Accelerating Tensor Swapping in GPUs With Self-Tuning Compression

Ping Chen, Shuibing He, *Member, IEEE*, Xuechen Zhang, *Member, IEEE*, Shuaiben Chen, Peiyi Hong, Yanlong Yin, and Xian-He Sun, *Fellow, IEEE*

Abstract—Data swapping between CPUs and GPUs is widely used to address the GPU memory shortage issue when training deep neural networks (DNNs) requiring a larger amount of memory than that a GPU may have. Data swapping may become a bottleneck when its latency is longer than the latency of DNN computations. Tensor compression in GPUs can reduce the data swapping time. However, existing works on compressing tensors in the virtual memory of GPUs have three major issues: lack of portability because its implementation requires additional (de)compression units in memory controllers, sub-optimal compression performance for varying tensor compression ratios and sizes, and poor adaptation to dense tensors because they only focus on sparse tensors. We propose a self-tuning tensor compression framework, named CSwap+, for improving the virtual memory management of GPUs. It uses GPUs for (de)compression directly and thus has high portability and is minimally dependent on GPU architecture features. Furthermore, it only applies compression on tensors that are deemed to be cost-effective considering their compression ratio, size, and the characteristics of compression algorithms at runtime. Finally, to adapt to DNN models with dense tensors, it also supports cost-effective lossy compression for dense tensors with nearly no model training accuracy degradation. We conduct the experiments through six representative memory-intensive DNN models. Compared to vDNN, CSwap+ reduces tensor swapping latency by up to 50.9% and 46.1% with NVIDIA V100 GPU, for DNN models with sparse and dense tensors, respectively.

Index Terms—DNN, GPU, tensor, swapping, compression

1 Introduction

Deep Neural Networks (DNNs) have been successfully used in various domains, such as computer vision [1], recommendation systems [2], speech recognition [3], etc. DNN models become larger and deeper to achieve higher prediction accuracy [4], [5]. Training such DNN models often requires a larger amount of memory. For example, the latest BERT model training needs more than 70 GB of memory with batch size 64 [4]. The newest language model presented by Google has 137 billion parameters and requires more than 100 GB of memory for training [6]. Additionally, a prior study [7] shows that the number of neural network parameters has nearly doubled every 2.4 years since the 80s. These trends lead to a higher memory demand for training future DNN models.

To accelerate the training of DNN models, hardware accelerators such as Graphics Processing Units (GPUs) are widely used to compute tensors [8]. However, GPUs have limited memory capacity compared to what is demanded in the training of many popular DNNs. For instance, the powerful NVIDIA V100 GPU is configured with up to 32 GB onboard memory, which is inadequate for training the BERT model which consumes up to 73 GB of memory [9]. The lack of global GPU memory greatly constrains the development of more advanced DNN architectures.

Because GPU memory could be under-provisioned for training large models, both scale-out and scale-up approaches may be used to overcome this limitation. The scale-out approaches exploit distributed memory of multiple GPUs in a cluster. Its downside is that their performance may be constrained by networking latency [10]. The scaling-up approaches swap intermediate tensors between GPUs and CPUs in training [11], [12], [13], [14], [15], [16]. They can be further improved by overlapping tensor swapping with computations of the next layer to hide application-perceived swapping latency. Nevertheless, Rhu *et al.* observed that the swapping latency of large tensors cannot be effectively hidden for the increasingly larger gap between drastically improved TFLOPS performance of GPUs and limited data transfer bandwidth of PCIe links for tensor swapping between GPUs and CPUs [17]. They implement a sparse tensor compression engine located in memory controllers of GPUs and reduce the DNN training time through swapping smaller tensors.

- *Ping Chen, Shuibing He, Shuaiben Chen, and Peiyi Hong are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China, and also with the Zhejiang Laboratory, Hangzhou 311100, China. E-mail: {zjuchenping, heshuibing, jsxnh, hongpeiyi}@zju.edu.cn.*
- *Xuechen Zhang is with the School of Engineering and Computer Science, Washington State University Vancouver, Vancouver, WA 98686 USA. E-mail: xuechen.zhang@wsu.edu.*
- *Yanlong Yin is with the Institute of Open Source Chip, Beijing 100000, China. E-mail: yyin@bosc.ac.cn.*
- *Xian-He Sun is with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616 USA. E-mail: sun@iit.edu.*

Manuscript received 13 February 2022; revised 15 July 2022; accepted 21 July 2022. Date of publication 26 July 2022; date of current version 23 August 2022. This work was supported in part by the National Key Research and Development Program of China under Grant 2021ZD0110700, in part by the National Science Foundation of China under Grant 62172361, in part by Zhejiang Lab Research Project under Grant 2020KC0AC01, and in part by US National Science Foundation under Grant CNS 1906541.
(Corresponding author: Shuibing He.)
Recommended for acceptance by V. Cardellini.
Digital Object Identifier no. 10.1109/TPDS.2022.3193867

Compressing tensors using additional (de)compression units seems a straightforward approach because no changes are required for DNN applications. However, current compression schemes have three major issues.

First, *they require hardware changes [17], thus having no portability to mainstream GPUs*. Existing GPUs cannot benefit from these tensor compression schemes because they do not have dedicated compression units in their memory controllers. A practical solution should be independent of additional hardware features. Second, *they always use a static compression method to compress all tensors [17], [18]*, and are not optimal because they ignore the compression and decompression time, the changing of tensor sparsity, and the potential of hiding swapping overhead. For example, our study shows that tensors' compression ratio changes constantly during training. Tensors with a low compression ratio and high additional overhead, are not worth being compressed (Section 3.2). Third, *they do not adapt to dense tensors*. The tensors in DNN models can be classified into sparse tensors and dense tensors depending on the types of activation functions used in the models. For example, the ReLU [19] activation will generate sparse tensors and the Leaky ReLU [20] activation will produce dense tensors. However, current schemes only focus on sparse tensors [17], [18], [21]. This may lose the opportunity of compressing dense tensors for further performance optimization.

In this paper, we propose a high-performance, self-tuning, and fully automated GPU memory compression framework, named CSWAP+, for software-level tensor compression management. It has three novel features. First, CSWAP+ uses GPUs for (de)compression directly without relying on fixed compression units in the memory controllers of GPUs. Currently, it supports four GPU-optimized lossless compression algorithms (i.e., zero-value compression (ZVC) [17], run-length encoding (RLE) [22], compressed sparse row (CSR) [23], and LZ4 [24]). CSWAP+ caters to tensor characteristics of a DNN workload and selects one of these four algorithms to achieve the best trade-off between compression ratio and compression time.

Second, CSWAP+ dynamically decides whether to compress sparse output tensors of DNN layers in forward propagation based on the cost-effectiveness of (de)compression. Specifically, it compares the swapping cost with (de)compression to that without (de)compression at runtime. It only executes (de)compression when it is deemed to reduce tensor swapping cost.

Third, to adapt to DNN models with dense tensors, CSWAP+ also supports dynamic cost-effective lossy compression for dense tensors. One challenge is that the lossy compression may significantly degrade the training accuracy. To tackle this issue, CSWAP+ proposes a sliding-down scheme to carefully set the compression parameter of the lossy algorithm, so that the compression yields nearly no training accuracy loss. Compared to CSWAP [18] (the conference version), CSWAP+ can further optimize the swapping performance for DNNs with dense tensors.

In summary, we make the following contributions in this paper:

- We propose CSWAP+, a self-tuning compression framework to reduce tensor swapping cost in DNNs without relying on compression units in the memory controllers of GPUs. It uses GPUs for (de)compression directly.
- We propose a selective cost-effective compression scheme, which adaptively executes lossless compression for sparse tensors and lossy compression for dense tensors, according to the cost-effectiveness of tensor compression at runtime.
- We design a sliding-down scheme to carefully set the compression parameter for the lossy compression algorithm during the entire training process, so that the cost-effective lossy compression yields negligible model training accuracy degradation.
- Our study shows the performance of tensor compression is sensitive to the tensor size, compression ratio, and the characteristics of compression algorithms. Therefore, we design the machine-learning algorithms to predict the tensor (de)compression time for both lossless and lossy compression algorithms.
- We implement a software prototype of CSWAP+ using Torch [25] and apply it to six popular DNN models (e.g., AlexNet [1], VGG16 [26], ResNet [27], etc.). Our experimental results show that CSWAP+ reduces tensor swapping latency by up to 50.9% and 46.1% on sparse and dense DNN models, respectively. Furthermore, CSWAP+ reduces the DNN training time by 18.4% and 16.7% on average for sparse and dense DNNs with NVIDIA V100 GPU, compared to vDNN [14].

2 Background

2.1 DNN Training Architecture

The main goal of DNN training is to find the correct mathematical manipulation to provide high classification accuracy. DNNs consist of multiple layers between input and output. In the training of a DNN, we first perform forward propagation from the first to the last layer in a sequential manner, then we perform backward propagation from the last layer to the first layer to update the parameters of DNNs.

In DNN models, there are many activation-convolution layers which make DNNs non-linear for better accuracy. Users can choose one of the activation functions in their models for different scenarios, such as ELU [28], ReLU [19], Tanh [29], PReLU, and Leaky ReLU [20]. It is very common to use ReLU to build the DNN models because of its simplicity and improved performance. Specifically, ReLU allows positive input values to pass through but resets all negative input values to zeros for fast model convergence. As many values are reset to zeros, ReLU generates sparse tensors. Besides, in some training scenarios with a larger learning rate to speed up convergence, users may choose activation functions, such as Tanh, PReLU, or Leaky ReLU to avoid the dead neurons problems [30] in DNN training. For example, these activation functions are frequently used in popular DNN models, such as the improved ResNet with ELU [31] and the famous YOLO [32]. As their outputs are non-zeros, these functions produce dense tensors.

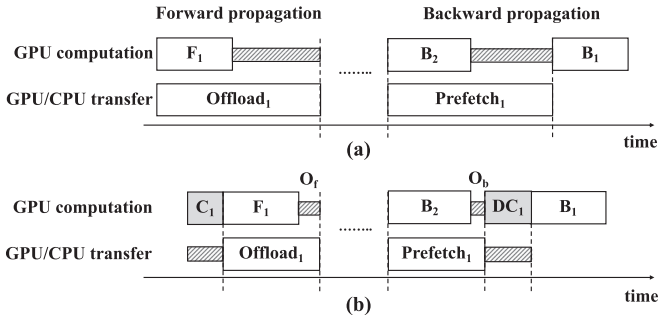


Fig. 1. (a) DNN execution flow with tensor swapping but without tensor compression; (b) The execution flow with both tensor compression and swapping in cDMA [17].

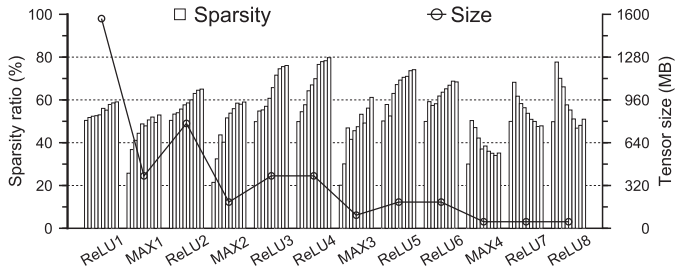


Fig. 2. Changing sparsity of tensors during the training of VGG16 in the first 50 epochs (left axis), while the solid line (right axis) denotes the changing size of tensors of VGG16.

2.2 DNN Memory Management Frameworks

Virtualized Deep Neural Networks. vDNN uses virtual memory to support training a DNN whose memory demand might be larger than the size of GPU memory [21]. It swaps out tensors that are not in use in the forward propagation from GPUs to CPUs and then swaps them back in when they are referenced in the backward propagation of DNN training, as shown in Fig. 1a. In the figure, F_n/B_n denotes the time of forward/backward computation at layer n. $Offload_n$ denotes the time of swapping a tensor from GPUs to CPUs and $Prefetch_n$ denotes the time of swapping a tensor from CPUs to GPUs. If $Offload_n \leq F_n$, $Offload_n$ can be overlapped with F_n, thus resulting in no additional swapping overhead. Similarly, if $Prefetch_n \leq B_{n+1}$, there will be no swapping latency because $Prefetch_n$ can be overlapped with B_{n+1}.

Compressing DMA Engine. Recently, researchers show that tensor swapping latency can no longer be overlapped with DNN forward/backward computation [17]. This is because data transfer bandwidth offered by the powerful PCIe and NVLINK has remained unchanged while the performance of datacenter GPUs is almost tripled since 2014 [17]. To reduce swapping overhead, cDMA compresses all tensors before offloading and decompresses them after prefetching through exploiting the tensor sparsity in GPUs [17].

Fig. 1b illustrates the execution flow of memory swapping with tensor compression in cDMA. O_f and O_b denote the portion of the data transfer time that cannot be effectively hidden from the DNN propagation time, respectively. Only one tensor is swapped per layer in the training process. For cDMA, the compression operations are executed by dedicated (de)compression units in memory controllers of GPUs. It introduces compression latency C_n and decompression latency DC_n. To make cDMA truly effective, (1) C_n and DC_n should be insignificant compared to F_n and B_{n+1} and, (2) $Offload_n$ and $Prefetch_n$ after compression needs to be smaller than their corresponding computation time.

3 Motivation

3.1 Changing Sparsity and Size of Sparse Tensors

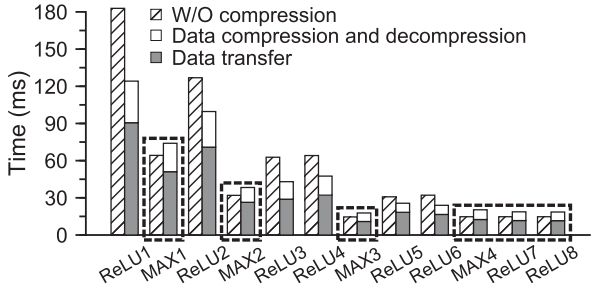
Tensor sparsity is observed in many popular sparse DNN models (using the ReLU activation), e.g., VGG16 and AlexNet. One major cause of tensor sparsity is the nature of ReLU operations, which makes the output tensors of ReLU and POOL tend to contain zeros mostly. We use VGG16 training as an example. We studied its tensor sparsity as the percentage of zeros among all the elements in the output tensors in the first 50 epochs. In the experiments, we use the ImageNet dataset [33], NVIDIA Tesla V100, and the Torch framework [34]. The model is trained with the batch size of 128 until converged with 78.6% top-5 accuracy. (More details of the experimental platforms are described in Section 6).

Fig. 2 shows the tensor sparsity (left y-axis) and the sizes (right y-axis) of each ReLU and MAX layer during the training of VGG16. We can observe that *the sparsity of tensors (bars) varies between 20% and 80% across layers*. To show the trend of changing sparsity of tensors, for a particular layer, we also show the average sparsity of every five epochs as indicated by a bar in each group in the figure. We observe that *for the same layer the sparsity is also dynamically changed*. For example, for *ReLU4*, its sparsity is increased from 50% to 80% over the time of training. In contrast, the tensor sparsity of *ReLU7* is increased in the first 10 epochs and then decreased by 20% afterward.

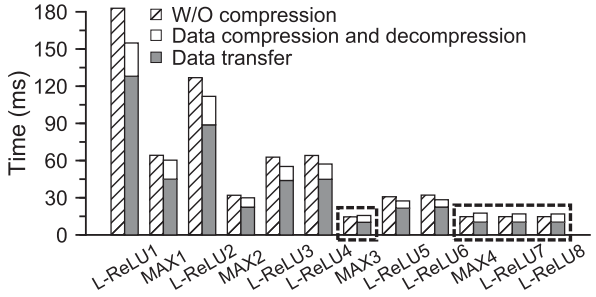
We also measure the tensor sizes during the training of VGG16 on the ImageNet dataset. We find that the tensor size changes across layers (the solid line in Fig. 2). For example, the tensor size is reduced from 1568 MB to 49 MB from the first to the last layer during the training of the model. Furthermore, we find that the tensor size does not change across epochs for the same tensor. We also evaluate the tensor sparsity and tensor size with other models and datasets (Section 6). The results show similar observations. CSwap+ opportunistically applies tensor compression considering the changing tensor size and sparsity.

3.2 Ineffectiveness of Static Lossless Compression for Sparse Tensors

We then study the effectiveness of sparse tensor compression in GPU virtual memory in existing works. Instead of relying on the (de)compression units which are not available in markets, we implement a new *static compression* (*SC*) lossless scheme which replicates the zero-value compression algorithm in cDMA by using GPUs to emulate the (de) compression units in memory controllers. Because GPUs have more cores and higher capacity than those of the (de) compression units in memory controllers, we expect that the (de)compression performance using GPUs directly will be superior to or comparable to that of cDMA. For cDMA, tensor (de)compression is applied to all the layers consisting of ReLU and MAX operations with the *SC* scheme.



(a) Static compression for sparse tensors (ReLU activation).



(b) Static compression for dense tensors (L-ReLU denotes Leaky ReLU activation).

Fig. 3. Swapping time of VGG16 with static compression compared to that without compression. The swapping time using the static compression consists of data transfer time (the lower part of the right bar) and data compression and decompression time (the upper part of the right bar).

Fig. 3a shows the execution time per layer during the training of VGG16 without the lossless compression compared to the time with *SC* using NVIDIA Tesla V100 GPUs and the same experimental setup as described in the previous sections. It also shows the execution time breakdown when *SC* is used. We can observe that the swapping latency with static compression is longer than that without compression for MAX[1-4] and ReLU[7-8]. As the compression ratio and size of tensors are varied, blindly applying lossless compression to all the sparse tensors does not reduce the overall swapping latency when the tensor size is small and its compression ratio is low.

3.3 Motivation and Challenges for Dense Tensor Compression

As mentioned in Section 2.1, tensors in DNN models include sparse tensors and dense tensors, depending on the activation functions used in DNN models. Although existing compression approaches are promising to accelerate tensor swapping performance, they all use *lossless compression* and can only work well for sparse tensors. For dense tensors, such schemes are inefficient because lossless compression algorithms keep the size of the compressed tensors unchanged, which can not reduce the data swapping time.

Because dense tensors are widely generated in DNN models, such as ResNet with ELU [31] and YOLO [32], we need a new approach to reduce the data amount in the tensor swapping process. Considering the fact that approximate computing may require less computing while having a negligible impact on training accuracy [35], we propose to use *lossy compression* to accelerate the dense tensor swapping performance. Lossy compression can be used in many applications, such as graphics applications. In this study, we only focus on DNN models.

There are two unique challenges when applying lossy compression to dense tensor compression. First, the compression algorithm may cause a sub-optimal trade-off between swapping performance and DNN training accuracy. A lossy compression algorithm usually uses a parameter to control the compression ratio. If the parameter is not carefully selected, the lossy compression may bring decent swapping performance improvement but with unacceptable accuracy degradation. For example, with the lossy compression algorithm (i.e., ZFP [36]) the VGG16 model training may lose more than 2% accuracy when the compression ratio is set casually (See Section 6.4). Meanwhile, DNN training often has multiple epochs and in each epoch a given compression ratio has various impacts on model accuracy, making the trade-off hard to achieve. Hence, CSWAP+ leverages a well-designed scheme to carefully choose the compression ratio to achieve a better trade-off between compression performance and training accuracy.

Second, static compression may still be inefficient for dense tensor compression. To illustrate this, we use Leaky ReLU (L-ReLU) instead of ReLU in VGG16 to generate dense tensors and evaluate the execution time per layer using lossy compression during the training. Because a higher compression ratio will cause a higher accuracy loss, we set the ZFP compression ratio as 30% to avoid explicit DNN accuracy loss [37]. In Fig. 3b, we observe that the swapping latency with dense tensor compression is longer than that without compression for MAX[3-4] and L-ReLU [7-8]. The reason is that there isn't enough data to compress for these tensors and therefore the compression does not lead to large enough data and time savings to be cost-effective. Compressing such tensors will slow down the DNN training.

In summary, while tensor compression has been widely adopted to reduce tensor swapping latency, it may not achieve optimal performance and model accuracy without meticulous designs. A novel compression framework is required to dynamically determine when and how to compress tensors at runtime considering the characteristics of DNN networks and GPU architectures.

4 Related Work

Model Compression. DNN training streams need to manage feature maps and model weights. There are two kinds of compression approaches to reduce the size of feature maps: *lossless compression* and *lossy compression*.

Lossless compression algorithms (e.g., RLE [22], CSR [23], LZ4 [40], ZVC [17]) usually work effectively for sparse tensors containing a large number of zero floats. However, they cannot reduce the size of dense float numbers because of the randomness of the ending mantissa bits. Other popular lossless algorithms, such as GZIP [41], FPZIP [42], and BlosC [43], can achieve a good compression ratio for sparse floats on CPUs. But they only achieve suboptimal performance on GPUs because it is hard to accelerate these algorithms using many cores of GPUs.

In contrast, lossy compression has a higher compression ratio on dense floats than does lossless compression.

TABLE 1
Comparison of CSwap+ With Existing Tensor Swapping Frameworks for GPU-Based Deep-Learning Systems

Technique	*Compression unit/location*	*Targeted Tensor*	*Tensor selection*	*Portability*
vDNN [14]	N/A	Sparse & Dense	N/A	Yes
Other swapping [11], [13], [15], [38], [39].	N/A	Sparse & Dense	N/A	Yes
cDMA [17]	Memory Controller	Sparse	No	No
vDNN++ [21]	CPU	Sparse	No	Yes
CSWAP [18]	GPU	Sparse	Yes	Yes
CSwap+	GPU	Sparse & Dense	Yes	Yes

Recently, some GPU-based lossy compression techniques have been developed, such as ZFP [36] and SZ [44]. ZFP splits the whole dataset into many small blocks and compresses the data in each block separately. SZ predicts each data's value with its neighboring points and utilizes the customized Huffman coding to shrink the data size.

Because DNN model weights are over-parameterized [45], many approaches of weight quantization and pruning have been proposed [46]. Besides, DeepSZ [47] designs an error-bounded lossy compression for better DNN inference accuracy for edge devices. However, these approaches are generally used in model inference and are not effective for DNN training tasks because the memory footprint of feature maps is significantly larger than that of weight matrices. For example, the size of feature maps used in training VGG16 is 50× larger than the size of its weight matrices when batch size is 256. Therefore, we focus on feature map compression in the process of DNN training in this paper.

Tensor Swapping Frameworks. We compare CSwap+ to the existing tensor swapping frameworks of GPU virtual memory in Table 1. vDNN studies the characteristics of different DNN layers and chooses to swap convolution input tensors to reduce memory footprint in GPUs [14]. moDNN [38], SuperNeurons [13], SwapAdvisor [11], and HOME [48] introduce different heuristics and profiling technology to swap data between heterogeneous memories. Besides, Capuchin [39] uses the greedy policy and AutoTM [15] chooses Integer Linear Programming to make tensor swapping decisions. However, none of them uses tensor compression in swapping which loses the opportunity for further performance optimization. cDMA [17] was the first swapping framework that compresses sparse tensors using compression hardware in memory controllers of GPUs. vDNN++ [21] supports sparse tensor compression using host CPUs to reduce the pinned memory requirement in the host. Nevertheless, it does not address the tensor transfer bottleneck caused by the limited data transfer bandwidth of PCIe links. CSwap [18] is the first tensor swapping framework using GPUs for tensor (de)compression in the swapping of GPU memory. It is adaptable to all GPUs and automates tensor compression management using machine learning algorithms. However, CSwap can only work well for sparse tensors. In contrast, CSwap+ is efficient for DNN models with both sparse and dense tensors.

5 Design of CSwap+

The design objective of CSwap+ is to opportunistically apply tensor compression for swapping in the training of DNNs when their memory demand is larger than GPU memory capacity. In this section, we describe the architecture of CSwap+ and explain how it improves the DNN training throughput with comparable model accuracy as the default DNNs. Then, we explain how it determines the cost-effectiveness of tensor compression. To make our framework portable and compatible with different GPU architectures, we implement all components of CSwap+ in an existing machine learning software framework.

5.1 Overview of Software Architecture

CSwap+ consists of three components including *tensor profiler*, *execution advisor*, and *swapping executor* as shown in Fig. 4. The *tensor profiler* is executed when a new DNN training task is submitted for the first time. DNN training process usually consists of multiple iterations. During the first iteration, it scans the DNN architecture to judge whether the tensors are sparse or not (i.e., using ReLU activation or other activations) and then collects their profile information for different kinds of tensors.

For sparse tensors, the *tensor profiler* collects the profile including tensor size, tensor sparsity, execution time of each DNN layer without compression, and effective data transfer bandwidth of PCIe links. For dense tensors, the *tensor profiler* collects the same information except tensor sparsity. Additionally, to determine the proper parameter in the final lossy compression, it records the compression ratios with different compression parameters. The detailed profile information is listed in Table 2.

It is notable that we profile the system real-time PCIe bandwidth instead of the manufacturer-claimed bandwidth because the effective bandwidth is usually affected by other factors (e.g., the available PCIe links on the motherboard and the number of GPUs in the system). Most values in a

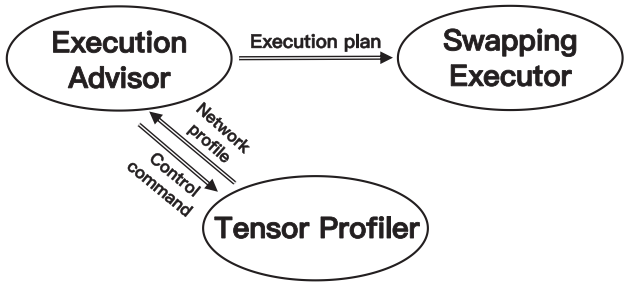


Fig. 4. Architecture overview of CSwap+. The *execution plan* includes compression decision and GPU settings for (de)compression operations. The *network profile* consists of DNN type, tensor sparsity (P for dense DNNs), size, and execution time of layers. The *control command* manages tensor profiles.

TABLE 2
Parameters in the Swapping Cost Model

Notation	Parameter	Profiling frequence
$Size^t$	Size of tensor t	Once
BW_{h2d}	Effective PCIe bandwidth from CPU to GPU	Once
BW_{d2h}	Effective PCIe bandwidth from GPU to CPU	Once
$Hidden_f^t$	Overlapped swapping latency in forward propagation of tensor t	Once
$Hidden_b^t$	Overlapped swapping latency in backward propagation of tensor t	Once
P	Lossy compressor parameter for dense DNNs	Once per epoch
$Sparsity^t$	Sparsity of tensor t for sparse DNNs	Once per epoch
$Time_c^t$	Compression time of tensor t	Offline
$Time_{dc}^t$	Decompression time of tensor t	Offline

DNN profile are unique and fixed under the same GPU and system configurations. To minimize the profiling overhead, we execute the *tensor profiler* to collect the sparsity only once in each epoch. Then the profiling data are stored in an in-memory database for retrieval with low latency.

The *execution advisor* is executed to fetch DNN profiles to decide which compression algorithms to use and whether to compress a tensor for swapping. Finally, the *swapping executor* selects proper tensors, exploits multiple GPU threads to execute compression in parallel before swapping from GPUs to CPUs, and executes decompression after swapping back from CPUs to GPUs.

5.2 Efficient Tensor Compression Algorithms

To compress tensors, CSWAP+ needs efficient compression algorithms that are easy to be implemented on GPUs. CSWAP+ executes tensor (de)compression using multiprocessing units in the GPU in parallel to further reduce the (de)compression time. CSWAP+ integrates both lossless and lossy compression into the framework. In this section, we will discuss the corresponding algorithms for lossless and lossy compression, respectively.

5.2.1 Lossless Compression for Sparse Tensors

CSWAP+ applies a lossless compression algorithm to sparse tensors that are not in use and to be swapped out from GPUs. This compression is relatively straightforward and the DNN training accuracy keeps unchanged as the default training without compression because no float accuracy loss occurs in the compression.

CSWAP+ currently supports four lossless algorithms: run-length encoding (RLE) [22], compressed sparse row (CSR) [23], LZ4 [24], and zero-value compression (ZVC) [17]. RLE stores data sequences where the same value occurs in many consecutive positions as a single value and counts to reduce data size. For example, it can compress an original sequence *(A0000000)* to *(A70)*, decreasing the sequence length from 8 to 3. However, it will increase the original sequence size when the length of consecutive zeros cannot be efficiently reduced. CSR compresses an original sequence as a non-zero value sequence and an additional index representing the locations of non-zero values. For example, a sequence *(A00B0C000)* can be compressed as *(ABC)* and *(035)*. LZ4 uses a dictionary-matching stage to reduce data size. For example, a string *(abcde_bcde)* can be compressed as *(abcde_(5,4))*, where 5 denotes the position how far back the redundant string *(bcde)* can be found and 4 denotes the length of the matched string. Similar to CSR, ZVC stores a float sequence as non-zero values and additional indexes. Instead of using a float as an index for each non-zero value, it utilizes a 32-bit bitmap as the index for 32 consecutive floats. It improves computation speed because of fewer index operations and higher compressibility for less index space overhead.

There is a tradeoff between computation and compressibility for these compression algorithms. The efficiency of an algorithm also depends on the sequence patterns. These algorithms are widely used since they have a relatively high compression ratio. Because PCIe bandwidth is limited, we observe that CSWAP+ favors the most efficient algorithm (i.e., ZVC). This is because ZVC uses a compact bitmap data structure to index compressed data. For example, its memory overhead is only 3% compared to 50% for CSR (i.e., compressing data of 50% sparsity). As a result, swapping using ZVC for sparse tensors achieves lower latency. Currently we implemented these four compressors for GPUs and study their performance. We wish to support more compression algorithms in future work.

5.2.2 Lossy Compression for Dense Tensors

Different from the lossless compression, the lossy compression may significantly degrade the model training accuracy due to the nature of the compression algorithm, therefore the lossy compression scheme needs to be designed carefully.

In CSWAP+, we choose ZFP [36] as the lossy compressor because of its higher throughput than SZ compressor [44], [47] and its configurable interface. ZFP provides a parameter, i.e., P in Table 2, for users to control the desired compression ratio for the tensor, making our design flexible to achieve a trade-off between swapping performance and model accuracy. A large P means a small compression ratio, and vice versa. ZFP provides a maximum value of P, i.e., MAX, as the upper bound of P. For a given P, the tensor size after compression is $\frac{P}{MAX}$ of its original size.

However, there are two challenges to set the proper P for the tensor compression during the entire training process. First, a small P may bring a large compression ratio, i.e., high swapping performance owing to the small compressed tensor size, but it may be at the cost of low DNN training accuracy, and vice versa. Second, a fixed P for all epochs may lead to sub-optimal performance or accuracy. The DNN training consists of multiple epochs. We experimentally observe that the model training accuracy is more

sensitive to the float accuracy at the beginning epochs but less sensitive at the latter epochs (Section 6.4). Therefore, a fixed P cannot achieve the best trade-off between training performance and accuracy.

Algorithm 1. Determine the Parameter Range Through Bisection Search

Require: $Time\_model(t, P)$: the time cost model to predict compression and decompression time; $Benefit(t, P)$: get the time benefit when swapping tensor t compressed under the compressor parameter P; $Hidden(t, P)$: judge whether tensor swapping is fully overlapped with DNN computing;

1: $Top = 0$, $Bottom = MAX$
2: **for** $t = 0, 1, \ldots, N-1$ **do** ▷N tensors
3: $left = 1, right = MAX$ ▷initialization.
4: **while** $left < right$ **do** ▷search for the upper bound.
5: $Mid = (left + right + 1)/2$
6: $Time_c^t, Time_{dc}^t \leftarrow Time\_model(t, Mid)$
7: $cost = Time_c^t + Time_{dc}^t$
8: **if** $Benefit(t, Mid) < cost$ **then**
9: $right = Mid - 1$
10: **else**
11: $left = Mid$
12: **end if**
13: **end while**
14: $top = MAX(top, left)$ ▷get the Top
15: $left = 1, right = MAX$ ▷initialization.
16: **while** $left < right$ **do** ▷search for the lower bound.
17: $Mid = (left + right + 1)/2$
18: $Time_c^t, Time_{dc}^t \leftarrow Time\_model(t, Mid)$
19: $cost = Time_c^t + Time_{dc}^t$
20: **if** $Hidden(t, Mid) == False$ **then**
21: $right = Mid - 1$
22: **else**
23: $left = Mid$
24: **end if**
25: **end while**
26: $bottom = MIN(bottom, left)$ ▷get the $Bottom$
27: **end for**
28: **if** $Top > Bottom$ **then**
29: **return** $Top, Bottom, True$
30: **end if**
31: **return** $Top, Bottom, False$

To overcome these issues, we first propose a scheme to shrink the wide parameter range for improving swapping performance. The main idea is to drop parameters that cannot yield any performance benefits. It divides the parameter range into three parts: 0–*Bottom*, *Bottom*–*Top*, and *Top*–MAX. We assume that a parameter larger than *Top* will not bring performance benefits since the compression ratio is too low to reduce swapping latency. We also assume a parameter smaller than *Bottom* should be excluded since the swapping time has been fully hidden by the computing time and thus further reducing the tensor size is useless. The practical parameter should fall into *Bottom*–*Top*.

Algorithm 1 shows the detailed process to calculate the parameter range. After the initialization (Line #1), CSWAP+ finds *Bottom* and *Top* for each tensor using a bisection method. Specifically, to get the *Top*, CSWAP+ updates the *Mid* constantly until the time benefit ($Benefit(t, Mid)$) is just larger than the compression and decompression overhead (i.e., *cost*) (Line #3-14). The time benefit is calculated as the swapping overhead reduction when the tensor is compressed. To get *Bottom*, CSWAP+ updates the *Mid* until this tensor swapping is just fully hidden in the DNN computing through $Hidden(t, Mid)$ function (Line #15-26). Finally, if *Top* is larger than *Bottom*, CSWAP+ uses the lossy compressor for DNN tensors and returns *True* (Line #28-30). Otherwise, CSWAP+ will not execute the lossy compression because compression does not yield any benefits for the DNN training (Line #31).

Once the parameter range is determined, we then design a sliding down scheme to choose the parameter within the range for optimized model training accuracy. The idea is to gradually decrease the parameter as the number of epoch increases. Specifically, we set P to *Top* in the first epoch, and then decrease it by ΔP each interval until it reaches *Bottom*. The P of each epoch can be calculated by Equation (1), where $Epoch_{cur}$ denotes the current epoch's number (i.e., 0, 1, ..., n-1). ΔP is calculated by Equation (2), where *Epoch* denotes the number of user-customized epochs (i.e., n). For example, supposing *Top* is 100, *Bottom* is 20, and the model needs to train 90 epochs, then ΔP is set to 1. Therefore, we will initialize P as 100 at the beginning, and decrease it to 99 in the next epoch, until it reaches 20 in the 81st epoch. Then, the parameter stays at 20 until the end of the training. Compared to the schemes with a fixed parameter, such a dynamic method can improve tensor swapping performance with nearly no accuracy loss, as shown in Section 6.4

$$P = \max(Top - Epoch_{cur} \times \Delta P, Bottom) \tag{1}$$

$$\Delta P = \left[\frac{Top - Bottom}{Epoch} + 0.5\right], \Delta P \in integer. \tag{2}$$

5.3 Determining Cost-Effectiveness of Tensor Compression

For both sparse and dense tensors, with the changing tensor compression ratio and size, the cost-effectiveness of tensor compression for swapping should be dynamically determined. To achieve this goal, we build a model of swapping cost to evaluate the cost-effectiveness of tensor compression at runtime. The related parameters are listed in Table 2. Given a tensor t with $Size^t$ and $Sparsity^t$ or P, we determine its cost-effectiveness of compression by comparing the swapping cost with compression T to the swapping cost without compression T'. If $T' > T$, a compression plan for the tensor t will be generated and forwarded to the swapping executor; otherwise, no compression is needed.

As shown in Fig. 1a, T' is the data transfer time that cannot be hidden from DNN propagation time (i.e., the portion with shade and slash in the timeline). Consequently, we use the Equation (3) to compute T'. $Hidden_f^t$ and $Hidden_b^t$ are the DNN forward and backward propagation times, respectively. They are collected by the *tensor profiler*. If the swapping latency can be hidden behind the DNN propagation time, the value of T' can be effectively 0

$$T' = \max\left(\frac{Size^t}{BW_{d2h}} - Hidden_f^t, 0\right) + \max\left(\frac{Size^t}{BW_{h2d}} - Hidden_b^t, 0\right). \tag{3}$$

Equation (4) computes the tensor swapping cost when compression is used. $Time_c^t$ and $Time_{dc}^t$ are determined by the tensor characteristics and compression algorithms. They are computed by the *tensor profiler* using a machine learning model as described in Section 5.4. O_f and O_b are the portion of the compressed data transfer time that cannot be effectively hidden from the DNN propagation time (as Equations (5)–(7)). If the compressed tensor is adequately small, $Time_c^t$ and $Time_{dc}^t$ will dominate in T.

$$T = Time_c^t + Time_{dc}^t + O_f + O_b \tag{4}$$

$$O_f = \max\left(\frac{Size_{comp}^t}{BW_{d2h}} - Hidden_f^t, 0\right) \tag{5}$$

$$O_b = \max\left(\frac{Size_{comp}^t}{BW_{h2d}} - Hidden_b^t, 0\right) \tag{6}$$

$$Size_{comp}^t = \begin{cases} Size^t \times \frac{P}{MAX}, & DNN \in Density \\ Size^t \times (1 - Sparsity^t), & DNN \in Sparsity \end{cases} \tag{7}$$

CSwap+ uses the swapping cost model in DNN training. At beginning of the DNN training, the *tensor profiler* collects the effective data transfer bandwidth of the PCIe link of the current system, judges the tensor type (i.e., sparse or dense), and records the tensor size ($Size^t$). Then it detects the tensor compression ratio and records the hidden latency ($Hidden^t$). Based on these data, the *execution advisor* makes a preliminary decision for all tensors. During the training, the tensor may change. To calculate the exact compression ratio, for sparse tensors, CSwap+ will collect the tensor sparsity in each epoch, while recording P in each epoch for dense tensors. The *execution advisor* then asks the *tensor profiler* for the latest results generated by the cost model, including $Time_c^t$ and $Time_{dc}^t$. T' and T are then re-computed for updating tensor compression decisions.

5.4 Prediction of (De)Compression Time

To dynamically determine the tensor compression plan, the *execution advisor* of CSwap+ needs to predict the compression time $Time_c^t$ and decompression time $Time_{dc}^t$ given tensor size, sparsity for sparse tensors (or P for dense tensors), and compression algorithms. We experimentally observe that the tensor size and compression ratio have a linear relationship with $Time_c^t$ and $Time_{dc}^t$. This is because the compression (decompression) time is dominated by the data searching time, which is greatly related to the compression ratio. Therefore, CSwap+ models the relationship offline using linear regression algorithms [49]. The (de)compression time model is then used to predict $Time_c^t$ and $Time_{dc}^t$ online. To have comprehensive coverage of tensor characteristics, we develop a synthetic tensor generator which can output tensors of different sizes and compression ratios.

Specifically, we use the following steps to build and deploy a (de)compression time model. We first collect data samples for training the time model. Each training sample includes the following measures: tensor size, tensor compression ratio, compression algorithm, $Time_c^t$ and $Time_{dc}^t$. For lossless compressors, we calculate the compression ratio with tensor sparsity (i.e., $Sparsity^t$). In the experiments, we find that randomly sampling the tensor size and compression ratio will likely over-fit the models. To solve the problem, we only train models using samples whose sparsity falls between 20% and 80% because we observe that tensor sparsity is mostly located in this range as shown in Fig. 2. For lossy compressors, we use $1 - P/MAX$ to calculate the compression ratio. We scan all the possible values of P to make the performance model cover all compression ratios because each of them may appear.

Second, to improve the model accuracy, CSwap+ trains n sub-models. Sub-model i is trained using samples whose compression ratio is in $[Ratio_{base} + R * i/n, Ratio_{base} + R * (i+1)/n)$, where $0 \leq i <$n, and R is a percentage to represent the compression ratio range. For example, for the sparse tensors with range [20%, 80%] sparsity, we will set $Ratio_{base}$ as 20% and R as 60%. For dense tensors, we will set $Ratio_{base}$ as 0 and R as 100%. After training, the sub-models are combined to form a holistic model which is then deployed for inference. In training, we vary the tensor size from 20 MB to 2000 MB in addition to the changes of tensor sparsity or parameter P. Third, the (de)compression time model is stored in the in-memory database for retrieval.

6 Evaluation

To demonstrate the performance of CSwap+, we implement its prototype in Torch 1.5.1. To achieve parallel swapping, we create an asynchronized cuda stream using *cudaStreamCreateWithFlags()* and use *cudaMemcpyAsync()* to transfer data. Furthermore, we add *GPUcompression()* as the kernel compression function into CSwap+, then set *GPUdecompression()* for decompressing. However, the frequent GPU/CPU memory allocation/free decreases the performance severely. To solve this problem, we use memory pool functions in Torch, *getCUDADeviceAllocator ()* and *getPinnedMemoryAllocator()* to avoid using the expensive *cudaMalloc()* and *cudaMallocHost()* functions.

Experimental Platforms. Our experiments are conducted on two CPU-GPU hybrid servers. The first is equipped with a 2.60 GHz Intel(R) Xeon(R) Gold 6126 CPU and 32 GB main memory. Besides that, it has an NVIDIA Tesla V100 GPU with 32 GB GPU memory. The second server has two 2.10 GHz Intel(R) Xeon(R) Gold 5218R CPUs, 128 GB main memory, an RTX 2080Ti GPU, and 11 GB GPU memory. The CPU and GPU on a server are connected via the PCIe 3.0×16 bus. The first server (V100) has a higher peak compute capability than the second one (2080Ti). In both servers, we run Ubuntu-18.04, CUDA 10.0.13, CuDNN 7, and Torch 1.5.1.

Workloads and Datasets. To show the effectiveness of our approach for extensive workloads, we evaluate CSwap+ with four *linear* sparse DNN models (i.e., AlexNet [1], Plain20 [50], VGG16 [26], and MobileNet [51]), and two *nonlinear* models (i.e., ResNet [27] and SqueezeNet [52]). We use ReLU and Leaky ReLU Activations in the DNN models to generate sparse and dense tensors, respectively. In the evaluation section, we name the models with ReLU as *sparse DNNs* and the models with Leaky ReLU as *dense DNNs*. We tune the parameters of these DNN models (e.g., learning rate and optimizer) based on the specifications in the related papers and documents [1], [26], [27], [50], [51], [52] and configured training batch sizes are shown in Table 3.

TABLE 3
Batch Size Configurations for Different Models, GPUs(V100 or 2080Ti), and Datasets (CIFAR10 or ImageNet)

DNN Model	CIFAR10-V100	ImageNet-V100	CIFAR10-2080Ti	ImageNet-2080Ti
AlexNet	2560	512	2560	256
VGG16	2560	128	2560	32
MobileNet	2560	128	1280	32
Plain20	2560	32	1024	-
ResNet	2560	64	1280	16
SqueezeNet	2560	512	1280	128

We use two representative datasets to cover different data sizes. The CIFAR10 [53] dataset is a collection of 60, 000 (50, 000 for training and 10, 000 for testing) labeled color images (32×32 pixels each). The ImageNet [54] is a large dataset. It has 1.4 million 224×224 pixel images across 21841 non-empty synsets.

DNN Frameworks for Comparison. We compare CSWAP+ with the state-of-the-art GPU memory swap frameworks, vDNN [14], vDNN++ [21], and cDMA [17]. The vDNN scheme offloads all convolution input tensors from GPUs to CPUs and prefetches them back through overlapping data transfer with computation. However, there is no data (de)compression in the tensor swapping between GPUs and CPUs. The vDNN++ scheme only compresses sparse tensors on host CPUs to reduce the size of pinned memory. Although there are two other techniques in vDNN++, we omit them in our implementation since they are orthogonal to our design of CSWAP+. To make (de)compression more efficient, we use 64 threads in CPUs to compress and decompress the tensors when the sparsity is more than 60%. Because cDMA [17] relies on the specific (de)compression units which are not available in markets, we implement an emulated cDMA scheme, i.e., static compression (SC), which uses GPUs to emulate the (de)compression units in memory controllers and applies the lossless and lossy compression algorithms without effectiveness analysis. In the evaluation, we compare CSWAP+ to SC instead of cDMA.

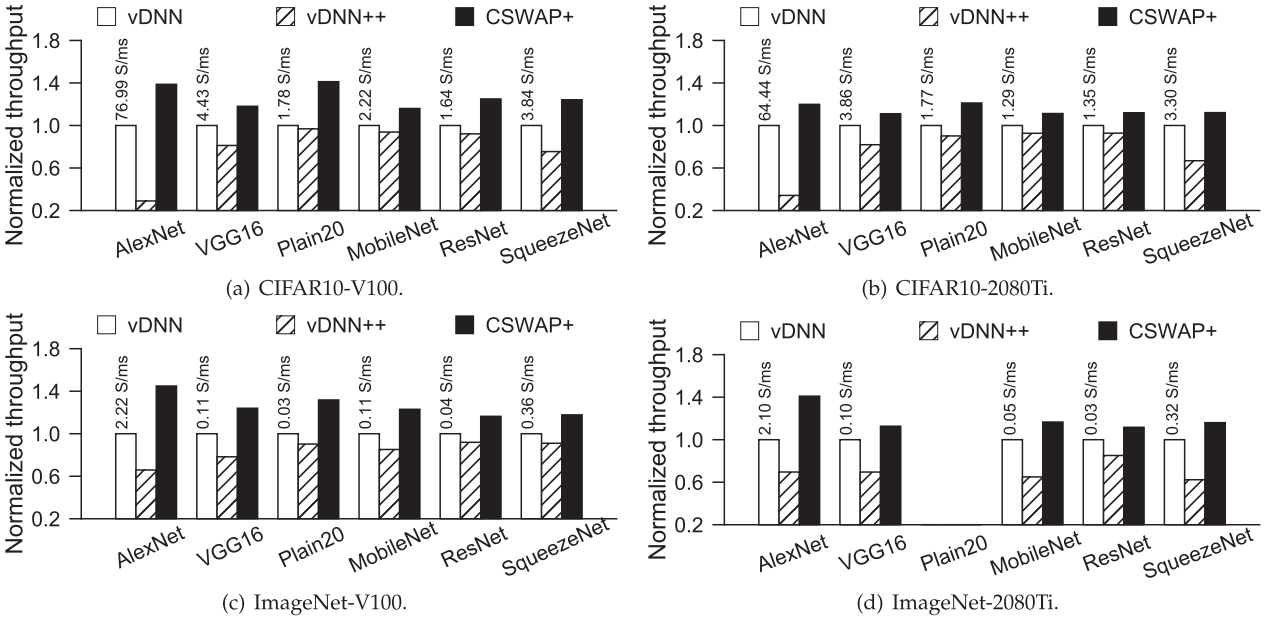
6.1 General Results

Figs. 5 and 6 show the system throughputs of different frameworks on different GPUs and datasets across sparse DNNs and dense DNNs, respectively. For comparison, the throughput (samples/ms) is normalized to that of vDNN.

6.1.1 Results for Sparse DNNs

Fig. 5a shows the system throughput when training the sparse DNNs on V100 with the CIFAR10 dataset. Overall, CSWAP+ outperforms vDNN and vDNN++ by 25% and 190% on average with sparse DNNs. We also have the following observations. First, compared to vDNN, CSWAP+ improves the model training time by up to 31% for sparse models. CSWAP+ is better than vDNN because it uses dynamic compression to reduce tensor transfer time while vDNN transfers the original tensors regardless of their compression ratio, leading to high data transfer cost in swapping. Second, compared to vDNN++ on sparse models, CSWAP+ increases system throughput by up to 470% on AlexNet and reduces the model training time by up to 445ms on ResNet. This is because vDNN++ only compresses and decompresses tensors on the host side after tensor swapping. It does not reduce data transfer time by compressing tensors in GPUs.

Fig. 5b shows the system throughput when training the models with the same dataset CIFAR10 but on a different GPU 2080Ti, which has lower peak compute capability than V100. The system throughput is decreased by 9% on average for all the sparse models compared to the results on



(a) CIFAR10-V100. (b) CIFAR10-2080Ti.

(c) ImageNet-V100. (d) ImageNet-2080Ti.

Fig. 5. The performance comparison of four different GPU virtual memory management frameworks. We conduct the experiments with six sparse DNN models on two datasets (CIFAR10 and ImageNet) and two GPUs (V100 and 2080Ti). The caption of each subfigure denotes its dataset and GPU configuration.

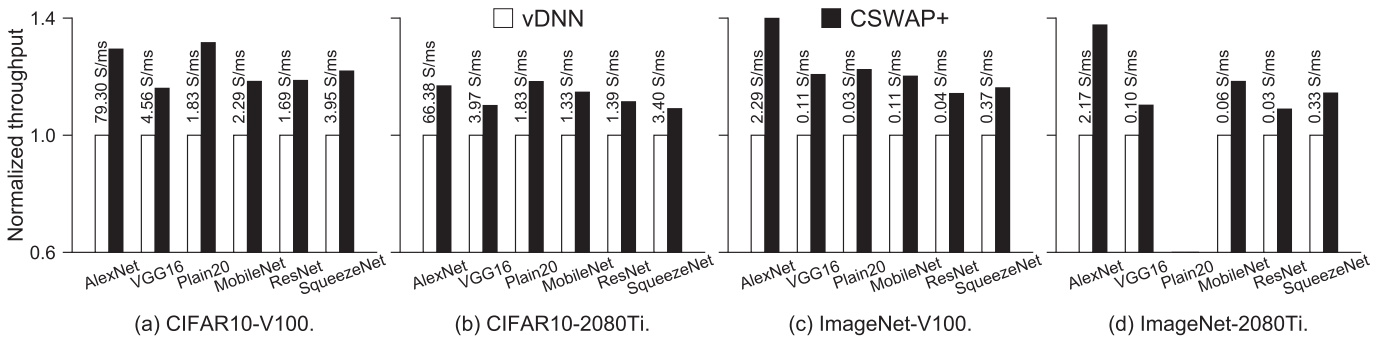


Fig. 6. The performance comparison of CSwap+ and vDNN on dense DNNs. We conduct the experiments with six models using Leaky ReLU activations on two datasets (CIFAR10 and ImageNet) and two GPUs (V100 and 2080Ti).

V100. The decreased throughput stems from two reasons. First, effective data transfer bandwidths are not the same for the two GPU platforms because they have different GPU and CPU memory configurations. We have examined such bandwidths using the NVIDIA *bandwidthTest* tool [55]. The effective host to device and device to host bandwidths are 10.6 GB/s and 11.7 GB/s respectively on V100 and 11.8 GB/s and 12.9 GB/s respectively on 2080Ti. The higher effective data transfer bandwidth in 2080Ti alleviates the data transfer bottleneck in the DNN training process. Second, 2080Ti has a lower performance than V100. The computation time of DNN models on 2080Ti is relatively longer. As a result, CSwap+ has a better chance to hide the data transfer time behind the computation time, decreasing the data transfer overhead.

To show the effectiveness of CSwap+ on different datasets, we train the models on the large dataset ImageNet. Figs. 5c and 5d show the system throughput of CSwap+ for all the models on V100 and 2080Ti, respectively. Overall, compared to the CIFAR10 dataset, ImageNet leads to similar performance trends on the two GPU platforms. The experiments show that the model training time is reduced by 20.3% and 16.9% on average on V100 and 2080Ti respectively. These results show that our approach is effective for the ImageNet dataset. In Fig. 5d, we do not display the performance results for the Plain20 model. This is because Plain20 is a large model and 2080Ti only has 11GB GPU memory, which cannot meet the memory requirement of Plain20 even when the batch size is set to one.

6.1.2 Results for Dense DNNs

Fig. 6 shows the throughput results on dense DNNs under CSwap+ and vDNN frameworks. In the figure, we do not show the results of vDNN++ because vDNN++ cannot work for dense tensors.

Fig. 6a shows the results on V100 with the CIFAR10 dataset. We have two observations. First, compared to vDNN, CSwap+ achieves an average 22.5% throughput improvement with lossy compression. The improvement occurs because CSwap+ reduces tensor transfer cost in swapping through dynamic lossy compression. Second, CSwap+ brings different improvements for different models. For example, CSwap+ improves the training throughput by 29% and 31% for AlexNet and Plain20, respectively, while it only achieves the improvement by up to 21% on other dense models. The main reason is that the data swapping time dominates in the training time for AlexNet and Plain20 (71% and 73% of the total model training time). For other models, the data transfer time accounts for less than 50% of the training time.

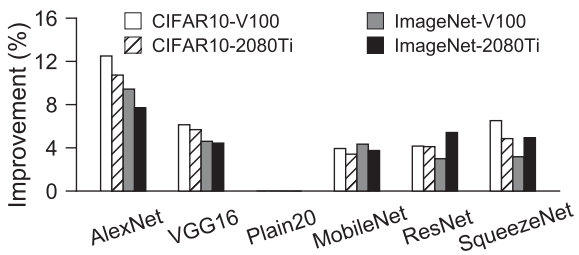
Fig. 6b shows the throughputs of different swapping frameworks on 2080Ti under the same dataset CIFAR10. Similar to the previous results in Fig. 6a, CSwap+ still outperforms vDNN: CSwap+ improves the DNN training throughput by 13% on average. However, CSwap+ achieves relatively less throughput improvement on 2080Ti compared to the improvements on V100 (13% versus 22.5%). The improvement decreases mainly because the lower GPU performance on 2080Ti increases the computing time and thus brings more chances to hide the data transfer time, making the benefits of data compression in data swapping less significant.

Figs. 6c and 6d show the experimental results on ImageNet. CSwap+ yields similar trends on the two GPU platforms with ImageNet compared to CIFAR10. Overall, The dense model training time is reduced by 19.7% and 16.5% on average under V100 and 2080Ti, respectively. These results indicate that CSwap+ is effective for the ImageNet dataset with lossy compression. Moreover, we also observe that CSwap+ obtains more performance benefits on the DNNs whose swapping times dominate their overall training times than other models. For example, CSwap+ performs the best on AlexNet and improves its training throughput by 43% and 37% on V100 and 2080Ti, respectively.

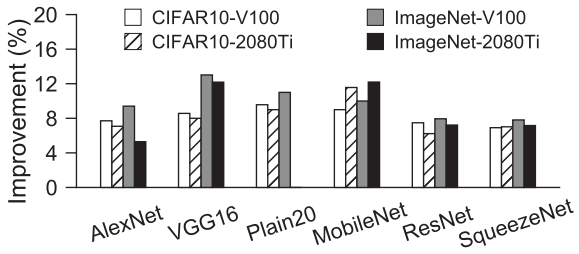
6.2 Effectiveness of Dynamic Tensor Compression

To further evaluate the effectiveness of CSwap+, we compare its training time to that with SC, which is a replica of cDMA using GPUs. Note that cDMA cannot work for dense tensors, so we empower cDMA with the capability of compressing dense tensors by replacing the lossless algorithm with a lossy compressor to analyze the effectiveness of the dynamic compression in CSwap+. While CSwap+ performs (de)compression in tensor swapping based on the cost-effectiveness analysis of tensor compression, SC blindly compresses all tensors in DNNs by switching off effectiveness analysis. We train the models on V100 and 2080Ti with CIFAR10 and ImageNet.

We show the experimental results in Fig. 7. We can observe that for the models containing sparse tensors, CSwap+ improves the performance by 5.5% and 5.1% on average compared to SC for all the models except Plain20



(a) Lossless compression.

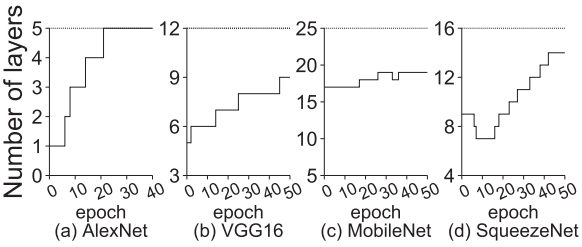


(b) Lossy compression.

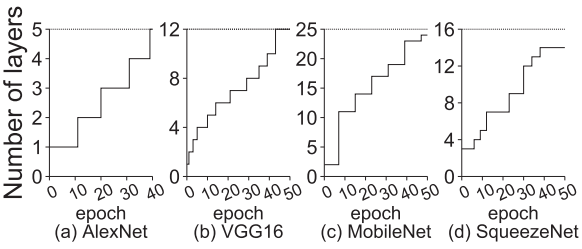
Fig. 7. Performance improvement of CSWAP+ over the static compression (SC) scheme.

on V100 and 2080Ti, respectively. The maximal performance improvement brought by CSWAP+ can be 12.5% and 10.7% on the two GPUs respectively. Because tensors in all ReLU layers of Plain20 are sparse and have a larger size on average than other models, CSWAP+ determines that tensors in all the layers of Plain20 need to be compressed. As a result, CSWAP+ has the same performance as SC. For the models with dense tensors, CSWAP+ improves the performance by 9% and 8.4% on average on the two GPU platforms. Similarly, since the 11GB memory on 2080Ti GPU cannot support the Plain20 training, we do not show the result with ImageNet on 2080Ti GPU.

Fig. 8 shows the number of layers whose tensors are compressed with CSWAP+ during the training of AlexNet, VGG16, MobileNet, and SqueezeNet. For lossless compression, we

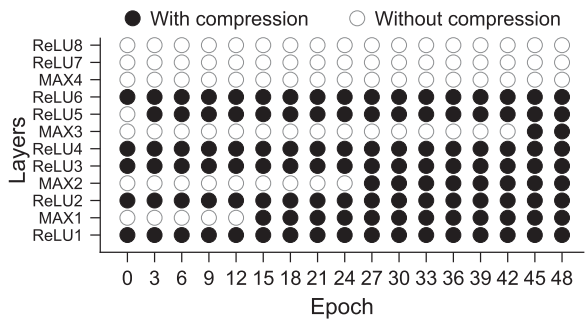


(a) Lossless compression.

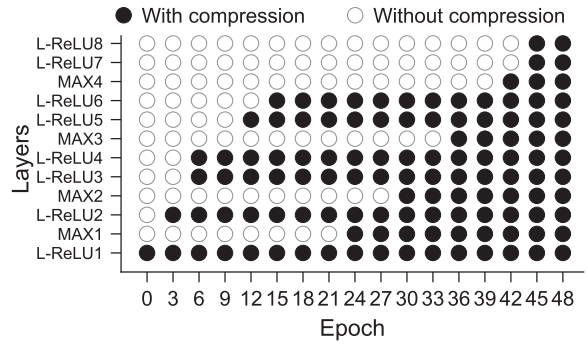


(b) Lossy compression.

Fig. 8. The number of layers executing tensor compression for four DNN models for every epoch in their training processes.



(a) Lossless compression.



(b) Lossy compression.

Fig. 9. VGG16 layer-wise compression detail. The x-axis denotes the training epoch and y-axis represents all the layers in the model. The black dot denotes that the tensor of this layer needs to be compressed while the white dot means that the tensor is only transferred without compression.

have two observations from Fig. 8a. First, the number of such layers tends to increase while epochs are increased. Let's use VGG16 as an example, as shown in Fig. 9a, the number of its layers executing tensor (de)compression is increased from 5 at epoch 0 to 9 at epoch 48. This is because as epochs are increased, the tensors in more layers become sparse enough to trigger the (de)compression operations in CSWAP+ for reducing data transfer overhead in swapping. However, some layers may never be compressed during the model training. Taking the VGG16 as an example, since the tensor in MAX4 always has low sparsity (i.e., lower than 45%) and the tensors in ReLU7 and ReLU8 are relatively small, which make the compression cost-ineffective, these layers are never be compressed, as shown in Fig. 9a. Second, models have distinct characteristics that lead to varied tensor compression decisions. For instance, the number of layers executing tensor (de)compression for MobileNet does not change too much as the epochs are increased because its tensor sparsity changes slightly as shown in Fig. 8a. For SqueezeNet, there exist two tensors whose sparsity is decreased between epoch 5 and epoch 17 and is increased after epoch 17. The reason for the fluctuation is that their model convergence may change during training.

Fig. 8b shows the experimental results for lossy compression. We observe that as the epoch number increases, more tensors are compressed for lossy compression. This is because CSWAP+ adapts a gradually increasing compression ratio (by decreasing the parameter P) during the training, which will reduce the swapping time and increase the swapping benefit, thus more tensors are worth being compressed

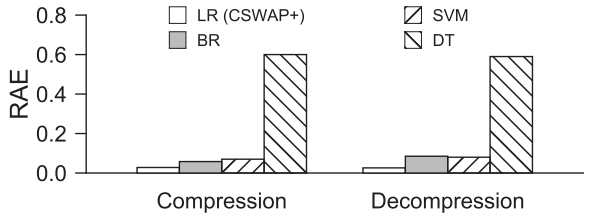


Fig. 10. The accuracy of (de)compression time predication using LR, BR, SVM, and DT models.

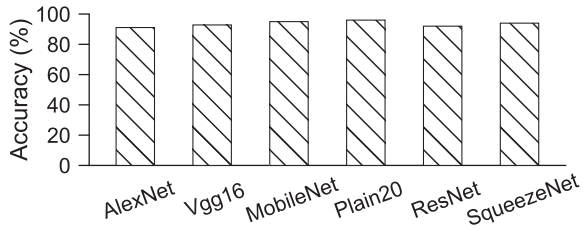


Fig. 11. The compression decision accuracy based on the LR model.

as the number of epoch increases. Fig. 9b shows the details of layer-wise compression on VGG16. It is also notable that some tensors are not compressed from beginning to end (i.e., one tensor for MobileNet and two for SqueezeNet). This is because the swapping of these tensors can be fully hidden into the normal DNN training without compression. Besides the four models, the other three models show their own characteristics concerning the number of layers having compressed tensors. Because of the space limitation, we do not show their curves in Fig. 8.

6.3 (De)Compression Time Model Verification

An important component of CSwap+ is the (de)compression time model, which influences the effectiveness of its execution advisor (Section 5). We compare the linear regression (LR) model in CSwap+ for (de)compression time modeling to three other regression models including Bayesian regression (BR) [56], support vector machine (SVM) [57], and decision tree (DT) [58] from scikit-learn [59]. To evaluate prediction accuracy, we use relative absolute error (RAE) which is defined as $\frac{\sum_{i=1}^{N}|\hat{y}_i - y_i|}{\sum_{i=1}^{N}|\bar{y}_i - y_i|}$, where N is the size of test samples, $\hat{y}_i$ and y_i are predicted and measured values respectively, and $\bar{y}_i$ is the mean value of y_i.

CSwap+ currently supports four lossless compression algorithms, i.e., RLE, CSR, LZ4, and ZVC, and a lossy compression algorithm, i.e., ZFP. For each lossless algorithm, we generate a total of 3000 sparse tensors, whose compression ratio ranges between 20% to 90%. For the lossy algorithm, we generate the same number of samples with all compression ratios (i.e., 0 to 100%). The sample sizes are varied from 20 MB to 2000 MB as real DNN training tensors. We train all the models with the same collected 3000 test samples for fairness. As shown in Fig. 10, LR achieves the best prediction accuracy. Its mean relative absolute error for compression and decompression time prediction is only 2.7% on average, which is 61% and 64% smaller than those of BR and SVM respectively.

We also evaluate the compression decision accuracy based on the swapping cost model for both sparse and dense tensors, which relies on the LR model. CSwap+ uses the swapping cost model to make a tensor compression decision. If the decision based on the swapping cost model matches the decision based on the measured time at runtime for a tensor compression, we regard the decision as correct. We define decision accuracy as the ratio of the number of correct decisions to the number of all decisions. We train the six DNN models and show the decision accuracy in Fig. 11. We observe that the decision accuracy using the swapping cost model is 93.5% on average.

6.4 DNN Training Accuracy Under Lossy Compression

In CSwap+, we use a sliding down scheme to set the compressor parameter from *Top* to *Bottom* for dense tensors during the training (Section 5.2.2). To verify the effectiveness of such dynamic lossy compression, we compare its accuracy with four counterparts: the default framework without any accuracy loss, ZFP parameter fixed schemes (i.e., ZFP-64 and ZFP-32), and the Inverse Scheme which increases the value of lossy compressor parameter P as the epoch number is increased.

We train the VGG16 model using the Leaky ReLU activation and record the training accuracy at the end of each epoch. As Fig. 12 shows, CSwap+ achieves the comparable DNN accuracy as the default framework. Furthermore, CSwap+ brings better DNN accuracy than the schemes with a fixed parameter: it has 89.3% accuracy while ZFP-64 and ZFP-32 achieve 88.2% and 87.5% in the end, respectively. CSwap+ outperforms the two counterparts because CSwap+ adopts the gradually decreasing parameter, which may always lead to negligible accuracy loss during the entire training process, while the two baselines use a fixed parameter, which may lead to more accuracy loss in the beginning training epochs.

To explain why we design CSwap+ as a sliding down scheme, we also compare CSwap+ with the Inverse Scheme. It is notable that CSwap+ always achieves better accuracy than the Inverse Scheme in all the training periods (i.e., 1.7% more accuracy). CSwap+ achieves higher accuracy than Inverse Scheme because the DNN training accuracy is more sensitive to the float accuracy in the beginning epochs and becomes less sensitive to it in the latter epochs. The sliding down scheme used in CSwap+ gradually decreases the

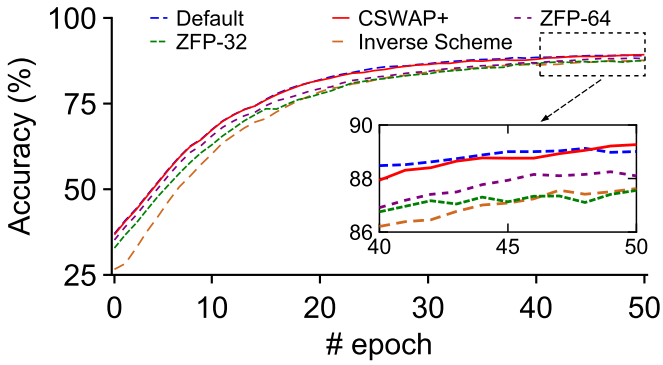


Fig. 12. The VGG16 accuracy with different compression schemes on CIFAR10. ZFP-32 and ZFP-64 denote that the compressor parameter is fixed at 32 and 64.

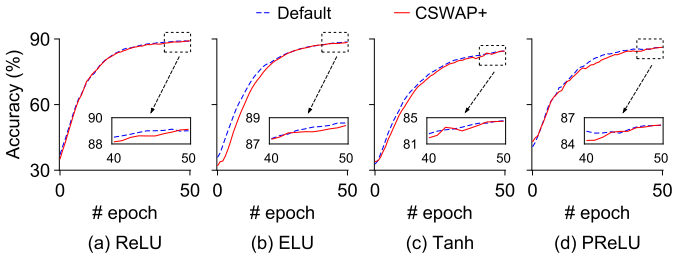


Fig. 13. The accuracy of VGG16 using different activations on CIFAR10. The Default baseline means the DNN training without enabling CSwap+.

value of P and increases the compression ratio. In this way, it will take less float accuracy loss at the beginning of the training, leading to less training accuracy loss. In contrast, the Inverse Scheme gradually increases the value of P, which will cause more float accuracy losses in the beginning of the training process and thus lead to more training accuracy degradation. We only show the result of VGG16 because other DNNs show a similar trend.

6.5 DNN Training Accuracy for Different Activations

There are several popular activation functions used in deep learning, such as ReLU, ELU, Tanh, and PReLU. The first activation generates sparse tensors and the latter three generate dense tensors. To verify the effectiveness of the lossy compression in CSwap+, we evaluate the training accuracy of DNNs with the lossy compression in CSwap+ for different activation functions.

We train VGG16 with ReLU, ELU, Tanh, and PReLU activations for 50 epochs. We compare CSwap+ to Default, which does not use compression in swapping. As shown in Fig. 13, for all activations, CSwap+ achieves comparable accuracy as Default. Specifically, it achieves the average accuracy of 87.01% across the four DNNs while Default yields 87.05% accuracy. This result shows that CSwap+ works effectively for all the activations and thus can benefit different DNNs. Moreover, we observe that the lossy compression in CSwap+ also works well for sparse tensors. For example, applying the lossy compression to the sparse tensors generated by ReLU causes nearly no accuracy loss, as shown in Fig. 13a. In this paper, we use the lossless compression for sparse tensors because the performance improvement of the lossless compressor is larger than that of the lossy compressor, as shown in Figs. 5 and 6.

6.6 Overhead Discussion

CSwap+ introduces the following overheads. However, the overheads are either negligible compared to the overall training time or can be amortized over the training.

Runtime Overhead. The profiling of tensor characteristics in CSwap+ introduces overhead to the model training process. To make an effective decision with minimum runtime overhead, CSwap+ is set with a fine-grained detecting cycle (i.e., each epoch). Because the hidden time and tensor size do not change across epochs, CSwap+ only needs to update $Sparsity^t/P$, $Time_c^t$, and $Time_{dc}^t$ periodically to make dynamic decisions. For sparse DNNs, CSwap+ utilizes GPU multi-cores to profile tensor sparsity (e.g., only 8 ms overhead every 10 sec for training VGG16). Besides, one prediction of $Time_c^t$ or $Time_{dc}^t$ is only 1 ms which is negligible compared to the overall training time.

Offline Overhead. CSwap+ needs to determine the compression ratio of each epoch for dense DNN models. It calculates the parameter range once by running Algorithm 1 within an average of 280ms. Further, CSwap+ needs to train a (de)compression time model of tensor compression offline as discussed in Section 5.4. It only takes on average 6 minutes to generate all training samples and 25ms to build the time model because of the lower complexity of the linear regression method used in the paper.

7 Conclusion

In this paper, we present CSwap+, a self-tuning compression framework to reduce data transfer overhead in tensor swapping. First, it does not require additional (de)compression units in memory controllers of GPUs or expertise in setting GPU parameters for effectively executing (de)compression. Second, it integrates both lossless and lossy compressions into the framework and accelerates DNN training with negligible model accuracy loss. Third, it uses the cost model of tensor swapping to selectively apply (de)compression to tensors according to the cost-effectiveness of tensor compression at runtime. We experimentally demonstrate that CSwap+ offers lower swapping latency and higher training throughput for both sparse and dense DNN models than the existing tensor swapping frameworks.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, pp. 84–90, 2017.
[2] S. Zhang, L. Yao, A. Sun, and Y. Tay, "Deep learning based recommender system: A survey and new perspectives," *ACM Comput. Surv.*, vol. 52, pp. 1–38, 2019.
[3] H. Ze, A. Senior, and M. Schuster, "Statistical parametric speech synthesis using deep neural networks," in *Proc. Int. Conf. Acoust. Speech Signal Process.*, 2013, pp. 7962–7966.
[4] J. Devlin, M.-W. Chang, K. Lee, K. T. Google, and A. I. Language, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.
[5] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4: Inception-ResNet and the impact of residual connections on learning," 2016, *arXiv:1602.07261*.
[6] N. Shazeer *et al.*, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," 2017, *arXiv:1701.06538*.
[7] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.
[8] TensorFlow, "Introduction to tensors," 2020. [Online]. Available: https://www.tensorflow.org/guide/tensor
[9] NVIDIA, "NVIDIA V100 Tensor Core GPU," 2020. [Online]. Available: https://www.nvidia.com/en-us/data-center/v100/
[10] Z. Zhang, C. Chang, H. Lin, Y. Wang, R. Arora, and X. Jin, "Is network the bottleneck of distributed training?," in *Proc. Workshop Netw. Meets AI ML*, 2020, pp. 8–13.
[11] C.-C. Huang, G. Jin, and J. Li, "SwapAdvisor: Push deep learning beyond the GPU memory limit via smart swapping," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2020, pp. 1341–1355.
[12] J. Ren, J. Luo, K. Wu, M. Zhang, and D. Li, "Sentinel: Runtime data management on heterogeneous main memory systems for deep learning," 2019, *arXiv:1909.05182*.
[13] L. Wang *et al.*, "SuperNeurons: Dynamic GPU memory management for training deep neural networks," in *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2018, pp. 41–53.

[14] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "VDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *Proc. Annu. Int. Symp. Microarchit.*, 2016, pp. 1–13.
[15] M. Hildebrand, J. Khan, S. Trika, J. Lowe-Power, and V. Akella, "AutoTM: Automatic tensor movement in heterogeneous memory systems using integer linear programming," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2020, pp. 875–890.
[16] T. D. Le, Y. Negishi, H. Imai, and K. Kawachiya, "TFLMS: Large model support in TensorFlow by graph rewriting," 2018, *arXiv:1807.02037*.
[17] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing DMA engine: Leveraging activation sparsity for training deep neural networks," in *Proc. Int. Symp. High-Perform. Comput. Archit.*, 2018, pp. 78–91.
[18] P. Chen *et al.*, "CSWAP: A self-tuning compression framework for accelerating tensor swapping in GPUs," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2021, pp. 271–282.
[19] A. F. Agarap, "Deep learning using rectified linear units (ReLU)," 2018, *arXiv:1803.08375*.
[20] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," 2015, *arXiv:1505.00853*.
[21] S. B. Shriram, A. Garg, and P. Kulkarni, "Dynamic memory management for GPU-based training of deep neural networks," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2019, pp. 200–209.
[22] A. H. Robinson and C. Cherry, "Results of a prototype television bandwidth compression scheme," *Proc. IEEE*, vol. 55, no. 3, pp. 356–364, Mar. 1967.
[23] A. Alabaichi, A. H. Alhusiny, and E. MohammedThabit, "A novel compressing a sparse matrix using folding technique," *Res. J. Appl. Sci. Eng. Technol.*, vol. 14, pp. 310–319, 2017.
[24] NVIDIA, "NVIDIA/nvcomp: A library for fast lossless compression/decompression on the GPU," 2020. [Online]. Available: https://github.com/NVIDIA/nvcomp
[25] Pytorch, "Pytorch/pytorch: Tensors and dynamic neural networks in python with strong GPU acceleration," 2020. [Online]. Available: https://github.com/pytorch/pytorch
[26] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arxiv: 1409.1556*.
[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[28] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (ELUs)," in *Proc. Int. Conf. Learn. Representations*, 2016, pp. 1–14.
[29] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2015, pp. 1026–1034.
[30] Quora, "Dead neurons," 2020. [Online]. Available: https://www.quora.com/What-are-the-disadvantages-of-using-the-ReLu-when-using-Neural-Networks
[31] Amihaeseisergiu, "ResNet with ELU," 2022. [Online]. Available: https://github.com/Amihaeseisergiu/Cifar-10-ResNet-ELU-Cutout
[32] Tianxiaomo, "YoLo," 2022. [Online]. Available: https://github.com/Tianxiaomo/pytorch-YOLOv4/blob/master/models.py
[33] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, 2010, pp. 248–255.
[34] PyTorch, "Pytorch/vision," 2020. [Online]. Available: https://github.com/pytorch/vision/tree/master/torchvision
[35] W. C. Wang, Y. H. Chang, T. W. Kuo, C. C. Ho, Y. M. Chang, and H. S. Chang, "Achieving lossless accuracy with lossy programming for efficient neural-network training on NVM-based systems," *ACM Trans. Embedded Comput. Syst.*, vol. 18, 2019, Art. no. 68.
[36] ZFP, "ZFP compression algorithm," 2019. [Online]. Available: https://github.com/LLNL/zfp
[37] S. Patel, T. Liu, and H. Guan, "FreeLunch: Compression-based GPU memory management for convolutional neural networks," in *Proc. IEEE/ACM Workshop Memory Centric High Perform. Comput.*, 2021, pp. 1–8.
[38] X. Chen, D. Z. Chen, and X. S. Hu, "MoDNN: Memory optimal DNN training on GPUs," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2018, pp. 13–18.
[39] X. Peng *et al.*, "Capuchin: Tensor-based GPU memory management for deep learning," in *Proc. Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2020, pp. 891–905.
[40] Wikipedia, "LZ4 (compression algorithm)," 2020. [Online]. Available: https://en.wikipedia.org/wiki/LZ4_(compression_algorithm)
[41] P. Deutsch, "RFC1952: GZIP file format specification version 4.3," *RFC Editor*, pp. 1–13, 1996.
[42] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE Trans. Vis. Comput. Graphics*, vol. 12, no. 5, pp. 1245–1250, Sep./Oct. 2006.
[43] Blosc, "Blosc, an extremely fast, multi-threaded, meta-compressor library," 2017. [Online]. Available: https://github.com/Blosc/c-blosc
[44] J. Tian *et al.*, "CuSZ: An efficient GPU-based error-bounded lossy compression framework for scientific data," in *Proc. ACM Int. Conf. Parallel Archit. Compilation Techn.*, 2020, pp. 3–15.
[45] Y. Cheng, F. X. Yu, R. S. Feris, S. Kumar, A. Choudhary, and S.-F. Chang, "An exploration of parameter redundancy in deep networks with circulant projections," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2015, pp. 2857–2865.
[46] S. Yang, W. Chen, X. Zhang, S. He, Y. Yin, and X.-H. Sun, "AUTO-PRUNE: Automated DNN pruning and mapping for ReRAM-based accelerator," in *Proc. ACM Int. Conf. Supercomput.*, 2021, pp. 304–315.
[47] S. Jin, S. Di, X. Liang, J. Tian, D. Tao, and F. Cappello, "DeepSZ: A novel framework to compress deep neural networks by using error-bounded lossy compression," in *Proc. 28th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2019, pp. 159–170.
[48] S. He *et al.*, "HOME: A holistic GPU memory management framework for deep learning," *IEEE Trans. Comput.*, to be published, Jun. 09, 2022, doi: 10.1109/TC.2022.3180991.
[49] D. C. Montgomery, E. A. Peck, and G. G. Vining, *Introduction to Linear Regression Analysis*. Hoboken, NJ, USA: Wiley, 2012.
[50] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "AMC: AutoML for model compression and acceleration on mobile devices," in *Proc. Eur. Conf. Comput. Vis.*, 2018, pp. 248–255.
[51] A. G. Howard *et al.*, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*.
[52] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," 2016, *arXiv:1602.07360*.
[53] A. Krizhevsky, "Learning multiple layers of features from tiny images," *Citeseer*, pp. 1–60, 2009.
[54] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.
[55] NVIDIA, "NVIDIA/Cuda-samples: Samples for CUDA developers which demonstrates features in CUDA toolkit," 2020. [Online]. Available: https://github.com/NVIDIA/cuda-samples
[56] A. E. Raftery, D. Madigan, and J. A. Hoeting, "Bayesian model averaging for linear regression models," *J. Amer. Statist. Assoc.*, vol. 92, pp. 179–191, 1997.
[57] W. S. Noble, "What is a support vector machine?," *Nature Biotechnol.*, vol. 24, pp. 1565–1567, 2006.
[58] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE Trans. Syst., Man, Cybern.*, vol. 21, no. 3, pp. 660–674, May/Jun. 1991.
[59] sklearn, "scikit-learn: Machine learning in Python — scikit-learn 0.23.2 documentation," 2020. [Online]. Available: https://scikit-learn.org/stable/

Ping Chen is currently working toward the PhD degree in the College of Computer Science and Technology, Zhejiang University, China. His research focuses on intelligent computing and memory management for AI systems.

Shuibing He (Member, IEEE) received the PhD degree in computer science and technology from the Huazhong University of Science and Technology, in 2009. He is now a ZJU100 Young professor with the College of Computer Science and Technology, Zhejiang University, China. His research areas include Intelligent computing, high-performance computing, memory, and storage systems. He is a member of the ACM.

Xuechen Zhang (Member, IEEE) received the MS and PhD degrees in computer engineering from Wayne State University. He is currently an associate professor with the School of Engineering and Computer Science, Washington State University Vancouver. His research interests include the areas of file and storage systems and high-performance computing. He is a member of the ACM.

Shuaiben Chen received the MS degree from the College of Computer Science and Technology, Zhejiang University, China. His research areas include intelligent computing and systems for AI.

Peiyi Hong is currently working toward the MS degree in the College of Computer Science and Technology, Zhejiang University, China. Her research areas include intelligent computing and systems for AI.

Yanlong Yin received the PhD degree in computer science from the Illinois Institute of Technology, in 2014. He is now the project director of cloud computing platform with the R&D Department, Beijing Institute of Open Source Chip, China. His research interests include parallel computing, cloud computing, and parallel file systems.

Xian-He Sun (Fellow, IEEE) received the BS degree in mathematics from Beijing Normal University, China, in 1982, and the MS and PhD degrees in computer science from Michigan State University, in 1987 and 1990, respectively. He is a distinguished professor with the Department of Computer Science, Illinois Institute of Technology (IIT), Chicago. His research interests include parallel and distributed processing, memory and I/O systems.