CSWAP: A Self-Tuning Compression Framework for Accelerating Tensor Swapping in GPUs

Ping Chen^{+}, Shuibing He^{+*}, Xuechen Zhang†, Shuaiben Chen^{+}
Peiyi Hong^{+}, Yanlong Yin‡, Xian-He Sun*, Gang Chen^{+}
$^{+}$ *Zhejiang University,* † *Washington State University,* ‡ *Zhejiang Lab,* * *Illinois Institute of Technology*

Abstract—**Graphic Processing Units (GPUs) have limited memory capacity. Training popular deep neural networks (DNNs) often requires a larger amount of memory than that a GPU may have. Consequently, training data needs to be swapped between CPUs and GPUs. 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 two major issues: sub-optimal compression performance for varying tensor sparsity and sizes and lack of portability because its implementation requires additional (de)compression units in memory controllers.**

We propose a self-tuning tensor compression framework, named CSWAP, for improving the virtual memory management of GPUs. It has high portability and is minimally dependent on GPU architecture features. Furthermore, its runtime only applies compression on tensors that are deemed to be cost-effective considering their sparsity and size and the characteristics of compression algorithms. Finally, our framework is fully automated and can customize the compression policy for different neural network architectures and GPU architectures. Our experimental results using six representative memory-intensive DNN models show that CSWAP reduces tensor swapping latency by up to 50.9% and reduces the DNN training time by 20.7% on average with NVIDIA V100 GPUs compared to vDNN.

Index Terms—**DNN, GPU, Tensor, Swapping, Data Compression**

I. INTRODUCTION

Deep Neural Networks (DNNs) have recently gained unprecedented success 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]–[9]. For example, the latest BERT model needs more than 70 GB memory during the training period with batch size 64 [7]. The newest language model presented by Google has 137 billion parameters and requires more than 100 GB memory for training [10]. Additionally, evidence shows that the number of neural network parameters in models has nearly doubled every 2.4 years since the 80s [11]. These trends lead to a higher memory demand for training future DNN models.

To accelerate model training and inference of DNNs, accelerators such as Graphic Processing Units (GPUs) are widely used for high-performance tensor computation [12]–[14]. 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 on-board memory, which is inadequate for training the BERT model which consumes up to 73 GB memory [15]. 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 [16]. Prior works for scaling up swap intermediate tensors between GPUs and CPUs in training [17]–[22]. 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 [23]. They implement a new tensor compression engine located in memory controllers of GPUs and show that swapping compressed tensors reduces DNN training time by 32%.

Compressing tensors using additional (de)compression units seems a straightforward approach because no changes are required for DNN applications. However, the static compression scheme has two major issues. (1) *It did not consider varying tensor sparsity and sizes*. For example, our study shows the tensor sparsity of the VGG16 DNN model varies between 20% and 80%. The existing schemes only achieve suboptimal performance when applying the same compression algorithms and parameters to the tensors whose sparsity dynamically changes during training. (2) *The solution requires hardware changes, thus having no portability to existing GPUs*. Existing GPUs cannot benefit from tensor compression because they do not have dedicated compression units in their memory controllers. A practical solution should be minimally dependent on additional hardware features.

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 compression algorithms

[*] Shuibing He is the corresponding author.

2168-9253/21/$31.00 ©2021 IEEE
DOI 10.1109/Cluster48925.2021.00019

(i.e., zero-value compression (ZVC) [23], run-length encoding (RLE) [24], compressed sparse row (CSR) [25], and LZ4 [26]). CSWAP caters for 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, we observe that the cost-effectiveness of tensor compression is very sensitive to the sparsity, sizes of tensors, settings of GPU parameters including the size of GPU grids and blocks, and data compression algorithms. Therefore, we build a time model which uses a machine learning approach (i.e., linear regression) to accurately predict the (de)compression time at runtime. Our results show that the model has a relative absolute error of 3% on average. Furthermore, instead of searching the optimal GPU configuration manually through trials and errors, we use the Bayesian optimization algorithm to set GPU parameters. The algorithm can reduce the time of search optimal setting by 98% compared to the approaches using trials and errors.

CSWAP makes the following contributions:

- We design a selective software-level tensor compression framework to reduce the tensor swapping cost without relying on compression units in the memory controllers of GPUs. CSWAP applies tensor compression adaptively in the ReLU and MAX layers according to the cost-effectiveness of tensor compression at runtime.
- Our study shows the performance of tensor compression is sensitive to the GPU settings, tensor size and sparsity, and the characteristics of compression algorithms. Therefore, we design two machine-learning algorithms to predict the tensor (de)compression time and facilitate the search for optimal GPU settings.
- We implement a software prototype of CSWAP using Torch [27] and apply it to six popular DNN models (e.g., AlexNet [1], VGG16 [4], ResNet [6], etc.). Our experimental results show that it reduces the swapping latency and training time by up to 50.9% and 34.6%, respectively, compared to vDNN [20] while achieving comparable performance to hardware solutions, e.g., cDMA [23].

II. Background and Motivation

A. DNN 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.

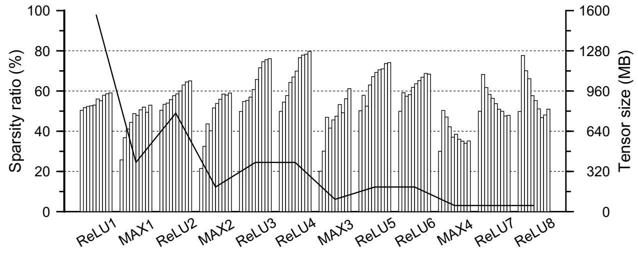


Fig. 1. Changing sparsity of tensors during the training of VGG16 in the first 50 epochs (left axis), while the broken line (right axis) denotes the changing size of tensors of VGG16 .

Most DNNs are designed using a combination of convolution layers (CONV), activation layers (ACTV), pooling layers (POOL), and fully connected layers (FC). The convolution layers extract meaningful features in input data. The activation layers apply an activation function (typically ReLU [1] because of its simplicity and efficiency) to the input feature maps. ReLU allows positive input values to pass through but resets all negative input values to zero. The pooling layers are designed to reduce the spatial size of the input data using average or maximum operations (MAX) over feature maps. The fully connected layers find the hidden correlation between the extracted features and the classified categories.

B. Changing Sparsity and Size of DNN Tensors

Tensor sparsity is observed in many popular DNN models, e.g., VGG16 and AlexNet. One major cause of tensor sparsity is the nature of ReLU operations, which makes the output tensors of ACTV 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 [28], NVIDIA Tesla V100, and the Torch framework [29]. 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 V).

Figure 1 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 (broken line in Figure 1). 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

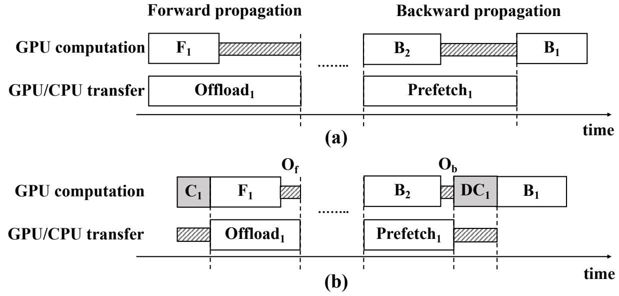


Fig. 2. (a) DNN execution flow with tensor swapping but without tensor compression; (b) The execution flow with both tensor compression and swapping in cDMA.

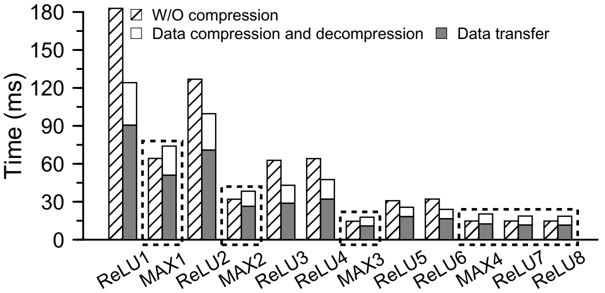


Fig. 3. Swapping time of VGG16 with static compression compared to that without compression. The swapping time using the static compression consists of data transfer time (the lower part of the right bar) and data compression and decompression time (higher part of the right bar).

change across epochs for the same tensor. We also evaluate the tensor sparsity and tensor size with other models and datasets (Section V). The results show similar observations. CSWAP opportunistically applies tensor compression considering the changing tensor size and sparsity.

C. Ineffectiveness of Static Compression Schemes

vDNN uses virtual memory to support training a DNN whose memory demand might be larger than the size of GPU memory [30]. 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. Because of the tensor sparsity in the ACTV and POOL layers (discussed in Section II-B), cDMA further reduces the tensor swapping latency by compressing all the tensors exploiting their sparsity in GPUs [23]. Figure 2 illustrates the execution flow of memory swapping with tensor compression in cDMA. F_n/B_n denotes the time of forward/backward computation at the 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. 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.

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}. Recently, researchers show that tensor swapping latency can no longer be overlapped with DNN forward/backward computation [23]. This is because data transfer bandwidth offered by the powerful PCIe link (gen3) has remained unchanged at 16 GB/sec while the performance of datacenter GPUs is almost tripled since 2014 [23]. And we think the performance gap between I/O bus and GPU computing to be continued in the future despite the emerging PCIe gen4 and NVLink techniques [31]. To reduce swapping overhead, cDMA compresses tensors before offloading and decompresses them after prefetching. It introduces compression latency C_n and decompression latency DC_n. For cDMA, the compression operations are executed by dedicated (de)compression units in memory controllers of GPUs. 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 its corresponding computation time.

We then study the effectiveness of tensor compression in GPU virtual memory. Instead of relying on the (de)compression units which are not available in markets, we implement a new *static compression* (SC) 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.

Figure 3 shows the execution time per layer during the training of VGG16 without 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]. The results show that there are three reasons for SC's ineffectiveness. (1) Compression time accounts for around 30% of swapping latency. (2) As the sparsity and size of tensors are varied, blindly applying compression to all the tensors does not reduce the overall swapping latency when the tensor size is small and its sparsity is low. (3) The compression performance is affected by many GPU parameters (e.g., block size and grid size) which are difficult to be determined statically before launching the kernels.

In summary, while the tensor compression has been well implemented in GPUs to reduce tensor swapping latency, it may not achieve optimal performance if being applied to all the ReLU and MAX layers blindly. 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.

TABLE I
COMPARISON OF CSWAP WITH EXISTING TENSOR SWAPPING FRAMEWORKS FOR GPU-BASED DEEP-LEARNING SYSTEMS.

Technique	*Compression unit/location*	*Tensor selection*	*Portability*
vDNN [20]	N/A	N/A	Yes
Other swapping [17], [19], [21], [32], [33].	N/A	N/A	Yes
cDMA [23]	Memory Controller	No	No
vDNN++ [30]	CPU	No	Yes
CSWAP	GPU	Yes	Yes

III. RELATED WORK

Model compression. DNN training streams need to manage feature maps and model weights. To reduce the size of feature maps, Courbariaux et al. proposed to train a model using low precision multiplications [34]. It compresses training data in floating point, fixed point, and dynamic fixed point formats and show that low-precision feature maps are sufficient for training Maxout networks. Gist exploits existing value redundancy and proposes both lossless and lossy encoding schemes to reduce the memory footprint of the feature maps [35]. These approaches to compress feature maps are complimentary to CSWAP and can be used for tensor compression in swapping.

Because DNN model weights are over-parameterized [34], [36], many approaches of weight quantization and pruning have been proposed [37]–[42]. 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 I. vDNN studies the characteristics of different DNN layers and chooses to swap convolution input tensors to reduce memory footprint in GPUs [20]. moDNN [32], SuperNeurons [19], SwapAdvisor [17], and Sentinel [43] introduce different heuristics and profiling technology to swap data between heterogeneous memories. Besides, Capuchin [33] uses greedy policy and AutoTM [21] 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 [23] was the first swapping framework that compresses tensors using compression hardware in memory controllers of GPUs. However, it has two major issues. (1) It is not adaptable to the existing GPUs without the compressors. And (2) it does not consider the cost-effectiveness of tensor compression trading off tensor size, sparsity, compression algorithm, and swapping latency. vDNN++ supports tensor compression using host CPUs to reduce the pinned memory requirement in the host [30]. Nevertheless, it does not address the tensor transfer bottleneck caused by the limited data transfer bandwidth of PCIe links. CSWAP 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.

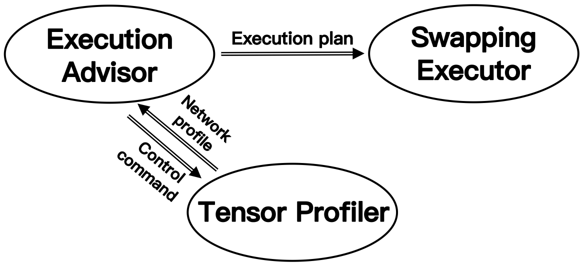


Fig. 4. Architecture overview of CSWAP. The *execution plan* includes compression decision and GPU settings for (de)compression operations. The *network profile* consists of tensor sparsity, size, and execution time of layers. The *control command* manages tensor profiles.

IV. DESIGN OF CSWAP

The design objective of CSWAP is to opportunistically apply tensor compression for swapping in the training of DNNs when its memory demand is larger than GPU memory capacity. In this section, we describe CSWAP architecture as well as how it determines the cost-effectiveness of compression operations and manages GPU settings for compression streams. To make our framework portable and be easily adopted with different GPU architectures, we implement all components of CSWAP in the existing library of machine learning.

A. Overview of Software Architecture

CSWAP consists of three components including *tensor profiler*, *execution advisor*, and *swapping executor* as shown in Figure 4. The *tensor profiler* is executed when a new DNN training task is submitted the first time. DNN training process usually consists of multiple iterations. During the first iteration, it collects the DNN characteristics including tensor size and sparsity, the execution time of each DNN layer without compression, and effective data transfer bandwidth of PCIe links. We did not use the name-tag bandwidth number because its effective bandwidth is affected by other factors (e.g., memory configurations of CPUs and GPUs). The DNN profile is unique and does not change given the same GPU and system configurations except the tensor sparsity. We only need to execute the *tensor profiler* to collect the sparsity once in each epoch to reduce profiling overhead. Then the profiling data is stored in an in-memory database for retrieval with low latency. Another functionality of the *tensor profiler* is to manage the GPU parameter settings given a DNN (discussed in Section IV-D) for the compression GPU stream.

The *execution advisor* is executed to fetch DNN profiles to decide on whether to compress a tensor for swapping. If a tensor is to be compressed, it also needs to retrieve the GPU compression setting given the tensor characteristics to achieve optimal compression performance. Finally, the *swapping executor* selects proper tensors and exploits multiple GPU threads to execute compression in parallel before swapping from GPUs to CPUs and execute decompression after swapping back from CPUs to GPUs.

TABLE II
PARAMETERS IN THE MODEL. ONE TIME OR EPOCH MEANS WE PROFILE ONLY ONCE AT THE FIRST ITERATION OR EVERY EPOCH.

Symbol	Meaning	Profiling
$Size^t$	size of tensor t	one time
BW_{h2d}	effective PCIe bandwidth from CPU to GPU	one time
BW_{d2h}	effective PCIe bandwidth from GPU to CPU	one time
$Hidden_f^t$	overlapped swapping latency in forward propagation of tensor t	one time
$Hidden_b^t$	overlapped swapping latency in backward propagation of tensor t	one time
$Sparsity^t$	sparsity of tensor t	epoch
$Time_c^t$	compression time of tensor t	offline
$Time_{dc}^t$	decompression time of tensor t	offline

B. Determining Cost-Effectiveness of Tensor Compression

With the changing tensor sparsity 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 II. Given a tensor t with size of $Size^t$ and sparsity of $Sparsity^t$, 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 Figure 2(a), 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 following equation 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.

$$T' = \max(\frac{Size^t}{BW_{d2h}} - Hidden_f^t, 0) + \max(\frac{Size^t}{BW_{h2d}} - Hidden_b^t, 0) \tag{1}$$

Equation 2 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 IV-C. O_f and O_b are the portion of the data transfer time that cannot be effectively hidden from the DNN propagation time. If the compressed tensor is adequately small, $Time_c^t$ and $Time_{dc}^t$ will dominate in T.

$$T = Time_c^t + Time_{dc}^t + O_f + O_b \tag{2}$$

$$O_f = \max(\frac{Size^t \times (1 - Sparsity^t)}{BW_{d2h}} - Hidden_f^t, 0) \tag{3}$$

$$O_b = \max(\frac{Size^t \times (1 - Sparsity^t)}{BW_{h2d}} - Hidden_b^t, 0) \tag{4}$$

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 and tensor size $Size^t$. Then it detects the tensor sparsity and records the hidden latency ($Hidden^t$). Based on these data, the *execution advisor* makes a preliminary decision for all sparse tensors. During the training, tensors may become more sparse or denser. The *execution advisor* then asks the *tensor profiler* for the up-to-date data (e.g., $Time_c^t$ and $Time_{dc}^t$) used by the swapping cost model. T' and T are then re-computed for updating tensor compression decisions.

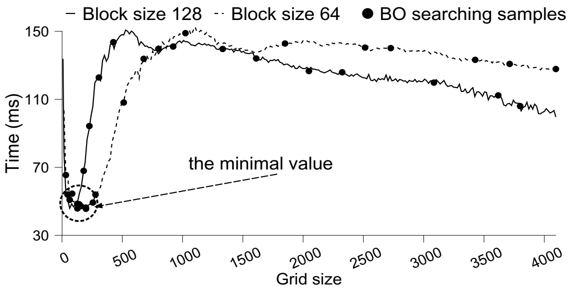


Fig. 5. The sum of ZVC compression and decompression time with different GPU grid and block settings.

C. 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 ratio, and compression algorithms. We experimentally observe that the tensor size and sparsity have a linear relationship with $Time_c^t$ and $Time_{dc}^t$. Therefore, CSWAP models the relationship offline using linear regression algorithms [44]. 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 size and sparsity.

Specifically, we use the following steps to build and deploy a (de)compression time model. We first collect data samples for training the model. Each training sample includes the following measures: tensor size, tensor sparsity, compression algorithm, $Time_c^t$ and $Time_{dc}^t$. In the experiments, we found that randomly sampling the tensor size and sparsity will likely over-fitting 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 Figure 1. This sparsity range is denoted as R (e.g., 60%). Second, to improve the model accuracy, CSWAP trains n sub-models. Sub-model i is trained using samples whose sparsity is in $[Sparsity_{base} + R * i/n, Sparsity_{base} + R * (i+1)/n)$, where $0 \leq i <$n. The sub-models are combined to form a holistic model after training and deployed for inference. In training, we vary the tensor size from 20 MB to 2000 MB in addition to the changes of tensor sparsity. Third, the (de)compression time model is stored in the in-memory database for retrieval.

D. Setting GPU Parameters for Compression Kernels

The execution time of compression kernels is mainly determined by the GPU grid and block sizes. A wrongly tuned compression kernel may not saturate GPU warps failing to exploit its maximum parallelism or may reduce GPU utilization. Figure 5 shows the variation of the sum of compression and decompression time as the GPU grid size is increased from 1 to 4096 and the block size is increased from 64 to 128. In the experiments, we compressed and decompressed a synthetic tensor (500 MB and 50% sparsity). When the block size is 64, we can observe that the total time is decreased from 146 ms to 44 ms when the grid size is increased from 10 to 197 and then it is increased to 150 ms when the grid size is further increased to 1024. The time with 128 block size shows a similar trend.

Because both $Time_c^t$ and $Time_{dc}^t$ change significantly with GPU settings, it is very hard, if not impossible, for regular users to find an optimal GPU setting for the compression kernels. Conventional compression schemes require domain experts to explore a large design space trading off among grid size, block size, and compression time, resulting in suboptimal and time-consuming tuning processes. To find an effective GPU grid/block set, search algorithms can be used to accelerate the process. However, since the goal function of choosing GPU grid and block sizes for (de)compression is almost non-convex and expensive to calculate, many algorithms (e.g., random search [45] and grid search [46]) may only find a suboptimal sample or require a long searching time. To solve this issue, we choose Bayesian optimization (BO) algorithm [47] and customize it for our task. It is efficient for searching GPU grid/block size and can automatically tune the performance of compression algorithms.

BO is a global search method for solving multi-parameter modeling problems that are hard to find a global optimal solution. The inherent idea of BO derives from the famous "Bayes theorem", which can search efficiently with guidance [47], [48]. BO is instructed by posterior distribution and acquisition function. The posterior distribution determines the estimated values (i.e., sum of $Time_c^t$ and $Time_{dc}^t$) and prediction uncertainty of points (i.e., tuples of grid size and block size) in the entire search space. The acquisition function determines the next point to search. It is designed to avoid getting trapped in local optima (exploration) and to refine the search in the vicinity of a promising solution (exploitation).

Algorithm 1 shows the pseudocode of the BO search engine used in CSwap. We first generate s_1 points (i.e., tuples of (grid size, block size)) in the search space and obtain the corresponding values to initailize $\boldsymbol{D}$, the dataset of candidate points (Line #3-9). BO then uses $\boldsymbol{D}$ to estimate the posterior distribution in the entire search space (Line #10). The acquisition function instructs BO to pick out the next search point. Next, under the guidance of posterior distribution and acquisition function, BO searches s_2 points (Line #11-16), e.g., the dotted circle in Figure 5. The search results update the posterior distribution and acquisition function in return. Finally, BO returns the optimal point from $\boldsymbol{D}$ (Line #17).

Algorithm 1 BO search algorithm for choosing GPU parameters for (de)compression kernels

```
Require: $s_1$: the number of initial samples; $s_2$: the times of attempts to find the optimal solution;
 1: $bayes\_opt \leftarrow$ new $bayes\_opt()$   ▷ Create the BO engine
 2: $\boldsymbol{D} \leftarrow \varnothing$   ▷ Dataset of previously observed samples
 3: for $i$ = 1, 2, ..., $s_1$ do
 4:     $g \leftarrow random(0..4096)$   ▷ $g$ denotes grid size
 5:     $b \leftarrow random(64, 128)$   ▷ Set block size as 64 or 128
 6:     $p \leftarrow (g, b)$
 7:     $y \leftarrow bayes\_opt.exec(p)$   ▷ obtain sum of $Time_c^t$ and $Time_{dc}^t$
 8:     $\boldsymbol{D}.append(p, y)$   ▷ Add the new sample to $\boldsymbol{D}$
 9: end for
10: $bayes\_opt.update(\boldsymbol{D})$   ▷ estimate posterior distribution and acquisition function
11: for $i$ = 1, 2, ..., $s_2$ do
12:     $p \leftarrow bayes\_opt.select()$   ▷ select the next sample
13:     $y \leftarrow bayes\_opt.exec(p)$
14:     $\boldsymbol{D}.append(p, y)$
15:     $bayes\_opt.update(\boldsymbol{D})$
16: end for
17: return $bayes\_opt.optimize(\boldsymbol{D})$ ▷ return an optimal point
```

In the experiments, we set s_1 and s_2 as 10 and 25 respectively and limit the maximum grid size to 4096. We find that the settings provide a large enough search space to find an optimal solution with minimum time cost (i.e., less than 1 minute compared to hours when full searches are implemented). We configure the block size as 64 or 128 because each SM of most existing GPUs features two or four warp schedulers [49], [50], which support 64 or 128 threads to execute concurrently. The BO search engine is executed before DNN training begins using CSwap.

E. Tensor Compression Algorithms

To compress sparse floating-point tensors, we need an efficient compression algorithm that is easy to be implemented on GPUs. CSwap executes tensor (de)compression using GPU multi-computing units in parallel to further reduce the (de)compression time. Among floating-point compressors, it currently supports run-length encoding (RLE) [24], compressed sparse row (CSR) [25], LZ4 [26] , and zero-value compression (ZVC) [23].

RLE stores data sequences where the same value occurs in many consecutive positions as a single value and count 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 nonzero 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 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 less 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. Currently, we implemented these four compressors for GPUs and study their performance. We wish to support more compression algorithms in the future work. 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 achieves lower latency when PCIe bandwidth becomes a performance bottleneck.

V. 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) [51]. In both servers, we run Ubuntu-18.04, CUDA 10.0.13 [52], CuDNN 7 [53], and Torch 1.5.1 [29].

Workloads and datasets. To show the effectiveness of our approach for extensive workloads, we evaluate CSWAP with four *linear* DNN models (i.e., AlexNet [1], Plain20 [54], VGG16 [4], and MobileNet [55]), and two *non-linear* models (i.e., ResNet [6] and SqueezeNet [56]). 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], [4], [6], [54]–[56] and configured training batch sizes are shown in Table III.

TABLE III
WORKLOAD AND DATASET CONFIGURATIONS. IN THE TABLE, WE USE X/Y TO DENOTE THE BATCH SIZES FOR CIFAR10 (X) AND IMAGENET (Y), RESPECTIVELY, GIVEN A PARTICULAR GPU.

DNN Model	Batch size (V100)	Batch size (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

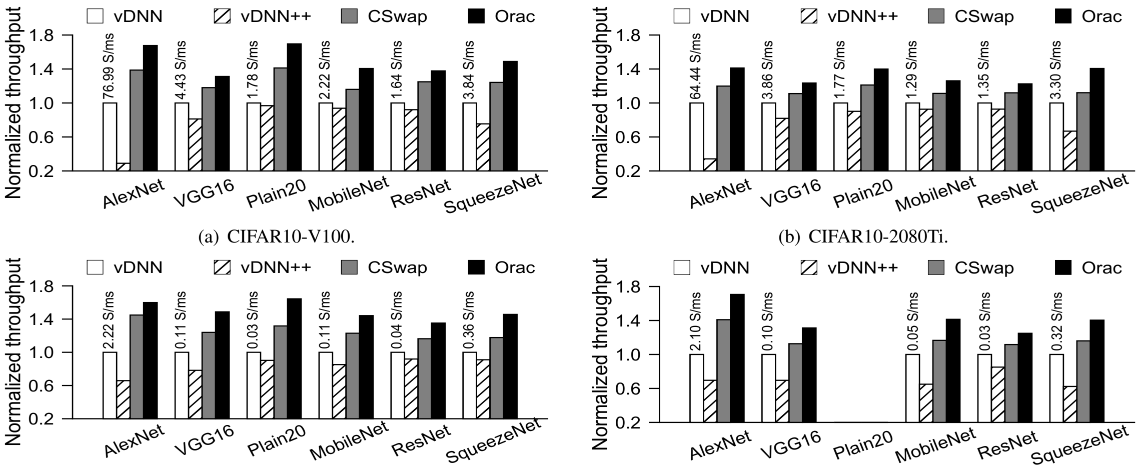
We use two representative datasets including CIFAR10 [57] and ImageNet [58]. The CIFAR10 dataset is a collection of 60, 000 (50, 000 for training and 10, 000 for testing) labeled color images (32 × 32 pixels each). The second is ImageNet dataset which has 1.4 million 224 × 224 pixel images across 21841 non-empty synsets.

DNN frameworks for comparison. We compare CSWAP with the state-of-the-art GPU memory swap frameworks, vDNN [20], vDNN++ [30], and cDMA [23]. 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 compresses 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 [23] relies on the (de)compression units which are not available in markets, we implement a static compression (SC) scheme which replicates the zero-value compression algorithm used in cDMA. In SC, we use GPUs to emulate the (de)compression units in memory controllers. In the evaluation, we compare CSWAP to SC instead of cDMA. Furthermore, to show the potential of our proposed framework, we evaluate the performance of CSWAP on an oracular system (Orac), where the GPU is fast enough so that the compression and decompression time is effectively zero.

A. General Results

Figure 6 shows the system throughputs of different frameworks on different GPUs and datasets. For comparison, the throughput (samples/ms) is normalized to that of vDNN.

Figure 6(a) shows the system throughput when training the models on V100 with the CIFAR10 dataset. Overall, CSWAP outperforms vDNN and vDNN++ by 25% and 190% on average. We also have the following observations. First, compared to vDNN, CSWAP reduces the model training time by up to 29.2% across the six 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 sparsity, leading to high data transfer cost in



(a) CIFAR10-V100.

(b) CIFAR10-2080Ti.

(c) ImageNet-V100.

(d) ImageNet-2080Ti.

Fig. 6. The performance comparison of four different GPU virtual memory management frameworks. We conduct the experiments with six 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.

swapping. Compared to vDNN++, 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. Second, we observed that CSwap performs the same number of compression and decompression operations as that of Orac. Its training time is up to 20.8% longer than Orac because of the compression and decompression operations. Third, Figure 6(a) also shows that the performance benefit of using CSwap varies for different DNN models. For example, the training time of AlexNet and Plain20 is reduced by up to 31% and 24.1% respectively. For other models, CSwap only reduces the training time by up to 19.3%. The main reason is that the data swapping time dominates in the training time for AlexNet and Plain20. Specifically, we observed that the data transfer time accounts for 71% and 73% of the total model training time for AlexNet and Plain20 respectively. For other models, the data transfer time accounts for less than 50% of the training time. As a result, reducing the data transfer time using CSwap greatly improves the overall training performance of AlexNet and Plain20.

Figure 6(b) 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 models compared to the results on 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 [59]. 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 compute capability 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. Figure 6(c) and Figure 6(d) 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 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 ImageNet dataset. Note that in Figure 6(d), 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.

B. *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. While CSwap performs (de)compression in tensor swapping based on the cost-effectiveness analysis of tensor compression, SC blindly compresses the sparse tensors in all layers of DNNs. We train the models on V100 and 2080Ti with CIFAR10 and ImageNet.

We show the experimental results in Figure 7. We can observe that CSwap can improve the performance by 5.5% and 5.1% on average compared to SC for all the models except Plain20 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

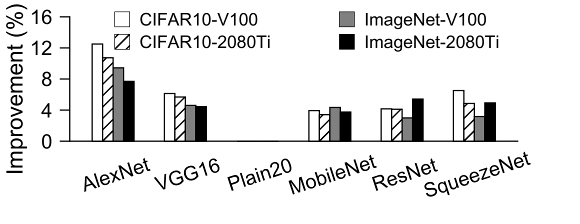
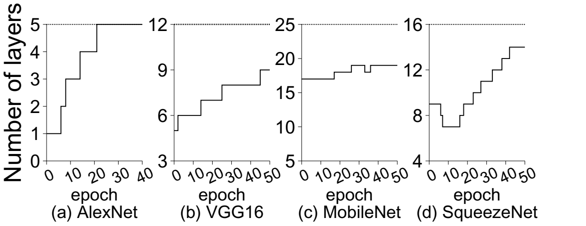


Fig. 7. Performance improvement of CSWAP over the static compression (SC) scheme.



(a) AlexNet (b) VGG16 (c) MobileNet (d) SqueezeNet

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

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.

Figure 8 shows the number of layers whose tensors are compressed with CSWAP during the training of AlexNet, VGG16, MobileNet, and SqueezeNet. We have two observations. First, the number of such layers tends to increase while epochs are increased. Let's use VGG16 as an example, as shown in Figure 9, 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, there may exist some layers never to 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 Figure 9. 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 Figure 8(c). For SqueezeNet, there exist two tensors whose sparsity is decreased between epoch 5 and epoch 17 and is increased after epoch 17, as shown in Figure 8(d). The reason for the fluctuation is that their model convergence may change during training. 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 Figure 8.

C. (De)compression Time Model Verification

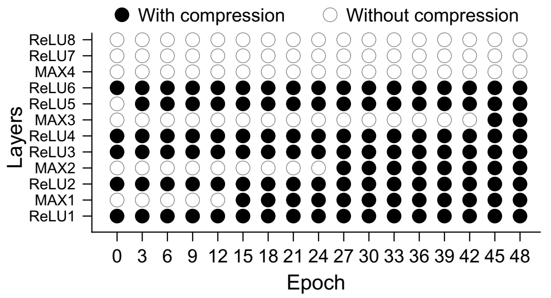


Fig. 9. VGG16 layer-wise compression detail. The x-axis denotes the training epoch and y-axis represents all the layers in the model. The black dot denotes that the tensor of this layer needs to be compressed while the white dot means that the tensor is only transferred without compression.

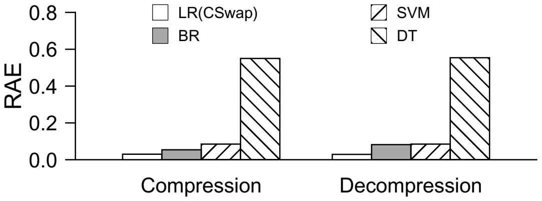


Fig. 10. The accuracy of (de)compression time predication using LR, BR, SVM, and DT models.

tion advisor (Section IV). We compare the linear regression (LR) model in CSWAP for (de)compression time modeling to three other regression models including bayesian regression (BR) [60], support vector machine (SVM) [61], and decision tree (DT) [62] from scikit-learn [63]. 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 current supports four compression algorithms including RLE, CSR, LZ4, and ZVC. For each algorithm, we generate a total of 3000 sparse tensors, whose sparsity ranges between 20% to 90%. 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 Figure 10, LR achieves the best prediction accuracy. Its mean relative absolute error for compression and decompression time prediction is only 3%, which is 56% and 80% smaller than those of BR and SVM respectively.

We also evaluate the compression decision accuracy based on the swapping cost model, 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 the decision accuracy as the ratio of the number of correct decisions to the number of all decisions.

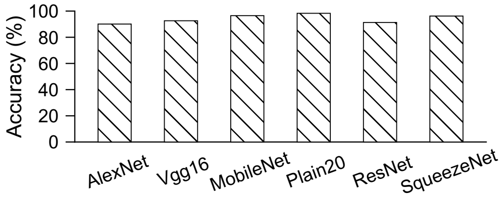


Fig. 11. The compression decision accuracy based on the LR model.

D. Effectiveness of Bayesian Optimization for Setting GPU Parameters

To demonstrate the effectiveness of the Bayesian optimization, we train the VGG16 model with four GPU grid/block setting approaches, i.e., *random search*, *expert knowledge*, *CSwap BO search*, and *grid search*. For the random search, we randomly choose a GPU setting for tensor (de)compression. For the expert knowledge, we manually choose a GPU setting which may achieve the minimum (de)compression time given a particular set of grid and block size. More specifically, we configure the block size as 128 because this configuration will make all threads concurrently execute since each SM (Streaming Multiprocessor) has four warp schedulers and each warp has 32 threads in our GPUs [64]. The grid search finds the best GPU setting by going through all grid/block configurations.

Figure 12 illustrates the average training time for one VGG16 iteration under the four design space exploring methods. The overall training time consists of two parts: the sum of compression and decompression time and the remaining time including DNN computing time, data swapping time, etc. Figure 12 shows that the (de)compression time varies greatly under different searching approaches. The random search (i.e., 2024, 64) has the maximum (de)compression time of 547 ms, which is 54% of the total training time, thus unacceptable for DNN training. The expert knowledge approach needs 178 ms to compress and decompress, which is 32% of that with the random search. CSwap BO search uses the algorithm in Section IV-D to find the optimized grid and block configuration (199, 64). Its compression and decompression time is 66 ms which is only 30% of the time cost with the expert knowledge approach and 12% of the time cost with the random search. The grid search achieves an optimal time cost of 56 ms with a global brute-force search. The CSwap BO search achieves similar performance as the grid search. It reduces the search overhead by 224 × compared to the grid search.

E. 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

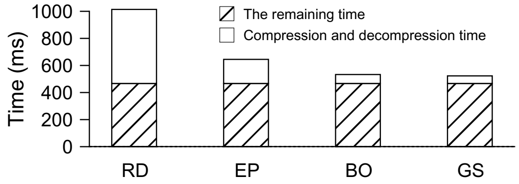


Fig. 12. The average training time of VGG16 for one iteration. RD: random search, EP: expert knowledge, BO: CSwap BO search, and GS: grid search.

do not change across epochs, CSwap only needs to update $Sparsity^t$ and $Time_c^t$/$Time_{dc}^t$ periodically to make dynamic decisions. 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 train a (de)compression time model of tensor compression offline as discussed in Section IV-C. It only takes on average 4.5 minutes to generate all training samples and 21ms to build the time model because of the lower complexity of the linear regression method used in the paper. In addition, CSwap executes Bayesian optimization search before DNN training begins (Section IV-D). This one-time overhead is 50 seconds compared to 3 hours when full grid searches are used.

VI. Conclusion

The virtual memory manager of GPUs needs to swap tensors between GPUs and CPUs to increase the effective size of GPU memory for training large DNN models. Existing tensor compression schemes blindly apply the same compression algorithm to all tensors without considering their sparsity and sizes, resulting in suboptimal tensor swapping performance. 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 uses the cost model of tensor swapping to selectively apply (de)compression to tensors in the ReLU and MAX layers according to its cost-effectiveness of tensor compression at runtime considering tensor sparsity and sizes. Third, it uses a machine-learning algorithm to facilitate the search for optimal GPU settings with low latency and high accuracy. We implement CSwap using Torch and evaluate it using six linear and non-linear DNNs. The experimental results show that it reduces the tensor swapping time by 50.9% and 47.6% on V100 and 2080Ti respectively. It reduces the total training time by up to 34.6%.

VII. Acknowledgment

This work was supported in part by the Key Scientific Technological Innovation Research Project by Ministry of Education, the Zhejiang Lab Research Project No. 2020KC0AC01,

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification With Deep Convolutional Neural Networks,” *Communications of the ACM*, pp. 1097–1105, 2017.

[2] S. Zhang, L. Yao, A. Sun, and Y. Tay, “Deep Learning Based Recommender System: A Survey and New Perspectives,” *ACM Computing Surveys*, pp. 1–38, 2019.

[3] H. Ze, A. Senior, and M. Schuster, “Statistical Parametric Speech Synthesis using Deep Neural Networks,” in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 7962–7966.

[4] Karen Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” *arxiv: 1409.1556*, 2014.

[5] B. Zoph, G. Brain, V. Vasudevan, J. Shlens, and Q. V. Le Google Brain, “Learning Transferable Architectures for Scalable Image Recognition,” in *Proceedings of the Conference on Computer Vision and Pattern Recognition*, 2018, pp. 8697–8710.

[6] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[7] J. Devlin, M.-W. Chang, K. Lee, K. T. Google, and A. I. Language, “BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding,” *arXiv: 1810.04805*, 2018.

[8] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, “Inception-v4: Inception-ResNet and the Impact of Residual Connections on Learning,” *arXiv: 1602.07261*, 2016.

[9] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going Deeper with Convolutions,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9.

[10] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, “Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer,” *arXiv: 1701.06538*, 2017.

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, 2016.

[12] TensorFlow, “Introduction to Tensors,” 2020. [Online]. Available: https://www.tensorflow.org/guide/tensor

[13] T. Yu, P. Petoumenos, V. Janjic, H. Leather, and J. Thomson, “COLAB: A Collaborative Multi-Factor Scheduler for Asymmetric Multicore Processors,” in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, p. 268–279.

[14] T. Yu, R. Zhong, V. Janjic, P. Petoumenos, J. Zhai, H. Leather, and J. Thomson, “Collaborative Heterogeneity-Aware OS Scheduler for Asymmetric Multicore Processors,” *IEEE Transactions on Parallel and Distributed Systems*, pp. 1224–1237, 2021.

[15] NVIDIA, “NVIDIA V100 TENSOR CORE GPU,” 2020. [Online]. Available: https://www.nvidia.com/en-us/data-center/v100/

[16] Z. Zhang, C. Chang, H. Lin, Y. Wang, R. Arora, and X. Jin, “Is network the bottleneck of distributed training?” in *Proceedings of the Workshop on Network Meets AI & ML*, 2020, p. 8–13.

[17] C.-C. Huang, G. Jin, and J. Li, “SwapAdvisor: Push Deep Learning Beyond the GPU Memory Limit via Smart Swapping,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1341–1355.

[18] J. Ren, J. Luo, K. Wu, M. Zhang, and D. Li, “Sentinel: Runtime Data Management on Heterogeneous Main Memory Systems for Deep Learning,” *arXiv:1909.05182*, 2019.

[19] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, “SuperNeurons: Dynamic GPU Memory Management for Training Deep Neural Networks,” in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018, pp. 41–53.

[20] 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 *Proceedings of the Annual International Symposium on Microarchitecture*, 2016, pp. 1–13.

[21] 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 *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 875–890.

[22] T. D. Le, Y. Negishi, H. Imai, and K. Kawachiya, “TFLMS: Large Model Support in TensorFlow by Graph Rewriting,” *arXiv:1807.02037*, 2018.

[23] 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 *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2018, pp. 78–91.

[24] A. H. Robinson and C. Cherry, “Results of a Prototype Television Bandwidth Compression Scheme,” in *Proceedings of the IEEE*, 1967, pp. 356–364.

[25] A. Alabaichi, A. H. Alhusiny, and E. Mohammed Thabit, “A Novel Compressing a Sparse Matrix using Folding Technique,” *Research Journal of Applied Sciences, Engineering and Technology*, pp. 310–319, 2017.

[26] NVIDIA, “NVIDIA/nvcomp: A Library for Fast Lossless Compression/Decompression on the GPU,” 2020. [Online]. Available: https://github.com/NVIDIA/nvcomp

[27] Pytorch, “Pytorch/pytorch: Tensors and Dynamic Neural Networks in Python With Strong GPU Acceleration,” 2020. [Online]. Available: https://github.com/pytorch/pytorch

[28] J. Deng, W. Dong, R. Socher, L.-J. Li, Kai Li, and Li Fei-Fei, “ImageNet: A Large-Scale Hierarchical Image Database,” in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2010, pp. 248–255.

[29] PyTorch, “Pytorch/vision,” 2020. [Online]. Available: https://github.com/pytorch/vision/tree/master/torchvision

[30] S. B. Shriram, A. Garg, and P. Kulkarni, “Dynamic memory management for GPU-based training of deep neural networks,” in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2019, pp. 200–209.

[31] NVIDIA, “NVLink:Advanced Multi-GPU Systems — NVIDIA,” 2020. [Online]. Available: https://www.nvidia.com/en-us/data-center/nvlink/

[32] X. Chen, D. Z. Chen, and X. S. Hu, “MoDNN: Memory Optimal DNN Training on GPUs,” in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2018, pp. 13–18.

[33] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian, “Capuchin: Tensor-Based GPU Memory Management for Deep Learning,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 891–905.

[34] M. Courbariaux, Y. Bengio, and J.-P. David, “Training Deep Neural Networks with Low Precision Multiplications,” *arXiv:1412.7024*, 2014.

[35] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, “GIST: Efficient Data Encoding for Deep Neural Network Training,” in *Proceedings of the International Symposium on Computer Architecture*, 2018, pp. 776–789.

[36] 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 *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 2857–2865.

[37] V. Vanhoucke, A. Senior, and M. Z. Mao, “Improving the Speed of Neural Networks on CPUs,” in *Proceedings of the Deep Learning and Unsupervised Feature Learning Workshop*, 2011, pp. 1–6.

[38] S. Anwar, K. Hwang, and W. Sung, “Fixed Point Optimization of Deep Convolutional Neural Networks for Object Recognition,” in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, 2015, pp. 1131–1135.

[39] K. Hwang and W. Sung, “Fixed-Point Feedforward Deep Neural Network Design using Weights +1, 0, and -1,” in *Proceedings of the IEEE Workshop on Signal Processing Systems*, 2015, pp. 1–6.

[40] Y. Gong, L. Liu, M. Yang, and L. Bourdev, “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding,” in *Proceedings of the International Conference on Learning Representations*, 2016, pp. 91–104.

[41] C. Ding, S. Wang, N. Liu, K. Xu, Y. Wang, and Y. Liang, “Req-yolo: A resource-aware, efficient quantization framework for object detection on fpgas,” in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, p. 33–42.

[42] 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 *Proceedings of the ACM International Conference on Supercomputing*, 2021, p. 304–315.

[43] J. Ren, J. Luo, K. Wu, M. Zhang, and D. Li, “Sentinel: Runtime Data Management on Heterogeneous Main MemorySystems for Deep Learning,” *arXiv:1909.05182*, 2019.

[44] D. C. Montgomery, E. A. Peck, and G. G. Vining, *Introduction to Linear Regression Analysis*, 2012.

[45] J. Bergstra and Y. Bengio, “Random Search for Hyper-Parameter Optimization,” *Journal of Machine Learning Research*, 2012.

[46] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, “Algorithms for Hyper-Parameter Optimization,” in *Proceedings of the Advances in Neural Information Processing Systems*, 2011, pp. 2546–2554.

[47] E. Brochu, V. M. Cora, and N. de Freitas, “A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning,” *arXiv: 1012.2599*, 2010.

[48] AlbertAlonso, “Bayesian Optimization,” 2020. [Online]. Available: https://github.com/fmfn/BayesianOptimization

[49] NVIDIA-fermi, “NVIDIA’s Next Generation CUDA TM Compute Architecture: Fermi TM,” 2009. [Online]. Available: https://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA’s_Fermi-The_First_Complete_GPU_Architecture.pdf

[50] NVIDIA, “Tuning CUDA Applications for Turing,” 2019. [Online]. Available: https://docs.nvidia.com/cuda/turing-tuning-guide/index.html

[51] NVIDIA-V100, “Deep Learning GPU Benchmarks - Tesla V100 vs RTX 2080 Ti vs GTX 1080 Ti vs Titan V,” 2018. [Online]. Available: https://lambdalabs.com/blog/best-gpu-tensorflow-2080-ti-vs-v100-vs-titan-v-vs-1080-ti-benchmark/

[52] CUDA, “Cuda toolkit 10.0 archive,” 2018. [Online]. Available: https://developer.nvidia.com/cuda-10.0-download-archive

[53] CUDNN, “cudnn release 7.0,” 2020. [Online]. Available: https://docs.nvidia.com/deeplearning/cudnn/release-notes/rel_7xx.html#rel_765

[54] 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 *Proceedings of the European Conference on Computer Vision*, 2018, pp. 248–255.

[55] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,” *arXiv: 1704.04861*, 2017.

[56] 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,” *arXiv: 1602.07360*, 2016.

[57] A. Krizhevsky, “Learning Multiple Layers of Features from Tiny Images,” Tech. Rep., 2009.

[58] Jia Deng, Wei Dong, R. Socher, Li-Jia Li, Kai Li, and Li Fei-Fei, “ImageNet: A large-scale hierarchical image database,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[59] NVIDIA, “NVIDIA/Cuda-samples: Samples for CUDA Developers Which Demonstrates Features in CUDA Toolkit,” 2020. [Online]. Available: https://github.com/NVIDIA/cuda-samples

[60] A. E. Raftery, D. Madigan, and J. A. Hoeting, “Bayesian model averaging for linear regression models,” *Journal of the American Statistical Association*, pp. 179–191, 1997.

[61] W. S. Noble, “What is a support vector machine?” *Nature Biotechnology*, pp. 1565–1567, 2006.

[62] S. R. Safavian and D. Landgrebe, “A survey of decision tree classifier methodology,” *IEEE Transactions on Systems, Man, and Cybernetics*, pp. 660–674, 1991.

[63] sklearn, “scikit-learn: machine learning in Python — scikit-learn 0.23.2 documentation,” 2020. [Online]. Available: https://scikit-learn.org/stable/

[64] StackOverflow, “How Do I Choose Grid and Block Dimensions for CUDA Kernels?” 2020. [Online]. Available: https://stackoverflow.com/questions/9985912/how-do-i-choose-grid-and-block-dimensions-for-cuda-kernels