Capuchin: Tensor-based GPU Memory Management for Deep Learning

Xuan Peng[†], Xuanhua Shi[†*], Hulin Dai[†]
Hai Jin[†], Weiliang Ma[†], Qian Xiong[†], Fan Yang[★], Xuehai Qian[‡]
[†]*National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology*
[★]*Microsoft Research Asia*, [‡]*University of Southern California*

Abstract

In recent years, deep learning has gained unprecedented success in various domains, the key of the success is the larger and deeper deep neural networks (DNNs) that achieved very high accuracy. On the other side, since GPU global memory is a scarce resource, large models also pose a significant challenge due to memory requirement in the training process. This restriction limits the DNN architecture exploration flexibility.

In this paper, we propose *Capuchin*, a tensor-based GPU memory management module that reduces the memory footprint via tensor eviction/prefetching and recomputation. The key feature of *Capuchin* is that it makes memory management decisions based on dynamic tensor access pattern tracked at runtime. This design is motivated by the observation that the access pattern to tensors is regular during training iterations. Based on the identified patterns, one can exploit the total memory optimization space and offer the fine-grain and flexible control of *when and how* to perform memory optimization techniques.

We deploy *Capuchin* in a widely-used deep learning framework, Tensorflow, and show that *Capuchin* can reduce the memory footprint by up to 85% among 6 state-of-the-art DNNs compared to the original Tensorflow. Especially, for the NLP task BERT, the maximum batch size that *Capuchin* can outperforms Tensorflow and gradient-checkpointing by 7× and 2.1×, respectively. We also show that *Capuchin* outperforms vDNN and gradient-checkpointing by up to 286% and 55% under the same memory oversubscription.

* Corresponding author.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378505

CCS Concepts • **Software and its engineering → Memory management**; • **Computer systems organization →** *Single instruction, multiple data.*

Keywords deep learning training; GPU memory management; tensor access

ACM Reference Format:
Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, Xuehai Qian. 2020. *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 (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3373376.3378505

1 Introduction

Deep learning applications are both compute and memory intensive. Deep neural networks (DNNs) become very popular nowadays due to the tremendous compute power provided by several kinds of emerging heterogeneous hardware, such as TPU [17], ASIC, and GPU — currently the most prevalent training option. The training of the increasingly deeper and wider DNNs poses a great challenge for on-board GPU memory. For example, the latest BERT [9] has 768 hidden layers consuming 73 GB memory (with batch size 64) in training. However, the high-bandwidth GPU on-board memory is a scarce resource: the latest most powerful NVIDIA GPU V100 has only maximum up to 32 GB on-board memory while the mainstream GPU type in commercial cloud, e.g., P100, has only 16 GB on-board memory. This restriction limits the flexibility of exploring more advanced DNN architectures.

Prior works [13, 24] show that the major memory footprint in DNN training is due to intermediate layer outputs, i.e., feature maps. These feature maps are produced in the forward propagation and used again in the backward propagation. Thus, major deep learning frameworks such as Tensorflow [2], MXNet [4] and Pytorch [23] usually maintain these feature maps in GPU memory until they are no longer needed in backward propagation computation. However, there is usually a large gap between two accesses to the same feature map in forward and backward propagation, which incurs high memory consumption to store the intermediate results.

To tackle this challenge, there are two main techniques to reduce the memory footprint: swapping and recomputing. Both methods are based on the design principle of releasing the memory for feature maps in the forward propagation and re-generating the intermediate feature maps in backward propagation. They differ in how re-generation is performed. Specifically, *swapping* leverages the CPU DRAM as a larger external memory and asynchronously copies data back and forth between GPU and CPU; while *recomputing* obtains the needed intermediate feature maps by replaying the forward computation. Both approaches do not affect training accuracy. An alternative approach is to use low precision computations during training to save memory consumption. However, it is not always easy to analyze the the effects of lower precision computation on the final training accuracy. This paper does not consider this option.

Prior works [5, 24, 31] perform layer-wise GPU memory management based on computation graph and characteristics of different layers. However, a layer of neural networks is a high-level computation abstraction composed of many low-level mathematical functions, even a neural network with tens of layers may contain thousands of nodes in its corresponding computation graph. For instance, there are more than 3000 nodes in ResNet-50, 7000 nodes in BERT when they are converted to the internal computation graph in Tensorflow. Therefore, this coarse-grain memory management limits the optimization space. Moreover, they choose swapping and recomputation through *static* analysis on the computation graph. For example, based on the assumption that convolution layer is more expensive and time-consuming to compute, vDNN [24] chooses the input of convolution layer as swapping target, hoping to increase the overlap of swapping with convolution layer computation. SuperNeurons [31] also avoids recomputation of convolution layer to reduce overhead.

The static analysis leads to three problems: (1) The heterogeneity of the hardware and input sizes make it difficult to predict computation time, which varies largely even for the same type of layers. Thus, determining memory optimization targets statically based on layer type will limit potential of memory optimization. (2) The decision based on coarse-gained "qualitative" information cannot quantify the overhead of a specific memory operation, making it difficult to prioritize the memory optimization candidates or make choices between swapping and recomputation. (3) The DNNs are continuously and rapidly evolving, e.g., from Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) to Transformer and Graph Neural Network (GNN), even with user-defined operations. For the new type of DNNs, apriori knowledge is not available. In addition, the memory management based on computation graph does not work well for deep learning frameworks that do not have computation graph before execution, e.g., Pytorch [23] and Tensorflow's eager mode [3]. Thus, this approach is not general enough to be applied to all frameworks.

In this paper, we propose *Capuchin*, a tensor-based GPU memory management module for deep learning frameworks to reduce the memory footprint via tensor eviction/prefetching and recomputation. The key feature of *Capuchin* is that it makes memory management decisions based on tensor access pattern tracked at runtime with a unique ID for each tensor. This approach enables memory management at the granularity of tensor, allowing specific decisions to be triggered by certain tensor access pattern. *Capuchin* not only makes tensor eviction decisions but also determines the timing of prefetching and recomputation. The module is transparent to the user and the implementation just needs a few supports from the underlying deep learning frameworks. Essentially, *Capuchin* iteratively refines memory management policy by collecting feedbacks from runtime tensor accesses.

The design of *Capuchin* is based on two key observations. First, all deep learning frameworks are based on the dataflow execution model, in which the processing procedures are based on *tensor operations*. Similar to reuse and locality analysis of memory accesses for traditional programs, tensor accesses in deep learning training also exhibit data reuse and certain access patterns. Thus, we believe that dynamically tracking fine-grained tensor accesses is a fundamental and general technique that enables effective memory management optimizations. This paper demonstrates that this essential idea can be implemented efficiently on top of major deep learning frameworks.

Second, the properties of deep learning applications ensure the effectiveness of our approach. The training process is composed of *millions of iterations* with clear boundaries, and the tensor accesses have *regular and repeated* access patterns across iterations. This means that analyzing the timing and tensor access patterns can easily reveal the memory optimization opportunities with concrete guidance, e.g., whether and when to choose eviction/prefetching or recomputation. In addition, such memory management policy can be applied timely to the next iteration and refined iteratively from runtime feedbacks. To the best of our knowledge, *Capuchin* is the first computation graph agnostic memory optimization module that can be applied in imperative programming environment.

We implemented *Capuchin* on top of the widely used deep learning framework, Tensorflow. We evaluate *Capuchin* on P100 GPU and show that: (1) *Capuchin* can accommodate larger batch size by up to 9.27× and 2.14× compared to Tensorflow and OpenAI's gradient-checkpointing [1] , respectively; (2) *Capuchin* achieves up to 3.86× speedup over vDNN. We also show that, in Tensorflow's eager mode, *Capuchin* increases batch size by 2.71× while no other works are capable of optimizing memory in this mode.

2 Background

2.1 Deep Learning Model Training

DNNs are the key ingredients that enable the success of deep learning and typically consist of an input and output layer with multiple hidden layers in between them. Various kinds of layers are used for different purposes and applications, such as Convolution layer, Pooling layer, Activation layer in image classification, Attention layer in Natural Language Processing (NLP). With the increasing applications of deep learning, DNN models are rapidly and continuously evolving. Even considering the existing layers, the connection between these layers can be arbitrary explored to seek suitable neural network architecture for various problems, e.g., ResNet [14].

The goal of DNN training is to find an appropriate set of model parameters to satisfy an *objective function*, which expresses the desired properties of the model, such as the error incurred when identifying the object categories in image classification. The training process typically consists of millions of iterations, each comprised of two computation phase, i.e., forward and backward propagation. In forward propagation, input feature maps, current layer's weights and bias produce the output feature maps which become the next layer's input data. The forward propagation concludes with the calculation of loss by comparing the output with ground truth label at the output layer. The backward propagation starts from the output layer and reversely traverses layers to optimize the weights and bias. There are many optimizers for backward propagation algorithm [20], including the most commonly used stochastic gradient-descent (SGD), Momentum [36], and Adam [18], etc.

To achieve high throughput, training samples are fed into computation phases in *batch*. It has also been demonstrated that, with larger batch, the training tends to achieve higher accuracy, recent work [10] shows the use of batch up to 8K.

Memory usage. In the training process, the GPU memory is mainly consumed by three parts: feature maps, i.e., output in the forward propagation; gradient maps, i.e., output in the backward propagation; and convolution workspace, i.e., extra memory space needed by convolution algorithm. Compared to them, the model weights consume very small amount of memory and are usually persistent in GPU memory to be continuously updated. Among the three parts, the latter two are temporary memory usage which can be released immediately after current computations are finished. Nevertheless, some recent works [24, 31] attempt to improve performance by dynamically choosing convolution workspace. Similar heuristic is also implemented in Tensorflow and cuDNN [6] and considered in our evaluation (Section 6). The feature maps are needed in both forward and backward propagation. However, there exists a large time gap between the two usage points for computations in forward and backward phase. Due to this reason, the memory consumption of feature maps became the major optimization target in prior works.

2.2 Deep Learning Framework Execution Modes

The current popular deep learning frameworks are typically based on either imperative or declarative programming. The imperative programs are similar to Python and C++ programs, which perform computations during the execution. In Tensorflow, it is called *Eager Execution*. PyTorch adopts it as the default and only execution mode. In contrast, declarative programs are based on *Graph Execution*, in which the actual computations do not take place when the functions are defined. It is supported by many frameworks including Tensorflow, CNTK [26] and Caffe [14]. In the following, we compare the two in more details.

Eager Mode. The imperative programming environment evaluates operations immediately without building graphs. In this mode, the operations return concrete values immediately instead of constructing a computational graph to execute later. Obviously, this approach makes it convenient to deploy and debug a new model. In particular, eager mode is popular in academic community for its convenience of quickly prototyping a new model. Following this trend, Tensorflow announced that eager mode will become its default execution mode from Version 2.0. On the other side, this mode can be slower than executing the equivalent graph due to the overhead of interpreting Python code and lack of graph optimizations.

Graph Mode. In this mode, the computations of a model are abstracted as *computation graph* specifying the data flow among computation nodes connected by edges that indicate flow of tensors. The computation graph is built before execution starts, and the actual computations are scheduled only when necessary, thus it is also known as lazy programming. Certain optimizations (e.g., pruning and constant folding) can be applied to the program when it is converted to the internal computation graph. Thus, the memory consumption and training performance may be better than the eager mode.

Based on the above discussion, we believe that the efficient memory management for imperative programming environment is very important due to its popularity and convenience.

3 Observation and Motivation

3.1 Limitation of Static Analysis

Synchronization overhead in swapping. The performance of swapping in prior works depends on the swapping time and the execution time of the layer which tries to overlap the swapping. The swapping time is determined by memory size and lane bandwidth between CPU and GPU while a layer's execution time depends on the GPU model and its input

Figure 1. Synchronization Overhead of vDNN on Vgg16.

scale. We pro le execution of vDNN on Vgg16 (batch size is 230) using P100 GPU equipped with PCIe 3.16, and show the maximum size memory's swapping timeline as Figure 1. The execution time of each operation is proportional to the length of each block. We can see that the time of swapping out/in are more than 3 as much as the overlapped layer's execution time, thus introducing signi cant synchronization overhead. With the prevalent synchronization overhead at each swapping out/in, the total performance loss reaches 41.3%.

This performance penalty is due to the xed control on two aspects of static analysis: (i) when to release memory after swapping out; (ii) when to prefetch. The synchronization overhead is huge when the current layer's execution time is inadequate to overlap data transfer, and such overhead persists in subsequent iterations.

Execution time variation of same type of layer. Prior works make memory management decisions according to layer type. Given the high computation cost of convolution, vDNN [24] tries to overlap data transfer with computation of convolution layer while Chen et al. [5] avoids recomputing convolution layer. This is based on the empirical assumption, i.e., computation of convolution layer is time-consuming. We observe that the execution time of di erent convolution layers in one neural networks varies largely, as shown in Figure 2 when running InceptionV3 on P100 GPU. The minimum and maximum execution time are 474us and 17727us, respectively, which varies as large as 37. Among these 94 convolution layers, 95.7% of their execution time are less than 3ms while the left four exceed 8ms.

This brings two problems: (i) It is still di cult to overlap the data transfer perfectly with the computation of convolution layer on a powerful GPU. 3ms can only overlap a data transfer within 40 MB under PCIe 3.0 16. (ii) Given that convolution layer occupies the majority layers in CNN, completely neglecting convolution layer as a recomputation target will dramatically limit the memory optimization space. In fact, recomputing the low overhead convolution layers is quite feasible since the computation time (less than ms) only account for less than 1 3 of one iteration's time (over 1s).

Figure 2. InceptionV3 Convolution Layers Execution Time.

3.2 Opportunity: Regular Tensor Access Pattern

The deep learning training consists of millions of iterations. To understand the tensor access pattern, we choose three tensors in ResNet-50 and pro le their access timestamp (relative to the time at which each iteration begins) at iteration 5, 10, 15 on P100 GPU, the result is shown as Figure 3. The tensor access clearly follows a regular pattern, i.e., the number of occurrences and timestamps in a iteration are mostly xed. The T1 appears four times in which two are in the forward phase and the other two are in the backward while T2 and T3 appear six times. The time variance of the same tensor access across iterations is less than 1ms.

Although the results only show the behavior a CNN task running in graph mode, we found that other kinds of workloads such as speech, NLP and using eager mode exhibit a similar pattern. Leveraging the regular tensor access pattern in deep learning training, Capuchin derives the smart memory management policy. First, we can clearly identify those tensors which can be used to reduce memory footprint. Second, the tensor access time interval is e ective information to make the memory management decision, discussed in Section 4.3. In Figure 3, it is obvious that doing swapping for T1 is more bene cial than doing so for T2 and T3 since the time interval of two consecutive accesses is bigger for T1. This decision is more likely to reduce overhead (assume three tensors have the same size).

Figure 3. ResNet-50 Tensor Accesses Timeline.

4 Capuchin Memory Management Mechanisms

4.1 Design Goals

We attempt to achieve two goals in the design of Capuchin. First, we should make the best e ort to minimize overhead

It is well-known that the training of modern DNNs can take days or even weeks, even moderate performance overhead (e.g., 50%) can significantly increase the overall cost of training due to the use of expensive GPU resources. Thus, the design of Capuchin must minimize performance overhead with memory oversubscription. Second, the framework should be general and incur little code modification for different deep learning frameworks. It can be achieved by identifying general design characteristics among frameworks.

4.2 Design Overview

Due to the regularity of tensor access pattern, we divide the training into two phases: (1) measured execution on the execution of the first mini-batch (iteration) from where we observe the dynamic tensor access sequence, based on which tensor memory management decisions can be made; and (2) guided execution on the training process after the first iteration based on the memory management policy generated using observed tensor access patterns.

The essential goal of Capuchin is to support large batch sizes that are not possible in the current frameworks. Thus, during the whole execution Capuchin should be able to continue execution even if Out of Memory (OOM) and Access Failure occur. To achieve this goal, the system needs to intercept the tensor accesses on null pointer and then trigger tensor eviction. We call this *passive mode* which performs on-demand swapping. On a tensor eviction, the tensor's memory will be copied to CPU memory synchronously and the ID of the tensor will be recorded in the framework so that it can be swapped in from CPU memory when later accessed again.

In essence, the passive mode behaves similar to the virtual memory in traditional OS, which operates at page granularity for swapping in/out and uses disk as an extended space beyond CPU memory to provide the illusion of large virtual memory space. In comparison, our system operates at tensor granularity and uses CPU memory as an extended buffer for limited GPU memory. With passive mode, Capuchin can continue execution even of the required memory is larger than the available GPU memory with one exception. When a computation function takes tensor A as input and produces tensor B as the output, if the size of A and B combined is already larger than the whole GPU memory, our passive mode cannot handle this case since there is no chance the tensor swapping can be performed. However, we believe this situation is extremely rare.

During measured execution, Capuchin keeps the tensor access sequence with additional information for each tensor, including access count, timestamp, and operation that produced a tensor and its input tensors for recomputation decisions. With passive mode, the eviction of tensor A is always triggered by the access of another tensor B, at which point memory is insufficient and cannot allocate space for B. It is similar to cache replacement. However, the eviction (or swap-out) takes time and is in the critical path of execution. To improve performance, Capuchin leverages the similar idea of self-invalidation in cache coherence protocol. Specifically, based on the observed tensor access pattern, the framework can decide to proactively evict the tensor A after an access to itself. Compared to on-demand swapping, the proactive eviction can release memory early for other tensors in future and hide the swap-out overhead.

When an evicted tensor is accessed again, it needs to be re-generated. As discussed before, Capuchin considers two methods — swap and recomputation. Similar to proactive eviction, to minimize the overhead of tensor re-generation, we can also initiate tensor generation before it is used. The key question is to select an appropriate time point that is neither too early, which will unnecessarily increase memory pressure; nor too late, which will expose re-generation overhead in execution critical path. To decide the right time point, the we need to determine the cost of swap and recomputation.

Before discussing the detailed mechanisms, we first define several terms. The *evicted-access* is a tensor access that triggers the self-eviction after it is used in the computation. *Back-access* is the first tensor access after it is evicted from GPU memory — the access after the tensor's evicted-access. Intuitively, longer interval between evicted- and back-access provides better opportunity to hide tensor re-generation overhead. It is important to note that when back-access is performed the tensor may or may not be in GPU memory. Without proactive tensor re-generation, the tensor is guaranteed not to be in GPU memory — adding swap-in overhead in critical path. To reduce such overhead, tensor A's re-generation can be triggered earlier by another tensor access called *in-trigger*. It can be an arbitrary access between a tensor's evicted-access and back-access. When it is not specified, we can consider that back-access is served as the trigger to bring in the tensor, after access failure.

In the following, we will answer the natural questions based on Capuchin's general approach.

- How to estimate swap and recomputation cost?
- What tensors to evict?
- When to evict and re-generate?
- How to re-generate (swap vs. recomputation)?

4.3 Estimating Swap and Recomputation Benefit

For both tensor re-generation methods, i.e., swap and recomputation, the goal is to hide the overhead as much as possible to minimize the wait time of back-access. In general, for swap, we should increase the overlap between swap and normal computation; for recomputation, we should select cheap operations to recompute. The key to achieve these goals is the accurate estimation of computation time and swapping time

Figure 4. An example of evaluating the swap and recomputation for a tensor. T_3 is evicted in its second access. T_1 is an input of it.

Capuchin estimates the overhead based on the profiling of access time in measured execution, as shown in Figure 4.

The SwapTime can be calculated by dividing tensor's memory size by PCIe bandwidth. The PCIe bandwidth is quite stable and can achieve 12 GB/s under a PCIe 3.0 with 16 channels. To quantify how promising to swap a tensor, we introduce a metric Free Time (FT).

$$FT = SwapInStartTime - SwapOutEndTime \quad (1)$$

SwapOutEndTime is equal to the evicted-access time plus the SwapTime, and SwapInStartTime indicates the time to prefetch this tensor. The ideal SwapInStartTime can be calculated by subtracting the SwapTime from back-access time.

For recomputation, the challenge is how to estimate the cost of recomputing an operation, which is tightly related to the operation algorithm, device computing capability and the inputs size. The Hyper-Q technology complicates the matter even more due to the interference between parallel kernel computation, making it nearly impossible to estimate through static analysis. Therefore, Capuchin measures recomputation time during measured execution by recording tensors' lineage and runtime access time. Specifically, the recomputation time of an operation can be obtained by comparing the access time of output and input tensors. In this way, it is feasible to assess the overhead of recomputing a tensor starting from arbitrary recomputation sources.

Similar to swap, we define Memory Saving Per Second (MSPS) to indicate how favorable to recompute a tensor, i.e., the higher memory reduction with less recomputation time will lead to a higher MSPS.

$$MSPS = \frac{Memory\ Saving}{Recomputation\ Time} \quad (2)$$

Given a tensor, the memory saving is a constant but the recomputation time varies with where we start to calculate this tensor. Next, we discuss how to calculate the swap and recomputation time.

4.4 Determining Tensor Re-generation Cost

This section discusses how to estimate the cost of tensor re-generation. The goal is select the appropriate time to initiate the operation, i.e., swap or recomputation, so that it does not increase memory pressure (too early) or adds wait time in the execution (too late). We consider swap and recomputation separately and will discuss the choice of the two in Section 4.5.

For swap, we can infer the latest time to prefetch a tensor by calculating SwapInStartTime (back-access time minus SwapTime), then we can traverse reversely from the back-access in tensor access list to look for the first tensor access whose time is earlier than SwapInStartTime. This calculation is based on the ideal scenario without considering the overall execution environment and other tensors. In reality, the time may not be accurate or needs to be slightly adjusted. First, the in-trigger should not be set at a point within the time range of peak memory. Doing so will likely cause the eviction of other tensors, similar to on-demand swapping in passive mode. Second, because pinned memory transfer occupies unidirectional PCIe lane exclusively, a swap cannot start until its preceding swap finishes. Therefore, the actual swap-in may happen later than in-trigger.

To reduce the uncertainty due to dynamic scheduling and PCIe interference between multiple swaps, we introduce feedback-driven adjustment in Capuchin to adjust the in-trigger time of a tensor dynamically at runtime. The key insight is to get the runtime feedback at the back-access of a tensor, if the tensor is still being swapped in, it means the in-trigger time should be adjusted earlier. The mechanism can be easily implemented by keeping the swapping status in the tensor data structure. The detailed data structure design will be discussed in Section 5.2. In our experiments, based on the feedback, the in-trigger is adjusted earlier by 5% of its swapping time in the next iteration.

The recomputation differs from swap which only needs the PCIe resource and can be overlapped with current computation. Instead, the recomputation also needs GPU computing resource, whether it can be overlapped with current computation depends on the free GPU resources at that time. According to our observation, the GPU computing resource is usually also run out when GPU memory is insufficient. For this reason, we do not set in-trigger for recomputation and perform it in on-demand manner.

We still need to estimate the recomputation time from the measured execution to obtain the MSPS in tensor selection for eviction. It is more complex than swap time due to the dependence in tensor lineage. Take the lineage $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$ as an example, and assume T_2, T_3 and T_4 are recomputation candidates. The cost of recomputing T_4 depends on the recomputation source. If the last access to T_3 is at the computation of T_4 in forward propagation, T_3 must be unavailable when recomputing T_4 since it will be evicted from GPU memory after its last access. In this case, T_4's recomputation will start from T_2. Even if T_2 is also a candidate, in the calculation we consider that it is in GPU memory, which means that we do not need to recompute it from T_1. Alternatively, when T_3 is also accessed in backward

propagation, we compare T_3's last access time with T_4's back-access time. If the former is larger, we assume T_3 is available at T_4's recomputation, and T_4 can be recomputed from T_3; otherwise, T_3 itself also needs to be recomputed. Based on this method, we can generate the recomputation cost of each candidate tensor with two pieces of information available: (1) the lifetime of tensors in the lineage to determine whether a tensor (not one of the recomputation candidates) can serve as the source; and (2) whether a tensor in the lineage is recomputation candidate – they are assumed to be in GPU memory. The key observation from the above calculation is that once a tensor is confirmed for recomputation, it will affect the MSPS of the other candidate tensors. The measured execution can obtain (1) and we will discuss the generation of candidates and iterative MSPS update algorithm next in Section 4.5.

4.5 Selecting Memory Optimization

In this section, we consider how to make the choice between swap and recomputation for a selected set of tensors. Based on the previous discussion, it is clear that swap can be largely overlapped with computation, introducing small or no overhead; while the recomputation will certainly incur performance penalty since it requires computing resources. Therefore, we choose the swap as the first choice until we cannot choose a in-trigger to perfectly hide the prefetching overhead. At this point, we will consider both options by comparing the smallest overhead of swapping or recomputing all tensors in consideration, respectively. The tensor that incurs less overhead will be selected. The procedure is shown in Algorithm 1. Next we explain each step in detail.

Algorithm 1: Hybrid Policy

```
Input : Tensor access sequence (tas), mem_saving
candidates ← IdentifyAndSortCandidates(tas);
for t in candidates do
    ChooseSwapInTrigger(t);
    s_overhead ← SwapOverhead(t);
    r_overhead ← RecomputeOverhead(t);
    if s_overhead < r_overhead then
        Swap(t);
    end
    else
        Recompute(t);
    end
    mem_saving ← mem_saving − t.mem;
    if mem_saving ≤ 0 then
        break;
    end
    UpdateCandidates(candidates);
end
```

Based on the tensor access sequence obtained in measured execution with passive mode, we insert a tensor to the eviction candidate set if: (i) access count of the tensor is more than one; (ii) the access of the tensor occurs in tensor lives in peak memory usage period. During profiling, we can keep track allocation and deallocation time of tensors to infer memory usage.

Next, we determine the tensor eviction set, containing tensors that are confirmed to be evicted, using swap. Based on the access lists of tensors in eviction candidate set, we generate a ranked list based on the length of FT between two consecutive tensor accesses, assuming the tensor is swapped out between the two accesses. Since longer FT provides more opportunity to hide tensor swapping overhead, we select tensor access pairs for eviction from the top of the list with no overhead. The selected tensors are removed from eviction candidate set. Using this method, eviction set contains the list of {evicted-access, back-access} pairs of selected tensors. During guided execution, Capuchin will trigger the eviction of the specific tensors at evicted-access and re-generate the tensor for the back-access by the in-trigger determined using swap overhead discussed in Section 4.4. Note that the required memory reduction to make the whole training fit in GPU memory can be obtained from measured execution with passive mode – the amount is the total memory size of evicted tensors. If the reduced memory using swap is larger than this amount, we have selected enough tensors in eviction set and the process can terminate. Otherwise, we will further consider both swap and recomputation for the tensor left in the candidate set.

To select evicted tensors with recomputation, we need to calculate MSPS for all tensors in the current candidate set using the method described in Section 4.4. The major complication for recomputation is that once a tensor is chosen to be inserted in eviction set, the recomputation source of other tensors in the candidate set will be affected, eventually resulting in the change of MSPS. We insert tensor into eviction set iteratively by: (1) update: computing the MSPS for tensors in the current candidate set based on the current candidate set and eviction set; and (2) selection: inserting the tensor with the largest MSPS to eviction set and remove it from candidate set. To illustrate the affect on MSPS with chosen tensors, let us consider an example. In $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$, suppose the candidate set is $\{T_1, T_2, T_4\}$ and we choose T_2 to insert into the eviction set first. We need to update the MSPS for the other tensors for the next selection. For T_4, the recomputation source will not be T_2 but T_2's source T_1, it will affect T_4's recomputation time. Moreover, all the recomputation targets with T_4 as the source should also start from T_1. This makes the computation from T_1 to T_4 repeat multiple times. For T_1, T_2 and all the recomputation targets whose sources include T_2 will repeat T_1's recomputation multiple times. We reveal this extra recomputation overhead by adding the repeated recomputation time multiple times in the MSPS calculation

of T_4 and T_1 respectively, which are corresponding to the line 23-28 and 30-32 of Algorithm 2.

After choosing a tensor for recomputation, its overhead is compared with the overhead of the chosen tensor for swapping, the one with smaller overhead is finally confirmed. The other one is inserted back to candidate set for further consideration.

Algorithm 2: Recomputation policy

```
Input  : candidates, mem_saving
Output : recomputation targets
1  InitMSPS(candidates);
2  recomps ← {};
3  while mem_saving > 0 do
4      t ← MaxMSPS(candidates);
5      ext_ct ← 1;
6      foreach rp in recomps do
7          if t ∈ rp.srcs then
8              rp.srcs.Remove(t);
9              rp.srcs.Add(t.srcs);
10             ext_ct += 1;
11         end
12     end
13     recomps.Add(t);
14     candidates.Remove(t);
15     mem_saving -= t.mem;
16     /* Update candidates' MSPS */
17     foreach cand in candidates do
18         if t ∈ cand.srcs then
19             cand.srcs.Remove(t);
20             cand.srcs.Add(t.srcs);
21             cand.rp_time += t.rp_time;
22             cand.ext_time ← 0;
23             foreach rp in recomps do
24                 if cand ∈ rp.srcs then
25                     cand.ext_time += cand.rp_time;
26                 end
27             end
28             cand.UpdateMSPS();
29         end
30         if cand ∈ t.srcs then
31             cand.ext_time = ext_ct × cand.rp_time;
32             cand.UpdateMSPS();
33         end
34     end
35 end
```

5 Capuchin Implementation

This section explains the implementation of Capuchin, we first discuss the requirements on the underlying deep learning framework, and show the system architecture with two key modules. Then we propose two performance optimizations. Finally we discuss the specific issues related to GPU architecture.

Figure 5. Capuchin System Architecture.

5.1 Framework Requirements

Capuchin assumes that the underlying framework has the two modules.

Executor. It refers to the processing unit for the basic computing operations. Typically, the outputs can be produced by a vector of tensors and a computing kernel as inputs. In order to obtain tensor access sequence, we can instrument `RecordTensorAccess` function before and after the computation on the tensor. To support Capuchin's recomputation service, we adopt an on-the-fly lineage-based recomputation, similar to Spark's RDD lineage. Instead of being built before execution for fault-tolerance purpose, Capuchin's lineage is generated at runtime. Given a tensor's unique ID as the parameter, we can search for the closest input tensors for recomputation.

Allocator. It is a wrapper of the raw `cudaMalloc` and `cudaFree` for dynamic GPU allocation, which usually provides the two methods for the higher level modules, i.e. `Allocate` and `Deallocate`. In general, `Deallocate` will be called automatically when a tensor's reference count reaches zero. In order to support Capuchin's swapping service, allocator needs to support two extra methods, i.e. `SwapOut` and `SwapIn`. These two methods accept an address as parameter, and then allocate an equal size memory at GPU or CPU according to destination device. Finally, a device-specific copy operation will be invoked, e.g., in NVIDIA GPU, the runtime API `cudaMemcpyAsync` is responsible for this operation.

5.2 System Architecture

Figure 5 shows the system architecture of Capuchin. To monitor tensor accesses, Capuchin adds extra fields in `Tensor` structure as shown in Listing 1. The `tensor_id` is a unique ID of a tensor to locate the same tensor across multiple iterations since a tensor's underlying memory address could be different across multiple iterations. Thus, `tensor_id` can ensure that certain memory optimizations be applied to the

correct tensor in the following iterations. The access_count and timestamp indicate the number of times the tensor has been accessed and the timestamp of the most recent tensor access. The pair of {tensor_id, access_count} can specify a specific tensor access that may trigger a memory optimization policy. There are five status in Tensor's status field: IN, SWAPPING_OUT, OUT, SWAPPING_IN and RECOMPUTE. For the evicted tensor for future recomputation, the tensor's memory is simply released, so only three status are required, i.e., IN, OUT and RECOMPUTE. Based on the structure, Capuchin can check the tensor's status at runtime to adjust the memory management policy, which is described in Section 4. The inputs and operation_name together build the lineage of tensor, which is used for recomputation.

Capuchin is composed of two modules: Tensor Access Tracker (TAT) and Policy Maker (PM). TAT interacts with Executor, Tensor, and Allocator in deep learning frameworks to track tensor accesses and carry out memory management operation related to a specific tensor access. PM is responsible for making memory management policy decision according to the tensor accesses information provided by TAT.

```
class Tensor{
  string tensor_id;
  int access_count;
  int timestamp;
  int status;
  // for recomputation
  vector<Tensor*> inputs; // input tensors
  string operation_name; // the op that produced
   this tensor
  ...
}
```

Listing 1. Tensor structure

Tensor Access Tracker (TAT) The TAT maintains a tensor access list in which each element includes {tensor_id, access_count, timestamp}. When a tensor is generated (its first access), the access count is initialized to one and the timestamp for the current time is stored in timestamp. Each time a tensor is accessed, these three values will be recorded and inserted into the tensor access list.

The TAT is designed for two purposes. First, it supports on-demand memory swapping to overcome Out of Memory (OOM) and Access Failure in measured execution. When they happen, the system will trigger tensor eviction and intercept the tensor accesses with null pointer. Second, it tracks tensor access pattern so that the PM can make and dynamically adjust the memory optimization decisions based on the complete tensor accesses sequence of one iteration.

To support on-demand swapping, TAT works in passive mode as shown in Figure 6. When an OOM is detected, TAT will look for one or multiple tensors with an approximate size to be evicted in tensor access list from the beginning until the current allocation is successful. Then, the evicted tensors' memory will be copied to CPU memory synchronously and the {tensor_id, cpu_addr} pair will be saved in TAT. After evicting a tensor, access failure will occur when this tensor is accessed again. At this point, TAT will provide this tensor's corresponding CPU address and copy the data to GPU.

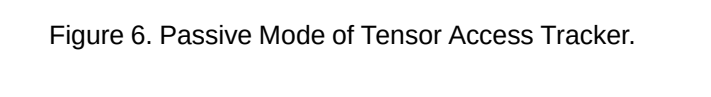


Figure 6. Passive Mode of Tensor Access Tracker.

With tensor access information in passive mode during measured execution, the main problem is to identify the tensor accesses boundary between complete iterations. In some frameworks such as TensorFlow, this information is directly known since the iteration number is indicated as the step ID. For others such as PyTorch, there is no explicit ID for each iteration but the boundary can be still identified when the same tensor is generated twice. Since the tensor access time in passive mode includes the on-demand swapping time, we need to subtract this time from tensor access time to obtain the actual time point that tensor access happens, assuming a hypothetical infinite GPU memory. Although a lock is required to maintain the integrity of tensor access sequence, the calls in TAT are all asynchronous and are not in the critical path. The low overhead is demonstrated in Section 6.3.2.

PM determines memory optimization policy based on tensor access sequence. The specific tensor access tracked by TAT will trigger the corresponding memory optimization operations, which are executed in Allocator or Executor depending on swapping or recomputation.

5.3 Optimizations

Decoupled computation and swapping. In prior works [22, 24, 31], each time a tensor has been swapped out, the next computation can only start when current computation that uses this tensor and this tensor's swapping-out both finish. It is because the computation and data transfer are coupled, i.e., the computation needs to synchronization the data transfer at a tensor's swapping-out. This is shown as the left of Figure 7. Such synchronization introduces non-trivial overhead when current computation is not enough to overlap the swapping-out. However, all the next computations, including current computation that uses this tensor can be performed concurrently with tensor swapping-out. In Figure 7, it means that the forward computation FWD2 and FWD3 can overlap with SwapOut1. To leverage this opportunity, we decouple the computation and data transfer at a tensor's swapping-out and only synchronize the earliest unfinished swapping-out when OOM occurs. In this way, the overhead of tensors' swapping-out can be significantly reduced.

Figure 7. Decoupled Computation and Swapping.

Collective recomputation. It is a mechanism for storing multiple recomputation target tensors with one recomputation. Take the lineage of $T_1 ! T_2 ! T_3 ! T_4$ as an example (T_2 to T_4 are recomputation targets). After releasing these three tensors' memory in forward propagation, T_4 will be first recomputed in backward propagation. At this point, T_2 and T_3 have not been triggered to start recomputation they are not in GPU memory when starting T_4's recomputation. Therefore, T_4's recomputation will start from T_1 (assume T_1 is in GPU memory now) and generate T_2, T_3 during this process. Given that the T_2 and T_3 will be needed next, we can consider storing them during the recomputation of T_4. This is essentially a trade-off between memory saving and recomputation efficiency. The more tensors we keep, the high memory pressure will be, but the recomputation complexity will be reduced from $O(n^2)$ to $O(n)$ in the best scenario.

However, it is difficult to make the decision before execution as we do not know how much free memory is available to keep multiple recomputation targets. Therefore, we apply the collective recomputation optimization dynamically at runtime. In the previous example, during the recomputation of T_4, we will keep T_2 once it is calculated. When proceeding to compute T_4, T_2 will be still kept if the memory is enough; otherwise, its memory will be released. In this way, we can retain the latest recomputation tensors as much as we can in one recomputation procedure.

5.4 GPU-specific Issues

Access time profiling. We need the access time of tensors to evaluate the efficacy of swapping and recomputation. In Tensorflow, the CPU processing is parallel with the GPU computations in order to saturate the GPU as much as possible. This means that the computing function returns immediately after enqueuing the kernel into the GPU stream. If we record the time before and after this computation, we only get the time in CPU process, but not the real GPU processing time. To resolve the issue, we measure it through CUDA Profiling Tools Interface (CUPTI), which is also implemented in Tensorflow (enable `run_metadata` in session). The small overhead incurred is acceptable since it is only done once.

Asynchronous and delayed operation. For a kernel to run on GPU, CPU just enqueues it into GPU stream. Accordingly, Capuchin's functions should not execute immediately either, instead they should be delayed to the proper point in the stream for execution. For example, when a swap-out operation is triggered by a specific tensor access, it should not perform swapping at that time, but wait until the GPU reach the corresponding point. Moreover, the memory optimization operations should be executed asynchronously to avoid block execution in the critical path. Note that recomputation is just like the normal computation, so it is naturally an asynchronous and delayed operation. Thus, we only need to care about the swapping in Tensorflow. Capuchin achieves it via leveraging the CUDA stream and CUDA event. By inserting the CUDA event into the CUDA stream when necessary and monitoring the CUDA event's status, Capuchin only executes the function when the corresponding CUDA event is accomplished. We implement swapping using the asynchronous memory copy API, i.e., `cudaMemcpyAsync()`, and two separate CUDA streams to perform swapping out/in respectively.

6 Evaluation

6.1 Methodology

Experimental setup. Our experiments were conducted on server equipped with NVIDIA Tesla P100 GPU, dual 2.60GHz Intel Xeon CPU E5-2680 v4 processors with 28 logical cores, 256 GB RAM, PCIe 3.0 16, running Ubuntu 16.04. The CUDA Toolkit version is 9.0 and cuDNN is 7.3.1. Tensorflow's version is 1.11 where Capuchin is also modified based on this version.

Workloads. We evaluate Capuchin on 7 state-of-the-art deep learning workloads as shown in Tabel 1. We use synthetic data for these CNNs instead of ImageNet [6] in order to better reveal the effect of memory optimization because the data pre-processing can help the training to be bounded in CPU [35] not fully demonstrating the benefits of our approach. BERT [9] is proposed by Google AI Language and achieved state-of-the-art results in a wide variety of NLP tasks. The base version of BERT includes 768 hidden layers of 110 million parameters.

	Architecture	Dataset	Run-Mode
VGG16 [27]	CNN	synthetic	graph
ResNet-50 [11]	CNN	synthetic	graph&eager
ResNet-152 [11]	CNN	synthetic	graph
InceptionV3 [30]	CNN	synthetic	graph
InceptionV4 [30]	CNN	synthetic	graph
DenseNet [12]	CNN	synthetic	eager
BERT [9]	Transformer	Gutenberg [19]	graph

Table 1. Workloads.

Baselines. The first baseline is TF-ori, i.e., the Tensorflow original version[1]. We also compare Capuchin with other state-of-the-art frameworks, which are all base on computation

[1]Tensorflow already used memory sharing and in-place operation optimization techniques in static memory allocation strategy framework, like

graph. No existing memory optimizations can be applied to eager mode, so we only compare Capuchin with the original eager mode of Tensor ow.

In graph mode, the second baseline is vDNN [24], which o oads the appropriate memory and uses a static prefetching strategy to overlap data transfer and computation time. We implemented the o oading and prefetching strategy of vDNN in Tensor ow using its pre-de ned operator for swap. As vDNN is designed for CNNs, it is not available on BERT. The third baseline is OpenAI's gradient-checkpointing [1], which is a re-implementation of recomputation strategy of Chen et al.'s recomputation [5] on Tensor ow. It includes two modes, memory and speed. The memory mode, which uses a heuristic to automatically select a set of nodes to checkpoint, aims to achieve $O(\sqrt{n})$ memory usage. The heuristic works by automatically identifying articulation points in the graph, i.e., tensors which split the graph into two disconnected parts when removed, and then checkpointing a suitable number of these tensors. The speed mode tries to maximize training speed by checkpointing the outputs of all operations that are typically expensive to compute, namely convolutions and matrix multiplies. Thus, this mode does not work for other neural networks except for CNNs.

6.2 Breakdown Analysis

We evaluate Capuchin mechanisms in swapping and recomputation respectively.

Swap. We compare the training speed to vDNN by only enabling swapping on Capuchin, and present the performance improvement of each mechanism. We conduct experiment on InceptionV3 and the results are shown in Figure 8(a). Here, DS (delayed synchronization) refers to decoupled computation and swapping optimization; ATP (Access Time-based Pro ling) refers to enabling the measured execution; FA refers feedback-driven adjustment of in-trigger time. Note that neither the architecture of the neural network nor its computation graph is known by Capuchin, hence we cannot perform layer-wise synchronization at swapping out/in, and delayed synchronization must be enabled.

The maximum batch size that vDNN can achieve is 400 on InceptionV3. With this batch size, the evaluation shows that Capuchin only improves the training speed by 5.5%. We found that the total memory needs to be evicted is about 25 GB, the swapping out/in time are 1.97s and 2.60s respectively (the bandwidth of device to host is a little bit faster than host to device). As a result, the computation time which can overlap the data transfer is just about 2.0s. However, the total data transfer time is more than twice as much as computation time, so the computation time is far from enough to overlap this data transfer. Therefore, the improvement due to swapping is very limited under the large batch size. With a relatively smaller batch size of 200, the breakdown analysis shows that ATP+DS improves the training speed by 73.9%, and FA improves further by 21.9% on the basis of ATP+DS.

(a) Swap on InceptionV3. (b) Recomputation on ResNet-50.

Figure 8. Breakdown Analysis.

Recompute. We compare the training speed with two modes of OpenAI's recomputation by only enabling recomputation in Capuchin, and present the performance improvement of each mechanism separately. We conduct experiment on ResNet-50 and the result is shown in Figure 8(b).

The memory mode of OpenAI's recomputation is denoted as OpenAI-M while the speed mode is denoted as OpenAI-S. The x-axis represents the maximum batch size that OpenAI-S and OpenAI-M can achieve respectively, which is 300 and 540. We only enable ATP, then enable Collective Recomputation (CR) to evaluate Capuchin's recomputation under these two batch sizes. We can see that the training speed of OpenAI-S is even lower than OpenAI-M about 8.3%, which also demonstrate that it is not appropriate to choose memory optimization targets according to layer type. At batch size of 300, the training speed remains the same when enabling ATP alone and enabling both ATP and CR. This is because there is only one target tensor in all recomputation procedure, thus collective recomputation does not provide improvement. Therefore, all the improvements are due to ATP and it delivers 37.9% performance improvement compared to OpenAI-S. At batch size of 540, Capuchin outperforms the OpenAI-M by 17.8% in which the ATP contributes 10.7% performance improvement and CR contributes another 7.1%.

6.3 Evaluation on Graph Mode

6.3.1 Memory Footprint Reduction

We use the batch size to represent the degree of memory footprint reduction. As there are two modes of OpenAI, we choose the bigger one as the representation.

Table 2 presents the maximum batch size that original Tensor ow, vDNN, OpenAI and Capuchin can achieve. We observe that Capuchin always achieves the maximum batch size. Compared to original Tensor ow, Capuchin promotes the maximum batch size by up to 9.2 for ResNet-50 and 5.49 in average. Especially for the BERT, Capuchin achieves 7 batch size improvement. Compared to the second best

MxNet, since dynamic allocation will automatically release memory which is no longer needed, and in-place operation is already been integrated into implementation of algorithms.

Models	TF-ori	vDNN	OpenAI	Capuchin
Vgg16	228	272	260	350
ResNet-50	190	520	540	1014
ResNet-152	86	330	440	798
InceptionV3	160	400	400	716
InceptionV4	88	220	220	468
BERT	64	-	210	450

Table 2. Maximum Batch Size in Graph Mode.

(vDNN shows the best on Vgg16 while OpenAI wins on the others), Capuchin can still achieve batch size promotion up to 2.14 for BERT and 1.84 in average. This is due to Capuchin's tensor-wise memory management which can extract all the memory optimization opportunities. The improvement on Vgg16 is the minimum across all workloads as Vgg16 has the fewest layers in which several layers need enormous memory to compute, e.g., its first ReLU layer needs approximately 6 GB. Such rigid memory footprint requirement cannot be optimized by swapping or recomputation.

6.3.2 Performance

In this section, we evaluate the performance of Capuchin against original Tensorflow, vDNN and OpenAI. We choose the better performance to represent OpenAI between the memory and speed mode.

Runtime overhead. First, we measure the runtime overhead of Capuchin due to runtime information tracking. We run the batch size which original Tensorflow can accommodate, measuring the training speed averaged over 20 iterations. The result is presented in Figure 9. The first two (three on Vgg16) batch sizes are original Tensorflow can run. At the maximum batch size in Tensorflow, the overhead Capuchin introduced among all workloads are less than 1%, and the average is 0.36%. With a relatively small batch size, the maximum overhead is 1.6% for ResNet-152 and the average is 0.9%. This means the overhead introduced by Capuchin is negligible when GPU computation is heavy.

Performance comparison. As Capuchin will refine the policy at runtime, we will ignore those iterations and only measure the performance until the policy is stable (usually within 50 iterations). As for the performance baseline, we use the training speed under the maximum batch size that original Tensorflow can achieve. But there is an exception for Vgg16, the performance degrades by 25% when increasing batch from 208 to 228, which is shown as Figure 9(a). It is caused by some convolution layers falling back to a slower convolution algorithm due to memory limit. Thus, we use the performance at batch size of 208 as Vgg16's baseline.

Figure 9 summarizes the performance of Capuchin compared to the baselines. Capuchin consistently demonstrates the best performance across all neural networks, the second best is OpenAI while vDNN is the worst. The performance of vDNN and OpenAI nearly remain the same under different batch size due to their static memory optimization strategy. For the CNN workloads, vDNN exhibits extremely performance loss on the ResNet networks, which is up to 74.4% for ResNet-152 and 70.0% for ResNet-50. This extremely performance loss results from that the synchronization overhead at layer-wise is huge on a powerful P100 GPU as the computation time is so fast that cannot overlap the data transfer time. The speed mode of OpenAI exhibits better performance on Inception networks at a smaller batch size while the memory mode always shows better performance on the other neural networks. Therefore, we can clearly see a performance drop down of OpenAI on InceptionV3/4 at OpenAI's maximum batch size. Compared to vDNN and OpenAI, Capuchin can achieve by up to 3.86 and 1.55 performance speedup respectively, in which the 1.55 speed up is on BERT.

Overall, Capuchin can achieve averaged 75% of each workloads' maximum batch size in Section 6.3.1 where the performance loss is within 26%. We can clearly see that the performance of Capuchin has slowly deteriorated along with the increasing batch size. When batch size only increases by 20% compared to TF-ori's maximum batch size, Capuchin's policy only consists of swapping candidates which delivers within 3% performance degradation across all workloads. As it grows further, the policy is mixed by swapping and recomputation. However, we observe that Capuchin, instead of showing performance degradation, results in some performance improvement on Vgg16 and BERT, which is shown as Figure 9(a) and Figure 9(f). On Vgg16, it is because that some convolution layers fall back to slower convolution algorithms due to memory limit at a larger batch size. On the other side, Capuchin releases a lot of memory in forward propagation which gets more free memory to opt for faster convolution algorithms. For BERT, we observe that its GPU utilization goes up from 31.7% to 37.2% at batch size of 48 $\sim$ 64. Further, we found that the GPU utilization is 73.7% at batch size of 200 in Capuchin. Therefore, this performance improvement results from more computation saturating the GPU.

6.4 Evaluation on Eager Mode

6.4.1 Memory Footprint Reduction

Table 3 shows the maximum batch size which TF-ori and Capuchin can be reached in eager mode. The maximum batch size of ResNet-50 is 122 and 190 in eager and graph mode respectively.

This is due to the lack of some optimization techniques in eager mode, e.g., pruning and constant folding, which can only be applied in graph mode. In summary, Capuchin achieves 2.46 and 2.71 batch size increment on ResNet-50 and DenseNet respectively.

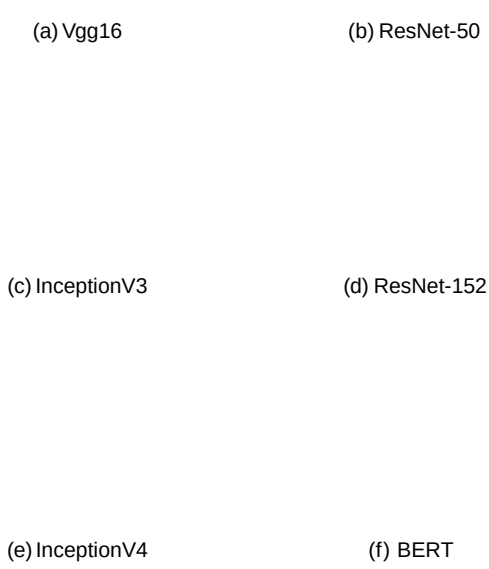
Models	Tensor ow	Capuchin
ResNet-50	122	300
DenseNet	70	190

Table 3. Maximum Batch Size in Eager Mode.

6.4.2 Performance

Runtime overhead. Capuchin introduces 1.5% and 2.5% runtime overhead on ResNet-50 and DenseNet respectively. This overhead is a little higher than in graph mode since the processing of operations are sequential in eager mode while there are lots of nodes can be processed parallel in graph mode. Thus the frequent lock/unlock will stall the execution.

Performance comparison. The performance overhead is 23.1% in ResNet-50 when Capuchin promotes the batch size by 83.6% shown as Figure 10(a). The degradation is a bit larger than in graph mode since eager mode is not that e cient as graph mode. On the other side, we can see that in Figure 10(b), DenseNet exhibits a performance promotion when increasing the batch size. This is analogous to BERT. Through pro ling, we observe that the GPU utilization increase from 41.8% to 45.3% at batch size of 670. The performance improvement from increasing GPU utilization overshadows the overhead introduced by recomputation.



(a) Vgg16 (b) ResNet-50

(c) InceptionV3 (d) ResNet-152

(e) InceptionV4 (f) BERT

Figure 9. Performance in Graph Mode.

(a) ResNet50 (b) DenseNet

Figure 10. Performance in Eager Mode.

7 Related Work

Deep learning framework support. Data parallelism has been widely adopted in current major deep learning frameworks. Each GPU has its own network replica. Although it can reduce GPU memory footprint by decreasing batch size per GPU, recent research [34] has shown that the training pipeline will bound in communication due to frequent model aggregation. Model parallelism splits the total neural network to multiple GPU and each GPU takes charge of its own part computation respectively. The trade-o of data and model parallelism is analyzed in [15, 28, 29, 32, 33]. This approach is orthogonal to Capuchin.

Computation graph dependent techniques. The main majority works perform memory optimization based on computation graph, including three categories, i.e., swapping, recomputation and compression, which are implemented by inserting corresponding operations to original computation graph before real running. vDNN [24], Superneurons [31] and [22] swap the data out to the CPU in forward phase and prefetch it at backward propagation. [16] proposes a intra-layer memory reuse and inter-layer data migration to reduce memory footprint. They tries to overlap this data transfer with computation time, which are all based on the apriori knowledge about characteristics of di erent layers of CNN, such as Conv layer is time-consuming while Pool, ReLU layer are cheap-computation. However, this characteristic depends on the device computing power, link speed of GPU-CPU lane and input size, which is not a static property. Therefore, when the computation is inadequate to overlap the data transfer, Chen et al. [5, 31] proposes a recomputation strategy to release cheap-computations' memory in forward phase and get it back via recompution. However, they also ignore the variation among the same type layers, thus cannot fully extract the memory reduction opportunity. CDMA [25] and Gist [13] compress the data by leveraging the sparsity in particular neural network architecture, such as ReLU-Pool, which are all lossless compressing methods. Gist [13] also introduces a lossy compressing strategy by reducing precision in forward phase and reverting it back at backward propagation. Nevertheless these works are at the algorithm level which are orthogonal to our work.

Computation graph agnostic techniques. [7, 21] tries to virtualize the GPU memory via leveraging host memory as the extended memory. However, these works are not sensitive to the characteristics of deep learning training, thus delivering a poor performance due to the large overhead of on-demand data transfer between GPU and CPU. [37] resorts to good memory swapping algorithm by pro ling the training process. However, these works are all not aware the information of computation, therefore, they lose a vital opportunity to reduce memory e ciently under a powerful GPU, i.e., by recomputation.

8 Conclusions

This paper proposes Capuchin, a tensor-based GPU memory management module that reduces the memory footprint via tensor eviction/prefetching and recomputation. The key feature of Capuchin is that it makes memory management decisions based on dynamic tensor access pattern tracked at runtime. This design is motivated by the observation that the access pattern to tensors is regular during training iterations. Based on the identi ed patterns, one can exploit the total memory optimization space and o er the ne-grain and exible control of when and how to perform memory optimization techniques. We deploy Capuchin in Tensor ow and show that Capuchin can reduce the memory footprint by up to 85% among 6 state-of-the-art DNNs compared to the original Tensor ow. Especially, for the NLP task BERT, the maximum batch size that Capuchin can outperforms Tensor ow and gradient-checkpointing by 7 and 2.1 , respectively. We also show that Capuchin outperforms vDNN and gradient-checkpointing by up to 286% and 55% under the same memory oversubscription.

Acknowledgments

We sincerely thank the anonymous reviewers for their constructive comments. We thank Xipeng Shen and Yongluan Zhou for their valuable comments. This work is supported by the National Key Research and Development Plan (No. 2017YFC0803700) and NSFC (No. 61772218 and No. 61832006). It is also partly supported by NSF grants CCF-1750656 and CCF1919289.

References

[1] Gradient-checkpointing. h ps://github.com/cybertronai/gradient-checkpointing

[2] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. TensorFlow: A System for Large-Scale Machine Learning. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16) (2016), vol. 16, USENIX Association, pp. 265 283.

[3] Agrawal, A., Modi, A. N., Passos, A., Lavoie, A., Agarwal, A., Shankar, A., Ganichev, I., Levenberg, J., Hong, M., Monga, R., et al. Tensor ow eager: A multi-stage, python-embedded dsl for machine learning. arXiv preprint arXiv:1903.01855 (2019).

[4] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A exible and e cient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274 (2015).

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

[6] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: E cient primitives for deep learning. arXiv preprint arXiv:1410.0759 (2014).

[7] Cui, H., Zhang, H., Ganger, G. R., Gibbons, P. B., and Xing, E. P. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In Proceedings of the Eleventh European Conference on Computer Systems (2016), ACM, p. 4.

[8] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on (2009), IEEE, pp. 248 255.

[9] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv preprint arXiv:1810.04805 (2018).

[10] Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd: Training imagenet in 1 hour. arXiv preprint arXiv:1706.02677 (2017).

[11] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (2016), pp. 770 778.

[12] Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. Densely connected convolutional networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (2017), pp. 4700 4708.

[13] Jain, A., Phanishayee, A., Mars, J., Tang, L., and Pekhimenko, G. Gist: E cient data encoding for deep neural network training. In 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA) (2018), IEEE, pp. 776 789.

[14] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Ca e: Convolutional architecture for fast feature embedding. In Proceedings of the 22nd ACM international conference on Multimedia (2014), ACM, pp. 675 678.

[15] Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. arXiv preprint arXiv:1807.05358 (2018).

[16] Jin, H., Liu, B., Jiang, W., Ma, Y., Shi, X., He, B., and Zhao, S. Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures. ACM Transactions on Architecture and Code Optimization (TACO) 15 (2018), 37.

[17] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-Datacenter Performance Analysis of a Tensor Processing Unit. In Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017 (2017), pp. 1 12.

[18] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014).