Neural Computing and Applications
https://doi.org/10.1007/s00521-021-06433-5

ORIGINAL ARTICLE

EPMC: efficient parallel memory compression in deep neural network training

Zailong Chen[1] · Shenghong Yang[1] · Chubo Liu[1] · Yikun Hu[1] · Kenli Li[1] · Keqin Li[2]

Received: 4 November 2020 / Accepted: 17 August 2021

Abstract
Deep neural networks (DNNs) are getting deeper and larger, making memory become one of the most important bottlenecks during training. Researchers have found that the feature maps generated during DNN training occupy the major portion of memory footprint. To reduce memory demand, they proposed to encode the feature maps in the forward pass and decode them in the backward pass. However, we observe that the execution of encoding and decoding is time-consuming, leading to severe slowdown of the DNN training. To solve this problem, we present an efficient parallel memory compression framework—*EPMC*, which enables us to simultaneously reduce the memory footprint and the impact of encoding/decoding on DNN training. Our framework employs pipeline parallel optimization and specific-layer parallelism for encoding and decoding to reduce their impact on overall training. It also combines precision reduction with encoding for improving the data compressing ratio. We evaluate *EPMC* across four state-of-the-art DNNs. Experimental results show that *EPMC* can reduce the memory footprint during training to 2.3 times on average without accuracy loss. In addition, it can reduce the DNN training time by more than 2.1 times on average compared with the unoptimized encoding/decoding scheme. Moreover, compared with using the common compression scheme Compressed Sparse Row, *EPMC* can achieve data compression ratio by 2.2 times.

Keywords Data compression · Deep neural networks training · Multitask parallelism · Pipeline parallelism.

1 Introduction

The outstanding performance of deep neural networks (DNNs) has made them more popular and widely used [1, 2]. Coupled with the support of large datasets and powerful computing resources, DNNs can address increasingly complex problems, such as action detection, face recognition, translation, and speech processing [3–8]. Increasingly complex and extensive intelligent applications have put forward higher requirements on the DNNs accuracy, leading to DNNs being getting larger and deeper, from LeNet [9] with five layers to ResNet [10] with thousands of layers. However, this also poses greater challenge on memory, and more data need to be stored during DNN training. For instance, AlexNet [11] with five convolution layers and two fully connected layers only needs 1.1 GB memory during training. On the contrary, VGG-16 [12] with 16 convolution layers and three fully connected layers occupies 28 GB memory footprint when training with a batch size of 256. This far exceeds the 12 GB memory capacity of the state-of-the-art Nvidia Titan X. Therefore, memory becomes one of the most important bottlenecks in DNN training [13, 14].

Most of the existing techniques (e.g., network pruning, network quantization, low-rank approximation, and knowledge distillation) for memory optimization aim at DNN inferring, and the key point is to cut back model size (reduction in weights' number or bit width) [15–28]. However, DNN training can hardly benefit from these techniques, because the feature maps generated by the intermediate layers of the network are the major reason for

✉ Shenghong Yang
ysh@hnu.edu.cn

✉ Kenli Li
lkl@hnu.edu.cn

[1] College of Information Science and Engineering, Hunan University, Changsha 410082, Hunan, China

[2] Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

Published online: 29 August 2021

the excessive memory consumption, and the weights only account for a small part [29]. Moreover, model compression methods require a trade-off between model size and model performance, because they more or less lead to model accuracy loss. Therefore, many optimization techniques targeting feature maps have emerged. For example, Chen et al. proposed an effective approach that recalculates the feature maps in the backward pass of DNN training instead of storing them in GPU main memory [30]. Jain et al. [29] proposed to encode the feature maps in the forward pass and decode them in the backward pass for reducing memory stress during DNN training. In [13], Rhu et al. developed an approach to reduce memory usage in training by transferring feature maps back and forth between CPU and GPU memory. Unfortunately, these approaches mentioned above all suffer from severe time overhead when solving the problem of memory.

As we all know, the training of DNN is time-consuming. How to effectively reduce the memory usage without affecting the efficiency is a great challenge. In order to deal with this, we present a framework—*EPMC*, which aims at reducing the memory footprint of feature maps in DNN training and optimizing the execution of data compression algorithm to reduce the increased overhead. *EPMC* 's implementation mainly has three stages. First, it analyzes the execution graph of the DNN and identifies where the compression and decompression modules can be inserted. Second, it reasonably allocates GPU computing resources to create the parallel framework based on the computational complexity of the encoding/decoding task and the specific ReLU layer. ReLU is usually used as an activation function in most existing DNNs to increase the nonlinear relationship between the layers of the neural network and has been proven to be superior to other activation functions (such as Sigmoid and Tanh) because of its smaller computational overhead and excellent effects in preventing overfitting and gradient vanishing. Therefore, we use ReLU as an example to demonstrate the effectiveness of our algorithm. Third, it builds a new directed computing graph with efficient encoding and decoding components.

The main contributions and differences of this paper are listed as follows.

- *Systematic execution analysis* We perform systematic analyses of computation resources and execution time of each module in DNN training. We observe that the data compression algorithm occupies a lot of training time and incurs a GPU underutilization. We also make a new observation that the data compression algorithm can perform the pipeline parallelism and multitask parallelism with ReLU to reduce its impact on training.
- *PipeCompress* We present PipeCompress, which is a pipeline parallel optimization approach for the data compression algorithm of compressed sparse row (CSR). This approach significantly improves the execution speed of encoding and decoding.
- *Specific-Layer parallelism* We propose to execute PipeCompress and specific ReLU layer in parallel, leading to a further influence reduction in DNN training.
- *VPEncoding* We present VPEncoding, which combines the precision reduction with encoding to further lower the memory footprint of feature maps without affecting the model accuracy. This approach can be extended to other data compression algorithms.

We evaluate *EPMC* with four state-of-the-art image classification DNNs in Pytorch (a deep learning framework). Our evaluation shows that *EPMC* can reduce the memory footprint during training to 2.3 times on average without accuracy loss. In addition, it can reduce the DNN training time by more than 2.1 times on average compared with the unoptimized encoding/decoding scheme. Moreover, compared with using the common compression scheme compressed sparse row (CSR), *EPMC* can achieve data compression ratio by 2.2 times. Therefore, it achieves the purpose of simultaneously reducing the memory consumption and the impact on efficiency during training.

This paper is organized as follows. Section 2 introduces related work. Section 3 introduces the challenges and solutions. Section 4 proposes the design and implementations of *EPMC* and its three key components. Section 5 evaluates the performance of *EPMC* architecture in DNN training. Section 6 concludes the paper.

2 Related work

Memory has become one of the most important bottlenecks in training as DNNs become larger and deeper, which significantly limits their deployments and applications. In view of this, many optimization approaches have been proposed (as shown in Table 1).

2.1 Model compression

This is a type of technique to reduce the model size for decreasing memory demand and computing requirement of increasingly deeper DNNs, including pruning [20, 27], quantization [15, 23, 28], low-rank approximation [24, 31], etc. However, these methods run the risk of losing model accuracy and usually require fine-tuning which causes additional time overhead to regain the lost performance.

Table 1 Comparison of the pros and cons of various memory optimization techniques

Characteristics of memory optimization approaches		
Model compression	Reduce storage for working data	Data compression
Pruning [20, 27] ①③, Quantization [15, 23, 28] ①⑤, Low-Rank Approximation [24, 31] ①⑤,	Reducing Batch Size ①②⑤, Recomputing [30] ①③, vDNN [13] ①③④	Weight Compression [15, 32–34] ①⑤, Gist [29] ①②
Main characteristics of this paper		
① No severe slowdown of training	✓	
② No GPU underutilization	✓	
③ No excessive additional energy consumption	✓	
④ No additional data transmission cost	✓	
⑤ No model accuracy loss	✓	

2.2 Storage reduction for working data

Reducing batch size can effectively solve the problem of memory shortage during training. However, this approach significantly increases the training time and slows down convergence of a model. Also, it lowers GPU utilization and risks losing accuracy [35]. Chen et al. designed an approach that cancels the storage of intermediate outputs in the forward pass and instead recalculates the output of a layer′s forward pass again in the backward pass [30]. This technique sacrifices computing for saving memory. However, it brings significant time and energy overhead. Rhu et al. proposed a potential approach that simultaneously utilizes CPU and GPU memory for DNN training and employs the PCIe links and intelligent prefetch analysis to transfer some working data between them for mitigating GPU memory pressure [13]. However, this approach suffers from obvious data transmission overhead, including energy, power consumption and a certain time cost.

2.3 Data compression

This type of technique is more relevant to our work. Han et al. proposed to quantize the link weights using weight-sharing and then employ Huffman coding to the quantized weights, as well as the codebook to optimize memory [15]. However, this method requires pruning the weights first, thus the network fine-tuning is inevitable. There is also a kind of aggressive lossy weight compression technology. For instance, Zhu et al. designed a novel quantization technology that reduces the precision of network weights to ternary values [32]. Binary weight network training is a more extreme lossy technology, such as Bi-Real Net [33] and IR-Net [34]. However, these approaches all focus on the weight and risk losing model accuracy. Jain et al. proved that feature maps are the major portion of memory usage in training and proposed to encode/decode them in the forward/backward pass for memory saving [29]. Unfortunately, encoding and decoding processes are relatively time-consuming and incur insufficient GPU utilization, resulting in a severe slowdown of training. However, the *Binarize* lossless encoding proposed in this work is still applicable to some specific layers (such as ReLU-Pool pair).

Although the method of encoding/decoding the feature maps mentioned above has some shortcomings, it is suitable for solving the insufficient memory of DNN training. And we find some opportunities to optimize its flaws, while also further optimizing the memory. Let us briefly introduce the optimization scenario. The gradient calculation of some layers requires their input/output in the forward pass. For example, convolution layer uses its input and ReLU layer uses its output. And the combination of convolution layer and ReLU layer can be seen everywhere in CNNs, referred to Conv-ReLU pair. As shown in Fig. 1, the feature maps output by the intermediate layer needs to complete two operations in the forward pass. On the one hand, they are the input to the calculation of the following layer. On the other hand, after the forward calculation, they are encoded into a smaller memory format for storing. When the backward computing reaches the corresponding layer, feature maps are decoded and then participate in gradient calculation of the layers related to them. From this scenario, we observe that the encoding/decoding process can be parallelized with specific layer to hide the latency overhead, and thus leads to memory saving with negligible performance overhead (see Fig. 1). We also observe that the combination of encoding and precision reduction can further reduce memory occupation without model accuracy loss. Based on these observations, we present *EPMC*, which aims to optimize the encoding and decoding of CSR

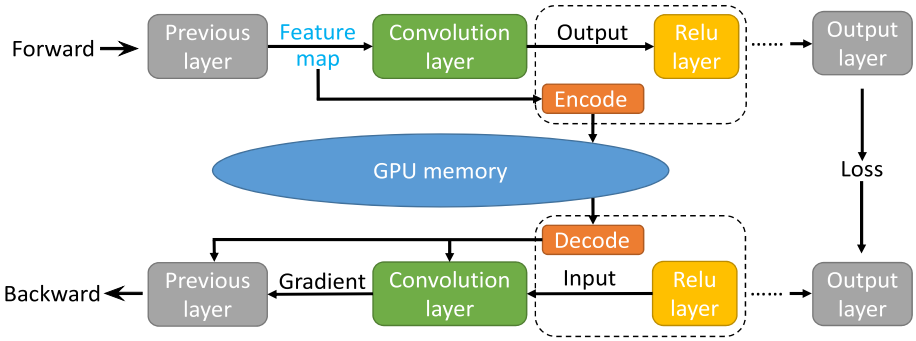

Fig. 1 A scheme of employing CSR-cuSPARSE in CNN training. Specifically, CSR-cuSPARSE encodes the feature maps in the forward pass and decodes them in the backward pass to participate in the gradient calculation, highlighting the parts that can be parallelized

(a common compression scheme) to achieve the above purposes.

3 Challenges and solutions

As mentioned in the previous section, there are huge opportunities to reduce memory consumption with negligible performance overhead. However, we also encounter significant challenges. In this section, we first describe these challenges and then show our considered solutions.

3.1 Challenges

The challenges we encounter mainly include three key points.

3.1.1 Expensive encoding/decoding overhead

Feature maps are typically stored and transmitted in high-dimensional tensor format. However, CSR compression algorithm in Nvidia cuSPARSE library (referred to CSR-cuSPARSE) can only deal with two-dimensional matrices, leading to inevitable computational overhead of dimension reduction. In addition, CSR-cuSPARSE cannot perform parallel operations between elements like ReLU. These problems make execution of CSR-cuSPARSE to be more time-consuming than ReLU. In Fig. 2, we show the time proportion occupied by each module in the network when employing CSR-cuSPARSE in the training of four CNNs. We can observe that for AlexNet [11], OverFeat [36], and VGG-16 [12], over 70% of training time is consumed by CSR-cuSPARSE compression. In Inception [37], this occupation is about 62%. Obviously, the encoding and decoding of CSR-cuSPARSE become a major reason of the severe slowdown in training.

3.1.2 Barriers to parallelize encoding/decoding

The data dependence between DNN layers in the forward and backward passes makes it difficult to separate

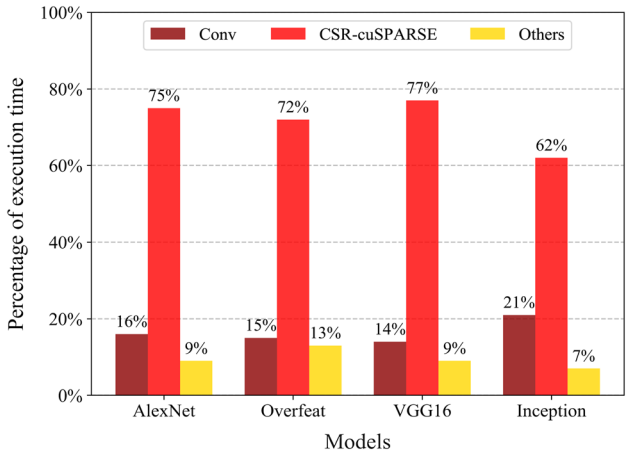

Fig. 2 The proportion of time consumption by each module when applying CSR-cuSPARSE in DNN training. CSR-cuSPARSE consumes a major portion of the training time

encoding/decoding from training. In addition, the execution time of encoding/decoding is several times that of other layers (such as convolution layer) when they have the same input, which is also not conducive to implement parallelization. The reason is that the best performance improvement can only be achieved when the time consumption of each parallel task is close. GPU utilization is also one of the necessary conditions to be considered for parallelism.

3.1.3 Memory overhead of high-precision storage

Applying high-precision working data can retain more information and achieve higher model accuracy. However, it inevitably brings serious memory overhead. For example, as shown in Table 2, training of VGG-16 with a batch size of 128 consumes 16.07 GB GPU memory in total (i.e., 1.54GB for model storage and 14.53 GB for storing layer outputs), which exceeds memory size of an Nvidia 1080Ti Card. To solve this, batch size has to be reduced or more GPU cards are required, which indirectly increases training overhead and cost. In addition, reckless precision reduction in the working data in the network, such as the unified

Table 2 Memory usage of VGG-16 training

Type	Module	Memory usage
Model	Memory for params	528 MB
	Memory for SGD	528 MB
	Memory for momentum	528 MB
Layer outputs	Memory for SGD	14.53 GB
Total		16.07 GB

modification of all data structures to a certain format, results in severe accuracy loss.

3.2 Solutions

Though challenges exist, we find many key points that can be optimized and developed to solve the challenges.

3.2.1 PipeCompress

To solve the first challenge, we present an efficient encoding/decoding scheme. When applying CSR-cuSPARSE to deal with the high-dimensional feature maps, it is inevitable to perform multiple loop iterations in the encoding and decoding processes. As a result, CSR-cuSPARSE is called multiple times, and the data processed each time becomes smaller, which is not conducive to efficiency. To solve this problem, we add a data processing step before encoding and decoding. In the forward pass, we flatten the feature map from a high-dimensional tensor to a two-dimensional matrix before encoding it. In the backward pass, the encoded feature map is decoded into a two-dimensional matrix and then restored to the original high-dimensional tensor. This step allows CSR-cuSPARSE to be performed only once when encoding/decoding the feature maps of a layer.

We employ CSR-cuSPARSE with the data flatten operation in VGG-16 [12] training under different batch sizes and observe its GPU utilization (see Fig. 3). Unfortunately, CSR-cuSPARSE leads to a serious GPU underutilization under a small batch size, wasting about 81% of computing resources. And its GPU utilization is still insufficient even at larger batch size. However, this means that we can employ the unused computing resources to implement parallel optimization.

CSR-cuSPARSE encodes a sparse matrix into three vectors, which record the value, column index, and row offset of nonzero elements, respectively. The storage order among elements in the CSR encoding format (referred to CSR-format) makes encoding/decoding in a row-by-row serial manner, making it difficult to implement parallelization between elements. Therefore, we design to split

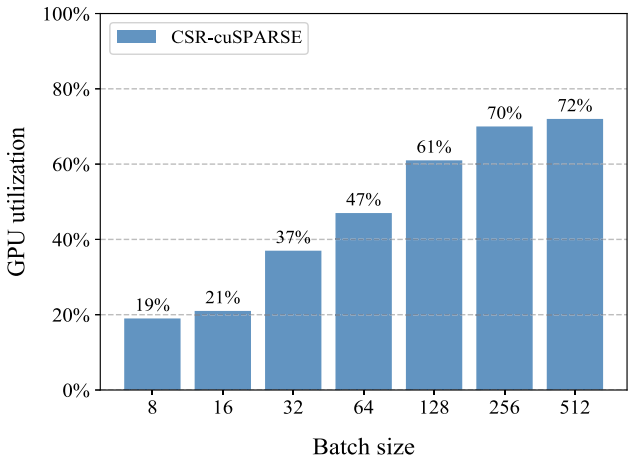

Fig. 3 The GPU utilization of CSR-cuSPARSE when employing it in VGG-16 [12] training under different batch sizes. The ratio increases with the growth of batch size. However, it has an upper limit

the input by row and parallelize the processing to achieve efficient encoding and decoding. For instance, we divide a matrix into several equal parts by row and encode/decode each part in parallel to cover the waiting time between their processing.

Based on these three operations: data flattening, segmentation, and parallel processing, we propose the first solution, referred to as PipeCompress.

3.2.2 Parallel strategy

To solve the second challenge, we propose a parallel scheme. In Fig. 1, we show the position where CSR-cuSPARSE encodes and decodes the feature maps in the Conv-ReLU pair. The execution order of optimizable modules in the forward pass is convolution layer → CSR-cuSPARSE → ReLU layer. Their execution order in the backward pass is ReLU layer → CSR-cuSPARSE → convolution layer. In the forward pass, CSR-cuSPARSE encodes the feature maps output by the previous layer, and ReLU deals with the output of the convolution layer. In the backward pass, the encoded feature maps need to participate in the gradient calculation of the convolution layer and the previous layer (if necessary). Therefore, they only need to be decoded before that. The gradient calculation of the ReLU layer uses the feature maps output by its forward calculation. As a result, there is no data dependence between CSR-cuSPARSE and ReLU in either forward or backward pass.

Moreover, we observe that CSR-cuSPARSE and ReLU have the same inputs both in forward and backward propagation.

Based on these characteristics, we can utilize the unused GPU computing resources to execute ReLU layer during encoding/decoding without any impact, leading to

significant efficiency improvement. This becomes the basis of our parallel strategy.

3.2.3 VPEncoding

In order to solve the third challenge, we propose an effective encoding scheme that can reduce the memory footprint of the feature maps through precision reduction without affecting the model accuracy. We observe that the feature maps are computed and stored in FP32 format. In addition, when the feature maps are encoded into the CSR-format, the data precision of nonzero element vector is also FP32. And the data precision of the column-index vector and row-offset vector is fixed to INT32 format.

In view of this, we investigate some effective technologies of precision reduction used in DNN training, such as the selective precision reduction proposed by De Sa *et al* in [38] and the delayed precision reduction developed by Jain *et al* in [29]. They have proved that DNNs have a certain tolerance for low-precision data in training. We also discover that CSR-format fixes the data precision of three vectors only for conveniently applying. However, this is not friendly for memory saving. When training the model under small batch size, the data to be processed are small in each iteration. When encoding the feature maps into CSR-format, the element values in the column-index vector and row-offset vector are not very large and non-negative. Therefore, data format with a smaller memory footprint can meet their storage requirements. In addition, the low precision tolerance of training also allows us to further reduce the data precision of nonzero element vector. It is worth noting that we do not modify the precision of the feature maps that need to be used in the forward calculation. This prevents data errors from being propagated in the forward pass, thus ensuring the model accuracy.

Based on these observations, we present variable precision encoding, called VPEncoding, which can modify the precision of encoded data in line with storage requirements to further save memory.

4 Designs and implementations

We design specific implementations based on the above solutions, which are introduced as follows.

4.1 PipeCompress

In Fig. 4, we show the entire data compression process. First, in the forward pass, we flatten the high-dimensional input tensor into a two-dimensional matrix and limit the column number to be less than 256. Then, we divide this matrix into k equal parts by row. Second, as shown in Fig. 5b, we use the multi-stream technology [39] in CUDA to create k streams and put each part of the data into different streams for processing. There are copy operations of input/output data before and after the task execution in a stream. The data copy operations in different streams cannot be performed in parallel. However, the copy operations and the CSR-cuSPARSE kernel functions among different streams can be parallelized. The CSR-cuSPARSE kernel functions in different streams can also be parallelized. Based on these characteristics, the compression process is designed as pipeline parallelism to cover the waiting time between adjacent operations and to achieve performance improvement (see Fig. 5b). Third, we integrate the output data of all streams.

We rewrite the data compression algorithm to achieve the pipeline parallel optimization and the data integration of all streams. When all streams are completed, the output data are spliced in order. We record the length and the last element of row offset vector output by each stream. The encoded data can be divided according to these records and decoded through k streams in the backward pass.

4.2 Specific-layer parallelism

As introduced in the second solution in Sect. 3.2, no data dependency and GPU underutilization enable us to execute CSR-cuSPARSE and ReLU simultaneously. We observe that the time consumption of encoding in CSR-cuSPARSE is higher than the forward computing of ReLU. Similarly, the time consumption of decoding in CSR-cuSPARSE is also higher than gradient calculation of ReLU. In addition, ReLU obtains good parallel support in the deep learning framework. These lead to huge differences in execution efficiency among them. PipeCompress narrows the elapsed time difference between them.

We employ multi-stream technology [39] in CUDA to achieve dynamical allocation of GPU computing resources and parallelization. As shown in Fig. 6, we focus on two key points in the forward and backward passes. In the forward pass, the encoding of PipeCompress is parallelized with the forward calculation of ReLU. In the backward pass, the decoding of PipeCompress and the gradient calculation of the ReLU are performed in parallel.

We formalize our parallel problem as follows. Let $T(N)$ be the parallel execution time of the two tasks, where N is their shared GPU computing resources. We have

$$T(N) = \max(T_1(n), T_2(N-n)), \tag{1}$$

where $T_1(n)$ is the execution time of PipeCompress with n denoting the number of computing resources allocated to PipeCompress, and $T_2(N-n)$ is the execution time of ReLU. $T(N)$ is determined by the execution time of the slower task. Through allocating an appropriate amount of

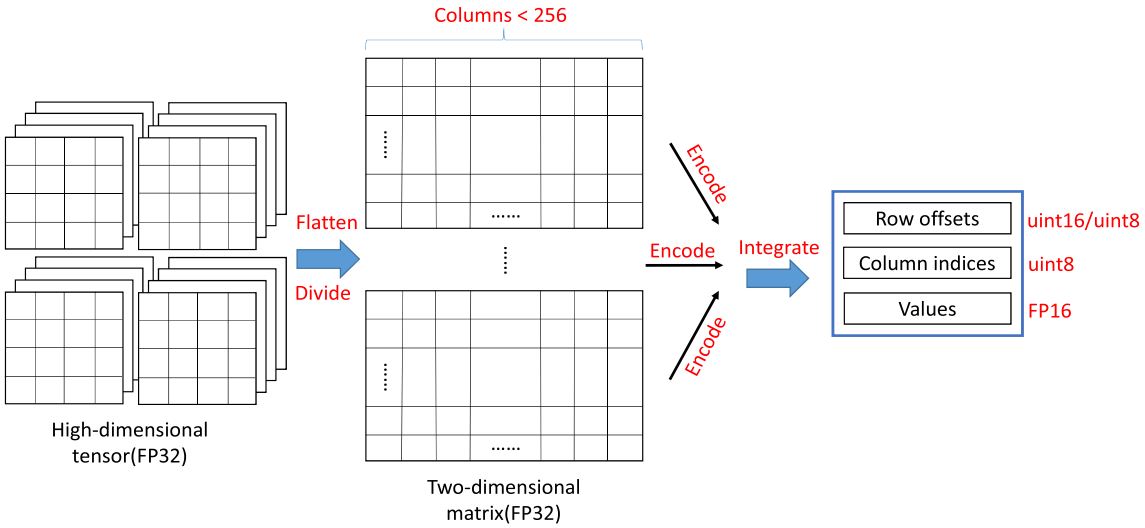

Fig. 4 An example of efficient encoding. This example shows the processes of flattening, splitting, encoding parallelization, integrating, and precision reduction in the input data

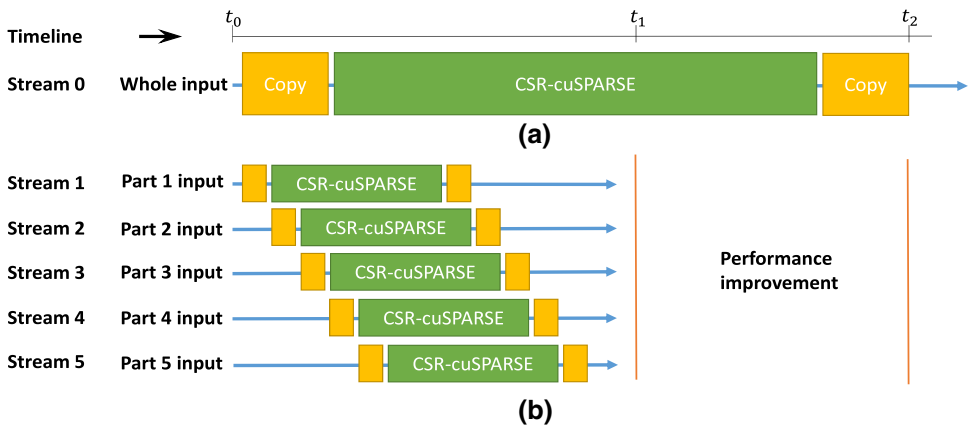

Fig. 5 **a** An example of a CSR-cuSPARSE compression without any optimization. **b** An example of a data compression process using five streams for pipeline parallel optimization, highlighting the performance improvement. The input data are accordingly divided into five parts

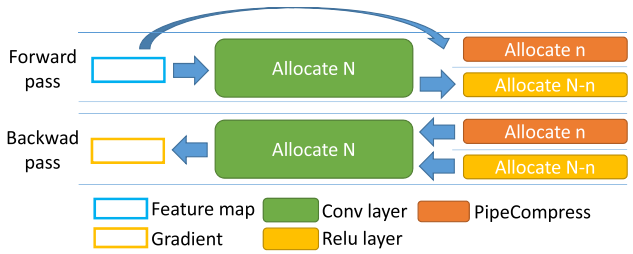

Fig. 6 In the forward and backward passes, the PipeCompress is executed in parallel with specific ReLU through multitask parallelism

computing resources to each of them, the execution times of $T_1(n)$ and $T_2(N-n)$ are as close as possible, thereby obtaining an approximate optimal solution for parallel execution.

Based on the above scheme, we built a parallel framework. In each Conv-ReLU pairs, PipeCompress is inserted into training and executed in parallel with ReLU. The total number of streams is determined according to the training and GPU parameters, denoted as m. The dynamical allocation of computing resources we employ is derived from the Hyper-Q technology supported by Nvidia′s GPUs of the Kepler architecture [40]. In addition, our framework allocates computing resources to each stream in line with the computational complexity of the tasks in them. Specifically, let N be the total amount of computing resources shared by the ReLU and PipeCompress. We assign n and $N-n$ to PipeCompress and ReLU, respectively. In addition, we assign $m-1$ streams to execute PipeCompress. In each stream of pipeline parallelism in PipeCompress, the allocated input data are $\frac{1}{m-1}$ of the original input, and the allocated computing resources are $\frac{n}{m-1}$.

4.3 VPEncoding

When encoding the feature maps into CSR-format during DNN training, we reduce the data precision of the nonzero element vector from FP32 to a smaller data format (such as

FP16). In addition, when the column number of input matrix is less than 256, we can use UINT8 format instead of INT32 format for data storage in the column-index vector. The data format of the row-offset vector can also be modified from INT32 to UINT16 and can even be set to UINT8 format when the batch size is smaller.

4.4 Efficient parallel memory compression: EPMC

Based on the above three schemes, we design the framework of *EPMC*, as shown in Fig. 7. Typically, DNN frameworks such as Pytorch/TensorFlow orderly integrate all layers of a network into a directed computation graph. According to the original computation graph, GPU performance parameters, and the DNN training parameters, the *EPMC*'s scheduler generates a new computation graph, which employs our proposed three optimization components (PipeCompress, VPEncoding, and Specific-Layer Parallelism) to efficiently solve memory problem.

5 Experimental evaluation

In this section, we provide extensive experiments to show the performance of our framework.

5.1 Parameter configuration

5.1.1 Platforms

We evaluate *EPMC*'s ability of to reduce memory footprint and its impact on DNN training in Pytorch deep learning framework. The evaluation is performed on an Nvidia Kepler Tesla K40c card [41] with 12 GB of GDDR5 memory using CUDA9.0 and cuDNN v7.1 and an Nvidia GeForce GTX 1070 card [42] with 8 GB of GDDR5 memory using CUDA10.1 and cuDNN v7.6 (Table 3).

5.1.2 Applications

We evaluate *EPMC* on four state-of-the-art image classification CNNs: AlexNet [11], OverFeat [36], VGG-16 [12], and Inception [37], employing ImageNet training dataset [43]. These CNNs exhibit a wide range of layer sizes and shapes and also represent the development of CNNs over the past few years.

5.1.3 Baselines

Our first baseline is called *Pytorch baseline*, which is the original memory allocation strategy of the Pytorch framework without any of our optimization. This baseline is used to evaluate the memory footprint reduction in *EPMC* and study the impact of our framework on training and accuracy of the model. The feature maps that need to be stored and participate in the gradient calculation have been proved to be the major reason for the excessive memory occupation [29], thus Pytorch baseline consists of only feature maps. It does not include weights, gradients, etc.

The second baseline is to serially insert CSR-cuSPARSE in Pytorch baseline to encode/decode feature maps, called *serial baseline*, which is used to compare the memory compression ratio of our framework and evaluate the effect of our specific-layer parallel strategy.

5.1.4 Comparison metrics

We use Memory Footprint Ratio (MFR) to evaluate the memory footprint reduction in our framework. In addition, we use execution time ratio (ETR) to evaluate the performance improvement of our proposed PipeCompress and specific-layer parallel strategy. MFR and ETR are defined as follow:

$$\mathrm{MFR} = \frac{\text{Memory Footprint of Baseline}}{\text{Memory Footprint after Encoding}}, \tag{2}$$

$$\mathrm{ETR} = \frac{\text{Execution Time after optimization}}{\text{Execution Time of Baseline}}. \tag{3}$$

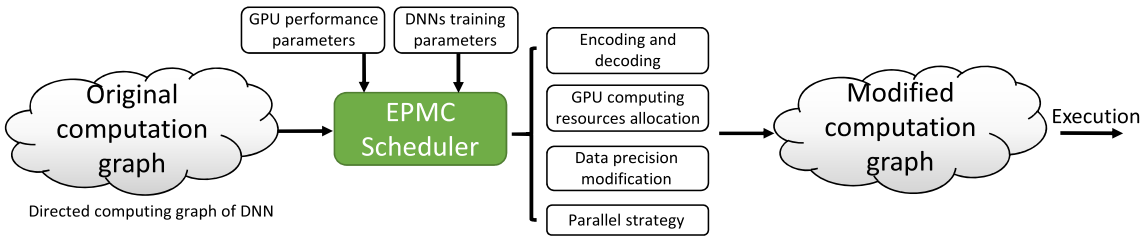

Fig. 7 Architecture of *EPMC* – Scheduler identifies where the PipeCompress can be employed. And it allocates computing resources and builds parallel frameworks for layers according to their computational complexity

Table 3 Experimental infrastructure

Experimental infrastructure		
GPU Card	Architecture	Configuration
K40C [41]	Tesla	12 GB GDDR5 memory, CUDA9.0, cuDNN v7.1.
GTX 1070 [42]	Pascal	8 GB GDDR5 memory, CUDA10.1, cuDNN v7.6.
Baseline	Application	
Pytorch baseline	Evaluate the memory footprint reduction and model accuracy.	
Serial baseline	Evaluate parallel strategy and memory compression ratio.	

5.2 Performance of EPMC

Our framework *EPMC* is to reduce the memory footprint of feature maps and minimize the impact of encoding/decoding on DNN training. In this section, we evaluate these indicators separately. In addition, we also present the impact of applying *EPMC* optimization on the convergence and accuracy of models. The experimental results are shown as follows.

5.2.1 Memory footprint reduction

In Fig. 8, we show the Memory Footprint Ratio (MFR) achieved by *EPMC* and serial baseline optimizations when compared to Pytorch Baseline. Feature maps in different networks have varied sparsity. Therefore, the MFR values are also different. We observe that the serial baseline results in a MFR of about 1.4 times on average for the four CNNs, and *EPMC* achieves a MFR of up to 2.6 times for VGG-16 [12], with an average of 2.3 times. This experiment shows that *EPMC* brings more significant memory saving compared to serial baseline, which enables us to train a larger network with a larger batch size in the single-GPU environment, thereby indirectly accelerating the model convergence.

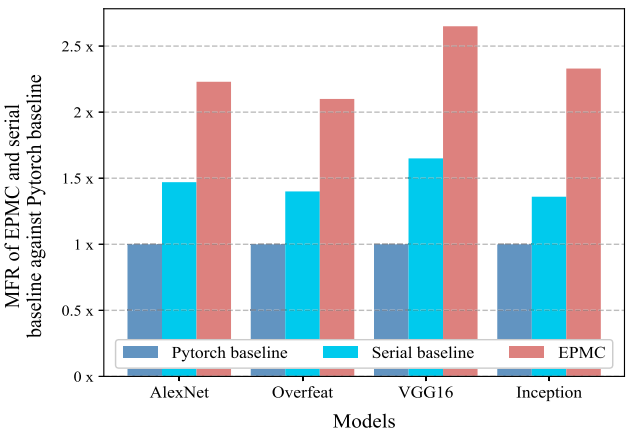

Fig. 8 Evaluation of memory footprint reduction. *EPMC* significantly optimizes the memory usage

5.2.2 Performance overhead

We train the networks that employ our framework for memory optimization (shown as *EPMC*). Pytorch baseline is utilized as the basis for comparison to evaluate the efficiency of *EPMC* and serial baseline. As shown in Fig. 9, serial baseline results in about 3.3 times Execution Time Ratio (ETR) for AlexNet [11] and OverFeat [36], and it even reaches up to 4.5 times ETR for VGG-16 [12] and Inception [37]. This means that serial application of CSR-cuSPARSE compression scheme leads to severe slowdown of DNN training. However, when using our framework *EPMC* to optimize memory, the impact on the DNN training is greatly reduced. *EPMC* results in only about 1.5 times ETR for AlexNet [11] and OverFeat [36]. For VGG-16 [12] and Inception [37], it reaches 1.7 times ETR on average. In other words, it brings up to 2.1 times performance improvement compared with serial baseline.

5.2.3 Impact on convergence and accuracy

In VPEncoding of our framework, we separately represent the data structure in nonzero element vector in three formats, FP16, FP10, and FP8. We find that when FP10 and FP8 are applied, the networks suffer varying degrees of accuracy losses, while FP16 does not. Therefore, in following experiment, we represent nonzero element vector, column-index vector, and row-offset vector with FP16, UINT8, and UINT16, respectively. Figure 10 shows the training accuracy for different networks with memory optimization (shown as *EPMC*) and without memory optimization (shown as Pytorch baseline). Training accuracy curve is a way to observe the convergence and accuracy of model over time. At the beginning, the accuracy of the network is almost 0%. And then, it continues to improve over time. For example, at 90-th epoch, VGG-16 [12] achieves 81% training accuracy. The figure compares the deviation between the accuracy of Pytorch baseline (red

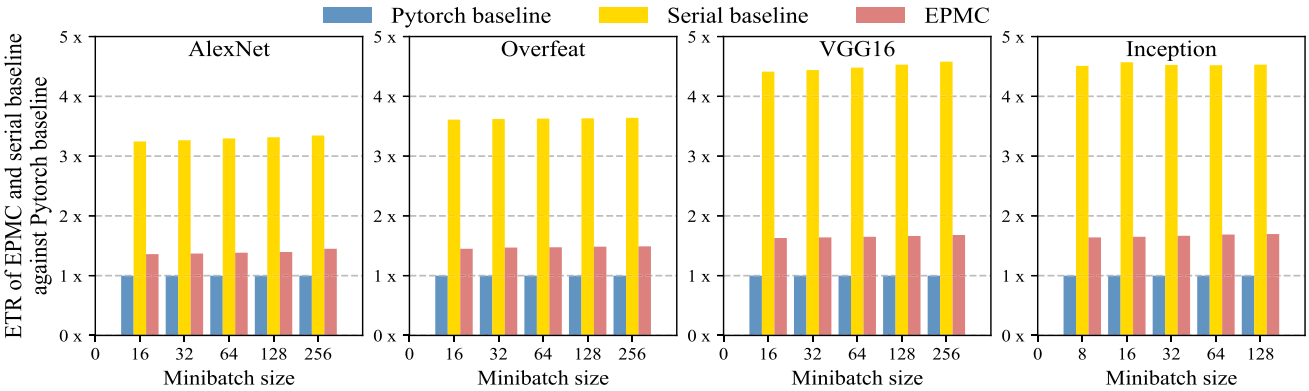

Fig. 9 Impact of *EPMC* on training. Compared with serial baseline, our framework greatly improves the efficiency and reduces the impact of encoding/decoding on training

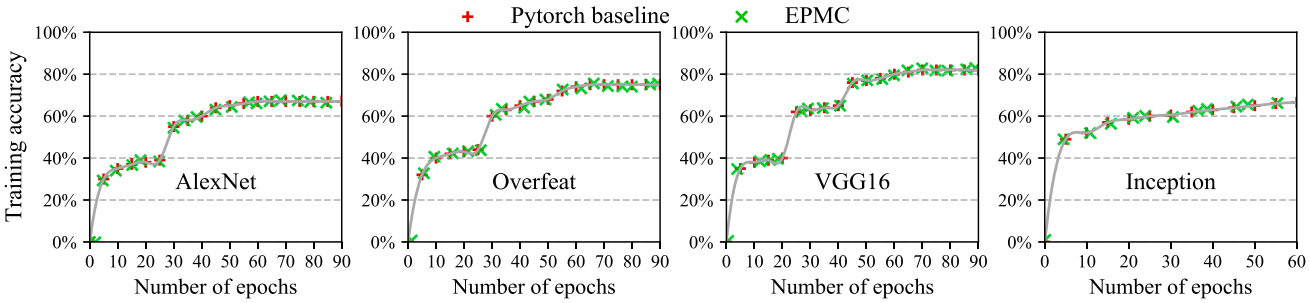

Fig. 10 Impact of *EPMC* on the convergence and accuracy of model

+) and *EPMC* (green x) as the training epoch increases. As shown in Fig. 10, the two curves overlap, which means that our framework has no effect on the convergence and accuracy of models. The reason is that what we reduce is the precision of the feature maps employed for the gradient calculation in the backward pass, and FP16 has enough precision to meet the requirements of gradient computation.

5.3 PipeCompress

In this section, we evaluate the performance improvement of our proposed pipeline parallel optimization (PipeCompress) and how much ETR it achieves when compared to CSR-cuSPARSE. In the experiment, we use 50% sparsity inputs with different sizes for evaluation. We set the inputs as four sets of matrix data with volumes 2000, 8000, 32000, and 128000, respectively. As shown in Fig. 11, when the input size is relatively small (such as 2000, 8000, and 32000), the ETR increases linearly with the increase in stream number. When setting seven streams in PipeCompress, it results in up to 4.5–5 times ETR. When the input size is large, the ETR it reaches no longer increases even if the stream number grows. We argue that the reason is the limitation of the number of GPU cores and bandwidth, leading to competition for computing resources between streams.

5.4 Specific-layer parallel strategy

In this section, we evaluate the efficiency improvement brought by our specific-layer parallel strategy and calculate its ETR when compared to the serial baseline. The input data are the same as the setting in Sect. 5.3. In parallel strategy, appropriate computing resources are allocated to the two algorithms according to the approach in Sect. 4.2. In PipeCompress, we use seven streams to implement pipeline parallelism. Figure 12 shows that the PipeCompress-ReLU parallel strategy reaches close to 0.7 times ETR under different input sizes, which means that it brings an average 30% (100–70 %) performance promotion compared to the serial strategy. We can observe that the performance increment slowly reduces as the input size increases. We argue that the reason is that the available GPU computing resources are limited, which sets an upper limit on the efficiency of parallel strategy.

5.5 Variable precision encoding

Finally, we evaluate the effect of VPEncoding in reducing memory usage. In the experiment, VPEncoding and other

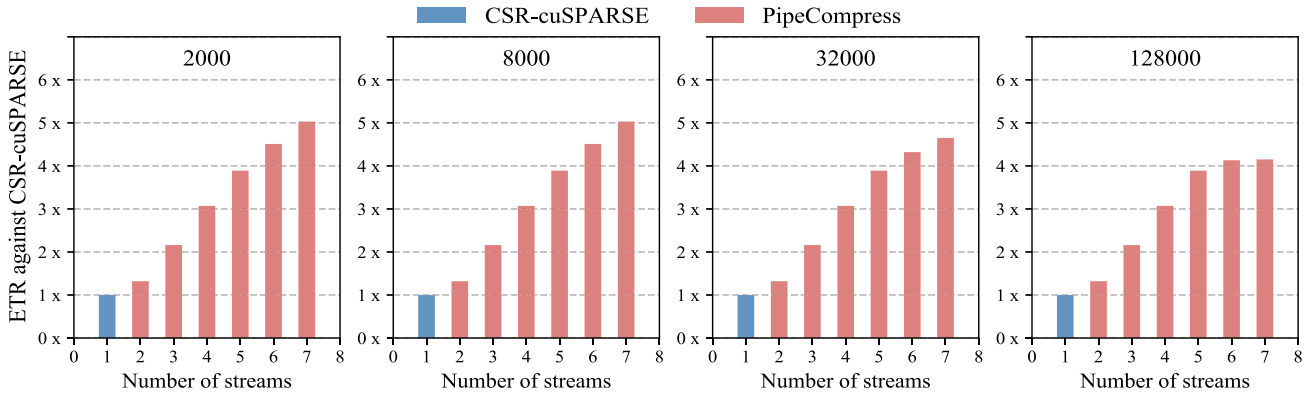

Fig. 11 ETR of PipeCompress when compared to the unoptimized CSR-cuSPARSE under different input sizes and different numbers of streams. PipeCompress achieves a significant acceleration as the number of streams increases

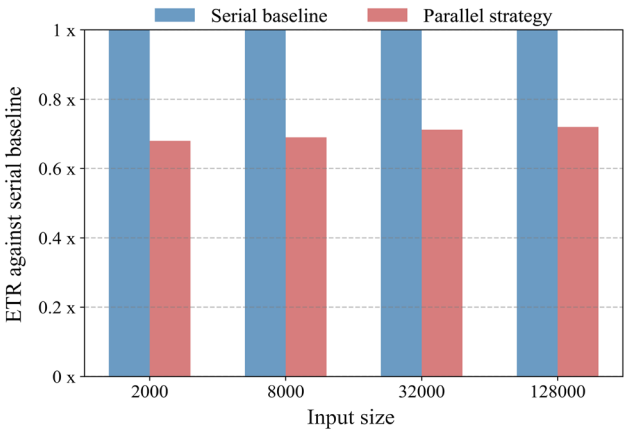

Fig. 12 ETR when PipeCompress and ReLU are executed serially and in parallel. The parallel strategy shortens their execution time by nearly 30%

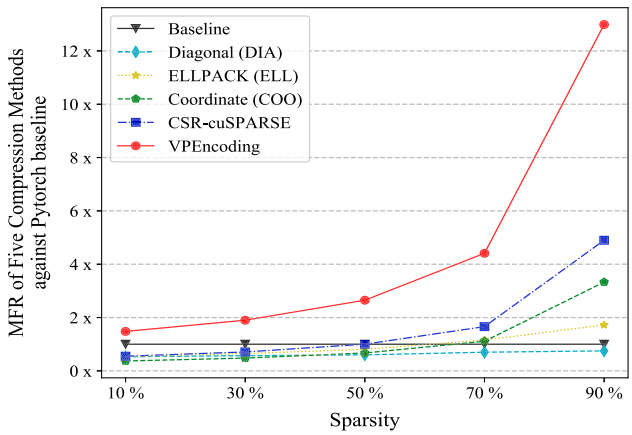

Fig. 13 MFR of VPEncoding and CSR-cuSPARSE under different sparsity input. The memory footprint reduction in VPEncoding is about two times that of the CSR-cuSPARSE

four traditional compression methods (i.e., Coordinate (COO), Compressed Sparse Row (CSR), ELLPACK (ELL), and Diagonal (DIA)), are employed to encode feature maps with different sparsity, and their compression ratios are shown as MFR when compared to the Pytorch baseline.

Figure 13 shows that the compression ratio of VPEncoding significantly higher than other four methods under different sparsity. DIA is mainly aimed at compressing data with sparse diagonal, and feature maps usually have non-uniform sparsity, so this method has a poor compression effect on feature maps. CSR-cuSPARSE can reduce the memory footprint only when the sparsity of the input is higher than 50%. ELL and COO can really compress data only when the sparsity exceeds 70%. However, our VPEncoding can reach close to 1.5 times MFR when the input sparsity is only 10%, which reduces the application threshold of the data compression algorithm. In addition, VPEncoding produces up to 13 times MFR when the input sparsity reaches 90%. The data compression ratio of VPEncoding is approximately 2.2 times that of CSR-cuSPARSE on average.

6 Conclusion

In this paper, we investigate a variety of the memory optimization schemes and present a framework, *EPMC*, which enables us to simultaneously reduce the memory footprint and the impact of encoding/decoding on DNN training. We evaluate *EPMC* with four state-of-the-art image classification DNNs. Experimental results show that *EPMC* can reduce the memory footprint during training to 2.3 times on average without accuracy loss. In addition, it can reduce the DNN training time by more than 2.1 times on average compared with the unoptimized encoding/decoding scheme. Moreover, compared with the CSR-cuSPARSE, *EPMC* can achieve data compression ratio by 2.2 times.

Acknowledgements The research is partially supported by the Program of National Natural Science Foundation of China (Grant Nos. 62072165, U19A2058), Open Research Projects of Zhejiang Lab (No. 2020KE0AB01), and the Fundamental Research Funds for the Central Universities.

References

1. LeCun Y, Bengio Y, Hinton G (2015) Deep learning. Nature 521(7553):436–444
2. Goodfellow I, Bengio Y, Courville A, Bengio Y (2016) Deep learning, vol 1. MIT press Cambridge, Cambridge
3. Bojarski M, Del Testa D, Dworakowski D, Firner B, Flepp B, Goyal P, Jackel LD, Monfort M, Muller U, Zhang J, et al., End to end learning for self-driving cars, arXiv preprint arXiv:1604.07316
4. Sun Y, Wang X, Tang X (2013) Deep convolutional network cascade for facial point detection. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 3476–3483
5. Weimer D, Scholz-Reiter B, Shpitalni M (2016) Design of deep convolutional neural network architectures for automated feature extraction in industrial inspection. CIRP Ann 65(1):417–420
6. Kalchbrenner N, Grefenstette E, Blunsom P, A convolutional neural network for modelling sentences, arXiv preprint arXiv:1404.2188
7. Karpathy A, Toderici G, Shetty S, Leung T, Sukthankar R, Fei-Fei L (2014) Large-scale video classification with convolutional neural networks. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 1725–1732
8. Vinyals O, Toshev A, Bengio S, Erhan D (2015) Show and tell: a neural image caption generator. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 3156–3164
9. LeCun Y, et al (2015) Lenet-5, convolutional neural networks. URL: http://yann.lecun.com/exdb/lenet 20(5): 14
10. He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770–778
11. Krizhevsky A, Sutskever I, Hinton GE (2012) Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems, pp. 1097–1105
12. Simonyan K, Zisserman A, Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556
13. Rhu M, Gimelshein N, Clemons J, Zulfiqar A, Keckler S (2016) vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. Memory-Efficient Neural Network Design, MICRO-2016
14. Rhu M, O'Connor M, Chatterjee N, Pool J, Kwon Y, Keckler SW, Compressing dma engine: leveraging activation sparsity for training deep neural networks. In: (2018) IEEE International symposium on high performance computer architecture (HPCA). IEEE 2018:78–91
15. Han S, Mao H, Dally WJ, Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149
16. He Y, Lin J, Liu Z,Wang H, Li L-J, Han S (2018) Amc: automl for model compression and acceleration on mobile devices. In: Proceedings of the European conference on computer vision (ECCV), pp. 784–800
17. Guo S, Wang Y, Li Q, Yan J (2020) Dmcp: Differentiable markov channel pruning for neural networks. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. pp. 1539–1547
18. Li Y, Gu S, Mayer C, Gool LV, Timofte R (2020) Group sparsity: the hinge between filter pruning and decomposition for network compression. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. pp. 8018–8027
19. Lin M, Ji R, Wang Y, Zhang Y, Zhang B, Tian Y, Shao L (2020) Hrank: filter pruning using high-rank feature map. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. pp. 1529–1538
20. Liu Z, Mu H, Zhang X, Guo Z, Yang X, Cheng K-T, Sun J (2019) Metapruning: Meta learning for automatic neural network channel pruning. In: Proceedings of the IEEE international conference on computer vision, pp. 3296–3305
21. Neill JO, An overview of neural network compression. arXiv preprint arXiv:2006.03669
22. An overview of model compression techniques for deep learning in space. https://medium.com/gsi-technology/an-overview-of-model-compression-techniques-for-deep-learning-in-space-3fd8d4ce84e5 (2020)
23. Xu Y, Wang Y, Zhou A, Lin W, Xiong H (2018) Deep neural network compression with single and multiple level quantization. In: Proceedings of the AAAI conference on artificial intelligence. 32
24. Kim H, Khan MUK, Kyung C-M (2019) Efficient neural network compression. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. pp. 12569–12577
25. Ge S (2018) Efficient deep learning in network compression and acceleration. In: Digital systems, IntechOpen
26. Paupamah K, James S, Klein R (2020) Quantisation and pruning for neural network compression and regularisation. In: International SAUPEC/RobMech/PRASA Conference. IEEE 2020:1–6
27. Jin S, Di S, Liang X, Tian J, Tao D, Cappello F (2019) Deepsz: A novel framework to compress deep neural networks by using error-bounded lossy compression. In: Proceedings of the 28th international symposium on high-performance parallel and distributed computing. pp. 159–170
28. Kozlov A, Lazarevich I, Shamporov V, Lyalyushkin N, Gorbachev Y, Neural network compression framework for fast model inference. arXiv preprint arXiv:2002.08679
29. Goyal P, Dollár P, Girshick R, Noordhuis P, Wesolowski L, Kyrola A, Tulloch A, Jia Y, He K, Accurate, large minibatch sgd: Training imagenet in 1 hour. arXiv preprint arXiv:1706.02677
30. Jain A, Phanishayee A, Mars J, Tang L, Pekhimenko G (2018) Gist: Efficient data encoding for deep neural network training. In: ACM/IEEE 45th annual international symposium on computer architecture (ISCA). IEEE 2018:776–789
31. Chen T, Xu B, Zhang C, Guestrin C, Training deep nets with sublinear memory cost. arXiv preprint arXiv:1604.06174
32. Wen W, Xu C, Wu C, Wang Y, Chen Y, Li H (2017) Coordinating filters for faster deep neural networks. In: Proceedings of the IEEE international conference on computer vision. pp. 658–666
33. Zhu C, Han S, Mao H, Dally WJ, Trained ternary quantization. arXiv preprint arXiv:1612.01064
34. Liu Z, Wu B, Luo W, Yang X, Liu W, Cheng K-T (2018) Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm. In: Proceedings of the European conference on computer vision (ECCV), pp. 722–737
35. Qin H, Gong R, Liu X, Shen M, Wei Z, Yu F, Song J (2020) Forward and backward information retention for accurate binary neural networks. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. pp. 2250–2259
36. Sermanet P, Eigen D, Zhang X, Mathieu M, Fergus R, LeCun Y, Overfeat: Integrated recognition, localization and detection using convolutional networks. arXiv preprint arXiv:1312.6229

37. Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, Erhan D, Vanhoucke V, Rabinovich A (2015) Going deeper with convolutions. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 1–9
38. De Sa C, Feldman M, Ré C, Olukotun K (2017) Understanding and optimizing asynchronous low-precision stochastic gradient descent. In: Proceedings of the 44th annual international symposium on computer architecture, pp. 561–574
39. Cheng J, Grossman M, McKercher T (2014) Professional CUDA c programming. Wiley, Hoboken
40. Nvidia kepler architecture (2012) https://www.nvidia.cn/content/apac/pdf/tesla/nvidia-kepler-gk110-architecture-whitepaper-cn.pdf
41. Nvidia tesla k40c (2013) https://www.techpowerup.com/gpu-specs/tesla-k40c.c2505
42. Nvidia geforce gtx 1070 (2016) https://www.nvidia.com/en-in/geforce/products/10series/geforce-gtx-1070/
43. Russakovsky O, Deng J, Su H, Krause J, Satheesh S, Ma S, Huang Z, Karpathy A, Khosla A, Bernstein M et al (2015) Imagenet large scale visual recognition challenge. Int J Comput Vision 115(3):211–252