. The number in brackets is the relative training acceleration compared to a dense model.

Dataset	CIFAR-10		CIFAR-100	
Pruning ratio	90%	95%	90%	95%
ResNet-32	Dense: 94.80		Dense: 74.64	
RU	92.81±0.19 (1.0×)	91.38 ±0.04 (1.0×)	69.48±0.21 (1.0×)	67.03±0.66 (1.0×)
LTH (Frankle & Carbin, 2019)	92.31 (1.0×)	91.06 (1.0×)	68.99 (1.0×)	65.02 (1.0×)
SNIP (Lee et al., 2019)	92.59 (1.0×)	91.01 (1.0×)	68.89 (1.0×)	65.22 (1.0×)
GraSP (Wang et al., 2020)	92.38 (1.0×)	91.39 (1.0×)	69.24 (1.0×)	66.50 (1.0×)
RC	90.27±0.24 (1.5×)	87.19±0.48 (1.6×)	62.71±0.22 (1.5×)	56.33 ±0.37 (1.6×)
Regrouping (Rumi et al., 2020)	91.63±0.11 (1.1×)	90.77±0.07 (1.2×)	66.97±0.18 (1.1×)	64.35±0.45 (1.2×)
HRBP	91.76±0.17 (1.4×)	89.80±0.02 (1.6×)	67.20±0.22 (1.4×)	63.19±0.73 (1.6×)
HRBP++	92.30±0.20 (1.4×)	90.84±0.41 (1.6×)	69.22±0.50 (1.4×)	65.94±0.25 (1.6×)
VGG-19	Dense: 94.23		Dense: 74.16	
RU	92.91±0.10 (1.0×)	91.91±0.13 (1.0×)	70.39±0.43(1.0×)	68.63±0.40 (1.0×)
LTH (Frankle & Carbin, 2019)	93.51 (1.0×)	92.92 (1.0×)	72.78 (1.0×)	71.44 (1.0×)
SNIP (Lee et al., 2019)	93.63 (1.0×)	93.43 (1.0×)	72.84 (1.0×)	71.83 (1.0×)
GraSP (Wang et al., 2020)	93.30 (1.0×)	93.04 (1.0×)	71.95 (1.0×)	71.23 (1.0×)
RC	90.84±0.38 (1.7×)	87.12±0.18 (1.8×)	59.61±0.52 (1.7×)	49.31±1.17 (1.8×)
Regrouping (Rumi et al., 2020)	92.81±0.25 (1.2×)	91.86±0.24 (1.2×)	70.52±0.36 (1.2×)	68.60±0.08 (1.2×)
HRBP	92.48±0.19 (1.4×)	90.81±0.16 (1.9×)	68.42±0.17 (1.4×)	64.98±0.47 (1.9×)
HRBP++	92.88±0.12 (1.4×)	91.66±0.14 (1.9×)	70.25±0.29 (1.4×)	67.89±0.49 (1.9×)

Meanwhile, we compare it against unstructured pruning methods with well-defined criteria such as SNIP (Lee et al., 2019) and GraSP (Wang et al., 2020). We also conduct experiments with the conventional regrouping method (Rumi et al., 2020) as a baseline, even if it fails to accelerate the backpropagation. Following (Wang et al., 2020), we set the sparse ratio to 90% and 95%.

Results. The results are summarized in Table 1. From the accuracy perspective, HRBP++ achieves similar performance to unstructured pruning methods, which is much better than the channel-wise pruning. For example, at 95% sparsity, HRBP++ achieves an accuracy of 90.84% on CIFAR-10 with ResNet-32, which is 3.65% higher than the RC and just 0.17% negligible drop to SNIP (Lee et al., 2019). Meanwhile, HRBP, as our baseline with more zero kernels, usually suffers 1% accuracy drop compared to HRBP++, but it still performs much better than RC. For the overall training acceleration, unstructured pruning can not be accelerated on hardware due to its irregularity of non-zero weights. However, HRBP can almost match the acceleration of channel-wise pruning, with the benefit of our optimized convolution implementation. Moreover, HRBP obtains more acceleration compared to the convention regrouping (Rumi et al., 2020), as the later can only accelerate the forward computation. In summary, our HRBP(++) successfully combines the advantages of both unstructured pruning and channel-wise pruning.

Layer-wise speedup. We also evaluate the layer-wise speedup of each convolutional layer, which includes the acceleration in both forward and backward pass, for a detailed exploration. We take VGG-19 with 90% overall sparsity as an example, and report the speedup of each layer in Figure 5. At shallow layers with small sparse ratio, channel-wise pruning performs slightly better than ours. However, at deeper layers, HRBP achieves comparable or even better results, with at most 7.5x acceleration on `Conv14`. This is consistent with the conclusion in (Rumi et al., 2020) that grouping algorithm prefers CNNs with a large number of kernels. Noticeably, common tools like cuDNN are only optimized for kernel matrices with a multiple of 32 rows, thus channel-wise pruning with an arbitrary number of channels does not guarantee better acceleration (Rumi et al., 2020).

4.3 Dynamic Sparse Training with HRBP++

We further explore the dynamic sparse training with HRBP++, which prunes and grows the connections based on the proposed block-wise updating. We compare the accuracy of DST with HRBP++ to several DST methods with unstructured pruning, including DeepR (Bellec et al., 2018), DSR (Mostafa & Wang, 2019), SET (Mocanu et al., 2018), and RigL (Evci et al., 2020). We follow all sparse training hyperparameters in (Evci et al., 2020). As the speedup is similar to sparse training

Table 2: Comparison of dynamic sparse training methods on CIFAR-10 and CIFAR-100.

Network	ResNet-32				VGG-19			
Dataset	CIFAR-10		CIFAR-100		CIFAR-10		CIFAR-100	
Pruning ratio	90%	95%	90%	95%	90%	95%	90%	95%
Deep-R (Bellec et al., 2018)	91.62	89.84	66.78	63.90	90.81	89.59	66.83	63.46
DSR (Mostafa & Wang, 2019)	92.97	91.61	69.63	68.20	93.75	93.86	72.31	71.98
SET (Mocanu et al., 2018)	92.30	90.76	69.66	67.41	92.46	91.73	72.36	69.81
RigL (Evci et al., 2020)	92.84±0.13	92.02±0.29	70.98±0.30	68.50±0.15	93.15±0.09	92.30±0.43	71.63±0.28	69.13±0.46
DST-HRBP++	92.72±0.23	91.25±0.12	69.51±0.52	66.41±0.30	93.07±0.12	91.79±0.18	70.49±0.19	68.04±0.37

with fixed mask, we only show the accuracy comparison in Table 2 for simplicity. Specifically, in DST, our HRBP++ can still match the accuracy of these DST methods. Nevertheless, when comparing results in Table 1 and 2, in HRBP++, the advantage of DST is not as much as that in unstructured pruning. One potential reason is that the mask updating mechanism of HRBP++ has a smaller mask diversity (Hubara et al., 2021) than the unstructured pruning.

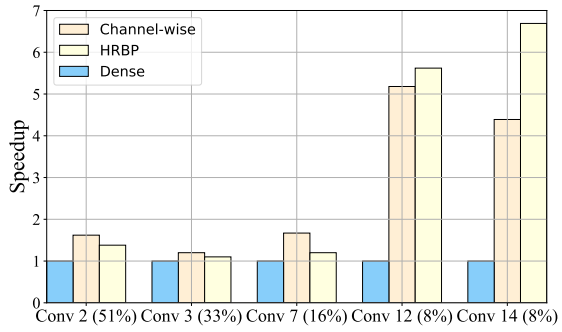


Figure 5: The layer-wise performance comparison of convolution operations in VGG-19 on CIFAR-10.

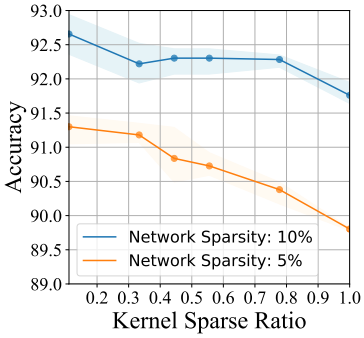


Figure 6: Different kernel dense ratio for HRBP with ResNet-32.

5 Results on ImageNet

We evaluate the static sparse training with HRBP(++)-based Random Mask on ImageNet. Follow Lee et al. (2019); Wang et al. (2020), we set the sparse ratio to 60% and 80%. As shown in Table 3, HRBP++ can also achieve similar performance as well-designed unstructured pruning methods such as SNIP Lee et al. (2019) and GraSP Wang et al. (2020). Meanwhile, it provides 1.26× and 1.44× training acceleration on hardware when the pruning ratio is 80% and 90% respectively. To this end, our HRBP++ is still effective on complicated tasks like ImageNet.

Table 3: Test accuracy and overall training acceleration of pruned ResNet-50 on ImageNet datasets. All methods apply a fixed mask during the training. The number in brackets is the relative training acceleration compared to a dense model.

Pruning ratio	60%		80%	
Accuracy	top-1	top-5	top-1	top-5
ResNet-50 (baseline)	75.70	92.81	-	-
SNIP Lee et al. (2019)	73.95 (1.0×)	91.97	69.67 (1.0×)	89.24
GraSP Wang et al. (2020)	74.02 (1.0×)	91.86	72.06 (1.0×)	90.82
HRBP++	74.84 (1.17×)	92.35	70.90 (1.26×)	89.93

5.1 Ablation Studies

Dense Ratio of Kernel Pattern s. We take the ResNet-32 on CIFAR-10 as an example to explore the effect of the kernel sparse ratio s in HRBP++. Typically, a smaller s introduces more non-zero kernels but inevitably reduces the number of non-zero weights within each kernel. As the shape of most kernels is 3×3, we explore s from $\frac{1}{9}$ to $\frac{9}{9}$. And we also set the overall density ratio r to 10% and 5%. As shown in Fig. 6, with a smaller s, sparse training with HRBP++ achieves slightly better

results, especially for network with large sparsity. This is reasonable as a smaller s can reduce the number of zero kernels, which even renders a network untrainable (Tanaka et al., 2020).

6 RELATED WORK

6.1 NETWORK PRUNING

Pruning is a popular method to compress overparameterized networks into a lightweight one. Based on the distribution of zero weights, pruning can be divided into three types: 1) *Unstructured pruning*, where zero weights are distributed at arbitrary locations based on the important score of each weight. The score can be obtained from magnitude (LeCun et al., 1990; Han et al., 2015a;b; Xu et al., 2021; Zhu & Gupta, 2017), gradient (Molchanov et al., 2017; 2019) or Hessian (LeCun et al., 1990). The unstructured pruning can match the accuracy of dense networks at a high sparsity ratio (more than 90%). However, it is difficult to achieve better hardware performance due to the irregularity. 2) *Structured pruning*, where the weights of entire channels are pruned. Earlier works (Li et al., 2016; He et al., 2017; Wen et al., 2016; Liu et al., 2017) adopt mathematics-oriented regularization-based algorithms to generate sparsity. Other works such as HRank (Lin et al., 2020), SCOP (Tang et al., 2020), DMCP (Guo et al., 2020), MetaPruning (Liu et al., 2019) use complicated rules to generate the sparsity distribution in the channel level. By pruning the entire filters or channels, the sparse model preserves the spatial regularity, thus, the pruned convolution layers can be transformed to a full matrix multiplication with reduced matrix size and accelerates computation on the hardware level. However, structured pruning suffers from significant accuracy loss as one entire activation maps can be zero. 3) *fine-grained structured pruning*, which includes block-based pruning and pattern-based pruning. In block-based pruning (Rumi et al., 2020), the unstructured weights are partitioned into several dense blocks. However, all above works aim to accelerate inference stage, our work differs substantially from them such that we do not rely on a pre-trained dense model.

6.2 SPARSE TRAINING

Sparse training aims to train a sparse network from early stage. Based on the sparse mask updating schemes, it is usually divided into two categories: 1) *Static sparse training*, where the sparse mask is obtained at early stage of training and is fixed during the course of training. Previous works obtain the sparse mask by random pruning or utilizing some saliency criteria to find important connections. The Single-Shot Network Pruning (SNIP) (Lee et al., 2019) uses the gradients of the training loss as connection sensitivity to prune the network at initialization. Later on, many criteria have been proposed, such as gradient flow in GraSP (Wang et al., 2020), synaptic strengths in SynFlow (Tanaka et al., 2020), Fisher information (Sung et al., 2021), etc. 2) *Dynamic sparse training*, which starts from random sparse mask and dynamically update connections during training. DST avoids the pruning-at-initialization process that usually involves the full dense model computation that makes the end-to-end training computation and memory-intensive. Specifically, SET (Mocanu et al., 2018) and MEST (Yuan et al., 2021) update sparse mask by pruning weights that has least magnitude, and grow back the same amount of inactivated weights in a random fashion. RigL (Evci et al., 2020) proposes to update sparse mask by magnitude-based pruning and grow back inactivated weights by their gradients. DSR (Mostafa & Wang, 2019) and STR (Kusupati et al., 2020) designs a dynamic reparameterization method that allows weights to be re-distributed across layers by providing a global sparsity allocation dynamics. DeepR (Bellec et al., 2018) combines dynamic sparse parameterization with stochastic parameter updates for training, but it primarily targets small and shallow fully-connected networks. However, these literatures focus on the unstructured sparsity.

7 CONCLUSION

In this paper, we firstly analyze the implementation of CNN's forward and backward pass with GEMM, and point out that current fine-grained structured pruning methods cannot promise the backward acceleration. Then we propose a new hardware-friendly regrouping algorithm called HRBP/HRBP++, aiming to accelerate the sparse training at both forward and backward pass on hardware with the same fine-grained structured sparsity pattern. Extensive results suggest that our sparse training with HRBP(++) sparsity pattern can achieve comparable accuracy with unstructured pruning methods. Meanwhile, it brings significant acceleration and even performs better than channel-wise pruning under certain circumstances. We believe our method can integrate the advantage of both unstructured pruning and structured pruning and make sparse training more hardware-friendly.

REFERENCES

Guillaume Bellec, David Kappel, Wolfgang Maass, and Robert Legenstein. Deep rewiring: Training very sparse deep net-works. In *International Conference on Learning Representations (ICLR)*, 2018.

Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pp. 233–244, 2009.

Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R Gao. Dynamic load balancing on single-and multi-gpu systems. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–12. IEEE, 2010.

Tianlong Chen, Xuxi Chen, Xiaolong Ma, Yanzhi Wang, and Zhangyang Wang. Coarsening the granularity: Towards structurally sparse lottery tickets. *ICML*, 2022.

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners. In *International Conference on Machine Learning*, pp. 2943–2952. PMLR, 2020.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2019.

Shaopeng Guo, Yujie Wang, Quanquan Li, and Junjie Yan. DMCP: Differentiable Markov channel pruning for neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1539–1547, 2020.

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

Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pp. 1135–1143, 2015b.

Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1389–1397, 2017.

Itay Hubara, Brian Chmiel, Moshe Island, Ron Banner, Joseph Naor, and Daniel Soudry. Accelerated sparse neural training: A provable and efficient method to find n: m transposable masks. *Advances in Neural Information Processing Systems*, 34:21099–21111, 2021.

Aditya Kusupati, Vivek Ramanujan, Raghav Somani, Mitchell Wortsman, Prateek Jain, Sham Kakade, and Ali Farhadi. Soft threshold weight reparameterization for learnable sparsity. In *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 5544–5555, 2020.

Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pp. 598–605, 1990.

Namhoon Lee, Thalaiyasingam Ajanthan, and Philip Torr. Snip: Single-shot network pruning based on connection sensitivity. In *International Conference on Learning Representations*, 2019.

Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

Mingbao Lin, Rongrong Ji, Yan Wang, Yichen Zhang, Baochang Zhang, Yonghong Tian, and Ling Shao. Hrank: Filter pruning using high-rank feature map. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1529–1538, 2020.

Shiwei Liu, Lu Yin, Decebal Constantin Mocanu, and Mykola Pechenizkiy. Do we actually need dense over-parameterization? in-time over-parameterization in sparse training. In *International Conference on Machine Learning*, pp. 6989–7000. PMLR, 2021.

Zechun Liu, Haoyuan Mu, Xiangyu Zhang, Zichao Guo, Xin Yang, Kwang-Ting Cheng, and Jian Sun. Metapruning: Meta learning for automatic neural network channel pruning. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 3296–3305, 2019.

Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE international conference on computer vision*, pp. 2736–2744, 2017.

Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*, 2017.

Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications*, 9(1):1–12, 2018.

Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *International Conference on Machine Learning*, pp. 2498–2507. PMLR, 2017.

Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 11264–11272, 2019.

Hesham Mostafa and Xin Wang. Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization. In *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 4646–4655, 2019.

Masuma Akter Rumi, Xiaolong Ma, Yanzhi Wang, and Peng Jiang. Accelerating sparse cnn inference on gpus with performance-aware weight pruning. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pp. 267–278, 2020.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Wei Sun, Aojun Zhou, Sander Stuijk, Rob Wijnhoven, Andrew O Nelson, Henk Corporaal, et al. Dominosearch: Find layer-wise fine-grained n: M sparse schemes from dense neural networks. *Advances in Neural Information Processing Systems*, 34:20721–20732, 2021.

Yi-Lin Sung, Varun Nair, and Colin Raffel. Training neural networks with fixed sparse masks. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=Uwh-v1HSw-x.

Hidenori Tanaka, Daniel Kunin, Daniel L Yamins, and Surya Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow. *Advances in Neural Information Processing Systems*, 33:6377–6389, 2020.

Yehui Tang, Yunhe Wang, Yixing Xu, Dacheng Tao, Chunjing Xu, Chao Xu, and Chang Xu. Scop: Scientific control for reliable neural network pruning. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking winning tickets before training by preserving gradient flow. *arXiv preprint arXiv:2002.07376*, 2020.

Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems*, pp. 2074–2082, 2016.

Dongkuan Xu, Ian EH Yen, Jinxi Zhao, and Zhibin Xiao. Rethinking network pruning–under the pre-train and fine-tune paradigm. *NAACL*, 2021.

Geng Yuan, Xiaolong Ma, Wei Niu, Zhengang Li, Zhenglun Kong, Ning Liu, Yifan Gong, Zheng Zhan, Chaoyang He, Qing Jin, et al. Mest: Accurate and fast memory-economic sparse training framework on the edge. *Advances in Neural Information Processing Systems*, 34, 2021.

Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. Learning n:m fine-grained structured sparse neural networks from scratch. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=K9bw7vqp_s.

Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 359–371, 2019.

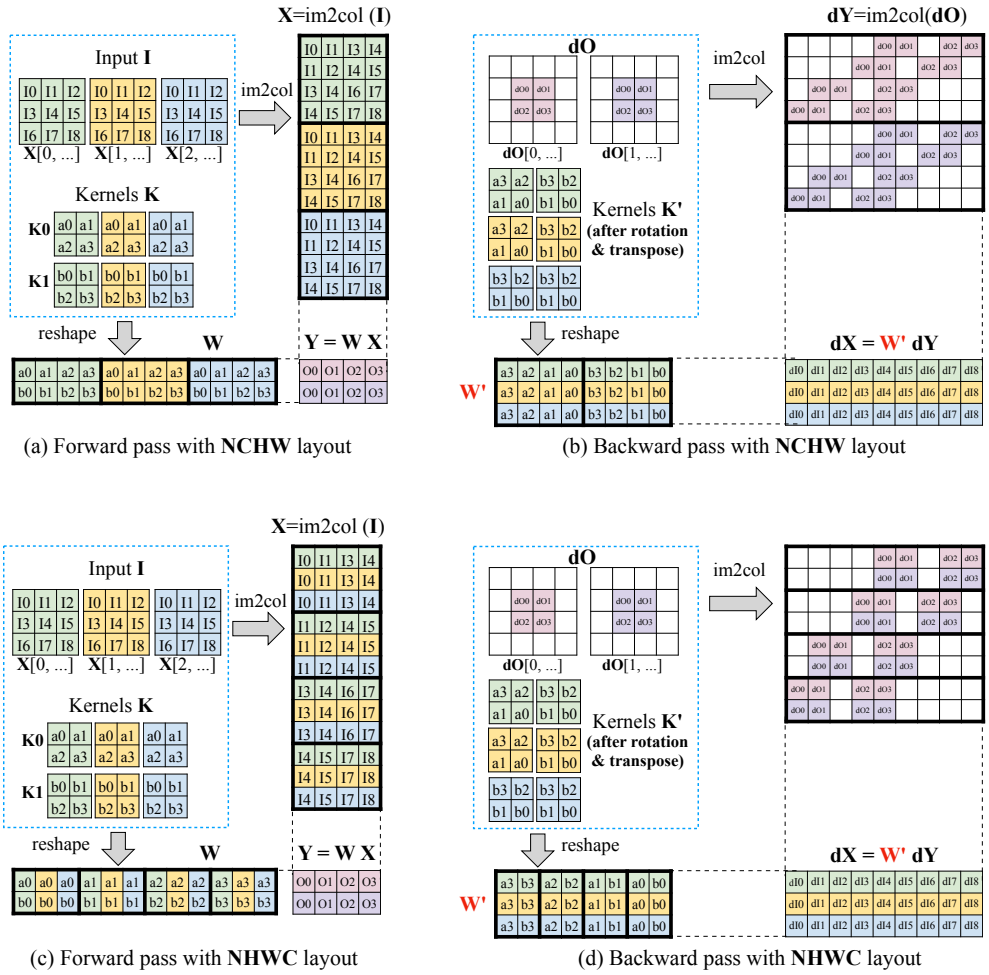
Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.

A FORWARD AND BACKWARD PASS IN NCHW AND NHWC LAYOUT

NCHW and NHWC data layout formats are two common types of cuDNN tensors arrangement in memory (Chetlur et al., 2014). These two layouts produce the same shape matrix and the same outputs. The only difference is the order of element in the flattened tensor, as shown below:

NCHW layout As shown in Fig. 7(a), in forward pass, the flattened tensor $\mathbf{W}$ begins with the first input channel (green color), the elements are arranged contiguously in row-major order (*i.e.*, $a0, a1, a2, a3$ with green color for the kernel $\mathbf{K}$ in forward). Then, it continues with second (orange color) and subsequent channels until the elements of all the channels are laid out.

NHWC layout As shown in Fig. 7(c), in forward pass, the flattened tensor $\mathbf{W}$ begins with the first element of the first input channel (*i.e.*, $a0$ with green color for $\mathbf{K}$), then proceed to the first element of the second input channel (*i.e.*, $a0$ with orange color for $\mathbf{K}$), and so on, until the first elements of all the C channels are laid out. Next, select the second element of the first input channel (*i.e.*, $a1$ with green color for $\mathbf{K}$), then proceed to the second element of the second input channel (*i.e.*, $a1$ with orange color for $\mathbf{K}$), and so on, until the second element of all the channels are laid out.



(a) Forward pass with **NCHW** layout

(b) Backward pass with **NCHW** layout

(c) Forward pass with **NHWC** layout

(d) Backward pass with **NHWC** layout

Figure 7: Illustration of GeMM with NCHW and NHWC format. Different colors represent different channels.

B N:M TRANSPOSABLE MASK IN THE BACKWARD OF CNN

As suggested in (Zhou et al., 2021), the N:M sparsity The N:M transposable mask (Hubara et al., 2021) guarantees that both the weight matrix $\mathbf{W}$ and its transpose $\mathbf{W}^T$ follow the same sparsity

pattern. This design works well for linear layer, whose forward pass is calculated with $Y = WX$ and backward pass is calculated with $dX = W^T dY$. However, the implementation of CNN with GEMM is different. Specifically, we have $\mathbf{W} \in \mathbb{R}^{C_O \times (C_I K_h K_w)}$ for the weights matrix of CNN. In the backward pass, CNN flips each kernel first vertically and then horizontally and performs the kernel-wise transpose. Thus, as in Fig. 1(c), $\mathbf{W}^T \in \mathbb{R}^{(C_I K_h K_w) \times C_O}$ is different from the matrix $\mathbf{W}' \in \mathbb{R}^{C_I \times (C_O K_h K_w)}$ in the backward pass of CNN in most cases. Nevertheless, this N:M transposable mask can still accelerate the CNN backward under some conditions.

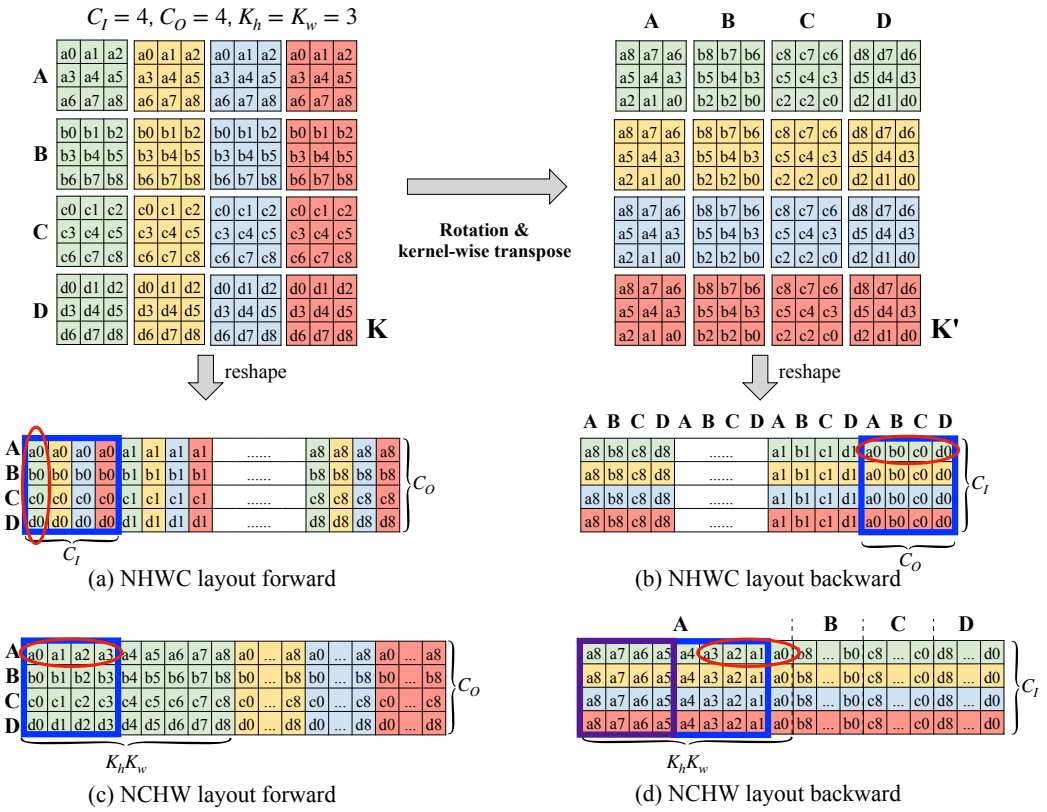
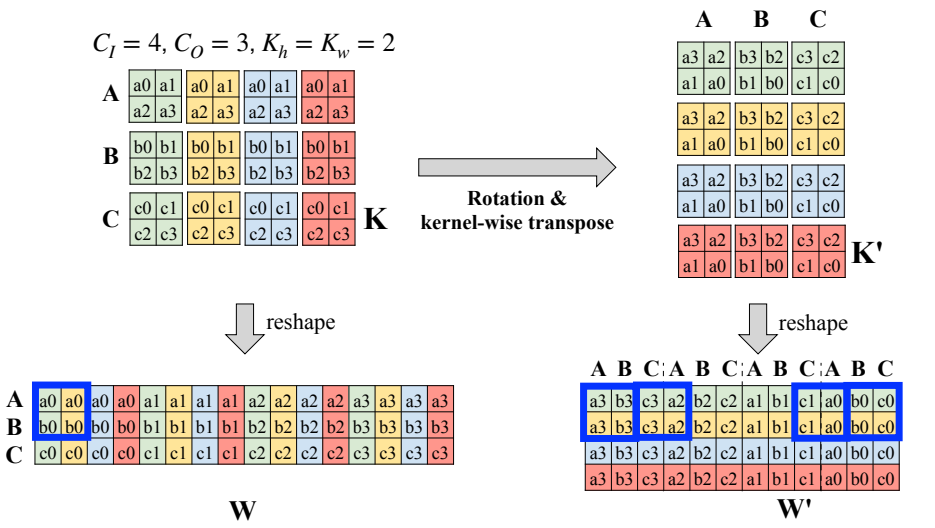


Figure 8: Comparison of NHWC and NCHW layouts of N:M mask in both forward and backward pass.

Layout. The first requirement is the layout of the weights in the memory. For example, suppose $N = 2$, $M = 4$, the size of the input channel is $C_I = 4$ and the size of the output channel is $C_O = 4$. Thus, we can divide the kernel matrix $\mathbf{W} \in \mathbb{R}^{4 \times (4 \times 3 \times 3)}$ into several 4×4 blocks. As shown in Fig. 8(a)(b), when we store kernels with NHWC format, we collect $a0$ with green color, $a0$ with orange color, $a0$ with blue color, and $a0$ with red color into a row of a block (*i.e.*, the blue box) for the forward pass, and this row meets the N:M constraint. Based on the N:M transposable mask, the column of $a0$, $b0$, $c0$, and $d0$ with green color also satisfies the N:M constraint. In the backward pass, this blue block can still be kept. Meanwhile, $a0$, $b0$, $c0$, and $d0$ with green color is collected as a row. Thus, the same sparse pattern can be maintained in the backward with N:M transposable design. When the weights is stored with NCHW format, as shown in Fig. 8(c)(d), the green cell $a0$, $a1$, $a2$, and $a3$ form a row of the blue block. Again, the column of $a0$, $b0$, $c0$, and $d0$ with green color also satisfies the N:M constraint. However, in the backward pass, the same block cannot be kept. Thus, the sparse pattern may not meet the N:M constraint. For example, in forward pass, green cells $a2$ and $a3$ are kept, and the green cells $a4$ and $a7$ are kept. In the backward, the green cell $a1$, $a2$, $a3$, and $a4$ form a new vector with the size $M = 4$, but this vector has three cells $a2$, $a3$, and $a4$ with mask "1". To this end, additional operations such as re-indexing and regrouping are required to make the weight matrix in the backward meet the N:M constraint. In summary, the N:M transposable mask can accelerate the CNN backward pass with NHWC layout GEMM implementation, but it cannot be directly applied to the NCHW layout.



(a) NHWC layout forward

(b) NHWC layout backward

Figure 9: Example of N:M transposable mask when $C_O \bmod M \neq 0$.

Shape. Moreover, even if the weights are arranged in `NHWC` layout, we show some cases that the N:M transposable mask may not bring the acceleration on backward as well. Considering the example with $C_O = 3$ and $M = 2$, as shown in Fig. 9, we collect $a0$ with green color, $a0$ with orange color into a row of a 2×2 block (*i.e.*, the blue box) for the forward pass, and this row meets the N:M constraint. Based on the N:M transposable mask, the column of $a0$, $b0$ with green color also satisfies the N:M constraint. However, in the backward, the $a0$ is grouped with $c1$, and the $b0$ is grouped with $c0$. Thus, the same sparsity pattern cannot be guaranteed. To this end, the output channel C_O should be divisible by M, *i.e.*, $C_O \bmod M = 0$.

C PYTORCH-STYLE PSEUDOCODE FOR HRBP AND HRBP++

In this section, we give the detailed implementation of our HRBP and HRBP++ in a Pytorch-style pseudocode.

Algorithm 1: PyTorch-style Pseudocode for HRBP(HRBP++)

```
# m:  unstructured mask with size (Co,Ci,Kh,Kw)
# r:  density ratio of m; t:  number of clusters
# s:  dense ratio of kernel (s=1 in HRBP)
# Output:  groups, patterns(HRBP++ only)
# Generate kernel-wise mask M
ms=m.sum(dim=(2,3)) # (Co,Ci)
M=torch.zeros(ms.shape) # (Co,Ci)
M.view(-1)[ms.view(-1).topk(r/s*Ci*Co)[1]]=1
# Regrouping and refill based on M
clusters=EqualPartition(M,t1)
for row_id in clusters:  # length is Co/t
   if len(row_id)>=B1:
      col_id=M[row_id].sum(0).topk(r/s*Ci)[1]
      groups.append((row_id,col_id))
M'=Refill(M, groups) # (Co,Ci)
if HRBP++:  # Find common pattern for HRBP++
   for (row_id,col_id) in groups:
      groupK=Index(m,row_id,col_id) # (..,Kh,Kw)
      Kcount=groupK.sum(0).view(-1) # (Kh*Kw)
      patterns.append(Kcount.topk(s*Kh*Kw))
```