FreeLunch: Compression-based GPU Memory Management for Convolutional Neural Networks

Shaurya Patel
University of Massachusetts, Amherst
University of British Columbia
spatel27@student.ubc.ca

Tongping Liu
University of Massachusetts, Amherst
tongpingliu@umass.edu

Hui Guan
University of Massachusetts, Amherst
huiguan@umass.edu

***Abstract*—Recently, there is a trend to develop deeper and wider Convolutional Neural Networks (CNNs) to improve task accuracy. Due to this reason, the GPU memory quickly becomes the performance bottleneck since its capacity cannot keep up with the increase of the memory requirement of CNN models. Existing solutions exploit techniques such as swapping and recomputation to accommodate the shortage of memory. However, they suffer from performance degradations due to either the limited CPU-GPU bandwidth or the significant recomputation cost. This paper proposes a compression-based technique called FreeLunch that actively compresses the intermediate data to reduce the memory footprint of large CNN models. Based on our evaluation, FreeLunch has up to 35% less memory consumption and up to 70% better throughput than swapping and recomputation.**

***Index Terms*—Systems for ML, GPU Memory Management, Memory Compression**

I. Introduction

Recent years have seen increasing adoption of Convolutional Neural Networks (CNN) in AI applications due to their superior accuracy on many computer vision tasks. Developing deeper and wider CNNs is an effective approach to improving task accuracy. However, the development is hindered by the memory constraints of GPUs. Deeper and wider CNN models will consume a large amount of memory during the training. For example, ResNet152 [1] consumes around 18GB of memory for a batch size of only 32 while the mainstream type of GPUs in cloud platforms P100 has only 16GB memory. As GPU memory sizes grow at a slower rate than memory requirements of large CNNs, there is a strong need to design memory management techniques to support the training of large CNN models on a single GPU.

Several memory-management frameworks have been proposed to accommodate larger models on a single GPU [2]–[5]. They introduce two major memory-saving techniques, swapping and recomputation, to reduce the memory footprint of CNN training. Their basic idea is to release the memory for intermediate results (also referred to as *feature maps* interchangeably) in the forward propagation and re-generating them in the backward propagation. More specifically, swapping copies some temporarily-unused feature maps to the CPU memory when the GPU memory is tight, and then brings them back during the backward propagation phase. Recomputation, instead, drops some feature maps and then recomputes them when necessary.

However, these techniques have their drawbacks that lead to performance slowdown. For swapping, the back-and-forth data transferring between CPU and GPU is very expensive, especially given the fact that the memory bandwidth is a well-known performance bottleneck [6]. Prior work even reported that the time of swapping in/out a layer's outputs could be three times higher than a layer's execution [4]. This indicates that swapping itself is not sufficient to release all the memory for feature maps. Although the recomputation technique avoids the bandwidth issue, it introduces significant performance overhead for recomputing previously-released tensors. The recomputation is essentially partial of the forward propagation, which has high resource demands and complex dependencies thus can't be executed concurrently with the backward propagation.

This paper proposes a compression-based memory management approach that can be employed complementarily to alleviate the above-mentioned issues. The proposed approach aims at reducing peak memory consumption during CNN training with minimal performance overhead. Instead of copying intermediate results to the CPU memory or discarding them, they are compressed and stored in the GPU memory and decompressed whenever necessary. Therefore, it does not suffer from the CPU-GPU bandwidth issue of swapping and is less computation-intensive than recomputation. To significantly compress the memory, FreeLunch chooses a lossy algorithm that has a higher compression rate than lossless algorithms. Prior work has shown that lossy compression doesn't degrade model accuracy with acceptable compression rates [7]. FreeLunch employs an existing lossy compression algorithm (called ZFP [8]) that could compress intermediate results (feature maps) to around 20% with no explicit accuracy loss. Although the idea of using compression is not completely novel [7], [9], FreeLunch's contribution lies in its systematic approach in reducing the performance overhead of compression.

First, FreeLunch proposes a *parallel workflow* that performs the compression in parallel with the forward computation, and the decompression in parallel with the backward computation. By performing the compression/decompression outside the critical path, FreeLunch hides these operations in the normal training, which is one major reason why it does not introduce significant performance overhead, as discussed further in

Section III-A.

Second, FreeLunch further reduces the overhead introduced by memory management operations that are known to be very expensive. We explain our mechanisms using compression as an example. The decompression is just the opposite. Naively, one needs to allocate a compression workspace to perform the compression and a compressed space to store the compressed data. After that, one needs to copy the compressed data from the compression workspace to the compressed space, and then free both the original tensor space and the compression workspace. This naive mechanism involves two memory allocations, one copy, and two de-allocations for each feature map.

To reduce these expensive operations, FreeLunch proposes two mechanisms called *Sliding Compression Workspace* and *Persistent Tensor Buffers*. The first mechanism exploits the property of the compression algorithm that it always stores the compressed data in the compression workspace. In particular, FreeLunch pre-allocates a global compression buffer, performs the compression inside, and then shifts the compression workspace right with the size of the compressed data. *Persistent Tensor Buffers* pre-allocates multiple tensor buffers that persist throughout the training phase and re-uses these tensor buffers along with the forward and backward propagation. For example, given a linear CNN model, only three buffers are required to hold the feature maps to be compressed, the input and output of the next layer. After each compression, these tensor buffers will be re-utilized immediately. Overall, FreeLunch reduces $2n$ allocations and deallocations (for n layers) to a constant number of allocations and reduces n memory copies to zero by using these two mechanisms. Details are discussed in Sections III-B and III-C separately.

Based on our evaluation on a range of popular Convolutional Neural Networks (CNNs), FreeLunch reduces up to 35% memory and achieves up to 70% higher throughput compared to swapping and recomputation of Superneurons [3]. This is the reason why it is called as "FreeLunch".

Overall, this paper has the following major contributions:

1) We design and implement a *parallel workflow* that could compress feature maps in parallel with the training process to hide the compression and decompression performance overhead.
2) We propose the combination of *Sliding Compression Workspace* and *Persistent Tensor Buffers* mechanisms that greatly reduce the number of memory management operations to mitigate the performance overhead of compression.
3) We perform the evaluation on a range of widely-used models. These evaluations confirm that FreeLunch could significantly reduce memory consumption (up to 35%) but without explicit performance and accuracy loss.

II. Background and Motivation

A. CNNs and CNN Training

As a major type of Deep Neural Networks (DNNs), Convolutional Neural Networks (CNNs) typically consist of different types of layers such as convolution layers (CONV), pooling layers (POOL), batch normalization layers (BN), and fully-connected layers (FC). Some layers contain parameters that are adjustable through a training process. A CNN training process aims to adjust these parameters to make the CNN meet the desired properties specified by training data. It iterates over three steps: forward propagation, backward propagation, and gradient updates. In the forward propagation, each CNN layer transforms the input feature maps into output feature maps, which becomes the input of the next layer. The backward propagation reversely traverses these layers to calculate gradients of feature maps and gradients of model parameters. In gradient updates, gradients are applied to model parameters that differ depending on the optimizer used during training. In each iteration, a batch of training data is fed into the DNN model. For example, in image classification tasks, a batch could contain 256 images randomly selected from the training data.

A common trend in CNN model development is to design deeper and wider models for capturing the complex patterns in a large amount of training data. It is because larger models usually result in improved task accuracy. For example, on the 1K ImageNet recognition challenge [10], ResNet-18 delivers 89% top-5 accuracy with 18 CONV layers while ResNet152 is able to achieve 94% top-5 accuracy with 152 CONV layers. However, wider and deeper CNN's expose high memory demand that easily exceeds the available memory sizes of the mainstream commercial GPU's on the market. This problem has become one of the major hurdles preventing deep learning practitioners from efficiently exploring complex DNN architectures [11].

B. Existing GPU Memory Management Techniques and Their Limitations

During the CNN training, memory consumption comes mainly from three parts: model parameters, intermediate results, and convolution workspace. Intermediate results further include both feature maps generated in the forward propagation and gradients of feature maps (called gradients maps) in the backward propagation. Model parameters are usually persistent in GPU memory to allow iterative updates. Gradient maps and convolution workspace can be freed immediately after the respective layer computation is finished. However, a feature map can be released only after the corresponding layer finishes execution in backward propagation, as the feature map is used to calculate the gradients of model parameters.

Due to the time gap between the forward and backward propagation, feature maps are the major source of the high memory footprint for CNN training, and therefore the major optimization target of reducing CNN training memory footprint [3], [4], [12]. SuperNeurons [3] proposes a dynamic memory management framework using two memory policies, swapping and recomputation, to release the memory of selected feature maps and enable the training of deeper and wider models on a single GPU. Subsequent works [4], [5] utilized and improved upon these policies. Capuchin [4]

made memory management decisions using the time of tensor accesses tracked at runtime, making their technique applicable to dynamic computation graphs. It combines asynchronous swapping and heuristic-based recomputation and determines the time to apply these policies based on a training phase. SwapAdvisor [5] focuses on improving the swapping-based policy. It pre-generates a schedule for swapping using a genetic algorithm prior to training by jointly optimizing operator scheduling, memory allocation, and swap decisions. However, swapping could suffer from performance degradation because of the limited CPU-GPU bandwidth. The resource-demanding nature of recomputation makes it suitable for only a few lightweight CNN layers. With these policies, one can aggressively reduce the GPU memory consumption but at the cost of a much lower training throughput due to either CPU-GPU communication bottleneck or recomputation cost.

This work introduces data compression as an alternative approach that can reduce the memory footprint of DNN training as swapping and recomputation but achieves higher training throughput than these policies.

III. Design and Implementation

FreeLunch aims at enabling larger models to be trained efficiently on one single GPU. Its contribution lies in its systematic approach of reducing the performance overhead of compression. FreeLunch develops a *parallel workflow* that can overlap the compression/decompression with the DNN computation. Further, FreeLunch reduces frequent memory management operations via *sliding compression workspace* and *persistent tensor buffer*. All of them are discussed as follows.

A. Parallel Workflow

FreeLunch implements a *parallel workflow* to hide the compression and decompression of the feature maps within the DNN training. It can hide these operations because the feature map of a layer generated in the forward propagation will be consumed later in the backward propagation. We can compress the feature map of a layer concurrently with the DNN training of its subsequent layer in forward propagation, and then decompresses it before the reuse in the backward propagation.

FreeLunch uses different mechanisms to trigger a compression in the forward propagation and a decompression in the backward propagation. For the forward propagation, the feature map of a layer can be compressed immediately as long as it is not used in the computation of its subsequent layers. In the implementation, FreeLunch maintains a queue of tensors to be compressed and then compresses tensors upon their appearances in the queue. The compression is performed by a thread on its own stream, which is in parallel with the normal execution of the DNN training. The backward propagation won't start until the queue is empty.

A compressed tensor should be decompressed before its usage in order to hide the decompression. FreeLunch triggers the decompression on demand. Initially, only multiple tensors

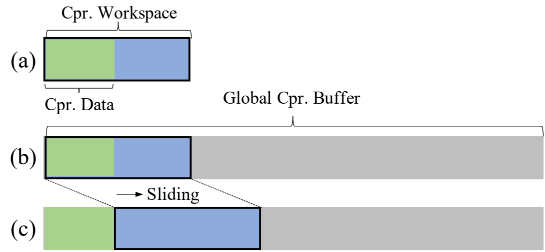

Fig. 1. Sliding Compression Workspace. (a) shows that the compression algorithm always stores the *compressed data* at the front of the compression workspace. (b) shows the layout of *global compression buffer* after the first compression. (c) shows the layout after the sliding.

are decompressed. Then a decompression will start when a new layer is computed in the backward propagation. Note that decompressing two tensors is typically sufficient to hide the decompression for the normal situation. The decompression is also performed by a separate thread running on its own stream, which is in parallel with the backward propagation.

B. Sliding Compression Workspace

As mentioned before, a naive compression algorithm requires two memory allocations, one memory copy, and two memory free operations for each compression. In particular, it needs to allocate a compression workspace that can be used to perform the compression, allocate a compressed space for storing the output compressed tensor, and free the space of the original tensor as well as the compression workspace. Similarly, the decompression needs to allocate space for the decompressed tensor and needs to free the space of the compressed tensor.

The *sliding compression workspace* mechanism is proposed to reduce the number of memory allocations, deallocations, and copies. In the forward propagation of a DNN model with n layers, this technique reduces the number of memory allocations from $2n$ to one and the number of memory copies from n to zero. This technique reduces memory operations in the backward propagation in a similar way. It is based on the observation of the DNN training: a sequence of tensors generated in the forward propagation will be accessed in the reverse order in the backward propagation. Based on this, we could append all compressed tensors in the compression workspace instead of copying them out.

The sliding compression workspace pre-allocates a *global compression buffer* and performs the compression inside. The size of the global compression buffer is determined at the setup phase when the CNN's computation graph is built. During the setup phase, each layer has its associated input and output tensors. If a tensor is eligible for compression, a space is reserved in the global compression buffer which amounts to the size of the tensor divided by the targeted compression rate. Besides the space for compressed tensors, the global compression buffer also needs to reserve enough space as the *compression workspace*, which is necessary for a compression algorithm to compress a tensor. The size of the compression workspace can be estimated via ZFP API. In summary, the size

of the global compression buffer is the size of all compressed tensors plus the size of the compression workspace.

As shown in Figure 1(a), the ZFP compression algorithm will store the compressed data at the beginning of the compression workspace. Figure 1(b) shows the result after the first compression in the global compression buffer. To avoid copying the first compressed tensor, the compression workspace will be shifted right by the size of the compressed tensor (see Figure 1(c)). Then the second compressed tensor will be appended just after the first compressed tensor. Since the compressed tensors will be decompressed in the Last-In-First-Out order, the sliding compression workspace effectively avoids any memory copy.

C. Persistent Tensor Buffers

Persistent tensor buffers also aims to reduce memory management operations. More specifically, it avoids the deallocation of the space for the original tensor after the compression and the allocation of the space for decompression. The basic idea is to pre-allocate spaces that can be reused by all tensors. Note that traditional memory pool-based solutions (e.g., tensor pool in Superneurons [3]) are not applicable because the memory space for a tensor to be compressed has to be continuous to facilitate data compression.

Persistent tensor buffers pre-allocates the minimum number of spaces called *tensor buffers* at the beginning of the training. These tensor buffers are persisted during the training. The size of a persistent tensor buffer is set to be the maximum size in order to ensure its reuse for any tensor. Currently, we statically determine the minimum number of buffers that a model needs. For example, linear models like AlexNet and VGG require only three tensor buffers while residual networks require six tensor buffers due to their residual and join connections.

D. Theoretical Analysis

This section compares the theoretical memory consumption of FreeLunch and some counterparts. Following the practice in Superneurons [3], memory consumption from both model parameters and the convolution workspace are ignored as they are the same across different approaches.

Let the memory consumption of the i-th layer be $l_i^{(f)}$ in the forward propagation and $l_i^{(b)}$ in the backward propagation. A naive memory allocation strategy is to pre-allocate independent space for each tensor, resulting in a memory consumption of $\sum_{i=1}^{n} l_i^f + \sum_{i=1}^{n} l_i^b$, where n is the number of layers in the CNN. One approach to reduce memory footprint of CNN training is liveness analysis [3], which frees gradients maps generated in the backward propagation along with the stashed feature maps. Liveness analysis results in a peak memory consumption of $\sum_{i=1}^{n} l_i^f + l_n^b$, implying up to 50% of memory savings. We next discuss how swapping, recomputation, or compression can be added on top of liveness analysis to provide more memory savings.

For the swapping policy, some feature maps are swapped out to the CPU. The peak memory consumption is reduced to $\sum_{i=1}^{n}(l_i^f \notin checkpoints) + l_n^b$, where $checkpoints$ are the tensors swapped to the CPU.

For the recomputation policy, we use the implementation from Superneurons [3], where tensors from the four least computation-intensive layers (pooling layer, activation layer, local response normalization layer, and batch normalization layers) are recomputed in the backpropagation. The peak memory consumption for this policy is $\sum_{i=1}^{n}(l_i^f \notin released) + l_n^b$, where $released$ refers to the tensors from the four layers.

The sliding compression workspace in FreeLunch pre-allocates a global compression buffer, which stores compressed tensors and acts as compression workspace for a compression algorithm to compress a tensor. It leads to a peak memory consumption of $(\sum_{i=1}^{n} l_i^f)/r + C + l_n^b$, where r is the compression rate and C is the size of compression workspace. Implementing persistent tensor buffers improves throughput but increases this peak memory consumption as k tensor buffers are pre-allocated. The peak consumption for FreeLunch with persistent tensor buffers is $(\sum_{i=1}^{n} l_i^f)/r + C + k * max(l_i^f) + l_n^b$.

IV. Evaluations

We conduct a set of experiments to examine the efficacy of FreeLunch. Our evaluation aims at answering the following questions. (1) Can FreeLunch improve training throughput while reducing memory consumption of CNN training compared with other GPU memory management policies? (2) How effective are the optimizations in FreeLunch compared with other compression-based baselines? (3) What is the effect of compression in FreeLunch on model accuracy? (4) One can combine compression with other policies to achieve memory reduction without affecting accuracy. What is the performance of hybrid policies that combine FreeLunch with other policies?

We first describe the experiment settings in Section IV-A and then report our experiment results in Sections IV-B and IV-C.

A. Experiment Settings

Workloads. We evaluated FreeLunch on five state-of-the-art CNN models and workloads, including the AlexNet [13], VGG16 [14], ResNet32, ResNet152, and ResNet256 [1]. We choose these workloads because they represent commonly-used CNN architectures. VGG and AlexNet have layers that take more computation time, whereas the ResNet models are deeper but less computation-intensive layers. We use Cifar10 [15] for ResNet32 and ImageNet [10] for the rest of these models.

Counterparts for Comparison. We compare FreeLunch to three policies from prior works as follows. The memory consumption of these policies is analyzed in Section III-D.

- *Liveness:* This policy is the liveness analysis optimization from Superneurons [3]. It allocates memory spaces for tensors from each layer in the forward propagation. Then it tracks the live tensors before and after each layer in the backward propagation and free tensors if no subsequent layers need them. Liveness analysis enables different

tensors to reuse the same physical memory at different time.

- *Swapping:* This policy is synchronous swapping from Superneurons [3]. If the system runs out of memory upon a tensor allocation, some feature maps are swapped to the CPU memory synchronously.
- *Recomputation:* This policy is from Superneurons [3], where tensors from the four least computation-intensive layers (e.g., pooling and batch normalization layers) are recomputed.

The experiments were conducted on a server equipped with Intel Xeon CPU E5-2620v3 processors with 24 logical cores, 64GB RAM and Nvidia Titan X GPU with 12GB memory. The CUDA version is 11.1 and cuDNN is 8.0.

B. Comparison to Other Policies

Figure 2 shows the performance and memory consumption of FreeLunch and other policies with different batch sizes. The compression rate of FreeLunch was set to five for these experiments. Overall, FreeLunch allows the training of larger models by reducing the peak memory footprint during training. At the same time, FreeLunch achieves higher throughput than swapping and recomputation thanks to its parallel workflow and systematic optimizations that reduce memory operations. Quantitative results are reported below.

Throughput. According to Figure 2, FreeLunch achieves much better throughput than recomputation and swapping across models and batch sizes. For example, for ResNet152, it achieves up to 70% throughput improvement over the swapping and up to 32% over the recomputation (at batch size 44).

The throughput improvement is due to two major reasons. First, FreeLunch reduces memory footprints and thus the overhead of memory operations during the training. When the available GPU memory is tight, each memory management operation will take a longer time. This also explains why the throughput of *Liveness* could be worse than FreeLunch for ResNet32 and VGG16 with large batch sizes. *Liveness* involves many management memory operations to allocate space for feature maps in forward propagation and free feature maps that are already consumed in the backpropagation. These memory operations could greatly slow down the training. In contrast, FreeLunch reduces memory management operations via memory optimizations described in Sections III-B and III-C.

Second, with larger batch size, the computation overhead of compression can be better overlapped with forward propagation in FreeLunch. The swapping, however, suffers from the increased communication overhead as the batch size increases because of the limited CPU-GPU bandwidth. Similarly, the recomputation also suffers from a higher throughput degradation because the computation time increases with larger batch size. These experiments also indicate that even when the GPU memory is exhausted, FreeLunch can still effectively overlap compression computation with the model training by leveraging idle computing resources.

We also noticed that the training throughput (images/second) become worse with larger batch sizes. This phenomenon is observed in several prior work as well [3], [4]. It is because memory operations take more time when the limited available GPU memory. Also, as mentioned in [4], the convolution operators could fall back to a slower convolution algorithm due to memory limit.

Memory Consumption. Figure 2 shows that similar to swapping and recomputation policies, FreeLunch enables CNN training with significantly larger batch sizes that is not feasible for the *Liveness* approach. The *Liveness* approach goes out of memory for larger batch sizes as shown in Figure 2. Compared to swapping, FreeLunch can accommodate the same batch size in 35% less memory and have up to 70% better throughput (see ResNet152 with batch size 44). Compared to recomputation, FreeLunch has more memory consumption for ResNet32, AlexNet and VGG16 but not for ResNet152. It is because ResNet152 has a larger number of layers and thus more feature maps to persist in GPU memory.

Memory savings from FreeLunch are higher when feature maps take more GPU memory compared with model parameters, making FreeLunch a more appealing solution for parameter-efficient and deeper CNNs (e.g., ResNets). For Alexnet [13] with batch size 128, the size of all feature maps is 940.5 MB. The size of the sliding compression workspace is 214.5 MB and the persistent tensor buffers are 425.3 MB. The memory saving for feature maps is 31%. However, the memory consumption coming from model parameters and other overheads is around 3 GB, which results in a smaller overall memory reduction. For ResNet152 with batch size 24, the total size occupied by feature maps is around 4.6 GB. The size of the sliding compression workspace is 933 MB and the persistent tensor buffers are 456.8 MB. The size of the persistent tensor buffers is much smaller than the total size of feature maps, leading to higher memory savings.

Comparison with Asynchronous Swapping. Superneurons uses synchronous memory copies for its swapping, but state-of-the-art solutions Capuchin [4] and SwapAdvisor [5] use asynchronous memory copies to overlap the swapping with the model training. For comparison, we also implemented an asynchronous swapping policy (called *async swapping*) by replacing the compress/decompress with `cudaMemcpyAsync` in the parallel workflow. We also enable persistent tensor buffers optimization to remove the `cudaMalloc/cudaFree` operations for the async swapping.

Table I shows the performance comparison of FreeLunch and the asynchronous swapping. Overall, the throughput of FreeLunch is around 4-38% higher than async swapping across all models. When we increase the batch size, FreeLunch could achieve a higher speedup because the memory transfer bandwidth is the bottleneck for the async swapping. In particular, FreeLunch can achieve a speedup of 1.2×, 1.38× and 1.14× for ResNet32 with batch size 512, VGG16 with batch size 32 and ResNet152 with batch size 44 respectively.

Comparison with the prior work [7]. The prior work [7]

TABLE I
THE IMPROVEMENT WITH PERFORMANCE OPTIMIZATIONS OVER ASYNCHRNOUS SWAPPING. THE BATCH SIZE OF ALEXNET AND RESNET32 IS 128, AND FOR THE REST IS 24.

images/second	ResNet32	Alexnet	VGG16	Resnet152	ResNet256
Async swapping	314	290	19.9	10.68	6.12
FreeLunch	327	383	26.3	11.1	6.9
speedup	1.04×	1.32×	1.32×	1.04×	1.13×

proposes to leverage data compression to reduce memory consumption during CNN training. Their focus is on how to dynamically adjust compression rate during the training process to avoid accuracy degradation while reducing memory footprint. However, their implementation lacks systematic optimization to avoid performance overhead caused by compression algorithms. The memory management framework without optimizations resembles the memory management framework in [7]. We implemented their approach in our framework and compared the throughput to FreeLunch.

Table II reports the performance improvements in FreeLunch due to the sliding compression workspace and persistent tensor buffers optimizations as compared to [7]. The two optimizations proposed in this work remove the non-trivial overheads from memory allocations and copies of parallel workflow, leading to 1.3×-1.9× speedups in throughput.

TABLE II
THE IMPROVEMENT OF PARALLEL WORKFLOW IN FREELUNCH WITH OPTIMIZATIONS IN SECTIONS III-B AND III-C. THE BATCH SIZE OF ALEXNET AND RESNET32 IS 128, AND THE REST IS 24.

images/second	ResNet32	Alexnet	VGG16	Resnet152	ResNet256
Without optimizations [7]	255	212	15.6	6	3.6
FreeLunch	327	383	26.3	11.1	6.9
speedup	1.28×	1.81×	1.69×	1.85×	1.92×

C. Performance of A Hybrid Policy with Compression and Async Swapping

Prior works [7] along with our analysis on accuracy have shown that accuracy can be maintained if tensors from only convolution layers are compressed. Figure 3 shows the effects of the compression algorithm (ZFP) on the model convergence using ResNet32 on Cifar10. Compression applies to the tensors from convolutional layers. The compression ratio for these experiments is set to 8. The accuracy curve with compression is similar to the one without compression, indicating a minor influence on model convergence. These results echo the observations in prior work [7], which reports no or negligible model accuracy degradation with proper compression rates.

To further reduce memory footprint without affecting accuracy, one can combine compression with other policies and apply compression on tensors from certain layers and apply other policies (e.g. swapping or recomputation) on the tensors from the rest of the layers. In this section, we present the performance results of a hybrid policy that combines FreeLunch with async swapping. We use FreeLunch on the activation maps from convolutional layer and perform async swapping on the intermediate results from other layers. This policy is implemented in the parallel pipeline along with the optimizations described in the paper. The compression ratio is set to 8, the same as our accuracy experiments.

Table III shows the results of async swapping compared to the memory management policy described above. Overall, although this hybrid policy doesn't have better performance than FreeLunch alone, it has much better performance than just async swapping. It demonstrates that compression can help us achieve better throughput when integrated with other policies that have been optimized in previous frameworks.

TABLE III
ASYNC SWAPPING COMPARISON WITH FREELUNCH + ASYNC SWAPPING

images/second	ResNet32	Alexnet	VGG16	Resnet152	ResNet256
Async swapping	314	290	19.9	10.68	6.12
FreeLunch + Async swapping	318	366	23.3	11.52	6.9
speedup	1.012×	1.26×	1.17×	1.07×	1.13×

V. RELATED WORK

CNN memory management is an active area of research owing to the constantly increasing memory requirements of these models. Recent works [2]–[5] have focused on using the policies of swapping or recomputation in different capacities. vDNN [12] and Superneurons [3] use swapping by offloading and prefetching tensors. They use a static computation graph to make memory policy decisions. Capuchin [4] uses a similar approach but it leverages the predictive nature of tensor accesses during CNN training. They use a timestamp-based technique to decide when to prefetch a tensor asynchronously and update the time of prefetch if the model training has to synchronize. FreeLunch is based on a similar observation but instead of using time, it leverages the order of accesses to improve the throughput. Swapadvisor [5] trains RL simulations to find the best order of memory swapping, allocations, and compute to perform in the system. They constrain their search space by using a size-class based memory allocator.

Recomputation has also been incorporated into Superneurons [3] and Capuchin [4]. Superneurons statically decides the layers to recompute and then drops their tensors in the forward phase and regenerates them in the backward phase. Capuchin shows that this static policy cannot work and uses a heuristic to decide when to use recomputation compared to swapping. While FreeLunch does use more memory than Recomputation, it offers better throughput and can be executed in parallel. Recomputation is harder to execute in parallel because of lineage dependencies and complex forward kernels.

Compression as a memory management technique has been explored in Gist [9]. Gist includes a fixed lossless compression policy that applies to fixed layer combinations, for example CONV-ReLU. They also implement a lossy compression policy, which is similar to quantization. They encode 2,3 and 4 values inside 4 bytes using their own encoding technique and call them FP16, FP10 and FP8. The maximum compression they can obtain is 4×. While quantization can have faster implementations, this work is motivated by newer techniques that can offer much higher compression ratios, in our work we

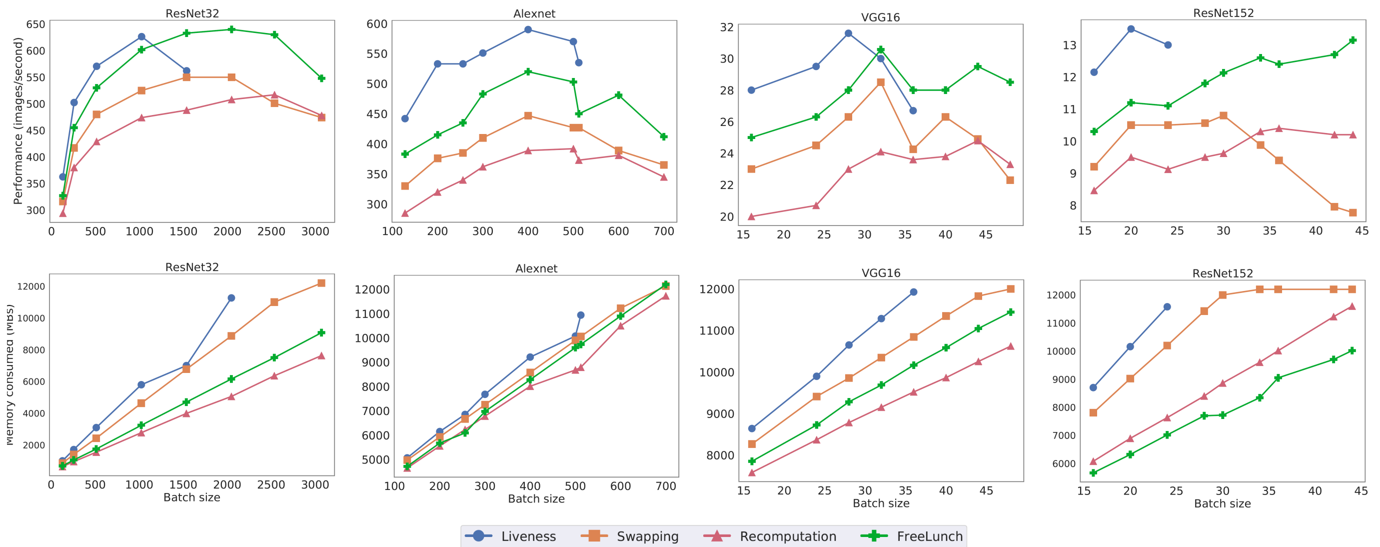

Fig. 2. Performance and memory comparison of various policies during model training. FreeLunch has up to 35% less memory consumed and up to 70% better throughput as compared to swapping and recomputation. The *Liveness* approach goes out of memory for larger batch sizes.

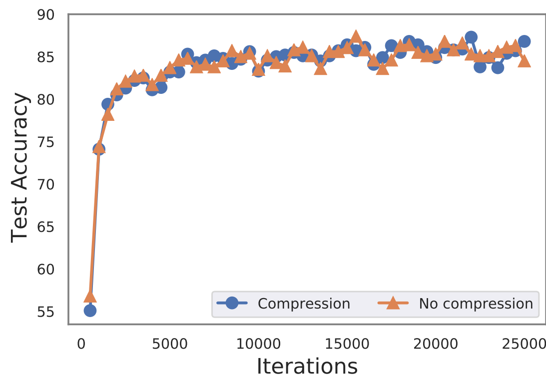

Fig. 3. Accuracy for FreeLunch vs no compression (ResNet32 on Cifar10). Compression rate is eight. Compression is applied to tensors from convolutional layers. It shows that a higher compression ratio than five can be used without having an effect on the accuracy of the training process.

used a compression ratio of $5\times$ and $8\times$ but the compression ratio can go upto $13\times$ if using a dynamic compression rate schedule [7], leading to higher memory savings. Notably, Gist also uses a static memory allocator from CNTK [16] to hide the overhead caused by dynamic memory management. The allocator attempts to reuse memory spaces for tensors that have different temporal lifetimes and requires an analysis of temporal uses of tensors as input. The memory consumption of this allocator will be similar or higher than the persistent tensor buffers optimization as we only allocate enough space for tensors that share temporal lifetimes.

VI. Conclusion

Previous CNN memory management policies incur significant overhead due to the limited CPU-GPU memory capacity or the lack of a parallel implementation. We introduce FreeLunch a compression-based policy that improves throughput over previous policies by up to 70% while consuming up to 35% less memory. FreeLunch can be incorporated into existing frameworks like Capuchin [4]. For example, Capuchin proposes a mechanism to decide the timings to swap in/out a tensor. A similar mechanism can be applied to determine the timings to compress/decompress tensors. Also, we use an additional cudaStream for compression/decompression, which could potentially decrease training throughput due to GPU resource contention. To address the problem, understanding GPU utilization from each forward and backward layer computation can help us decide the best timings for compression and decompression.

References

[1] 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.

[2] 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 *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[3] 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 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18, 2018, p. 41–53. [Online]. Available: https://doi.org/10.1145/3178487.3178491

[4] 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 Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 891–905.

[5] C.-C. Huang, G. Jin, and J. Li, "Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1341–1355.

[6] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and

gpudirect," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, 2019.

[7] S. Jin, G. Li, S. L. Song, and D. Tao, "A novel memory-efficient deep learning training framework via error-bounded lossy compression," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 485–487.

[8] P. Lindstrom, "Fixed-rate compressed floating-point arrays." [Online]. Available: doi: 10.1109/TVCG.2014.2346458

[9] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient data encoding for deep neural network training," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 776–789. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00070

[10] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[11] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *arXiv preprint arXiv:1604.06174*, 2016.

[12] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," 2016.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. Red Hook, NY, USA: Curran Associates Inc., 2012, p. 1097–1105.

[14] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2015.

[15] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)." [Online]. Available: http://www.cs.toronto.edu/ kriz/cifar.html

[16] F. Seide and A. Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 2135. [Online]. Available: https://doi.org/10.1145/2939672.2945397