2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-2235-2/20/$31.00 ©2021 IEEE | DOI: 10.1109/HPCA51647.2021.00057

Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning

Jie Ren*, Jiaolin Luo*, Kai Wu*, Minjia Zhang†, Hyeran Jeon*, Dong Li*
*University of California, Merced, †Microsoft
{jren6, jluo38, kwu42, hjeon7, dli35}@ucmerced.edu, minjiaz@microsoft.com

***Abstract*—Memory capacity is a major bottleneck for training deep neural networks (DNN). Heterogeneous memory (HM) combining fast and slow memories provides a promising direction to increase memory capacity. However, HM imposes challenges on tensor migration and allocation for high performance DNN training. Prior work heavily relies on DNN domain knowledge, unnecessarily causes tensor migration due to page-level false sharing, and wastes fast memory space. We present Sentinel, a software runtime system that automatically optimizes tensor management on HM. Sentinel uses dynamic profiling, and coordinates operating system (OS) and runtime-level profiling to bridge the semantic gap between OS and applications, which enables tensor-level profiling. This profiling enables co-allocating tensors with similar lifetime and memory access frequency into the same pages. Such fine-grained profiling and tensor collocation avoids unnecessary data movement, improves tensor movement efficiency, and enables larger batch training because of saving in fast memory space. Sentinel reduces fast memory consumption by 80% while retaining comparable performance to fast memory-only system; Sentinel consistently outperforms a state-of-the-art solution on CPU by 37% and two state-of-the-art solutions on GPU by 2x and 21% respectively in training throughput.**

***Index Terms*—heterogeneous memory, deep neural network training, memory management**

I. Introduction

Deep neural networks (DNN) have been shown preliminary success in many fields. However, training those models can be extremely memory-consuming. For example, the recent language models and translation models have 100s of billions of parameters [1] requiring 100s of GB of memory for training. Although it has been repeatedly demonstrated that larger models and more data lead to improved model accuracy on many tasks [2]–[4], the memory becomes a major bottleneck either when training models with more weight parameters or with larger batch sizes. Lack of memory causes DNN training to have out-of-memory crashes and limits the sizes of the model and batch for training, causing degradation in training effectiveness and efficiency [5], [6]. Adding more DRAM can mitigate the problem, but often comes with huge costs. In this work, we look into overcoming the memory scaling issue for DNN training by leveraging heterogeneous memory (HM) to achieve larger memory capacity.

HM is an emerging memory architecture. Within HM, multiple memory components with different technologies are combined to construct main memory. HM is typically composed of a high-capacity memory (but with relatively worse performance, such as non-volatile memory) and a high-performance memory (but with smaller capacity, such as DRAM). HM brings a promising solution to increase memory capacity and avoids the limitation of existing memory technologies.

HM for DNN training has been explored by several studies [5]–[11]. Most of them focus on mitigating GPU-side memory space limitation by leveraging larger CPU-side system memory [5], [6], [8]–[11], while a recent study demonstrates a HM that uses a persistent memory to scale the CPU-side DRAM capacity [7]. As observed in these studies, computation efficiency of using HM requires a careful memory management, such as timely tensor placement and migration, subject to the access patterns of tensors and the performance disparity of different memory components. More specifically, existing solutions explore methods to proactively release and prefetch temporarily inactive tensors, determine inactive tensors based on DNN topology, and find data swap time between memories by analyzing tensor access order [7]–[10] or using detailed domain knowledge [5], [6], [11].

However, there has not been a study that thoroughly evaluates individual tensor characteristics and the semantic gap between operating system/architecture and memory management in deep learning frameworks. Missing this study leaves many performance improvement opportunities on the table. For example, most of the existing solutions focus on the order of tensor accesses to determine the timing and target tensors to swap. However, such solutions are oblivious to tensor characteristics such as number of tensor accesses in memory and tensor lifetime, which would lead to unnecessary tensor migrations. For example, some tensors such as short-lived ones might be better to stay in fast memory rather than unnecessarily migrated to be used only a few times or never used again.

More importantly, most of the existing studies tackle tensor placement at the granularity of individual tensors. However, as the memory management of operating system (OS) and underneath architecture is conducted at the page level, such a tensor-level mapping would lead to memory fragmentation or unexpected tensor migration if multiple tensors having different access patterns are mapped to the same page. For example, if there are four pages, where 25% of each page is mapped with tensors frequently accessed in the similar time

978-1-6654-2235-2/21/$31.00 ©2021 IEEE
DOI 10.1109/HPCA51647.2021.00057

and having similar lifetime, all four pages would need to be placed in fast memory, even though the remainder of the four pages is filled with rarely accessed tensors whose lifetimes are different from that of the frequently accessed tensors. However, if we co-locate all frequently accessed tensors to one page in this example, placing this single page in fast memory would be sufficient for high performance training.

To address the aforementioned issues, we propose *Sentinel*, a software runtime system that automatically manages and optimizes tensor migration and allocation in HM, and allows to train DNN models with a much smaller size of fast memory but achieves performance similar to that on the fast memory-only system. To do that, we first conduct an extensive study on *workload characteristics of DNN*. We observe that there are a large number of small (less than one page) and short-lived (lifetime shorter than one layer) tensors. On the other hand, the peak memory consumption of frequently accessed tensors (hot tensors) is not big (tens of MB), and tensors commonly share pages but with different access frequencies. These observations indicate that an ideal memory management strategy for DNN training should co-locate tensors with similar lifetime and access frequency in fast memory while minimizing page-level false sharing to reduce peak consumption of fast memory. To our best knowledge, this is the first in-depth tensor and memory mapping characterization of DNN training; It is generally applicable on linear and non-linear network topologies and includes *all* tensors, which is different from those of existing studies [5], [6], [12], [13] that focus on specific tensors (e.g., input tensors of convolution operations or model weights) on certain DNN.

Driven by the observations, Sentinel enables efficient DNN training with three major innovations. First, Sentinel implements a *tensor-level dynamic profiling* to collect characteristics of individual tensors which is impossible in the traditional page-level profiling. This method bridges the semantic gap between OS and DNN application. It allows the runtime to associate tensors with the DNN topology, dynamically identifying long-lived but sparsely accessed tensors that can be migrated. More importantly, our profiling method counts tensor accesses in memory (i.e., the number of accesses to each tensor in memory), not just checks whether tensors are referenced in operations as in many HM management solutions for DNN training [5]–[9], [12], [14]. Counting tensor accesses in memory is fundamental for memory optimization for DNN, because it leads to new optimization techniques, such as tensors co-location and being graph-agnostic.

Second, Sentinel improves *tensor migration efficiency by avoiding page-level false sharing* and unnecessary tensor movement, which is often ignored in related work [5]–[9], [12], [14]. Sentinel aggregates small tensors having a similar lifetime and access count into the same page to prevent page-level false sharing. Sentinel also pins short-lived tensors to a reserved fast memory space to prevent their unnecessary movement to slow memory. Note that the unnecessary movement of short-lived tensors are commonly observed in existing page-level data migration [15] and hardware-managed caching mechanisms [16]–[18], which leads to memory bandwidth waste and performance loss.

Third, Sentinel employs a *performance model-directed proactive migration strategy*. Similar to existing solutions [9], Sentinel dynamically moves unused tensors out of fast memory and moves to-be-used tensors into fast memory to save its space. However, unlike existing studies [5], [6], we consider performance trade-off between migration frequency and performance benefit, as frequent tensor movement can be exposed to the critical path and cause performance loss. To identify the optimum migration interval that not only reduces memory capacity but also avoids performance loss, we introduce analytical performance models that allow exploration of various migration intervals and effectively find the optimum one with negligible runtime overhead. The performance models bring great flexibility and high performance for tensor migration across layers for various DNN topologies. The tensor migration is controlled purely subject to the performance models to maximize overlap between tensor migration and DNN training, unlike existing studies that use migration algorithms heuristically designed for given network topologies [5], [6], [12], [13] or limited memory capacity [8], [9].

In summary, the key contributions are as follows.

- **Characterization study.** We systematically analyze how tensors are allocated and accessed in TensorFlow.
- **Runtime system.** We introduce a runtime system, Sentinel, which is featured with a novel profiling method counting memory accesses at the tensor level. Guided by analytical performance models, Sentinel enables efficient tensor migration by avoiding page-level false sharing and unnecessary data movement.
- **Evaluation.** We evaluate Sentinel on two HM systems: one is based on DDR4 (fast) and Optane DC persistent memory (slow), and the other is based on NVIDIA V100 GPU (fast) and CPU (slow). On the Optane-based system, we show that using only 20% of peak memory consumption of DNN models as fast memory size (a 5X reduction), Sentinel achieves similar performance (9% performance difference on average) to DRAM-only system. Furthermore, Sentinel consistently outperforms two state-of-the-art solutions (IAL [19] and AutoTM [7]) as well as DRAM-cached Optane (using DRAM as a hardware cache) and first-touch NUMA policy, by 37%, 17%, 23% and 70% in training throughput, respectively. On the GPU-based system, Sentinel enables larger batch size in training by 1.9X and higher training throughput by 2X than vDNN [6], and enables comparable batch size and higher training throughput by 16%, 17% and 65% than three state-of-the-art solutions (Capuchin [9], AutoTM [7] and SwapAdvisor [8]), respectively.

II. Background

Training DNN models. A typical DNN model comprises of a stack of *layers*, each of which is a group of neurons. Each neuron in a layer computes a non-linear function of the outputs of neurons in the preceding layer, using a set of weights. Training DNN often involves a large number

TABLE I
COMPARISON BETWEEN EXISTING WORK ON HM MANAGEMENT FOR DNN TRAINING.

	Dynamic profiling	Fast memory usage minimization	Graph agnostic	Count tensor accesses in memory	Page-level false sharing avoidance
vDNN [6]	N	N	N	N	N
Superneurons [5]	N	N	N	N	N
Layrub [12]	N	N	N	N	N
SwapAdvisor [8]	N	N	Y	N	N
AutoSwap [14]	N	N	Y	N	N
AutoTM [7]	N	Y	Y	N	N
Capuchin [9]	Y	N	Y	N	N
HALO [25]	Y	N	Y	N	N
Sentinel	**Y**	**Y**	**Y**	**Y**	**Y**

of training iterations (each iteration is a training step). In each step, a *batch* of training samples are fed into DNN. Performance of each step (e.g., execution time and memory access pattern) remains stable across steps, hence highly predictable [20]–[22]. Training DNN often uses a framework, such as TensorFlow [23] and PyTorch [24]. These frameworks use a dataflow execution model where the workload of DNN is modeled as a directed graph. *Operations*, such as 2D convolution, matrix multiplication, and array concatenation, are implemented as primitives. Those operations are represented as nodes in the graph. Within the graph, edges between nodes capture dependencies between nodes.

Recent efforts. Heterogeneous memory is used for DNN training recently [5]–[9], [12]. We comprehensively compare state of the art with Sentinel in Table I from multiple perspectives. In Table I, dynamic profiling captures the effects of inter-operation parallelism on memory accesses and is generally applicable on various input data sizes and architectures, which are often missed in static profiling. Minimization of fast memory usage means making best efforts to reduce fast memory size; This also means targeting on all tensors (not just a few tensors such as feature maps) to look for migration opportunities. Being graph agnostic means there is no need of detailed DNN knowledge (such as which tensor is feature map or weight), which makes the solution more general, instead of just for some specific DNN models. Counting tensor accesses in memory totals the number of memory accesses at data object level, which is much more than just checking whether data objects are referenced in operations [5]–[9], [25]. Counting tensor accesses provides optimization opportunities to co-locate tensors and prioritize data migration. Avoiding page-level false sharing is necessary to improve page migration efficiency and achieve additional savings of fast memory usage, revealed in Section III-B. Sentinel excels, because it uses dynamic profiling, count tensor accesses in memory, and is graph agnostic; Sentinel has high migration efficiency and enables larger model (or larger batch) training. Sentinel is applicable to both CPU and GPU, as demonstrated in this paper, while some state-of-the-arts (e.g., Capuchin [9]) focus only on GPU and use expensive recomputation to save GPU memory, whose effectiveness on CPU remains to be studied.

There are large differences between generic memory management (GMM) (e.g, tcmalloc [26] and garbage collection (GC) in a managed language/runtime) and Sentinel: (1) Tensor lifetime management in GMM lacks DNN semantics and hence misses opportunities to timely migrate tensors to avoid performance loss or save fast memory, hence fails to minimize fast memory usage, evidenced in Table IV; (2) GMM cannot work well on GPU at tensor levels; (3) Without coordination of OS, GC ignores the impact of CPU cache hierarchy on main memory accesses; (4) Current DNN training frameworks are not based on managed runtime and cannot easily employ GC.

III. ANALYSIS AND CHARACTERIZATION OF MAIN MEMORY ACCESSES IN DNN

We characterize main memory accesses to drive our design.

A. Profiling Framework

We build a profiling framework. It is integrated into TensorFlow and used in a profiling phase (one training step) to direct tensor management at runtime (Sec. IV). The profiling framework collects the following information: (1) the number of *main memory accesses* per tensor, (2) tensor size and (3) lifetime. To collect the above information, the profiling framework includes the support at *both* OS and TensorFlow runtime levels. At OS level, Sentinel collects the number of memory accesses at the page level. This is implemented by a software-only solution. In particular, to track a page for access counting, Sentinel sets a reserved bit (bit 51) in its PTE (i.e., poisoning PTE) and then flush the PTE from TLB. When the page is accessed, a TLB miss occurs and triggers a protection fault. Sentinel uses a customized fault handler to count this page access, poisons the PTE, and flushes it from TLB again to track next page access.

To bridge the semantic gap between OS and DNN framework, each memory page has only one tensor (but a tensor can use more than one pages). This is implemented by making object allocation aligned with memory page. Using this method, page-level profiling becomes tensor-level profiling. This method slightly increases memory footprint (Sec. VII) but it only happens during the profiling phase of Sentinel on slow memory. After the profiling phase, tensors are re-organized to reduce memory footprint and improve performance. Data reorganization happens *during memory allocation* (Sec. III-B), and hence does not stop training process and does not impact performance. The profiling method does not increase the consumption of fast memory.

At the TensorFlow runtime, Sentinel leverages memory (de)allocation to get the size and lifetime of tensors. Moreover, Sentinel introduces an API that allows the user to annotate DNN to indicate the end of each layer in DNN. Based on the above infrastructure, Sentinel is able to associate a tensor with the DNN topology (i.e., we can know which layer(s) a tensor is alive), which is helpful to direct tensor migration.

Our profiling method uses only one training step for profiling. During the profiling, Sentinel captures each page read and write by repeatedly poisoning the page. This is expensive because of system calls and TLB misses. However, it does

not lose profiling accuracy. Also, considering that a typical DNN training involves millions of training steps, the profiling overhead is easily amortized. The traditional profiling methods face a dilemma between profiling overhead and accuracy. In particular, frequently collecting memory access information brings high profiling accuracy at the cost of large runtime overhead, and vice versa [27]–[31]. Leveraging the repetitiveness of DNN training, Sentinel breaks the dilemma, and enables both high profiling accuracy and low profiling overhead.

Our profiling method is featured with the coordination between OS and TensorFlow runtime. This provides accurate profiling, which is unachievable by TensorFlow runtime alone. In particular, OS allows us to track memory accesses filtered by processor caches; Working with the coordination between OS and TensorFlow runtime, we do not need to handle pointer aliasing commonly found in TensorFlow implementation, which is difficult to be handled by a runtime solution.

B. Observations and Preliminary Analysis

We profile DNN models listed in Table III and have the following observations to guide our design:

Observation 1: There are a large number of small tensors with short lifetime in DNN training workloads.

We define a tensor as small if it is smaller than a page size. A tensor is alive after it is allocated and before it is freed. We define the lifetime of a tensor in terms of the number of layers where the tensor is alive. In the rest of the paper, *we define short-lived tensor as those whose lifetime is no longer than one layer.* Taking ResNet-32 as an example (its configuration is in Table III), 92% of its tensors have lifetime no longer than one layer. Among them, 98% is small tensors. The peak memory consumption of short-lived tensors is small, and typically bounded by a few GB.

Observation 2: The uneven distribution of hot and cold tensors provides opportunities for tensor management.

For example, 52.3% of tensors (using 907 MB, which is 54% of total memory pages) in ResNet-32 are accessed less than 10 times in main memory. On the other hand, some tensors in ResNet-32 are frequently accessed (having $>$ 100 accesses), taking only 4 MB (0.2% of total memory pages). They are the candidates to be placed into fast memory, and their size is a small portion of total memory pages.

Observation 3: Page-level false sharing exists in DNN. The page-level profiling (not tensor-level) for tensor management can be misleading.

For example, in ResNet-32, if we perform tensor-level profiling, in a training step, for those less-frequently accessed tensors that have only 1-10 accesses in main memory, their total object size is 908 MB. However, if we perform page-level profiling, in the same training step, for those memory pages with 1-10 accesses in main memory, their total page size is 764 MB. This indicates that some less-frequently accessed tensors fall into some pages that are counted as more frequently accessed in page-level profiling. Hence, if one uses page-level profiling to guide data management, those less-frequently accessed tensors can be placed into fast memory and waste memory space and bandwidth. We refer to the above result as page-level false sharing.

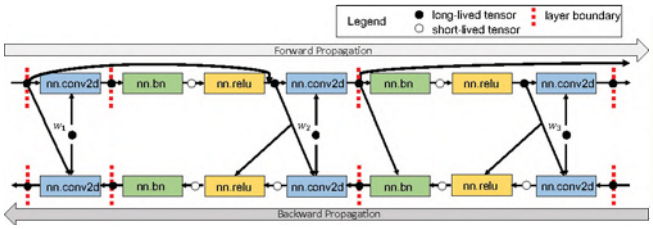

Fig. 1. An example to show tensor access patterns across layers in ResNet-32. This figure shows the first three layers and the last three layers in ResNet-32.

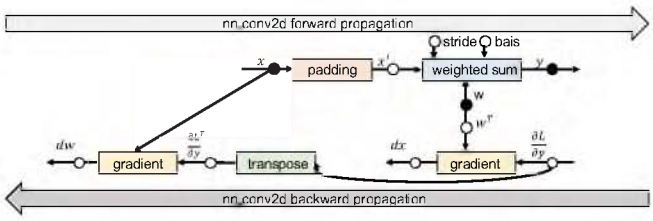

Fig. 2. Data processing in a TensorFlow operation, `nn.conv2d`.

Example. We use ResNet-32 as an example to characterize tensors. Figure 1 shows operations in six layers; Figure 2 shows tensor processing in Operation `nn.conv2d` commonly used in DNN's forward and backward propagation.

Short-lived tensors. We find two cases. (1) Inside an operation, tensor processing (e.g., padding and transpose shown in Figure 2, expansion, concatenation and squeeze) often generates short-lived tensors, which are only used in that operation; (2) The output tensor of some operation is short-lived, exemplified by the output of batch normalization (i.e.,*nn.bn* in Figure 1). Memory allocation and free for a short-lived tensor always happen in one layer.

Long-lived tensors. We find two cases. (1) Weights associated with each layer (shown as "w1" and "w2" etc. in Figure 1). They are allocated before training steps, and updated throughout them. (2) Intermediate results generated in a layer and consumed by the downstream operations in another layers. An example is the output tensors of operations `nn.conv2d` and `nn.relu` in the forward propagation layers shown in Figure 2. These output tensors are consumed by the backward propagation layers to calculate gradients. The memory space for these intermediate results is allocated when they are generated and then freed after they are consumed.

Memory access patterns. Memory accesses to tensors are associated with layers. Memory accesses to short-lived tensors tends to be *ephemeral* and *bursty*, which means in a layer, there can be a number of short-lived tensors created, accessed a few times, and freed. Memory accesses to a long-lived tensor tend to be *sparse* and *periodical*, which means memory accesses happen in a couple of specific layers, but not all layers. In addition, memory allocations for long-lived intermediate results and short-lived tensors are interleaved throughout the training process, which causes page-level false sharing.

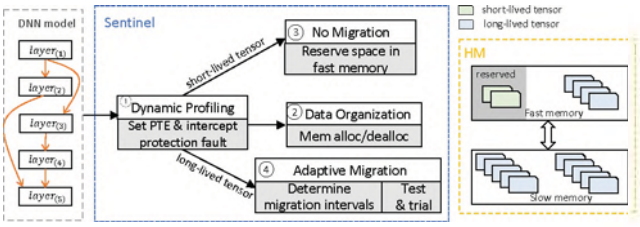

Fig. 3. Overview of Sentinel. The white and shadow boxes represent functionality and mechanisms, respectively.

Design choices. Profiling results motivate us to make three design choices. (1) We choose DNN layer as the basic granularity for tensor management, given the fact that lifetime and memory access patterns of tensors are associated with layers. This choice brings convenience for tensor prefetching and migration overhead controlling. (2) We treat tensors differently, instead of using a unified policy to manage them as in [19], [27]–[31]. This choice allows us to enable high performance and minimize fast memory capacity. (3) We do not use static analysis as in [32] to decide data placement, because static analysis lacks timing information needed to overlap data migration and computation; It cannot accurately capture main memory accesses and ignores the impact of thread-level parallelism on data locality.

IV. Design

A. Overview

Figure 3 overviews Sentinel. Sentinel uses dynamic profiling (Sec. IV-B) to collect the number of main memory accesses at tensor level and lifetime of tensors based on customized memory allocation. The dynamic profiling uses one training step to collect the information. After that, Sentinel re-organizes memory allocation for short-lived tensors to facilitate tensor management and avoid page-level false sharing.

Driven by the profiling results, Sentinel treats short- and long-lived tensors separately. Short-lived tensors are allocated in contiguous memory space in fast memory and not involved in tensor movement. This method (Sec. IV-C) avoids unnecessary tensor movement. To handle long-lived tensors, Sentinel uses an adaptive migration algorithm (Sec. IV-D). It partitions each training step into many migration intervals based on DNN model topology. In a migration interval, Sentinel migrates tensors needed for the next interval, overlapping application execution with tensor migration. Sentinel must determine an appropriate migration interval length [1], such that tensors can be timely migrated from slow to fast memory before they are needed by the next migration interval and without running out of fast memory. We formulate the problem and determine the optimum length. We also use a test-and-trial algorithm to determine if the migration cannot finish before the next interval, whether continuing migration can lead to better performance.

[1]We distinguish *migration interval* and *migration interval length* in the rest of the paper. The number of layers in a migration interval is the migration interval length.

B. Dynamic Profiling and Data Reorganization

Sentinel integrates the profiling framework into TensorFlow. Based on the profiling results, Sentinel uses a customized memory allocation policy in the remaining training steps, described as follows. (1) For those short-lived tensors alive in the same layer, they are allocated into the same pages, because of their similarity in lifetime and the number of memory accesses. (2) For those long-lived tensors that reside in the exactly same layers, we use the following algorithm to determine their memory co-allocation. We first sort them in terms of the number of memory accesses in descending order, and then allocate them in contiguous memory pages, following the order. As a result, tensors with the similar memory access pattern can be allocated into the same memory pages. (3) For those long-lived tensors that do not reside in the same layers, they never share any memory page. (4) Long- and short-lived tensors never share any memory page.

The above data reorganization happens to long- and short-lived tensors allocated in the middle of the training. Those tensors are allocated and freed in each training step, allowing Sentinel to reorganize them across training steps without impacting program correctness. A few long-lived tensors (e.g., weights and input samples) are allocated before the training process; They cannot be reorganized in the middle of training, because that changes memory addresses of the tensors and causes wild pointers. Sentinel ensures that these tensors never share pages to avoid page-level false sharing.

C. Handling Short-Lived Tensors

During training, an individual short-lived tensor is not accessed many times (e.g., less than 10 times in ResNet-32) in main memory, compared to many long-lived tensors. Our profiling results show that there are a large amount of short-lived tensors throughout the whole training, and they share the same memory access characteristics (i.e., short life time, small size, and infrequent accesses in main memory). We must use a general policy to manage them.

We use the following algorithm to manage short-lived tensors. We allocate a continuous memory space in fast memory for them. Tensors in this space are never considered for migration. This space is reused for short-lived tensors as they are allocated and freed throughout the training. The space is reserved at the beginning of each migration interval to accommodate short-lived tensor in the interval. Doing this, Sentinel guarantees that there is always memory space for short-lived tensors (i.e., no competition from long-lived tensors), because the placement of short-lived tensors is critical for performance. Within a migration interval, the space can be dynamically shrunk to free space for long-lived tensors, when a memory page in the space is no longer needed by short-lived tensors.

The above method addresses the limitation of the existing methods that use a caching algorithm [17], [28], [29], [33] to decide tensor placement. Those methods move short-lived

tensors to slow memory, even though they are not accessed any more. This causes unnecessary tensor movement and wastes memory bandwidth. Furthermore, short-lived tensors unnecessarily stay longer in fast memory, wasting valuable space in fast memory. The above problem is caused by the fact that making the decision on the movement of short-lived tensors takes some time, due to the necessity of counting memory accesses to run the caching algorithm. Also, counting memory accesses for individual tensors can be inaccurate, because tensors with different memory access patterns share memory pages.

In our design, fast memory is always large enough to host short-lived tensors. If not, short-lived tensors will be frequently moved between fast and slow memories. This tensor movement is highly inefficient in terms of both performance and energy efficiency. Hence, we assume that the fast memory size is at least larger than the peak memory consumption of those short-lived tensors (discussed in Section IV-E). Since short-lived tensors are frequently allocated and freed and we reuse the same memory space to host them, the size of peak memory space for short-lived tensors is small, and typically bounded by a few GBs, making it feasible to host them in fast memory without consuming too much space.

D. Adaptive Layer-Based Migration

We migrate long-lived tensors because they are used sparsely and periodically. The tensor migration is controlled by the migration interval length which determines how frequently we migrate tensor between fast and slow memories. In particular, we partition a training step (i.e., one forward step plus one backward step) into equal-sized intervals, exemplified in Figure 4.

Tensor migration from slow to fast memory is triggered at the beginning of each interval, aiming to prefetching tensors needed by the next interval into fast memory before *the next interval* starts. Tensor migration follows a decreasing order of tensors in terms of number of memory accesses to each tensor, such that tensors with the largest number of memory accesses are migrated to fast memory first. The order information is available after data reorganization (Sec. IV-B). Following this order for migration allows Sentinel to make the best use of fast memory for high performance, in case certain tensors are left out in slow memory, which is discussed later. The tensor migration is overlapped with DNN training computation as much as possible, such that the overhead of tensor migration is removed from the critical path.

Tensor migration from fast to slow memory happens in the middle of the interval, when the long-lived tensor is no longer accessed by any operation in the interval. Such tensor migration is used to save fast memory space as much as possible, in order to accommodate upcoming tensor migration. We can know if a long-lived tensor will be used by any operation in an interval by using the profiling results.

We define the migration interval in terms of layers in DNN, not in terms of execution time, because of the following three reasons. First, the layer-based migration interval naturally

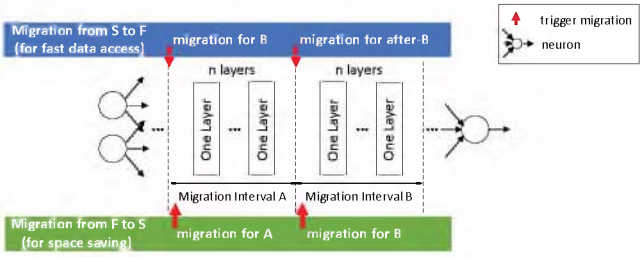

Fig. 4. Tensor migration based on the migration intervals. "S" and "F" stand for slow and fast memories respectively.

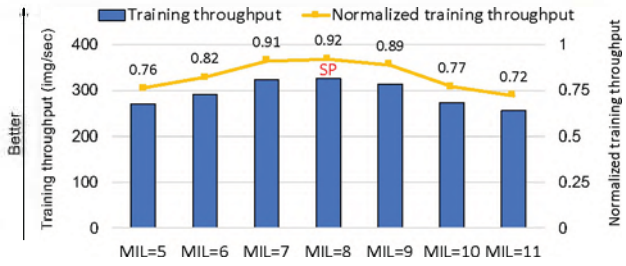

Fig. 5. Performance (training throughput) variance as we change the migration interval length (MIL). "SP" stands for sweet spot (the optimum migration interval length).

guarantees the completion of operations at the end of the interval, because no operation runs across layers. The time-based migration interval cannot guarantee that, and hence needs inevitable synchronization between application execution and tensor migration, causing performance loss. Second, each layer is associated with a computation phase with a memory access pattern (e.g., which tensors are accessed and their lifetime). The layer-based migration interval allows us to easily leverage the memory access patterns collected at the profiling phase to guide tensor migration. Third, the time-based migration imposes challenges on deciding which operations are executed in which migration interval, because of operation-level parallelism.

Determining an appropriate migration interval length is challenging. If the migration interval length is too long or too short, we cannot achieve the best performance. Figure 5 shows the performance when we use different interval lengths to train ResNet-32 on an Optane-based platform (shown in Table II). There is a 21% performance variance when we change the interval length from 5 to 11. When the interval length is 8, we achieve the best performance. Hence, determining an appropriate interval length is critical for performance.

We analyze the trade-off between long and short migration interval lengths as follows. If the interval length is long, then the tensors to migrate for an interval is large. The interval length cannot be too long. Otherwise the tensors to migrate can be larger than the available space in fast memory. This constraint on the interval length is the *space constraint*, formulated in Equation 1.

If the interval length is short, then the available execution

time to overlap tensor migration with application execution is short. The interval length cannot be too short. Otherwise, the tensor migration time is largely exposed to the critical path. We want to minimize the migration time exposed to the critical path, which is formulated in Equation 2.

In Equations 1 and 2, RS is the fast memory space for short-lived tensors, S is the fast memory size, and MIL stands for the migration interval length. RS is a function of the migration interval length (different migration interval lengths have different RS). In Equation 1, $Tensor$ is the size of tensors for migration in an interval; In Equation 2, BW is the migration bandwidth from slow to fast memory, and T is the DNN training time in an interval. $(S - RS(MIL))/BW$ is the tensors migration time. $Tensor$ and T are functions of the interval length (different interval lengths have different $Tensor$ and T).

$$\text{Space constraint:} \quad Tensor(MIL) < S - RS(MIL) \quad (1)$$

$$\text{Goal:} \quad \arg\min_{MIL}((S - RS(MIL))/BW - T(MIL)) \quad (2)$$

RS is relatively stable, according to our profiling results: There is only a small variance as we change MIL. Hence $S - RS(MIL)$ is near constant. $Tensor(MIL)$ and $T(MIL)$ are monotonically increasing functions of MIL (i.e., a larger MIL indicates larger $Tensor$ and T, and vice versa).

After collecting the profiling results, Sentinel uses Equation 1 to narrow down the search space of finding the optimum migration interval length; Sentinel then uses profiling results to estimate performance of using various interval lengths, based on which to determine the best one according to Equation 2. This exploration is quick, because it does not need to run any training operations, and the search space has only one dimension (i.e., the migration interval length). Due to the quick exploration and low-dimensional search space, using a statistical algorithm such as the genetic algorithm or Markov Chain Monte Carlo for multi-dimensional space as the existing work [8], [34] is not necessary.

We cannot use training steps to try every possible migration interval length to determine the best one without using performance modeling, because it raises concerns on runtime overhead, when the number of layers (and sub-layers that can be used as an interval) in a DNN model is very large.

We encounter three possible tensor migration cases at the end of a migration interval. We discuss them as follows. Assume that we have two intervals, A and B, and B is right after A. Sentinel migrates tensors at the beginning of A for B. At the end of A, we have three cases.

- Case 1: All tensor migration has been finished;
- Case 2: Tensor migration cannot finish, because of lack of space in fast memory;
- Case 3: Tensor migration cannot finish because of lack of time for migration (there is still space in fast memory).

In Case 1, once B starts, all of the migrated tensors are in fast memory, which is the ideal case. Case 2 can happen, even though the space constrain is respected for tensor migration for B, because some tensors in A may not be timely migrated from fast to slow memory to save space for B; Case 3 can happen, because the optimization goal ensures the migration time exposed to the critical path is minimized but the migration may not necessarily finish when B starts. We must avoid Cases 2 and 3 for the best performance. The migration interval length has impact on how often the three cases happen. Given a fast memory size, a short interval length can create more Case 3 while a long interval length can create more Case 2.

To avoid Case 2, long-lived tensors are immediately moved out of fast memory in the middle of A to save space, once the remaining operations in A do not need them. This solution prevents all occurrences of Case 2 in our evaluation.

To handle Case 3, we can either continue migration and let B wait for the migration completion, or leave tensors in slow memory. The continuation of tensor migration exposes tensor migration into critical path, but the execution of B uses tensors in fast memory; On the contrary, leaving tensors in slow memory uses the tensor in slow memory but avoids tensor migration overhead. This is a classic trade-off between data locality and data movement. To determine which method leads to the better performance, we use a test-and-trial algorithm.

In particular, whenever Case 3 happens at the end of an interval, we use one training step to try the continuation of tensor migration, and use another training step to try no-tensor-migration. We measure the performance of the two methods and use the better method in the remaining training steps.

The above algorithm does not cause significant runtime overhead, because Sentinel uses at most two training steps for test and trial to handle each occurrence of Case 3 and the number of occurrences is less than 10 in our evaluation, while the total number of training steps is easily millions. We quantify runtime overhead in Sections VII-B and VII-C.

E. Discussions

The lower bound on fast memory size. Although fast memory can be smaller with Sentinel, there is a lower bound on fast memory size to avoid significant performance loss. This lower bound is the peak memory consumption of short-lived tensors among all migration intervals plus the largest long-lived tensor. Smaller than this bound, the runtime system has to either frequently migrate short-lived tensors or has no space to accommodate long-lived tensors, which easily causes performance loss larger than 20%.

Handling dynamic graphs. Sentinel focuses on common DNN models with static graphs, similar to other work [7], [22]. Some frameworks, such as PyTorch and TensorFlow 2.0, support dynamic graphs. Depending on the input size within a batch, these frameworks generate a different dataflow graph with a right shape to accommodate the batch. Hence there could be multiple graphs. To handle dynamic graphs, the existing solution pads zeros at the end of input [35], such that batches have the same structure. This transforms a dynamic graph into a static one, but at the cost of larger memory footprint and unnecessary computation. We use a solution similar to [22] that uses bucketed profiling. In particular,

Sentinel bucketizes input sizes into a small number of buckets (at most 10). Input sizes in the same bucket have a similar graph. Sentinel profiles each bucket to decide tensor migration.

Handling control dependencies. A static graph can have control flow. Depending on input values in a batch, the graph can have different dataflow, causing different memory access patterns. Sentinel handles this by tracking dataflow. Whenever a new dataflow is encountered, Sentinel triggers profiling and makes migration decisions again.

Support of dynamic migration interval length. Sentinel uses the same length for all migration intervals. An alternative approach is to use different lengths for different intervals. Using such a dynamic interval length is helpful to avoid Cases 2 and 3. However, this method brings minimal performance benefit in practice, because Cases 2 and 3 do not happen often (see Table III). Also, determining appropriate dynamic migration interval lengths has to explore a large search space of migration interval length, causing larger runtime overhead.

V. Applying Sentinel to GPU

Sentinel can be applied to HM on CPU; With slight extension, it can also be applied to address memory capacity limitation on GPUs by treating GPU's global memory and CPU's main memory as fast and slow memories respectively. We name Sentinel for GPU, *Sentinel-GPU*.

Profiling method. GPU typically uses a proprietary driver that we cannot modify to trigger protection faults to enable tensor-level profiling as for CPU. Although there is an open-source GPU driver [36], it supports limited types of GPU and is not stable; Although recent GPU has a paging mechanism to trigger traditional page faults [37], it cannot be used to count memory accesses on GPU, once pages are loaded into GPU memory; A binary instrumentation tool such as NVBit [38] or compiler tool can instrument load/store instructions but cannot directly measure main memory accesses. Hence, there is no tool to count memory accesses at page or tensor level for GPU. Also, introducing new hardware counters to collect page access statistics is possible, but hardware modification is expensive and unscalable.

To address the above problem, we use a customized pinned memory mechanism to enable tensor-level profiling for GPU. The traditional pinned memory mechanism allows GPU to access pages resident on CPU memory. By allocating tensors on pinned memory on CPU, we can use the existing profiling mechanism in Sentinel to count memory accesses from GPU. In particular, whenever GPU accesses a pinned memory page on CPU, a protection fault is triggered on CPU and handled by the fault handler in Sentinel to count it. Using the above method, we do not lose accuracy of counting memory accesses on GPU, because protection faults are caused by memory accesses on GPU, not on CPU.

However, implementing the above idea faces a challenge. The traditional pinned memory mechanism disables paging, such that when GPU accesses a page resident on CPU memory, the page is guarantee to be there. The implementation of this mechanism includes using the system call *mlock()* to lock PTE. As a result, Sentinel cannot modify the specific bit (bit 51) to trigger protection fault.

To address the above problem, we customize the pinned memory mechanism. In particular, Sentinel intercepts *mlock()* and bypasses it. To disable paging to ensure the correctness of the pinned memory mechanism, Sentinel temporarily disables OS-level page swapping mechanisms during the profiling using the existing system calls. This does not lock PTE, and hence allows Sentinel to set the bit to trigger protection faults.

Sentinel uses the above customized mechanism during the profiling, but must revert to the traditional GPU memory allocation and accesses in TensorFlow to avoid expensive CPU memory accesses. This reversion is feasible for those tensors that are repeatedly allocated and freed across training steps, but not possible for a few tensors that are allocated before all training steps and freed after them. For each of those tensors, Sentinel creates two copies, one using pinned memory and accessed during profiling, while the other using the traditional GPU memory allocation and used after profiling. Creating two copies does not require the user to change the implementation of DNN training, because it can be done by pointer switch through the runtime implementation. The two copies need to be synchronized after profiling to ensure the remaining training uses the most updated tensors. This synchronization overhead is paid in only one training step and ignorable in the whole training. We quantify this overhead in Section VII-C.

Handling Case 3. In Case 3, tensor migration cannot finish in time, because of lack of time for migration. A possible solution to handle this case on CPU-based HM is to leave tensors in slow memory on CPU. On GPU, however, the tensors must be placed on GPU memory when GPU accesses them (accessing CPU memory is too slow). Hence, there is no need to use the test-and-trial algorithm to handle Case 3. Handling this case must wait for tensor migration to complete, but subject to the optimization goal in Equation 2.

VI. Implementation

We implement Sentinel in Linux v5.6.0 and TensorFlow v1.14. We change OS kernel for memory profiling; We change the TensorFlow runtime system for page migration (Figure 6). Sentinel introduces three APIs to trigger/stop memory profiling and identify DNN layers, which are *start_profile()*, *end_profile*(), and *add_layer()*. *start_profile()* triggers a system call to enable tracking of main memory accesses, memory (de)allocation to record lifetime of tensors. *add_layer()*, placed at the end of each layer, informs the runtime of where is each layer to determine migration intervals. Adding *start_profile()* and *end_profile()* includes only two lines of changes to the DNN model. Adding *add_layer()* includes 10-100 lines, depending on how many layers there are in a DNN model.

Figure 6 shows implementation details. Sentinel skips the first 10 training steps used by TenorFlow to detect hardware configurations, and uses the 11th for profiling. During the profiling, Sentinel collects memory access and lifetime information for each tensor from OS and TensorFlow respectively. After the profiling phase, Sentinel uses three helper threads:

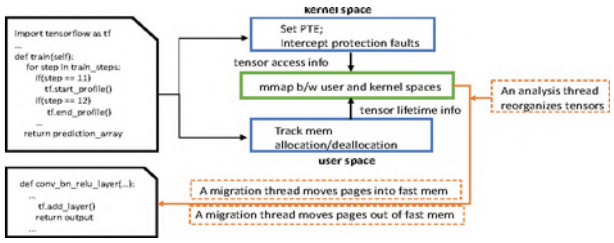

Fig. 6. Implementation overview.

one for information analysis to determine migration intervals and make migration decision, one for data migration from fast to slow memory, and one for migration in the opposite way. The two migration threads work in parallel to accelerate migration. Sentinel uses the Linux system call *move_pages()* to migrate pages. Sentinel extends TensorFlow memory allocation and free functions by adding the customized data reorganization policy. Before the training happens, tensor are allocated in slow memory. After collecting the profiling results, Sentinel manages tensor allocation and migration.

GPU implementation. Similar to Sentinel, Sentinel-GPU leverages the APIs to enable online profiling and tensor management. To manipulate tensor allocation, Sentinel-GPU intercepts TensorFlow GPU memory allocators (such as *AllocateRaw* and *gpu_bfc_Allocator*), similar to [9]. Sentinel-GPU replaces those allocators with the customized ones for pinned memory control or tensor collocation. *add_layer()* is implemented as a CUDA kernel to execute at the end of each DNN layer. For short-lived tensors, Sentinel-GPU manages a memory pool allocated by *cudaMalloc* and enforces data reorganization. For long-lived tensors, Sentinel-GPU triggers bi-direction tensor movement by using CUDA events and streams. In particular, Sentinel uses two CUDA streams: one for computation and the other for tensor movement. Sentinel inserts tensor movement events into the stream based on the decision on the migration intervals. Sentinel-GPU achieves asynchronous tensor movement with *cudaMemPrefetchAsync*.

VII. Experimental Results

A. Experimental setup

We evaluate Sentinel on two HM platforms. One, named Optane-based HM, uses DRAM and Intel Optane DC persistent memory (PMM) as fast and slow memories respectively on CPU; The other, named GPU-based HM, treats GPU global memory and CPU main memory as fast and slow memories respectively. Table II gives details. PMM has two operating modes, *Memory Mode* and *App-direct Mode*. In Memory Mode, DRAM works as a hardware-managed cache to PMM. Running the application in this mode does not require modifications to the application. App-direct Mode allows programmer to explicitly control memory accesses to PMM and DRAM. Sentinel works in App-direct Mode and beats Memory Mode for large model training.

TABLE II
Hardware overview of experimental system.

Optane-based HM	
CPU	An Intel Xeon Gold 6252 CPU @2.30GHz
Last Level Cache	36608KB
Fast Memory	DDR4 DIMM: 96GB
Slow Memory	Optane DC PMM: 756GB
GPU-based HM	
GPU	Nvidia V100 with 16GB with 15.75 GB of GDDR6
CPU	Intel(R) Xeon(R) E5-2670 with 128 GB of DDR4
Interconnect	PCIe 3.0×16

TABLE III
DNN model for evaluation."Sen.", stands for Sentinel.

DNN Model	Dataset (Batchsize)	Peak men. (GB)		# of steps used in Sen.	
		w/o Sen.	w/ Sen.	profiling	test & trial
ResNet-32	CIFAR-10 (128)	14.10	14.19	1	3
BERT-large	CoLA (32)	35.39	35.51	1	3
LSTM	PTB(20)	11.12	11.25	1	0
DCGAN	MNIST (128)	15.68	15.79	1	3
MobileNet	CIFAR-10 (128)	14.15	14.26	1	1
ResNet-200	CIFAR-10 (4K)	99.73	100.01	1	7
BERT-large	CoLA (128)	133.59	133.90	1	3
LSTM	PTB (4K)	29.97	30.09	1	0
DCGAN	celebA (10K)	115.10	115.23	1	7
MobileNet	CIFAR-100 (4K)	142.07	142.59	1	7

We evaluate five DNN models with small and large batch sizes (Table III). We use the implementations of LSTM and MobileNet from TensorFlow [39], ResNet from [40], Bert from [41], and DCGAN from [42]. We use the default precision setting (FP32 or FP16) for floating point numbers. We report training throughput when the execution time per step becomes constant (usually after the first couple of steps).

B. Sentinel on Optane-based HM

Evaluation methodology. We compare Sentinel with a state-of-the-art memory management solution for HM on CPU [19]. This solution is based on a FIFO-based active list, and we name it *improved active list* (IAL). We also compare Sentinel with AutoTM [7]. AutoTM uses Integer Linear Programming (ILP) to decide tensor movement and placement based on static profiling and nGraph compiler. To enable fair comparison, we implement the AutoTM's memory management solution in TensorFlow. We compare Sentinel with IAL and AutoTM using the same fast memory size, which is 20% of the peak memory consumption of DNN models. This setting follows previous work [7], [19]. In addition, we compare Sentinel with the default NUMA allocation policy (first-touch NUMA) and Memory Mode. In our platform, DRAM and PMM belong to two NUMA nodes.

Overall performance. Figure 7 shows performance of IAL, AutoTM and Sentinel normalized by that of slow memory-only system. We use DNN models with small batch sizes for evaluation, because IAL code cannot work well for large batch sizes, either due to segfault or more than 10x performance slowdown. The figure shows that performance difference between Sentinel and fast memory-only system (shown as the red horizontal line in the figure) is very small (no difference in

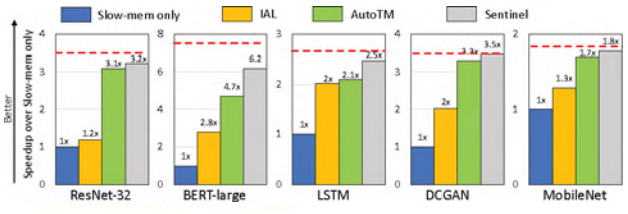

Fig. 7. Performance speedup of IAL, AutoTM and Sentinel over slow-memory only. The red horizontal line shows performance of fast-memory only. Performance is normalized by that of slow-memory only.

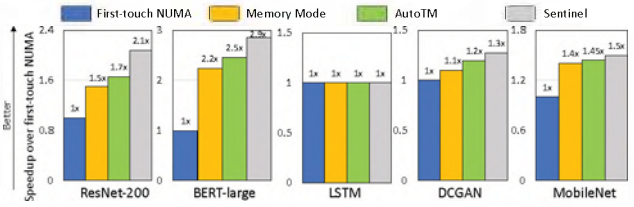

Fig. 8. Performance with first-touch NUMA, Memory Mode, AutoTM and Sentinel, normalized by that of first-touch NUMA.

DCGAN, and 9% difference on average), while IAL has 46% performance difference on average. Sentinel outperforms IAL by 37% on average (up to 56%). Sentinel outperforms AutoTM by 17% on average (up to 31%) because of the following reasons. First, all tensor movements in AutoTM between fast and slow memories are exposed to the critical path, which incurs runtime overhead. Second, AutoTM uses static profiling to conclude that the output of an operation should be placed into slow memory with negligible performance impact. This conclusion is not true when the output is large. Sentinel uses dynamic profiling and attempts to put all tensors needed by the upcoming operations into fast memory, hence avoiding the performance problem.

Table IV reports the total size of migrated tensors in one training step using IAL, AutoTM and Sentinel. Sentinel has 85% and 32% more migrations (on average) than IAL and AutoTM respectively. Frequent migrations allow Sentinel to make best use of fast memory for performance. Those migrations are overlapped with training to hide overhead.

Figure 8 shows performance with large batch sizes for first-touch NUMA, Memory Mode, AutoTM, and Sentinel. Training with large batch sizes consumes large memory consumption (Table III), creating challenges on data management in HM. The results are normalized by performance of first-touch NUMA. The results show that for the models whose peak memory consumption is larger than fast memory (e.g., ResNet200, BERT_large, DCGAN and MobileNet), Sentinel outperforms first-touch NUMA, Memory Mode and AutoTM by 1.7x, 1.2x and 1.1x (on average) respectively. For the models whose peak memory consumption is less than the fast memory size (LSTM), Sentinel has the same performance as first-touch NUMA, Memory Mode and AutoTM. In this case, DRAM is large enough to hold all tensors. This case shows ignorable overhead of Sentinel.

TABLE IV
TOTAL SIZE OF MIGRATED TENSORS IN ONE TRAINING STEP.

	ResNet	BERT	DCGAN	LSTM	MobileNet
IAL	3.1GB	2.8GB	0.8GB	0.7GB	0.55GB
AutoTM	5.1GB	2.3GB	0.7GB	1.2GB	0.8GB
Sentinel	8GB	4GB	1.2GB	1.2GB	0.95GB

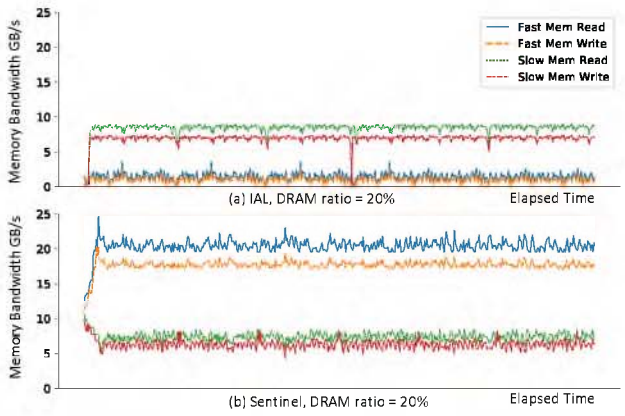

Fig. 9. Memory access bandwidth during training of ResNet-32.

Memory bandwidth. We analyze memory bandwidth consumption in IAL and Sentinel, shown in Figure 9. Compared with IAL, Sentinel consumes much higher (7.3x on average) memory bandwidth in fast memory, indicating that fast memory accesses happen much more often in Sentinel than in IAL. Sentinel also has lower memory bandwidth consumption in slow memory, compared with IAL, indicating that Sentinel reduces accesses in slow memory.

Runtime overhead. Table III shows total number of training steps used for profiling and test-and-trial. Those steps have longer execution time than regular training steps, hence introducing runtime overhead. On average, Sentinel uses only 1.8 steps. Each of those steps is extended by up to 5x in terms of execution time. However, such overhead is amortized by millions of steps. As a result, the runtime overhead of Sentinel is negligible (less than 1%).

Memory overhead. Using Sentinel for tensor-level profiling increases peak memory consumption, causing memory overhead. Table III shows peak memory consumption. Sentinel does not increase peak memory consumption much (by 2.4% at most). This is because tensors larger than one page dominate total memory consumption. Profiling those tensors with Sentinel does not cause memory overhead.

Sensitivity study. We change fast memory size and measure performance with small batches. Figure 10 shows the results. In general, larger fast memory gives better performance. When the fast memory size is 60% of peak memory consumption, all of DNN models on HM with Sentinel do not have any performance difference from the fast memory-only system. Also, with Sentinel, performance is not sensitive to fast memory size: There is at most 17% performance variance when fast memory size is changed from 20% to 40% of peak memory

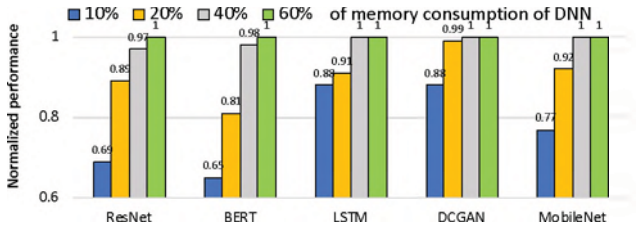

Fig. 10. Performance with Sentinel under various sizes of fast memory. The fast memory size is shown as the percentage of peak memory consumption of DNN models. Performance is normalized by that of the fast memory-only.

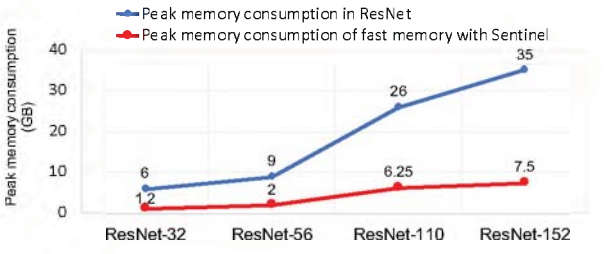

Fig. 11. Comparison between peak memory consumption of DNN models and fast memory size for ResNet variants.

consumption. This demonstrates how Sentinel effectively uses tensor movement to make the best use of fast memory.

Saving fast memory size. Figure 7 shows using 20% of peak memory consumption of DNN models as fast memory size, Sentinel on HM has similar performance (9% difference on average) as the fast memory-only. This brings 80% saving in fast memory. Figure 10 shows using 60% of peak memory consumption as fast memory size, no performance loss.

To further study Sentinel's effectiveness, we use ResNet with various topology and peak memory consumption. We report the minimum fast memory size with which Sentinel performs the same as the fast memory-only. Figure 11 shows peak memory consumption and fast memory size for all ResNet variants. The figure shows that although peak memory consumption increases quickly as ResNet becomes more complicated, the fast memory size increases in a much slower rate because of adaptive layer-based migration. This demonstrates the effectiveness of using Sentinel to save fast memory size.

C. Sentinel on GPU-based HM

Evaluation methodology. We use Nvidia Tesla V100 GPU shown in Table II with CUDA v10.1 for evaluation. We compare Sentinel-GPU with five existing work on GPU, including Unified Memory (UM) [37], vDNN [6], AutoTM [7], SwapAdvisor [8] and Capuchin [9]. UM automatically moves tensors from CPU to GPU in the event of a GPU page fault, and moves least-used pages from GPU to CPU. vDNN is a solution using GPU-based HM for DNN training. vDNN focuses on convolution layers and migrates input tensors of convolution layers between CPU and GPU memories; vDNN tries to overlap the migration of the input tensors with convolution computation. AutoTM works for both CPU and GPU, and uses ILP to decide tensor movement and placement. We implemented asynchronous tensor migration in AutoTM as described in [7]. SwapAdvisor uses the Generic Algorithm (GA) to find a good combination of memory allocation and operation scheduling on MXNet [43]. Capuchin is a state-of-the-art solution using GPU-based HM for DNN training. Capuchin overlaps tensor movement with training. When the tensor movement overhead is too large to be overlapped, Capuchin discards tensors and recomputes them when needed to save GPU memory. vDNN, SwapAdvisor, AutoTM and Capuchin are either close-sourced or not implemented on TensorFlow. We implement their migration strategies in TensorFlow.

Profiling method. We make the comparison in terms of profiling method. UM does not use any profiling, and uses on-demand tensor movement. As a result, UM causes large runtime overhead because most of tensor movement is exposed to the critical path. vDNN does not use any profiling, but heavily rely on domain knowledge to decide tensor migration. As a result, vDNN only works for specific models (feedforward CNN models) and cannot handle recursive structures in DNN models. AutoTM collects execution time of individual operations at compilation time. Such static profiling method misleads tensor migration, when the batch size or hardware used in production is different from the ones used in static profiling. SwapAdvisor uses a lot of training steps for dynamic profiling. Because of the GA algorithm SwapAdvisor uses to decide tensor migration, SwapAdvisor suggests to use 30 minutes to make the final migration decision at runtime [8]; According to our experimental results, for a large model such as BERT-large, SwapAdvisor cannot make the decision within 30 minutes. This decision process is too slow for some model training (e.g., NLP fine-tuning which can take less than two hours [41]). Capuchin and Sentinel use dynamic profiling, whose overhead is negligible (a few seconds and less than 1%).

Maximum batch size. We compare vDNN, AutoTM, SwapAdvisor, Capuchin and Sentinel-GPU in terms of maximum batch size each solution can achieve, given the same GPU memory capacity. This comparison aims to show how effectively these solutions save GPU memory. We do not evaluate UM, because its maximum batch size is limited by CPU memory and can be much larger than that with vDNN and Sentinel-GPU, but with much worse performance (shown in Figure 12). Table V shows the result. The result for "TensorFlow" in Table V is collected without using tensor migration. Compared with TensorFlow without tensor migration, Sentinel-GPU increases batch size by 4.18x on average. vDNN is designed for feedforward CNN models and cannot handle recursive structures in a DNN graph. Hence it cannot work for LSTM and BERT-large. For CNN models, Sentinel-GPU outperforms vDNN by 1.9x. This is because Sentinel-GPU migrates tensors as many as possible to save GPU memory, whereas vDNN only focuses on input tensors of the convolution layers. Sentinel-GPU outperforms SwapAdvisor by 1.1x. This is because SwapAdvisor aims to minimize training time instead of minimizing memory consumption of

tensors in GPU. AutoTM, Capuchin and Sentinel-GPU achieve a comparable maximum batch size, because all of them try to migrate tensors out of GPU memory as much as possible. They differ in training throughput, discussed as follows.

Training throughput. For each model, we use three batch sizes, shown in Figure 12. Throughput in Figure 12 is normalized by that of UM. With the largest batch size shown in Figure 12, Figure 13 shows the performance for two components (migration overhead in the critical path and recomputation) in one training step for deeper analysis, and percentage numbers on top of bars in Figure 13 are the ratio in terms of execution time of one training step. For Sentinel-GPU, Figure 13 shows performance breakdown results to quantify the contributions of various techniques in Sentinel-GPU. "Direct tensor migration" does not use migration interval and trigger tensor migration simply based on tensor forthcoming usage; It does not reserve space for short-lived tensors; "w/ det. MI" uses an optimal migration interval length but without space reservation; "w/all" is the full featured Sentinel.

UM vs. Sentinel-GPU. Sentinel-GPU has 1.1x-7.8x higher throughput than UM. Such a large performance gain comes from effectively prefetching tensors from CPU to GPU and reduction of migration overhead in Sentinel-GPU.

vDNN vs. Sentinel-GPU. For CNN models, Sentinel-GPU outperforms vDNN by 2x. Similar to Sentinel-GPU, vDNN tries to overlap tensor movement with computation. However, vDNN does not consider time difference between layers, which exposes most of tensor migration overhead to critical path (3x more than with Sentinel-GPU).

SwapAdvisor vs. Sentinel-GPU. Sentinel-GPU outperforms SwapAdvisor by 65% on average (up to 110%). The GA in SwapAdvisor is too slow (more than 30 minutes) to find an optimal solution for tensor placement and migration. The process of finding the solution causes slowdown; The tensor migration overhead in SwapAdvisor is 81% larger than that in Sentinel-GPU, because SwapAdvisor fails to hide that.

AutoTM vs. Sentinel-GPU. Sentinel-GPU outperforms AutoTM by 17% on average (up to 29%). Sentinel-GPU with the space reservation reduces migration overhead by 8% of training time, compared with AutoTM, because this technique avoids unnecessary tensor movement while AutoTM exposes tensor movement into the critical path. Avoiding page-level false sharing in Sentinel-GPU contributes additional 9% benefit over AutoTM.

Capuchin vs. Sentinel-GPU. Sentinel-GPU outperforms Capuchin by 16% on average (up to 21%). Because of the pervasiveness of page-level false sharing, Sentinel-GPU improves performance by 11% - 21%. In Capuchin, recomputation takes about 11% of the training time while Sentinel-GPU does not have recomputation overhead. As a result, although the migration time in Capuchin is shorter than in Sentinel-GPU, the net effect is that Sentinel-GPU outperforms Capuchin.

VIII. Related Work

Comparison with recent efforts on using HM for DNN training. We review and evaluate them in Sections II and VII.

TABLE V
MAXIMUM BATCH SIZE WITH VDNN, AUTOTM, SWAPADVISOR, CAPUCHIN AND SENTINEL-GPU

	ResNet-200 (CIFAR10)	BERT-large (CoLA)	LSTM (PTB)	DCGAN (celebA)	MobileNet (CIFAR100)
TensorFlow	1K	5	800	0.6K	0.8K
vDNN	4.2K	not work	not work	1.4K	1.2K
AutoTM	5.6K	27	1.4K	2.5K	3.2K
SwapAdvisor	5.4K	25	1.2K	2.4K	3.1K
Capuchin	5.9K	27	1.4K	2.7K	3.2K
Sentinel-GPU	5.7K	28	1.5K	2.5K	3.2K

Using Managed Runtime for Data Management on HM. Existing efforts [32], [44]–[46] leverage managed runtime such as JVM. There are two differences between them and Sentinel. (1) The existing efforts couple data migration with garbage collection, and hence miss opportunities to minimize data migration overhead; (2) The existing efforts do not proactively migrate data objects to save fast memory space.

Page-based Runtime Data Management on HM. Existing proposals [19], [27]–[31], [47], [48] explore various page placement polices based on memory access profiling. Some works [27]–[29] track page accesses by setting and resetting PTE as Sentinel does, but this tracking mechanism incurs high runtime overhead. Unlike the above work, Sentinel leverages DNN domain knowledge, and hence only profiles a small portion of total execution (one training step) without paying large runtime overhead and losing accuracy. Also, Sentinel associates page-level profiling results with tensors, making profiling results more meaningful for tensor migration.

IX. Conclusions

Training DNN faces a problem on memory capacity. This paper focuses on how to use HM to address this problem without losing training throughput while saving fast memory. We introduce a runtime system (Sentinel) based on a unique and comprehensive performance study on all tensors in various linear and nonlinear models. The runtime system is featured with a novel tensor-level profiling method and runtime techniques to improve tensor migration efficiency for high performance and saving fast memory capacity. Evaluating on Optane-based HM and CPU-GPU-based HM, we show Sentinel outperforms seven software- and hardware-based solutions.

Acknowledgment

We thank anonymous reviewers for their instructive comments. This work was partially supported by U.S. National Science Foundation (CNS-1617967, CCF-1553645 and CCF-1718194).

References

[1] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. V. Le, G. E. Hinton, and J. Dean, "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer," *CoRR*, vol. abs/1701.06538, 2017.

[2] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

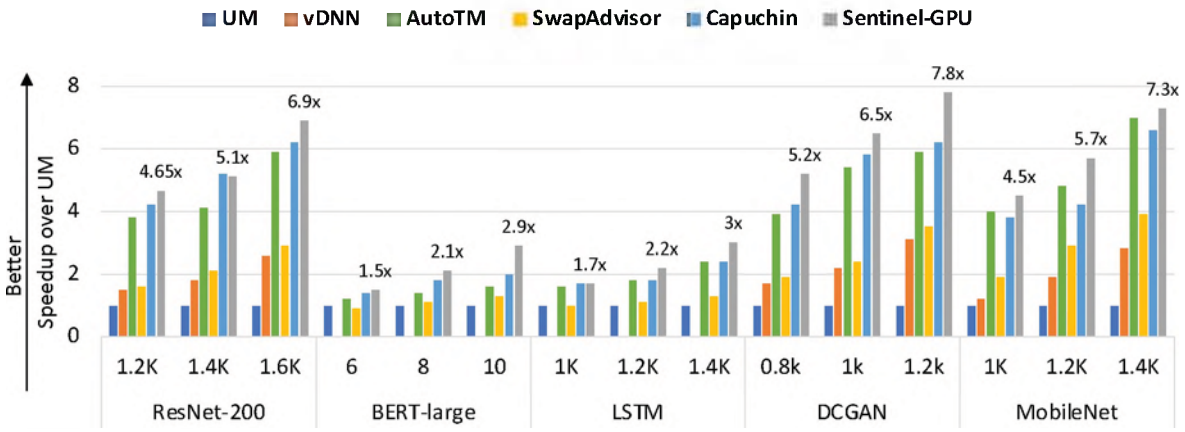

Fig. 12. Performance of UM, vDNN, AutoTM, SwapAdvisor, and Capuchin and Sentinel-GPU, normalized by that of UM. vDNN cannot work for BERT and LSTM.

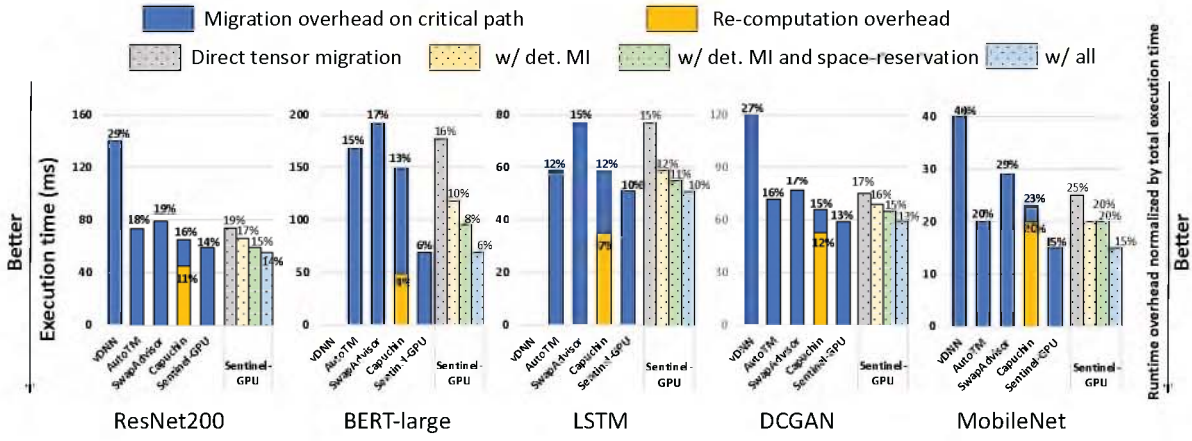

Fig. 13. Performance breakdown for vDNN, AutoTM, SwapAdvisor, Capuchin and Sentinel-GPU. "det. MI" stands for "determine an appropriate migration interval length". Percentage numbers on top of bars are the ratio in terms of execution time of one training step.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[5] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "Superneurons: Dynamic gpu memory management for training deep neural networks," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, (New York, NY, USA), p. 41–53, Association for Computing Machinery, 2018.

[6] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, 2016.

[7] M. Hildebrand, J. Khan, S. Trika, J. Lowe-Power, and V. Akella, "Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020.

[8] C.-C. Huang, G. Jin, and J. Li, "Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, (New York, NY, USA), p. 1341–1355, Association for Computing Machinery, 2020.

[9] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian, "Capuchin: Tensor-based gpu memory management for deep learning," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, (New York, NY, USA), p. 891–905, Association for Computing Machinery, 2020.

[10] T. D. Le, H. Imai, Y. Negishi, and K. Kawachiya, "TFLMS: Large model support in tensorflow by graph rewriting," in *arXiv preprint arXiv:1807.02037*, 2018.

[11] C. Meng, M. Sun, J. Yang, M. Qiu, and Y. Gu, "Training deeper models by GPU memory optimization on TensorFlow," in *ML Systems Workshop in NIPS*, 2017.

[12] H. Jin, B. Liu, W. Jiang, Y. Ma, X. Shi, B. He, and S. Zhao, "Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures," *ACM Trans. Archit. Code Optim.*, vol. 15, Sept. 2018.

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

[14] J. Zhang, S. Yeung, Y. Shu, B. He, and W. Wang, "Efficient memory management for gpu-based deep learning systems," *CoRR*, vol. abs/1903.06631, 2019.

[15] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), pp. 331–345, ACM, 2019.

[16] Intel, "Big Memory Breakthrough for Your Biggest Data Challenges." https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[17] L. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid

memory systems," in *Proc. Int. Conf. Supercomputing (ICS '11)*, 2011.
[18] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu, "Row buffer locality aware caching policies for hybrid memories," in *Proc. IEEE 2012 30th Int. Conf. Computer Design (ICCD '12)*, 2012.
[19] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, 2019.
[20] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 655–668, Oct 2018.
[21] J. Liu, D. Li, G. Kestor, and J. S. Vetter, "Runtime Concurrency Control and Operation Scheduling for High Performance Neural Network Training," in *International Parallel and Distributed Processing Symposium*, 2019.
[22] M. Sivathanu, T. Chugh, S. S. Singapuram, and L. Zhou, "Astra: Exploiting Predictability to Optimize Deep Learning," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.
[23] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015.
[24] "Pytorch." https://pytorch.org/, 2019.
[25] M. Han, J. Hyun, S. Park, and W. Baek, "Hotness- and Lifetime-Aware Data Placement and Migration for High-Performance Deep Learning on Heterogeneous Memory Systems," *IEEE Transactions on Computers*, vol. 69, no. 3.
[26] Google, "tcmalloc." https://github.com/google/tcmalloc.
[27] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pp. 631–644, 2017.
[28] T. Hirofuchi and R. Takano, "Raminate: Hypervisor-based virtualization for hybrid main memory systems," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, (New York, NY, USA), pp. 112–125, ACM, 2016.
[29] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, "Heteroos — os design for heterogeneous memory management in datacenter," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 521–534, June 2017.
[30] K. Wu, J. Ren, and D. Li, "Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2018.
[31] K. Wu, Y. Huang, and D. Li, "Unimem: Runtime Data Management on Non-volatile Memory-based Heterogeneous Main Memory," in *SC*, 2017.
[32] C. Wang, H. Cui, T. Cao, J. Zigman, H. Volos, O. Mutlu, F. Lv, X. Feng, and G. H. Xu, "Panthera: Holistic memory management for big data processing over hybrid memories," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, 2019.
[33] W. Zhang and T. Li, "Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.
[34] Z. Jia, M. Zaharia, and A. Aiken, "Beyond Data and Model Parallelism for Deep Neural Networks," in *SysML Conferenc*, 2019.
[35] Google, "Tensorflow Bucketing." https://www.tensorflow.org/versions/r0.12/, 2017.
[36] Nvidia, "Nouveau: Accelerated Open Source driver for nVidia cards." https://nouveau.freedesktop.org/wiki/, 2019.
[37] Nvidia, "Unified Memory." https://devblogs.nvidia.com/unified-memory-in-cuda-6/, 2019.
[38] O. Villa, M. Stephenson, D. Nellans, and S. Keckler, "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs," in *IEEE/ACM International Symposium on Microarchitecture*, 2019.
[39] "TensorFlow models." https://github.com/tensorflow/models, 2019.
[40] "A TensorFlow Implementation of Deep Convolutional Generative Adversarial Networks." https://github.com/carpedm20/DCGAN-tensorflow, 2018.
[41] "TensorFlow code and pre-trained models for BERT." https://github.com/google-research/bert, 2019.
[42] "ResNet in TensorFlow." https://github.com/wenxinxu/resnet-in-tensorflow, 2017.
[43] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015.
[44] S. Akram, J. B. Sartor, K. S. McKinley, and L. Eeckhout, "Write-rationing garbage collection for hybrid memories," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, 2018.
[45] M. Wu, Z. Zhao, H. Li, H. Li, H. Chen, B. Zang, and H. Guan, "Espresso: Brewing java for more non-volatility with non-volatile memory," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, 2018.
[46] S. Akram, J. B. Sartor, K. S. McKinley, and L. Eeckhout, "Crystal gazer: Profile-driven write-rationing garbage collection for hybrid memories," in *Abstracts of the 2019 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '19, 2019.
[47] J. Liu, J. Ren, R. Gioiosa, D. Li, and J. Li, "Sparta: High-performance, element-wise sparse tensor contraction on heterogeneous memory," in *Proceedings of the 26rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, 2021.
[48] J. Ren, M. Zhang, and D. Li, "Hm-ann: Efficient billion-point nearest neighbor search on heterogeneous memory," in *34th Conference on Neural Information Processing Systems (NeurIPS 2020)*, November 2020.