Accelerating convolutional neural network by exploiting sparsity on GPUs

Weizhi Xu, Shengyu Fan, Hui Yu and Xin Fu, *Member, IEEE,*

Abstract—**Convolutional neural network (CNN) is an important deep learning method. The convolution operation takes a large proportion of the total execution time for CNN. Feature maps for convolution operation are usually sparse. Multiplications and additions for zero values in the feature map are useless for convolution result. In addition, convolution layer and pooling layer are computed separately in traditional methods, which leads to frequent data transfer between CPU and GPU. Based on these observations, we propose two new methods to accelerate CNN on GPUs. The first method focuses on accelerating convolution operation, and reducing calculation of zero values. The second method combines the operations of one convolution layer with the following pooling layer to effectively reduce traffic between CPU and GPU. For the first method, we extract some convolution layers from LeNet, AlexNet and GoogLeNet, and can achieve up to 3.6X speedup over cuDNN for the single-layer convolution on GPU. Experiment on VGG-19 achieves 3.5X speedup over cuDNN for convolution operation on average. For the second method, experiment on VGG-19 achieves 4.3X speedup over cuDNN on average.**

Index Terms—**Convolutional neural network, GPU, Performance tuning, Shared memory, SpMV**

I. INTRODUCTION

CONVOLUTIONAL neural network (CNN) plays a crucial role in deep learning, which facilitates the development of computer vision [1], medical image classification [2], natural language processing [3], recommender system [4] etc. CNN is also an essential part of many complex neural network models [5], [6]. However, it is time-consuming for training and inference of CNN. Improving the execution speed of CNN is of great importance for accelerating its applications. Convolution operations usually take up a large part of the total execution time of CNN, even more than 70% in the well-known convolution neural networks such as VGG, ResNet and YOLO [7], [44]. From the hardware perspective, designing an architecture suitable for convolution computation to achieve acceleration is a feasible strategy [8], [9], e.g. Tensor Cores [10], [11]. From the software perspective, a

Weizhi Xu, Shengyu Fan and Hui Yu are with School of Information Scinece and Engineering, Shandong Normal University, Jinan, China. Xin Fu is with Electrical and Computer Engineering Department, University of Houston, Houston, USA. Weizhi Xu and Hui Yu are also with Electrical and Computer Engineering Department, University of Houston, Houston, USA.

series of algorithmic optimization techniques were developed, such as im2col-based method [12], [13], FFT-based method [14], and Winograd-based method [15]. The above acceleration methods have been integrated into cuDNN library, which is a state-of-the-art library for deep learning on GPU [16].

The convolution operation in CNN refers to the process in which the convolution kernel samples on the feature map. In the sampling process, the convolution kernel carries out a weighted summation operation on the sampling area, and the entire feature map is sampled according to the stride size. The process of convolution contains a lot of multiplications and additions, so reducing these operations are effective for acceleration. Network pruning and RELU activation are common operations in deep neural networks, which result in a large number of zero values in the network. For the feature map that needs to be calculated for the convolution operation, the ratio of zero values can achieve more than 0.7 in the deep layers of the network after multiple epochs. The pursuit of precision in deep neural networks leads to dozens of iterations, so the large sparsity in the feature map is inevitable. The calculation of these zero values, however, is useless for the convolution result [17]. In other words, if we can skip the zero-value calculation in the convolution operation, this will enormously reduce operations of multiplication and addition.

Some efforts have focused on reducing the calculations of zero values in neural networks, such as SqueezeFlow [18], SCNN [19]. A new FPGA hardware architecture was proposed to use sparsity to reduce memory and computational requirements [20]. Besides, a quantized approach is proposed to accelerate and compress convolutional networks on mobile devices. [33] Above works obtain outstanding acceleration effects and provide new development directions for deep learning acceleration. However, new architectures based on FPGA or ASIC are not flexible and have a long development cycle, compared with CPU or GPU platform. Some algorithms exploiting sparsity for convolution operations on CPU are proposed and can achieve considerable speedup over their dense counterparts [17], [21]–[23]. GPU is widely used for accelerating CNN because of its strong computing power based on highly-optimized cuBLAS [24], [25]. Current implementations of sparsity optimization on GPU are usually based on sparse libraries such as cuSPARSE [26]–[28]. For the convolution layer, the feature maps and filters can be processed with three steps, extension, compression and sparse matrix computation by cuSPARSE. However, exploiting sparsity in convolution operation can hardly achieve satisfactory performance when CNN is implemented on GPU [29]. Very limited speedup over cuBLAS-based convolution methods is obtained [11],

[30]. The reasons are as follows. 1) The above three steps are separately completed on GPU, so the data are loaded from and stored into global memory at least three times. When using cuSPARSE, the data are even needed to be transferred between CPU and GPU. 2) The cost of real-time compression affects the performance. 3) Sparse matrix computation is less efficient than its dense counterparts on GPUs [31].

On the other hand, the pooling operation starts after the convolution results are transferred from GPU to CPU in traditional CNN implementations. This will increase the traffic between CPU and GPU. To address this problem, some works [41], [42] integrate convolution and pooling into one GPU kernel to decrease communication overhead. However, the sparsity of convolution operation is not considered in these works when fusing the two operations. In this paper, we propose two novel methods to accelerate CNN on GPUs. Our contributions can be summarized as follows.

- We propose a novel storage format (ECR) for sparse feature maps. We design a convolution algorithm based on ECR format, which calculates the convolution by skipping zero values and reduces the amount of computation. Furthermore, the ECR method can complete extension, compression and sparse matrix computation by only accessing global memory once.
- We propose a new storage format (PECR). Besides exploiting sparsity, convolution layer and pooling layer are computed together, which effectively reduces not only the traffic between CPU and GPU but also the traffic from off-chip memory to on-chip memory of GPU.
- The code for the proposed algorithms is open source. A data set of feature maps is provided, which can be used for convolution optimization research.
- We evaluate the proposed algorithms on GPU platform. The result shows that our methods can achieve state-of-the-art speedup over cuDNN.

The rest of this paper is organized as follows. Section II presents the background. Section III introduces the motivation. Section IV describes the proposed ECR format and convolution method. Section V describes the PECR format and the corresponding convolution and pooling method. Section VI introduces the data set of feature maps and evaluates the proposed methods. Section VII discusses the related work. Section VIII concludes this paper.

II. BACKGROUND

In this section, we first introduce Graphic Processing Units (GPUs) and Compute Unified Device Architecture (CUDA) platform. Then, we present the convolution operation and the pooling operation for CNN. The notations used in the paper are described in Table I.

A. GPU/CUDA platform

GPUs are widely used to meet the needs of high performance computing for many applications, such as scientific computing, deep learning, Bioinformatics [32]. One GPU usually contains multiple Streaming Multiprocessors (SMs), and each SM contains multiple Streaming Processes (SPs)

TABLE I
NOTATIONS IN THE PAPER

B_i	block id of GPU thread block	T_i	thread id of GPU thread
i_w	width of feature map	i_h	height of feature map
k_w	width of the kernel	k_h	height of the kernel
p_w	width of pooling window	p_h	height of pooling window
c_s	stride size for convolution	p_s	stride size for pooling

[34]. Each SM can access a register file that works at the same speed as SP. Usually, each SM contains L1 cache, and all SMs shared an L2 cache. Each SM also has a shared memory that is similar to the L1 cache, but its data replacement is controlled by the programmer. The shared memory are shared between SPs within the same SM. The L1 cache in each SM shares a common on-chip memory with shared memory [35]. Global memory is off-chip and can be used for data transfer between GPU and CPU. CPU can access the global memory through the PCI-E bus. The latency of transferring data from CPU to GPU and from global memory to shared memory is very high, so it is beneficial to put reusable data in shared memory instead of frequently visiting CPU memory or global memory [36].

CUDA is a programming platform designed for GPU architecture. CUDA makes parallel programming on GPU more acceptable and promotes the development of parallel applications. On CUDA platform, all threads are contained in a thread grid, which consists of multiple thread blocks. The threads in a thread block share the same shared memory space. According to GPU's characteristics, threads are organized in the form of warp, which generally includes 32 threads [36].
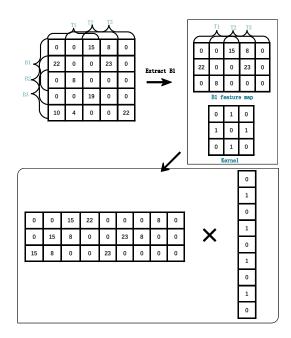
B. Convolution operation on GPU

Fig. 1. Example of convolution computation by matrix-vector multiplication.

GPU is suitable for matrix-related calculations, such as matrix multiplication. Convolution calculations have been

converted to matrix multiplications on GPU [12]. As shown in Fig. 1, the feature map is vertically divided into three large convolution block rows ($B1$, $B2$ and $B3$). One row of convolution blocks corresponds to one row of final convolution results. One thread block of GPU is assigned to compute one convolution block row. In each convolution block row, convolution kernel moves horizontally. Therefore, each convolution block row is divided into three convolution windows, and each convolution window corresponds to one convolution result. Each convolution result is assigned to one thread ($T1$, $T2$ or $T3$) on GPU. The values in the convolution window are extended to one row of a matrix, and the convolution kernel is extended to a vector. In this way, the convolution calculation is converted to matrix-vector multiplication. The result of the matrix-vector multiplication corresponds to a row of convolution results. Matrix-vector multiplication is much faster than convolution operation on GPUs, so the calculation speed is substantially improved. Accordingly, convolution operation with multiple convolution kernels can be transformed to matrix-matrix multiplication, which is also very suitable for GPU computing. In recent years, there have been some optimization methods for convolution on GPUs, which will be discussed in related work.

C. Pooling operation

To reduce dimensions of the output from convolution layer and prevent overfitting, the pooling layer is added after the convolution layer in most convolution neural networks. Convolutional neural networks usually have three kinds of pooling operations, mean-pooling, max-pooling, and stochastic-pooling. Similar to the convolution operation, the pooling operation also has a sliding window on the feature map, but no multiplication operation is required in this window. In the sliding window, the mean-pooling will take the average value from all values in the window. The max-pooling will take the max value from all values in the window. The stochastic-pooling will give a probability to each value in the window, then the one with the highest probability will be selected.

III. MOTIVATION

Convolutional neural network is an important machine learning tool that extracts the characteristic information of input data through convolution layers. However, convolution is a time-consuming task, which requires sampling the entire feature map. The convolution kernel performs multiplications and additions in the sample area throughout the feature map. If the size of feature map is $i_w \times i_h$ and kernel size is $k_w \times k_h$, a convolution operation requires num_{mul} multiplications (1) and num_{add} additions (2).

$$num_{mul} = \left(\frac{i_w - k_w}{c_s} + 1\right)\left(\frac{i_h - k_h}{c_s} + 1\right)(k_w k_h) \quad (1)$$

$$num_{add} = \left(\frac{i_w - k_w}{c_s} + 1\right)\left(\frac{i_h - k_h}{c_s} + 1\right)(k_w k_h - 1) \quad (2)$$

Feature maps of the deeper convolution layers are usually smaller, and the size can be as small as 5×5. Convolution

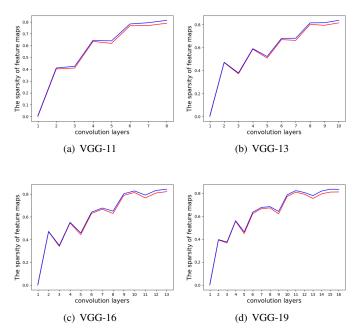(a) VGG-11   (b) VGG-13

(c) VGG-16   (d) VGG-19

Fig. 2. The sparsity of the feature maps for VGG network. The red curve is the original feature map, and the blue curve is the feature map after matrix transformation as in im2col.

calculation on GPU is generally converted into matrix multiplication (GEMM), which is often more efficient for large matrix. Because the number of GPU threads is small due to the limited size of the feature map, GEMM cannot fully exploit GPU's computation ability.

The sparsity of the deep feature map is relatively large due to RELU activation and pruning operations. We can see it from Fig. 2 that the sparsity of the feature map which will enter the convolution layer can reach more than 0.7 for deep layers in the network [45]. When the feature map is extended as in im2col, the matrix is even more sparse (blue curve). The traditional methods will calculate these zero values, which are useless for the final convolution result.

To address the above problems, we reduce the number of multiplications and additions in the convolution operation by compressing the feature map. In order to reduce the time consumption, we transform the traditional matrix conversion operation [12] into a compression operation. After compression, the convolution calculation is transformed into sparse matrix vector multiplication (SpMV) [37], which eliminates the calculation of redundant zero values. Furthermore, the process of data format transformation and SpMV only require one time global memory access.

On the other hand, traditionally, convolution layers are usually calculated in an independent GPU kernel. After the convolution results are transferred to CPU memory, the calculation of pooling layer is started, such as in Caffe [38] and PyTorch [39]. This will increase the traffic between CPU and GPU. As shown in Fig. 3, the time for tansferring data between CPU and GPU occupies a large proportion in the entire convolution and pooling calculation process. If the time consumption of data transfer between the CPU and GPU during convolution and pooling operations can be reduced,

the total time of CNN can be reduced effectively.

Fig. 3. Convolution and pooling of VGG-19 by cuDNN considering the data transfer between CPU and GPU. The total time is divided into data transfer time and computing time. CP_1 means the group of the first convolutional layer and the first pooling layer. The left vertical coordinate is time (s). The right vertical coordinate is proportion.

IV. SPARSE CONVOLUTION METHOD ON GPU

In this section, we will introduce a sparse convolution method for CNN on GPU. First, we present the whole procedure of CNN forward computing with the sparse convolution method. Then, a new storage format suitable for sparse convolution calculation on GPUs is proposed. At last, convolution method based on the new storage format is described in detail.

A. Whole algorithm procedure for CNN forward computing

In order to better understand the proposed method, we design an algorithm procedure with only one-time data transfer to reduce the traffic between CPU and GPU for the CNN forward computing. The algorithm procedures are as follows.

Step 0: Transfer input data (images/feature maps and filters) from CPU to GPU. Start GPU kernel.

Step 1: Load input data from global memory to shared memory and transform the data into the new format.

Step 2: Perform SpMV for convolution layer. One thread computes one convolution result.

Step 3: Perform pooling operations, and output the pooling result to global memory.

Step 4: Go to Step 1 unless it is the last pooling layer.

Step 5: Complete the computation for remaining layers.

Step 6: Transfer the result from GPU to CPU.

B. Extended and compressed row storage format

Zeros in feature maps for the convolution operation can lead to useless multiplications and additions. Therefore, we convert the feature map into a new storage format (Extended and Compressed Row, ECR), then the convolution can be converted to sparse matrix vector multiplication (SpMV). As shown in Fig. 4, the feature map is vertically divided into three large convolution block rows ($B1$, $B2$ and $B3$) according to the size of the stride c_s and the height of convolution kernel k_h. One convolution block row corresponds to one row of final convolution results. One thread block on GPU is assigned to compute one convolution block row. In each convolution

block row, convolution kernel moves horizontally. Therefore, each convolution block row is divided into three convolution windows, and each convolution window corresponds to one convolution result. Each convolution result is computed by one thread ($T1$, $T2$ or $T3$) on GPU.
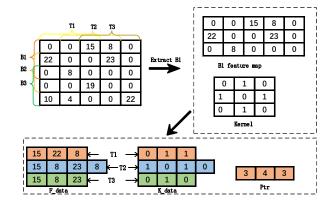

Fig. 4. A novel storage format (ECR) for one feature map and one kernel. The size of feature map is 5×5, and kernel size is 3×3, and the stride is 1.

One block row $B1$ is extracted to describe the new storage format. We store the three non-zero values in convolution window of $T1$ into F_data of $B1$ and store the corresponding values in the convolution kernel into K_data of $B1$. The non-zero values of all remaining convolution windows are stored into F_data in turn, along with the corresponding convolution kernel values into K_data. Besides, the number of non-zero values in each convolution window is stored in Ptr, which is also the number of multiplications required for each thread. If there is no non-zero value for a convolution window, a value of -1 should be stored in Ptr of $B1$ for markup. In all, non-zero values are stored in F_data, kernel values are stored in K_data, and the number of non-zeros values are stored in Ptr. Because the filter is also extended with feature map, bank conflicts will be reduced when all the threads visit the filter together in shared memory.

C. Load and transform data

After images/feature maps and filters are transferred from CPU to GPU's global memory, GPU computing starts. The feature maps are loaded, extended and transformed into ECR format at the same time (Fig. 4). After that, the feature map and the filter in ECR format are stored in shared memory.
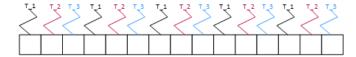

Fig. 5. Coalesced global memory access by threads in a warp.

As shown in Fig. 5, feature maps are stored in global memory as one-dimensional arrays for continuous access by adjacent threads. In this way, non-zero values of feature maps are loaded from global memory to shared memory, which can reduce the time of the global memory access. This advantage

of coalesced global memory access performs best when the convolution stride is 1.

For a single feature map, we can allocate $\frac{i_h - k_h}{c_s} + 1$ blocks, each of which contains $\frac{i_w - k_w}{c_s} + 1$ threads. As shown in Algorithm 1, the algorithm of one thread is described for converting one feature map and one filter to ECR format. F_{data} and K_{data} are declared shared memory. In Algorithm 1, $temp$ is a counter for non-zero values in the feature map. Each thread needs to read $k_w \times k_h$ values of feature map from global memory (Line 2 and 3). Line 4 sets the address $offset$ that this thread needs to access in global memory. $offset$ equals $block\_idx * i_w + thread\_idx * c_s + pos$. pos is the relative position in the convolution window. Line 5 judges the data value corresponding to the global memory position $offset$. If it is a non-zero value, it is stored in F_{data} with the position $thread\_idx * k_w * k_h + num_{nonzero}$, and the corresponding value in convolution kernel is stored in the corresponding position of K_{data} (Line 6, 7 and 8). After the loop, the number of non-zero values is stored in Ptr, and -1 is stored if there is no non-zero value (Line 12 to 16).

Algorithm 1 Storage format conversion algorithm for ECR

1: $temp \leftarrow 0$
2: **for** $i = 0$ to k_h **do**
3:    **for** $j = 0$ to k_w **do**
4:       $offset \leftarrow block\_idx * i\_w + thread\_idx * c\_s + i * i\_w + j$
5:       **if** $input[offset]! = 0$ **then**
6:          $F\_data[thread\_idx * k\_w * k\_h + temp] \leftarrow input[offset]$
7:          $K\_data[thread\_idx * k\_w * k\_h + temp] \leftarrow kernel[i + j * k_w]$
8:          $temp + +$
9:       **end if**
10:    **end for**
11: **end for**
12: **if** $temp! = 0$ **then**
13:    $ptr[thread\_idx] \leftarrow temp + 1$
14: **else**
15:    $ptr[thread\_idx] \leftarrow -1$
16: **end if**

D. Perform SpMV for convolution

In this subsection, we will describe the algorithm for sparse convolution calculation based on ECR and use Fig. 6 as an example to analyze how the number of multiplication and addition are reduced.

In the previous subsection, we obtained a new data format ECR, F_{data} for feature map, K_{data} for kernel data, and Ptr for the number of non-zero values. As shown in Fig. 6, we can consider F_{data} as a sparse matrix, and K_{data} as a vector with varying lengths. The length of the vector is the same as the corresponding value in Ptr. Therefore, the convolution operation is converted to a variant of sparse matrix vector multiplication calculation (SpMV), and zero values in the feature map are skipped without affecting convolution result.
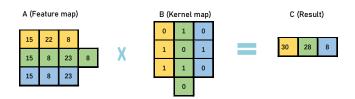

Fig. 6. Sparse matrix vector multiplication. The value in the same color is calculated by the same thread.

As shown in Fig. 6, for GPU implementation, one thread multiplies a row in the compressed feature matrix (A) with the corresponding vector in the kernel map (B) to obtain a value in a convolution result matrix (C). Therefore, a thread block contains $\frac{i_w - k_w}{c_s} + 1$ threads, and can get one row of the final convolution results. After parallel calculation by multiple thread blocks, the convolution result for the whole feature map can be obtained.

The pseudo-code for a single thread to get a convolution result is described in Algorithm 2. Line 1 accesses the corresponding ptr vector in the shared memory to determine whether there are non-zero values stored in F_{data}. If $ptr[thread\_idx]$ is -1, it is immediately judged that no operation is needed, and the output is 0 (Line 2). If the value stored in $ptr[thread\_idx]$ is not -1, the values in K_{data} and F_{data} are accessed in turn and multiply-add operations are performed (Line 4~6). The convolution result is finally obtained and stored into global memory for the computation of next layer (e.g. pooling layer).

As shown in Fig. 4 and Fig. 6, for three convolution block rows (B1, B2, and B3), the conventional algorithm requires 24 additions and 27 multiplications, while our algorithm only requires 7 additions and 10 multiplications, reducing 71% additions and 63% multiplications. Besides, K_{data}, F_{data}, and ptr are all stored in shared memory. Therefore, the proposed method can greatly improve the speed of convolution, and accelerates the whole convolution neural network.

Algorithm 2 Convolution algorithm for ECR

1: **if** $ptr[thread\_idx] == -1$ **then**
2:    $output[thread\_idx + block\_idx * blockDim.x] \leftarrow 0$
3: **else**
4:    **for** $i = thread\_idx * k_w * k_h$ to $thread\_idx * k_w * k_h + ptr[thread\_idx] - 1$ **do**
5:       $output[thread\_idx + block\_idx * blockDim.x] + = F_{data}[i] * K_{data}[i]$
6:    **end for**
7: **end if**

E. Other steps for CNN forward computing

The convolution results are processed by RELU before they are outputted to the global memory. As threads in different thread blocks cannot communicate through shared memory, we have to store the activation results into global memory before pooling operation starts. Because pooling operation itself take up only a little time [44], we don't discuss it's implementation

here. After pooling results are outputted to global memory, a new round is started by loading the new feature map and filter to shared memory (Fig. 5), and transforming them into ECR format for the next convolution layer. This process is repeated until all convolution layers and pooling layers are finished.

F. Discussion

In this section, we describe in detail the sparse convolution algorithm for one feature map and one kernel. The proposed ECR method can complete extension, compression, and sparse matrix computation by only accessing global memory once, which can effectively reduce the off-chip memory traffic. However, in the actual application, multiple feature maps and multiple kernels can be included in one convolution layer. The proposed algorithm can be extended to process this case by increasing the number of GPU threads. Since the amount of calculation in a single thread is reduced by the proposed algorithm, the calculation speed can also be improved with more threads. In addition, similar to Sparse Tensor Core in the Ampere architecture [43], the proposed method can also be applied to the hardware design.

V. COMBINING CONVOLUTION AND POOLING

Traditionally, after the convolution results are transferred from GPU to CPU, the calculation of pooling layer is started. This will increase the cost of data transfer. In this section, we propose an optimization method which calculates convolution layer and pooling layer together without transferring intermediate results, which can effectively reduce the memory traffic and accelerate CNN.

A. Whole algorithm procedure for CNN forward computing

For the forward computing of CNN, the algorithm procedure should also try to ensure one-time data transfer from CPU to GPU. Therefore, the traffic between CPU and GPU is reduced. For better understanding the proposed method, we present the whole process of CNN forward computing as follows.

Step 0: Transfer the images and filters from CPU to GPU. Start GPU kernel.

Step 1: Load data (images/feature maps and filters) from global memory to shared memory. At the same time, transform the data to the new format.

Step 2: Perform SpMV for convolution layer. One thread computes one pooling unit, which includes several convolution results, e.g. 4 (2*2).

Step 3: Each thread gets a pooling result, and stores it to global memory.

Step 4: Go to Step 1 unless it is the last pooling layer.

Step 5: Complete the computation for remaining layers.

Step 6: Transfer the final result from GPU to CPU.

B. Sparse storage format considering both convolution and pooling

In the traditional convolutional neural network, the result of the convolution operation will enter the pooling layer after the activation operation. Existing calculation methods will transfer

the convolution results to the CPU side, and then again to the GPU side for pooling operations. Such a calculation process will increase the overall CNN calculation time due to the bandwidth limitation between GPU and CPU. Therefore, we redesign the calculation process, and combine the pooling operation with the convolution calculation, which reduces the data transfer between CPU and GPU and improves the overall performance of CNN.
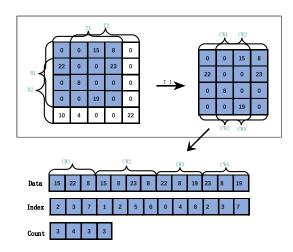

Fig. 7. PECR format. The size of feature map is 5×5. The size of convolution kernel is 3×3. The size of pooling window is 2×2. The stride of both convolution and pooling is 1.

The new storage format is named as Pooling-pack Extended and Compressed Row (PECR) format. As shown in Fig. 7, the feature map is vertically divided into two large pooling-pack rows ($B1$ and $B2$) according to the size of the stride c_s and the height of convolution kernel k_h. One pooling-pack row corresponds to one row of pooling results. One thread block on GPU is assigned to compute one pooling-pack row. Each pooling-pack row is divided into two pooling windows, and each pooling window corresponds to one pooling result. Each pooling result is obtained by one thread ($T1$ or $T2$) on GPU.

Non-zero values in the corresponding pooling window are stored in $Data$. The non-zero values of all convolution windows are stored into $Data$ in an order of left-to-right and top-to-bottom. In order to complete the convolution calculation, the index pointing to corresponding value in the filter is stored in $Index$. Using such a storage method, each thread will process a sub-feature map with the size of $T_w \times T_h$ in the original feature map. T_w equals $k_w + c_s(p_w - 1)$, and T_h generally equals T_w. For a pooling window with the size of $p_w \times p_h$, each thread will process $p_w \times p_h$ convolution windows. For example, there are 2×2 pooling windows in Fig. 7, CW1, CW2, CW3, and CW4.

C. Load and transform data

The feature maps and filters are loaded from global memory to shared memory in a coalesced way similar to Fig. 5. At the same time, the input data are transformed to PECR format. The process of loading and converting data to PECR format for one GPU thread is described in Algorithm 3. Each thread processes $p_w * p_h$ convolution windows. For each convolution window

$(k_w * k_h)$ in the feature map, the thread loads non-zero values from global memory to shared memory, and computes the according index in the filter. Therefore, the time complexity of Algorithm 3 is $O(p_w * p_h * k_w * k_h)$. Each feature map is processed by $n_o{}^2$ threads (3).

$$n_o = \frac{(I_w - k_w + c_s - c_s * p_w + p_s * c_s)}{p_s * c_s} \qquad (3)$$

Algorithm 3 Convert feature map and filter into PECR format

1: $start \leftarrow thread\_idx * c_s * p_s + i_w * (block\_idx * c_s)$
2: $cnt \leftarrow 0$
3: **for** $n = 0$ to $p_w * p_h$ **do**
4:    $num \leftarrow 0$
5:    $n\_start \leftarrow n/(p_w) * c_s * i_w + n\%p_h * c_s$
6:    **for** $i = 0$ to k_w **do**
7:      **for** $j = 0$ to k_h **do**
8:        $offset \leftarrow start + n\_start + i * i_w + j$
9:        **if** $Input[offset]! = 0$ **then**
10:          $Data[cnt] \leftarrow Input[offset]$
11:          $Index[cnt] \leftarrow i * j + i$
12:          $cnt+ = 1$
13:          $num+ = 1$
14:        **end if**
15:      **end for**
16:    **end for**
17:    $count[n] \leftarrow num$
18: **end for**

D. Convolution and pooling

After the feature map and the filter are transformed to PECR format and stored in shared memory, the convolution and pooling operations are started. Based on PECR format, each thread performs four SpMV operations to get four convolution results (Fig. 8). After that, RELU is used as the activation function, and a value less than zero is set to zero. Then, maximum pooling is used to get the final pooling result. By combining convolution and pooling, pooling result is obtained in one thread without outputting it, which can greatly reduce the traffic to global memory and CPU. Furthermore, the synchronization between threads is also avoided.
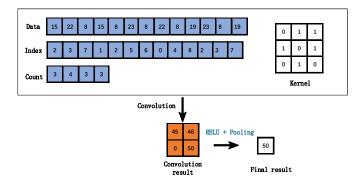

Fig. 8. Convolution and pooling with PECR format.

Algorithm 4 describes the convolution and pooling algorithm based on PECR. A thread needs to process $p_w * p_h$ convolution windows and compute one pooling result by max-value operation (Line 8~10). Because different thread blocks cannot share data in shared memory, the pooling results are outputted to global memory. However, if only one thread block is started, the pooling results can be outputted to shared memory for the computation of the next convolution layer.

Algorithm 4 Convolution and pooling based on PECR

1: $temp \leftarrow 0$
2: $max \leftarrow 0$
3: **for** i to $p_w * p_h$ **do**
4:    $Output[i] \leftarrow 0$
5:    **for** $n \leftarrow temp$ to $temp + count[i]$ **do**
6:      $Output[i]+ = Data[n] * Kernel[Index[n]]$
7:    **end for**
8:    **if** $max < Output[i]$ **then**
9:      $max \leftarrow Output[i]$
10:      $temp+ = Count[i]$
11:    **end if**
12:    $maxvalue[thread\_idx] \leftarrow max$
13: **end for**

E. Discussion

In this section, we describe in detail the convolution and pooling algorithm for one feature map and one kernel. However, there are sometimes multiple feature maps and multiple filters. In this case, more traffic between CPU and GPU is needed for cuDNN. Therefore, time consumption of data transfer between CPU and GPU can be further reduced with the proposed method, and the speedup will also be improved. In addition, the proposed method can also be used on other hardware platforms. It is worth noting that in the multi-channel convolution calculation, the data of other channels follow the same operation in turn. After all the channels are compressed, SpMV starts to run to ensure that the calculation results are correct.

VI. EXPERIMENTS AND DATA SET

In this section, we present and analyze the experimental results of the proposed algorithms with various neural network models on two GPU platforms. First, we introduce the experimental environment. Then, we introduce a new data set for convolution optimization. At last, we present the speedup comparison results.

A. Experiment platform and data set

As shown in Table II, the experiment is carried out on two different platforms. A data set is provided for convolution optimization, which is a collection of all input feature maps for the convolution layers. The data set is obtained by having a picture of a cat (from ImageNet) through the entire network. Directly using this data set to analyze and optimize the convolution calculation is more convenient and improves work

TABLE II
EXPERIMENT ENVIRONMENT

	Platform 1	Platform 2
CPU	Intel Xeon CPU E5v3, 128GB DRAM	AMD Ryzen 3600X, 12GB DRAM
GPU	NVIDIA GeForce GTX 1080	NVIDIA GeForce RTX 2080
OS	Ubuntu18.04	Ubuntu18.04
CUDA	10.0	10.0
cuDNN	7.6.1	7.1.4

efficiency. The current data set contains all the feature maps of VGG-19 that require convolution calculations. It also lists file names and sizes of all feature maps. The code as well as the data set in this paper are open access [1]. In future, we will add more network models.
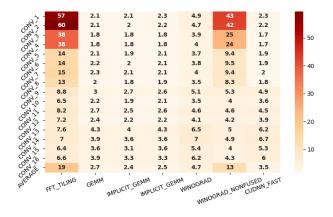
B. Speed comparison for convolution layers

TABLE III
SPEEDUP OF ECR OVER CUDNN-FAST FOR SINGLE CONVOLUTION
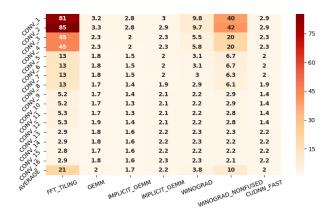LAYER ON NVIDIA GEFORCE GTX 1080

Network	Layer	Size	Sparsity	Speedup
LeNet	Conv2	11×11	0.95	1.54
AlexNetC	Conv3	6×6	0.9	2.04
AlexNetI	Conv4	5×5	0.9	3.40
GoogLeNet	Inception4a.1	14×14	0.9	2.42
GoogLeNet	Inception4a.2	14×14	0.9	2.42
GoogLeNet	Inception4e.3	14×14	0.9	3.57
GoogLeNet	Inception5a.1	7×7	0.95	2.40
GoogLeNet	Inception5a.2	7×7	0.9	2.25
GoogLeNet	Inception5b.3	7×7	0.95	3.43
GoogLeNet	Inception4a.7	7×7	0.95	2.08

In this subsection, we carried out speed comparison experiments for a single convolution layer. As shown in Table III, some convolution layers are extracted from different models, such as LeNet, AlexNet and GoogLeNet. We can see that the convolution speed of some layers can be up to $3.5\times$ compared to CUDNN-FAST, which can automatically choose a best implementation in cuDNN. For feature maps in deep networks, the size is very small comparing with the initial input feature map. The traditional GEMM-based method is not suitable for matrix multiplication with small feature maps [15]. ECR algorithm reduces the amount of computation for a single thread according to the characteristic of sparsity and performs better for these small feature maps.

Fig. 9(a) shows comparison of execution time for convolution layers of VGG-19 on NVIDIA Geforce GTX 1080 between ECR and other methods, which are implemented in cuDNN. ECR has better performance than all the other methods. Compared with CUDNN-FAST, ECR can achieve up to 6.7X speedup for a single convolution layer (CONV_14). On average, the speedup over CUDNN-FAST is 3.5X. Fig. 9(b) shows comparison of execution time for convolution layers of VGG-19 on NVIDIA Geforce RTX 2080 between ECR and other methods. ECR has better performance than all the other methods. Compared with CUDNN-FAST, ECR

(a) NVIDIA Geforce GTX 1080.

(b) NVIDIA Geforce RTX 2080.

Fig. 9. Speedup of ECR over other methods for convolution layers of VGG-19. Stride is 1. The x-axis represents different convolution methods, and the y-axis represents different convolution layers.
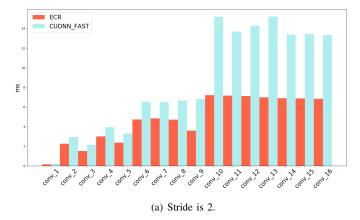
can achieve up to 2.9X speedup for a single convolution layer (CONV_1 and CONV_2). On average, the speedup over CUDNN-FAST is 2X. As shown in Fig. 10(a) and Fig. 10(b), we also performed experiments with strides of 2 and 3 using the feature map data set of VGG-19 on NVIDIA Geforce GTX 1080. Our method can achieve 1.8X (stride is 2) and 1.75X (stride is 3) speedup over CUDNN-FAST on average.

According to Fig. 2, the sparsity is usually larger and the size is usually smaller for the feature map of a deeper convolution layer. In order to better understand how the sparsity and size of feature maps can together affect speedup, we present a quantized value $\Theta = Sparsity/Size$. When computing Θ, $Sparsity$ is multiplied by 100 and $Size$ is the width of the feature map. As shown in Fig. 11, the value of Θ becomes larger when the network becomes deeper. The speedup has the similar trend to Θ. In other words, ECR algorithm performs better for convolution calculations of deep networks with smaller feature map and larger sparsity.

C. Speed comparison for both convolution and pooling

Fig. 12(a) describes the speedup of PECR over other methods for pooling and convolution layers of VGG-19 on Platform

---

[1]https://github.com/milk2we/conv_pool_algorithm
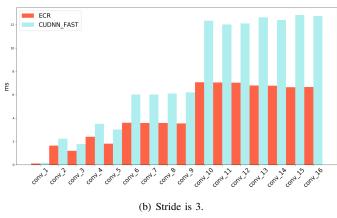
(a) Stride is 2.

(b) Stride is 3.

Fig. 10. Convolution calculation time for different strides. The x-axis represents different convolution layers, and the y-axis represents time (s).
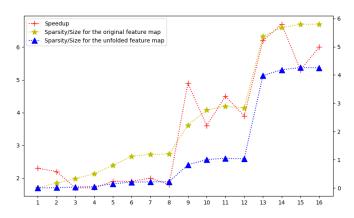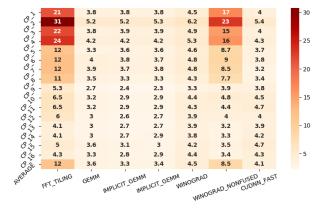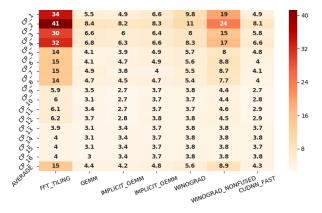

Fig. 11. Relationship between sparsity, size and speedup. The left axis represents speedup, and the right axis represents $Sparsity/Size$

1. PECR has better performance than all the other methods. Compared with the CUDNN-FAST, PECR can achieve up to 5.4X speedup. On average, the speedup over CUDNN-FAST is 4.1X. Fig. 12(b) describes the speedup of PECR over other methods for pooling and convolution layers of VGG-19 on Platform 2. PECR has better performance than all the other methods. Compared with CUDNN-FAST, PECR can achieve up to 8.1X speedup for a single convolution layer. On average, the speedup over CUDNN-FAST is 4.3X. It is found that larger speedup over CUDNN-FAST can be achieved after combining convolution and pooling (4.1X vs 3.5X on Platform 1, 4.3X

vs 2X on Platform 2). The reason is that the traffic between CPU and GPU is effectively reduced by PECR.

	FFT_TILING	GEMM	IMPLICIT_GEMM	IMPLICIT_GEMM	WINOGRAD	WINOGRAD_NONFUSED	CUDNN_FAST
conv_1	21	3.8	3.8	3.8	4.5	17	4
conv_2	31	5.2	5.2	5.3	6.2	23	5.4
conv_3	22	3.8	3.9	3.9	4.9	15	4
conv_4	24	4.2	4.2	4.2	5.3	16	4.3
conv_5	12	3.3	3.6	3.6	4.6	8.7	3.7
conv_6	12	4	3.8	3.7	4.8	9	3.8
conv_7	12	3.9	3.7	3.8	4.8	8.5	3.2
conv_8	11	3.5	3.3	3.3	4.3	7.7	3.4
conv_9	5.3	2.7	2.4	2.3	3.3	3.9	3.8
conv_10	6.5	3.2	2.9	2.9	4.4	4.8	4.5
conv_11	6.5	3.2	2.9	2.9	4.3	4.4	4.7
conv_12	6	3	2.6	2.7	3.9	4	4
conv_13	4.1	3	2.7	2.7	3.9	3.2	3.9
conv_14	4.1	3	2.7	2.9	3.8	3.3	4.2
conv_15	5	3.6	3.1	3	4.2	3.5	4.7
conv_16	4.3	3.3	2.8	2.9	4.4	3.4	4.3
AVERAGE	12	3.6	3.3	3.4	4.5	8.5	4.1

(a) NVIDIA Geforce GTX 1080.

	FFT_TILING	GEMM	IMPLICIT_GEMM	IMPLICIT_GEMM	WINOGRAD	WINOGRAD_NONFUSED	CUDNN_FAST
conv_1	34	5.5	4.9	6.6	9.8	19	4.9
conv_2	41	8.4	8.2	8.3	11	24	8.1
conv_3	30	6.6	6	6.4	8	15	5.8
conv_4	32	6.8	6.3	6.6	8.3	17	6.6
conv_5	14	4.1	3.9	4.9	5.7	8	4.8
conv_6	15	4.1	4.7	4.9	5.6	8.8	4
conv_7	15	4.9	3.8	4	5.5	8.7	4.1
conv_8	14	4.7	4.5	4.7	5.4	7.7	4
conv_9	5.9	3.5	2.7	3.7	3.8	4.4	2.7
conv_10	6	3.1	2.7	3.7	3.7	4.4	2.8
conv_11	6.1	3.4	2.7	3.7	3.7	4.6	2.9
conv_12	6.2	3.7	2.8	3.8	3.8	4.5	2.9
conv_13	3.9	3.1	3.4	3.7	3.8	3.8	3.7
conv_14	4	3.1	3.4	3.7	3.8	3.8	3.8
conv_15	4	3.1	3.4	3.7	3.8	3.8	3.7
conv_16	4	3	3.4	3.7	3.8	3.8	3.8
AVERAGE	15	4.4	4.2	4.8	5.6	8.9	4.3

(b) NVIDIA Geforce RTX 2080.

Fig. 12. Speedup of PECR over other methods for pooling and convolution layers of VGG-19. The x-axis represents different methods. The y-axis represents different convolution and pooling layers.

VII. RELATED WORK

The methods of accelerating CNN can be classified into algorithm level and architecture level. We introduce some algorithm-level optimization technologies which are most-related to this research in this section.

im2col im2col first extends the feature maps and filters, and then transforms convolution into matrix multiplication [12]. This method gets a good acceleration effect because GPU is good at accelerating GEMM. However, im2col cannot achieve satisfactory performance for small feature maps because GEMM for small matrices cannot fully exploit the computation ability of GPU [24].

FFT Fast Fourier Transform (FFT) is a computational tool commonly used for signal analysis. FFT-based methods compute convolutions as pointwise products in the Fourier domain while reusing the same transformed feature map many times [12]. The time complexity of FFT-based method is reduced compared with direct convolution. FFT-based methods usually performs better at a larger filter size.

Winograd Winograd [13], [40] is a method based on Winograd minimal filtering algorithms. Winograd method dramatically reduces the arithmetic complexity compared with direct convolution. Winograd-based methods perform well for small kernels and small batch sizes.

VIII. CONCLUSION

In this paper, two new methods are proposed to optimize CNN on GPUs. First, the method based on ECR format skips the computation for zero values in the feature map. Experimental results show that the proposed ECR method can reach 3.5X speedup over cuDNN. Second, a PECR method is proposed not only to avoid computing zero values but also to compute convolution layer and pooling layer together in one thread, which effectively reduce the time for transferring data between CPU and GPU. Experimental results show that the proposed PECR method can reach 4.3X speedup over cuDNN.

REFERENCES

[1] Tan, Mingxing, and Quoc Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks." International Conference on Machine Learning. 2019.
[2] Han Z, Wei B, Zheng Y, et al. "Breast cancer multi-classification from histopathological images with structured deep learning model". Scientific reports, 2017, 7(1): 4172.
[3] He Y, Xiang S, Kang C, et al. "Disan: Directional self-attention network for rnn/cnn-free language understanding." Thirty-Second AAAI Conference on Artificial Intelligence. 2018.
[4] Zheng Lei, et al. "Joint deep modeling of users and items using reviews for recommendation." Proceedings of the Tenth ACM International Conference on Web Search and Data Mining. ACM, 2017.
[5] Xuezhe Ma, and Eduard Hovy. "End-to-end Sequence Labeling via Bidirectional LSTM-CNNs-CRF." Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics, 2016.
[6] Jason P.C. Chiu, and Eric Nichols. "Named Entity Recognition with Bidirectional LSTM-CNNs." Transactions of the Association for Computational Linguistics, Vol.4, 2016.
[7] X. Li, et al. "Performance Analysis of GPU-Based Convolutional Neural Networks," 2016 45th International Conference on Parallel Processing (ICPP), Philadelphia, PA, 2016, pp. 67-76.
[8] Zhang C, Li P, Sun G, et al. "Optimizing fpga-based accelerator design for deep convolutional neural networks" Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2015: 161-170.
[9] Zhang, Chen, et al. "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2018).
[10] NVIDIA. 2018. CUDA Documentation. http://docs.nvidia.com/cuda/cublas/index.html. (2018)
[11] Zhu, M., Zhang, T., Gu, Z., Xie, Y. (2019, October). Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (pp. 359-371).
[12] Chetlur S, Woolley C, Vandermersch P, et al. cuDNN: Efficient primitives for deep learning[J]. arXiv preprint arXiv:1410.0759, 2014.
[13] Li, Xiuhong, et al. "A coordinated tiling and batching framework for efficient GEMM on GPUs." Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. ACM, 2019.
[14] Mathieu, et al. "Fast training of convolutional networks through ffts." arXiv preprint arXiv:1312.5851 (2013).
[15] Lavin, Andrew, and Scott Gray. "Fast algorithms for convolutional neural networks." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016.
[16] NVIDIA cuDNN - GPU accelerated deep learning. https://developer.nvidia. com/cuDNN, 2014.
[17] Shi S, Chu X. Speeding up convolutional neural networks by exploiting the sparsity of rectifier units[J]. arXiv preprint arXiv:1704.07724, 2017.
[18] Li, Jiajun, et al. "SqueezeFlow: A Sparse CNN Accelerator Exploiting Concise Convolution Rules." IEEE Transactions on Computers (2019).

[19] Parashar, Angshuman, et al. "Scnn: An accelerator for compressed-sparse convolutional neural networks." 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture. IEEE, 2017.
[20] Dey, Sourya, et al. "Accelerating training of deep neural networks via sparse edge processing." International Conference on Artificial Neural Networks. Springer, Cham, 2017.
[21] Liu, Baoyuan, et al. "Sparse convolutional neural networks." Proceedings of IEEE Conference on Computer Vision and Pattern Recognition. 2015.
[22] Park, J., Li, S., Wen, W., Tang, P. T. P., Li, H., Chen, Y., Dubey, P. (2016). Faster cnns with direct sparse convolutions and guided pruning. arXiv preprint arXiv:1608.01409.
[23] Fan S, Yu H, Lu D, et al. CSCC: Convolution Split Compression Calculation Algorithm for Deep Neural Network[J]. IEEE Access, 2019.
[24] Rovder, Simon, et al. "Optimising Convolutional Neural Networks Inference on Low-Powered GPUs." (2019).
[25] Li C, et al. Optimizing memory efficiency for deep convolutional neural networks on GPUs. SC'16. IEEE, 2016: 633-644.
[26] M. Naumov, et al. cuSPARSE Library, https://www.nvidia.com/content/GTC-2010/pdfs/2070_GTC2010.pdf, 2010.
[27] Gale, T., Zaharia, M., Young, C., Elsen, E. (2020). Sparse GPU Kernels for Deep Learning. arXiv preprint arXiv:2006.10901.
[28] Lee, C. L., Chao, C. T., Lee, J. K., Hung, M. Y., Huang, C. W. (2019, August). Accelerate DNN Performance with Sparse Matrix Compression in Halide. In Proceedings of the 48th International Conference on Parallel Processing: Workshops (pp. 1-6).
[29] Han, S., Pool, J., Tran, J., Dally, W. (2015). Learning both weights and connections for efficient neural network. In Advances in neural information processing systems (pp. 1135-1143).
[30] Yao, Z., Cao, S., Xiao, W., Zhang, C., Nie, L. (2019, July). Balanced sparsity for efficient dnn inference on gpu. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 33, pp. 5676-5683).
[31] Chen, X. (2018). Escort: Efficient sparse convolutional neural networks on gpus. arXiv preprint arXiv:1802.10280.
[32] Wan X, Zhang F, Chu Q, et al. High-performance blob-based iterative three-dimensional reconstruction in electron tomography using multi-GPUs. BMC bioinformatics. BioMed Central, 2012, 13(10): S4.
[33] J. Cheng, J. Wu, C. Leng, Y. Wang and Q. Hu, "Quantized CNN: A Unified Approach to Accelerate and Compress Convolutional Networks," in IEEE Transactions on Neural Networks and Learning Systems, vol. 29, no. 10, pp. 4730-4743, Oct. 2018, doi: 10.1109/TNNLS.2017.2774288.
[34] Lindholm E, Nickolls J, Oberman S, et al. NVIDIA Tesla: A unified graphics and computing architecture. IEEE micro, 2008, 28(2): 39-55.
[35] Nickolls J, Dally W J. The GPU computing era[J]. IEEE micro, 2010, 30(2): 56-69.
[36] Kirk D. NVIDIA CUDA software and GPU parallel computing architecture. ISMM. 2007, 7: 103-104.
[37] Xu W, et al. Optimizing sparse matrix vector multiplication using cache blocking method on Fermi GPU. 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing. IEEE, 2012: 231-235.
[38] Jia, Yangqing, et al. "Caffe: Convolutional architecture for fast feature embedding." Proceedings of the 22nd ACM international conference on Multimedia. 2014.
[39] Paszke, Adam, et al. "Pytorch: An imperative style, high-performance deep learning library." Advances in neural information processing systems. 2019.
[40] Da Yan, et al. 2020. Optimizing batched winograd convolution on GPUs. In Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, NY, USA, 32-44.
[41] Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen et al. "Pytorch: An imperative style, high-performance deep learning library." arXiv preprint arXiv:1912.01703 (2019).
[42] Dillon, Joshua V., et al. "Tensorflow distributions." arXiv preprint arXiv:1711.10604 (2017).
[43] Choquette, Jack, et al. "3.2 The A100 Datacenter GPU and Ampere Architecture." 2021 IEEE International Solid-State Circuits Conference (ISSCC). Vol. 64. IEEE, 2021.
[44] Xu QQ, An H, Wu Z, Jin X. Hardware Design and Performance Analysis of Mainstream Convolutional Neural Networks. Computer Systems and Applications, 2020, 29(2): 49-57(in Chinese).http://www.c-s-a.org.cn/1003-3254/7257.html
[45] Wen W, Wu C, Wang Y, et al. Learning structured sparsity in deep neural networks[J]. arXiv preprint arXiv:1608.03665, 2016.

Weizhi Xu received his Ph.D. degree in computer architecture from Institute of Computing Technology, Chinese Academy of Sciences. He has worked as a postdoctoral researcher in Institute of Microelectronics, Tsinghua University. He is currently an associate professor in School of Information Science and Engineering, Shandong Normal University. His research interests include high performance computing, deep learning, video processing and natural language processing.

Shengyu Fan received the BE degree in the internet of things from the Shandong Normal University, Jinan, China, in 2020. Currently, he is working toward a master's degree majoring in computer science at the School of Information Science and Engineering, Shandong Normal University, China, under Prof. Weizhi Xu's advising. His main research interests include high performance computing and natural language processing.

Hui Yu received her Ph.D. degree in computer science from Institute of Computing Technology, Chinese Academy of Sciences. She is currently an assistant professor in Shandong Normal University. Her research interests include natural language processing and deep learning. She has published more than 20 papers in academic conferences and journals. She has participated several machine translation evaluation competitions such as WMT, IWSLT, NIST and CWMT.

Xin Fu (Member, IEEE) received the PhD degree in computer engineering from the University of Florida, Gainesville, Florida, in 2009. She was a NSF computing innovation fellow with the Computer Science Department, the University of Illinois at Urbana-Champaign, Urbana, Illinois, from 2009 to 2010. From 2010 to 2014, she was an assistant professor with the Department of Electrical Engineering and Computer Science, the University of Kansas, Lawrence, Kansas. Currently, she is an associate professor with the Electrical and Computer Engineering Department, the University of Houston, Houston, Texas. Her research interests include high-performance computing, machine learning, energy-efficient computing, and mobile computing. She is a recipient of 2014 NSF Faculty Early CAREER Award, 2012 Kansas NSF EPSCoR First Award, and 2009 NSF Computing Innovation fellow.