A Survey on Accelerating Sparse CNN Inference on GPUs

by

Qilin Chen
Supervisors: Hasan Mohamed, Shih-Chii Liu, Nergis Tomen

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 19, 2022

Abstract

Convolutional neural networks (CNNs) are often pruned to achieve faster training and inference speed while also requiring less memory. Nevertheless, during computation, most modern GPUs cannot take advantage of the sparsity automatically, especially on networks with unstructured sparsity. Therefore, many libraries that exploit sparsity, have been proposed for accelerating CNN inference on GPUs. However, there is little research on systematically comparing them. In this paper, some state-of-the-art libraries for accelerating sparse CNN inference on GPUs are reviewed and benchmarked. Most of the libraries speedup the convolution and/or pooling operations by skipping zero calculations, therefore, they are able to perform sparse matrix calculations faster. However, many of them have hardware and software restrictions and are hard to integrate into a new model to perform end-to-end inference.

1 Introduction

Deep learning, especially convolutional neural networks (CNNs) has a wide range of applications such as computer vision, speech recognition and bioinformatics. However, training and inference for CNNs can be time and memory consuming. One way to accelerate the training and inference is by network pruning, including both structured and unstructured pruning. Together with the use of the Rectified Linear Unit (RELU) activation function, which is frequently used in CNNs, the sparsity of a trained network can be as high as 70%. However, most modern GPUs are not designed for sparse operations, so how to make use of the sparsity for faster inference on them remains a challenge.

There have been a few recent reports on methods that exploit the sparsity of CNNs for accelerating the inference [1] [2], but there lacks a systematic summary and comparison of those methods. Therefore, the objective of this paper is to review and benchmark some of the state-of-the-art research on exploiting sparsity for CNNs on GPUs in terms of the speedup they can achieve compared to some vendor libraries from NVIDIA.

The paper is structured as follows. Section 2 provides the background information. Section 3 describes how the different methods are evaluated. Section 4 presents the experimental setup and comparison results for the different methods. Section 5 discusses the ethical issues related to this paper. Section 6 reflects on the experimental results, and presents the conclusion and possible directions for future research.

2 Background

To better understand how different methods accelerate sparse CNNs inference on GPUs, some background information regarding the hardware and CNN pruning is provided. In this section, Compute Unified Device Architecture (CUDA) is first introduced. Then, structured and unstructured pruning (also referred to as weight pruning) are explained.

2.1 CUDA

As Graphics Processing Units (GPUs) become increasingly popular for high-performance computing, CUDA was developed by NVIDIA as a toolkit for some mainstream GPUs to better support parallel computing and acceleration on GPUs. It provides many GPU-accelerated libraries, debugging and optimization tools, a C/C++ compiler, and a runtime library to deploy applications [3].

GPUs are made up of an array of streaming multiprocessors (SMs). When executing a program using CUDA on a GPU, processes are divided among threads. Each thread can access their own private registers as well as a constant cache provided by the GPU. Threads can be divided into thread

groups [4], which generally are organized in the form of a warp containing 32 threads. Multiple thread groups can be further grouped together as thread blocks. Within a thread block, there is a scratchpad memory space called shared memory with which the thread groups can communicate. Convolution calculations on GPUs are converted to matrix multiplications. The matrix multiplications are performed on each thread block, and each thread corresponds to a convolution result. The thread-based architecture provides the possibility for acceleration of calculations.

2.2 Weight Pruning

Pruning refers to the process of eliminating parameters from an existing neural network. Pruning is often applied to neural networks to cut down the computational power required. Following Han's work on model compression and pruning [5], many pruning methods have been proposed. The pruning methods can be categorized into two different types: structured pruning and unstructured pruning.

Structured pruning removes entire layers or channels of the neural network. Structured pruning speed up inference directly as the dense weight matrix becomes smaller. The neural networks become lighter due to fewer parameters after pruning therefore requiring less memory. However, structured pruning often results in large accuracy loss [6].

Unstructured pruning, on the other hand, removes connections. Unstructured pruning sets the weights of pruned parameters to zero, which is why it is also called weight pruning [7]. It is more intuitive and can be easily implemented as most modern deep learning libraries allow easy access to all the parameters. Because unstructured pruning focuses on pruning the smallest element of a neural network, it is possible to prune a network without affecting the performance. Nevertheless, most modern hardware does not natively support accelerating sparse matrix computations. Therefore, much research has been conducted on accelerating neural networks with unstructured sparsity.

3 Methods

To accelerate CNNs with unstructured sparsity, most research focuses on speeding up the calculations of the convolution and pooling layers. Many libraries have been proposed for performing convolution and pooling operations. In this section, first related work to discuss exploiting sparsity on CNNs is presented, then some state-of-the-art libraries are reviewed.

3.1 Related Work

Sparsity in CNNs. Han et al [5] proposed using pruning to eliminate unnecessary weights and prune away around 90% of the network while maintaining the accuracy. "The lottery ticket hypothesis" [9] explains how to consistently prune the network by finding subnetworks (winning tickets) that have test accuracy comparable to the original network. Gale et al [8] evaluated some common techniques and provides an overview of each technique.

Sparse CNNs accelerators. Accelerators for sparse CNNs have been developed to perform inference. ASpT [10] proposed an adaptive tiling strategy to perform sparse matrix-matrix multiplication (SpMM) and uses the standard Compressed Sparse Row (CSR) representation. Escort [11] computes the sparse convolutions directly instead of lowering them onto matrix multiplication and introduces some optimization techniques and eventually achieves 1.43x speedup compared to cuBLAS. Some more accelerators used for this review will be explained in more detail in the next section.

3.2 Libraries for Exploiting Sparsity on CNNs

The following state-of-the-art libraries are reviewed and the methods used in these libraries are summarized: *TensorRT* [12], *SparseRT* [1], *Sputnik* [2], *Conv_ Pool_ Algorithm* [13] and *MinkowskiEngine* [14]. They are all developed for NVIDIA GPUs and depend on CUDA.

TensorRT: As the library provided by NVIDIA for accelerating deep learning training and inference, TensorRT is widely used and supported by multiple deep learning libraries such as PyTorch and TensorFlow. The recent TensorRT update (v8.0) added support for accelerating sparse CNN inference on Ampere architecture GPUs with 2:4 fine-grained, unstructured sparsity. As shown in Figure 1, the white squares are zeros and the green ones are non-zeros, in every 4 blocks of the row, 2 of them are zeros. *TensorRT* stores the sparse matrix in a compressed format with only the non-zero values and their indices. Calculations are only applied to the non-zero values in the compressed matrix. A small speedup is expected using *TensorRT* because can accelerate different types of operations during inference, but for accelerating unstructured weight sparsity only the 50% sparsity is supported.

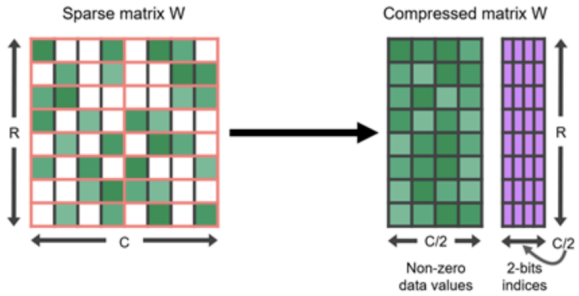


Figure 1: Structured sparse matrix W, and its compressed representation. [15]

SparseRT: This framework speeds up the SpMM and sparse convolution by coming up with a new tiling strategy and load balancing strategy. An example is to calculate the matrix multiplication $A \times B = C$ where matrix A has dimension $M \times K$, B has dimension $K \times N$ and C has dimension $M \times N$. The SpMM problem is first treated as a dense matrix-matrix multiplication (GeMM) and the calculation is distributed to thread blocks as shown in Figure 2. The grid of thread blocks has size (8, 16), each thread block produces a part of C. Within a thread block, first a portion of B is loaded in a thread group, then A is iterated through and the multiplication result is accumulated. In the tiling for GeMM, each thread block gets the same-sized portion of the A matrix. However, if A is sparse, the amount of non-zero values in each portion differs. So to extend the tiling strategy, a load balancing among thread blocks is needed. As illustrated in Figure 3, there are two different methods. Keep the number of M elements constant while changing K elements for different thread blocks, or the other way around. Both the methods try to balance the non-zero values in the thread blocks. After this, load balancing within thread groups is performed. Different thread groups are assigned different numbers of elements so that they contain the same number of non-zeros, as demonstrated in Figure 4. When running inference, the sparse matrix is already known, so the tiling and load balancing can be performed at compile time and used later as a part of the code. This shortens the time for matrix calculation and the acceleration should be higher as the sparsity increases.

a

	N[0:32]	…	N[480:512]
M[0:32]	Block (0,0)	…	Block (0,16)
…			
M[224:256]	Block (8,0)		Block (8,16)

b

Block 0 Thread Group 0	
Step 0	Load B[0:0:32]
Step 1 (K0 M0)	C[0,0:32] += A[0,0] × B[0,0:32]
Step 2 (K0 M1)	C[1,0:32] += A[1,0] × B[0,0:32]
…	…
Step 32 (K0 M31)	C[31,0:32] += A[31,0] × B[0,0:32]

c

Thread Group 0	Thread Group 1	…	Thread Group 15
K0 M0	K1 M0	…	K15 M0
…	…	…	…
K0 M31	K1 M 31	…	K15 M31
…	…	…	…
K3040 M0	K3041 M0	…	K3055 M0
…	…	…	…
K3040 M31	K3041 M31	…	K3055 M31

Figure 2: Tiling strategy for GeMM ($M = 512$, $K = 3056$, $N = 256$). a) Each block in the 2-D grid of blocks processes a rectangular tile of the output. b) Thread group computation for one B element. c) Tiling strategy at granularity of K and M. Each thread group processes a range of M indices for each K index [1].

a

K0	M[0:32]	M[32:64]	…	M[480:512]
K1	M[0:32]	M[32:64]	…	M[480:512]
…	…	…	…	…
K3055	M[0:32]	M[32:64]	…	M[480:512]

b

K0	M[0:36]	M[36:63]	…	M[485:512]
K1	M[0:36]	M[36:63]	…	M[485:512]
…	…	…	…	…
K3055	M[0:36]	M[36:63]	…	M[485:512]

Figure 3: Thread block level load balancing strategies for SpMM ($M = 512$, $K = 3056$, $N = 256$). Different colors denote assignment to different thread blocks a) Different thread blocks can have different portions of the reduction axis. b) Different thread blocks can have different portions of the external axis [1].

a

Thread Group 0	Thread Group 1	…	Thread Group 15
K0 M0	K1 M0	…	K15 M0
…	…	…	…
K0 M31	K1 M 31	…	K15 M31
…	…	…	…
K3040 M0	K3041 M0	…	K3055 M0
…	…	…	…
K3040 M31	K3041 M31	…	K3055 M31

b

Thread Group 0	Thread Group 1	…	Thread Group 15
K0 M0	K1 M0	…	K15 M0
K0 M15	…	…	…
…	…	…	K3054 M11
K200 M25	K380 M27	…	
K200 M31	K380 M29	…	

Figure 4: Thread group level load balancing strategies for SpMM ($M = 512$,$K = 3056$,$N = 256$). Orange cells denote example nonzero locations. a) The computation that needs to be performed across the different thread groups. b) How the computation is assigned to each thread group [1].

Sputnik: Similar to *SparseRT*, Sputnik achieves acceleration through tiling and load balancing, but most of the methods proposed by *Sputnik* aim to solve the shared memory load bottleneck. Figure 6(a) shows the 1-dimensional tiling scheme, which split the computation across different elements of the GPU architecture. Take the same example of matrix multiplication $A \times B = C$, A is stored in CSR format, marked in green. B is dense, marked in blue. The output matrix C is dense, marked in orange. From left to right the structure for calculation becomes smaller. The first level shows the entire matrix calculation. Then, the second level shows in thread block tile, one block stores all the values of A, while for B the thread block only loads a contiguous vector. For thread groups (warps), the calculation is similar. At the last level of thread, one vector from B is stored and calculated with

the entire thread block containing A, and eventually the results are accumulated. Rows in a sparse matrix can have arbitrary lengths, so using vector memory instructions as shown in 1-dimensional tiling would result in an increased number of values loaded simultaneously by a thread block and no alignment guarantees. Instead of loading one row of the matrix in a warp, subwarp tiling, shown in Figure **??**(a), allows mapping of subsets of a warp. The warp accesses are split across multiple rows and therefore reducing the amount of wasted work. Reverse offset memory alignment, shown in Figure 6(b), pads each row with zeros so the values in each row are multiples of four. The address of each row is decremented to the nearest aligned address. Values from the previous row are masked in the first loop iteration to maintain correctness. Another method, row swizzle load balancing, is proposed to balance the workload across the processing elements. This method remaps where work is scheduled so each processing element has roughly the same amount of work. Row swizzle load balancing is applied to two levels of processing elements, warps and thread blocks, as presented in Figure 7. For threads within a warp, the rows of similar lengths are grouped into bundles. For SMs, row bundles are processed in decreasing order of size. Combining all the methods, SpMM can be accelerated. But as the library goes through many steps for rearranging the storage of the matrices in memory, it may not be optimal for low sparsity models.

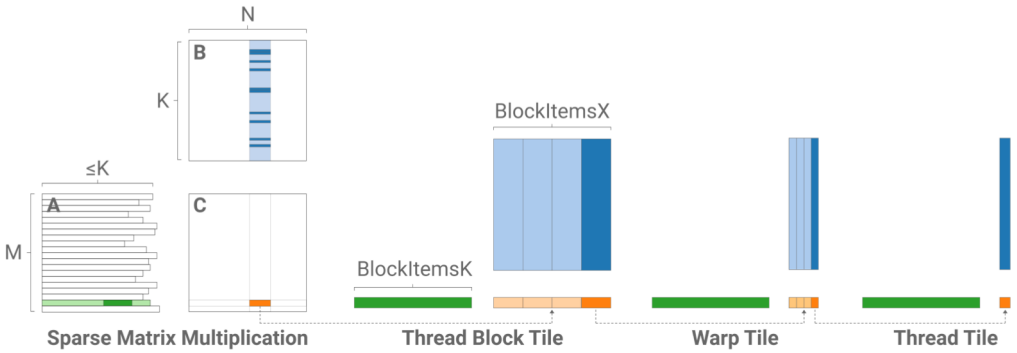
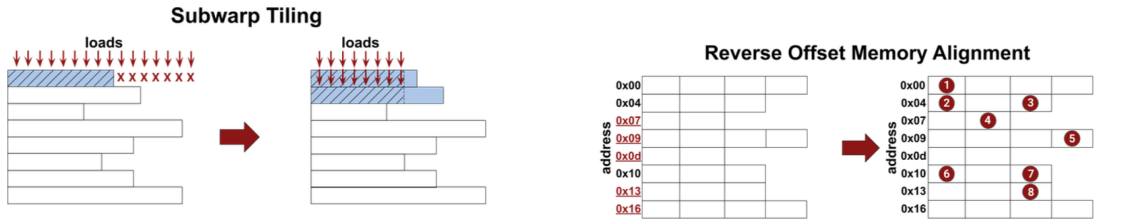


Figure 5: Hierarchical decomposition of SpMM with 1-dimensional tiling [2].



(a) Subwarp tiling maps subsets of a warp to independent 1-dimensional tiles of the output.

(b) Reverse offset memory alignment backs up the address of each row to the nearest aligned address

Figure 6: Subwarp tiling and reverse offset memory alignment [2].

Conv_Pool_Algorithm: The framework considers the sparsity of the feature map and proposes using two novel storage formats: Extended Compressed Rows (ECR) and Pooling-pack Extended Compressed Rows (PECR). ECR calculates the convolution by skipping zero values and complete extension, compression and sparse matrix calculation by only accessing the global memory once. As shown in Figure 8(a), B1 is a thread block containing 3 threads T1, T2 and T3. The kernel size for convolution is 3×3. Only the non-zero values in the feature map B1 are stored in F_data and their corresponding kernel values are stored in K_data. Ptr indicates the number of non-zero calculation that needs to

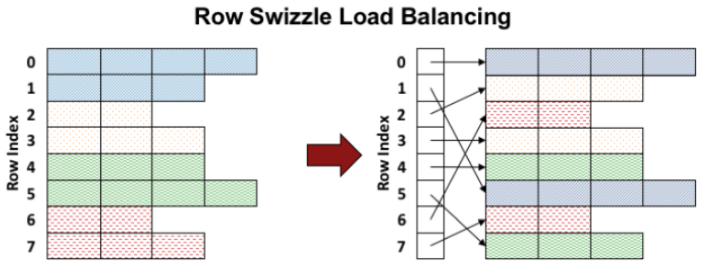
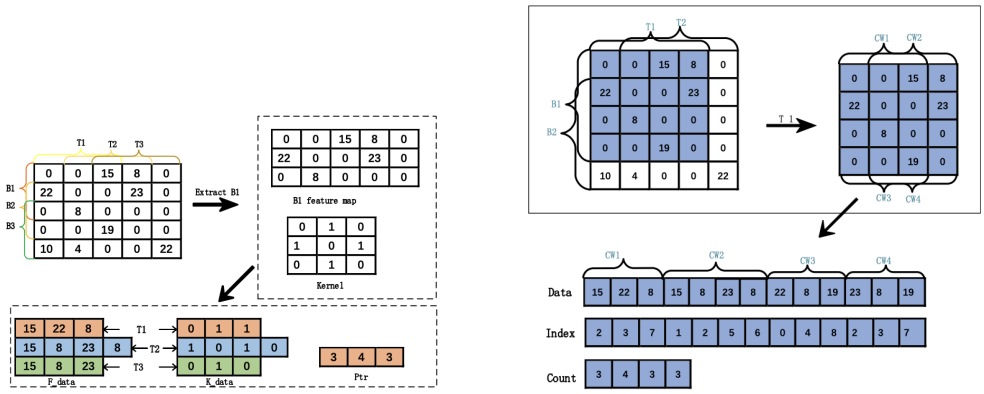


Figure 7: Row swizzle approach for load balancing sparse matrix computation [2].



(a) ECR storage format

(b) PECR storage format

Figure 8: The two storage formats proposed in [13].

be performed. In Figure 8(b), the PECR is shown, the pooling operation has a sliding window of size 2×2. Similar to ECR, the non-zero values are stored in *Data* with their corresponding indices stored in *Index*. *Count* indicates the non-zero operations that need to be performed. Figure 9 shows how a complete calculation with convolution and pooling is completed. Combining the convolution operation and pooling operation, the framework is able to reduce the calculation time and traffic between CPU and GPU, therefore achieving a relatively high speedup for sparse CNNs. Using the two storage format proposed, both the runtime and memory usage can be improved.

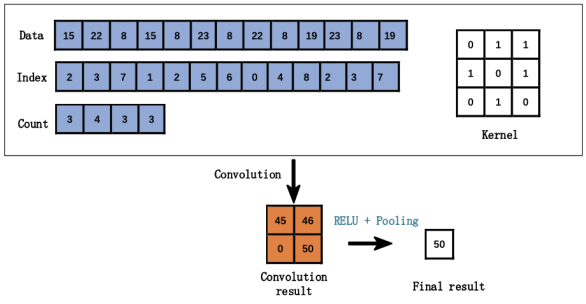


Figure 9: Convolution and pooling with PECR [13]

MinkowskiEngine: *MinkowskiEngine* is an auto-differentiation library for sparse tensors and gen-

eralized sparse convolution. The most important functions it provides are sparse tensor quantization, generalized sparse convolution, max pooling, and global/average/sum pooling. The *MinkowskiEngine* first generates a sparse tensor given a sparse CNN and converts input into unique coordinates, associated features and optional labels. Then the output coordinates are generated based on the input coordinates. Combing the input and output coordinates a kernel map is generated and stored as pairs of list of input and output indices. The convolution can be computed by iterating through the list of input and output indices. For pooling, first the number of inputs per output coordinate and their indices are found, then depending on the type of pooling the corresponding CUDA library function is called.

4 Results

Experiments are carried out on the different libraries to gather information about their performance and potential shortcomings. In this section, first the experiment setup is introduced. Then the performance of the different libraries is presented. The main metrics used to measure their performance are time and speedup compared to either not using the libraries or using vendor libraries from NVIDIA.

4.1 Experiment Setup

To compare all the libraries, a desktop GPU, GeForce RTX 3080 was chosen. The GPU has 10GB of memory and 272 Tensor Cores. It was selected as it is a common GPU used in modern personal computers. Furthermore, it has the Ampere architecture, on which TensorRT supports accelerating sparse CNN inference. The operating system is Ubuntu 18.04, CUDA toolkit v11.4, cuDNN v8.2 and Python 3.6.9 are installed on the GPU.

This paper focuses on the speedup of the reviewed libraries on CNN inference, thus the only metric used is time, the accuracy of the libraries can also be found in the original papers. As not all libraries can be easily integrated, and the most important operations when performing inference are convolution and pooling, some libraries are benchmarked without the entire CNN model but only the feature map and filter from one convolution layer.

Due to the incompatibility of different libraries, the libraries are divided into groups and benchmarked using different setups.

- *TensorRT* is benchmarked using a model developed based on ResNet-18 [16]. The inference speed under three settings is measured: before pruning and without *TensorRT*, before pruning and with *TensorRT*, and after pruning with *TensorRT*.
- *Sputnik* and *SparseRT* are similar as they try to accelerate inference by skipping the zero calculations. Their implementation focus on the convolution operations and other linear algebra calculations. Thus they are benchmarked on SpMM given different matrix dimensions and different sparsity. The matrices are randomly generated, it has already been proven that the sparsity pattern does not have a significant impact on both the libraries [1] [2].
- *Conv_Pool_Algorithm* calculates the operations for an entire convolutional layer or combined convolution layer and pooling layer, therefore it is benchmarked individually against a vendor library cuDNN using VGG-19 from [13] and the dataset from ImageNet [19].

MinkowskiEngine is tested but not used for benchmarking because it requires implementing and pruning the CNN from scratch and cannot be used with pretrained models, which makes it difficult to compare the performance under the same conditions. Theoretically, it should be able to accelerate inference because it generates the kernel map after scanning through the matrix and stores the coordinates for the non-zero values and skips the zero calculations, similar to the *Conv_Pool_Algorithm*.

4.2 Evaluation

The evaluation is divided into three parts as mentioned above. For *TensorRT*, the most important performance metrics are the speedup and accuracy as it can be used to perform inference. The effect of sparsity on inference speed is not analyzed in *TensorRT* as it only supports 2:4 fine-grained sparsity, which is 50%. For the other libraries, the performance for linear algebra calculations is measured. The metrics used is time and speedup compared to vendor libraries and the influences of matrix size and sparsity are analyzed.

4.2.1 TensorRT

Table 1 shows the result for running *TensorRT* on the model TrafficCamNet [17] based on ResNet18. As shown in the table, the model is pruned to only $\frac{1}{8}$ of its original size, while retaining the accuracy of the original unpruned model. When TensorRT is used on the unpruned model, the inference is 1.15 times faster than without it. The speedup is mostly due to the smaller model size and fewer connections in the model. On the pruned model, the speedup is higher, achieving 1.21 times faster. To get more insight into how TensorRT is able to accelerate the inference, the runtime for each operation in the CNN was analyzed. It can be noticed that convolutions are performed in different thread blocks, and only some of the thread blocks are able to achieve around 2 times speedup using *TensorRT*. This is because *TensorRT* uses a 2:4 fine-grained structured sparsity, which means in each contiguous block of four values, two values must be zero. For blocks that satisfy the 2:4 fine-grained structured sparsity, *TensorRT* stores only the non-zero values and compresses the matrix to be $\frac{1}{2}$ of its original size. When using *TensorRT* on the unpruned model, there is a small speedup because *TensorRT* can not only accelerate the convolution but also provides other functionality such as layer and tensor fusion. The final speed up is only around 1.21x compared to the unpruned model without using *TensorRT*, this is because even though the convolution can be computed around 2x faster, it is not the only type of operation used when performing inference. Other operations such as pooling and data transfer are also included in the execution time.

Condition	Unpruned w/o TensorRT	Unpruned w/ TensorRT	Pruned w/ TensorRT
Model size	44.32MB	44.32MB	5.2MB
Speedup	1x	1.15x	1.21x
Accuracy	84%	84%	84%

Table 1: Performance of TrafficCamNet running inference on object detection on KITTI [20] in different settings.

4.2.2 SparseRT and Sputnik

Figure 10 shows the speed of calculating SpMM using *SparseRT* and *Sputnik* on matrices of different shapes, the filter matrices all have 90% sparsity. The 90% sparsity was chosen because many frequently used CNNs such as MobileNet [18] and ResNet can be pruned to 90% sparse without loss in accuracy. For a multiplication $A \times B = C$, where matrix A (the filter matrix) has dimension $M \times K$, B has dimensions $K \times N$ and C has dimensions $M \times N$, the matrix shape is defined as (M, K, N) and shown in the x-axis. The computation time is in nanoseconds and obtained through running the SpMM for different matrices with the same size repeatedly and averaging the results. To show the speedup of the libraries, a vendor library cuBLAS is also used to perform the same computation. Both the libraries are able to accelerate SpMM regardless of the dimensions of the matrices. On average *SparseRT* can achieve 1.86x speedup and *Sputnik* can achieve 1.78x speedup compared to cuBLAS.

The graph shows that *SparseRT* and *Sputnik* have comparable performance for all the calculations. For smaller filter matrices *SparseRT* has better performance while for larger ones its performance degrades. This is because *SparseRT* uses an aggressive unrolling strategy to store the sparse matrix information in the source code to minimize cache contention while *Sputnik* stores the matrices in shared memory. For *SparseRT*, when a thread computes a matrix multiplication, one matrix is fetched from the shared instruction and the constant cache, and the other matrix is fetched from the on-chip data cache. Therefore, the performance of *SparseRT* is limited by the instruction fetch latency. For large matrices, the constant cache will be used up and some compile time constants cannot be stored but instead are compiled to immediate constants in the instructions.

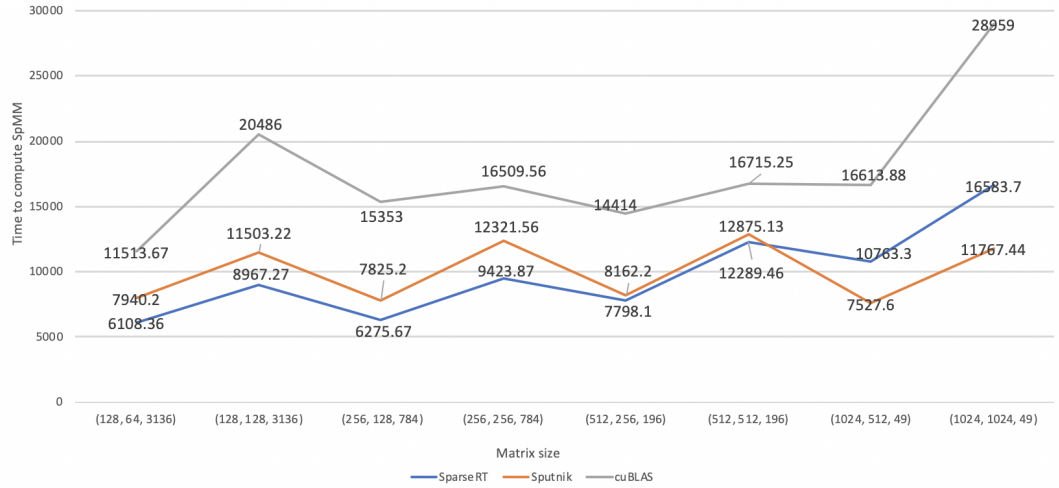


Figure 10: The time to calculate SpMM for different size matrices with 90% sparsity using SparseRT, Sputnik and cuBLAS.

To understand how sparsity affects the performance of SpMM, both libraries are benchmarked using matrix dimension of (128, 128, 3136) with sparsity levels of 50%, 75%, 80%, 85%, 90% and 95% for the filter matrix. The runtime for different matrices is presented in Figure 11. The runtime for cuBLAS is used as the baseline, it is constant because SpMM is treated the same as GeMM using cuBLAS. Both the libraries perform better when the sparsity is higher, and as the sparsity becomes higher, the speed increase also becomes larger. For *SparseRT*, all the SpMM with above 50% sparsity can be accelerated with around 2x speedup, while *Sputnik* can only accelerate matrices with above around 60% sparsity level. The result for *Sputnik* is in line with the original paper, but it was not explained in the paper why it would only work with higher sparsity matrices. Potential reasons could be that the subwarp tiling and reverse offset memory alignment proposed by the paper cannot reduce the workload when the sparsity is low but instead adds to it during padding, or the row swizzle load balancing does not lead to a significant improvement in the runtime because the rows do not differ in length too much.

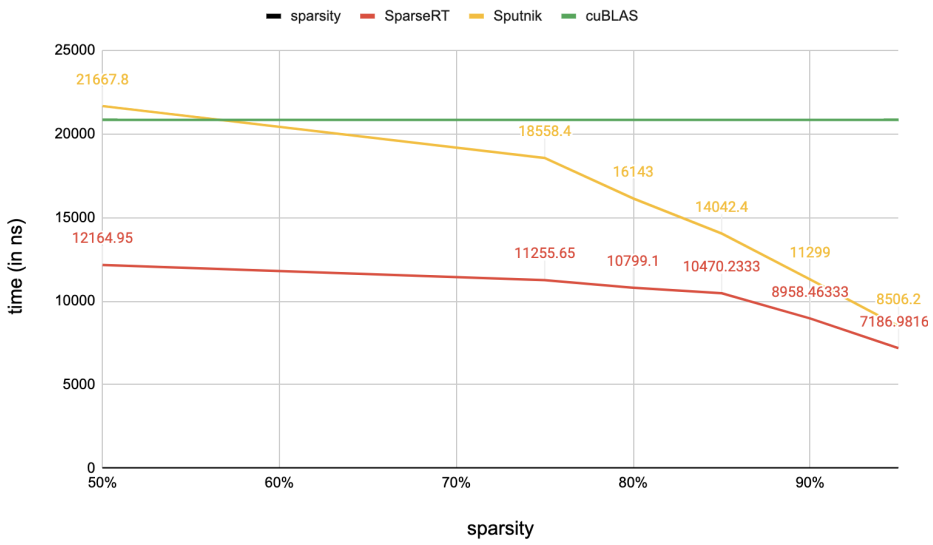
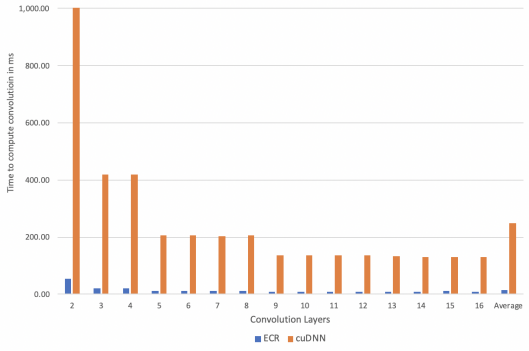


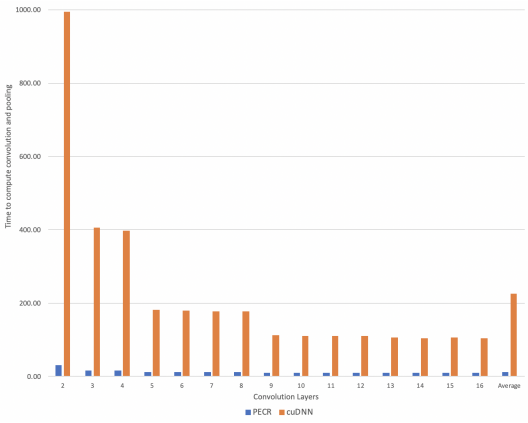
Figure 11: The time to calculate SpMM for matrices of dimension (128, 128, 3136) with different sparsity using SparseRT, Sputnik and cuBLAS.

4.2.3 Conv_Pool_Algorithm

Conv_Pool_Algorithm is benchmarked using VGG-19, the performance using this library is compared with using the vendor library cuDNN. Figure 12(a) shows the time (ms) to perform the calculation of each convolution layer and the speedup using ECR compared to cuDNN, Figure 12(b) shows the time (ms) to perform the calculation of each convolution layer together withing pooling and the speedup using PECR compared to cuDNN. The first convolution layer is disregarded as at the beginning of the calculation cuDNN uses an exhaustive search to find a suitable algorithm for the calculations. It is clearly shown that both the ECR and PECR proposed can achieve a high speedup compared to cuDNN, with an average speedup of 16.63x and 17.61x respectively.



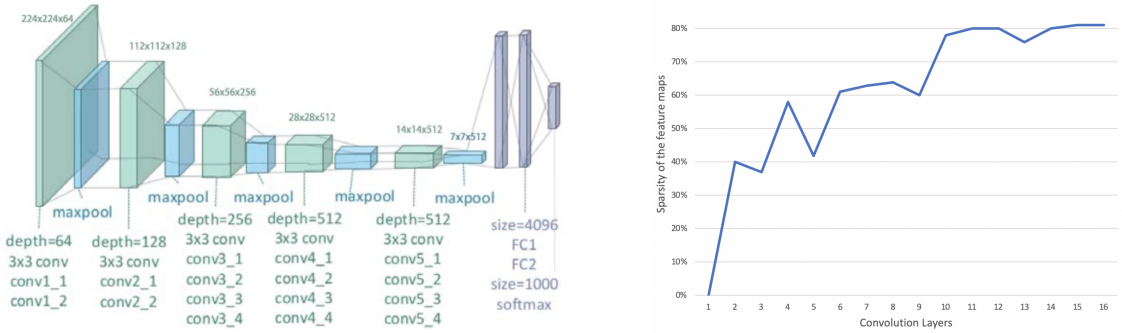
(a) The time to calculate one convolution layer using ECR compared to cuDNN.



(b) The time to calculate one convolution layer together with pooling using PECR compared to cuDNN.

Figure 12: The computation time using ECR and PECR compared to cuDNN.

Figure 13 shows the network architecture of VGG-19, conv1_1 corresponds to conv_1, conv1_2 corresponds to conv_2, conv2_1 corresponds to conv_3 and so on. Figure 13(b) shows the sparsity of the feature map of each convolution layer in VGG-19. As the convolution layer goes deeper, the size of the feature map becomes smaller and the sparsity becomes higher. Combining with Figure 12, it is clear that the computation speed is greatly affected by the size of the feature map, the larger feature maps can achieve higher speedups. Therefore, to get a better understanding of how sparsity affects the computation speed, the convolution layers with the same sizes are grouped together. As illustrated in Figure 14, each group is separated by a red line, the blue line indicates the sparsity of the layer, the grey line indicates the speedup of PECR in this layer and the orange line indicates the speedup of ECR in this layer. The speed up for each group is relatively stable no matter how the sparsity changes. One possible explanation could be that the time difference is due to how the matrix is stored and accessed in GPU. When using cuDNN to perform the convolution, it reads in the feature map and allocates space to store the entire matrix. Later, it iterates through the entire matrix again for the computation. But when using ECR or PECR, the feature map is only iterated through once and only the non-zero values are stored. In later computations, only the non-zero values are computed. Another reason could be that the sparsity difference is not enough to result in a noticeable change in the computation time.



(a) The network architecture of VGG-19 [21].

(b) The sparsity of the feature map in each convolution layer in VGG-19 [13].

Figure 13: The architecture and sparsity of feature map for each convolution layer of VGG-19.

It can also be found that for larger feature maps, PECR can achieve a higher speedup than just using ECR, this is because PECR stores the result of convolution in GPU and computes the pooling and convolution together, which reduces the data transfer time between GPU and CPU. While for smaller feature maps, the data transfer time is short enough and PECR does not have an advantage. As the algorithm tries to load the entire matrix and stores the intermediate result on GPU for further computation rather than constantly transferring partial data between GPU and CPU, one problem that appeared during the experiment is that this library has a memory constraint and cannot be used for calculating large matrices on the GPU used. This can be solved by reducing the model size or using a GPU with more memory, however, the memory issue still means that this library is not suitable for large models or industrial-level usage.

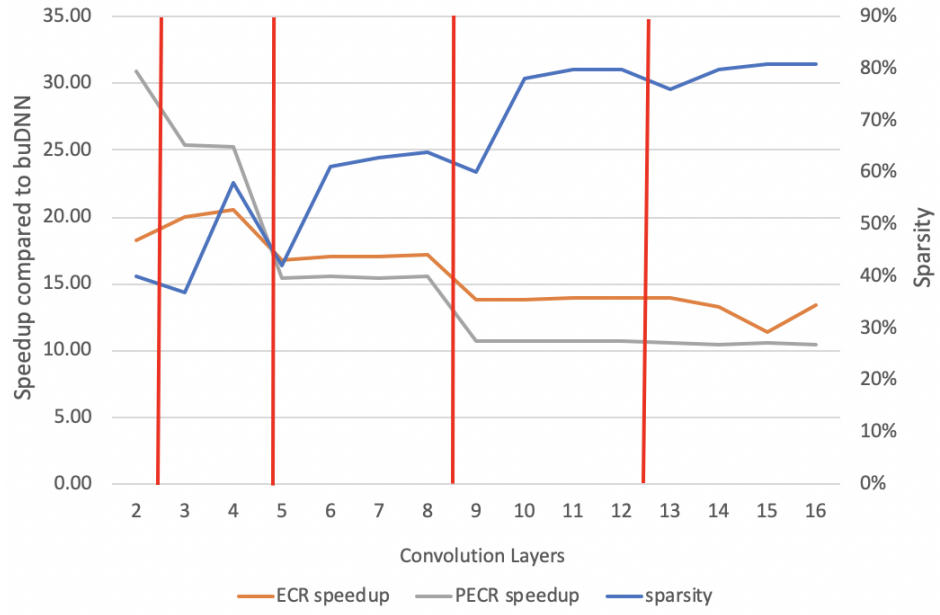


Figure 14: Sparsity (in %) and speedup using ECR and PECR for each convolution layer in VGG-19.

5 Responsible Research

In this section, some ethical issues related to the paper are discussed. An analysis of the reproducibility of the work described in this paper is performed.

5.1 Ethical Implications

This paper focuses on accelerating CNN inference on existing models using existing datasets. The only relevant result is the inference speed, therefore, the work in this paper does not involve any potential ethical issues. However, deep learning and convolutional neural networks have a wide range of applications, the most popular ones being computer vision and natural language processing. Ethical implications arise from data collection, to the use of predicted outcomes and the bias in Artificial Intelligence (AI). There are many examples, such as the AI developed by Facebook putting the label "Primates" on a video of black men in 2021, or a self-driving car such as Tesla fails to predict the potential danger on the road and causes a car accident. Possible solutions towards these problems include removing the bias in the training data and setting up strict regulations about how data should be used to protect data privacy and ensure data security. Nevertheless, many issues remain unresolved and the predictions never be 100% trusted. Humans should always have the ultimate control over the actions of AI systems.

5.2 Reproducibility of Results

The experiments conducted can be reproduced easily as the hardware and software requirements are listed in Section 4.1. The libraries used are all open source and can be found on GitHub with installation guides. The models and datasets used for the experiment are all cited and can be easily downloaded. Additionally, a GitHub repository has been set up with instructions on installation and solutions to potential problems: https://github.com/Darcy-Chen/rp2022. To change the parameters or input used for testing, follow the instructions in the README on GitHub. To reproduce the experiment, basic knowledge of CUDA and Python is required.

6 Conclusion and Discussion

In this paper, a survey on the state-of-the-art libraries for accelerating sparse CNNs on GPUs has been performed. The libraries are summarized and evaluated under different settings. *TensorRT*, *Conv_Pool_Algorhithm* and *MinkowskiEngine* all proposed a way to compress the matrices and only stores non-zero values. *TensorRT* only supports the 2:4 fine-grained sparsity structure (50% sparsity) and accelerates unstructured weight sparsity while *Conv_Pool_Algorithm* and *MinkowskiEngine* consider the sparsity in the input matrix. *SparseRT* and *Sputnik* both aims to accelerate weight sparsity by using tiling and load balancing. *SparseRT* tries to balance the non-zero values across thread groups at compile time and use them as a part of the code, while *Sputnik* explores different ways to improve vector memory access.

Most of the libraries show a relatively high speedup. *TensorRT* achieved 1.21x speedup for inference using a pruned model with no loss in accuracy. *SparseRT* and *Sputnik* achieved 1.86x and 1.78x speed compared to cuBLAS respectively when computing SpMM. For both libraries, the speedup is higher when the sparsity is higher. *Conv_Pool_Algorithm* achieved around 17x speedup for convolution and pooling operations compared to cuDNN, and the speedup is mostly affected by the size of the matrix.

The work presented has some limitations. For *Sputnik*, the result shows that it only achieves a high speedup when the matrix is highly sparse. To uncover the reason behind this, more experiments are needed to analyze which step of the SpMM calculation is slowing down the performance. For *Conv_Pool_Algorithm*, sparsity shows no effect on the speedup. Conducting more experiments using matrices of the same dimension but different sparsity could provide more information on this observation. However, due to the low compatibility of the libraries, it is very difficult to adapt them to support input of different formats. Another challenge of using these libraries is that some of them are designed to perform linear algebra operations for the sparse matrices such as matrix multiplication and pooling. This adds difficulties to using the libraries on end-to-end models as the supported input format is not standard PyTorch or Tensorflow models. To further utilize the libraries and benchmark them using the same model, a program to convert the models and their input to the format each library supports needs to be developed.

As shown in the result, some accelerators perform better when the sparsity of the model is higher, the work by Pietron et al [22] studies when CNN models with unstructured sparsity can be accelerated. Aside from using GPUs for CNN inference, there is also much research done on accelerating CNN inference on FPGAs and CPUs [23] [24]. As GPUs are more commonly used in desktop computers and data centers, these research might provide further insight into the deployment of CNNs on embedded systems and edge devices.

Acknowledgement

The author thanks Pijus Krisiukenas for providing the RTX 3080 for the experiment and Zuowen Wang for help with hardware setup and suggestions. This research is supported by UZH/ETH Institute of Neuroinformatics.

References

[1] Z. Wang, "SparseRT: Accelerating Unstructured Sparsity on GPUs for Deep Learning Inference", arXiv:2008.11849, 2020.

[2] T. Gale, M. Zaharia, C. Young, E. Elsen, "Sparse GPU Kernels for Deep Learning", arXiv preprint arXiv:1902.10901, 2020.

[3] "CUDA Toolkit", NVIDIA Developer. [Online] Available: https://developer.NVIDIA.com/cuda-toolkit

[4] M. Harris, K. Perelygin, "Cooperative groups: Flexible CUDA thread programming", 2017.

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

[6] E. Crowley, J. Turner, A. Storkey, O. Michael, "A closer look at structured pruning for neural network compression", arXiv preprint arXiv:1810.04622v3, 2019.

[7] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, en A. Peste, "Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks", arXiv:2102.00554, 2021.

[8] T. Gale, E. Elsen, S. Hooker, "The State of Sparsity in Deep Neural Networks", arXiv:1902.09574, 2019.

[9] J. Frankle, M. Carbin, "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks" arXiv:1803.03635, 2018.

[10] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication", In Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, ACM, 2019, pp. 300-314.

[11] X. Chen, "Escoin: Efficient Sparse Convolutional Neural Network Inference on GPUs", arXiv:1802.10280, 2019.

[12] "NVIDIA TENSORRT," NVIDIA Developer. [Online]. Available: https://developer.NVIDIA.com/tensorrt.

[13] W. Xu, S. Fan, H. Yu, X. Fu, "Accelerating convolutional neural networks by exploiting sparsity on GPUs", arXiv preprint arXiv:1909.09927, 2019.

[14] C. Choy, J. Gwak and S. Savarese, "4D Spatio-Temporal ConvNets: Minkowski Convolutional Neural Networks," 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2019, pp. 3070-3079, doi: 10.1109/CVPR.2019.00319.

[15] "Accelerating Inference with Sparsity Using the NVIDIA Ampere Architecture and NVIDIA TensorRT". [Online]. Available: https://developer.NVIDIA.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt/

[16] K. He, X. Zhang, S. Ren, J. Sun, "Deep Residual Learning for Image Recognition", arXiv preprint arXiv:1512.03385, 2015.

[17] "TrafficCamNet", [Online]. Available: https://catalog.ngc.NVIDIA.com/orgs/NVIDIA/models/tlt_trafficcamnet

[18] A. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications", arXiv preprint arXiv:1704.04861, 2017.

[19] J. Deng, W. Dong, R. Socher, L. -J. Li, Kai Li and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," 2009 IEEE Conference on Computer Vision and Pattern Recognition, 2009, pp. 248-255, doi: 10.1109/CVPR.2009.5206848.

[20] A. Geiger, P. Lenz, R. Urtasun, "Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite", 2012 Conference on Computer Vision and Pattern Recognition (CVPR), 2012.

[21] Y. Zheng, c. Yang, A. Merkulov, "Breast cancer screening using convolutional neural network and follow-up digital mammography", Computational Imaging III, 2018, pp.4, doi: 10.1117/12.2304564.

[22] M. Pietron, D. Zurek, "When deep learning models on GPU can be accelerated by taking advantage of unstructured sparsity", arXiv:2011.06295, 2020.

[23] K. Abdelouahab, M. Pelcat, J. Serot, F. Berry, "Accelerating CNN inference on FPGAs: A Survey", arXiv preprint arXiv:1806.01683, 2018.

[24] S. Singh, D. Alistarh, "WoodFisher: Efficient Second-Order Approximation for Neural Network Compression", arXiv preprint arXiv:2004.14340, 2020.