DeepTM: Efficient Tensor Management in Heterogeneous Memory for DNN Training

Haoran Zhou, Wei Rang, Hongyang Chen, *Senior Member, IEEE*, Xiaobo Zhou, *Senior Member, IEEE*, and Dazhao Cheng

***Abstract*—Deep Neural Networks (DNNs) have gained widespread adoption in diverse fields, including image classification, object detection, and natural language processing. However, training large-scale DNN models often encounters significant memory bottlenecks, which ask for efficient management of extensive tensors. Heterogeneous memory system, which combines persistent memory (PM) modules with traditional DRAM, offers an economically viable solution to address tensor management challenges during DNN training. However, existing memory management methods on heterogeneous memory systems often lead to low PM access efficiency, low bandwidth utilization, and incomplete analysis of model characteristics. To overcome these hurdles, we introduce an efficient tensor management approach, DeepTM, tailored for heterogeneous memory to alleviate memory bottlenecks during DNN training. DeepTM employs page-level tensor aggregation to enhance PM read and write performance and executes contiguous page migration to increase memory bandwidth. Through an analysis of tensor access patterns and model characteristics, we quantify the overall performance and transform the performance optimization problem into the framework of Integer Linear Programming. Additionally, we achieve tensor heat recognition by dynamically adjusting the weights of four key tensor characteristics and develop a global optimization strategy using Deep Reinforcement Learning. To validate the efficacy of our approach, we implement and evaluate DeepTM, utilizing the TensorFlow framework running on a PM-based heterogeneous memory system. The experimental results demonstrate that DeepTM achieves performance improvements of up to 36% and 49% compared to the current state-of-the-art memory management strategies AutoTM and Sentinel, respectively. Furthermore, our solution reduces the overhead by 18 times and achieves up to 29% cost reduction compared to AutoTM.**

***Index Terms*—Deep neural network training, heterogeneous memory, memory management, performance optimization.**

I. Introduction

Deep Neural Networks (DNNs) [7] have gained widespread recognition and application in both academic research and various industrial domains, spanning tasks such as image classification [1], object detection [2], and natural language processing [3]. To train more accurate models [11], DNNs require more input data [8], training parameters, and larger batch sizes [24]. However, this pursuit of complexity introduces a heightened performance loss caused by memory tensions. Intel Optane$^{\text{TM}}$ persistent memory (PM) [45] combines high capacity with cost-effectiveness but at the expense of slower bandwidth and increased I/O latency compared to traditional DRAM [10]. Typically, PM is integrated into heterogeneous memory systems as an extension of DRAM. In this configuration, PM expands the memory capacity, providing partial relief for memory-intensive tasks. Heterogeneous memory systems have thus become a valuable solution for addressing challenges in DNN training.

Motivation: In DNN training, two significant challenges arise in the realm of heterogeneous memory management. First, persistent memory (PM) presents inherent limitations, including lower bandwidth and increased access latency compared to DRAM. Recent research indicates that PM exhibits over ten times lower write bandwidth compared to DRAM, with a notable disparity in random access [22], [47]. In our investigation, we observe that PM exhibits a 30.6 times higher write latency compared to DRAM at the page level. Furthermore, the utilization of both DRAM and PM's bandwidth is hindered when copying small buffers due to latencies associated with thread creation and address mapping. Second, there is a clear demand for more efficient tensor swapping strategies in heterogeneous memory. Existing heuristic algorithms, often focused on analyzing single tensor characteristics, struggle to adapt to the diverse requirements of different DNN models. Attempting to accommodate various conditions and characteristics can significantly increases the algorithm complexity. Additionally, the global optimal solution for tensor management is known to be NP-hard, and practical approximations often rely on mathematical tools [18]. Moreover, real-time monitoring of the DNN model and heterogeneous memory system during training is crucial to prevent out-of-memory (OOM) errors.

Manuscript received 10 November 2023; revised 10 July 2024; accepted 17 July 2024. Date of publication 22 July 2024; date of current version 9 September 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFE0205700, in part by the National Natural Science Foundation of China under Grant 62341410, in part by Science and Technology Development Fund, Macao S.A.R (FDCT) under Grant 0078/2023/AMJ, in part by Zhejiang Lab Open Research Project under Grant K-2022PI0AB01, and in part the Special Fund of Hubei Luojia Laboratory under Grant 220-100016. Recommended for acceptance by B. Nicolae. *(Corresponding author: Dazhao Cheng.)*

Haoran Zhou is with the School of Computer Science, Wuhan University, Wuhan, Hubei 430072, China, and also with the Laboratory of Internet of Things for Smart City, University of Macau, Macau 999078, China (e-mail: zhrzhr@whu.edu.cn).

Wei Rang is with the School of information science and engineering, Shandong Normal University, Jinan, Shandong 250098, China (e-mail: wrang@sdnu.edu.cn).

Hongyang Chen is with the Zhejiang Lab, Hangzhou, Zhejiang 311500, China (e-mail: hongyang@zhejianglab.com).

Xiaobo Zhou is with the Laboratory of Internet of Things for Smart City and Department of Computer and Information Science, University of Macau, Macau 999078, China (e-mail: waynexzhou@um.edu.mo).

Dazhao Cheng is with the School of Computer Science, Wuhan University, Wuhan, Hubei 430072, China (e-mail: dcheng@whu.edu.cn).

Digital Object Identifier 10.1109/TPDS.2024.3431910

Limitation of state-of-the-art approaches: While numerous efforts have been made to tackle heterogeneous memory management challenges, they often exhibit specific limitations.

1) *PM Characterization and Analysis:* Previous works have integrated PM into various applications such as database and deep learning [22], [52]. NVSL [50] and BjÃ¶rn Daase et al. [46] only analyze the basic bandwidth and random/sequential access latency of PM in multi-threaded scenarios. However, they fall short in mitigating the high latency of multiple PM accesses and the low bandwidth of PM read and write. Furthermore, these efforts have often overlooked the challenge of optimizing the actual bandwidth utilization of heterogeneous memory.

2) *Analysis of Application Characteristics:* Existing approaches tend to consider only a limited set of characteristics when making the replacement for hot and cold data in heterogeneous memory systems [18], [29]. For example, SwapAdvisor [18] and Capuchin [29] only focus on the size of tensors and only deploy a local greedy strategy. More importantly, these approaches primarily rely on static characteristic analyses, failing to adapt to the varying requirements of individual kernels in deep learning scenarios. For instance, Sentinel [26] employs heuristic methods for tensor data management. However, its tensor analysis considers only the tensor's lifetime. Additionally, Sentinel lacks dynamic weight adjustments for the tensor lifetime characteristic during training. These limitations lead to a sub-optimal solution and hinder the improvement of its performance.

3) *Real-time Adaptability:* offline solutions struggle to adapt promptly to environmental changes. AutoTM [12] and STR [28] employ brute-force mathematical techniques to calculate allocation strategies, yet such methods suffer from inaccuracies, high computational overhead, and the risk of out-of-memory (OOM) errors. Consequently, neither approach optimizes PM access latency based on its read and write traits, nor do they effectively address incomplete bandwidth utilization during buffer copying.

Key insights and contributions: We propose DeepTM, an efficient tensor management approach that is tailored for heterogeneous memory to alleviate memory bottlenecks during DNN training. DeepTM features a page-level aggregation approach for heterogeneous memory systems guided by tensor access patterns, and a tiered memory management strategy based on the quantitative analysis of the DNN model and Deep Reinforcement Learning (DRL) optimization. We provide two important insights: (1) Unorganized tensor accesses and migrations can introduce significant overhead of cache misses and low bandwidth utilization. (2) Tensor analysis based on a single characteristic, employing static methods, or focusing solely on local properties can result in sub-optimal optimization outcomes. We analyze tensor access patterns, considering the number and distance of the future accesses per static computational graphs. To enhance the cache hit rate for frequently accessed tensors, we introduce a batching operation, which groups micro tensors with similar access patterns into the same page. Simultaneously, we propose an adjoin operation to allocate tensors with comparable access frequencies to contiguous pages. When tensor migrations are required, we implement a continuous page migration approach to optimize memory bandwidth utilization.

Furthermore, we analyze two state-of-the-art optimization methods that lead to sub-optimal and incomplete analysis of tensor characteristics. We dissect four key characteristics influencing tensor heat and design a dynamic adaptation algorithm for fine-tuning the weight for tensor heat. By quantifying the performance impact of tensor allocation and migration, we deploy a DRL optimizer to derive a global memory management solution. To adapt to real-world DNN training environments and minimize system overhead, we employ a collaborative offline-online training approach for avoiding OOM errors. In addition, we configure DRL parameters to deliver cost-effective solutions to DNN training.

In summary, our key contributions are as follows:

1) We design a page-level aggregation approach that groups micro tensors with similar access patterns into the same page, thereby enhancing cache hit rates. Additionally, we allocate large tensors with comparable access frequencies to sequential pages, effectively optimizing memory bandwidth utilization during tensor migration.
2) Leveraging four key tensor characteristics to quantify heat of tensors, we develop an algorithm that dynamically adapts tensor heat to the current state. By profiling the computational graph of DNN models, we devise a Deep Reinforcement Learning (DRL) solution that performs global performance optimization based on these heats.
3) Through comprehensive performance evaluations, DeepTM emerges as a high-precision, low-overhead, and cost-effective tensor management approach tailored for DNN training in heterogeneous memory systems. It achieves substantial performance improvements over two state-of-the-art methods.

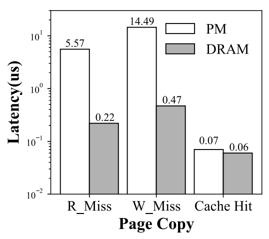
We evaluate the performance of DeepTM for DNN training in a heterogeneous memory system that comprises of 128 GB DRAM and 512 GB PM. Experiments show that DeepTM achieves up to 36% and 49% improvement compared to the state-of-the-art tensor management methods AutoTM and Sentinel, respectively.

In the rest, Section II presents the background and motivation. Section III describes the system design of DeepTM. Section IV gives the analysis of tensor characteristics, DNN training and DRL-based optimization. Section V describes the system implementation. Experimental results are presented in Section VI. Section VII discusses the related work and Section VIII concludes the paper.

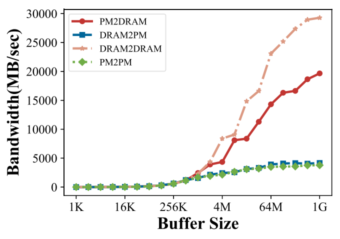
II. Background and Motivation

A. Heterogeneous Memory System

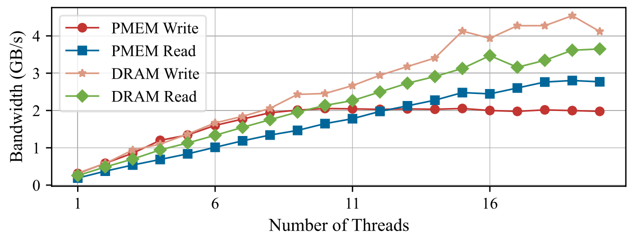
While previous research has shed light on the imbalance in PM read and write bandwidth [45], [46], we find that other factors that impacts heterogeneous memory system performance as well. In typical heterogeneous memory systems, PM devices exhibit severe miss penalty, particularly for write. Fig. 1(a) demonstrates that reading and writing pages in PM results in significant latency of 5.57 μs and 14.49 μs, respectively, which are 25.3 times and 30.6 times greater than those DRAM latency



(a) Overall latency for page Write/Read.



(b) Actual bandwidth for multi-threaded buffer copying.



(c) Performance with varying thread counts.

Fig. 1. Read and write latency and bandwidth of heterogeneous memory.

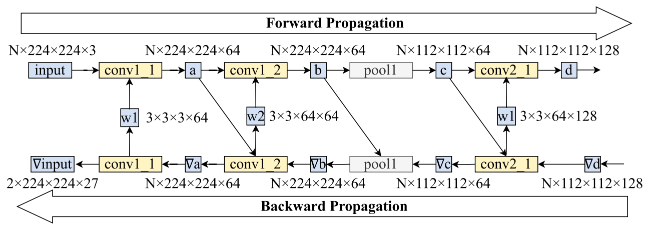


Fig. 2. Forward layers conv1_1, conv1_2, pool1, conv2_1 with output tensors a, b, c, d, and backward layers conv1_1', conv1_2', pool1', conv2_1' with output tensors ∇a, ∇b, ∇c, ∇d in part of the VGG16 network.

(0.22 μs and 0.47 μs). In contrast, the access latency is significantly reduced when the page is in either DRAM or PM with cache hits (about 0.06 μs). This highlights the significance of minimizing PM access or ensuring a high cache hit rate, so as to effectively reduce overhead in scenarios involving substantial tensor allocation and page copy.

In scenarios involving multi-threaded copy, the read/write performance of PM significantly lags behind that of DRAM. As depicted in Fig. 1(b), we observe a noticeable performance improvement in read and write latency for both PM and DRAM as the transferred buffer size increases in 16-threaded scenarios. When copying smaller buffers, the actual bandwidth is relatively limited due to the overhead associated with thread startup and address translation. The constraints posed by the cache size and PM write bandwidth lead to DRAM exhibiting write speeds more than seven times faster than PM when copying buffers larger than 1 GB in size. Furthermore, when the read/write buffer size falls below 64 KB (1 MB in 16-threaded scenarios), the read and write latency of PM is similar to that of DRAM. To harness the full memory bandwidth effectively, it becomes necessary to continuously move larger buffers during memory copying. This observation also proves that with the help of the cache, PM can quickly copy small buffers without reaching its read and write bandwidth bottleneck.

Finally, existing studies have demonstrated that the number of replication threads significantly impacts the read and write performance of both DRAM and PM [56]. OdinFS [57], for instance, considers leveraging delegation threads for concurrency to maximize PM utilization. We analyzed the impact of memory read and write operations on bandwidth performance under different thread counts, as shown in Fig. 1(c). We observed that for multi-threaded writes, the increase in PM bandwidth gradually slows down and ceases to improve beyond 8 threads. Further increasing the number of parallel threads does not benefit PM writes and instead reduces the bandwidth due to increased overhead. In contrast, the bandwidth for DRAM read/write operations and PM read operations generally peaks at 16 threads in our experimental environment. Beyond this threshold, further increasing the thread count results in decreased bandwidth. Therefore, we conclude that setting the thread count to 8 for scenarios involving PM writes, and 16 for other cases, achieves the maximum bandwidth utilization.

B. Memory Bottleneck in DNN Training

DNN training is highly memory-intensive due to its numerous parameters, complicated layer, and massive intermediate data used during the multiple training iterations. When training complex networks, a substantial amount of memory is required for storing intermediate results, gradients, and model parameters, resulting in memory tension.

Memory access patterns during DNN training can be profiled using computational graphs, which are categorized into two types: static computational graphs (SCG) and dynamic computational graphs (DCG). SCGs are generated by parsing the DNN application's source code before training begins. The SCG remains fixed throughout the entire training process, allowing for offline optimization and scheduling.

Fig. 2 shows an example of a static computational graph (SCG) from the VGG16 network [17]. Starting with forward propagation, input data is initially passed through the network, undergoing computations through multiple convolutional and pooling layers, ultimately yielding prediction results and loss. The forward propagation generates several intermediate tensors: a, b, c, and d. In the backward propagation process, the tensors a, b, c, and d are used to calculate the gradient. Then the output tensors $w1$, $w2$, and $w3$ are used to update the parameters in the corresponding convolutional layer. During the above computation process [1], a large number of tensors could occupy memory for a long time, which brings significant memory bottlenecks. Moreover, the sizes of the intermediate tensors (a, b, c, and d) are partially determined by the batch size N (larger Ns bring larger output tensors), which further exacerbates the memory bottleneck.

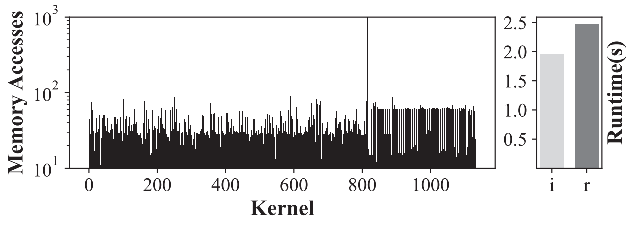


Fig. 3. When training ResNet152 with a batch size of 128 of AutoTM [12], the access number of heterogeneous memory in each kernel and the overall performance influence.

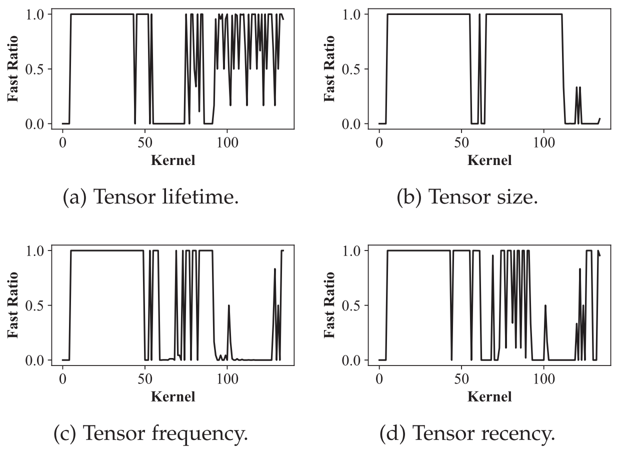


Fig. 4. Single-characteristic tensor management strategies for VGG19 training. Analyzing different characteristics will yield varying memory optimization results.

C. Performance Analysis of the State-of-the-Art

We conduct case studies to obtain a qualitative and quantitative analysis of the data access patterns of DNN training in heterogeneous memory systems.

Fig. 3 provides an overview of the memory access patterns during the training of the ResNet152 model using AutoTM [12], with only 25% of DRAM capacity allocated. The figure reveals a high volume of memory accesses occurring throughout the training process. Notably, certain computational kernels exhibit an exceptionally large number of memory accesses to tensors, some even exceeding 200,000 accesses. Within this extensive stream of memory accesses, a significant portion involves reading from and writing to both DRAM and PM, primarily for micro tensors with sizes smaller than 4 KB. The repetitive reading and writing of these micro tensors in PM can lead to pronounced cache miss latencies and low memory bandwidth utilization due to the sheer volume of random memory accesses. It is our expectation that addressing and eliminating the latency associated with these memory accesses can yield a substantial performance improvement of up to 26%. The extensive memory accesses observed in DNN training scenarios present a considerable challenge to the current heterogeneous memory systems, underscoring the need for optimization measures.

Fig. 4 illustrates the rate of required tensors in fast memory based on the migration strategy by analysis of four distinct tensor characteristics: lifetime, size, frequency, and recency, for migration. Fig. 4 illustrates the actual rate of input tensors in fast memory resulting from the analysis of four distinct tensor characteristics (lifetime, size, frequency, and recency) for migration. Existing approaches tend to manage tensors based on a single tensor characteristic, which can pose challenges when optimizing the training of various DNN models. For instance, Sentinel [26] relies on collecting tensor lifetimes to differentiate and allocate tensors, while SwapAdvisor [18] and Capuchin [29] primarily concentrate on tensor size for swapping decisions. The subplots represents the rate of required data size in the fast DRAM for training under the corresponding kernels. Notably, these approaches do not consistently ensure that tensors crucial for computation are consistently placed within fast DRAM. This observation underscores the pressing need for an efficient tensor management approach capable of identifying tensor heat and placing them optimally by leveraging all four tensor characteristics.

On the other hand, existing mathematical methods based on Integer Linear Programming for tensor management may encounter Out of Memory (OOM) errors [12], [28]. These errors can occur when there is a significant disparity in tensor sizes during training, leading to inaccuracies in Gurobi calculations [27]. As depicted in Fig. 5(a), during VGG19 training with AutoTM [12], the DRAM usage exceeded the 128 GB limit multiple times, reaching a maximum of 24%, thus increasing the risk of OOM errors during training. Fig. 5(b) demonstrates the impact of extensive memory swapping to disk or PM due to memory overflow on training performance. While OOM errors can be avoided by utilizing the virtual memory space on disk, the substantial number of major page faults results in frequent disk space swaps, incurring a 30% overhead relative to the ideal scenario. Furthermore, additional strategies are required to swap the overflow memory into PM, resulting in a runtime overhead of 121% compared to the ideal scenario. These observations underscore the need for a more effective solution to ensure precise tensor management and prevent OOM errors.

D. Challenges

In summary, we identify three challenges in tensor management for DNN training in heterogeneous memory systems.

1) Mitigating Cache Misses and Maximizing Memory Bandwidth Utilization: The challenge arises from frequent reads and

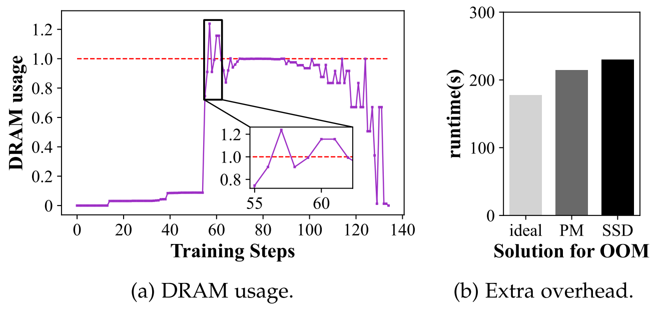


Fig. 5. OOM issues in AutoTM for VGG19(512) training process.

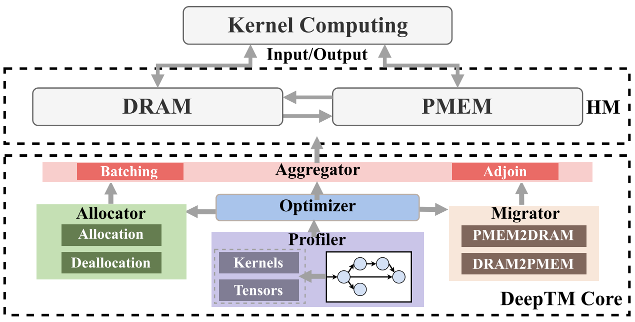


Fig. 6. DeepTM System overview.

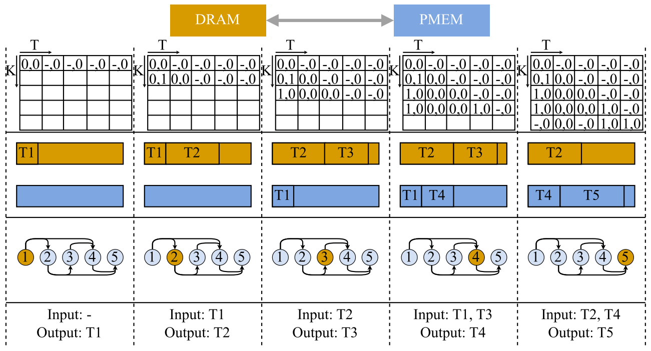


Fig. 7. A profiling example for 5 tensors and 5 kernels.

writes of small tensors, as well as the fragmented copying of tensors in DNN training.

2) Leveraging Diverse Tensor and Kernel Characteristics: The challenge lies in harnessing the full potential of various tensor and kernel characteristics to optimize data swapping in the heterogeneous memory system.

3) Balancing Performance, Overhead, and Cost: Existing memory optimization approaches often lack efficiency and precision, and come with high overhead as well. Furthermore, while configuring more fast DRAM can improve system performance, yet it also incurs significant cost.

III. System Design

A. System Overview

Fig. 6 provides an overview of the DeepTM framework, which consists of three primary components: DeepTM Core, Heterogeneous Memory System, and Kernel Computing. Within DeepTM Core, we have developed five key sub-modules: Profiler, Optimizer, Allocator, Migrator, and Aggregator, all of which manage the heterogeneous memory system. Finally, the kernel computes using the input tensor and generates the output tensor.

The main contributions of this work are reflected in the implementation of DeepTM Core: In the Profiler, the corresponding Static Computational Graph is analyzed to obtain information about all the kernels and tensors with execution orders and dependencies. After that, the Optimizer formulates the tensor management problem and provides an optimization solution based on the information from the Profiler. The Allocator handles allocation and deallocation in the heterogeneous memory for intermediate tensors. The Migrator conducts tensor migration in the heterogeneous memory system, which launches a memory read process and a memory write process to finish tensor movements between DRAM and PM. Finally, the Aggregator organizes tensors at the page level for their allocation and migration.

B. SCG Profiling

In the Profiler, our goal extends beyond merely extracting the fundamental information and dependencies of tensors and kernels within the Static Computational Graph (SCG). We strive to dynamically analyze tensor behavior and characteristics throughout the whole training process. For each certain tensor t and kernel k, the following attributes are maintained by the Profiler before training: the size of a tensor s_t; the lifetime l_t, which is the number of kernels that tensor t goes through from allocation to deallocation; input and output tensors for each kernel tin^k and $tout^k$. Besides, we need to obtain the number of accesses to the living tensor in each kernel, and calculate the future access counts f_t^k and future access distance r_t^k of tensor t and kernel k. The number of accesses is not equal to the number of input/output tensors in the kernel, as the same input tensor may be called multiple times in the kernel's computation (e.g. convolutional kernels)

Furthermore, the profiling approach delves into the real-time monitoring of each tensor's state within each kernel and their dynamic behaviors, encompassing allocation and migration activities during the training process. To illustrate this concept, Fig. 7 provides an example of real-time profiling during DNN training, involving a scenario with 5 kernels (K1, K2, K3, K4, K5) and their corresponding 5 tensors (T1, T2, T3, T4, T5). Within the context of a specific solution space denoted as $A_G^{L_D}$, we employ a status record table to meticulously document the behavior exhibited by each tensor t with a given kernel k. Each entry in the table is represented as $a(k,t) = (p_t^k, m_t^k)$, encapsulating two crucial attributes: Position (p_t^k): This attribute signifies the memory location associated with tensor t within kernel k. Specifically, it employs the values '0' (DRAM), '1' (PM), or '-' (others) to denote the position. Migration m_t^k: The migration attribute m_t^k indicates whether tensor t undergoes migration within kernel k. It serves as a binary indicator where '1' signifies migration, and '0' indicates no migration. In our scenario, every entry in the table effectively becomes a factor that influences training performance.

Additionally, we delve into quantifying the computation time for each kernel in scenarios where input and output tensors reside in different memory locations, specifically, DRAM and PM. This analysis entails evaluating the time taken by a kernel when processing all input tensors and all output tensors within a particular combination of memory types, whether DRAM or PM. By assessing the actual distribution of input and output tensors between DRAM and PM, we can derive estimates for the computation time incurred by a kernel under different memory configurations. It is important to note that this comprehensive

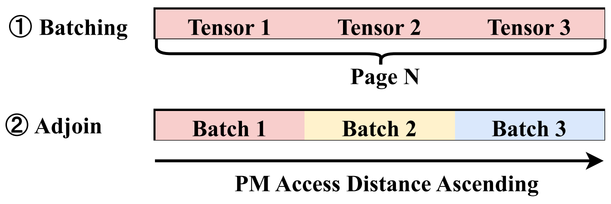


Fig. 8. The Aggregator working principle.

profiling procedure spans a total of four training iterations for each distinct combination of DRAM and PM, ensuring a robust understanding of kernel behavior and performance characteristics across various memory scenarios.

C. Page-Level Memory Management

1) Tensors Aggregation: As analyzed in Section II-A, multiple accesses to heterogeneous memory can lead to a significant number of cache misses, which can severely impact performance. To mitigate access latency and improve the cache hit rate, particularly for PM, we propose a page-level aggregation approach to optimize tensor organization within the virtual address space. Fig. 8 provides a visual representation of the batching and adjoining memory allocation processes within the Aggregator.

Tensor Batching: During DNN training, numerous micro tensors, each smaller than 4 KB, are frequently used for computation. These small tensors often have a short lifetime, typically lasting for only one kernel. They are distributed across different memory pages, leading to inefficient memory access. To address this issue, we introduce a batching operation for these micro and short-lived tensors, enhancing temporal data locality during training. Initially, we determine the expected access count c_t^k and the size s_t of each tensor t from the Profiler for those tensors to be accessed in the current kernel k. Next, we sort the tensors by c_t^k in descending order (or by s_t if c_t^k is identical). At the beginning of a kernel, we allocate the most frequently accessed tensors to the same page in order, ensuring that tensors with the same c_t^k are grouped together. If a page cannot accommodate all tensors with the same c_t^k, we allocate them in descending order of s_t until all tensors are allocated in memory. This batching of micro temporary tensors offers two key advantages: (1) It enhances the temporal locality of data access, increasing the cache hit probability when accessing this page; (2) It reduces the potential for intra-page fragmentation, optimizing memory space utilization.

Tensor adjoining: In addition to batching, we tackle the issue of buffer size affecting bandwidth utilization, as discussed in Section II-A. DeepTM optimizes memory bandwidth during tensor copying operations by employing a tensor adjoining mechanism. This mechanism is particularly effective for larger, less frequently accessed and long-term tensors, where migration operations are more commonly applied. At the beginning of a kernel k, we define the future reuse distance r_t^k and the future reuse count f_t^k for each tensor. Long-term tensors, typically feature maps generated by kernel computations that persist in memory for extended periods, are crucial to manage efficiently. DeepTM takes an adjoining approach to allocate these tensors across contiguous pages in ascending order of r_t^k. To ensure memory continuity, we batch tensors with the same r_t^k and allocate them to contiguous memory spaces in ascending order of f_t^k before performing adjoining. The adjoining operation is specifically carried out on tensors within the current kernel requiring memory allocation. As it progress to the next kernel, DeepTM continues this process by allocating memory space for them at new addresses. Beisdes, tensors allocated in neighboring kernels often end up being positioned next to each other in memory. This, in turn, enhances spatial locality in memory access and facilitates sequential read and write operations, leading to improved memory bandwidth utilization.

Coordination of batching and adjoining: Allocating space for both large and small tensors in alternating patterns can be challenging and may result in a high number of intra-page fragments. To address this issue, we reserve an additional memory space for tiny tensors, approximately 10% of total peak memory usage. This decision was informed by a comprehensive analysis and evaluation of tensor statistics across various DNN models, which revealed that the total size of microtensors that smaller than 4 KB typically does not surpass 10% of the total peak memory usage. When considering memory allocation for a tensor t, we first determine whether s_t is less than or equal to PAGE_SIZE and if the number of future accesses is greater than or equal to 2. If these conditions are met, we perform a batching operation and allocate the tensor to the additional memory space. For other tensors, they are allocated to the remaining memory space, and adjoining actions are applied.

2) Contiguous Migration: To optimize memory bandwidth utilization, DeepTM employs a continuous migration strategy within the Migrator module to consolidate contiguous memory accesses into a single transaction. When the Optimizer decides to migrate a tensor, DeepTM seeks out all tensors residing in a contiguous address space that share the same number of future access distances as the target tensor. Subsequently, the Migrator orchestrates the migration of all these contiguous pages. In cases where contiguous tensors with identical future access distances are unavailable, DeepTM identifies tensors with adjacent future access distances within a contiguous address space and migrates them as a group. Furthermore, DeepTM proactively remaps adjacent pages when tensors are released from memory. This remapping strategy is pivotal in ensuring page continuity and mitigating excessive memory migration.

Section IV-B provides a detailed quantitative analysis of migration costs and benefits, which informs our decision-making process regarding whether to migrate a single tensor or multiple contiguous tensors.

IV. Analysis and Optimization

A. Analysis of Tensor Characteristics

1) Heat of Tensor: As discussed in Section II-C, traditional heuristics often struggle to guarantee efficient access to input tensors in fast DRAM. To address this challenge, it is crucial to identify the relevant characteristics required to analyze the heat of tensors. Drawing inspiration from established cache

Algorithm 1: Dynamic Weight Adjustment.

```
Input: $W = (w_r^k, w_f^k, w_s^k, w_l^k)$, DRAM Limit
       $L_D$, Size of living tensors $s_T$, Kernel $k$
       , Characteristics $(r, f, s, l)$, Threshold $Th$
Output: $H_t$, $W^{k+1}$
1  while $S_h \geq 1.1L_D$ or $S_h \leq 0.9L_D$ do
2      $S_h \leftarrow 0$ ;
3      if $s_T \geq 2.5L_D$ then
4          $w_s^k \leftarrow w_s^k \times (1 \pm 0.02)$ ;
5      else
6          $w_r^k \leftarrow w_r^k \times (1 \pm 0.02)$ ;
7      end
8      foreach Living Tensor $t$ do
9          $H_t \leftarrow f_t^k w_f - r_t^k w_r - s_t w_s + l_t w_l$;
10         if $H_t \geq T_h$ then
11             $S_h \leftarrow S_h + s_t$;
12         end
13     end
14 end
15 $W^{k+1} \leftarrow (w_r^k, w_f^k, w_s^k, w_l^k)$ ;
```

management strategies, the widely used approach involves offloading data based on either Least Recently Used (LRU) or Least Frequently Used (LFU) metrics, corresponding to future reuse distances and future reuse counts, respectively. In our context, we leverage these two key metrics to measure the heat of tensors: future frequency and future recency. In practice, frequency-based policies tend to provide a more accurate reflection of data access patterns over a specific time frame when compared to recency-based policies [51]. However, it is important to recognize that when tensor access patterns undergo substantial changes, such as transitioning between different layers in the context of DNN training, recency-based policies may offer greater adaptability to the evolving access patterns. Additionally, our strategy aims to minimize the access latency of PM by prioritizing the placement of small tensors in DRAM whenever possible. Conversely, tensors with longer lifetimes, which may occupy memory for extended duration without frequent access, are candidates for allocation to PM. Consequently, we consider four primary characteristics-size, lifetime, recency, and frequency–to signify the heat of tensors. Our overarching objective is to ensure that tensors with higher heat are predominantly retained in fast DRAM to optimize overall training performance.

2) Dynamic Weight Adjusting: Due to the various DNN models and training steps, we need to define the heat of each tensor and kernel to reflect their importance. In the complex process of DNN training, it is necessary not only to define the threshold for hot tensors but also to dynamically adjust the weights of tensor characteristics to adapt to the training environment. DeepTM introduces the implementation of dynamic characteristic weight adjustments, markedly improving adaptability across different steps of DNN training as depicted in Algorithm 1. This innovation aims to refine the heat H_t of tensors, thereby equipping optimizers with the data necessary for making well-informed decisions. At the outset, we specify each tensor's input characteristics–namely, their reuse distance r_t^k, frequency of reuse f_t^k, size s_t, and lifespan l_t, along with relevent memory-related data as detailed in Algorithm 1. Initially, we applied min-max normalization to all characteristics to scale them to the same dimension. We then set all weights w to 1, making sure all characteristics are treated equally at the beginning of training. Besides, we established a constant threshold $T_h = 0$ for identifying "hot" tensors. An imbalance in the distribution of tensor heat is indicated when the total size of 'hot' tensors exceeds 1.1 times the available DRAM capacity or falls below 0.9 times the same. In such cases, our algorithm requiring a dynamic readjustment of weights to prevent the system from overheating or undercooling. For instance, too many 'hot' tensors necessitate lowering their H_t value, achieved by increasing w_r^k or w_s^k. Conversely, to increase H_t, we might decrease w_r^k or w_s^k, depending on the current DRAM pressure. Further evaluations check the stress levels on DRAM resources. If the total sizes of tensors are more than 2.5 times the DRAM capacity, it points to a bottleneck, which matches Intel's suggested DRAM to PM ratio of 1:4. Thus, optimizing memory use to where $s_T \geq 2.5L_D$ marks a midpoint in training's peak memory demand, about half of the total memory capacity. During times of high DRAM pressure, the weight w_s^k related to tensor size is adjusted to highlight its critical role in memory allocation. On the other hand, with less DRAM pressure, the focus moves to w_r^k for its increased significance. Choosing to adjust by 2% in each iteration comes from thorough experimental validation, making sure the algorithm's weight adjustment process is strong and effective. This careful strategy ensures a good balance between accuracy and efficiency in how the algorithm works.

B. Objective Function

In order to better quantify memory usage and the performance overhead associated with tensor copying and computation during training, We will formulate the training performance by analyzing the parameters and metrics of model training in Section IV-B. We denote the DNN's computational graph as *G(V, E)* and its DRAM size limit as L_D. Our Optimizer aims to minimize the total training time with a certain limit of DRAM capacity via managing the intermediate tensors. We can formulate this optimization problem to minimize C_{total}:

$$C_{total} = C_e + C_m \tag{1}$$

where C_e and $C_m$represent kernels' computation time and tensors' migration time, respectively.

Due to synchronous computation, tensor operations must be completed before the kernel computation starts. We want to get an optimal solution space $A_G^{L_D}$ to operate on each living tensor $t \in T$ at the end of each kernel $k \in K$ execution a_t^k. After calculating the output tensors *t* in a kernel *k*, we use p_t^k to denote where it is stored:

$$p_t^k = \begin{cases} 0, & \text{if } t \text{ is in DRAM after kernel } k \\ 1, & \text{if } t \text{ is in PM after kernel } k \\ Null, & \text{if } t \text{ is not in memory } k \end{cases}$$

$$\text{s.t.} \sum_{t \in T} (1 - p_t^k) \times s_t \leq L_D, \forall k \in K \tag{2}$$

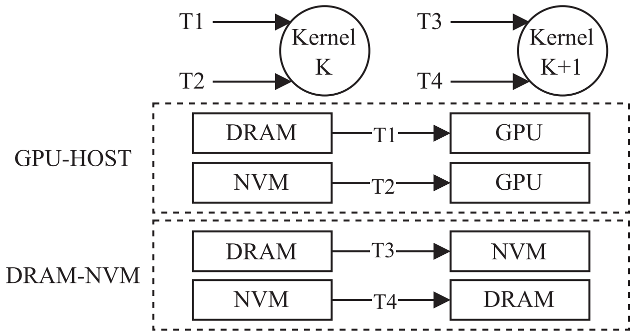


Fig. 9. Asynchronous GPU-Accelerated Execution Scenario.

m_t^k indicates whether tensor t starts migration after kernel k. Since we focus on the data migrations between two memory media in the heterogeneous memory system, there are only two possible states for this variable:

$$m_t^k = \begin{cases} 0, & \text{no migration} \\ 1, & \text{migration} \end{cases} \tag{3}$$

The operation would be suspended if a memory allocation p_t^k or tensor migration m_t^k causes the DRAM space L_D to be zero, which results in out-of-memory (OOM) errors.

Expanding the aforementioned scenario to K kernels and T tensors optimization, we need to formulate and calculate the costs of migration and computation. From (1) we can derive the overall time overhead includes two parts: C_e and C_m. The overall cost can be formulated as:

$$C_e = \sum_{k \in K} c_e^{tin^k_{status}} \tag{4a}$$

$$C_m = \sum_{k \in K} \sum_{t \in T} m_t^k \times s_t \div B_t \tag{4b}$$

where tin^k is the set of all input tensors of kernel k. $c_e^{tin_k}$ is the execution time of kernel k under the input tensor tin_k.

Specifically, in GPU or DMA scenarios, memory migrations through asynchronous replication are hidden under the computations [55]. In GPU scenarios, DRAM and NVM share the PCIe bandwidth from the host to the GPU, but data copying within the host can be fully parallelized with GPU-to-host operations. Fig. 9 illustrates DeepTM's memory management approach in an asynchronous GPU computation scenario. Notably, DeepTM does not manage GPU memory directly; instead, we use it as a higher-level cache to accelerate computation by default. For tensors T1 and T2, which participate in kernel K computations, if they are not in GPU memory, they are directly read from the respective main memory locations to the GPU via PCIe for computation. Concurrently, during GPU computation for kernel K+1, host-side data copying can occur in parallel, such as offloading T3 from DRAM to NVM or prefetching T4 from NVM to DRAM. This means that the overhead of data copying within the host can be entirely or partially overlapped by the GPU computation time, depending on the duration of the GPU computation.

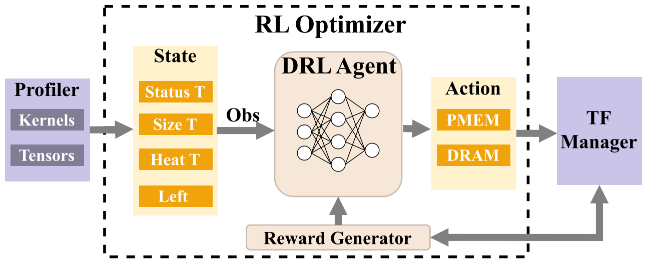


Fig. 10. DRL framework overview.

During the training process for asynchronous cases, we use each kernel execution time as the migration window, and the m_t^k should be replaced by am_t^k, which represents asynchronous migration coming from the tensor t in window k. For total migration cost ac_m^k of Kernel k, the migration overhead only needs to be calculated for the part that is on the critical path:

$$ac_m^k = \max\{m_t^k \times s_t \div B_t - c_e^{tin^k_{status}}, 0\} \tag{5}$$

Thus, C_m in (4b) can be calculated as:

$$C_m = \sum_{k \in K} ac_m^k \tag{6}$$

The ongoing occupation of the migration targets memory space during the migration process. When too many tensors are being migrated during training, the asynchronous migration method may lead to even tighter DRAM space. The memory limit for L_D becomes:

$$\sum_{t \in T} (p_t^k \times m_t^k \times s_t + (1 - p_t^k) \times s_t) \leq L_D, \forall k \in K \tag{7}$$

C. Deep Reinforcement Learning Model

Rationale for DRL: A more efficient solution is demanded to solve the formulated optimization problem, as the optimization problem of C_{total} for both synchronous and asynchronous scenarios are NP-C problems and the solution complexity is $O(2^N)$. A fundamental insight into this DNN training problem reveals its alignment with a Markov Decision Process. Leveraging this concept, we turn to the promising avenue of Deep Reinforcement Learning (DRL) to tackle this optimization problem. DRL offers several compelling advantages in this context. First, it empowers the DRL agent to dynamically sense and adapt to changes in the environment, enhancing its capacity to analyze evolving model characteristics and make informed decisions. More importantly, DRL can systematically account for the prospective benefits of each operation. Unlike conventional heuristics, which often lack foresight, DRL excels in optimizing for the future, allowing it to navigate the management of heterogeneous memory more effectively.

Fig. 10 shows the holistic framework of the DRL Optimizer. The DRL agent sources vital data from TensorFlow (TF) kernels, encompassing both tensors and kernels within the static computational graph. In the continuous decision-making process, the DRL agent operates in cycles. During each time step, it retrieves the current state from the environment and selects an action from the action space. The DRL Optimizer employs a neural network to approximate the Q-function, essential for mapping states to expected rewards for all possible actions. This network architecture comprises a flatten layer followed by three dense layers, designed to minimize the discrepancy between predicted Q-values and target Q-values derived from the Bellman equation. Through this approach, the DRL agent gradually refines its strategy to prioritize actions that maximize cumulative rewards, thus optimizing episodic training outcomes. Subsequently, the reward generator evaluates the effectiveness of the executed action. The key components in the RL Optimizer are defined as follows:

Environment: The environment consists of the information from the DNN model and real-time status obtained from the Profiler. Noteworthy parameters include tensor sizes s_t, available memory resources L_D and P_D, execution order and execution times $c_e^{tin^k_{status}}$ of each kernel in various tensor states, and quantities of tensors T and kernels K in the computational graph along with their respective tensor inputs tin^k and outputs $tout^k$.

Time-step: Given the extensive search space associated with intermediate tensors, our time-step strategy entails considering all living tensors in memory during kernels with DRAM bottlenecks. In a computational graph with T tensors and K kernels, this results in a maximum of $T \times K$ steps.

Agent: The agent is the pivotal decision-maker responsible for tensor allocation and migration during training. At each time step, it draws information from the environment to select an action and progress to the next state.

Episode: An episode represents the entire DNN training process, commencing from the initiation of the first kernel and culminating with the completion of the last kernel. Episodes may also terminate prematurely in the event of an OOM error stemming from tensor operations.

State Space: The state space in the training scenario manifests as a multidimensional discrete space, represented as a 1-dimensional vector: $[p_t, H_t, D_{left}, P_{left}]$. Here, p_t denotes the current tensor's position in memory, while H_t is a weighted combination of four significant tensor characteristics outlined in Section IV-A: size s_t, future frequency f_t, future recency r_t, and lifetime l_t. Furthermore, D_{left} and P_{left} signify the available memory resources during the current kernel.

Action Space: The action space signifies the placement of tensors within kernels. This can be succinctly represented as a discrete one-dimensional variable: $a_t^k \in [0, 1]$.

Reward: Upon the agent's selection of an action, an immediate reward R_t^k is conferred. R_t^k is contingent on the location of the current kernel's input tensor, calculable based on the benefit derived from the allocation or migration action:

$$R_t^k = \frac{s_t}{\sum_{ti \in tin^k} s_{ti}} - m_{nor} \tag{8}$$

When a migration of a tensor from one memory location to another takes place, a reward is assigned based on the normalized migration cost m_{nor}. To prevent OOM errors during training, a substantial negative reward is applied when tensor operations lead to inadequate available memory. Upon successful completion of an episode, an episodic reward is granted. The episodic reward (R_{epi}) represents the additional reward for completing the training of a single episode. To calculate R_{epi}, we consider the worst-case scenario where all input tensors of all kernels are executed in PM, resulting in a maximum overhead denoted as $C_{\max}$.

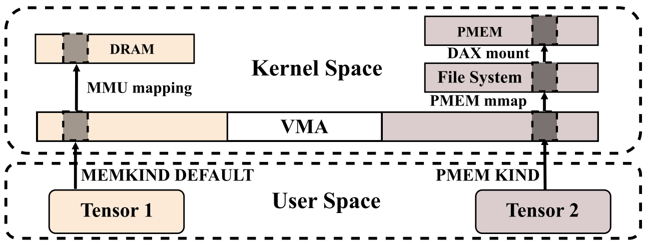


Fig. 11. Address mapping and memory management of tensors in FSDAX mode of PM.

$$C_{\max} = \sum_{k \in K} c_e^{PM} \tag{9}$$

Similarly, in the best-case scenario, all input tensors are in DRAM with the corresponding overhead of $C_{\min}$:

$$C_{\min} = \sum_{k \in K} c_e^{DRAM} \tag{10}$$

In this way, the execution time C_{normal} in this episode can be normalized to:

$$C_{normal} = \frac{C_{total} - C_{\min}}{C_{\max} - C_{\min}} \tag{11}$$

Consider that in certain scenarios, we may require varying ratios of fast and slow memory to achieve better efficiency and cost performance. The price of the entire heterogeneous memory, denoted as P_{normal}, can be expressed as:

$$P_{normal} = \frac{U_P \times L_P + U_D \times L_D - U_P \times L'_P}{U_D \times L'_D - U_P \times L'_P} \tag{12}$$

where U_D and U_P represent the unit price of DRAM and PM, respectively. L'_D and L'_P represent the capacity required to fully utilize DRAM and PM, respectively. To strike a balance between price and performance, and to determine the better memory ratio based on demand, we introduce the parameter β to indicate the importance of performance.

Finally, R_{epi} can be expressed as:

$$R_{epi} = \beta R_{fix}(1 - C_{normal}) + (1 - \beta) P_{normal} \tag{13}$$

The fixed reward R_{fix} is defined based on the current training scenario. In the end, the overall reward $R_{episode}$ for training an episode is the sum of the R_t^k for each step of the training plus the R_{epi}. The primary objective of the agent is to maximize $R_{episode}$.

V. Implementation

All components in DeepTM are implemented in Linux v5.15.0 and TensorFlow v1.14. As shown in Fig. 11, the persistent memory modules are mounted in the file system in fsdax mode, where PM is configured as a disk-like device through

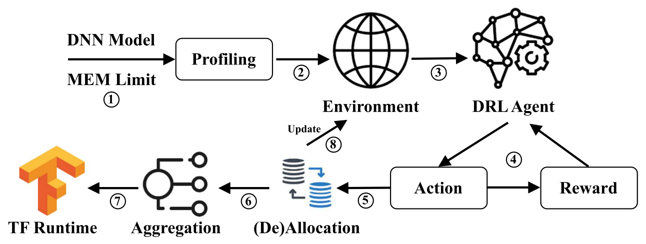


Fig. 12. The workflow of DeepTM.

pmdk [34]. The memkind tool [25] can control memory characteristics and heap partitioning among various memories. Virtual memory can be requested from the operating system for PM via the memkind partition interface. When the system needs to allocate PM space for tensors for the first time, we create PM space through the *memkind_create_PM()* function and map the address into virtual memory space by *memkind_PM_mmap()*. By allocating a page-sized or larger space in PM and assigning corresponding addresses for each tensor buffer, we can achieve a flexible allocation of tensors within and across pages, which takes the minimal overhead. Tensors located in DRAM only require memory space allocation in the form of *MEMKIND_DEFAULT*, followed by direct MMU mapping to DRAM.

The workflow of DeepTM is shown in Fig. 12. DNN model training tasks and DRAM limitations are provided to the Profiler for analysis and processing ❶. Using Profiler-derived information, DeepTM constructs the DRL environment and initializes the system state ❷. The DRL agent, informed by the current state, takes action regarding tensor placement ❸. A reward is generated by the reward generator based on the agent's action ❹. The Allocator and Migrator perform allocation, deallocation, or migration operations based on the agent's choice ❺. For tensors allocation and migration in memory, aggregation operations are executed ❻. TensorFlow continues with the training task ❼. If there are remaining kernels and tensors to process, DeepTM updates the environment and state accordingly and proceeds ❽. The episode concludes when all kernels have been executed, and the agent receives an episodic reward.

VI. Evaluation

A. Experimental Setup

Testbed: For the test environment, we run the experiment on a server equipped with a 20-core Intel Xeon Gold 6226R CPU and 4×32 GB DDR4 host memory. To implement the heterogeneous memory system, we configure 4×128 GB Intel Optane DIMMs as a part of slow memory in the system. All PMs are deployed in fsdax mode in the heterogeneous memory system. Besides, we implemented the memory mode of Intel Optane memory as a comparison to the heterogeneous memory systems. Our system is also suitable for DNN training in GPU-accelerated scenarios. An NVIDIA RTX 3080 GPU (CUDA version is 11.7) with 10.0 GB memory is equipped.

TABLE I
DNN Workload

DNN Model	Computational Graphs		Batch Size	Peak Memory (GB)
	kernels	tensors		
VGG19	135	2947	128	9.07
			512	33.03
ResNet152	1133	26584	128	1.18
			512	3.94
Inception	426	9348	128	0.64
			512	2.41
MobileNet	310	11556	128	0.79
			512	3.10
AlexNet	36	695	128	0.43
			512	1.67

Benchmarks: We evaluate the training of five DNN models under different batch sizes as shown in Table I. It shows that the number of kernels and tensors varies greatly between different DNN models. In addition, different models and batch sizes during training will also affect the memory usage peaks. AlexNet [17] and VGG [4] are typical computer vision workloads with deep convolutional structure and high computational complexity. ResNet [6] addresses the vanishing gradient problem in deep networks by allowing the construction of networks with hundreds of layers. Inception [5] features utilize multiple parallel convolutional layers of different kernel sizes to capture features at various scales within the same layer. MobileNet [40] achieves lightweight and efficient feature extraction through depthwise separable convolutions. For our experiments, we utilized a randomly generated performance evaluation dataset, comprising 60,000 images with dimensions of $224 \times 224 \times 3$. Of these, 50,000 images were designated for training, while the remaining 10,000 images were set aside for testing. All the evaluation times presented in the following sections correspond to the duration required to train for one epoch.

Baselines: We choose three baselines to evaluate DeepTM. These are:

1) *Origin-TF:* The default memory management policy in TensorFlow. To avoid OOM error, we allocate tensors directly to the PM space when the default DRAM space is insufficient.
2) *Auto-TM [12]:* AutoTM formalizes the tensor movement problem into an integer linear problem under a constraint of memory capacity. The problem is solved by Gurobi as it is NP-Hard.
3) *Sentinel [26]:* Sentinel guides the placement of tensors by considering their size, lifetime, and other characteristics, which can be regarded as a heuristic greedy strategy.

Hyperparameters: Table II shows the hyperparameter settings for the DRL agent, along with other environment parameters. The value of β indicates the proportion of rewards received from the optimization time cost. A higher β places more emphasis on the optimization time cost, while a lower β places more emphasis on the monetary cost. The discount factor γ determines the weight of future rewards in Q updates. A higher γ makes the agent more focused on long-term rewards, while a lower γ prioritizes short-term rewards. The parameter configuration for the greedy strategy is as follows: The agent is trained for a

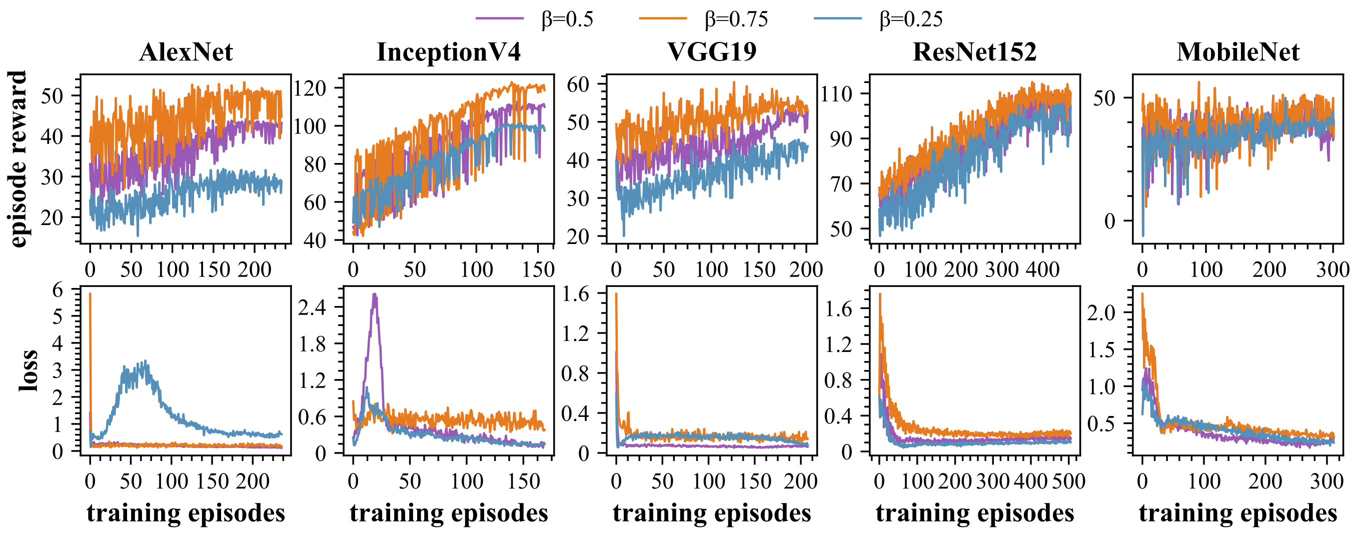


Fig. 13. DQN training reward and loss. The loss will approach convergence after 300 episodes.

TABLE II
HYPER-PARAMETERS AND ENVIRONMENT PARAMETERS

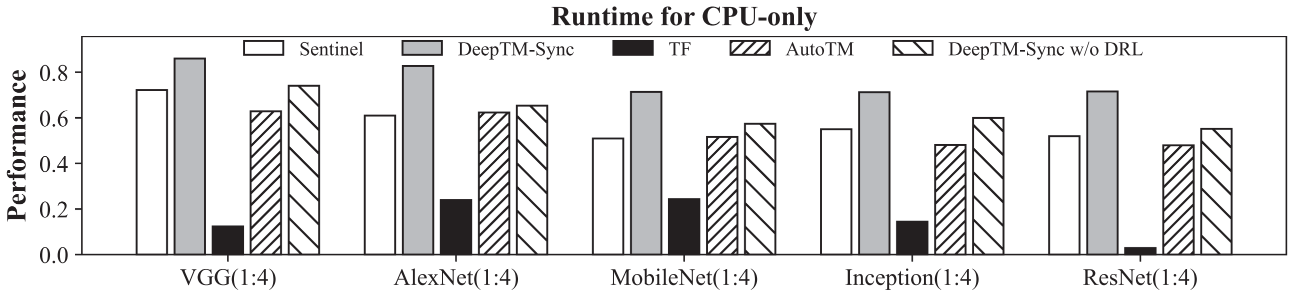
Parameter	Value
Greedy Episodes	150
Training Episodes	200
Replay Buffer Size	100
Batch Size	32
Learning Rate	0.0001
Cost Priority(β)	[0.25, 0.5, 0.75]
Discount Factor(γ)	0.99
Greedy Probability(ϵ)	[0.1, 1.0]
R_{fix}	50

total of 200 episodes. During the first 150 episodes of training, the probability of using the greedy strategy gradually increases from 0.1 to 1.0.

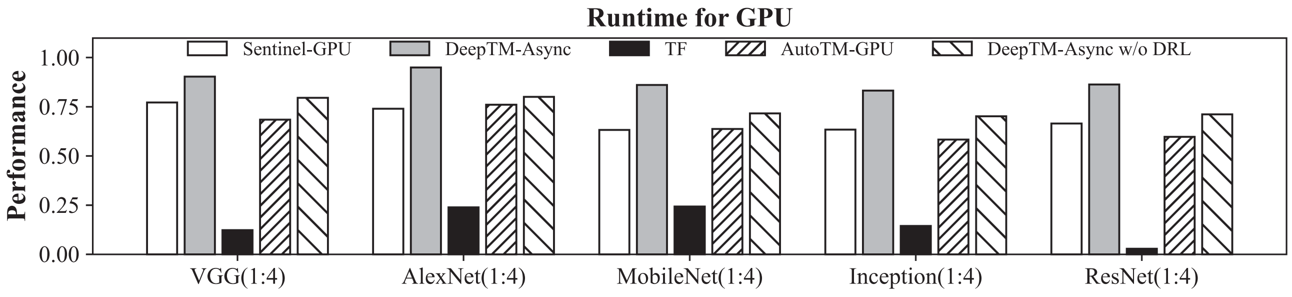
Fig. 13 illustrates the convergence behavior of our DQN algorithm across various network models, including AlexNet, InceptionV4, VGG19, Resnet152, and MobileNet. The loss within our DQN quantifies the difference between the predicted Q-values (as given by the current network) and the target Q-values (calculated according to the Bellman equation). The convergence performance is influenced by both the network architecture and the value of β. For InceptionV4, our agent converges after 130 episodes, while VGG19 and AlexNet require about 200 episodes of training. For models like ResNet and MobileNet, which have a higher number of tiny tensors, requiring over 300 episodes to converge, even though they may not occupy a large amount of memory. In addition, different β values bring different average rewards accumulated patterns of the training process. For larger β values, the impact of time-cost optimization by the agent becomes more significant. We vary the value of β from 0.25 to 0.75. For all the models we design, the reward convergence can be achieved after training on specific iterations. For those networks with fewer model parameters and a shallower network layer, fewer training rounds can earn a higher episode reward.

B. Average Training Time

We conducted an extensive performance evaluation of five DNN network models using various baseline scheduling algorithms. The results are summarized in Fig. 14. In the performance evaluation of this section, we set the fast-to-slow memory ratio used for training at 1:4 (the same as the experimental environment), meaning the total available DRAM space equals 1/5 of the DNN peak memory demand, rather than the full 128 GB. The performance metric employed here is actual performance achievable relative to the benchmark runtime on a server equipped with infinite DRAM space. For instance, a value of 0.8 on the vertical axis signifies that the system has achieved 80% of the optimal performance. DeepTM demonstrates significant performance improvements compared to two state-of-the-art memory management methods. In the context of a heterogeneous memory system with both CPUs and GPUs, we evaluated both synchronous and asynchronous migration strategies of DeepTM. In the optimal case where DRAM space is considered infinite (Performance=1), DeepTM achieves remarkable performance gains. Specifically, for a CPU-only heterogeneous memory system, DeepTM attains performance improvements of 90.2%, 94.9%, 86.1%, 83.2%, and 86.3% across the five DNN models, respectively. DeepTM-async outperforms Origin-TF with an average performance improvement of 8.83x. When compared to Sentinel and AutoTM in the CPU-only scenario, DeepTM achieves average performance enhancements ranging from 19.9% to 35.5% and 31.1% to 49.3%, respectively. Notably, the optimization strategies exhibit relatively less noticeable effects on AlexNet and MobileNet models compared to ResNet. This difference is attributed to the smaller variations in kernel execution times within PM for AlexNet and



(a) Runtime using synchronous migration strategy on a CPU-only heterogeneous memory system.

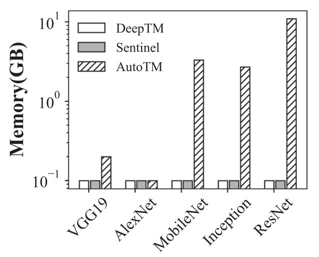


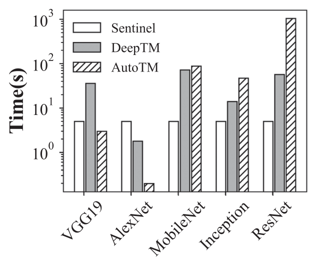
(b) Runtime using synchronous migration strategy with on a heterogeneous memory system with GPUs.

Fig. 14. Overall performance evaluation of five different deep learning models with a batch size of 512.

MobileNet, whereas ResNet benefits significantly from memory management, resulting in substantial performance gains. In GPU scenarios, DeepTM-async also delivers impressive results, with performance improvements ranging from 20.0% to 36.2% over Sentinel-GPU and 14.7% to 42.1% over AutoTM. Finally, to validate the impact of each component of DeepTM on the overall performance, we evaluated the system's performance using a State-Of-The-Art (SOTA) optimizer without DRL, that is, by substituting the DRL optimizer in DeepTM with a SOTA optimizer suited for the current model, without changing other components. Experimental results indicate that our DRL optimize brought the performance improvement ranging from 16.05% to 29.49% for DeepTM-Sync, and improvement ranging from 13.55% to 21.33% for DeepTM-Async. Moreover, we observed that models with less microtensors (such as VGG19 and AlexNet) benefit more significantly from DRL compared to other components. This enhancement can be attributed to the smaller action and state spaces and the larger reward stimuli, which enable the DRL to devise better management strategies. Overall, compared to SOTA methods, the DRL optimizer achieved notable improvements, with the extent of enhancement depending on the specific model being trained.

AutoTM employs a brute-force search approach, which, while capable of global optimization, suffers from inefficiency and limitations in solution accuracy. Sentinel, on the other hand, employs a limited heuristic tensor placement strategy that lead to local optimization issues. Additionally, Sentinel primarily focuses on saving DRAM space, which may not fully utilize fast DRAM resources in certain cases. In contrast, DeepTM not only achieves space optimization for memory allocation and bandwidth optimization for migration but leverages an agent to learn from the environment and achieve better global optimization, making it more adaptable and efficient.





(a) memory overhead of DeepTM (b) runtime overhead of DeepTM

Fig. 15. Overhead comparison of DeepTM and AutoTM.

C. Overhead

Fig. 15 provides a comprehensive comparison of the overhead between DeepTM and two state-of-the-art methods. It is important to consider the different components contributing to overhead in memory management for DNN training.

In the case of DeepTM, the primary sources of the overhead are the initial profiling and the training of the DRL neural network. Notably, these overhead components remain relatively stable regardless of the model size or complexity. DeepTM minimizes the overhead during actual DNN training as it does not require extensive profiling or optimization calculations for each training iteration. Once the DRL agent is trained to convergence, it can make efficient memory management decisions without

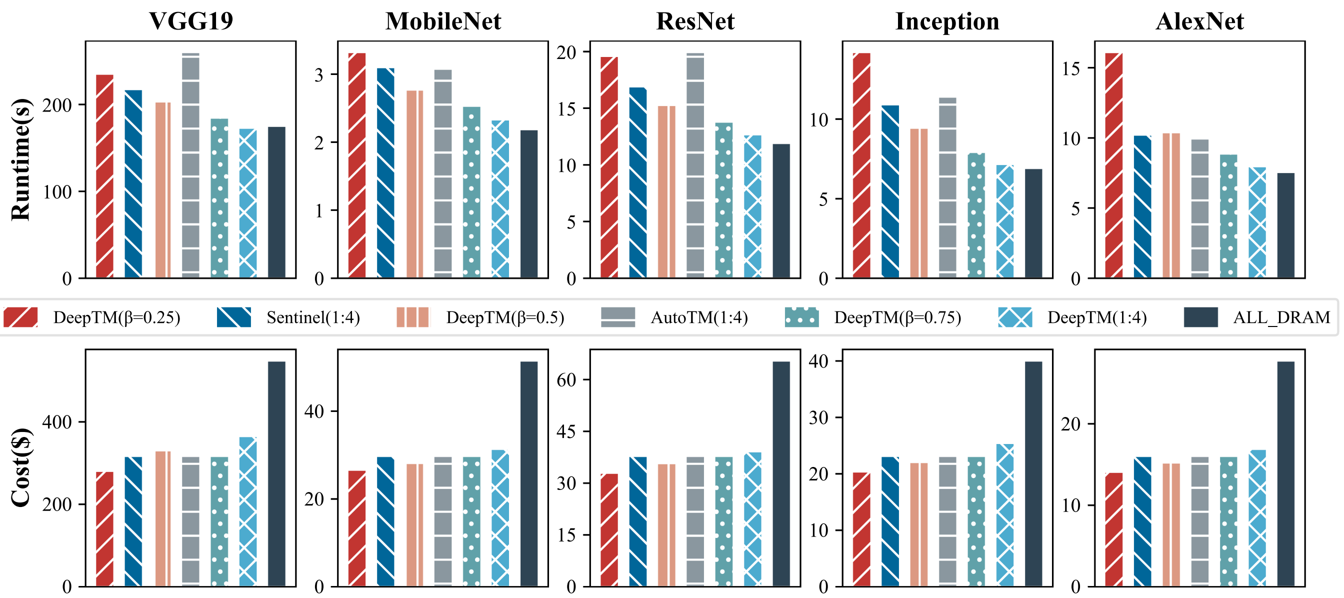


Fig. 16. Cost-efficiency analysis for different β.

significant additional overhead through offline training. This results in gradual diminishing of the overall training overhead as the total number of training iterations increases.

Conversely, other optimization methods exhibit different overhead characteristics. For instance, when dealing with large models like ResNet and MobileNet, Gurobi optimization can be computationally intensive and time-consuming. This results in substantial time and memory overheads. In the case of ResNet, AutoTM's memory overhead can be as much as 109 times that of DeepTM, with time overhead reaching 18.2 times the DeepTM counterpart. Sentinel, employing traditional profiling and heuristic algorithms, introduces little to no significant additional overhead.

In summary, while it incurs overhead during agent training, DeepTM excels in avoiding additional overhead during DNN training. All overheads of DeepTM only occur in the iterations at the beginning of the training. This feature makes DeepTM increasingly efficient as the total training iterations progress, rendering the overall training overhead negligible over time. In contrast, alternative methods like AutoTM often experience significant overhead, primarily due to profiling and optimization processes.

D. Sensitivity and Scalability

Sensitivity Analysis: A key consideration in optimizing memory management for DNN training is the cost-performance trade-off, primarily due to the substantial price difference between DRAM and PM [12]. We assess the cost-effectiveness of the DeepTM system by evaluating the price of all memory potentially utilized within the training system, priced on a per-gigabyte basis. Specifically, we calculate the running cost based on the amount of memory used. The commercial rates are \$23.81 per GB for DRAM and \$7.85 per GB for PM. To evaluate the runtime and cost efficiency of DeepTM, we conducted a sensitivity analysis across five diverse DNN models, comparing it with four default configuration baselines. Fig. 16 illustrates the performance and cost comparisons of DeepTM at varying β values.

In DeepTM, the objective is to strike a balance between leveraging DRAM resources for enhanced performance while minimizing cost. When we set β to 0.25, DeepTM demonstrates an impressive 7.63% average improvement in performance over a specific fast-to-slow memory ratio of 1:4, with a mere 5.7% increase in the total cost. At β values of 0.75 and 0.5, DeepTM continues to excel, delivering significant cost savings of up to 5.6% while ensuring virtually no performance degradation. The β parameter, which introduces the cost factor into the optimization process, plays a crucial role in determining the overall cost efficiency of DeepTM. In comparison to the common 1:4 fast-to-slow memory ratio, DeepTM demonstrates substantial cost savings, reducing expenses by 24.20%, 20.05%, and 6.40% when β is set to 0.25, 0.5, and 0.75, respectively. Based on the experimental analysis, DeepTM consistently performs well, particularly at β values of 0.5 and 0.75. At a β value of 0.75, it achieves superior cost-effectiveness while significantly enhancing operational performance. Even at a β value of 0.5, it strikes a balance between saving fast DRAM space and maintaining favorable performance. In summary, DeepTM consistently outperforms the state-of-the-art approaches in terms of cost-effectiveness and operational performance, with optimal results obtained at β values of 0.75 and 0.5.

Different Batch Size: Fig. 17 provides insights into the maximum fast DRAM capacity required to achieve 90% of training performance with full fast DRAM utilization. Our comprehensive runtime performance analysis underscores DeepTM's consistent optimization prowess across diverse batch sizes and DNN models. When considering batch sizes of 512, DeepTM showcases impressive efficiency, utilizing only 16.7%,

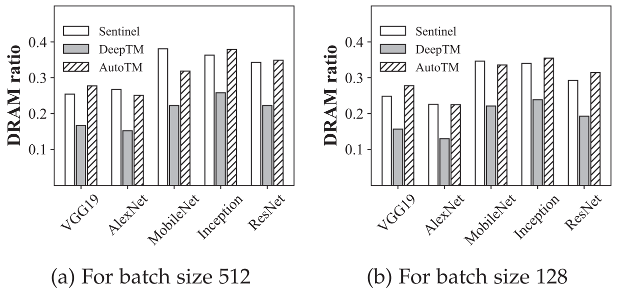


Fig. 17. The least DRAM space required to train in different batch sizes to achieve 90% optimal performance.

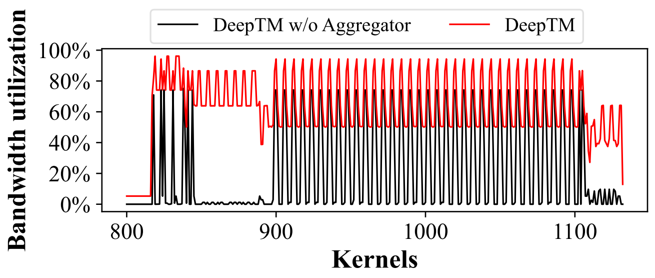


Fig. 18. Comparison of bandwidth utilization for ResNet152.

15.2%, 24.8%, 28.3%, and 24.4% of the fast DRAM capacity for AlexNet, ResNet, Inception, MobileNet, and VGG19, respectively, while maintaining the desired performance level. This translates to an average fast memory reduction of 11.7% and 11.0% compared to Sentinel and AutoTM, respectively. Even at a reduced batch size of 128, DeepTM continues to excel, saving an average of 10.2% and 11.3% of fast DRAM space on average when compared to Sentinel and AutoTM. The ability to navigate optimization challenges arising from varying batch sizes is a testament to DeepTM's adaptability and robust performance. As the batch size increases, the size disparities between different tensors become more pronounced, presenting greater optimization complexities. DeepTM adeptly addresses these challenges, consistently delivering superior solutions across different batch sizes.

Bandwidth Utility Analysis: In Fig. 18, we present a comparative analysis of bandwidth utilization between DeepTM and DeepTM without Aggregator during the training of ResNet152. The inclusion of the Aggregator module yields substantial benefits in terms of memory migration, resulting in a more proactive approach to memory management. This is particularly evident in the 800-900 kernel range, where DeepTM exhibits heightened migration activity, a dynamic not observed in its Aggregator-less counterpart. Furthermore, in kernels characterized by frequent memory access, DeepTM achieves notably higher bandwidth utilization. DeepTM sustains memory bandwidth utilization exceeding 80% for extended periods, compared to an average of just 40% in DeepTM without Aggregator. This significant improvement in bandwidth utilization is primarily attributed to the Aggregator's ability to enhance tensor continuity within the same time frame, consequently increasing the likelihood of executing migrations involving consecutive memory pages. As a result, the overall system performance experiences substantial enhancements.

VII. Related Works

Applications for Persistent Memory: Persistent Memory (PM) has garnered significant attention in various application domains due to its unique performance characteristics. Early studies explored these characteristics, emphasizing PM's utility in hybrid storage configurations, multi-threaded read/write operations, and data indexing structures. These investigations aimed to maximize PM bandwidth utilization based on its specific traits [45], [46], [50]. Leveraging PM's persistence feature, researchers have integrated it into applications such as key-value stores and file system recovery, addressing challenges related to read-write amplification and bandwidth imbalances in PM-persistent metadata and key-value pair management [47], [48], [49]. Another line of research harnesses PM's byte-addressable and fast read/write capabilities to extend DRAM's capacity in memory-intensive applications [12], [26]. Although these studies do not fully exploit PM's persistence characteristics, they effectively combine PM and DRAM to capitalize on PM's capacity and bandwidth while mitigating latency-related drawbacks.

Hardware Constitution of heterogeneous memory systems: Heterogeneous memory systems incorporate diverse hardware components, including SSDs, PMs, and GPUs. SSD-focused optimizations center on storage extension and efficient access methods [41], [42], [43]. In GPU-centric efforts, the focus shifts to data asynchronous migration and recomputation, capitalizing on the relatively independent nature of GPU task execution [29], [30], [36]. ZeRO-Infinity [53] available in DeepSpeed [54] present a heterogeneous systems that leverages slow, but massive, CPU or NVMe memory in parallel across multiple devices to achieve high bandwidth offloading and prefetching. In PM systems, researchers pay particular attention to memory read and write bandwidth characteristics, as well as PM's suitability for computing tasks [12], [26]. DeepTM takes this optimization a step further, with a specific emphasis on read-and-write performance within heterogeneous memory systems. DeepTM departs from traditional hierarchical storage architectures, involving GPUs, main memory, and SSDs, to establish a flat data cache architecture that optimizes performance in both fast and slow memory tiers at an extremely fine granularity level.

Optimization Strategies on heterogeneous memory systems: Addressing the complexities of heterogeneous memory management typically involves two prevailing optimization approaches: heuristic greedy algorithms and globally-oriented brute-force search methods. Many studies opt for greedy algorithms, which make local decisions to maximize immediate gains [18], [21], [29], [36]. However, relying solely on local optimization may lead to suboptimal global performance. A subset of research endeavors to achieve global optimization through mathematical methods, but these approaches can incur higher costs and lower accuracy that leads to OOM errors, particularly when applied to deep and complex models [12], [28], [44]. Reinforcement Learning (RL) has emerged as an alternative approach to addressing these challenges [31], [32], [33], capitalizing on RL-based agents' ability to find a balanced solution among competing objectives. DeepTM adopts deep reinforcement learning to attain high-precision global optimization strategies with minimal cost and overhead, offering a refined solution to the cost-effectiveness balance within PM systems.

VIII. Discussion

Reflecting on the recent shifts within the persistent memory landscape, including Intel's phase-out of Optane PM and Samsung's introduction of memory-semantic SSDs, it becomes evident that the realm of persistent memory is evolving dynamically. This evolution presents an opportunity for our DeepTM system, which is inherently designed to be adaptable and flexible across a wide range of memory technologies. The emergence of memory-semantic SSDs and advancements in the unified memory specification, which facilitate operations on PM-resident data structures, are particularly promising developments. Our DeepTM system, with its flexible DRL optimizer, is designed to accommodate across a variety of heterogeneous memory frameworks. Specially, DeepTM's advanced memory performance analysis and management can make CXL's unified memory management more efficient. In a CXL environment, DeepTM can also perform prefetching and copying of different types of memory in parallel with computation. Besides, CXL's large memory pools will enable dynamic and scalable allocation, ideal for the demands of DNN training, which prompts us to leverage the advantages of the CXL platform in our future work to optimize the training of larger models. This flexibility ensures the efficiency of DNN training processes, making DeepTM a robust solution adaptable to the changing dynamics of memory technology. While the current iteration of DeepTM has already been theoretically validated to be compatible with a wide range of memory architectures, our future work is committed to expanding this adaptability further.

IX. Conclusion

We propose DeepTM, an efficient solution designed for training large-scale deep learning models in heterogeneous memory systems. DeepTM significantly enhances heterogeneous memory access efficiency and overall training performance. With efficient aggregation management of memory pages and the global DRL optimization strategy through multi-characteristic analysis, DeepTM successfully addresses the key challenges of low-efficiency access to heterogeneous memory and sub-optimal memory management strategies. Our experiments demonstrate that DeepTM can achieve remarkable performance improvements, with up to a 49% boost over AutoTM and a 36% enhancement over Sentinel. Additionally, DeepTM offers cost savings of over 29% while maintaining the same performance level and reduces overhead by 18 times compared to AutoTM.

References

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[2] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2014, pp. 580–587.

[3] R. Collobert and J. A. Weston, "Unified architecture for natural language processing: Deep neural networks with multitask, in *Proc. 25th Int. Conf. Mach. Learn.*, 2008, pp. 160–167.

[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.

[5] C. Szegedy et al., "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1–9.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[7] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, 2015, Art. no. 436.

[8] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI Blog*, vol. 1, no. 8, p. 9, 2019.

[9] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali, "Single machine graph analytics on massive datasets using intel optane DC persistent memory," 2019, *arXiv:1904.07162*.

[10] "Speeding up flash... in a flash," The Inquirer. 2007–10-13, Archived from the original on Sep. 18, 2009. Retrieved 2014-01-11.

[11] N. Shazeer et al., "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," 2017, *arXiv: 1701.06538*.

[12] 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 *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2020, pp. 875–890.

[13] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," 2016, *arXiv:1603.04467*.

[14] "Pytorch," 2019. [Online]. Available: https://pytorch.org/

[15] S. Cyphers et al., "Intel nGraph: An intermediate representation, compiler, and executor for deep learning," 2018, *arXiv: 1801.08058*.

[16] M. Sivathanu, T. Chugh, S. S. Singapuram, and L. Zhou, "Astra: Exploiting predictability to optimize deep learning," in *Proc. Int. Conf. Architectural Support Programm. Lang. Operating Syst.*, 2019, pp. 909–923.

[17] O. Russakovsky et al., "Imagenet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, pp. 211–252, 2015.

[18] C. Huang, G. Jin, and J. Li, "Swapadvisor: Pushing deep learning beyond the GPU memory limit via smart swapping," in *Proc. Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2020, pp. 1341–1355.

[19] M. Kirisame et al., "Dynamic tensor rematerialization," 2020, *arXiv:2006.09616*.

[20] J. Jung, J. Kim, and J. Lee, "DeepUM: Tensor migration and prefetching in unified memory," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2023, pp. 207–221.

[21] 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 *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Piscataway, NJ, USA, 2016, pp. 18:1–18:13.

[22] "Intel Optane$^{\text{TM}}$ persistent memory performance sweep," 2021. [Online]. Available: https://github.com/intel/PM-perf-sweep

[23] J. Izraelevitz et al., "Basic performance measurements of the intel optane DC persistent memory module," 2019, *arXiv: 1903.05714*.

[24] B. CopelandJack, *Colossus: The Secrets of Bletchley Park's Code-Breaking Computers*. London, U.K.: Oxford Univ. Press, 2010.

[25] 2014. [Online]. Available: http://memkind.github.io/memkind/man_pages/memkind.html

[26] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning," in *Proc. IEEE Int. Symp. High- Perform. Comput. Archit.*, 2021, pp. 598–611.

[27] LLC Gurobi Optimization, *Gurobi Optimizer Reference Manual*, 2018.

[28] L. Wen, Z. Zong, L. Lin, and L. Lin, "A swap dominated tensor re-generation strategy for training deep learning models," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2022, pp. 996–1006.

[29] X. Peng et al., "Capuchin: Tensor-based GPU memory management for deep learning," in *Proc. Proc. Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2020, pp. 891–905.

[30] A. Shah, C.-Y. Wu, J. Mohan, V. Chidambaram, and P. Krahenbuhl, "Memory optimization for deep networks," 2020, *arXiv:2010.14501*.
[31] M. T. Islam, S. Karunasekera, and R. Buyya, "Performance and cost-efficient spark job scheduling based on deep reinforcement learning in cloud computing environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 7, pp. 1695–1710, Jul. 2022.
[32] Z. Cao, C. Lin, M. Zhou, and R. Huang, "Scheduling semiconductor testing facility by using cuckoo search algorithm with reinforcement learning and surrogate modeling," *IEEE Trans. Automat. Sci. Eng.*, vol. 16, no. 2, pp. 825–837, Apr. 2019.
[33] L. Jiang, H. Huang, and Z. Ding, "Path planning for intelligent robots based on deep Q-learning with experience replay and heuristic knowledge," *IEEE/CAA J. Automatica Sinica*, vol. 7, no. 4, pp. 1179–1189, Jul. 2020.
[34] Z. Duan et al., "Gengar: An RDMA-based distributed hybrid memory pool," in *Proc. Int. Conf. Distrib. Comput. Syst.*, 2021, pp. 92–103.
[35] 2017. [Online]. Available: https://github.com/PM/pmdk.git
[36] L. Wang et al., "Superneurons: Dynamic GPU memory management for training deep neural networks," in *Proc. 23rd ACM SIGPLAN Symp. Principles Practice Parallel Programm.*, 2018, pp. 41–53.
[37] A. M. Maia, Y. Ghamri-Doudane, D. Vieira, and M. F. de Castro, "Optimized placement of scalable IoT services in edge computing," in *Proc. IFIP/IEEE Symp. Int. Netw. Serv. Manage.*, 2019, pp. 189–197.
[38] S. Burer and A. N. Letchford, "Non-convex mixed-integer nonlinear programming: A survey," *Surv. Oper. Res. Manage. Sci.*, vol. 17, no. 2, pp. 97–106, 2012.
[39] X. Wang, J. Wang, X. Wang, and X. Chen, "Energy and delay tradeoff for application offloading in mobile cloud computing," *IEEE Syst. J.*, vol. 11, no. 2, pp. 858–867, Jun. 2017.
[40] A. G. Howard et al., "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv: 1704.04861*.
[41] P. Markthub et al., "DRAGON: Breaking GPU memory capacity limits with direct NVM access," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2018, pp. 414–426.
[42] J. Bae et al., "FlashNeuron: SSD-enabled large-batch training of very deep neural networks," in *Proc. 19th USENIX Conf. File Storage Technol.*, 2021, pp. 387–401.
[43] Z. Yang et al., "λ-IO: A unified IO stack for computational storage," in *Proc. 21st USENIX Conf. File Storage Technol.*, 2023, pp. 347–362.
[44] O. Beaumont, L. Eyraud-Dubois, and A. Shilova, "Optimal GPU-CPU offloading strategies for deep neural network training," in *Proc. 26th Int. Conf. Parallel Distrib. Comput.*, Springer, 2020, pp. 151–166.
[45] J. Izraelevitz et al., "Basic performance measurements of the intel optane DC persistent memory module," 2019, *arXiv: 1903.05714*.
[46] B. Daase et al., "Maximizing persistent memory bandwidth utilization for OLAP workloads," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 339–351.
[47] L. Benson, H. Makait, and T. Rabl, "Viper: An efficient hybrid PMem-DRAM key-value store," in *Proc. VLDB Endowment*, vol. 14, no. 9, pp. 1544–1556, 2021.
[48] Z. Lin et al., "P2CACHE: Exploring tiered memory for in-kernel file systems caching," in *Proc. USENIX Annu. Tech. Conf.*, 2023, pp. 801–815.
[49] Y. Chen et al., "FlatStore: An efficient log-structured key-value storage engine for persistent memory," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2020, pp. 1077–1091.
[50] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 169–182.
[51] T. Lee et al., "Memtis: Efficient memory tiering with dynamic page classification and page size determination," in *Proc. 29th Symp. Operating Syst. Princ.*, 2023, pp. 17–34.
[52] H. T. Kassa et al., "Improving performance of flash based key-value stores using storage class memory as a volatile memory extension," in *Proc. USENIX Annu. Tech. Conf.*, 2021, pp. 821–837.
[53] S. Rajbhandari et al., "Zero-infinity: Breaking the GPU memory wall for extreme scale deep," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2021, pp. 1–14.
[54] Microsoft, "DeepSpeed: Extreme-scale model training for everyone," 2020. [Online]. Available: https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scalemodel-training-for-everyone/
[55] H. Zhang, Y. E. Zhou, Y. Xue, Y. Liu, and J. Huang, "G10: Enabling an efficient unified GPU memory and storage architecture with smart tensor migrations," in *Proc. 56th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2023, pp. 395–410.
[56] J. Izraelevitz et al., "Basic performance measurements of the intel optane DC persistent memory module," 2019, *arXiv: 1903.05714*.
[57] D. Zhou, Y. Qian, V. Gupta, Z. Yang, C. Min, and S. Kashyap, "ODINFS: Scaling PM performance with opportunistic delegation," in *Proc. 16th USENIX Symp. Operating Syst. Des. Implementation*, 2022, pp. 179–193.

Haoran Zhou received the BS degrees in computer science and technology from Wuhan University, in 2022. He is currently working toward the PhD degree with the School of Computer Science, Wuhan University and Research Assistant from the Laboratory of Internet of Things for Smart City, University of Macau. His research interests include memory management and distributed computing.

Wei Rang received the BS degree in computer science from Shandong Normal University, China, in 2013, the MS degree in computer Sscience from Southern Illinois University Carbondale, in 2017, and the PhD degree in computer science from the University of North Carolina, Charlotte, in 2021. His research interests mainly focus on Cloud Computing and Parallel Computing.

Hongyang Chen (Senior Member, IEEE) received the BS and MS degrees from Southwest Jiaotong University, Chengdu, China, in 2003 and 2006, respectively, and the PhD degree from The University of Tokyo, Tokyo, Japan, in 2011. He is currently a senior research expert with Zhejiang Lab. His research interests include data-driven intelligent systems, graph machine learning, Big Data mining, and intelligent computing. He is also an adjunct professor with Hangzhou Institute for Advanced Study, The University of Chinese Academy of Sciences, and Zhejiang University, China.

Xiaobo Zhou (Senior Member, IEEE) received the BS, MS, and PhD degrees in computer science from Nanjing University, in 1994, 1997, and 2000, respectively. Currently he is a distinguished professor with IOTSC and Department of Computer and Information Science, University of Macau, Macau SAR. His research lies broadly in sistributed systems and cloud computing. He serves as the chair of IEEE Technical Community in Distributed Processing 2020-2023.

Dazhao Cheng received the BS degree in electrical engineering from the Hefei University of Technology, in 2006, the MS degree from the University of Science and Technology of China, in 2009, and the PhD degree from the University of Colorado, Colorado Springs, in 2016. He was an AP with the University of North Carolina at Charlotte, in 2016–2020. He is currently a professor with the School of Computer Science at Wuhan University. His research interests include Big Data and cloud computing.