Zico: Efficient GPU Memory Sharing for Concurrent DNN Training

Gangmuk Lim, *UNIST;* Jeongseob Ahn, *Ajou University;* Wencong Xiao, *Alibaba Group;* Youngjin Kwon, *KAIST;* Myeongjae Jeon, *UNIST*

https://www.usenix.org/conference/atc21/presentation/lim

This paper is included in the Proceedings of the 2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the 2021 USENIX Annual Technical Conference is sponsored by USENIX.

Zico: Efficient GPU Memory Sharing for Concurrent DNN Training

Gangmuk Lim
UNIST

Jeongseob Ahn
Ajou University

Wencong Xiao
Alibaba Group

Youngjin Kwon
KAIST

Myeongjae Jeon
UNIST

Abstract

GPUs are the workhorse in modern server infrastructure fueling advances in a number of compute-intensive workloads such as deep neural network (DNN) training. Several recent works propose solutions on sharing GPU resources across multiple concurrent DNN training jobs, but none of them address rapidly increasing memory footprint introduced by such job co-locations, which greatly limit the effectiveness of sharing GPU resources. In this paper, we present Zico, the first DNN system that aims at reducing the system-wide memory consumption for concurrent training. Zico keeps track of the memory usage pattern of individual training job by monitoring its progress on GPU computations and makes memory reclaimed from the job globally sharable. Based on this memory management scheme, Zico automatically decides a strategy to share memory among concurrent jobs with minimum delay on training while not exceeding a given memory budget such as GPU memory capacity. Our evaluation shows that Zico outperforms existing GPU sharing approaches and delivers benefits over a variety of job co-location scenarios.

1 Introduction

Recent advances in deep neural networks (DNNs) have made tremendous progress on a wide range of applications, including object detection [24], language model [11, 47], translation [40], and speech recognition [27]. As a number of new DNN models are being explored, developers take advantage of hardware accelerators to train the models, such as TPU [22] and GPU, which is the most popular choice. GPUs are the workhorse in server infrastructure and yet becoming highly contended resources at the same time [20, 43].

To utilize expensive GPU resources, efficient GPU sharing mechanisms have become indispensable. Prior work focuses on either *temporally* multiplexing GPU in its entirety [26, 43, 44] or *spatially* sharing compute units [10]. The temporal sharing is a software mechanism that dedicates both compute cores and memory in GPU solely to single training for a time quantum (e.g., 1 minute). Despite the good flexibility, this approach often cannot efficiently utilize GPU resources. For example, most compute resources are left idle for common translation models such as GNMT [42] and language models such as RHN [47]. These training algorithms include a number of RNN modules [28], such as LSTM [12] and GRU [8] networks, which exhibit a small degree of data parallelism to GPU, causing under-utilization of GPU resources. As a different approach, the spatial sharing can provide better throughput than the temporal sharing as long as a single training job does not fully saturate GPU compute resources [44]. However, a limitation in applying the spatial sharing is the working set size of concurrent jobs, which grows substantially with the job co-location. If the working set does not fit in GPU memory, the system has to kill a job or swap GPU memory to the host, which overshadows the performance benefit of the spatial sharing. Therefore, to make the spatial sharing widely applicable, it is essential to reduce the memory footprint of co-located training jobs.

We observe that sharing intermediate data generated during co-located training jobs significantly reduces the total memory footprint. Training is a highly iterative procedure first navigating layers in order (forward pass) and then the same layers in *reverse* order (backward pass) for each batch of input data. During the training procedure, intermediate outputs from model layers called *feature maps* dominate memory footprint [18, 32]. Feature maps are generated in each layer during the forward pass and later consumed in the backward pass to update the layer. Due to the regular bi-directional execution, memory consumption in a single training job commonly exhibits a *cyclic pattern* — memory consumption gradually increases in the forward pass and then decreases in the backward pass. Thus, a simple yet effective strategy to save memory consumption is creating a large GPU memory pool and elastically sharing the memory pool for concurrent training jobs. To increase sharing opportunity, coordination of concurrent training jobs is needed to make them run different passes, e.g., forward pass for job A (increasing its working set) and backward pass for job B (decreasing its working set). This

approach results in memory allocations of a job to happen simultaneously with memory deallocations of the other job, efficiently reducing the *system-wide* memory footprint.

Despite that the sharing idea is plausible, the way today's DNN frameworks execute training on GPU poses significant challenges. Current frameworks are mainly designed for a *solo training* case. Following dataflow dependency, they allocate the memory required for each DNN kernel computation ahead of time and issue as many kernels as possible to the GPU stream, i.e., work queue per job for its GPU computations, in order to saturate the GPU's internal compute resources. This leads to GPU computations *asynchronous* and in parallel with CPU processing. Thus, the platforms are *unaware* of progress on the GPU computations and when memory is indeed consumed by GPU. Without proper handling of the asynchrony, shared memory does not guarantee correctness such as preventing memory corruptions in shared untapped memory of a waiting kernel by other training jobs.

In this paper, we propose Zico, a DNN platform that enables efficient memory sharing for concurrent training. Zico retrofits a widely used DNN platform, TensorFlow, to maximize the overall throughput of concurrent training. The goal of Zico is finding the best coordinated executions of concurrent training to fully utilize GPU computational and memory resources. To achieve the goal, (i) Zico accurately monitors computational progress of training jobs. Based on that, Zico allocates and deallocates memory for DNN kernels, informing memory usage patterns closer to GPU's view. (ii) Zico incorporates runtime information (e.g., iteration times, memory usage patterns, and GPU memory limit) and executes a job scheduler, called *scrooge scheduler*, to efficiently steer concurrent jobs to utilize the shared memory pool. (iii) Zico efficiently organizes the entire GPU space as an elastic shared memory pool to support scrooge scheduler.

To detect computational progress of asynchronous kernels, Zico leverages a specific kernel called CUDA event, which notifies progress of GPU kernels. Zico uses CUDA event to identify allocation and release time of memory used by a GPU kernel. Based on the information, Zico executes our novel scrooge scheduler to forecast the memory consumption trend of concurrent training at the iteration boundary and introduces the minimum stall time on each iteration. Nevertheless, the memory usage trend of the co-scheduled jobs varies according to how they interfere each other in the use of GPU compute units. To apply their dynamic behaviors, scrooge scheduler refines decisions based on feedback collected at runtime.

Zico organizes the memory pool as a collection of chunks called *regions* and separates their uses based on data characteristics. DNN training generates several types of data as tensors, categorized mainly as *ephemeral* tensors with high occurrences and *long-lived* tensors like feature maps which constitute the most memory footprint. By separating regions by type, Zico ensures that memory stored with feature maps does not interfere with other transient data while making their demands follow the cyclic pattern of training iteration. This design choice allows our scheduling decisions to be applied with little disruption without losing sharing opportunity.

Being prototyped on a popular DNN framework, TensorFlow, we evaluate Zico experimentally using six models ranging from translation to object detection on V100 and RTX 2080 Ti GPUs. The results show that Zico enables effective memory sharing over a wide range of memory consumption trends. For high memory footprint, Zico is up to 8.3x and 1.6x faster than traditional spatial sharing and temporal sharing approaches, respectively, especially when concurrently training non-identical models. Furthermore, for low memory footprint, where no stall on concurrent training is needed, Zico behaves similarly to traditional spatial sharing and is up to 1.6x faster than temporal sharing. Overall, Zico achieves speedups, regardless of whether concurrent training is based on the same or distinct models.

2 Background

2.1 Deep Neural Network Training

The training process typically relies on iterative optimization methods like stochastic gradient descent (SGD) [13], Momentum [39], and Adam [23]. In each *iteration*, *forward pass* (FP) is followed by *backward pass* (BP) on a *batch* of training dataset. During FP, by computing on the layer's input, weights, and bias, each layer outputs *feature maps* to be used as an input to the next downstream layer. At the end of FP, the last layer produces a loss representing the accuracy of the current model on the input batch. Using the loss value, BP computes the gradients by flowing the layers in reverse order and aggregates the gradient values to update model *parameters* (i.e., layers' *weights* and *bias*). On finishing BP, the training repeats FP and BP on the next batch. As the batch size is usually fixed, the computation load and the memory usage characteristic are usually very similar across iterations [15, 43].

It is widely known that model parameters occupy only a small fraction of memory, and the majority is consumed to store feature maps generated in the FP computation [18, 32, 43]. BP needs feature maps to calculate the gradients at each layer. Hence, unless recomputed [6, 14, 19], feature maps are usually kept on memory for a long time until they are no longer accessed in BP. The amount of memory consumption is determined by several factors, such as the number of layers, layer size and type, input batch size, etc. There is also other intermediate data training iteration creates, e.g., *gradient maps* that represent the output of each layer during BP, local data local in each kernel, etc. They are *all* ephemeral as memory is released soon after its allocation [18, 32]. For brevity, we assume all memory allocations in DNN training are based on *tensors*.

2.2 GPU Sharing Use Cases

Users run training either on shared GPU clusters or on dedicated servers. For both cases, GPU sharing is becoming a fundamental technique to better utilize GPU resources. In this subsection, we introduce two specific scenarios that can take advantage of sharing GPUs.

Hyperparameter tuning (inter-job). With the increasing popularity of applications fueled by DNN, a number of new models are being developed by DNN practitioners every day. A model for developing exposes many high-level properties, e.g., learning rate and momentum, as hyperparameters that need to be optimized. This task is known as *hyperparameter tuning* [3]. As hyperparameters constitute a large search space, there are several popular tools such as Hyperdrive [35] and HyperOpt [4] that automate hyperparameter optimization and construct a new model with the best (or desired) quality for users. These tools usually generate a large number of closely related training jobs (as much as 100s [26, 43]) that explore a different set of hyperparameters for the same reference model. Nevertheless, spatial GPU sharing has greater performance potential for this workload, as discussed in Section 7.

Hyperparameter tuning jobs dominate training workloads run atop shared GPU clusters [20, 26, 43]. To get them timely done on heavily contended GPUs, prior works propose several techniques such as temporal and spatiotemporal sharing to apportion a single GPU over multiple training jobs [43, 44].

Gradient accumulation (intra-job). Gradient accumulation is a promising method to speed up model convergence when hyperparameters other than batch size are stabilized. It runs a set of consecutive mini-batches and accumulates the gradients of those mini-batches before updating model parameters. The essential goal is to give an illusion of training on a large batch that better improves convergence without oversubscribing GPU memory by using small mini-batches. A common practice has been to process these mini-batches sequentially.

Nonetheless, efficient spatial GPU sharing can offer sharing incentive to concurrent training on mini-batches. One might wonder whether spatial sharing on this training is indeed plausible. Based on our observation, translation or speech recognition models (e.g., GNMT [42]) often underutilize GPU compute resources, making it beneficial to share GPU computes. On top of it, our system supports highly effective sharing of GPU memory, enabling in concurrent training each to use a mini-batch size slightly smaller, if not the same, than the original mini-batch size. Altogether, our system opens up a new opportunity to speed up training for gradient accumulation, which we will discuss in Section 7.

2.3 Spatial GPU Sharing

NVIDIA has developed Multi-Process Service (MPS) [10], an alternative way to share GPU among multiple CUDA processes. With MPS, NVIDIA V100 GPU supports up to 48 processes to run concurrently on a single GPU, with each process assigned with separate GPU compute resources, i.e., SMs [10]. In NVIDIA A100 [29], a newer generation, GPU sharing architecture further partitions HW paths in the memory system, e.g., memory controllers and address busses, to prevent the concurrent processes from interfering with each other when demands for memory bandwidth are high. NVIDA's GPU sharing mechanisms have two commonalities. First, they are mainly designed for sharing "compute resources" spatially. Second, they attribute GPU sharing to demands for protection among untrusted users requiring strong isolation.

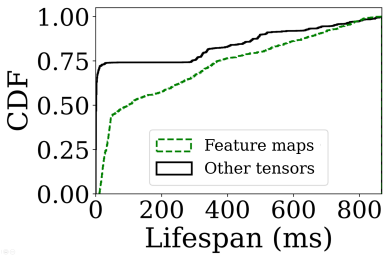


Figure 1: Cumulative distribution of NASNet tensor lifespan.

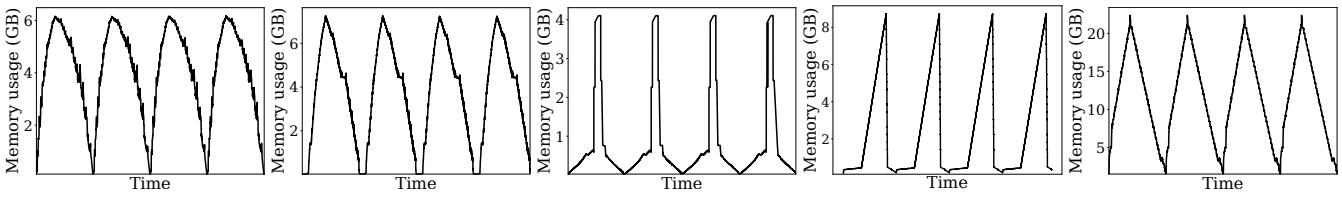
Since not all use cases require strong isolation among training jobs, e.g., hyperparameter tuning driven by a single user [26], recent work supports a spatial GPU sharing similar to MPS in a single process domain [44]. Regardless of protection level, the underlying mechanism enabling spatial sharing within GPU is very similar, if not the same — and so is the resulting performance.

3 Challenges for Memory Sharing

GPU has a limited amount of HW resources, requiring it to be used in high efficiency. As GPU's compute and memory resources are shared to run concurrent training limiting per-training resource capacity, it is crucial to thoroughly understand the current practices in DNN frameworks and uncover challenges for spatial GPU training. In this section, we discuss three major challenges to address in Zico.

3.1 Memory Bloating

Major DNN frameworks [1, 5, 31] typically maintain feature maps in memory until they are no longer accessed. As discussed earlier, feature maps have a relatively longer lifespan between the first access and the last access, making the most of in-use memory consumed to store the feature maps. Figure 1 compares cumulative distributions of lifespans for feature maps and other data in NASNet training. As the figure shows, feature maps exhibit longer lifespans with 134ms on average and 234ms as median value as opposed to 18ms on average and 2ms as median value in the other data. We further investigate cumulative distributions of tensor sizes of the two data types, showing that feature maps are larger. In consequence, as shown in Figure 2, feature maps lead to the peak



(a) ResNet-50 (b) NASNet (c) GNMT (d) RHN (e) BERT

Figure 2: Memory usage patterns for different DNN models over time.

memory consumption substantially higher than the minimum that corresponds to the model size in each iteration. We call this issue *memory bloating*.

Traditional spatial GPU sharing mechanisms are vulnerable to memory bloating. DNN frameworks like TensorFlow do not tend to be designed for memory sharing with internal memory manager maintaining a *local pool* of memory for single training. Thus, no memory release to GPU happens. This results in peak memory usages from concurrent training, all adding up and consequently saturating GPU memory even in the modest memory footprint for individual training. To illustrate, let us consider concurrent training of two identical models, with each demanding more than half of GPU memory as an example. In this case, the memory demand across the two local memory pools exceeds GPU memory capacity, beginning to take advantage of CPU-side memory as a swap space. To facilitate this, recent NVIDIA GPUs, including V100, provide a feature known as Unified Virtual Memory (UVM), which is transparent to DNN platforms.

We found that using UVM for DNN training is currently costly and severely affects overall performance despite great flexibility. To confirm the effect, we compare throughput between solo training versus concurrent training for ResNet-50 using NVIDIA V100 when a training job occupies 70% GPU memory. To make the comparison fair, we configure a single training job to use 50% GPU resources set by MPS. There is a dramatic throughput degradation in concurrent training (i.e., 8 times slower) as it suffers from GPU memory oversubscription. Therefore, we should decrease the risk of GPU memory being used up during concurrent training.

3.2 Workload Variability

As an additional challenge, we explain a workload characteristic that makes memory optimization for spatial GPU sharing fundamentally complicated. Although all models follow a cyclic pattern in memory usage, as shown in Figure 2, memory usage patterns are inherently different across models. For example, ResNet-50 has a beefy shape in which memory bloating appears for a fairly large time duration. In contrast, GNMT has a lean shape in which peak memory appears for a short period and quickly disappears. Therefore, such variability must be taken into account in designing a scheduling policy for concurrent training.

Nonetheless, we take advantage of an observation that a similar memory usage pattern repeats over iterations for a single model. DNN frameworks require that computation kernels be ordered in a specific way, e.g., following topological sort, thus keeping the corresponding memory operations ordered across iterations [32]. So, we believe this determinism is prevalent in most of the training tasks.

3.3 Asynchrony with GPU Processing

CPU processing in DNN frameworks is in parallel with GPU computations. Before issuing a kernel to a GPU stream, the memory manager in the platform allocates tensors required for the kernel computation. After issuing the kernel and returning immediately, CPU processing brings back its control and can do any subsequent task asynchronously with GPU computations. Meanwhile, GPU may or may not execute the issued kernel depending on whether earlier kernels are still pending or not.

Driven by this GPU-specific property, DNN frameworks produce a static schedule of DNN kernels mainly customized for single training, in which kernel operations are *ordered* based on dependencies every iteration. DNN frameworks usually allocate the memory required for kernel operations in sequence *ahead of time* while issuing *as many kernels as* possible to the stream to saturate GPU. This way of exercising GPU for single training has been common because there is no concern regarding correctness in memory allocations. Specifically, suppose we use a released tensor memory from a kernel to allocate it for the next kernel, even though the earlier kernel is not completed by GPU yet. In that case, memory corruption does not occur since kernels in the GPU stream are processed sequentially.

However, it is critical to make the CPU process keep track of GPU memory usage trends precisely to enable memory sharing for concurrent training. To illustrate the current limitation concretely, let us show effective memory observed in TensorFlow during training ResNet-50 in Figure 3. In the figure, training exhibits a cyclic pattern for a short period followed by a long pause due to kernels pending on GPU during an iteration. Unlike the CPU's view, the actual memory usage trend from GPU's view appears in Figure 2(a), which perfectly follows a continuous cycling pattern.

In summary, for concurrent training with multiple GPU streams, today's approaches that make memory sharing decisions based on CPU's view are vulnerable to memory cor-

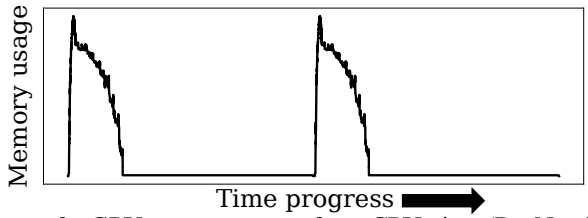


Figure 3: GPU memory usage from CPU view (ResNet-50).

ruption led by simultaneous accesses on the same memory address. This issue is the last challenge to address.

4 Design Overview

Zico aims at providing efficient spatial GPU sharing by enabling coordinated job scheduling and GPU memory management for concurrent training. As a system currently built on TensorFlow, the framework keeps tracking the lifespans of memory used in each training and produces a schedule of concurrent training executions to avoid GPU memory oversubscription. We seek to achieve the following goals in the design of Zico:

• **High performance.** A single iteration runs for a short time duration from tens to hundreds of milliseconds. Thus, a modest overhead with memory sharing in each iteration can manifest as a long delay in the entire training. We should attempt to minimize such overhead and achieve high performance.

• **Wide model support.** Section 3.2 demonstrates a wide range of patterns in memory demand across models during training. Our memory sharing strategy should be general, not restricted to specific memory patterns. The concurrent training may not even expose similar or the identical models.

• **Common approach.** Our approaches are mainly compatible with DNN platforms that express a job execution as a computation graph ahead of time, e.g., TensorFlow [1], Caffe [21], and MXNet [5]. The other platforms are based on imperative programming and train a model without constructing a computation graph, namely, the eager mode. The memory-aware scheduler in Zico makes decisions based only on observed memory demands at runtime, making it also applicable to the eager mode.

Zico has two key system modules as shown in Figure 4: (i) *scrooge scheduler*, a processing unit that orchestrates executions of concurrent training driven by memory demand patterns; and (ii) *memory manager*, a unified memory allocator that handles both inter- and intra-training memory requests.

Scrooge scheduler. Zico monitors GPU progress to capture precise memory usage in GPU for each training over time. On observing memory usage, Zico schedules concurrent jobs to achieve the best possible performance in training while putting the total memory consumption within a predefined budget, which does not exceed GPU memory capacity. This is a sophisticated problem to which a naive strategy rarely

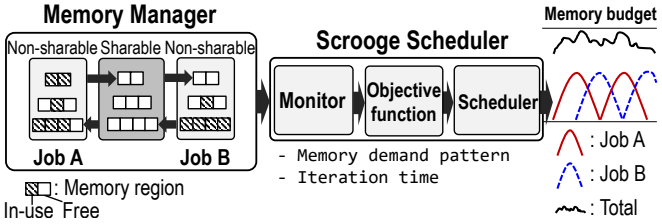


Figure 4: System architecture in Zico.

works. When memory is sufficient for running concurrent training without sharing, no coordination among jobs might be necessary. In contrast, when memory is tight, we need a schedule of concurrent training in order not to exceed the given memory budget. A simple approach towards saving memory consumption can be coordinating training jobs in such a way that forward pass execution (i.e., memory allocation phase) is overlapped with backward pass execution (i.e., memory deallocation phase) as much as possible. However, when memory demand patterns are beefy such as in Figure 2(a), the system-wide memory usage can blow up because the memory demand in the forward pass grows fast while the backward pass shrinks its memory demand relatively slowly.

To provide concrete guidance for memory sharing, we devise *scrooge scheduler* that facilitates an objective function helping forecast the system-wide memory demand in the future. Before starting a new iteration, the scheduler takes into account the lifespans of in-use memory regions, which are sharable memory units in Zico. Then, the scheduler predicts whether allowing the iteration immediately will consume memory less than the memory budget. If this is nearly impossible, the scheduler estimates the minimum stall time to be applied on the new iteration so that memory allocations in the forward pass to be issued later are safely fulfilled when more memory is available. It is essential to maintain precise region lifespan in order to make a correct decision in the scheduling. To meet this goal, Zico iteratively refines region lifespan based on feedback collected from prior iterations at runtime.

Scrooge scheduler is agnostic to programming model in DNN platforms and only relies on information on memory demand patterns. For this reason, Zico is able to perform spatial memory sharing as long as memory requests are appropriately clustered on regions and their lifespans can be estimated. As explained next, we facilitate tensor types to constitute regions for sharing, but many different ways are indeed possible — e.g., classification based on tensor access time intervals in eager mode [2].

Memory manager. Zico organizes the entire GPU memory space into a collection of *regions*, which is a contiguous memory space that stores a set of tensors in the same type. Using regions is a natural choice for us to mitigate the sharing overhead and to keep memory stored with feature maps not contended with other tensor data. Using regions further assists job scheduling decision to be made promptly. Scrooge scheduler makes use of memory demand pattern that changes dynami-

cally within an iteration. If we consider memory changes at the granularity of tensor, we may need to investigate too many time points along the way, putting a strain on the scheduler. Details will be discussed in Section 5.

The memory manager separates memory space into two areas, *sharable* and *non-sharable*. The sharable area represents currently unused memory that can be granted to any in-flight training in need of more space (mainly for feature maps), whereas the non-sharable area constitutes job-local memory pools. Zico analyzes the computational graph and extracts type information on tensors at runtime. During training iterations, allocation requests for feature map tensors are always served from their own regions first in the local memory pool. If regions in the local memory pool are used up, the memory manager assigns a new region from the sharable area. In general, feature maps demand the most regions in the system and these regions are mainly shared across concurrent training jobs.

The current design of Zico limits two training jobs to share a common memory area since many models we observe, including models in Figure 2, exhibit rather beefy memory demand patterns or high GPU utilization. For co-locating more than two jobs, a feasible approach is to organize the jobs into a group of pairs and schedule each pair independently with its own memory budget. This is a natural extension to Zico, so we leave it as a future work.

Protection level. We provide Zico as a single framework instance mainly for performance reason. Existing multi-process solutions such as MPS do not promise good performance for elastic memory space sharing across different processes. For example, to grant memory across two MPS processes, we require invoking CUDA APIs such as `cudaMalloc` and `cudaFree` quite frequently at each training iteration. Among these API, `cudaFree` is known to stall GPU's pipelined execution upon invoked [9], making itself harmful if recklessly used. Our measurement also reveals that a single ResNet-50 training that allocates memory using these APIs becomes 7x slower than the training that allocates memory locally.

Note that key scenarios discussed in Section 2.2 are enough to train models in the same protection domain. Apart from it, we design Zico to be useful for a variety of scenarios as long as isolation is not a primary concern, e.g., the same tenancy with trusting users in a shared GPU cluster.

5 Scheduling Algorithm

In this section, we formalize the scheduling problem for concurrent training and introduce the *scrooge scheduler*. All related implementation details on how to obtain memory usage patterns are explained in Section 6.

5.1 Problem Definition

We make use of memory consumption at the region level to shape the memory pattern of a job. Despite the coarser-grained leveling of memory utilized by tensors, using regions still provides meaningful information to compute memory sharing potential. Since regions allocated for feature maps are the main target for sharing, in the problem formulation, we assume all regions for a job have the lifetime that spans the forward-backward passes for a single iteration.

Formally, we denote M regions required for an iteration of a training job as $\{R_i : 1 \leq i \leq M\}$ following the *allocation order*, with region R_i having two parameters to indicate its *lifespan*: $Time(R_i(a))$ for the allocation time and $Time(R_i(d))$ for the deallocation time. Assume at time moment T that there are K active regions in the system that indicate the sum of memory footprint of the concurrent jobs. To achieve efficient GPU memory provisioning, we need to *minimize the time delay* to be applied on each training iteration without overcommitting the system-wide memory budget C. This objective turns into the following formulation:

$$\underset{TimeShift}{\text{argmin}} \ (Time(R_1(d)) - Time(R_1(a)) + TimeShift) \quad (1)$$

subject to

$$\sum_{i=1}^{K} Size(\hat{R}_i) \leq C \quad (2)$$

at any time T over a training iteration, where $\{Size(\hat{R}_i) : 1 \leq i \leq K\}$ are the sizes of active regions in the system.

Intuitively, Equation 1 represents that an iteration, whose duration corresponds to $Time(R_1(d))$ - $Time(R_1(a))$, must be delayed by the minimum *Timeshift*. Note that there are other costs such as model synchronization in distributed training that affect training time. They are mostly static [15,25,37,45] and can be easily factored in.

5.2 Time Shift Model

From the perspective of memory sharing, the worst case possible is having forward passes of all training jobs run simultaneously. This may lead to no memory sharing opportunity as regions being freed out in the backward pass of a job may not be used by the other training job. Therefore, when it comes to exceeding the memory budget C, scrooge scheduler adds a time delay driven by the $TimeShift$ parameter to a training iteration appropriately to ensure that memory allocations during the forward pass occur when enough memory is available.

Since DNN training is highly periodic and deterministic, when training on an apportioned GPU compute capacity is stabilized, we see almost no variations on the region lifespans over iterations. Moreover, for each iteration, allocation times for adjacent regions exhibit strong temporal dependency. In other words, the time interval between $Time(R_{i(a)})$ and

$Time(R_{i+1(a)})$ for a job is almost static and does not change drastically over iterations. Similarly, deallocations of any two consecutive regions have such strong temporal dependency. This observation suggests that $TimeShift$ is the most effective when applied to the entire iteration, not an individual region. Suppose we postpone the allocation of an arbitrary region R_i by $TimeShift$. In that case, deallocations of the regions between R_1 and R_i will be all postponed by $TimeShift$, because allocations for the regions subsequent to R_i get delayed by $TimeShift$ and temporal dependencies during deallocations are still preserved. This results in increasing overall memory usage for the iteration unnecessarily.

5.3 Memory Sharing Algorithm

Now, we explain how scrooge scheduler works to enable spatial memory sharing. Scrooge scheduler optimizes for the minimum possible iteration time based on Equation 1 at runtime. To solve the problem, the scheduling algorithm must address two challenges: C-1) The lifespan of the region, $L(R_i)$, changes according to how two training jobs execute under co-location; C-2) While $L(R_i)$ is changing, the schedule has to find an optimal $TimeShift$ in Equation 1. Scrooge scheduler performs iterative searching steps to reach the goal. For C-1, scrooge scheduler introduces a feedback-driven online process in which the scheduler monitors lifespans of all R_i during the current step and updates them to use in the next step. For C-2, at each step, scrooge scheduler decides $TimeShift$ of the co-located training jobs toward minimizing their iteration times. After several steps, the lifespans are adjusted enough and stabilized. $TimeShift$ should be estimated in an iteration basis to determine when the current iteration has to start.

Profiling phase (The first search step). When a new training job is issued, scrooge scheduler initiates *profiling phase* during which it collects basic information on the new job. In particular, the scheduler runs the first iteration of the new job in an isolated manner. During this profile phase, scrooge scheduler identifies regions by type and obtains lifespan for each feature map region in the solo run[1].

Informed phase. After the profile phase, scrooge scheduler knows useful information for co-locating jobs. In this informed phase, for a new iteration to be started for a job (e.g., job A) with $TimeShift = 0$, the scheduler navigates through time progress using lifespan information and predicts if the memory constraint in Equation 2 is violated or not. If violated, the scheduler waits for time T, which leads to $TimeShift$ += T, and does the prediction again. This time-shifting continues until the scheduler meets the memory constraint — this is guaranteed as the other co-located job (e.g., job B) will ultimately release all regions at the end of its backward pass.

To illustrate, for job A's forward pass, $Time(R_2^{Job\ A}(a))$ - $Time(R_1^{Job\ A}(a))$ indicates the time duration in which the job A uses $Size(R_1^{Job\ A}(a))$ amount of memory, which corresponds to the allocation of the first region $R_1^{Job\ A}$. Likewise, $Time(R_N^{Job\ A}(a))$ - $Time(R_1^{Job\ A}(a))$ indicates the time taken for the entire forward pass in which job A gradually demands more memory by allocating regions from $R_1^{Job\ A}$ to $R_N^{Job\ A}$. In this way, scrooge scheduler can forecast the memory demand trend for job A's forward pass and similarly the trend for its backward pass.

To fulfill the condition in Equation 2, scrooge scheduler also needs to know the memory demand trend of the co-located job B. Assume that job B deallocates its R_{i+1}, i.e., $R_{i+1}^{Job\ B}$, when the scheduler attempts to schedule job A's forward pass. Then, scrooge scheduler knows that during $Time(R_{i-1}^{Job\ B}(d))$ - $Time(R_i^{Job\ B}(d))$, job B will use $\sum_1^i Size(R_i^{Job\ B})$ amount of memory. As such, as job B gradually deallocates its in-use regions from $R_i^{Job\ B}$ to $R_1^{Job\ B}$ in order over time, job B will ultimately release $\sum_1^i Size(R_i^{Job\ B})$ amount of memory after $Time(R_1^{Job\ B}(d))$ - $Time(R_i^{Job\ B}(d))$.

With this memory trend information on job A and job B, scrooge scheduler can decide during Job B's backward pass, if Job A can start by computing 1) the amount of memory required by Job A as time progresses and 2) the amount of memory released by job B as time progresses in terms of regions. This brings out the system-wide active regions as time progresses, which scrooge scheduler uses to predict if memory consumption will always fit in the defined memory budget, i.e., if Equation 2 is satisfied.

In the informed phase, we first use a memory budget which is lower than the actual memory budget C to calculate region lifespans under a conservative schedule that incurs a smaller interference among jobs and thus a less aggressive memory sharing. At this time, the execution of concurrent training is far from optimal, i.e., having long time shifts. From this point, scrooge scheduler iteratively optimizes the objective function. The scheduler gradually increases the lowered memory budget to allow more interference over time and keeps updating region lifespans until it reaches back the memory budget C. It is necessary to calculate region lifespans under co-location with such adaptation because co-scheduling jobs that together have >100% compute demands than the GPU's capacity will inadvertently interfere with the lifespan calculations.

Note that scrooge scheduler works regardless of training two heterogeneous models or when forward passes of the two jobs overlap. Although scheduling is a per-iteration process, it operates at a low cost as memory patterns are already discretized into region level (the scheduling overhead will be discussed in Section 7.4). In reality, the slowdown for each training is very predictable when two training jobs exhibit very similar GPU compute demands. A rare but challenging case is when a beefy model is trained along with a lean model which suddenly suffers from a dramatic slowdown from co-location. This case results in memory not being released from the lean model for a long time while the beefy model keeps allocating memory, leading to the system-wide

[1] This profiling can also be done offline to reduce online profiling cost.

memory usage going over GPU memory capacity. In this case, Zico gives up an attempt on spatial GPU sharing and falls back to time-multiplexing.

Deadlock potential. With concurrent jobs in phases of increasing memory allocation, the deadlock could happen when there is insufficient memory to allocate to these in-flight jobs. Scrooge scheduler does not start the current iteration of a training job if global memory consumption would go beyond the memory budget during its forward-backward passes. In the worst case, scrooge scheduler will schedule the current iteration when the co-located job finishes its iteration (releasing all memory), guaranteeing training progress similar to temporal sharing and thus preventing the deadlock. Such planned execution should be accompanied by tracking of memory used by GPU, which we will explain in the next section.

6 Memory Management with Concurrency

We discuss how to track GPU memory usage for sharing, classify tensors according to usage type, and manage GPU memory for tensors of different types using regions.

6.1 Tracking Memory Usage in GPU

Most of existing DNN platforms are mainly customized for single-job training and unaware of memory sharing among concurrent training jobs. As described in Section 3.3, inherent asynchrony between CPU and GPU does not incur any correctness issue for memory operations when there is only one job to run on the platform. However, for concurrent training with each job assigned with its own GPU stream, we should not make memory released by CPU readily available for the co-located jobs, since kernel computations to be launched on the multiple GPU streams are independent and unordered. In essence, the memory corruption could occur because there is no dependency among kernels in different GPU streams. We apply two techniques to support efficient memory sharing without the corruption.

Memory deallocation. For memory deallocation, rather than immediately releasing the memory based on CPU point of view, we must wait until GPU actually completes the corresponding kernel execution. Zico facilitates *CUDA event* to meet this requirement. CUDA event is a specific kernel that can be inserted into the GPU stream and monitored by CPU for its completion. Once GPU reaches a CUDA event, it is guaranteed that kernels launched before the detected CUDA event have finished the execution. Hence, tensors of those kernels can be safely released if no longer accessed. Tracking in-use memory with CUDA event is not cost-free, requiring CPU cycles. To reduce the overhead, Zico inserts CUDA event periodically at a certain memory allocation granularity (currently, 8 MB for CNNs and 256 MB for RNNs). We discuss the sensitivity of the granularity in Section 7.4.

Memory allocation. The memory tracking brings out another issue, namely, *early memory allocation*. Although deallocation of memory occurs after the actual kernel completion as a result of the memory tracking, its allocation occurs by CPU when the kernel is issued to the GPU stream. Thus, the allocation time is typically earlier than the time the kernel actually starts its execution, accesses the memory, and completes the execution within GPU. It is always desirable to maintain a small number of in-flight kernels (i.e., kernels pending on GPU stream), since this way would narrow the above time gap and ultimately avoid unnecessary pre-allocations of memory by CPU: otherwise, the system makes memory allocations too early compared to the actual time the memory is utilized by GPU kernels. Oftentimes, we found memory consumption *soars up* due to a number of in-flight kernels issued at full CPU speed. To address this issue, Zico limits the number of in-flight kernels to a certain level just sufficient to keep GPU busy. Currently, the right number is obtained via offline profiling for each model while not causing a loss in overall training performance. It can be also easily found online by running a few iterations over varying numbers of in-flight kernels to be limited in the GPU.

6.2 Tensor Classification

With a computation graph constructed for training, Zico differentiates the tensors used by GPU kernel operators forming the graph. In general, we classify the current tensors into three types: feature map tensors, gradient map tensors, and temporary tensors, where temporary tensors are neither feature maps nor gradient maps. A temporary tensor is mostly created by an operator and used internally, not later accessed by other operators. To correctly classify tensors in this way, we exploit three pieces of information available for a tensor: by considering the operator creating the tensor as the *source* operator, (i) whether the source belongs to the forward pass; (ii) whether there is any *destination* operator accessing the tensor outside the source; and (iii) whether the destination belongs to the backward pass. Feature map tensors will meet all three conditions while gradient map tensors are distinguished from complying with (ii) and (iii) only. The remaining tensors are all sorted into temporary tensors.

For the proposed tensor classification, we need to identify whether a computation kernel is involved in the forward pass or the backward pass. Memory manager in Zico pinpoints the peak memory as the starting point of the backward pass. This method exploits the fundamental property of DNN training that a forward pass is a memory allocation phase and a backward pass is a memory deallocation phase. This method is a simple but generic approach that does not depend on DNN implementation and does not belong to a specific framework.

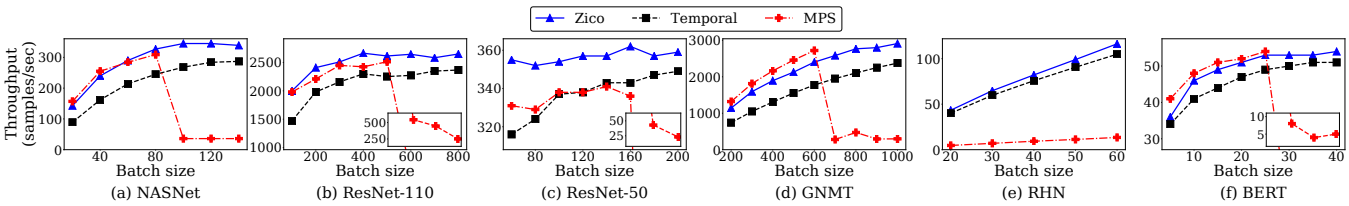


Figure 5: Throughput in training the same models. ResNet-50 and BERT are run on Machine 1 and NASNet, ResNet-110, GNMT, and RHN are on Machine 2 considering their model sizes.

6.3 Managing Memory Regions

Based on the tensor classification, the memory manager in Zico accepts the tensor type as parameter and then allocates the tensor on a region according to the type. The region-based memory management is a basic mechanism in TensorFlow and we extend it to build our own memory sharing system.

The essential goal of Zico's memory management policy is to promote spatial sharing with low interference between co-scheduled training jobs. Based on separating sharable regions (global) from non-sharable regions (job-local), we assure no contention occurs when non-sharable regions are enough to allocate new tensors for the local job. Further, within the local regions, temporary tensors are stored exclusively on a few regions managed by their own free block lists which manage empty memory space for allocating new tensors. Temporary tensors are small in size and frequently allocated and soon deallocated, thus contending with other types of tensors for accessing free block lists unless managed separately.

The size of the sharable and non-sharable area changes elastically depending on runtime demands. As a region stores tensors in the same type, for feature map tensors the demand will increase and decrease within each iteration. Thus, during forward pass, the local memory manager will continuously request regions from the sharable area and then during the backward pass, these regions will be returned back to the sharable area. The free regions in the sharable area are shared through a free list for which updates are synchronized by a lock. To prevent the potential contention on the lock, the granularity of the regions needs to be chosen carefully. We experimentally validated different region sizes over diverse training jobs. From the sensitivity study, we chose the region size to be at least tens of MB to minimize lock contention. The results of the sensitivity study can be found in Section 7.4.

7 Evaluation

Experimental setup. We implement Zico in TensorFlow 1.13.1 and compare it with spatial sharing using MPS (`MPS`)[2] and temporal sharing with no job switching overhead (`Temporal`), which is similar to the approach taken in the state-of-the-art [44]. We select six training benchmarks across different DNN tasks including `NASNet` [48], `ResNet-110` [16], `ResNet-50` [16], `GNMT` [42], `RHN` [47], and `BERT` [11]. All models use the stochastic gradient descent (SGD) optimizer.

The evaluation is performed on two machines. `Machine1` has an NVIDIA Tesla V100 GPU with 32 GB GPU memory, 3.8 GHz Intel Xeon(R) Gold 5222 4 CPU cores and 64 GB of host memory. `Machine2` has an NVIDIA RTX 2080 Ti GPU with 11 GB GPU memory, 3.8 GHz Intel Xeon(R) Gold 5222 4 CPU cores and 64 GB of host memory. Both machines run Ubuntu 16.04. We use Machine1 and Machine2 to evaluate large models and small models, respectively.

7.1 Training Same Models

We first compare Zico, MPS, and Temporal when two identical models are concurrently trained. The memory budget in Zico is configured as GPU memory capacity.

Figure 5 shows the throughput of the six models when training over different input batch sizes (i.e., number of samples) in x-axis. For each model, some of the batch sizes are chosen to have MPS exceed GPU memory capacity to show how effective Zico is in such cases. The figure shows that as compared to temporal sharing, Zico achieves higher throughput across all batch sizes in all models. In particular, Zico outperforms Temporal by on average 35% for NASNet and 37% for GNMT across the batch sizes. These results are rather surprising as the largest batch size in each model results in memory consumption in solo training that reaches close to the GPU memory limit. Even in such an extreme memory usage scenario, Zico finds an optimal time point to start the forward pass of a job while the backward pass of the other job is in progress. Therefore, Zico does not have any model being completely time-multiplexed, making it co-schedule the jobs more efficiently than Temporal.

Zico achieves comparable throughput with MPS from small to modest batch sizes for each model. MPS is sometimes slightly better than Zico. This is not because MPS provides a better schedule for concurrent training but mainly because the underlying setup is different: Zico runs on a single framework and MPS runs two training jobs on different framework instances and processes. Nonetheless, throughput in MPS drops significantly when models are trained on large batch sizes.

[2]Spatial sharing within a single framework exhibits similar limitations to MPS, i.e., performance degradation with memory oversubscription.

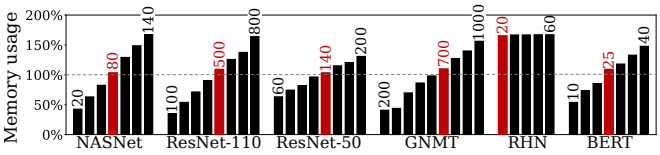


Figure 6: Aggregated memory usage for training the same models. For each model, bars are sorted by the batch sizes as used in Figure 5. The red bars indicate the batch sizes of aggregated memory usage beyond GPU memory capacity.

On training large batches, MPS suffers from GPU memory oversubscription that accompanies UVM overhead. Subsequently, as compared to MPS, Zico is up to 4.7 times faster across models. Note that the solo training of RHN incurs high memory usage even with small batch sizes, causing memory oversubscription for MPS across all batch sizes. Figure 6 presents the system-side memory usage (which sums up the memory usage of the two co-located jobs) to reveal the degree of GPU memory oversubscription handled by Zico.

In Figure 7, we show how memory usage patterns are coordinated in Zico to reduce the system-wide memory footprint. For the space reason, we select only two models, ResNet-110 and BERT, for which concurrent training is scheduled slightly differently. In ResNet-110, almost no delay on each iteration occurs, i.e., $TimeShift \approx 0$, making it behave similar to the non-coordinated spatial GPU sharing. On the contrary, in each scheduling interval of BERT, there is a slight delay applied to every iteration to keep memory consumption within the budget. It is worth noting that for training the same models, the memory-aware schedules across iterations for a job are very regular, making scheduling decisions across the co-located jobs rather deterministic. We also found out training the same models entails an almost similar, if not the same, slowdown for each job. Hence, scrooge scheduler can quickly stabilize its memory-aware scheduling across jobs even without beginning from a low memory budget during the informed phase.

In general, Zico delivers more benefit to less computation-intensive models such as GNMT. Over GPU generations, GPU compute capacity scales faster than GPU memory capacity does, keeping the bottleneck pushed towards GPU memory capacity. With this trend continued, the advantage of Zico over temporal sharing is likely to grow in the future.

7.2 Training Non-identical Models

Now, we use two distinct models in concurrent training. In this experiment, we select five models to make combinations based on different GPU compute demands: GNMT (`low`), RHN (`low`), NASNet (`high`), ResNet-110 (`high`), and BERT (`high`). Figure 8 shows the throughput normalized by Temporal over diverse co-location combinations which all oversubscribe GPU memory capacity. In the figure, we put the memory demand of individual training in the parenthesis, computed as the percentage of GPU memory capacity, which

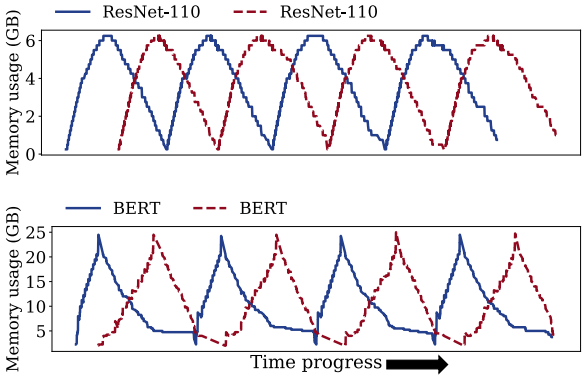


Figure 7: Memory usage over time for training the same models. ResNet-110 and BERT are shown as an example.

is obtained by varying batch sizes for the model.

For training non-identical models, we organize co-location combinations into three scenarios: (i) including both models with low GPU utilization (i.e., RH+GT), (ii) including one model with low GPU utilization (i.e., NN+GT and BT+GT), and (iii) including both models with high GPU utilization (i.e., NN+RN). First, Figure 8 shows that Zico significantly outperforms MPS regardless of GPU utilization between the co-located jobs. Zico is around 5.7x faster than MPS on average, specifically faster up to 5.1x in RH+GT, 8.3x in NN+GT, 6x in BT+GT, and 6.5x in NN+RN. Our conclusion repeats: MPS experiences significant performance degradation under GPU memory oversubscription.

In comparison to Temporal, Zico achieves higher throughput by 42% in RH+GT, 46% in NN+GT, 27% in BT+GT, and 15% in NN+RN on average. That is, Zico favors scenario (i) and (ii) over (iii) because ample GPU cycles are available by running a model with low compute demand. Nonetheless, in Zico, since any of co-located jobs starts training once memory constraint is met, no fair use of compute resources is guaranteed. As a result, in the NN(30%)+GT(90%), Zico obtains a great throughput improvement for GT over Temporal but slightly worse performance for NN due to a bit fewer iterations scheduled within a time duration as opposed to Temporal. We leave balancing individual job throughput on top of increasing the aggregated throughput as future work.

It is also worth noting that Zico achieves different scheduling ratio for the co-located jobs depending on their memory usage patterns. To illustrate, we show memory usage patterns in RN+NN in Figure 9, where ResNet-110 has relatively shorter iteration time compared to NASNet. On the given memory budget, Zico decides to schedule executions of the two jobs in such a way that ResNet-110 runs its iterations more frequently than NASNet does — Zico keeps scheduling ResNet-110 during time periods with low-to-moderate memory usages in NASNet to maximize GPU memory utilization.

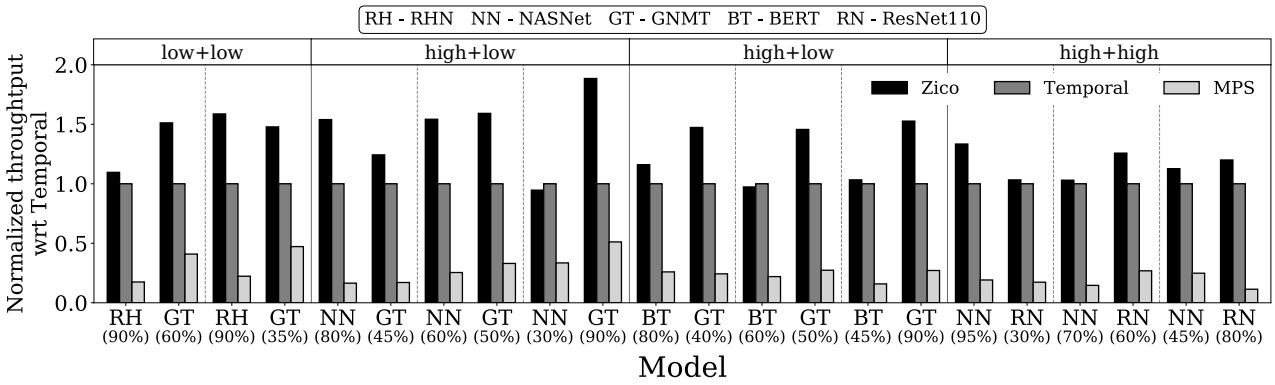


Figure 8: Throughput in training the distinct models concurrently. All co-locations are run on Machine 2.

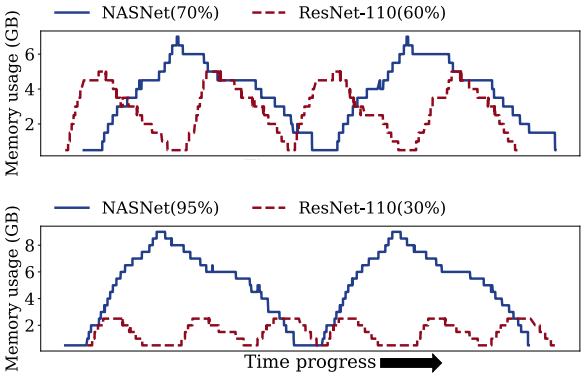


Figure 9: Memory usage over time for training NASNet and ResNet-110 concurrently. This co-location incurs different scheduling ratios for having different memory demands.

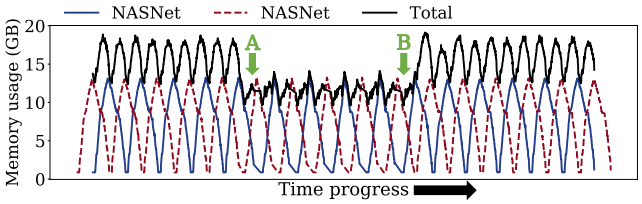


Figure 10: Memory usage patterns on dynamic memory budget changes. The budget is decreased from 70% GPU memory to 50% at time A and then increased back to 70% at time B.

7.3 Dynamic Memory Budget Change

Recall that for co-locating more than two jobs, e.g., four jobs, we propose to organize the jobs into a group of pairs and schedule each pair independently with a lower memory budget. Then, when some of the jobs depart, the system would have fewer pairs and need to make a schedule based on the increased memory budget. Therefore, adapting to the memory budget change is a fundamental functionality required in Zico to deal with the dynamic workload. In this section, we evaluate Zico scheduling decisions when several continuous changes are made on the memory budget.

Figure 10 shows how Zico schedules two NASNet training jobs while decreasing the memory budget and then increasing it back. Before the first change, the two jobs run concurrently with zero delay by virtue of scrooge scheduler between consecutive iteration executions of a job under the budget set as 70% GPU memory. Around *A* time point, the memory budget is set down to 50% and Zico begins to schedule the co-located jobs more conservatively. During this period, each job exhibits a wider gap (140 ms) between consecutive iterations. Around *B* time point, the memory budget returns back to 70% and Zico now takes a more aggressive schedule on memory sharing. At this moment, we attempt to further increase the memory budget to fully utilize GPU memory, but we do not see any change in both throughput and memory footprint. The reason is that 70% GPU memory is already enough to make an ideal scheduling decision for Zico with no TimeShift. That is, Zico does not overuse GPU memory unnecessarily.

7.4 Design Validation

GPU memory tracking. As explained in Section 6.1, tracking in-use memory is essential to share memory with the correctness between multiple GPU streams. Table 1 shows the sensitivity of memory tracking granularity. If CUDA event is inserted too frequently, it imposes overhead on CPU and leads to delaying training iteration. For instance, inserting CUDA event for every GPU kernel launch slows down the training up to more than 50% in GNMT. To mitigate this overhead, CUDA event is inserted periodically in Zico. For models which use CPU intensively like GNMT, due to the significant number of light-weight kernels to issue, coarse-grained tracking is required to avoid this overhead. On the

GPU tracking	NN	RN-110	RN-50	GNMT	RHN	BERT
All	78%	91%	97%	44%	94%	97%
Fine-grained	100%	100%	100%	89%	99%	99%
Coarse-grained	100%	100%	100%	97%	99%	100%

Table 1: Throughput with memory tracking (normalized to the throughput with no memory tracking). `All`, `Fine-grained`, and `Coarse-grained` track GPU memory for every kernel launch, 8 MB allocation, and 256 MB allocation, respectively.

Granularity	NN	RN-110	RN-50	GNMT	RHN	BERT
Tensor	94%	99%	98%	90%	96%	94%
Small region	94%	99%	100%	97%	99%	99%

Table 2: Throughput with different sharing granularity (normalized to sharing granularity of 64 MB region size).

other hand, for models which use CPU less like CNN models and BERT, even fine-grained tracking does not bring out the actual delay of the training iteration. Zico chooses the right memory tracking granularity, minimizing the overhead.

Sharing granularity. As mentioned in Section 6.3, if the sharing granularity is too fine-grained, e.g., tensor granularity, the contention of shared lock becomes non-negligible. Table 2 shows the sensitivity of different sharing granularity choices, where the size of small region is set to 512 KB. The table presents the normalized throughput with respect to using our default region size (64 MB) for memory sharing. The tensor-level sharing could introduce up to 10% of throughput degradation as shown in GNMT.

Scheduling. Making a scheduling decision in our scrooge scheduler takes $O(n)$ time complexity, where n is a small number of regions exercised by co-located jobs. The observed overhead is only a few hundreds of nanoseconds, and hence scrooge scheduler has nearly zero scheduling overhead. Moreover, the scheduling process of one job does not interfere with the scheduling process of counterpart co-located job, since each job has a dedicated CPU thread.

8 Related Work

Temporal/Spatial GPU sharing. Temporal GPU sharing represents software-based techniques that time-share GPU for DNN workloads. Gandiva [43] proposes a GPU time-slicing mechanism for the first time to mainly accelerate hyperparameter tuning jobs. It initiates job switching at iteration boundary to reduce CPU-GPU communication overhead. Salus [44] tries to remove the switching overhead by making model parameters of a job resident in GPU memory even when the job is inactive. It further integrates a spatial sharing mechanism to harness underutilized memory in a similar way to MPS [10]. We faithfully compare Zico with both temporal and spatial sharing in Section 7.

Tensor swapping/recomputation. Prior works make use of host memory as a swap storage for DNN training to mitigate memory footprint in GPU [17,32,36]. vDNN [36] predictively swaps tensors ahead of time to overlap CPU-GPU communication with GPU computation during training. It mainly focuses on swapping the inputs of convolutional layers as they tend to have long lifespans in CNN models. SwapAdvisor [17] considers memory allocation and operator scheduling to jointly optimize for swapping decisions. Capuchin [32] proposes a computational graph agnostic technique that estimates the costs of tensor swapping and recomputation to make the best choice between the two.

Other prior works study dropping tensors created in forward pass and recomputing them in backward pass [6, 19, 41]. SuperNeurons [41] introduces a cost-aware recomputation technique to remove tensors from convolution layer, which are cheap to recompute. Checkmate [19] formulates tensor recomputation into an optimization problem and provides an approximation algorithm to recompute tensors timely. Similar to tensor swapping, tensor recomputation reduces memory footprint for a single training. The goal of Zicois different; Zico reduces global memory footprint for concurrent training.

Compression. Many approaches were invented to reduce memory footprint of DNN training, including HW-based compression techniques [7, 30]. There are also a few SW-based memory compression techniques. Gist [18] proposes a series of layer-specific encoding techniques to compress tensors including feature maps. Echo [46] proposes a compression technique more effective on LSTM RNN training driven by internal operator dependencies. Zico is complementary and can be combined with tensor compression techniques.

9 Concluding Remarks

We present our attempt on realizing GPU memory sharing across concurrent training. The proposed system, Zico, is the first introducing a memory-aware scheduler that coordinates training iterations among co-located jobs with minimum stall times. Zico works generally for co-locating both identical models or non-identical models regardless of the iteration time and the memory pattern of each model. Our experimental results show that Zico outperforms existing GPU sharing approaches. With growing model sizes, very large models such as GPT family [33,34,38] are preferred to run with model parallelism or data parallelism to accommodate intermediate tensors on GPU memory. Despite diverse parallelism in use, we believe Zico benefits both cases as we still see increasing and decreasing memory usage within an iteration.

Acknowledgements

We thank our shepherd, Nandita Vijaykumar, and anonymous reviewers for their valuable comments and suggestions. We also thank Peifeng Yu and Chanho Park for their knowledge sharing and technical support during this work. This work was supported by Samsung Advanced Institute of Technology, the 2021 Research Fund (1.210050.01) of UNIST(Ulsan National Institute of Science and Technology), Electronics and Telecommunications Research Institute(ETRI) grant funded by the Korean government (21ZS1300), and the National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIT) (No. 2020R1C1C1014940 and NRF-2019R1C1C1005166).

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *USENIX OSDI*, 2016.

[2] Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, et al. TensorFlow Eager: A Multi-Stage, Python-Embedded DSL for Machine Learning. *arXiv preprint arXiv:1903.01855*, 2019.

[3] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. In *NeurIPS*, 2011.

[4] James Bergstra, Daniel Yamins, and David Cox. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *ICML*, 2013.

[5] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv preprint arXiv:1512.01274*, 2015.

[6] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training Deep Nets with Sublinear Memory Cost. *arXiv preprint arXiv:1604.06174*, 2016.

[7] Esha Choukse, Michael B Sullivan, Mike O'Connor, Mattan Erez, Jeff Pool, David Nellans, and Stephen W Keckler. Buddy Compression: Enabling Larger Memory for Deep Learning and HPC Workloads on GPUs. In *ISCA*, 2020.

[8] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[9] NVIDIA Corp. CUDA, release: 10.2.89. https://developer.nvidia.com/cuda-toolkit, 2020.

[10] NVIDIA Corp. Multi-process service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2020.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[12] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to Forget: Continual Prediction with LSTM. *Neural Computation*, 12(10), October 2000.

[13] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv preprint arXiv:1706.02677*, 2017.

[14] Andreas Griewank and Andrea Walther. Algorithm 799: Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.

[15] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *USENIX NSDI*, 2019.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *CVPR*, 2016.

[17] Chien-Chin Huang, Gu Jin, and Jinyang Li. SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *ASPLOS*, 2020.

[18] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient Data Encoding for Deep Neural Network Training. In *ISCA*, 2018.

[19] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *MLSys*, 2020.

[20] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, unjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *USENIX ATC*, 2019.

[21] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *MM*, 2014.

[22] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*, 2017.

[23] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NeurIPS*, 2012.

[25] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *USENIX OSDI*, 2014.

[26] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and Efficient GPU Cluster Scheduling. In *USENIX NSDI*, 2020.

[27] Avner May, Alireza Bagheri Garakani, Zhiyun Lu, Dong Guo, Kuan Liu, Aurélien Bellet, Linxi Fan, Michael Collins, Daniel Hsu, Brian Kingsbury, et al. Kernel Approximation Methods for Speech Recognition. *arXiv preprint arXiv:1701.03577*, 2017.

[28] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent Neural Network Based Language Model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[29] NVIDIA Corp. NVIDIA A100 Tensor Core GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

[30] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks. In *ISCA*, 2017.

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *arXiv preprint arXiv:1912.01703*, 2019.

[32] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-Based GPU Memory Management for Deep Learning. In *ASPLOS*, 2020.

[33] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving Language Understanding by Generative Pre-Training. 2018.

[34] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. *OpenAI blog*, 1(8):9, 2019.

[35] Jeff Rasley, Yuxiong He, Feng Yan, Olatunji Ruwase, and Rodrigo Fonseca. HyperDrive: Exploring Hyperparameters with POP Scheduling. In *Middleware*, 2017.

[36] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-efficient Neural Network Design. In *MICRO*, 2016.

[37] Alex Sergeev and Mike Del Balso. Meet Horovod: Uber's Open Source Distributed Deep Learning Framework for TensorFlow. https://eng.uber.com/horovod/, 2017.

[38] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[39] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the Importance of Initialization and Momentum in Deep Learning. In *ICML*, 2013.

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *NeurIPS*, 2017.

[41] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *PPoPP*, 2018.

[42] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv preprint arXiv:1609.08144*, 2016.

[43] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *USENIX OSDI*, 2018.

[44] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications. In *MLSys*, 2020.

[45] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *USENIX ATC*, 2017.

[46] Bojian Zheng, Nandita Vijaykumar, and Gennady Pekhimenko. Echo: Compiler-Based GPU Memory Footprint Reduction for LSTM RNN Training. In *ISCA*, 2020.

[47] Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutnık, and Jürgen Schmidhuber. Recurrent Highway Networks. In *ICML*, 2017.

[48] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning Transferable Architectures for Scalable Image Recognition. In *CVPR*, 2018.