Accelerating Sparse CNN Inference on GPUs with Performance-Aware Weight Pruning

Masuma Akter Rumi
The University of Iowa
masumaakter-rumi@uiowa.edu

Xiaolong Ma
Northeastern University
ma.xiaol@husky.neu.edu

Yanzhi Wang
Northeastern University
yanz.wang@northeastern.edu

Peng Jiang
The University of Iowa
peng-jiang@uiowa.edu

ABSTRACT

Weight pruning is a popular technique to reduce the size and computation complexity of the Convolutional Neural Networks (CNNs). Despite its success in reducing the model size, weight pruning has brought limited benefit to the CNN inference performance, due to the irregularity introduced in the sparse convolution operations. In this work, we aim to improve the performance of sparse convolutions on GPUs by mitigating the irregularity. We find that the existing performance optimization techniques for sparse matrix computations fail to accelerate sparse convolutions, and we observe that the main performance bottleneck is caused by the heavy control-flow instructions. Based on the observation, we proposed a new GEMM-based implementation of sparse convolutions. Our main idea is to extract dense blocks of non-zeros in the sparse convolution kernels, and use dense matrix-matrix multiplication for these dense blocks to achieve high throughput. For cases where many non-zero weights cannot be grouped into dense blocks, we propose a *performance-aware re-pruning* strategy that removes the least important weights in the sparse kernels to further improve the throughput. The experimental results with five real-world pruned CNN models show that our techniques can significantly improve the layer-wise performance of sparse convolution operations as well as the end-to-end performance of CNN inference.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Software and its engineering** → **Source code generation**;

ACM Reference Format:
Masuma Akter Rumi, Xiaolong Ma, Yanzhi Wang, and Peng Jiang. 2020. Accelerating Sparse CNN Inference on GPUs with Performance-Aware Weight Pruning. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3410463.3414648

1 INTRODUCTION

Convolutional Neural Networks (CNNs) have become the key component of AI systems due to their high accuracy in many inference and recognition tasks. The high accuracy of CNNs, however, comes with a high computation cost. Accelerating CNN inference is therefore of great importance to many AI applications, especially those with critical time constraints, such as self-driving cars [34] and real-time translation [15].

As convolution operations constitute most of the computational burden in CNNs [11, 36], pruning the weights in convolutional layers has become a popular technique to accelerate CNN inference. Fig. 1a shows an example of pruned convolutional layer. The dense convolution operation (kernel1) collects input values from cells in a $3 \times 3 \times 3$ grid and updates the corresponding cell in the output with a weighted sum of the collected values. In the pruned convolution (kernel2), some weights are removed (shown as elements with value of 0), and the memory access and computation for the weights are saved.

There are mainly two types of CNN pruning. The general but *non-structured* weight pruning can achieve a high compression ratio without accuracy loss [16, 18]; however, it brings little performance benefit because the sparse structure leads to poor data locality and low parallelism [20, 30, 44]. Alternatively, the *structured* pruning (i.e., an entire filter or channel can be pruned) [20, 30, 44, 46] generates more hardware-friendly models but results in high accuracy loss.

Achieving both high accuracy and high performance for CNNs with weight pruning is a challenging task. Prior work has approached this task in one of the following two directions. Some works have focused on efficient implementation of sparse convolutions with non-structurally pruned models, mainly by improving data locality and hardware utilization [10, 36]. The performance gains are limited, due to the sparse nature of the computation. Another approach is to design more hardware-amenable pruning strategies [8, 29]. For example, a hybrid strategy by combining structured and non-structured pruning can achieve good accuracy while maintaining some regular patterns in the pruned model for efficient hardware processing [29, 33]. These works, however, lack a careful examination of the code optimization opportunities, resulting in restricted pruning choices and sub-optimal performance.

In this work, we target the goal of high accuracy and high performance pruned CNNs from both directions. Given a non-structurally pruned model, we first explore the code optimization opportunities for sparse convolutions. We found that a sparse convolution can be implemented as a sparse-matrix dense-matrix multiplication (SpMM) if we store the sparse convolution kernels into a sparse matrix. However, existing SpMM implementations cannot deliver satisfactory performance for sparse convolutions. The reason is that these implementations mainly target large sparse matrices where data locality can be effectively improved by data reorganization [22, 23, 25], whereas the sparse matrices of the pruned convolution kernels are small and have less freedom of data reorganization. We observe that the main performance bottleneck in small sparse matrix multiplications is the control-flow instructions instead of data locality. Based on this observation, we reorganize the non-zero weights in the sparse convolution kernels into multiple

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

ACM ISBN 978-1-4503-8075-1/20/10…$15.00
https://doi.org/10.1145/3410463.3414648

dense blocks and a sparse block, and we use dense matrix-matrix multiplication (GEMM) for the dense blocks to reduce the control-flow instructions. The more non-zeros we put into the dense blocks, the fewer control-flow instructions, but the more extra computation we will have for the zero weights filled in the dense blocks. To achieve the best performance, we propose a new data reorganization algorithm that can be tuned to find the best trade-off between the dense blocks and the sparse block.

Although our implementation achieves some speedup compared with existing SpMM implementations, the performance is still unsatisfactory for many sparse convolution kernels. We found that, when some of the non-zeros in the sparse matrix are extremely scattered, they become the performance bottleneck no matter whether we put them into the dense blocks or leave them in the sparse block. Therefore, to further improve the performance, we propose a *performance-aware re-pruning* strategy to remove some of the least important weights in the sparse kernels. To maximize the speedup while maintaining a good accuracy, our re-pruning strategy considers both the location and the magnitude of a weight when making the pruning decision. Finally, with a slight re-training, we can recover the accuracy of the re-pruned model and achieve significant speedups for the convolutional layers.

To automate our code optimization and the performance-aware weight pruning, we propose OptPrune, a system that automatically performs CNN pruning and generates optimized CUDA code for sparse convolutions. Given a CNN model, our system first uses an ADMM-based pruning algorithm [46] to obtain a non-structurally pruned model. Then, it applies weight reorganization and generates CUDA code for each of the sparse convolutional layers. If the performance gain is small compared with unpruned convolutions, we further apply the performance-aware re-pruning and use a traditional SGD algorithm to recover accuracy for the re-pruned model. If the accuracy is recovered, our system will output the re-pruned convolutional layers along with the CUDA code for the sparse convolutions.

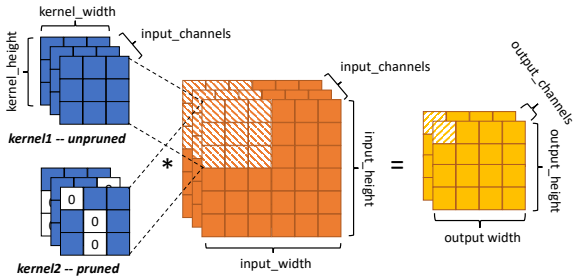
The experimental results show that, for sparse convolution kernels of different sizes found in real pruned CNN models, our implementation of sparse convolution consistently outperforms the state-of-the-art implementation of SpMM. Compared with the state-of-the-art implementation of unpruned convolutions, our code achieves up to 4.22x layer-wise speedup on an Nvidia Tesla P100 GPU, and up to 4.55x layer-wise speedup on an Nvidia Jetson TX2 GPU. With the performance-aware re-pruning, the layer-wise speedups can be up to 5.59x on P100 and up to 5.17x on TX2. If plugging our sparse convolution operations into PyTorch for end-to-end inference, the inference performance can be improved by up to 3.52x on P100 and up to 3.62x on TX2, compared with the state-of-the-art implementation of unpruned models.

2 BACKGROUND AND MOTIVATION

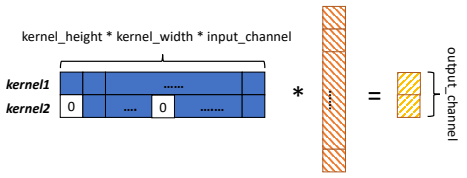
To facilitate our discussion and motivate our work, we give a background on convolution operation and its implementation methods, and then we explain the challenges in accelerating convolution operations by CNN pruning.

2.1 Convolution Operation and Its Implementation

Convolution is the most time consuming operation in CNN inference [11, 36]. As shown in Fig. 1a, a *convolution kernel* (also



(a) An example convolution with two kernels applied to one image



(b) Convolution operation represented as matrix multiplication

Figure 1: Computations in a convolution operation.

called *filter*) is a three dimensional array of size *kernel_height * kernel_width * input_channels*. The elements in the convolution kernel are called *weights*. The convolution operation maps the convolution kernel to an input position at a time and computes a weighted sum of the mapped input values. The weighted sum is stored to a cell in the output. The convolution operation slides over all the positions in the input feature map and generates all output cells. A convolutional layer may have multiple kernels. Each convolution kernel produces one output channel; hence the number of kernels is equivalent to the number of output channels. In this example, we apply two convolution kernels to the input, so we have two output channels.

If we unroll the input feature map and the convolution kernels, we can see that the computation is actually a matrix-matrix multiplication (GEMM), as shown in Fig. 1b. The convolution kernels are stored in different rows of a matrix, and the input data are stored as columns of another matrix. Each row of the output matrix is an output channel. This GEMM-based implementation is also called lowering method [12]. Because of its simplicity and the fact that GEMM is highly optimized, the lowering method is widely adopted by machine learning frameworks and libraries (e.g. TensorFlow [6], Caffe [24], cuDNN [12], and MKL-DNN [5]). The original lowering method needs to reorganize the input data into columns of the second matrix, resulting in huge amount of additional storage because the same input cells are repeated in different columns. An improved implementation collects the cells from input data at run-time, avoiding explicit storage of the second matrix [12].

Other implementation methods include direct convolution [28, 42], fast Fourier transform (FFT) based convolution [21, 31, 43], and the Winograd algorithm [27, 35]. FFT-based convolution can reduce the arithmetic complexity compared with direct methods; however, it needs to pad the convolution kernels to the same size as the input image, which incurs additional memory and computation time. Winograd algorithm can significantly reduce the arithmetic complexity for 3×3 kernels; however, it incurs increased memory usage and cannot be generalized to other kernel sizes. Moreover, these two algorithms are not applicable to sparse convolutions. Direct

convolution is similar to the improved GEMM-based convolution, except that it tiles the kernel matrix (the first matrix in Fig. 1b) only in the *input_channel* dimension, and thus its performance is sensitive to input channels [11, 28].

2.2 Performance Challenges with CNN Pruning

Due to the above reasons, we choose the improved GEMM-based implementation for sparse convolutions. It is obvious from Fig. 1b that the computation becomes sparse-matrix dense-matrix multiplication (SpMM) with pruned convolution kernels.

According to the structure of pruned models, there are mainly two CNN pruning approaches: *non-structured pruning* and *structured pruning*. In structured pruning, an entire kernel or channel can be removed [44]. Filter pruning eliminates entire convolution kernels, leading to the removal of entire rows of the kernel matrix. Channel pruning eliminates entire channels of the convolution kernels, leading to the removal of entire columns of the kernel matrix. It is obvious that the performance will be improved because we skip the computation and memory accesses for entire rows/columns of the kernel matrix. However, because the information of entire input channels or output channels (which are the input channels for the next layer) is lost, the structured pruning suffers from notable accuracy loss [20, 44].

In non-structured pruning, weights at any location of the model can be pruned [46]. Non-structured pruning can achieve a high pruning rate without accuracy loss; however, it is difficult to achieve better (hardware) performance due to the irregularity in memory access and computation. First, removing weights does not necessarily lead to fewer memory accesses. It is possible that the weights are scattered in different kernels and different input channels, which means that the convolution operation still needs to access all the cells in input and output. Second, the sparse weights require heavy control-flow instructions for indexing, which degrades instruction-level parallelism. Third, it introduces load imbalance as different kernels are likely to have different workloads. In fact, the throughput of cuBLAS's single-precision GEMM can be up to 8,000 GFLOPS [2] on an Nvidia P100 GPU, whereas the throughput of the state-of-the-art implementation of single-precision SpMM is about 800 GFLOPS on the same device [23, 25]. Because of the huge throughput gap, it is a major challenge to improve the performance of CNN inference on GPUs while retaining its accuracy.

3 OVERVIEW OF OPTPRUNE

Fig. 2 shows the workflow of our system. Given an unpruned CNN model, our system first performs non-structured weight pruning with the Alternating Direction Method of Multipliers (ADMM) algorithm. Previous works have shown that ADMM-based algorithms can achieve the state-of-the-art compression ratio for CNNs with little accuracy loss [33, 46]. Readers can refer to [46] for more details.

Weight reorganization: For each sparse convolutional layer in the pruned model, we next reorganize the non-zero weights and group them into multiple dense blocks and a sparse block. Fig. 3a and 3b shows an example of the weight reorganization procedure. Suppose we have a pruned convolutional layer with five kernels as shown in Fig. 3a (the indexed cells are non-zeros). We can group 12 of the non-zeros into two dense blocks and put the remaining 4 non-zeros in a sparse block as shown in Fig. 3b. The dense blocks are

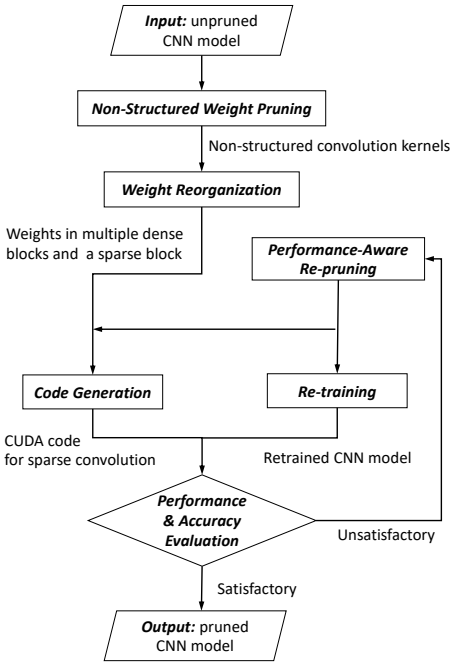
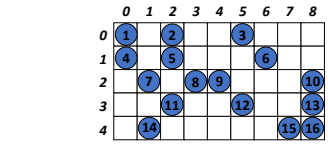
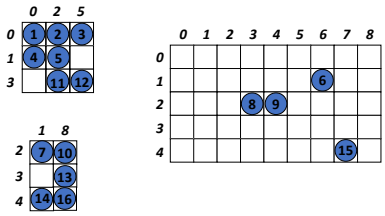


Figure 2: Workflow of our system.



(a) A pruned convolutional layer with five kernels



(b) Weights are reorganized into two dense blocks and a sparse block

Figure 3: An example of weight reorganization.

stored as dense matrices in row-major order with empty cells filled with zero, and the sparse block is stored in Compressed Sparse Row (CSR) format [22]. We took inspiration from the data reorganization idea proposed in [25] and proposed a new algorithm for extracting dense blocks in a sparse matrix. By using dense matrix-matrix multiplication (GEMM) for these dense blocks, we can eliminate the heavy control-flow instructions and improve the throughput.

Performance-aware re-pruning: In case the performance is still not satisfactory after weight reorganization, we seek to remove the least important weights from the sparse kernels. When choosing which weights to remove, we consider both their magnitudes and data reuses with other weights. Intuitively, removing a weight

in a row/column with few non-zeros is more beneficial to the performance than removing a weight in a denser row/column because the weights in the sparse rows have little data reuse. For example, in Fig. 3b, removing weight#6 is preferred to removing weight#8 because removing #6 will save the memory access to both row 6 of input matrix and row 1 of output matrix while removing #8 only saves the memory access to row 3 of input matrix. To minimize the effect on accuracy, we set an upper bound to the ℓ_2 norm of removed weights. Our goal is to maximize the benefit to the performance within the constraint that removed weights do not exceed the ℓ_2 norm upper bound. The upper bound is a tuning parameter, which introduces a trade-off between performance and accuracy. A larger upper bound allows more aggressive pruning but may result in more accuracy loss.

Re-training and parameter tuning: The re-pruned model needs to be re-trained to recover accuracy. For each convolutional layer, we mask the zero weights and insert it back to the CNN model. The empty cells in the dense blocks are not masked – they can be re-trained to have non-zero values to help recover accuracy. We apply the traditional SGD algorithm with the learning rate set to the value used in the last iteration of a normal training procedure. Because only a small number of weights are modified in re-pruning, the re-training starts from a good initial point and can converge to a good accuracy in a small number of iterations. This incremental training property allows our system to try different re-pruning configurations and tune the performance in a reasonable amount of time.

4 IMPLEMENTING SPARSE CONVOLUTIONS WITH GEMM

As explained in the background section, sparse convolutions can be implemented as SpMM. Although previous works have studied SpMM on GPUs [22, 23, 25], their optimization techniques mainly target large sparse matrices with at least 10,000 rows and columns that are found in scientific computing applications, and they can not deliver good performance for sparse convolutions where the number of convolution kernels is usually smaller than 1000. In fact, we adopted a state-of-the-art implementation of SpMM from [25] for sparse convolution with real-world pruned models from [46], and we found that the sparse convolutions do not run much faster (and can even be slower) compared with the original dense convolutions implemented as GEMM.

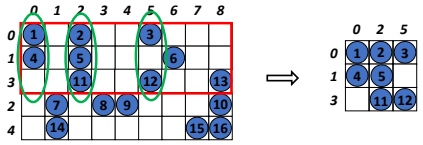
SpMM in general has a much lower throughput than GEMM on GPUs for three reasons: 1) the sparse memory access pattern has poor data locality, 2) the sparse computation pattern incurs heavy control-flow instructions, which degrade instruction-level parallelism, and 3) the unevenly distributed non-zeros lead to load imbalance among parallel tasks. While previous research has worked on improving data locality and load balancing with data reorganization, our observation is that, when the sparse matrix is small, data reorganization is less effective in improving data locality and load balancing, and reducing the control-flow instructions is more critical to the performance. If we can reorganize the non-zero weights and put most of them into dense blocks as in Fig. 3b, then we can use GEMM for the computation of the dense blocks, and the throughput will be improved.

This section introduces our weight reorganization algorithm and gives details of our implementation of sparse convolutions.

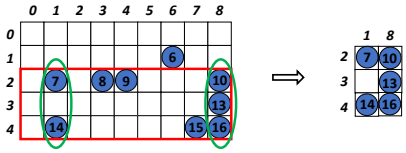
Algorithm 1: Reorganizing non-zeros in a sparse matrix into dense blocks

Input : sparse matrix S; tuning parameters t_1, t_2, B_1, B_2
Output: dense blocks of non-zeros

repeat
  Divide the rows in S into t_1 groups $\{g_1, g_2, ...g_{t_1}\}$ with hypergraph partitioning (hMETIS);
  foreach $g_i \in \{g_1, g_2, ...g_{t_1}\}$ **do**
    if g_i *has no less than* B_1 *rows* **then**
      Select columns in g_i with at least t_2 non-zeros;
      if *at least* B_2 *columns are selected* **then**
        Output selected columns in g_i as a dense block and remove non-zeros in the dense block from S;
until *no dense block is found*;



(a) Row 0, row 1 and row 3 are grouped together, and columns with at least two non-zeros are selected



(b) Row 2, row 3 and row 4 are grouped together, and columns with at least two non-zeros are selected

Figure 4: An illustration of Algorithm 1 applied to the sparse matrix in Fig. 3a.

4.1 Extracting Dense Blocks in the Sparse Kernel Matrix

To find a dense block of non-zeros in a sparse matrix, we need to bring similar rows and columns together. We adopt the idea from [25] and use the *Jaccard similarity* between non-zero column indices as the similarity between two rows in a sparse matrix. For example, in the sparse matrix in Fig. 3a, row 0 and row 1 share two columns in four distinct columns, so their similarity is 2/4; row 0 and row 3 also share two columns in four distinct columns, so their similarity is also 2/4. Intuitively, if two rows have a large similarity, they should be grouped together, and the non-zeros in the identical columns will form a dense block. For the sparse matrix in Fig. 3a, we can put row 0, row 1 and row 3 together, and pick out the columns with at least two non-zeros. As shown in Fig. 4a, this brings us the first dense block in Fig. 3b.

Different from [25] where the authors proposed an approximate algorithm for fast clustering of rows in large matrices, we use hypergraph partitioning for more accurate grouping of rows in small matrices. The main idea is to consider each row of the sparse matrix as a node in a hypergraph and each column as a hyperedge in the graph that connects the rows. An optimal partitioning of the hypergraph will correspond to a partitioning of the rows in which

the number of different columns among groups is minimum, which indicates that the identical columns within a group is maximum. In our implementation, we use hMETIS [1] for the hypergraph partitioning task.

After a dense block is identified in the sparse matrix, we remove the non-zeros in the dense block from the sparse matrix, and repeat the above procedure until no dense block with at least B_1 rows and B_2 columns can be found. For example, after we identify the first dense block in Fig. 3b, the non-zeros that are left in the sparse matrix are as shown in Fig. 4b. Since row 2, 3 and 4 have identical columns, we can group them together and pick out the columns with at least two non-zeros. This brings us the second dense block in Fig. 3b. Because the remaining non-zeros are in three different rows and four different columns, they will be left in the sparse block.

Algorithm 1 summarizes the above procedures. In each iteration of the algorithm, we first group similar rows of the sparse matrix together using the hypergraph partitioning function in hMETIS. The number of groups is set to t_1. Then, for each group with no less than B_1 rows, we select the columns with at least t_2 non-zeros. Next, if there are at least B_2 of such columns, we output these columns in the group as a dense block and remove the outputted non-zeros from the sparse matrix S. Finally, if no dense block is found, the algorithm stops and all the remaining non-zeros are put in a sparse block.

After the sparse matrix is divided into multiple dense blocks and a sparse block, we use GEMM for the computation with the dense blocks and use a row-wise implementation of SpMM [23] for the computation with the sparse block. The CUDA code can be easily generated with a GEMM-based implementation of dense convolutions and an SpMM kernel.

Parameter tuning: In Algorithm 1, t_1, t_2, B_1 and B_2 are tuning parameters. B_1 and B_2 control the minimum size of the dense blocks. They should be large enough so that the kernel launch overhead can be amortized. Parameter t_1 controls how clustered the rows are in the reordered groups. We want to set it to a value so that only rows with enough common columns will be grouped together. Parameter t_2 controls how dense each block is. A smaller t_1 leads to larger and sparser blocks, while a larger t_2 leads to smaller and denser blocks. We set these parameters to different values and produce a set of potential solutions for performance evaluation. As an example, the reorganization in Fig. 3b is a result of Algorithm 1 with $t_1 = 2, t_2 = 2, B_1 = 3, B_2 = 2$. If we set B_1 to 2, the algorithm will stop after first iteration, and weight #13 will be put in the sparse block. The performance of the different reorganizations of weights depends on different GPU architecture and input/output sizes. Since the reorganization procedure and the convolution operations are fast, our system simply runs all these different reorganizations and chooses the one with the best performance.

Data reorganization overhead: Note that Algorithm 1 is done at compile-time. Because the number of rows of the kernel matrix is small (normally ranges from 64 to 1024 in CNN models), the hypergraph partitioning in Algorithm 1 runs fast. In our experiments, our weight reorganization takes less than 5 seconds for each convolutional layer. The dense blocks are stored with their row and column indices in the kernel matrix. GEMM uses these indices to directly access the actual location in the input and output matrix, so that the data layouts of the input and output features of each convolutional layer are not changed. Because the threads in a warp compute different columns in the same row of the output matrix and the output matrix is stored in row-major order, the memory accesses within a warp are coalesced. The scattering of rows by different warps incurs no extra overhead on a GPU.

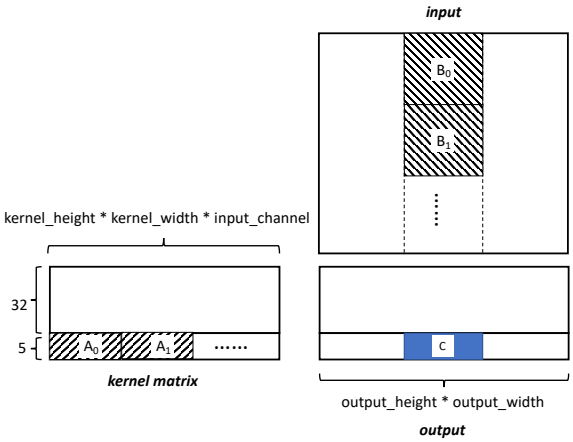


Figure 5: An inefficiency of existing GEMM-based implementations of dense convolutions on GPUs: the small data reuse with B_i cannot justify the overhead of loading it to shared memory.

4.2 Achieving Linear Performance for Arbitrary Matrix Size

A problem with existing GEMM-based implementations of dense convolution operations is that they are only optimized for kernel matrices with a multiple of 32 rows [38]. This is not a problem for unpruned convolutions since the numbers of kernels in CNN models are normally multiples of 32. However, it will be inefficient for our optimization method because the dense blocks extracted from the sparse kernels have arbitrary number of rows.

Fig. 5 illustrates the problem with existing GEMM-based implementations. To compute a block C in the output matrix, a thread block needs to load A_i and B_i into shared memory and accumulates their product to C. As we can see, each element in B_i is only used for 5 times, which cannot justify the overhead of loading B_i into shared memory. Also, when loading A_i to shared memory, many of the threads in the thread block are idle. In other words, convolution with 37 kernels is almost as expensive as convolution with 64 kernels in existing GEMM-based implementations. This means that even if the sparse matrix is perfectly divided into dense blocks by our weight reorganization algorithm, the program will not run faster if the numbers of rows in dense blocks are not multiples of 32.

We address this issue by dividing a dense block of r rows into a block of $\lfloor r/32 \rfloor \times 32$ rows and a block of r mod 32 (the remainder of r divided by 32) rows. For the first block, since the number of rows is a multiple of 32 and there are enough data reuse with the input and output, we adopt a standard GEMM-based implementation using shared memory to cache the input and output matrix. For the second block, because the data reuse with B_i is small, we only use shared memory for A_i and C. To reuse the data in B_i, we make each thread iterate over the elements in A_i column-by-column so that only one global memory access is needed for all elements in a column of A_i. To enable the output data reuse among elements

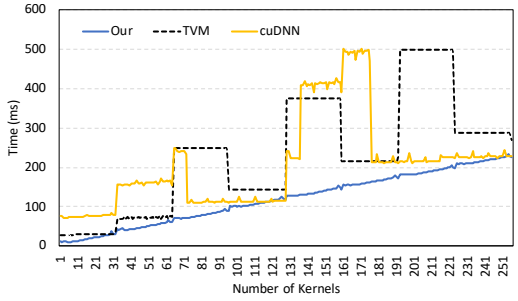


Figure 6: The execution time of applying different numbers of $3 \times 3 \times 64$ convolution kernels to a $255 \times 255 \times 64 \times 64$ input on an Nvidia P100 GPU with different implementations.

in each row of A_i, we store the temporary result of each row in a GPU register. The two blocks are assigned to two CUDA kernels in two different streams so that the kernel launch time for the second block can be hidden. The only problem is that there is not enough parallelism among the rows when r mod 32 is large, say 31. In our implementation, we divide the r mod 32 rows evenly into two parts if r mod 32 is greater than 16.

Fig. 6 shows a performance comparison of our implementation of dense convolution with the implementations in cuDNN7.6.5 [4] and TVM [9]. We can see that cuDNN and TVM only achieve good performance when the number of kernels is a multiple of 32, while our implementation achieves almost linear performance with the number of kernels. The performance with more kernels and other kernel sizes ($1 \times 1, 5 \times 5, 7 \times 7$) follows a similar pattern. This linear performance is critical to the efficiency of our GEMM-based implementation of sparse convolutions.

5 PERFORMANCE-AWARE RE-PRUNING

With GEMM-based implementation of sparse convolutions, we reduce the control-flow instructions at the cost of adding extra computation in the dense blocks. It is possible that the performance of the best trade-off is still unsatisfactory because there is a lack of dense structure in the pruned model. For example, if all the non-zero weights in the pruned kernel matrix are in different rows and columns, the convolution operation is not likely to run faster no matter how we reorganize the weights.

In traditional CNN pruning methods [18, 32] including the ADMM algorithm we used for initial pruning [46], weights are removed from the model based on their magnitudes. Intuitively, if a weight has a small magnitude, which means that it detects less important features than those with large magnitudes, we can remove the weight without much accuracy loss. The problem of this magnitude-based pruning is that the pruned model may not contain any dense structure because no structural constraint is put on the pruning procedure. In this section, we describe a performance-aware re-pruning method that re-structures the initially pruned convolution kernels to achieve a better performance while maintaining the accuracy.

5.1 A Simplified Performance Model

Our main idea is to remove some of the least important weights in the reorganized blocks based on both their magnitudes and their data reuse. To explain the idea, we first give a performance model of our implementation. Let us consider the reorganized weights in Fig. 3b. To reduce the computation overhead of dense blocks, we

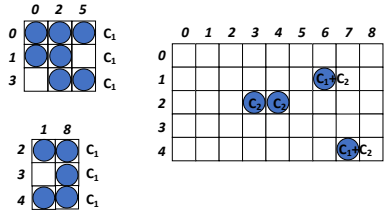


Figure 7: The performance benefit of removing the weights in Fig. 3b: C_1 is the overhead of writing an output row and C_2 is the overhead of reading an input row.

can only remove an entire row or column. Suppose the time for writing a row to the output matrix is C_1. The performance benefit of removing an entire row in a dense block is C_1.[1] Suppose the time for reading a row from the input matrix is C_2. If an element in the removed row happens to be the only element in its column, then the performance benefit is increased by C_2. For the sparse block, with the row-wise SpMM implementation [23], the results in each row are accumulated in registers with perfect data reuse, but there is little data reuse when accessing the input among the rows. If we remove a weight in the sparse block, we save the read of a row from the input matrix, and the benefit is C_2. If that weight happens to be the only weight in a row, we also save the write to the output matrix, and the benefit is increased by C_1. As an example, Fig. 7 shows the performance benefit of removing the weights in the reorganized blocks in Fig. 3b.

5.2 Re-pruning with Dynamic Programming

To maintain the accuracy, we want to keep the magnitudes of removed weights small. Specifically, we set the upper bound of the ℓ_2 norm of removed weights to a portion of the ℓ_2 norm of all non-zero weights in a convolutional layer. Given the upper bound, our goal is to maximize the performance benefit of removed weights. More formally, the problem can be defined as

$$\begin{aligned} &\max && Benefit(R), \\ &\text{subject to} && \sum_{w_i \in R} w_i^2 \le B_3 \sum_j w_j^2, \quad B_3 \in [0,1] \end{aligned} \tag{1}$$

where R is the set of removed weights, $Benefit$ is the total performance benefit of removing the weights in R, and B_3 is the upper bound ratio. The problem is similar to the knapsack problem [13] where we select items within limited capacity to maximize value. The only difference is that the performance benefit of removing each weight is not constant in our problem. For example, if weight#8 in Fig. 3b is removed, the performance benefit of removing weight #9 will increase to $C_1 + C_2$. However, since the change of benefits only happens when an entire row is removed, an algorithm for solving the knapsack problem is sufficient to find a good solution to our problem.

It is well known that the knapsack problem can be solved in pseudo-polynomial time by a dynamic programming algorithm if the constraint is integral [13]. To use the classic dynamic programming algorithm for our problem, we convert the squared value of each weight to an integer. More specifically, in our implementation, we set the weight to $\lceil 1000 w_j^2 \rceil$ for the algorithm. And the upper

[1] We do not consider the saved arithmetic operations because memory access is much more expensive.

bound is also set to $\lceil 1000 B_3 \sum_j w_j^2 \rceil$. Since C_1 and C_2 are almost the same (the input dense matrix and the output matrix have the same number of columns), we set both of them to 1.

Note that our re-pruning does not necessarily lead to reduction of non-zero weights in the model. Because the dense blocks are filled with zeros, which can be re-trained to be non-zeros, the total number of non-zero weights might actually increase after re-pruning. These added weights in the dense blocks help recover the accuracy while not affecting the performance.

Parameter tuning: The upper bound ratio $B_3 \in [0, 1]$ is a tuning parameter – a larger B_3 leads to more aggressive pruning. In our system, we initialize B_3 to a relatively large value for aggressive pruning and then gradually decrease it to achieve a good accuracy. Because our re-pruning is based on a pruned model with high accuracy and it only changes a small number of weights, the re-training starts from a good initial point and can recover good accuracy with only a few epochs (less than 10 epochs in our experiments). Due to this incremental training property, our system is able to try different values for B_3 in a reasonable amount of time.

6 EXPERIMENTAL RESULTS

In this section, we present performance results of our GEMM-based implementation and performance-aware re-pruning for sparse convolutions against existing convolution implementations on different GPU architecture.

6.1 Experimental Setup

Platform: Our experiments are conducted on two platforms: an Nvidia Tesla P100 GPU which represents data-center servers and an Nvidia Jetson TX2 which represents embedded computing devices. The P100 GPU has 3584 cores, 16GB global memory with bandwidth of 732GB/s, 4MB L2 cache, and 64KB shared memory per thread block. The GPU on Jetson TX2 has 256 cores, 8GB global memory with bandwidth of 59.7GB/s, 1.5MB L2 cache, and 48KB shared memory per block. The code is compiled using NVCC 10.1 with O3 optimization. All tests were run five times, and average numbers are reported. Because performance variances were small, we do not show the range of times of the reported results.

Benchmarks: All implementations were run for all convolutional layers found in ResNet18, ResNet50 [19], VGG16 [40], and GoogleNet [41]. The different convolutional layers in these three CNNs span a wide range of sizes of input, output and kernel weights. They are also commonly used as benchmarks for evaluating the performance of convolution implementations. Since the performance of sparse matrix operations highly depends on specific sparse patterns, it is important not to evaluate with random sparse matrices. We use the ADMM-based algorithm to prune ResNet18 by eight times (`ResNet18-8x`), VGG16 by four times (`VGG16-4x`), GoogleNet by eight times (`GoogleNet-8x`) on CIFAR10 dataset [26], and ResNet18 by four times (`ResNet18-4x`), ResNet50 by eight times (`ResNet50-8x`) on ILSVRC 2012 (ImageNet) dataset [14]. The pruning is uniform across the layers. The number of weights in each of the pruned convolutional layers is 1/4 or 1/8 of that in the original unpruned model.

6.2 Layer-wise Speedups with Our Implementation

We compared the performance of our GEMM-based implementation of sparse convolution (`Pruned Our`) with a state-of-the-art SpMM-based implementation [25] (`Pruned SpMM`). When applying Algorithm 1, We fix B_1 and B_2 to 8 for both P100 and TX2, as we found when the number of rows or columns of a dense block is smaller than 8, its convolution overhead will not decrease with the size of the block. We also fix t_1 to 8, as we found the performance are almost the same with t_1 from 4 to 10. The only parameter that needs to be tuned for different kernel matrices is t_2. We found the best performance can be achieved with t_2 being any value between 4 to 16, so we set t_2 to $\{4, 8, 12, 16\}$ one-by-one and report the best performance. The number of dense blocks extracted by the algorithm ranges from 4 to 10 for different layers in different models.

Fig. 8 shows the layer-wise performance of convolutions in `ResNet18-8x` with different implementations. The performance is measured with an input batch size of 32. We also collected the results with batch size of 16 and 64. The results follow a similar pattern, so we only report the performance for batch size of 32 here. We use the execution time of cuDNN7.6.5 [4] on dense convolutions as the baseline, and represent the performance of other versions as the speedup against cuDNN. We also include the performance of optimized code generated by TVM [9] for dense convolutions. Because TVM does extensive performance tuning based on kernel sizes and GPU architecture, it generally runs faster than cuDNN. Neither cuDNN nor TVM supports sparse convolutions, so we only report their performance with the unpruned model.

As we can see from Fig. 8, the SpMM-based sparse convolutions can achieve good speedups against dense convolutions for most layers on both P100 and TX2. This is because the computation is reduced by 8x in the pruned model. Our GEMM-based implementation (`Pruned Our`) further improves the performance of sparse convolutions. It runs 1.04x to 1.45x faster than the SpMM-based implementation on both P100 and TX2, achieving 1.66x to 7.58x speedups against cuDNN and 1.02x to 4.24x speedups against TVM on P100, and 1.58x to 7.36x speedups against cuDNN and 1.07x to 4.55x speedups against TVM on TX2. The speedups of GEMM-based implementation over SpMM-based implementation are small because the non-zeros in the pruned model are scattered and there is little room for reducing the overhead of control-flow instructions without increasing much computation.

Fig. 9 shows the layer-wise performance of convolutions in `VGG16-4x` with different implementations. Because the pruning ratio is small and the reduced computation cannot justify the overhead of irregular memory accesses and increased control-flow instructions in the sparse convolutions, SpMM-based sparse convolutions run slower than dense convolutions for most layers on both P100 and TX2. Our GEMM-based implementation runs 1.18x to 1.54x faster than SpMM-based implementation, achieving 1.03x to 5.40x speedups against cuDNN for all layers and 1.08x to 1.49x speedups against TVM for 6 out of the 13 layers on P100. On TX2, our implementation achieves 1.05x to 2.69x speedups against cuDNN for all layers and 1.02x to 2.44x speedups against TVM for 7 out of the 13 layers. The results validate our discussion in Section 4 that reducing the control-flow by using dense computation kernels can help improve the performance of sparse convolutions. However, we can see that nearly half of the convolutional layers still run slower than dense convolutions, suggesting that non-structured pruning is not effective in improving the performance of CNN inference when the pruning ratio is small.

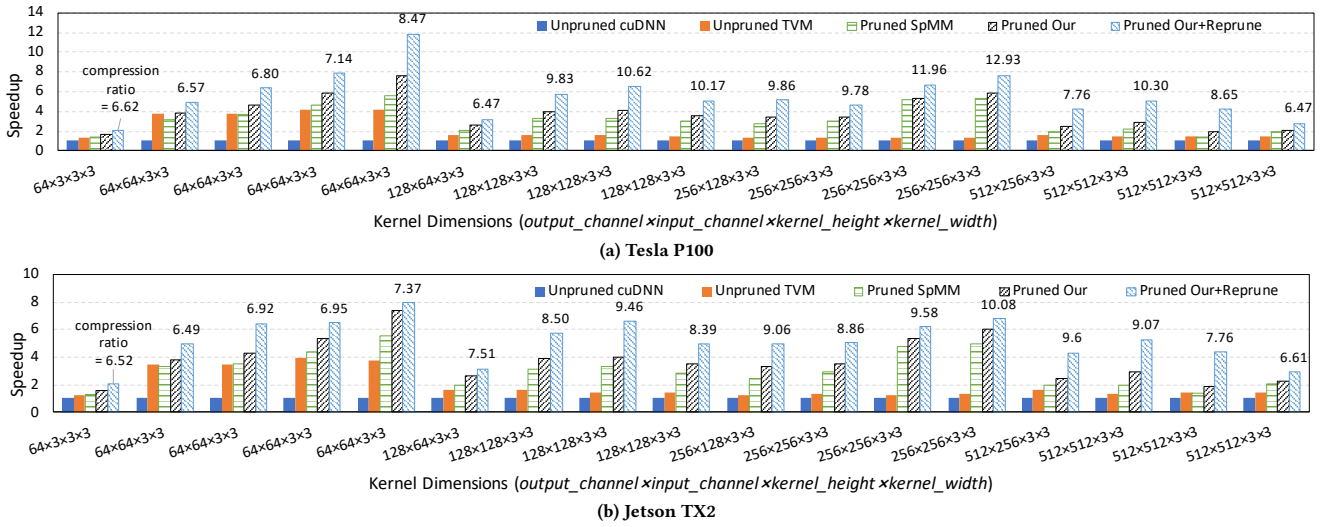


Figure 8: The layer-wise performance of convolution operations in `ResNet18-8x` with different implementations.

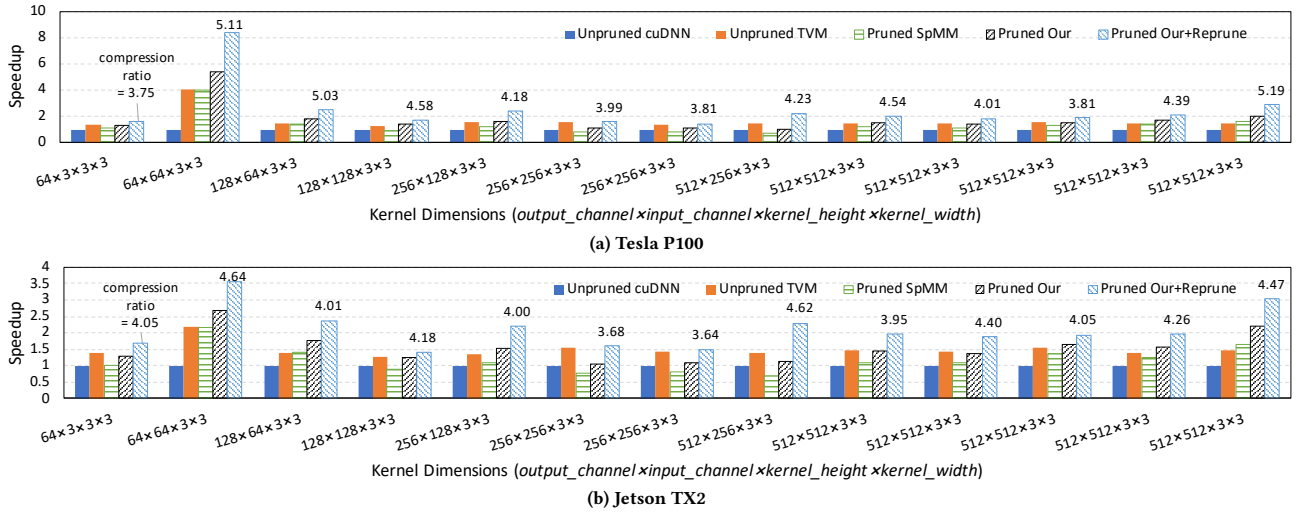


Figure 9: The layer-wise performance of convolution operations in `VGG16-4x` with different implementations.

The layer-wise performance of `GoogleNet-8x` on P100 GPU is shown in Fig. 10. Our implementation consistently outperforms the SpMM-based implementation and achieves 1.49x to 4.13x speedups against cuDNN and 1.27x to 3.94x speedups against TVM for all the layers. Similar to `ResNet18-8x`, the performance improvement with our GEMM-based implementation over SpMM-based implementation is limited. This is because when pruning ratio is high the non-zeros tend to be scattered in the pruned model and it is hard to maintain a high throughput for the sparse convolution operation. The performance on TX2 follows a similar pattern. We did not include the figure in the paper due to space limit. The speedups range from 1.42x to 4.08x against cuDNN, and ranges from 1.17x to 3.92x against TVM for all the layers.

Fig. 11 shows the layer-wise performance of convolution operations in `ResNet18-4x` with different implementations. Because the pruning ratio is small, SpMM-based sparse convolutions run slower than dense convolutions for most layers on both P100 and TX2. Our implementation achieves 1.42x to 4.68x speedups against cuDNN and 1.08x to 1.14x speedups against TVM for all layers on P100. On TX2, our implementation achieves 1.39x to 4.02x speedups against cuDNN and 1.04x to 1.19x speedups against TVM for all layers. Similar to `VGG16-4x`, the results here indicate that when pruning ratio is small and the pruned model is relatively dense, our GEMM-based implementation achieves good speedups over SpMM-based implementation by reducing the overhead of control-flow instructions; however, it achieves small speedups (and even slowdown) against unpruned models due to the small reduction of computation and the irregularity introduced in computation.

The performance of all the convolution operations in `ResNet50-8x` on P100 GPU is shown in Fig. 12. As we can see from the figure, sparse convolutions with SpMM-based implementation have no or little speedup against the dense convolutions with cuDNN or

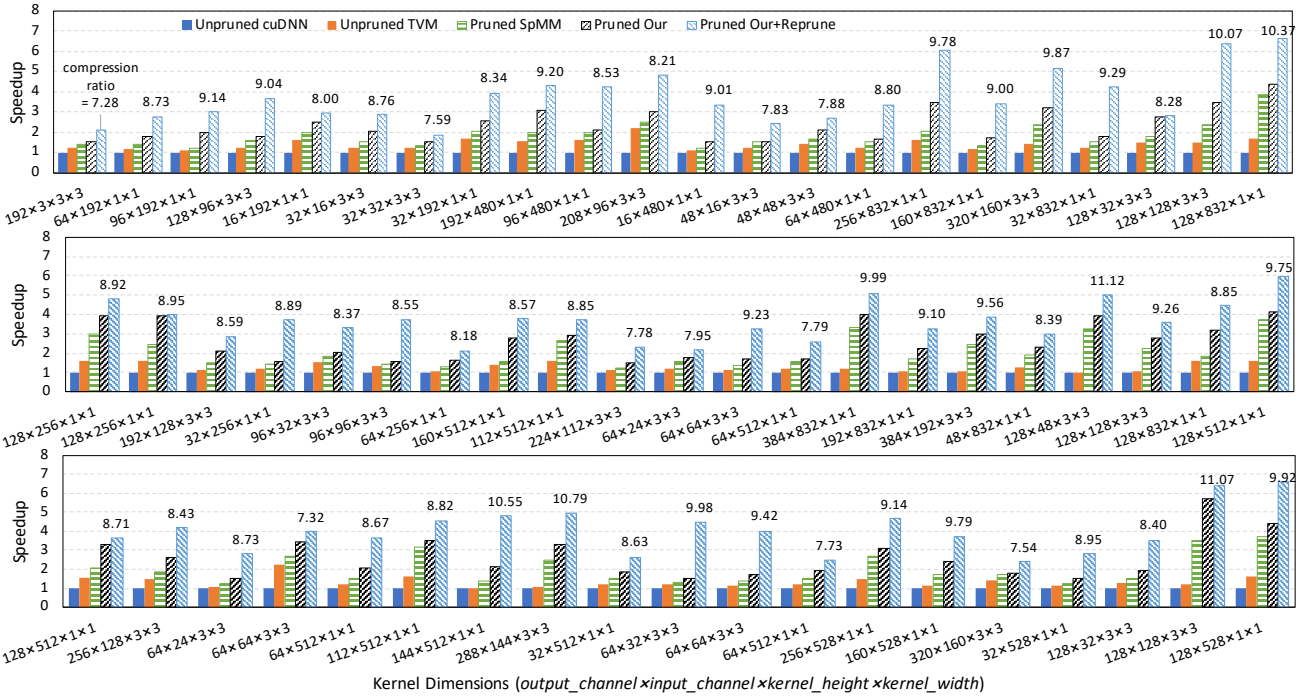
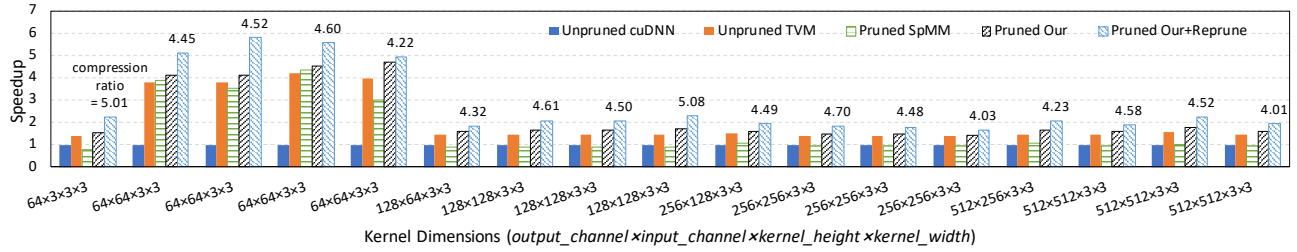
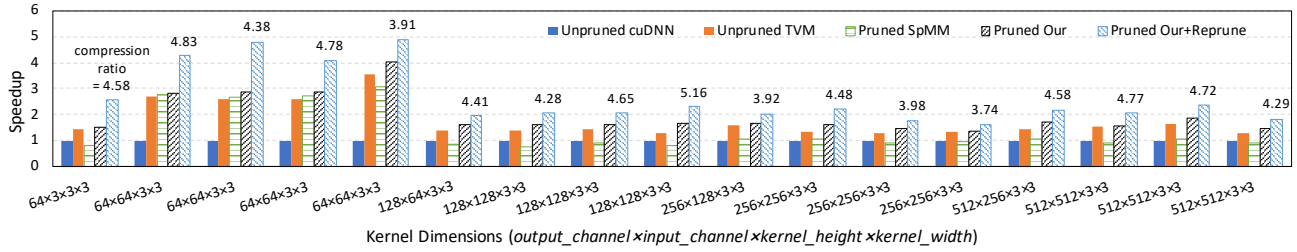


Figure 10: The layer-wise performance of convolution operations in `GoogleNet-8x` with different implementations on P100.



(a) Tesla P100



(b) Jetson TX2

Figure 11: The layer-wise performance of convolution operations in `ResNet18-4x` with different implementations.

TVM on both P100 and TX2. Our implementation achieves 1.14x to 4.91x speedups against cuDNN and 1.02x to 1.45x speedups against TVM for 46 out of the 49 layers on P100. The performance on TX2 follows a similar pattern. We do not include the figure due to space limit. Our implementation achieves 1.14x to 4.23x speedups against cuDNN and 1.07x to 1.81x speedups against TVM for 46 out of the 49 layers.

The above results illustrate the limitation of code optimization alone for non-structurally pruned convolutions. When the pruning ratio is high, the pruned models can achieve some speedups against unpruned models due to the significant reduction in computation. However, the pruned convolution suffers from a low throughput and there is little room for performance improvement with the GEMM-based implementation. When the pruning ratio is small, our GEMM-based implementation can maintain a relatively high throughput and achieve good speedups against SpMM-based implementation. However, because the computation is not reduced much compared

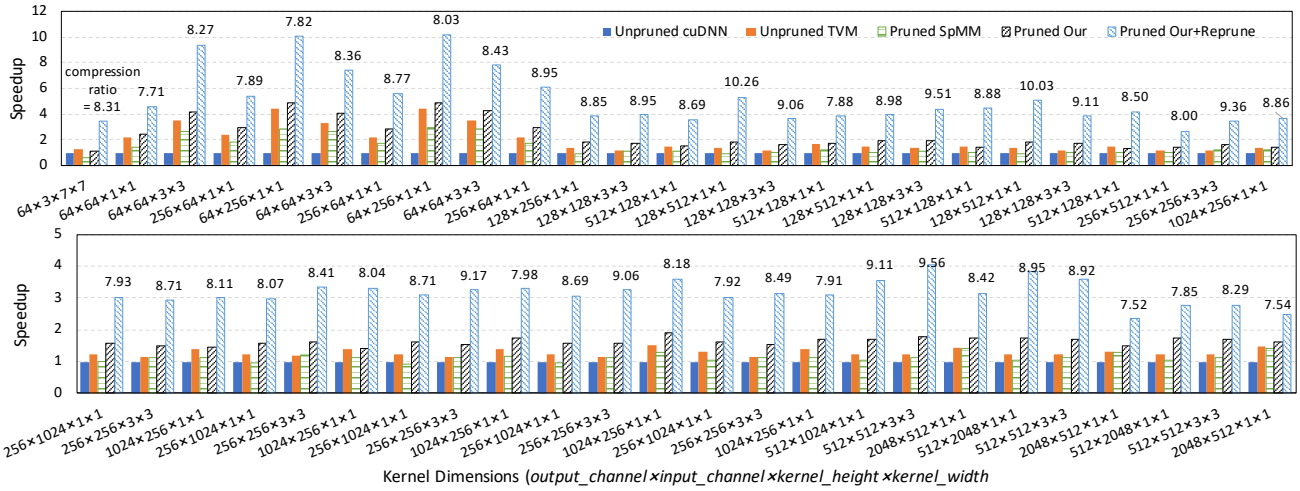


Figure 12: The layer-wise performance of convolution operations in `ResNet50-8x` with different implementations on P100.

Table 1: Top-1 test accuracy of different models on different datasets: `Full Acc` is the accuracy of unpruned model, `Orig Acc` is the accuracy of the ADMM pruned model, and `Retr Acc` is the accuracy of our re-pruned model after re-training.

Dataset	Model	Full Acc	Orig Acc	Retr Acc
CIFAR10	ResNet18-8x	94.14%	94.13%	94.34%
	VGG16-4x	93.54%	93.13%	93.43%
	GoogleNet-8x	95.48%	95.32%	95.42%
ImageNet	ResNet18-4x	69.92%	69.65%	69.75%
	ResNet50-8x	76.11%	75.55%	75.58%

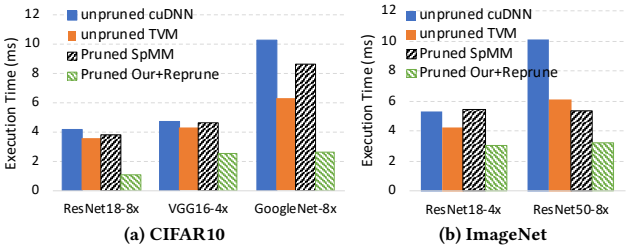


Figure 13: The end-to-end inference time for 32 images from different datasets on P100 GPU.

with dense convolutions, the pruned models can run even slower than the unpruned models.

6.3 Layer-wise Speedups with Weight Re-pruning

To overcome the limitation of code optimization, we propose a performance-aware weight re-pruning method in Section 5. To verify the effectiveness of re-pruning, we set the upper bound ratio B_3 in Section 5.2 to $\{0.3, 0.2, 0.1\}$ and re-prune all the convolutional layers in the CNN models.

For CIFAR10 dataset, we found the three models (`ResNet18-8x`, `VGG16-4x` and `GoogleNet-8x`) can recover accuracy after re-training for 10 epochs when we set B_3 to 0.2. Surprisingly, the accuracies of the re-pruned models are slightly higher than the original pruned model, as shown in Table 1. This is probably because the empty cells in the dense blocks are re-trained to non-zeros and help improve the accuracy. The performance of the re-pruned models are shown in Fig. 8, 9, and 10. The layer-wise compression ratio (i.e., the total number of weights in each convolutional layer of the unpruned models divided by the number of non-zeros after re-training) is also labeled in the figures. The total number of non-zeros is slightly decreased after re-training for all the models. We can see that re-pruning helps improve the performance. For `ResNet18-8x`, `Pruned Our+Reprune` achieves 1.29x to 5.59x layer-wise speedups against `Unpruned TVM` on P100, and 1.41x to 5.17x layer-wise speedups on TX2. For `VGG16-4x`, `Pruned Our+Reprune` achieves 1.06x to 2.11x layer-wise speedups over `Unpruned TVM` on P100 and 1.04x to 2.05x speedup on TX2. For `GoogleNet-8x`, the layer-wise speedups of `Pruned Our+Reprune` against `Unpruned TVM` range from 1.70x to 5.21x on P100 and 1.69x to 5.01x on TX2. As explained in Section 5, the performance is significantly improved by re-pruning because removing the non-zeros in the sparse block significantly reduces the memory access overhead, while changing the zero values in the dense blocks to non-zeros has no effect to the performance.

For ImageNet, when we set B_3 to 0.1, the accuracy of the two models (`ResNet18-4x` and `ResNet50-8x`) can be recovered after re-training for 30 epochs. As shown in Table 1, the recovered accuracy are also slightly higher than those of the original pruned model. The performance of the two re-pruned models are shown in Fig. 11 and Fig. 12. For `ResNet18-4x`, the layer-wise speedups of `Pruned Our+Reprune` against `Unpruned TVM` are 1.20x to 1.61x on P100, and 1.22x to 1.84x on TX2. For `ResNet50-8x`, the speedups against `Unpruned TVM` are 2.1x to 4.05x on P100, and 1.91x to 3.96x on TX2.

6.4 End-to-end Performance

As explained in Section 4, our GEMM-based implementation does not change the data layout of the input and output features of each convolutional layer. Thus, our generated CUDA code can be easily plugged into existing deep learning frameworks. To show the end-to-end performance with our GEMM-based implementation and weight re-pruning, we define a custom `Conv2d` operation in PyTorch with our generated code and use them to replace the original dense `Conv2d` operations. Fig. 13 shows the end-to-end inference time with different models for 32 images on P100 GPU. Compared

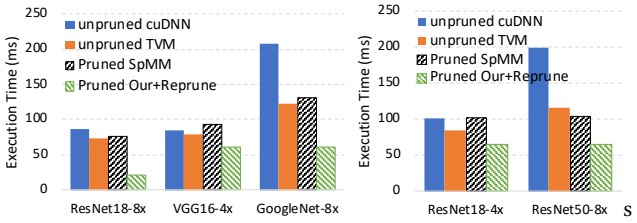


Figure 14: The end-to-end inference time for 32 images from different datasets on Jetson TX2.

with the original cuDNN implementation of unpruned models in PyTorch, our implementation and weight re-pruning achieves 4.03x overall speedup for ResNet18, 1.85x overall speedup for VGG16, 3.96x overall speedup for GoogleNet on CIFAR10 dataset, and 1.75x overall speedup for ResNet18, 3.11x overall speedup for ResNet50 on ImageNet dataset. Compared with TVM, our code achieves 1.39x to 3.52x overall speedups on P100 GPU. The performance on Jetson TX2 is shown in Fig. 14, the overall speedups against cuDNN range from 1.43x to 3.88x, and the overall speedups against TVM range from 1.21x to 3.62x, for different models on different datasets.

Since we simply replace the convolution operations in PyTorch with our code, all the end-to-end speedups above are due to the acceleration of convolution operations. Because convolution operations constitute most of the computational burden in CNNs [36] (75%∼ 85% in our experiment), the acceleration of convolution operations leads to a noticeable improvement of end-to-end performance. We can see that the models with higher pruning ratios (`ResNet18-8x`, `GoogleNet-8x`, `ResNet50-8x`) achieve higher speedups, while the models with smaller pruning ratios (`VGG16-4x`, `ResNet18-4x`) achieve smaller speedups. To show the benefit of our techniques, we also collected the execution time of the originally pruned models with an SpMM-based implementation (Pruned SpMM). The results show that, non-structured pruning brings little benefit to the end-to-end performance of CNN inference even with a large pruning ratio, and the pruned models in most cases run slower than an optimized implementation of unpruned models (unpruned TVM). Our GEMM-based implementation reduces the overhead of control-flow instructions in sparse convolutions and our re-pruning technique further reduces the memory access overhead, leading to 1.6x to 3.7x speedups over 'Pruned SpMM' on both platforms.

7 RELATED WORKS

This section summarizes the closely related works in CNN pruning and sparse matrix computation on GPUs.

CNN pruning: CNN pruning aims at reducing the size and complexity of CNN models so that they can fit on devices with limited compute and memory capacity. Early works on CNN pruning include [17, 18, 39, 46], in which iterative and heuristic methods with limited and non-uniform model compression rates are proposed. The pruned models in these works are non-structured, and their performance gain was not considered carefully. The performance issue with non-structured pruning is later identified in [20, 44], where the authors propose to prune the convolution kernels to smaller regular weight matrices so that they can be easily accelerated on CPUs and GPUs. The downside of these structured pruning methods is that they suffer from notable accuracy loss when the pruning rate increases. To combine the benefits of structured and non-structured pruning, hybrid pruning strategies have been proposed. Chen *et al.* proposed sparse complimentary convolution in which half of the weights with regular patterns in the original convolution kernels can be removed with little accuracy loss [8]. Ma *et al.* proposed pattern-based kernel pruning [29]. The convolution kernels can only be pruned to one of several pre-defined patterns so that the pruned model contains some regular structures. These hybrid approaches lack a careful examination of code optimization opportunities for the pruned models, resulting in restricted pruning choices and sub-optimal convolution performance. Zhu *et al.* proposed vector-wise pruning [47]. They divide the convolution kernel matrix into fixed size vectors and prune the kernel matrix in a relatively regular fashion by keeping the top-K elements in each vector. In addition to vector-wise pruning, the authors also adapt the tensor core architecture on GPUs to *sparse tensor core* in order to support efficient execution of their vector-wise pruned models. Different from their approach, we seek code optimization opportunities for sparse convolutions on existing GPU architecture and propose a pruning procedure that is aware of and amenable to the code optimizations on existing GPUs.

Code Optimization for Sparse Convolution: There has been little work on accelerating sparse convolutions on GPUs. Instead of pruning the convolution kernels, Peng *et al.* utilize the sparsity in input features of CNNs and propose a matrix splitting technique to achieve efficient CNN inference on heterogeneous platforms [37]. Nvidia's cuSparse [3] is a highly-optimized library for various sparse matrix computations including SpMM on GPUs. However, the SpMM routine in cuSparse cannot be directly used for sparse convolutions as it requires the input to be explicitly unrolled which incurs a large overhead. SpMM has caught the attention of researchers in HPC area in recent years due to their increasing application in scientific computing and data mining [7, 22, 23, 45]. Hong *et al.* [23] proposed an Adaptive Sparse Tiling (ASpT) technique to improve the data reuse for SpMM. ASpT achieves significant performance improvement over cuSparse and was considered the state-of-the-art of SpMM implementation. These previous works on SpMM however mainly focus on large matrices (with more than 10K rows and columns) found in scientific computing applications. The large size of the matrices prevents any expensive data reorganization to the matrices. Therefore, these techniques can only benefit sparse matrices with somewhat clustered non-zeros.

8 CONCLUSION

In this paper, we studied the problem of accelerating sparse CNN inference on GPUs. We focused on the efficient implementation of sparse convolutions and proposed a novel performance-aware re-pruning technique to improve the performance. We also implemented a system, OptPrune, that automatically performs CNN pruning and generates optimized CUDA code for pruned convolutions. We conducted experiments with five real-world pruned CNN models on an Nvidia Tesla P100 GPU and a Jetson TX2 GPU. The experimental results show that our techniques can achieve 1.05x to 5.59x layer-wise speedups against the state-of-the-implementation of unpruned convolutions on P100 and 1.04x to 5.17x layer-wise speedups on TX2, leading to 1.39x to 3.52x overall speedups for CNN inference on P100 and 1.21x to 3.62x overall speedups on TX2.

REFERENCES

[1] 2007. hMETIS. http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview

[2] 2016. CUDA8 Performance Overview. http://developer.download.nvidia.com/compute/cuda/compute-docs/cuda-performance-report.pdf

[3] 2019. The API reference guide for cuSPARSE, the CUDA sparse matrix library. https://docs.nvidia.com/cuda/cusparse/index.html Version 10.1.168.

[4] 2019. cuDNN Developer Guide. https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html.

[5] 2019. MKLDNN. http://intel.github.io/mkl-dnn/

[6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, and et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.

[7] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. 2014. Optimizing Sparse Matrix-Multiple Vectors Multiplication for Nuclear Configuration Interaction Calculations. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS '14)*. IEEE Computer Society, Washington, DC, USA, 1213–1222. https://doi.org/10.1109/IPDPS.2014.125

[8] Chun-Fu Chen, Jinwook Oh, Quanfu Fan, and Marco Pistoia. 2018. SC-Conv: Sparse-complementary convolution for efficient model utilization on CNNs. In *2018 IEEE International Symposium on Multimedia (ISM)*. IEEE, 97–100.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594.

[10] Xuhao Chen. 2018. Escort: Efficient Sparse Convolutional Neural Networks on GPUs. *CoRR* abs/1802.10280 (2018). arXiv:1802.10280 http://arxiv.org/abs/1802.10280

[11] Xiaoming Chen, Jianxu Chen, Danny Z. Chen, and Xiaobo Sharon Hu. 2017. Optimizing Memory Efficiency for Convolution Kernels on Kepler GPUs. In *Proceedings of the 54th Annual Design Automation Conference 2017 (DAC '17)*. ACM, New York, NY, USA, Article 68, 6 pages. https://doi.org/10.1145/3061639.3062297

[12] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *ArXiv* abs/1410.0759 (2014).

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[14] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

[15] Jonas Gehring, Michael Auli, David Grangier, and Yann Dauphin. 2017. A Convolutional Encoder Model for Neural Machine Translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vancouver, Canada, 123–135. https://doi.org/10.18653/v1/P17-1012

[16] Yiwen Guo, Anbang Yao, and Yurong Chen. 2016. Dynamic Network Surgery for Efficient DNNs. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. Curran Associates Inc., USA, 1387–1395. http://dl.acm.org/citation.cfm?id=3157096.3157251

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

[18] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'15)*. MIT Press, Cambridge, MA, USA, 1135–1143.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), 770–778.

[20] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel Pruning for Accelerating Very Deep Neural Networks. 1398–1406. https://doi.org/10.1109/ICCV.2017.155

[21] Tyler Highlander and Andres Rodriguez. 2016. Very Efficient Training of Convolutional Neural Networks using Fast Fourier Transform and Overlap-and-Add. *arXiv e-prints* (Jan 2016).

[22] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V. Çatalyürek, Srinivasan Parthasarathy, and P. Sadayappan. 2018. Efficient Sparse-matrix Multi-vector Product on GPUs. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '18)*. ACM, New York, NY, USA, 66–79.

[23] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive Sparse Tiling for Sparse Matrix Multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA, 300–314.

[24] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia (MM '14)*. ACM, New York, NY, USA, 675–678. https://doi.org/10.1145/2647868.2654889

[25] Peng Jiang, Changwan Hong, and Gagan Agrawal. 2020. A Novel Data Transformation and Execution Strategy for Accelerating Sparse Matrix Multiplication on GPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 376–388. https://doi.org/10.1145/3332466.3374546

[26] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. [n. d.]. CIFAR-10 (Canadian Institute for Advanced Research). ([n. d.]). http://www.cs.toronto.edu/~kriz/cifar.html

[27] A. Lavin and S. Gray. 2016. Fast Algorithms for Convolutional Neural Networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4013–4021. https://doi.org/10.1109/CVPR.2016.435

[28] Shigang Li, Yunquan Zhang, Chunyang Xiang, and Lei Shi. 2015. Fast Convolution Operations on Many-Core Architectures *(HPCC-CSS-ICESS '15)*. IEEE Computer Society, Washington, DC, USA, 316–323. https://doi.org/10.1109/HPCC-CSS-ICESS.2015.94

[29] Xiaolong Ma, Fu-Ming Guo, Wei Niu, Xue Lin, Jian Tang, Kaisheng Ma, Bin Ren, and Yanzhi Wang. 2019. PCONV: The Missing but Desirable Sparsity in DNN Weight Pruning for Real-time Execution on Mobile Devices. *arXiv e-prints*, Article arXiv:1909.05073 (Sep 2019), arXiv:1909.05073 pages. arXiv:cs.LG/1909.05073

[30] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J. Dally. 2017. Exploring the Regularity of Sparse Structure in Convolutional Neural Networks. *ArXiv* abs/1705.08922 (2017).

[31] Michaël Mathieu, Mikael Henaff, and Yann LeCun. 2013. Fast Training of Convolutional Networks through FFTs. *CoRR* abs/1312.5851 (2013).

[32] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2016. Pruning Convolutional Neural Networks for Resource Efficient Transfer Learning. *CoRR* abs/1611.06440 (2016). arXiv:1611.06440

[33] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. 2020. Patdnn: Achieving real-time DNN execution on mobile devices with pattern-based weight pruning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 907–922.

[34] Brilian Tafjira Nugraha, Shun-Feng Su, et al. 2017. Towards self-driving car using convolutional neural network and road lane detector. In *2017 2nd International Conference on Automation, Cognitive Science, Optics, Micro Electro-Mechanical System, and Information Technology (ICACOMIT)*. IEEE, 65–69.

[35] Hyunsun Park, Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. 2016. Zero and Data Reuse-aware Fast Convolution for Deep Neural Networks on GPU. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES '16)*. ACM, New York, NY, USA, Article 33, 10 pages. https://doi.org/10.1145/2968456.2968476

[36] Jongsoo Park, Sheng R. Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. 2016. Faster CNNs with Direct Sparse Convolutions and Guided Pruning. In *ICLR*.

[37] Kuo-You Peng, Sheng-Yu Fu, Yu-Ping Liu, and Wei-Chung Hsu. 2017. Adaptive runtime exploiting sparsity in tensor of deep learning neural network on heterogeneous systems. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE, 105–112.

[38] Valentin Radu, Kuba Kaszyk, Yuan Wen, Jack Turner, Jos'e Cano, Elliot J Crowley, Bj"orn Franke, Amos Storkey, and Michael O'Boyle. 2019. Performance Aware Convolutional Neural Network Channel Pruning for Embedded GPUs *(IISWC'19)*.

[39] Ao Ren, Tianyun Zhang, Shaokai Ye, Jiayu Li, Wenyao Xu, Xuehai Qian, Xue Lin, and Yanzhi Wang. 2019. Admm-nn: An algorithm-hardware co-design framework of dnns using alternating direction methods of multipliers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 925–938.

[40] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014).

[41] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *Computer Vision and Pattern Recognition (CVPR)*.

[42] Ben van Werkhoven, Jason Maassen, Henri Bal, and Frank Seinstra. 2013. Optimizing Convolution Operations on GPUs using Adaptive Tiling. *Future Generation Computer Systems* (09 2013). https://doi.org/10.1016/j.future.2013.09.003

[43] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann Lecun. 2014. Fast Convolutional Nets With fbfft: A GPU Performance Evaluation.

[44] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning Structured Sparsity in Deep Neural Networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. Curran Associates Inc., USA, 2082–2090.

[45] Carl Yang, Aydın Buluç, and John D Owens. 2018. Design principles for sparse matrix multiplication on the GPU. In *European Conference on Parallel Processing*. Springer, 672–687.

[46] Tianyun Zhang, Shaokai Ye, Kaiqi Zhang, Jian Tang, Wujie Wen, Makan Fardad, and Yanzhi Wang. 2018. A Systematic DNN Weight Pruning Framework using Alternating Direction Method of Multipliers. In *ECCV*.

[47] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. 2019. 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*. 359–371.