Layerwise Sparse Coding for Pruned Deep Neural Networks with Extreme Compression Ratio

Xiao Liu,[1] **Wenbin Li,**[1] **Jing Huo,**[1] **Lili Yao**[1] **Yang Gao**[1*]
[1]National Key Laboratory for Novel Software Technology, Nanjing University, China
liuxiao730@outlook.com, liwenbin.nju@gmail.com, yaolili93@163.com, {huojing, gaoy}@nju.edu.cn

Abstract

Deep neural network compression is important and increasingly developed especially in resource-constrained environments, such as autonomous drones and wearable devices. Basically, we can easily and largely reduce the number of weights of a trained deep model by adopting a widely used model compression technique, *e.g.,* pruning. In this way, two kinds of data are usually preserved for this compressed model, *i.e., non-zero weights* and *meta-data*, where meta-data is employed to help encode and decode these non-zero weights. Although we can obtain an ideally small number of non-zero weights through pruning, existing sparse matrix coding methods still need a much larger amount of meta-data (may several times larger than non-zero weights), which will be a severe bottleneck of the deploying of very deep models. To tackle this issue, we propose a *layerwise sparse coding (LSC)* method to maximize the compression ratio by extremely reducing the amount of meta-data. We first divide a sparse matrix into multiple small blocks and remove zero blocks, and then propose a novel *signed relative index* (SRI) algorithm to encode the remaining non-zero blocks (with much less meta-data). In addition, the proposed LSC performs parallel matrix multiplication without full decoding, while traditional methods cannot. Through extensive experiments, we demonstrate that LSC achieves substantial gains in pruned DNN compression (*e.g.,* 51.03x compression ratio on ADMM-Lenet) and inference computation (*i.e.,* time reduction and extremely less memory bandwidth), over state-of-the-art baselines.

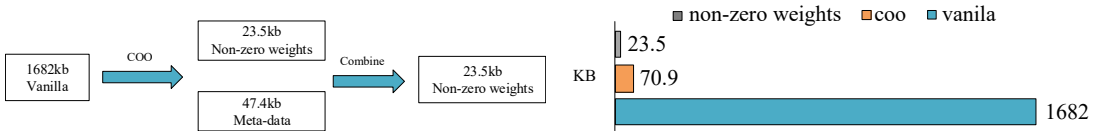
1 Introduction

Deep neural networks (DNNs), especially the very deep networks, have evolved to be the state-of-the-art techniques in many fields, particularly in computer vision, natural language processing, and audio processing (Krizhevsky, Sutskever, and Hinton 2012; Sutskever, Vinyals, and Le 2014; Abdel-Hamid et al. 2014). However, the huge growth numbers of hidden layers and neurons, which consume considerable hardware storage and memory bandwidth, posing significant challenges to many resource-constrained scenarios in real-world applications. In particular, the arrival of the *post-Moore era* slows down the hardware replacement cycle (Hu 2018). Specifically, there are two main bottlenecks of the current DNNs:

- Conflict with energy-constrained application platforms, such as autonomous drones, mobile phones, mobile robots and augmented reality (AR) in the daily life (Krajník et al. 2011; Floreano and Wood 2015; Kamilaris and Prenafeta-Boldú 2018). These application platforms are very sensitive to the energy consumption and computational workload of the DNN models. Therefore, DNN models with low energy consumption but good performance are urgently needed.
- Conflict with new accelerators, such as FPGA, custom ASIC and AI dedicated chips (Chen et al. 2014; Zhang et al. 2015; Han et al. 2016). They are powerful computing accelerators for DNNs, but are also sensitive to hardware storage, memory bandwidth, and parallelism. Obviously, DNNs are expected to be able to reduce the usage of hardware storage and memory while enjoying the ability of parallelism.

To tackle the above bottlenecks, a lot of compression methods have been proposed, such as pruning (Han et al. 2015; Anwar, Hwang, and Sung 2017) and quantization (Han, Mao, and Dally 2016; Park, Ahn, and Yoo 2017). Adopting these efficient compression methods, we can easily reduce the number of weights of a trained DNN model. For instance, using the classic magnitude-based pruning (Han et al. 2015), we can make the weight matrices of the target network very sparse. To store and migrate these sparse weights, we usually decompose them into two types of data, *i.e., non-zero weights* and *meta-data*. Particularly, these non-zero weights are the effective path of the original network that have high impact to the final prediction, while the meta-data (expressing index information) is adopted to encode and decode these non-zero weights. It seems that we can achieve a high compression ratio by reducing the number of the non-zero weights as much as possible, when guaranteeing an acceptable final performance. Unfortunately, we will still suffer from a high amount of meta-data, which may be several times larger than the non-zero weights. In fact, the large amount of meta-data is a roadblock for pruned DNNs to compress, store and migrate.

*Corresponding author
Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.



(a) Compression processes (b) Model size

Figure 1: Example of COO algorithm that illustrate the redundancy in the process of sparse weight storage.

An intuitive example is shown in Figure 1. Given a small network, *i.e.,* ADMM-Lenet (Zhang et al. 2018), with a size of 1682KB, which prunes Lenet into a very sparse network with a compression ratio of 98.6%. By using the Coordinate Format (COO) algorithm (Bell and Garland 2009), there are only 23.5KB non-zero weights in this compressed network, but we still need extra 47.4KB meta-data to recover this network. It means that, we need 70.9KB storage space to store these non-zero weights, which is three times larger than what we essentially want to store. Thus, the large size of meta-data will harm the actual compression ratio of the model, which is not deeply studied by traditional sparse coding methods in the literature. In addition, traditional sparse coding methods (Bell and Garland 2009; Zhu and Gupta 2018; Han, Mao, and Dally 2016) also ignore one important requirement of DNNs, *i.e.,* efficient inference. This will be another bottleneck of deploying the compressed deep models.

In this paper, we propose a *layerwise sparse coding (LSC)* method, which tries to extremely maximize the compression ratio of the pruned DNN models by specially design the encoding mechanism of the meta-data. Also, we take the requirement of efficient inference into consideration, making it possible to inference without full decoding. Specifically, LSC is a two-layer method. In the first layer (block layer), for the efficient inference purpose, we divide each sparse weight matrix into multiple small blocks, and then mark and remove zero blocks. Since zero blocks have no effect on the final prediction, we only need to pay attention to the small number of non-zero blocks, which can naturally accelerate the coding and inference processes. In the second layer (coding layer), we propose a novel *signed relative index (SRI)* algorithm to tackle these non-zero blocks with minimal amount of meta-data.

In summary, our main contributions are as follows:

- We find that the true bottleneck of the compression ratio is caused by the meta-data. Furthermore, we propose an LSC method to tackle this problem.
- To accelerate the inference process, we divide these sparse weight matrices into multiple small blocks to make better use of their sparsity. Moreover, the dividing mechanism makes the compressed model be able to infer efficiently without full decoding.
- We propose a novel SRI algorithm, which can encode the non-zero weights with minimal space (*i.e.,* minimal size of meta-data). In addition, we theoretically prove that our SRI is superior to other coding methods.
- Extensive experiments demonstrate that the proposed LSC achives substantial gains in pruned DNN compression (*e.g.* 51.03x compression ratio on ADMM-Lenet) and inference computation (*i.e.* less time and extremely less memory bandwidth), over state-of-the-art baselines.

2 Related Work

The general processes of learning a compressed DNN model are as follows. We first initialize and train an over-parameterized DNN model (Luo et al. 2018; Liu et al. 2019). Next, we eliminate weights that contribute less to the prediction by pruning and retrain this model. And then repeating the processes of pruning and retraining for several times. Finally, we obtain a model which can maintain the similar performance as original but has much less valid weights. Since the weights of the model become sparse after pruning, studies of how to store and migrate these sparse weights are the spotlight in recent years. Typically, these studies can be classified into two categories depending on the goal:

- **Compression ratio.** Most compression methods adopt some classic sparse coding algorithms just based on the programming frameworks they used. For example, MATLAB, TensorFlow and Pytorch integrate the COO algorithm as their default sparse coding method, while Scipy employs Compressed Sparse Row/Column (CSR/CSC) to encode the sparse matrices (Bell and Garland 2008; 2009). Recently, some new algorithms are also proposed, such as Bitmask (Zhu and Gupta 2018) and Relative index (Han, Mao, and Dally 2016). The above algorithms are capable of encoding sparse models, but they are all procedure methods that are difficult to be implemented in parallel.
- **Parallel computing.** To take full advantage of the resources of the deep learning accelerators, a series of novel sparse coding algorithms are presented in recent years, including Block Compression Sparse Column (BCSC) (Xie et al. 2018), Viterbi-Compressible Matrix and Nested Bitmask Format (VCM-Format) (Lee et al. 2018), and Nested Bitmask (NB) (Zhang et al. 2019). These algorithms are suitable for parallelized environments, but the compressed models still consumes considerable storage and memory bandwidth.

A large gap between the above two categories of methods is that, it is difficult to achieve high compression ratio and efficient computing simultaneously. Different from these sparse coding methods, our proposed LSC method can not only make the model inferring in parallel, but also enjoy extremely less meta-data, making a higher compression ratio for pruned deep models.

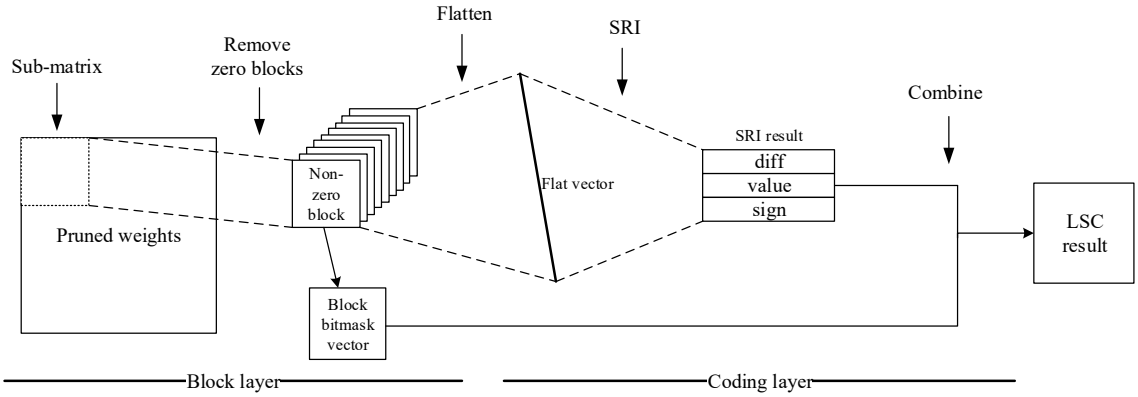


Figure 2: Structure of Layerwise Sparse Coding.

3 Structure of Layerwise Sparse Coding

In this paper, we adopt the following compression procedures to compress a trained DNN. First, given a trained network with a massive number of weights, one pruning technique (Han et al. 2015; Anwar, Hwang, and Sung 2017) is applied to this network, making it as sparse as possible. Second, we employ our proposed *layerwise sparse coding (LSC)* method to further compress and encode these sparse weights to achieve an extreme compression ratio.

As is shown in Figure 2, the proposed LSC mainly consists of two layers, *i.e.,* a *block layer* and a *coding layer*. In the block layer, for each sparse weight matrix, we propose a *block bitmask* mechanism to divide it into multiple small blocks. Next, all the non-zero blocks in this matrix are picked and flattened into one single vector. And then, the flattened vector is fed into the succeeding coding layer. Finally, a novel *Signed Relative Index (SRI)* algorithm is designed to effectively encode this flattened vector with extremely limited meta-data.

Block Layer: Block Bitmask Construction

The main purpose of the presented block bitmask mechanism is to reduce the computing workload at a large granularity (*i.e.,* block) by utilizing the sparsity of matrices. To be specific, we attempt to divide a compression task into several sub-problems by cutting a sparse matrix into multiple blocks. For each block, if there is any non-zero weight in this block, we mark it as *true* using a 1-bit signal, otherwise it is indexed by *false*. After that, we can obtain a mask consisting of many 1-bit signals, which can simply mark all of the blocks. The advantage is that we do not need to consider the zero blocks (marked by *false*) any more, and only need to focus on the non-zero blocks (marked by *true*). Generally, because of the high sparsity of these sparse matrices, there is a large number of zero blocks that do not need to be specially considered. Therefore, this kind of block mechanism can naturally accelerate the subsequent coding process and simultaneously save the storage space.

In particular, we flatten all of these non-zero blocks into one single vector and input the flattened vector into the coding layer. The superiority of the flattening operation is that we do not need separately encode each non-zero block which will introduce additional meta-data.

Coding Layer: Intensity Compression of Signed Relative Index

In the coding layer, we perform an intensity compression on these flattened non-zero blocks (*i.e.,* flattened vectors) to really compress and encode all the non-zero weights. Typically, Relative Index (RI) (Han, Mao, and Dally 2016) algorithm can be adopted to achieve this purpose. RI usually decomposes a sparse matrix into two types of vectors: A *diff* vector and a *value* vector. Specifically, the *diff* vector records the relative position between each non-zero weight and the nearest non-zero weight in front of it, and the *value* vector collects and stores all these non-zero weights accordingly (see Figure 3).

One limitation of the RI algorithm is that, when the pruned DNNs are highly sparse, there may be too many additional zero fillers in the *value* vector. It will lead to a large waste of space. This is because, since units in the *diff* vector take 3 bits each, to avoid overflow, RI adds a zero filler in the *value* vector whenever a *diff* unit reaches its maximum (*i.e.,* 7) that can be expressed. Take a 90% pruned deep network as an example, we find that 44% of the RI results are zero fillers, and each filler takes 35 bits (3 bits *diff* unit and 32 bits *value* unit). In fact, due to the high sparsity of deep models, the *diff* units overflow frequently and need many zero fillers, which extremely increases the size of meta-data.

To tackle the above limitation, we propose SRI, an intensity compression algorithm with high compression ratio, as a core algorithm of the coding layer. SRI decomposes a sparse matrix into three vectors: *value*, *diff* and *sign*. Specifically, the *value* vector records the non-zero values of a sparse matrix, the *diff* vector records the relative position between each non-zero value and the previous one. Like the RI algorithm, when the relative position exceeds the maximum value that *diff* unit can represent, a filler is added in case of overflow happens. However, the difference is that SRI does not need to add a filling zero into the *value* vector. Instead, we add a *sign* unit when the corresponding *diff* unit equals to its maximize value, to determine whether there is a filler.

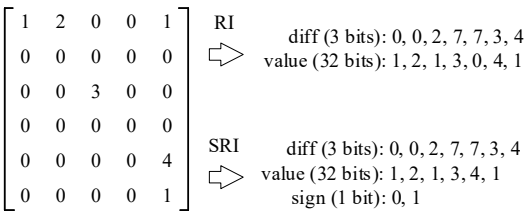


Figure 3: Comparison between RI and SRI

Therefore, SRI does not force a one-to-one correspondence between the *diff* vector and *value* vector. When one *diff* unit reaches the maximum value that can not be expressed, SRI takes only 4 bits (3 bits *diff* unit and 1 bit *sign* unit) to record this filler.

Efficient Inference without Full Decoding

Many previous works usually only focus on the sparse coding procedure, but ignore the decoding procedure in the inferring stage. In this sense, the high compression ratios of these works lack practical significance, because they still need a full decoding during the inferring. On one hand, full decoding still needs a large space which is in conflict with the purpose of model compression. On the other hand, full decoding is not very efficient. Therefore, the sparse coding algorithms should support matrix multiplication in the state of incomplete decoding. To this end, we propose a more efficient procedure for LSC codes for inference. Specifically, we utilize intermediate decoding results to early start-up the matrix multiplication procedure, which takes less memory and also accelerate the calculation.

In general, the inference procedure of the DNN models consists of a set of matrix calculations (Warden 2015). To maintain the computability of the LSC codes, the computational properties of the matrix must be preserved. The proposed efficient inference procedure for LSC divides one matrix into multiple sub-matrices and splits the full matrix multiplication into the calculations of the multiple sub-matrices. Fortunately, because of the dividing mechanism for matrix multiplication, we do not need to fully decode the entire LSC codes before the matrix calculation. It means that the intermediate decoding results can be directly used for the calculation during the inference, which can save considerable running memory. Similarly, because the weight matrices are usually very sparse, we can skip the calculations of zero blocks. The bitmask vector we obtained in the block layer can help us to distinguish zero blocks and eliminate a large number of sub-matrix calculations. We divide the implementation of LSC code matrix multiplication into the following four steps:

- **Computing tree construction.** The computing tree is used to determine the calculation flow of the matrix multiplication when inferring. We first split the multiplication of two matrices (*e.g.,* $W \times X$) into the calculations of multiple sub-matrices. Taking Figure 4 as an example, according to the principle of block matrix multiplication, we split W into multiple sub-matrices, and then convert $W \times X$ into the calculations among several sub-matrices. As seen, the calculation process of each row can be further converted into a computing tree.
- **Prune computing tree.** The highly sparse nature of the pruned DNNs makes a large number of zero blocks, which means that we can skip the multiplications of these zero blocks. As is shown in Figure 5, we can perform pruning on the computing tree based on the signals of these zero blocks according to the block bitmask.
- **SRI decoding & sub-matrices multiplication.** At the inference stage, if we perform the LSC decoding first and then do the sub-matrix multiplication, it will waste a large amount of memory space. In fact, since SRI only encodes the non-zero blocks, we can design a more efficient multi-procedure mechanism to implement the inference. Specifically, there are two procedures, *i.e.,* a *SRI decoding procedure* and a *sub-matrix multiplication procedure*, in this mechanism. The SRI decoding procedure recover the SRI codes into non-zero blocks, and the sub-matrix multiplication procedure adopts these non-zero blocks to perform the sub-matrix multiplication for the inference calculation. Since we only need to calculate non-zero blocks we can perform matrix multiplication efficiency. Further more, these two procedures are relatively independent, the decoded non-zero blocks will be destroyed once the sub-matrix multiplication is finished. In this way, we can significantly save the memory bandwidth compared with the full decoding manner. Notably, if these two procedures run at the same speed, the memory bandwidth can be further saved.
- **Intermediate results integration.** The final step is to integrate all the intermediate calculation results of the sub-matrix multiplications to obtain the final result. Because the results integration is independent, it can be implemented in parallel.

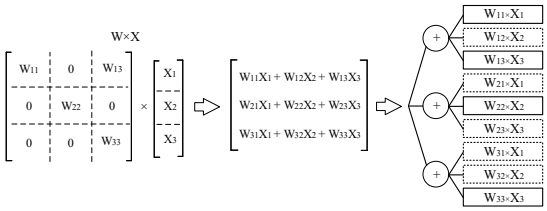


Figure 4: Computing tree construction.

4 Analyses of Layerwise Sparse Coding

In this section, we first give a few mathematical analyses of the proposed LSC, including its time complexity and the impact of different block sizes on the results. Finally, we take the Bitmask (Zhu and Gupta 2018) as an example to claim the advantages of the proposed SRI algorithm over other algorithms. We define some notations first. Suppose the spatial size of a full matrix is $n = n_r \times n_c$, where n_r denotes the width and n_c is the height. Similarly, let $k = k_r \times k_c$ indicates the block size in the block layer.

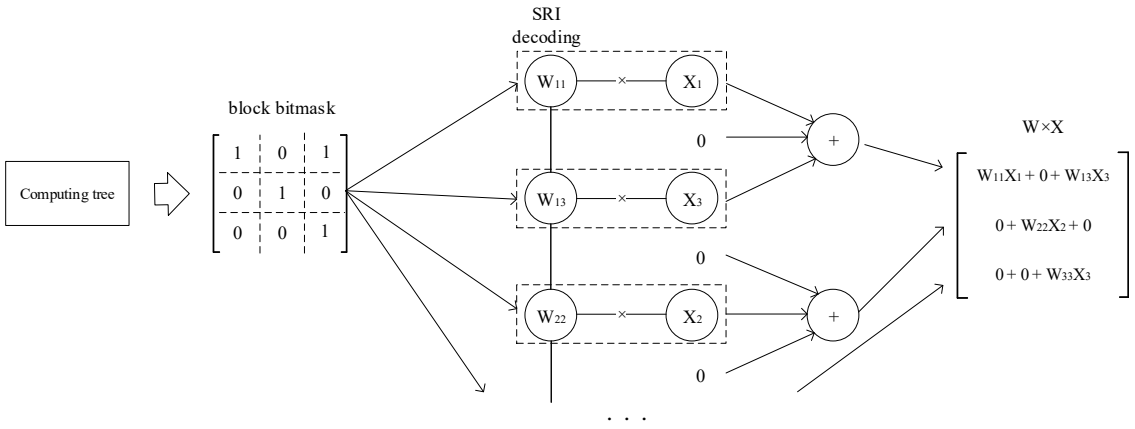


Figure 5: LSC in matrix multiplication.

Time Complexity

Let $T(n)$ be the total running time of LSC on any matrix with a size of n. Considering that the block layer and the coding layer in LSC are relatively independent, we have

$$T(n) = T_{bl} + T_{cl} \tag{1}$$

where T_{bl} and T_{cl} indicate the running time of LSC on the block layer and coding layer, respectively.

In the block layer, the main operation of LSC is to divide an input matrix into $\frac{n}{k}$ blocks, which only takes linear time cn with respect to the size n (*i.e.,* $T_{bl} = cn$, where c is a constant calculation time). In addition, a block bitmask is adopted to mark each block, where the non-zero blocks are flattened into a single vector. In the coding layer, the proposed SRI algorithm is performed to encode this flattened vector (*i.e.,* these non-zero blocks), which will take a time complexity of T_{cl}. Basically, the SRI coding only needs to traverse the matrix once to fill in its three kinds of vectors (*i.e., diff*, *value* and *sign*). Therefore, if the time complexity of the SRI algorithm for each block is $T_{SRI}(k) = ck$, we can obtain the time complexity of the coding layer as

$$T_{cl} \leq \frac{n}{k} T_{SRI}(k) = ck\frac{n}{k} = cn\,. \tag{2}$$

Thus, we can get the total time complexity of our LSC method

$$T(n) \leq 2cn\,. \tag{3}$$

Since c is a constant time, the time complexity of LSC is a linear complexity function of the scale variable n.

Block Related Settings

In the block layer, LSC divides the sparse weight matrix into several sub-matrices (*i.e.,* blocks) with the same size, where the block size k is a hyper-parameter and may be important for the final compression ratio. To make LSC more scalable, it is necessary to analyze the impact of the block size under different sparsity. In particular, let S be the size of a pruned sparse weight matrix

$$S = M_{bl} + M_{cl} + S_{non-0}\,, \tag{4}$$

where M_{bl} and M_{cl} indicate the meta-data size in the block layer and coding layer, respectively, and S_{non-0} denotes the size of non-zero weights.

Given a random matrix with the sparse rate of p, in the block layer, we divide this matrix into $\frac{n}{k}$ blocks. For each block, the probability to be one zero block is p^k. The number of zero blocks can be calculated as $\frac{np^k}{k}$. After removing all zero blocks, the number of non-zero blocks is $\frac{n(1-p^k)}{k}$. Next, these non-zero blocks are flattened into a single vector, which contains $n(1-p^k)$ weights. Specifically, the new sparse rate p_{bl} of this flatten vector becomes to

$$p_{bl} = \frac{p - p^k}{1 - p^k}\,. \tag{5}$$

In the block layer, since each block takes 1 bit signal to record, we will take $M_{bl} = \frac{n}{k}$ bits (*i.e.,* the meta-data size in the block layer) in total to record this matrix. In the coding layer, the proposed SRI algorithm further encodes the flatten vector into three vectors (*i.e.,* SRI codes): A *value* vector (32 bits per unit), a *diff* vector (3 bits per unit), and a *sign* vector (1 bit per unit). Since the *sign* vector takes up very little space, we just ignore it for simplicity. Then, we can calculate the average size of these SRI codes as below

$$\begin{aligned} S_{non-0} = S_{value} &= 32n(1-p) \\ M_{cl} \geq S_{diff} &= 3n(1-p) \end{aligned} \tag{6}$$

According to the above analyses, we can get the relationship between S and the block size k with respect to the sparse rate p as below

$$S \geq \frac{n}{k} + 35n(1-p)\,, \tag{7}$$

which means that we can easily obtain the compression ratio based on this function.

Table 1 shows the experiment results on different block sizes under various sparse rates. Empirically, the sparse rate of a pruned DNN model is generally around 0.95 (95%) (Han et al. 2015; Han, Mao, and Dally 2016; Dai, Yin, and Jha 2019; Molchanov, Ashukha, and Vetrov 2017; Luo et al. 2018; Guo, Yao, and Chen 2016), and thus we recommend setting the default block size to 3×3.

	Rate	2*2	3*3	4*4	5*5	6*6	7*7	8*8	9*9
Block bitmask & SRI (LSC)	99%	53.24x	66.05x	**69.22x**	66.60x	63.24x	58.48x	55.53x	51.45x
	98%	33.65x	37.52x	**37.64x**	35.91x	34.04x	32.20x	31.14x	29.84x
	97%	24.59x	**26.26x**	25.98x	24.94x	23.83x	22.98x	22.48x	22.01x
	96%	19.38x	**20.23x**	19.94x	19.24x	18.55x	18.10x	17.88x	17.68x
	95%	15.99x	**16.47x**	16.20x	15.73x	15.28x	15.05x	14.95x	14.86x
	90%	8.53x	**8.58x**	8.49x	8.39x	8.33x	8.32x	8.33x	8.33x
Block-bitmask & bitmask (NB)	99%	52.49x	**61.83x**	60.36x	55.08x	49.07x	43.46x	39.47x	36.15x
	98%	33.17x	**34.98x**	32.93x	30.06x	27.08x	24.74x	23.42x	22.00x
	97%	24.28x	**24.52x**	22.93x	21.09x	19.36x	18.17x	17.52x	16.95x
	96%	**19.18x**	18.96x	17.77x	16.54x	15.42x	14.76x	14.43x	14.17x
	95%	**15.87x**	15.50x	14.58x	13.74x	12.99x	12.59x	12.44x	12.30x
	90%	**8.58x**	8.25x	7.95x	7.75x	7.61x	7.59x	7.59x	7.59x

Table 1: Compression ratios for LSC and NB on different block sizes and sparse rates.

Core Algorithm for the Coding Layer

In this section, we take Bitmask (Zhu and Gupta 2018) as a comparison algorithm to claim the advantages of our proposed SRI as the core algorithm of the coding layer. According to Eq. (5), we know that the block layer will reduce the sparse rate of the matrix for removing the zero blocks. Also, the number of sub-matrices is reduced to $\frac{n(1-p^k)}{k}$. Therefore, the following analyses and derivations will be under this premise.

If the core algorithm of the coding layer is Bitmask, we can calculate the meta-data size of Bitmask for encoding the sub-matrices as $n(1-p^k)$ bits, which is equal to the number of weights. In contrast, for SRI, the meta-data size consists of M_{bl} and M_{cl}, where $M_{bl} = \frac{n}{k}$ and $M_{cl} \geq 3n(1-p)$. Assuming that Bitmask is better than SRI, the following condition must be satisfied

$$2 - 3p + p^k > 0\,. \tag{8}$$

Since the block size is set to 3×3 as recommended in the previous section, *i.e.,* $k = 9$, the above inequation can only be satisfied when $p < 0.67$ (67%). Note that here we don't take the size of *sign* vector into account. However, in general, the sparse rate of a pruned DNN model is larger than 0.95 (95%), which means that SRI is superior to Bitmask in most cases. In addition, our experiments (see Table 1) demonstrate that SRI is better than Bitmask even if the *sign* vector is taken into concern.

5 Experimental Results

Ablation Study

We perform an ablation study to analyze the influence of different block sizes on the final compression ratios. Specifically, we generate random matrices with different sparse rates and vary the block size from 2×2 to 9×9. Next, we perform the proposed LSC method on these matrices to calculate the final compression ratios. The results are shown in Table 1. As seen, its better to increase the block size as the sparse rate increases, which is consistent with intuition. However, when the block size is too large, we cannot get much better compression ratios. For example, when the sparse rate is larger than 97%, the best block size is 4×4, otherwise the best block size is 3×3. Therefore, in the real applications, we recommend 3×3 to be the default setting of the block size.

In addition, Table 1 also shows the compression ratios of SRI and Bitmask as the core algorithm of the coding layer, respectively. Under the same sparsity, the compression ratios of the Bitmask algorithm are always worse than the proposed SRI regardless of the block size, which well verifies the effectiveness of the proposed SRI algorithm.

Advantage Interval of Related Algorithms

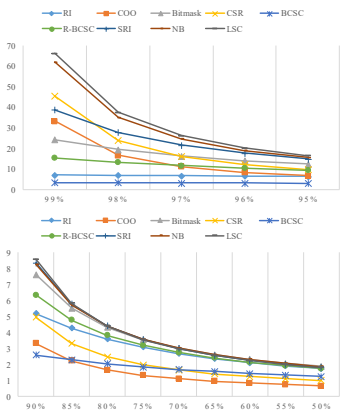
In fact, different sparse coding methods are suitable for different situations. To find the advantage intervals of different algorithms, we generate multiple sparse matrices by varying the sparse rate from 50% to 99% and perform different methods on these matrices. Figure 6 shows the experimental results of different methods. When the sparse rate is higher than 85%, our LSC can obtain the highest compression ratios compared with other sparse coding methods. Moreover, when the sparse rate is lower than 85%, our LSC is still competitive with these methods. Because the sparse rates of the pruned DNN models are generally larger than 95%, in this sense, LSC is consistently superior to other methods.

Compression Results on ADMM-Lenet

In addition, we take a real pruned DNN network, *i.e.,* ADMM-Lenet (Zhang et al. 2018), as an example to demonstrate the effectiveness of LSC. ADMM-Lenet is a 4-layers model, where only few weights are in the convolutional layers, and the most weights are aggregated in the fully connected (fc) layer, especially the *fc1* layer. Although the sparsity of each layer is different, the average sparsity reaches to 98.6%. According to Table 2, LSC performs the best on layers with higher sparse rates, *e.g,* conv2 and fc1. When performing on the entire model, we can see that our LSC obtains the highest compression ratio (*i.e.,* 51.03x) compared with all the other state-of-the-art baselines.

Rate	RI	COO	Bitmask	CSR	BCSC	R-BCSC	SRI	NB	LSC
99%	7.15x	33.32x	24.23x	45.50x	3.43x	15.38x	38.70x	61.87x	**66.10x**
98%	6.96x	16.66x	19.50x	23.82x	3.31x	13.27x	27.72x	35.01x	**37.54x**
97%	6.74x	11.11x	16.32x	16.13x	3.20x	11.67x	21.57x	24.57x	**26.27x**
96%	6.51x	8.33x	14.03x	12.20x	3.10x	10.42x	17.63x	18.98x	**20.23x**
95%	6.28x	6.67x	12.30x	9.81x	3.00x	9.41x	14.90x	15.51x	**16.46x**
90%	5.18x	3.33x	7.62x	4.95x	2.60x	6.34x	8.36x	8.25x	**8.57x**
85%	4.28x	2.22x	5.52x	3.31x	2.29x	4.78x	5.79x	5.70x	**5.82x**
80%	3.60x	1.67x	4.32x	2.49x	2.05x	3.83x	**4.42x**	4.38x	4.41x
75%	3.09x	1.33x	3.56x	1.99x	1.85x	3.20x	**3.57x**	**3.57x**	3.56x
70%	2.69x	1.11x	3.02x	1.66x	1.69x	2.75x	3.00x	**3.02x**	2.98x
65%	2.37x	0.95x	**2.62x**	1.42x	1.56x	2.41x	2.58x	**2.62x**	2.56x
60%	2.12x	0.83x	**2.32x**	1.25x	1.44x	2.14x	2.26x	2.31x	2.25x
55%	1.92x	0.74x	**2.08x**	1.11x	1.34x	1.93x	2.02x	2.07x	2.01x
50%	1.75x	0.67x	**1.88x**	1.00x	1.25x	1.75x	1.82x	1.87x	1.81x

(a)



(b)

Figure 6: Compression ratios of different algorithms with in various sparse rate.

Layers (sparse rate)	RI	COO	Bitmask	CSR	BCSC	R-BCSC	NB	SRI only	LSC
conv1 (0.8)	3.7x	1.66x	4.17x	2.20x	2.00x	3.65x	3.97x	**4.27x**	4.03x
conv2 (0.93)	5.39x	4.17x	8.97x	6.01x	2.75x	7.28x	10.43x	10.10x	**10.73x**
fc1 (0.991)	7.06x	37.03x	24.84x	51.04x	3.44x	15.63x	67.75x	40.08x	**71.85x**
fc2 (0.92)	4.98x	4.75x	9.78x	6.45x	2.82x	7.80x	**12.50**	10.92x	12.05x
Average (0.986)	6.69x	23.71x	22.05x	33.11x	3.38x	14.44x	48.77x	33.04x	**51.03x**

Table 2: Compression ratios of different algorithms on ADMM-Lenet model.

Time and Memory Bandwidth in the Inference

In addition to the compression ratio, for model compression, the inference time and memory bandwidth at the test stage are also very important. In this section, we conduct experiments on a CPU with two threads (Intel Core i7-5500U @2.40GHz) to verify the efficiency of the proposed LSC. Specifically, we calculate the time and memory bandwidth of our LSC used at the inference stage on a sparse matrix with a sparse rate of 98%. Moreover, Bitmask and SRI only are taken as the baselines. As for SRI only, a variation of our LSC, it directly encodes the original input matrix with the proposed SRI without the block layer. The results are shown in Table 3. As seen, our LSC is the most efficient method compared with other two baselines. This is because LSC benefits from both the block layer and the new designed efficient inference mechanism.

Decoding Procedure	Time (ms)	Memory (mb)
Bitmask	64.34	2.40
SRI only	81.65	2.37
LSC (ours)	**19.34**	**0.10**

Table 3: Time and memory usage of different methods at the inference stage.

6 Conclusion

Deep neural networks have been widely and successfully applied to a lot of fields. However, the huge requirements of energy consumption and memory bandwidth limit the development of these deep models especially in the resource-constrained environments. General works are to employ model compression techniques to compress a deep model as compact (sparse) as possible. Nevertheless, for the compressed models, we find that the existing sparse coding methods still consume a much larger amount of meta-data to encode these non-zero wights for these compressed models, resulting in the model compression efficiency not truly realized.

To tackle the above issue, we propose a *layerwise sparse coding (LSC)* method to maximize the compression ratio by extremely reducing the amount of meta-data. Specifically, we separate the compression procedure into two layers, *i.e.*, a *block layer* and a *coding layer*. In the block layer, we divide a sparse matrix into multiple small blocks and remove the zero blocks. Next, in the coding layer, we propose a novel SRI algorithm to further encode these non-zero blocks. Furthermore, we design an efficient decoding mechanism for LSC to accelerate the coded matrix multiplication in inference stage. Extensive experiments demonstrate the effectiveness of the proposed LSC over other state-of-the-art sparse coding methods.

Acknowledgements

This work is supported by Science and Technology Innovation 2030–"New Generation Artificial Intelligence" Major Project No.(2018AAA0100905), the National Natural Science Foundation of China (Nos. 61432008, 61806092), and Jiangsu Natural Science Foundation (No. BK20180326).

References

Abdel-Hamid, O.; Mohamed, A.-r.; Jiang, H.; Deng, L.; Penn, G.; and Yu, D. 2014. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on audio, speech, and language processing* 22(10):1533–1545.

Anwar, S.; Hwang, K.; and Sung, W. 2017. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13(3):32.

Bell, N., and Garland, M. 2008. Efficient sparse matrix-vector multiplication on cuda. Technical report, Nvidia Technical Report NVR-2008-004, Nvidia Corporation.

Bell, N., and Garland, M. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*, 18. ACM.

Chen, T.; Du, Z.; Sun, N.; Wang, J.; Wu, C.; Chen, Y.; and Temam, O. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, volume 49, 269–284. ACM.

Dai, X.; Yin, H.; and Jha, N. 2019. Nest: A neural network synthesis tool based on a grow-and-prune paradigm. *IEEE Transactions on Computers*.

Floreano, D., and Wood, R. J. 2015. Science, technology and the future of small autonomous drones. *Nature* 521(7553):460.

Guo, Y.; Yao, A.; and Chen, Y. 2016. Dynamic network surgery for efficient dnns. In *Advances In Neural Information Processing Systems*, 1379–1387.

Han, S.; Pool, J.; Tran, J.; and Dally, W. 2015. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, 1135–1143.

Han, S.; Liu, X.; Mao, H.; Pu, J.; Pedram, A.; Horowitz, M. A.; and Dally, W. J. 2016. Eie: efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 243–254. IEEE.

Han, S.; Mao, H.; and Dally, W. J. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding.

Hu, X. S. 2018. A cross-layer perspective for energy efficient processing:-from beyond-cmos devices to deep learning. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, 7–7. ACM.

Kamilaris, A., and Prenafeta-Boldú, F. X. 2018. Deep learning in agriculture: A survey. *Computers and electronics in agriculture* 147:70–90.

Krajník, T.; Vonásek, V.; Fišer, D.; and Faigl, J. 2011. Ar-drone as a platform for robotic research and education. In *International conference on research and education in robotics*, 172–186. Springer.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 1097–1105.

Lee, D.; Ahn, D.; Kim, T.; Chuang, P. I.; and Kim, J.-J. 2018. Viterbi-based pruning for sparse matrix with fixed and high index compression ratio.

Liu, Z.; Sun, M.; Zhou, T.; Huang, G.; and Darrell, T. 2019. Rethinking the value of network pruning. *International Conference on Learning Representations (ICLR)*.

Luo, J.-H.; Zhang, H.; Zhou, H.-Y.; Xie, C.-W.; Wu, J.; and Lin, W. 2018. Thinet: pruning cnn filters for a thinner net. *IEEE transactions on pattern analysis and machine intelligence*.

Molchanov, D.; Ashukha, A.; and Vetrov, D. 2017. Variational dropout sparsifies deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 2498–2507. JMLR. org.

Park, E.; Ahn, J.; and Yoo, S. 2017. Weighted-entropy-based quantization for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 5456–5464.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, 3104–3112.

Warden, P. 2015. Why gemm is at the heart of deep learning. *Peter Warden's Blog*.

Xie, X.; Du, D.; Li, Q.; Liang, Y.; Tang, W. T.; Ong, Z. L.; Lu, M.; Huynh, H. P.; and Goh, R. S. M. 2018. Exploiting sparsity to accelerate fully connected layers of cnn-based applications on mobile socs. *ACM Transactions on Embedded Computing Systems (TECS)* 17(2):37.

Zhang, C.; Li, P.; Sun, G.; Guan, Y.; Xiao, B.; and Cong, J. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 161–170. ACM.

Zhang, T.; Ye, S.; Zhang, K.; Tang, J.; Wen, W.; Fardad, M.; and Wang, Y. 2018. A systematic dnn weight pruning framework using alternating direction method of multipliers. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 184–199.

Zhang, Y.; Du, B.; Zhang, L.; Li, R.; and Dou, Y. 2019. Accelerated inference framework of sparse neural network based on nested bitmask structure. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Zhu, M., and Gupta, S. 2018. To prune, or not to prune: exploring the efficacy of pruning for model compression. In *International Conference on Learning Representations (ICLR)*.