Accelerating Sparse Convolution for Efficient Neural Network Inference

Anonymous Author(s)
Affiliation
Address
email

Abstract

Weight sparsity is a promising approach to reduce model size and computation cost of convolutional neural networks (CNNs). Nevertheless, non-zero weights often distribute randomly in sparse CNN models, introducing enormous difficulty to obtain actual speedup on common hardware (e.g., GPU) over their dense counterparts. Existing acceleration solutions either require hardware modifications for irregular memory access support, or reply on partially structured sparsity pattern. Neither of these methods are capable of achieving fruitful speedup on convolution layers.

In this work, we propose an algorithm-software co-designed sparse convolution, based on a novel out-vector-wise (OVW) sparse pattern. Building on the insight that vertical vector integrity can preserve continuous memory access in IM2COL, OVW pattern treats a $V \times 1$ vector as an entirety. To reduce the error caused by sparsity, we propose an equivalent transformation process, i.e., clustering-based channel permutation, to gather similar rows together. Experimental evaluations demonstrate that our method achieve an $1.7\times$ and $3.2\times$ speedup over the SOTA solution and the dense convolution of ResNet50 on NVIDIA V100 at 75% sparsity, respectively, with only negligible accuracy loss. More interestingly, comparing to the SOTA solution that achieves speedups only on data with 60% sparsity or more, our method starts to obtain speedups on data with only 10% sparsity.

1 Introduction

Convolutional neural networks (CNNs) have yielded astonishing results in many important domains such as vision [8], language [19] and gaming [16]. With CNN algorithm develops rapidly, the storage and computing overhead of CNN models grow exponentially. To significantly reduce both the computations and data accesses, weight sparsity has been taken as a promising approach to improve the hardware efficiency.

Although succeeded in reducing computations and data accesses, the unconstrained, fine-grained sparsity fails to bring practical speedups on common GPUs. Normally, sparsity is introduced in CNNs randomly, as synapses and neurons with negligible value will be skipped from computation and data accesses, for they have little affect to the following neurons. Such randomness causes a high overhead in processing sparsity. On NVIDIA V100, only with over 95% sparsity ratio[18, 3], the sparse matrix multiplication can achieve the same performance as the dense matrix multiplication.

Unfortunately, existing solutions either require hardware modifications, or only partially address the problem by constrained with structured, coarse-grained sparsity, resulting high accuracy loss. The formal is to leverage the sparse matrix-matrix multiplication (e.g., SPMM) operation on GPU. While

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

directly applying SPMM on sparse CNNs can run even slower than dense CNNs [**?**], productive SPMM acceleration solutions[22, 14] often require their unique need for dedicated hardware support to overcome discontinuous memory access, which is impractical. A software-only solution is more desirable. The latter is to leverage the general matrix-matrix multiplication (e.g., GEMM) operation on GPU. Many efforts have been made to explore regular, structured sparsity to reduce the overhead of processing sparsity [4, 9, 5]. Block sparsity[4] manages to restore the spatial locality of matrices to a large extent, at the cost of a strict restriction on the non-zero structure of sparse matrices, which suffers from severe inference accuracy loss. Balanced sparsity[2, 20, 15], newly introduced on NVIDIA A100 GPU[14], however, lacks of flexibility in choosing model sparsity rate that only exact 50% sparsity ratio could be deployed on this dedicated hardware. These efforts achieve some palpable acceleration comparing to dense GEMM operation, but they all struggle to attain similar result on convolution layers which has proven to be a greater challenge.

We find that the key reason for the inefficiency is the introducing of IM2COL operation, which is required to convert GEMM to a convolution operation. The IM2COL operation transforms one input matrix into a larger one for multiplication with purely IO operations. As the matrix multiplication process growing faster and faster, IM2COL operation begins to occupy an increasing proportion of the whole operation. However, modern DNN accelerators hides IM2COL's IO latency with implicit GEMM convolution algorithm. Thus GEMM-based sparse methods will be increasingly powerless in convolution computation.

In this paper, we present a novel implicit GEMM-based sparse convolution algorithm. Based on above key finding, we propose a novel out-vector-wise (OVW) sparse pattern for this algorithm. OVW sparse pattern is a carefully designed vector-wise sparsity pattern which can hold memory consistency for both input matrix while sharing a same amount of data reuse rate as the dense convolution operation by treating a $V \times 1$ vector as an entirety. We also propose an clustering technique assisting with channel permutation to recover network accuracy of the OVW pattern. These efforts assist our algorithm to establish a superiority acceleration results over other sparse kernels while maintaining the high performance of CNNs. We practice our methodology on various CNN models, showing an dominant acceleration rate over existing sparse convolution algorithm. What's more, our GPU kernel starts to demonstrate practical kernel speedup at an unprecedented low sparsity ratio comparing to its dense counterparts. The acceleration possible model sparsity threshold of our implementation is lowered to less than 10%, while prior arts can only reduce it to around 60%. This advancement in threshold decline renders us an abundant flexibility in choosing proportion of weights to be pruned in each layer.

The main contributions of this paper are listed as follows:

- We propose a vector-based sparsity pattern, i.e., the OVW pattern to balances inference accuracy loss and computation efficiency using hardware-friendly means.
- We implement a new GPU convolution kernel based on the OVW pattern. The technique of extracting filter location information further reduces inference runtime.
- We propose a heuristic clustering methods to obtain an appropriate channel permutation for reducing accuracy drop during weight pruning. This channel permutation operation is detached from the inference procedure.
- Our GPU kernel can accelerate convolution at a wide range of model sparsity rate. With few accuracy loss, it can speed up ResNet50 by $1.7\times$ and $3.2\times$ over the SOTA solution and the dense cuDNN convolution on NVIDIA V100 GPU at 75% sparsity level.

2 Related work

2.1 Software-only Acceleration For Sparse CNN Model

Weight pruning has been a popular technique for efficient CNN inference. Early studies[7, 6] show that removing a large proportion amount of unimportant connections in CNN models does

not necessarily lead to inference accuracy impairment. The reduction in parameters helps exploit redundancy in CNN models, which requires less computations and data accesses. However, for CNN inference, weight pruned sparse DNNs usually perform worse than dense CNNs, unless the CNN sparsity ratios are very large, i.e., very sparse CNNs. Experiments[18, 3] report a threshold of sparsity ratio at 95% for unstructured sparse CNN models to run faster than dense model on NVIDIA GPU.

In order to address this issue, methods other than unstructured sparsity are exploited. Researchers exploit various constraints on sparsity pattern in exchange for computation efficiency. A major domain of this region is filter pruning, where parameters of an entire filter are pruned or kept as a whole. However, this direct modification of channel size suffers sharp accuracy drop [10, 11, 17]. Moderate sparsity patterns are also examined, such as block sparsity[18], which is proposed to elevate spatial locality of sparse matrix. But this feature can achieve speedups only when sparsity ratios are larger than 70%. Tile-wise sparsity[5] renders weight patterns even more flexibility. Compared to previous methods, balanced sparsity[15, 20] is more feasible with a recent support from NVIDIA A100 GPU who directly optimizes 2:4 balanced sparsity. Recent work $Shfl_BW$[9] uses matrix transformation to utilize block sparsity's computation efficiency while removing some of its constrains, further reducing the acceleration possible model sparsity threshold of sparsity ratio from 70% to 60%. Different from prior works which can only work for very sparse matrices, in this paper, we propose a software-only sparse convolution for CNNs that can obtain speedup when the sparsity ratio is only 10%.

2.2 GEMM Based Convolution

GEMM has been adopted widely to perform convolution. Since most of the operations in the CNN layers can be directly mapped to GEMM, all modern commercial-grade hardware accelerators such as TPUs and GPUs provide a highly optimized general matrix-matrix multiplication (GEMM) operator. Thus GEMM-based convolution performs significantly better than other convolution methods such as FFT, Winograd on these hardware.

This GEMM-based algorithm could be further divided into explicit matrix multiplication and implicit matrix multiplication. Explicit matrix multiplication prepares transformed inputs for the GEMM engine using the IM2COL operation. IM2COL is an IO intensive operation, which brings in significant workload other than computation cost[1]. Implicit matrix multiplication merges these operations for more efficient memory accesses. It maintains and updates pointers of feature map in shared memory carefully while performing tile-based matrix multiplication simultaneously. On NVIDIA V100 GPUs, explicit GEMM convolution is on average 120%, 126% and 142% slower than implicit GEMM convolution on convolution layers of Alexnet, Resnet and Googlenet [21] .

Performing sparse convolution by GEMM is always through explicit matrix multiplication rather than implicit matrix multiplication. The reason is because of the IM2COL operation, which is split into pieces in implicit matrix multiplication. This IM2COL division is extremely difficult if not impossible for sparse matrix multiplication, as sparse matrices are compactly compressed and irregularly stored. Therefore, GEMM-based sparse convolution is only implemented with explicit GEMM, which has to suffer from the costly entire IM2COL operation.

Yet, few studies have investigate the sparse convolution with implicit GEMM. As explicit GEMM performs worse than implicit GEMM, sparse convolution with implicit GEMM has the potential to achieve higher performance and efficiency. In this paper, we investigate the implicit GEMM based sparse convolution to leverage the high performance GEMM on exist hardware.

3 Accelerating Sparse Convolution

In this section, we introduce our proposed sparse convolution, including the outer-vector-wise (OVM) pattern of sparsity for the proposed sparse convolution, and our detailed implementation on GPU.

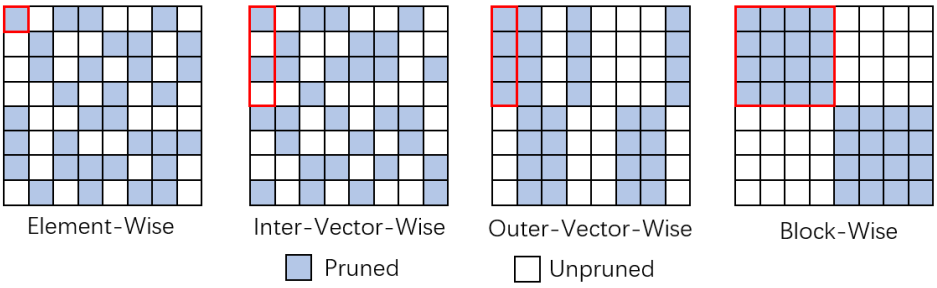

Figure 1: Comparison of four patterns with 50% sparsity.

3.1 OVW Pattern

Sparsity in matrix can be classified into three different patterns, as shown in Fig 1. The first sparsity pattern is element-wise(EW) pattern, corresponding to unstructured pruning, which evaluates each parameter individually. Having imposed no constraint on pruning, this pattern succeed at model flexibility but struggled at actual deployment acceleration due to its irregular memory accesses. The second sparsity pattern is vector-wise pattern. Here we summarizes two types of vector-wise sparsity pattern, the inter-vector-wise(IVW) and the outer-vector-wise(OVW). They both treat a V$\times$1 vector as an entirety while the IVW pattern prunes certain proportion of weights inside each vector and the OVW pattern focuses on the entire vector of weights. The third pattern is block-wise(BW) pattern and it's minimum pruning granularity is a V$\times$V block. This pattern has the highest computation efficiency but it's inference accuracy loss is unsatisfactory.

In this work, we use the OVW pattern as it shares the advantages of VW pattern, which balanced computation efficiency of BW and network accuracy of EW. The $Shfl_BW$ pattern is a variant of this pattern, who uses an extra channel reordering procedure to gather block-wise pattern utilities. The OVW pattern, however, could be directly used in our convolution algorithm which denotes a higher acceleration potential. Comparing to another vector-level sparse pattern the IVW pattern, also known as the N:M structured sparsity, our approach holds no requirements for specialized hardware supports and it is much more elastic in selecting sparsity ratio of each layer. Also the IVW pattern still faces the memory-bound issue, because the amount of redundant data that needs to be loaded into shared memory each time is equal to its sparsity ratio.

3.2 Overview of the Proposed Sparse Convolution

Utilization of efficient dense warp-level GEMM instruction is the key contributor of most of the sparse matrix acceleration methods. As shown in Fig2, the OVW pattern sparse convolution can be broken into multiple dense matrix multiplication of smaller size. During a normal convolution procedure, inputs data is loaded to shared memory as block shape for block GEMM computation. In the loading process, a column of filter data loaded into shared memory shares a specific position on the filter map where a continuous block on feature map is loaded to be computed with. Several column fetched from filter together form the left input matrix of the block matrix multiplication and their corresponding feature data blocks form the right input maatrix. Noticing that this forming process of input matrix does not require the loaded filter columns to be continuous, meaning that efficient dense operations can also be performed by grouping some unrelated columns. Building on this foundation, we could take in multiple columns of irrelevant column indices from the OVW pattern sparse matrix and handle it in the same way as the dense GEMM operation. This resemblance between our convolution algorithm and the implicit GEMM convolution guarantees us similar overall computation efficiency.

This procedure only additionally introduces the loading instruction workloads of column indices data and the calculation of the corresponding filter location compared to the original dense convolution.

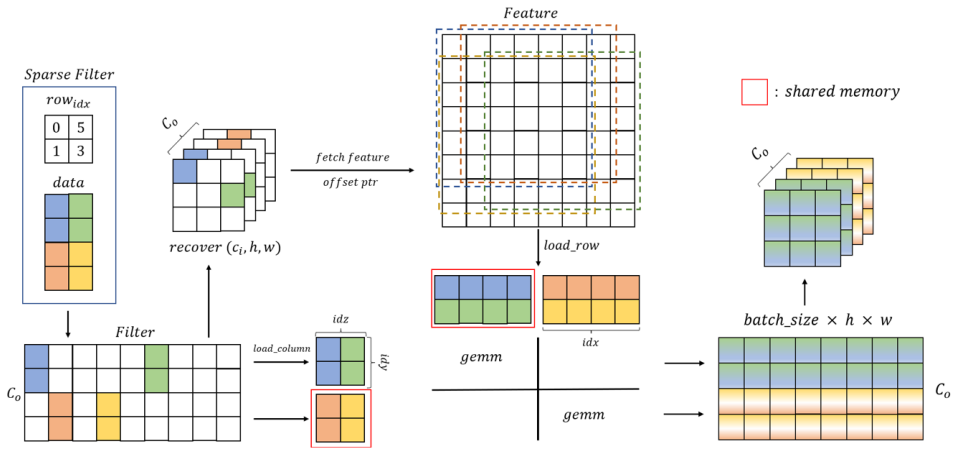

Figure 2: Illustration of GPU conv kernel.

Both of these workloads are negligible in comparison to the high computation cost reduced by sparsity. Vector length used in this format depends on the tile size that hardware shared memory could hold.

3.3 GPU Sparse Kernel

As shown in Algorithm 1, the convolution kernel contains two steps. The first step is to load data from both input matrix to shared memory. Some threads use function load_column to load column vector of length TM from filter to shared memory with DY threads. Threads then use function load_row to load row vector from feature map in the same way. The second step calls warp matrix multiplication operands. When calculating matrix $A_{M\times K} \times B_{K\times N} = C_{M\times N}$, a three dimension parallelism(DX, DY, DZ) thread is employed[]. (DX, DY, DZ) threads each loops along (M, N, K) individually. Several threads together use function Warp_MMA to multiply the loaded matrix and writes back after results accumulation. Each thread computes a tile matrix multiplication of the size (TM, TN, TK).

The procedure of calculating the exact pointer offset of feature map for corresponding filter columns contains two steps. Because during a convolution computation, corresponding location of $a_{ij} \times b_{jk}$ is not obvious. After loading a_{ij} into shared memory, firstly, GPU kernel has to fetch the column indices of a_{ij} in the original filter. The column indices is then used to recover the exact position of this column in the filter map. Secondly, the location offset of the corresponding data in feature map is calculated and after which b_{jk} can be finally located in the feature map.

3.4 Preprocessing Filter Location Information

If we only stores the column indices of sparse matrix, its location information has to be recovered each time before the corresponding activation is loaded. Resembling to the location offset in feature map, the corresponding data location in filter map is a constant for each thread during the whole process. If we prepare this information in advance and store its actual location instead of its column indices, the location calculation process could be saved, thus introducing could be further accelerated. Extra storage occupation of this technique is two extra dimension indices data array of filter map, which takes merely 3% total storage of a compressed model with vector length=64, and 6% with vector length=32, in exchange of 10% run time reduction of Resnet50's convolution layers in average. Considering the sparse model is already highly compressed, this additional model redundancy is totally acceptable.

Algorithm 1: Sparse convolution computation

Data: $row_idx[]$, $filter[]$, $input[]$
Result: $output[]$

```
Shared memory A[TM][TK], B[TK][TN], C[TM][TN];
for Thread idx=1 to DX, idy=1 to DY, idz=1 to DZ do
    offset = Cal_Thread_Offset(row_idx[], idx, idy, idz);
    if idx < TK then
        Load_column(A, filter[idz][idx], TM, idy);
    end
    if idy < TK then
        Load_row(B, input[offset], TN, idx);
    end
    Syncthreads();
    Warp_MMA(A, B, idx, idy);
    Accumulate_Results(C);
    Store(output, C);
end
Return output;
```

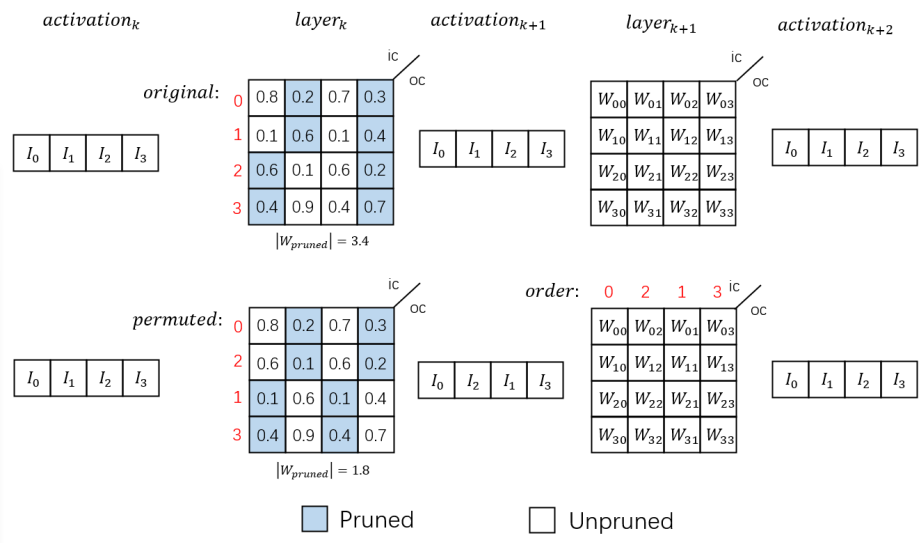

Figure 3: Illustration of permutation transfer process. The permuted matrix kept larger amount of absolute weight value after pruning at the same sparsity. Following activation is recovered by permuting output channel of layer k and input channel of layer k+1 simultaneously.

4 Pruning Algorithm

4.1 Channel Permutation

Our pruning methods can be divided into two steps: shuffling the filter matrix rows in each layer to a desirable one, applying an vector-wise pruning to the intermediate results.

Here we will explain why filter permutation will do no harm to the network inference. Matrix multiplication only swaps the order in the output dimension and does not change the actual computation. Permuted operation results can be recovered through a reversed permutation of the operand output. As we only permutes the output channel of each layer, new order of the current layer will absorbed by the input channel dimension of the next GEMM-based layer(convolution or linear). Fig. 3 shows one iteration for channel permutation between layer$_k$ and layer$_{k+1}$. As we were reordering the

output channel of layer$_k$, the activation$_k$ is changed to the same order, but after we permutes the input channel of layer$_{k+1}$, activation $k+1$ is restored. More weights value could be saved after permutation. The same operation is then repeated on layer $k+1$ and so on, until every GEMM layer in the network is permuted. This permutation transfer procedure allow us to choose a appropriate permutation row order for every filter without altering the output of the network.

Rest of the layers such as the pooling layer and the activation layer involve no modification along channel dimension thus are not affected by this process. BN layers and the bias adding at the end of convolution layers and linear layers do not produce any new permutation order, but they have to permute according to the permutation passing through.

4.2 Row Clustering

Algorithm 2: Row clustering

Data: The original weight W, number of clusters k, number of selected column m.
Result: The reordered weight RW.

```
RW = empty;
while W is not empty do
    Sort the columns of W by column variance;
    Build SampleW by selecting columns with the top-m largest variances;
    Get the k clustered groups G = Balanced_kmeans(SampleW, k);
    Select the group g with maximum sum;
    Append g to RW;
    Remove g from W;
end
Return RW;
```

The only remaining question is how to decide an appropriate permutation order. A heuristic indicator evaluating the quality of a permutation is the sum of absolute weight value pruned, assuming that the method that the preservation of more important weights corresponds to less inference accuracy loss. An obvious route is that the weight rows with shorter distance assigned to the same group as much as possible. $Shfl_BW$ chooses the kmeans method for clustering, but after carefully experiments this idea, we found that the kmeans method does not suit this issue well. For starters, the number of elements in each group is set to be a fixed value (vector length), and kmeans requires additional operations to meet this demand. Also the data dimension(input channel multiplies filter height and width) is too large, while the amount of data and groups are relatively small. Kmeans falls in local minima easily and the output cluster is extremely unstable. We modified the kmeans to better adapt the OVW pattern, which is able to palliate both symptom mentioned above.

Algo2 shows the key steps of our algorithm. First, we construct a characteristic matrix by assembling rows with highest variance, then cluster rows of this matrix to alleviate excessive dimension. We utilize a balanced clustering method[13] to solves the assignment of given cluster sizes. During the clustering process, this problem is equivalent to a linear sum assignment problem. We employ Kuhn-Munkres algorithm to solve the distance matrix and optimize the mean square error of it. Secondly, for each clustering result, we only adopt the most important group under this feature to increases operation stability. This group is removed from the original matrix and then the feature matrix is reconstructed and clustered. Repeat the above steps until all data grouping completes, the permuted matrix and its permuting order is obtained.

5 Evaluation

5.1 Model Accuracy

We evaluate our method on several popular CNN models on NVIDIA V100 GPU. We only calculate the speedup of the convolution layers in the following results. Table1 shows the accuracy of our

Table 1: Network accuracy on Cifar100.

Network	Baseline(Dense)	Unstructured			**OVW permuted**		OVW	
		80%	90%	V	80%	90%	80%	90%
Vgg19	71.41	71.51	71.30	64	71.41	71.44	71.39	71.26
Resnet18	77.19	76.47	74.95	64	74.45	72.40	74.02	71.24
Resnet50	78.60	78.07	78.02	64	76.17	73.48	75.96	72.85
SqueezeNet	71.01	70.20	70.42	44.48	69.40	65.49	69.01	65.46
MobileNet v2	68.99	68.95	68.59	32.02	68.43	68.36	68.13	66.01

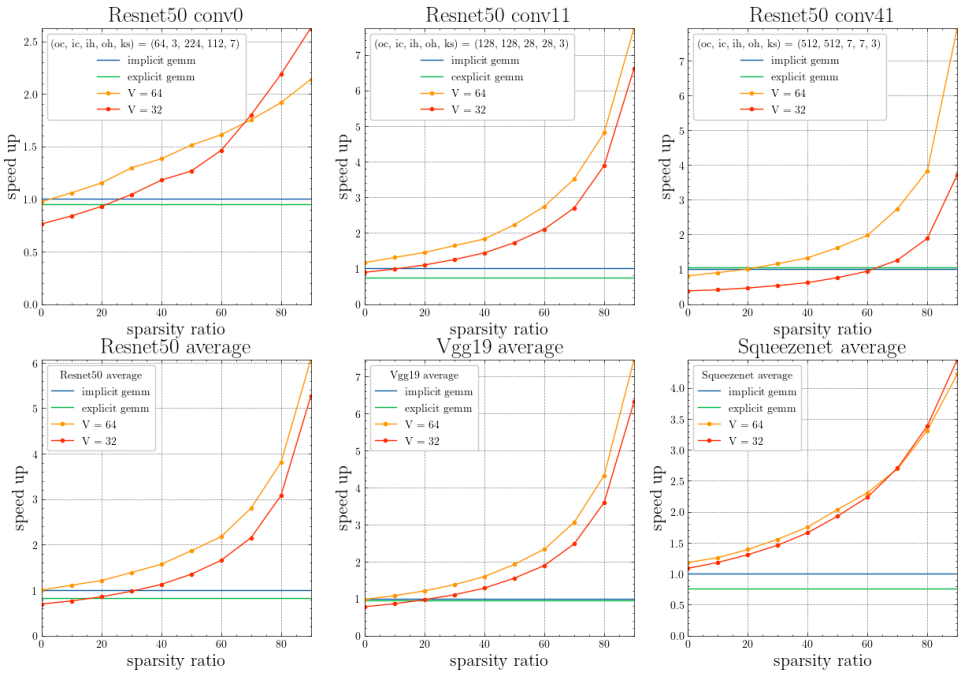

Figure 4: Speed comparison between our sparsity convolution kernel and two GEMM-based convolution in cuDNN library on NVIDIA V100 GPU.

method comparing to unstructured sparsity. OVW permuted shows a better accuracy over OVW non-permuted on all this network.

All the results in this table use the same fine-tuning process. We fine-tune each network for 40 epochs after pruning with the same learning rate of 0.0008. Also, each layer can hold different sparsity ratio credit to our acceleration for convolution at low sparsity ratio. Vector length for OVW pattern here is set to the tile shape of highest speedup ratio. For example, the first convolution layer of SqueezeNet has only 96 output channel. If vector length is 64, the second tile The layer will be only half loaded. In this case vector length is set to 32 to maximize computation resources utilization. In depthwise convolution of MobileNet, because filter column weight shares no common feature data to compute with, vector length larger than 1 has no speed advantages.

5.2 Convolution Kernel Speedup

As shown in Fig4, we evaluate the speedup of our method on three popular CNN models. We use cuDNN convolution operator as the dense baseline. The first three graph in Fig4 represent three typical convolution shape in CNN models: small channel size with large feature map, medium channel size with medium feature map and large channel size with small feature map. Our kernel can accelerate all these types of convolution layer while exceeds at the twelfth convolution layer of

Table 2: Comparing sparsity patterns on Resnet50 for ImageNet classification.

Network	Sparsity ratio	Accuracy	Speedup
Baseline	Dense	76.12	1×
Balanced sparsity (2:4)[15]	50%	76.29	1.3×
$Shfl_BW$[9]	90%	73.09	2.5×
OVW	30%	76.17	1.38×
OVW	50%	75.76	1.86×
OVW	70%	73.35	2.79×

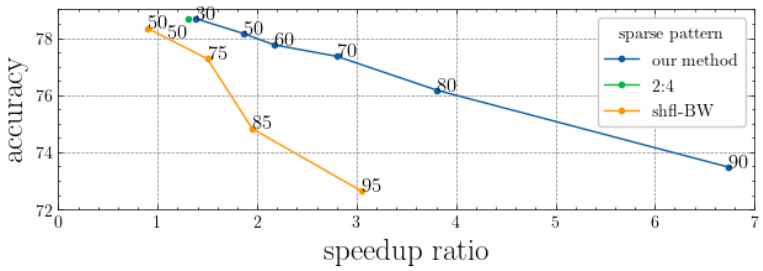

Figure 5: Speed up against the dense baseline on Resnet50 Cifar100 comparing to other vector-level sparse pattern on NVIDIA V100 GPU.

Resnet50 which is the most used kind of convolution layer $4.8\times$ and $3.87\times$ over cuDNN on V100 at 80% sparsity, vector length 64 and 32.

5.3 Comparing Sparsity Patterns

We identify two vector-level sparsity pattern, balanced sparsity(NVIDIA 2:4)[15] and $Shfl_BW$ [9] for comparison. Other vector-level sparsity pattern such as Tile-wise[5] are slower than the former two patterns. Also these pattern lack implementation for convolution are.

Table2 contains results of Resnet50 on ImageNet directly copied from $Shfl_BW$ paper, where an expensive method Grow and Prune[12] is used to recover its accuracy. Grow and Prune is a sparsity pattern independent method. Training from scratch for 990 epochs, this method can achieve high accuracy, but it is sparsity method independent, meaning that any sparsity pattern can utilize it to get better accuracy. We fine-tune our pretrained Resnet50 model for 20 epochs with the learning rate 0.001. We lower the sparsity of our network to 70% sparsity where our method demonstrate 73.35% top1 accuracy and $2.79\times$. Exhibiting a better speed-accuracy trade of comparing to $Shfl_BW$ pattern. The OVW pattern also achieves better speedup rate comparing to balanced sparsity while recovering full accuracy of the origin model.

To ensure fair comparison, we reproduce these results under the same setting on Resnet50 Cifar100, as shown in Fig5. We fine-tune each network for 40 epochs after pruning from pretrained dense models with the same learning rate of 0.0008. The OVW pattern dominates the speed-accuracy trade-off in vector-level sparsity.

6 Conclusion

Accelerating sparse convolution poses a greater challenge than accelerating sparse matrix multiplication. In this work, we recognize a novel sparsity pattern, the OVW pattern which leverages model accuracy and inference acceleration. Our GPU implementation outperforms all sparse acceleration approaches exceedingly with a sparse pattern of similar flexibility. We hope this paper can fill the vacancy of specialized sparse convolution kernel domain and our methodology can stimulate further exploitation on this sparsity pattern.

References

[1] MegEngine:A fast, scalable and easy-to-use deep learning framework. 2020.

[2] *Cao Shijie, Zhang Chen, Yao Zhuliang, Xiao Wencong, Nie Lanshun, Zhan Dechen, Liu Yunxin, Wu Ming, Zhang Lintao*. Efficient and effective sparse LSTM on FPGA with bank-balanced sparsity // Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2019. 63–72.

[3] *Gale Trevor, Zaharia Matei, Young Cliff, Elsen Erich*. Sparse gpu kernels for deep learning // SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. 2020. 1–14.

[4] *Gray Scott, Radford Alec, Kingma Diederik P*. Gpu kernels for block-sparse weights // arXiv preprint arXiv:1711.09224. 2017. 3. 2.

[5] *Guo Cong, Hsueh Bo Yang, Leng Jingwen, Qiu Yuxian, Guan Yue, Wang Zehuan, Jia Xiaoying, Li Xipeng, Guo Minyi, Zhu Yuhao*. Accelerating sparse dnn models without hardware-support via tile-wise sparsity // SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. 2020. 1–15.

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

[7] *Han Song, Pool Jeff, Tran John, Dally William*. Learning both weights and connections for efficient neural network // Advances in neural information processing systems. 2015. 28.

[8] *He Kaiming, Zhang Xiangyu, Ren Shaoqing, Sun Jian*. Deep residual learning for image recognition // Proceedings of the IEEE conference on computer vision and pattern recognition. 2016. 770–778.

[9] *Huang Guyue, Li Haoran, Qin Minghai, Sun Fei, Din Yufei, Xie Yuan*. Shfl-BW: Accelerating Deep Neural Network Inference with Tensor-Core Aware Weight Pruning // arXiv preprint arXiv:2203.05016. 2022.

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

[11] *Liu Zhuang, Sun Mingjie, Zhou Tinghui, Huang Gao, Darrell Trevor*. Rethinking the value of network pruning // arXiv preprint arXiv:1810.05270. 2018.

[12] *Ma Xiaolong, Qin Minghai, Sun Fei, Hou Zejiang, Yuan Kun, Xu Yi, Wang Yanzhi, Chen Yen-Kuang, Jin Rong, Xie Yuan*. Effective Model Sparsification by Scheduled Grow-and-Prune Methods // arXiv preprint arXiv:2106.09857. 2021.

[13] *Malinen Mikko I, Fränti Pasi*. Balanced k-means for clustering // Joint iapr international workshops on statistical techniques in pattern recognition (spr) and structural and syntactic pattern recognition (sspr). 2014. 32–41.

[14] *Mishra Asit, Latorre Jorge Albericio, Pool Jeff, Stosic Darko, Stosic Dusan, Venkatesh Ganesh, Yu Chong, Micikevicius Paulius*. Accelerating sparse deep neural networks // arXiv preprint arXiv:2104.08378. 2021.

[15] *Pool Jeff, Yu Chong*. Channel Permutations for N: M Sparsity // Advances in Neural Information Processing Systems. 2021. 34.

[16] *Silver David, Huang Aja, Maddison Chris J, Guez Arthur, Sifre Laurent, Van Den Driessche George, Schrittwieser Julian, Antonoglou Ioannis, Panneershelvam Veda, Lanctot Marc, others* . Mastering the game of Go with deep neural networks and tree search // nature. 2016. 529, 7587. 484–489.

[17] *Sui Yang, Yin Miao, Xie Yi, Phan Huy, Aliari Zonouz Saman, Yuan Bo*. CHIP: CHannel Independence-based Pruning for Compact Neural Networks // Advances in Neural Information Processing Systems. 2021. 34.

[18] *Wen Wei, Wu Chunpeng, Wang Yandan, Chen Yiran, Li Hai*. Learning structured sparsity in deep neural networks // Advances in neural information processing systems. 2016. 29.

[19] *Zhang Xiang, Zhao Junbo, LeCun Yann*. Character-level convolutional networks for text classification // Advances in neural information processing systems. 2015. 28.

[20] *Zhou Aojun, Ma Yukun, Zhu Junnan, Liu Jianbo, Zhang Zhijie, Yuan Kun, Sun Wenxiu, Li Hongsheng*. Learning n: m fine-grained structured sparse neural networks from scratch // arXiv preprint arXiv:2102.04010. 2021.

[21] *Zhou Yangjie, Yang Mengtian, Guo Cong, Leng Jingwen, Liang Yun, Chen Quan, Guo Minyi, Zhu Yuhao*. Characterizing and Demystifying the Implicit Convolution Algorithm on Commercial Matrix-Multiplication Accelerators // 2021 IEEE International Symposium on Workload Characterization (IISWC). 2021. 214–225.

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

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [No]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [N/A]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [N/A]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]

5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]