# TierTrain: Proactive Memory Tiering for CPU-Based DNN Training

Sathvik Swaminathan
Intel Labs
Bangalore, India
sathvik.swaminathan@intel.com

Sandeep Kumar
Intel Labs
Bangalore, India
sandeep4.kumar@intel.com

Aravinda Prasad
Intel Labs
Bangalore, India
aravinda.prasad@intel.com

Sreenivas Subramoney
Intel Labs
Bangalore, India
sreenivas.subramoney@intel.com

## Abstract

Deep neural networks (DNNs) are one of the popular models for learning relationships between complex data. Training a DNN model is a compute- and memory-intensive operation. The size of modern DNN models spans into the terabyte region, requiring multiple accelerators to train –driving up the training cost. Such humongous memory requirements shift the focus toward memory rather than computation.

CPU- memory, on the other hand, can be scaled to several terabytes with new emerging memory technologies such as HBM and CXL-attached memories. Furthermore, recent advancements to the CPUs in terms of dedicated instructions for DNN training and inference are bridging the compute gap between CPUs and accelerators.

Proposed is an exploratory work in the direction of cost-effective DNN training on CPUs where we aim to alleviate memory management challenges in DNN training. We propose *TierTrain*, a novel memory tiering solution based on a dynamic queuing system that leverage the periodic and deterministic memory access behavior in DNN training to manage data placement across memory tiers. TierTrain proactively manages tensors by aggressively offloading them to slow memory tiers (NVMM, CXL) and timely prefetching them back to fast memory tiers (HBM, DRAM).

Our evaluation of TierTrain on a tiered memory system with a real CXL-attached memory used for memory expansion and NVMM as a low cost memory results in average fast memory footprint reduction of 59–83% and peak fast memory footprint reduction of 25–74% with a performance overhead of 1–16%. In a memory-constrained scenario, TierTrain outperforms the state-of-the-art tiering by improving the performance by 35–84% for a set of popular DNN training models.

## CCS Concepts

• **Computing methodologies** → **Neural networks**; **Neural networks**; • **Computer systems organization** → **Heterogeneous (hybrid) systems**.

## Keywords

DNN training, memory, tiering, hybrid memory, tiered memory

## 1 Introduction

Deep Neural Networks (DNNs) are widely popular [11, 40, 41, 48, 66] and are critical for advancing the AI-first initiative that, today, has permeated almost every facet of our lives. Training a DNN model is a compute- and memory-intensive task that has grown exponentially over the past few years [26, 54]. The unprecedented surge in model size and deeper networks needs systems to be provisioned with more and more memory, thus demanding rapid memory scaling. This is leading to the shift in focus towards memory rather than compute, referred to as the memory wall problem [26]. Even though GPUs with high compute throughput [13, 54] are typically used for training DNN models, they increasingly suffer from the memory wall problem with larger models and deeper networks. This is because the limited GPU memory cannot fit the entire model, resulting in data movement to and from GPU memory, which in turn can overshadow the GPU compute benefits. Multiple GPUs are required to complete the process of a large DNN model.

We propose TierTrain, an exploratory work that aims at alleviating the memory challenges associated with training large models with CPU memory. Our work is a step in the direction of radical thinking of training a large model on CPUs. CPU-based general-purpose computing, which is

nowadays gaining traction in DNN training [28, 32, 54, 57], does not suffer from data movement overheads, as the entire physical memory is byte-addressable from the CPU. Furthermore, emerging memory technologies like CXL, HBM, and NVMM memories enable the scaling of CPU memory to several terabytes. Recently, there has been a thrust towards cost-effective DNN training[33] to bring the training of large models within the reach of everybody, and not just a few big players who can afford thousands of GPUs.

However, in a CPU-based system, physical memory provisioned with traditional DRAM is unable to cope with the capacity and cost scaling challenges of modern workloads, including DNN training [28, 59, 71, 78]. This has led to the adoption of tiered memory systems that have multiple memory tiers with different cost-performance-capacity spectra [50, 52, 55, 67, 69, 81]. In such systems, software is responsible for placing data across low-latency, low-capacity *fast* memory tiers and low-cost, high-latency, high-capacity *slow* memory tiers [50, 55, 67, 81].

The core principle of effective memory tiering is to hide the cost of accessing data from a slow memory tier by placing frequently accessed data in the fast tier. Traditional tiering solutions follow a *reactive* approach where the data migration between fast and slow memory tiers is triggered based on the monitored memory access pattern (telemetry) of the application in a certain time window. Hence, such tiering solutions require precise telemetry, which in turn requires aggressive monitoring of the application's memory access pattern, which impacts performance [58, 62].

*TierTrain* is a context aware memory tiering solution for CPU-based DNN training designed based on our characterization of multiple DNN workloads [20, 35, 36, 77]. The core concept for tiering is based on our observation that the memory access pattern of a CPU-based DNN training workload is *periodic* and *deterministic*. We leverage this observation to design a *proactive* tiering solution, which has a significant advantage over the state-of-the-art reactive general-purpose tiering solutions.

TierTrain manages tensors by aggressively offloading them to slow memory tiers and timely prefetching them to fast memory tiers based on the memory access pattern during different stages of the DNN training. This approach allows training deeper-layered models with the same amount of fast memory or training an existing model with less amount of fast memory. Our evaluation of TierTrain on a tiered memory system with a real CXL-attached memory used for memory expansion and NVMM as a low cost memory results in average fast memory savings of 59–83% and peak fast memory reduction of 25–74% with a performance overhead of 1–16%. In a memory-constrained scenario, TierTrain outperforms the state-of-the-art tiering by 35–84% for a set of popular DNN models.

The key contributions of the paper are as follows:

(1) We perform a thorough characterization of DNN training using different models and input to gain insights into memory access pattern, memory usage pattern, and life cycle of allocated data for DNN training (see Section 4).
(2) We propose an analytical queuing system that performs dynamic memory tiering during different execution phases of the DNN training workload based on various factors such as layer execution duration, size of the tensors, and system state.
(3) We comprehensively evaluate TierTrain on a real CXL-attached memory and Intel's Optane DC PMM to demonstrate its robustness on memory media with different latency and bandwidth characteristics.

## 2 Background

In this section, we explain the necessary background required for the rest of the paper.

### 2.1 DNN Training

Deep Neural Networks or DNNs are multi-layer neural networks with an input layer, multiple hidden layers, and an output layer [27]. The hidden layers allow DNNs to capture the complex relationships in data. During a DNN training, ❶ the input data is passed through the whole network, producing an output. In an $N$-layered network, we have:

$$Y_n = \sigma(W_n \cdot X_n) \tag{1}$$

Where $X_n$ and $Y_n$ are the input and output to layer $n$, respectively, $W_n$ is the weight matrix for the layer $n$, and $\sigma$ is the activation function [21]. $Y_n$ is then passed as input to the next layer, where it becomes $X_{n+1}$. This is repeated for every layer and is referred to as a *forward pass*. ❷ Once the output of all the layers is calculated, the loss is calculated with respect to the ground truth. ❸ After the loss computation is done, the *backward pass* begins. In this pass, for each layer, the gradient for the weights is calculated. This step requires the output that was generated in the forward pass of the same layer. This whole process – forward pass and backward pass – is repeated for multiple iterations, known as **epochs** until the loss converges – marking the end of the training process.

### 2.2 Memory Tiering

In order to reduce the memory cost and accommodate the growing memory requirements of modern applications, data centers typically deploy tiered memory systems where a single system is equipped with different memory tiers at different cost-performance-capacity spectra [22, 47, 52, 55, 67, 69, 81]. A *fast* memory tier such as DRAM has lower access latency, but is expensive and has limited memory capacity.

Whereas a *slow* memory tier such as CXL-attached memory [73] and NVMMs [4, 25] has a higher memory capacity and lower cost [25, 56], but has higher access latency. A slow memory tier allows a cost-effective solution to satisfy the growing memory needs of modern applications.

However, memory TCO savings with a slow memory tier is *not free* as the data stored in such a tier incurs performance overhead upon data access. Hence, prior works have explored novel and interesting data *tiering* techniques to effectively exploit memory tiers that place frequently accessed hot data in fast memory tiers and less frequently accessed warm or cold data in slow memory tiers [22, 47, 52, 55, 67, 69, 81].

## 3 Motivation

In this section, we provide a brief overview for the motivation of our work to optimize DNN training on CPUs.

### 3.1 Limitation with GPU-Based Training

DNN models are typically trained on GPUs as they provide high compute throughput [13, 54], which has grown on a year-to-year basis and cater well to the high compute demands of DNN training. In order to use the full potential of a GPU's compute resource, all the data has to be copied to the GPU memory. However, GPUs fail to meet the fast-growing memory requirements of DNN training and suffer from low memory capacity on a single node [38, 39, 65, 70]. Whereas the compute power of GPUs has grown by 410×, with proposals to further extend it by leveraging multiple GPUs [6], the memory capacity in GPUs has only grown by 2× [26] in the past two years. Consequently, the size of the state-of-the-art DNN model has already crossed the capacity of an enterprise class GPU [44, 46]. Furthermore, the promised *unified memory model* [54], or UM, where GPU and CPU memory share a common address space, is still not widely used due to severe performance overhead caused by expensive GPU page faults [44].

Apart from this, the skyrocketing price of GPUs is also a matter of concern among academia and mid-size industries as it restricts their ability to utilize and contribute significantly to the AI boom. Thus the ability to innovate in this space is mainly limited to a few large players [54, 72].

### 3.2 CPU-Based Training Gaining Traction

Training DNN models on CPUs has been a concern due to limited parallel processing capabilities, memory bandwidth, and optimization in training frameworks. However, new frontiers are being explored by the hardware and software community to address the computing aspect of training a DNN model on CPUs. The latest generation of CPUs from Intel [45, 60] and Apple [1] have dedicated AMX (Advanced Matrix eXtensions) instructions that enable fast general matrix multiplications (GEMM) – a core operation in DNN training [5, 76]. These CPUs also have support for new datatypes such as `int8` and `bf16` [61].

Software frameworks are also being optimized to leverage these latest features to extract maximum compute power from the CPUs. For example, latest Intel Xeon Scalable processors significantly improves training and inference performance compared to their previous generations [7, 61]. Efforts are underway to leverage multiple-CPU architecture to extract GPU-like performance using the latest advancements in compiler technology [86], developing algorithms specific to CPU such as Sub-Linear Deep Learning Engine, or SLIDE [15, 37], and proposing changes to the framework to make it CPU friendly [14, 17, 18, 74, 75, 80].

Furthermore, CPUs are preferred for low-latency inference tasks as there is no driven offload, enabling a close interaction between DNN and non-DNN workloads. Data centers are typically over-provisioned to handle maximum nodes and usually have idle CPUs. These idle CPUs can be leveraged to train large DNN models without investing for costly GPUs [23, 28, 31, 54, 61].

**Summary:** CPU-based DNN training is a viable, practical and economical solution which is gaining traction. While enormous amount of work has been previously done for optimizing GPU-based DNN training, we strongly believe it is time to look into and optimize CPU-based DNN training. This paper attempts to optimize DNN training on issues concerning memory, especially the memory wall problem, by considering the emerging trends in memory hardware and server design space, specifically targeting tiered memory systems.

## 4 DNN Training Characterization

We characterize CPU-based DNN training frameworks to gain insights into the impact of memory management subsystems on the overall performance. We analyze the life cycle of critical data structures such as tensors, and factors that drive the average and peak memory requirements of different DNN training frameworks.

We evaluate 6 DNN models, including two Graph Neural Networks (GraphSAGE [35], GAT [77]), three Convolutional Neural Networks (ResNet-34, ResNet-50, ResNet-152 [36]), and a Transformer (Vision Transformer [20]). We use standard Intel PEBS hardware counters [42] to collect the relevant statistics for our analysis. System configuration and workload details are mentioned in Section 7.

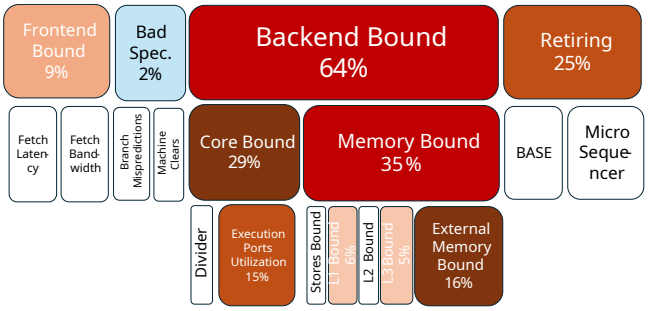



**Figure 1: Top-down analysis that shows CPU-based DNN training workloads are primarily memory bound.**

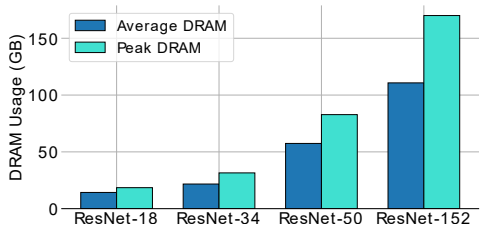



**Figure 2: Average and peak DRAM usage for ResNet models with different layers on the same input data set**

## 4.1 Top-Down Analysis

We quantifying the impact of memory on the overall execution time using a *top-down analysis* [84]. To identify DNN-training performance bottlenecks we measure high level metrics such as frontend bound, backend bound, bad speculation, retiring and then zoom into the dominant performance bottlenecks at each level. Our DNN training workload is run on DRAM with tiering disabled for the top-down analysis. As shown in Figure 1, the top-down analysis reveals that DNN training is predominantly (64%) backend bound. The next level shows that the workload is 29% core bound and 35% memory bound, implying that most stalls can be attributed to the memory system. Drilling further down into the memory bound category reveals that the workload is mainly bound by the external memory (16%). Hence, the DNN training workload is impacted by the latency and bandwidth of the memory media.

## 4.2 Memory Footprint

A deep learning model employs multiple layers to perform increasingly complex tasks and capture relationships between large amounts of data, such as in speech and audio. Increasing the number of layers can allow a model to learn complex data relationships [29].

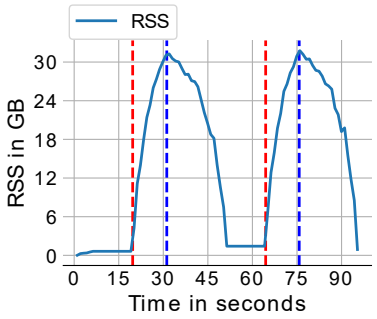



**Figure 3: Memory usage (resident set size (RSS)) during training of ResNet-34 on CIFAR-10 for 2 epochs. The dashed red and blue vertical lines mark the beginning of forward and backward pass**

However, the challenge with having more layers is the increase in compute and memory resources needed to train more parameters [68]. This is mainly because, the output tensors of the forward pass of each layer must be stored in memory to compute gradients in the backward pass. As networks grow deeper, the number of such tensors that need to be stored grows proportionately, increasing the average and peak memory requirements. Figure 2 shows the average and peak memory required to train ResNet model with 18, 50, and 152 layers. As the network depth grows from 18 to 152 layers, the average and peak memory requirements increase by 8.3× and 6.1×, respectively. However, memory is already accounting for 33–50% of TCO [3, 81]. To execute deeper networks more memory should be provisioned on the system which can significantly increase memory TCO.

**Takeaway:** This motivates the need for technologies that can enable execution of deeper layered networks without significantly increasing the memory cost.

## 4.3 Tensor Memory Management

Memory is a key aspect of DNN training. Hence, DNN training frameworks such as PyTorch manage tensors for optimal performance [8, 64]. Figure 3 shows the memory usage during the training of ResNet-34 with PyTorch for a total of 2 epochs. There are three important observations on how memory is used in each training epoch. ❶ At the beginning of the training epoch, PyTorch allocates memory for certain tensors representing the input dataset, weights, and the model itself. These tensors stay alive during the complete training process. ❷ During the execution of every layer, PyTorch allocates temporary tensors for use during that particular layer which are deallocated as the layer finishes its execution. ❸ During the execution of a layer in the forward pass, memory is also allocated for the "saved" tensors that do not get freed

**Table 1: Idle time of saved tensors in different layers for ResNet-34 on CIFAR-10**

| Layer ID | Functional block | Idle time |
|---|---|---|
| 1 | Conv. Layer | 30 sec |
| 2 | Batch Norm. | 29 sec |
| 3-8 | 3 Residual Blocks (3x2 Conv. Layers) | 20 sec |
| 9-16 | 4 Residual Blocks (4x2 Conv. Layers) | 14 sec |
| 17-28 | 6 Residual Blocks (6x2 Conv. Layers) | 4 sec |
| 28-33 | 3 Residual Blocks (3x2 Conv. Layers) | ≈ 0 sec |
| 34 | Fully connected Layer | ≈ 0 sec |

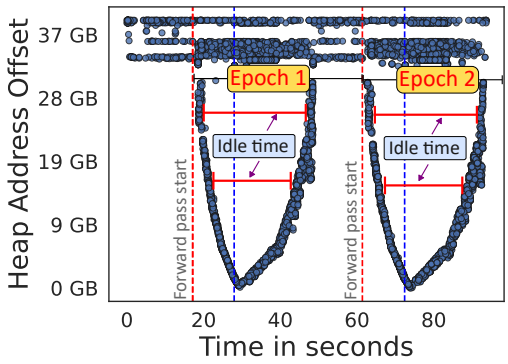



**Figure 4: Memory access pattern clearly indicating tiering opportunity with DNN training workloads. Each dot in the plot represents an accessed page. The dashed red and blue vertical lines mark the beginning of forward and backward pass.**

immediately after the execution of that layer. Instead these tensors are retained as they are used by the corresponding layers in the backward pass to compute loss gradients by the autograd engine. They are freed once the gradient computation is complete. These saved tensors contribute majorly to the overall memory footprint consumed during the training epochs.

## 4.4 Tiering Opportunity

For effective memory tiering, accurately identifying and placing hot and cold data sets in fast and slow memory is critical. We profile memory accesses by the DNN training workload to gain insights into memory access pattern in the process's virtual address space. We use Intel PEBS [42] to capture the memory access pattern by specifically tracking the virtual addresses of the retired `load` and `store` instructions.

**Idleness of saved tensors:** A saved tensor generated in a particular layer in the forward pass is accessed again only in the corresponding layer in the backward pass to compute the gradient. Thus, these saved tensors are *idle* till they are accessed again in the backward pass. Given that the backward pass processes layers in the opposite order of the forward pass, we observed the idle time of saved tensors generated in initial layers is significantly higher than the final layers (see Table 1). Figure 4 shows the memory access pattern for ResNet-34 for 2 epochs. Each dot in the plot represents an accessed page. The memory access pattern clearly shows the different idle times for saved tensors mapped at difference address space.

**Takeaway:** The idle time for tensors in each layer is the key observation for our proactive tiering technique. Migration of saved tensors that are idle during forward pass can be performed proactively by identifying them at the end of each layer's execution. This also avoids delay in migrating idle tensors; there is no need to wait for collecting page access profile over a window of time as in the reactive tiering techniques based on telemetry (60-120 seconds profiling window is used in production data centers [50]). In addition, this approach completely avoids the performance overheads associated with telemetry, such as checking and setting accessed bits in page tables or overheads due to PMU interrupts.

## 4.5 Importance of Timely Prefetch

Prefetching migrated idle tensors to fast memory tier on-time, i.e., before it is actively accessed again in backward pass, is critical. Failing to do so will result in accesses to the slow memory tier, severely impacting the performance. To assess the performance impact, we run DNN training workload from fast DRAM memory and slow Optane memory. Accessing tensors from Optane results in a slowdown of 15×.

| Setting | Epoch runtime (sec) |
|---|---|
| All DRAM | 30 |
| All Optane | 457 (15× increase) |

**Challenges in prefetching:** To avoid performance penalty, evicted tensors should be ideally prefetched on-time, i.e., before they are required in the backward pass. Initiating a prefetch at the beginning of the same layer in the backward pass will be too late resulting in all the accesses served from the slow tier, causing significant performance loss.

A naive approach such as prefetching idle tensors for layer $n$ at the beginning of layer $n$+1 (note that layers are processed in reverse order in the backward pass) is not an idle solution. As the size of the tensors and execution time for each layer can significantly vary, their prefetching time will also vary. Triggering a prefetch at the start of the previous layer may result in either a too-early or a too-late prefetch. The former increases the fast memory usage while the latter increases the performance overhead.

In this paper we propose a queuing system that efficiently migrates and prefetches idle tensors to and from slow memory tiers on-time.

# 5 Design

In this section, we discuss the design principles of TierTrain. We propose an efficient and *proactive* tiering solution custom designed for CPU-based DNN training workloads. One of the design goals is to minimize the performance impact on memory tiering on DNN workloads. Hence, our approach leverages the periodic and deterministic memory access behavior of DNN training workloads, instead of telemetry, for efficient memory tiering (see Section 4).

**Benefits over tradition tiering:** Traditional tiering work that relies on the telemetry events such as hardware counters of manipulating bits in the page table entry to generate a hotness profile cannot make aggressive and timely migration decisions as they have to wait for a data to become hot or cold. Instead, TierTrain's unique approach of directly hooking into the training framework allows it to perform context-aware *proactive* memory tiering.

## 5.1 Design Principle

**Aggressive eviction:** Evict a saved tensor from fast to slow memory as soon as the corresponding layer that generated the tensor finishes. This will ensure immediate reduction of fast memory footprint.
**Timely prefetch:** An evicted tensor must be prefetched back to the fast memory before it is accessed again in the backward pass. The prefetching must be timed properly – prefetching too early results in higher fast memory consumption, whereas a delay in prefetching results in memory accesses to slow memory, which negatively impacts the performance (see Section 4.5).

## 5.2 Design Overview

As shown in Figure 5, TierTrain collects the execution time of each layer in the forward pass and backward pass. This information is used in the subsequent epochs to evict data and ensure a timely prefetch.

**Eviction:** From second epoch onwards, in the forward pass, TierTrain is triggered when a layer finishes executing. Based on the information collected from hooks or triggers, the TierTrain's daemon, performs the following actions: identifies the saved tensors that can be evicted and the time when the prefetch for this tensor should be triggered. If on-time prefetching is not possible for the entire tensor, it calculates the optimal size for partial eviction of the tensor and computes the prefetch trigger time accordingly. It may decide to skip the eviction entirely and retain the tensor in fast memory (see Section 5.3). Evicts the data pages associated with the tensors identified for eviction in the previous step and schedules a prefetch request.

**Prefetching:** The prefetching schedule ensures that the evicted idle tensors are kept in the slow tier as long as possible and are prefetched on-time to fast memory tiers to avoid performance penalty of accessing the tensors from the slow tier. In the backward pass, TierTrain periodically checks if a tensor needs to be prefetched. If yes, then a a prefetching request is issued.

## 5.3 Queuing System

We present a queuing system, the *core* of TierTrain. Figure 6 shows the training of a DNN model along with the key structures and events such as the start and end time of different layers in the forward and backward pass and the size of saved tensors in each layer. The system processes all the information collected and issues eviction and prefetch requests.

**Idle time:** A saved tensor generated in the forward pass of a layer is used again in the same layer in the backward pass. The *idle time* ($\mathbf{IT_l}$) of a layer $l$ is defined as follows:

$$\mathbf{IT}_l = ts_l^B - te_l^F \tag{2}$$

Here, $ts_l^B$ and $te_l^F$ is the start and end time of the layer $l$ in the backward and forward pass, respectively. For efficient tiering, for each tensor migration request, *there should be enough time to evict it and prefetch it on-time.*

$$\mathbf{IT}_l \geq evict\_time_l + prefetch\_time_l \tag{3}$$

Here, $evict\_time_l$ and $prefetch\_time_l$ is the time taken to evict tensor from fast to slow memory and prefetch the same tensor from slow to fast memory for layer $l$, respectively.

*5.3.1 Tensor Eviction.* The eviction rate from fast memory to slow Memory ($ER$) and prefetching rate from slow memory to fast memory ($PR$) determines the time taken for eviction and prefetching, respectively. It can be noted that for memory technologies such as Optane with asymmetric read and write latency $ER$ and $PR$ can be different. In addition, system state such as number of worker threads performing tensor migration and memory bandwidth limit enforced by the user determines $ER$ and $PR$.
Using Equation 3, we get

$$\mathbf{IT}_l \geq \frac{sz_l}{ER} + \frac{sz_l}{PR} \tag{4}$$

$$\boxed{\mathbf{IT}_l \geq sz_l * MR} \tag{5}$$

Here, $sz_l$ is the size of the tensor for layer $l$ and $MR$ is the migration rate defined as $MR = (\frac{1}{ER} + \frac{1}{PR})$.

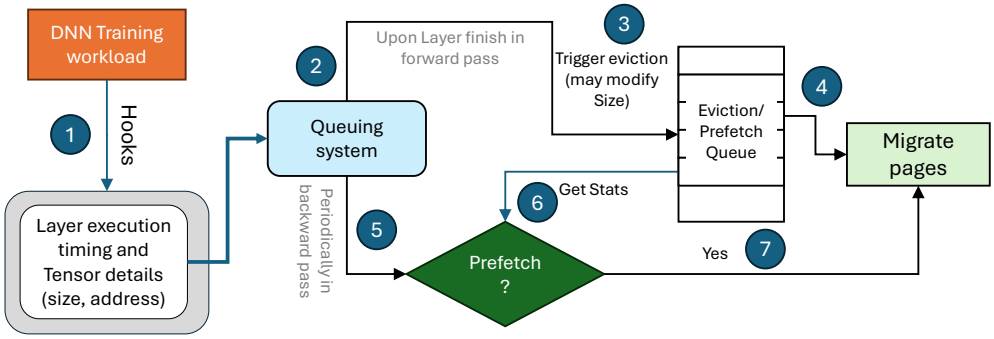



**Figure 5: An overview of working of TierTrain**

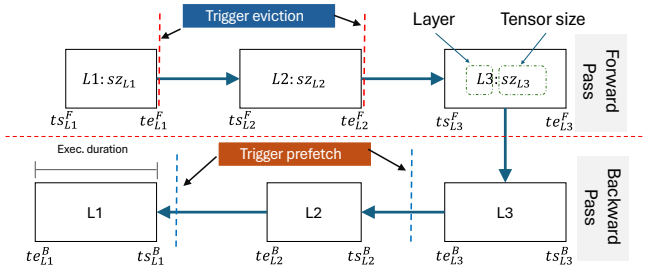



**Figure 6: An overview of the working of the queuing system.**

While deciding whether to evict or not, the system does the following:

$$evict? = \begin{cases} \text{Accept, } sz_l^E = sz_l & \text{if } \mathbf{IT}_l \geq sz_l * MR + \mathbf{ST}, \\ \text{Modify size} & \text{else} \end{cases} \quad (6)$$

Here, $sz_l^E$ is the size of tensor that can be evicted based on the idle time. **ST** or *stay time* is a constant that defines the minimum amount of time an evicted tensor should reside in the slow memory tier before it is prefetched to the fast memory. **ST** is used to avoid ping-ponging of tensors between slow and fast memory tiers.

**Modify eviction request:** If the system cannot evict the complete tensor due to idle time constraints, it attempts to evict a partial tensor. The partial size is calculated as follows:

$$\boxed{sz_l^E \leq \frac{(\mathbf{IT}_l - \mathbf{ST})}{MR}} \quad (7)$$

*5.3.2 Tensor Prefetching.* An evicted tensor ($sz_l^E$; either complete or partial) must be prefetched back to the fast memory before it is accessed again in the backward pass. It should be ensured that the system can prefetch it back on time by calculating a prefetch trigger time.

$$t\_start\_pre_l = ts_l^B - \frac{sz_l^E}{PR} \quad (8)$$

Here, $t\_start\_pre_l$ is the time to start prefetching for the tensor in layer $l$.

**Scheduling prefetch request:** While scheduling a prefetching request, we need to ensure that the prefetcher is not busy in servicing other prefetch requests. To do so, the system maintains a counter ($t\_busy$) that indicates till what time the prefetcher is busy. In the beginning, $t\_busy = 0$ indicating it is ready to service prefetch requests.

Now, for a successful prefetching of $sz_l^E$:

$$prefetch? = \begin{cases} \text{Accept, } sz_l^P = sz_l^E & \text{if } t\_busy \leq t\_start\_pre_l, \\ \text{Modify size} & \text{else} \end{cases} \quad (9)$$

Here, $sz_l^P$ is the size of the tensor that can be prefetched on time accounting for $t\_busy$.

**Modify prefetching request:** If the prefetcher is busy, the size is modified to accommodate for $t\_busy$. As the start time of the layer in the backward pass cannot be modified and prefetching earlier is not possible due to $t\_busy$, modifying the size of the request is the only possibility. Using Equation 8

$$sz_l^P \leq (ts_l^B - t\_busy) * PR \quad (10)$$

*5.3.3 Final eviction & prefetching.* If $sz_l^P <= 0$ then the request is dropped, else $sz_l^P$ is the final size of the tensor for eviction and a prefetching which will be triggered at time $(ts_l^B - \frac{sz_l^P}{PR})$. $t\_busy$ is updated to $ts_l^B$ to reflect the busy time.

## 6 Implementation

In this section, we discuss the implementation details of TierTrain.

**Algorithm 1** Using hooks used in PyTorch by TierTrain

```
procedure FORWARD_HOOK(output_tensors)
    ▷ Called at the end of every layer in forward pass
    send_to_TierTrain(size, addr, timestamp)
end procedure
procedure BACKWARD_HOOK
    ▷ Called at the beginning of every backward pass
    send_to_TierTrain(timestamp)
end procedure
procedure MAIN
    model ← create PyTorch model
    handle ← model.register_forward_hook
    handle ← model.register_backward_hook
end procedure
```

**Table 2: Systems configuration**

| Optane System Settings | | |
|---|---|---|
| Model: Intel(R) Xeon(R) Gold 6238M | DRAM: | 768 GB |
| CPUs: 4 Socket, 22 Cores, 2 HT | Optane: | 4.5 TB |
| **CXL System Settings** | | |
| Model: Intel(R) Xeon(R) CPU Max 9480 | DRAM: | 2 TB |
| CPUs: 2 Socket, 56 Cores, 2 HT | CXL: | 256 GB |
| **Software Settings** | | |
| Linux Kernel: 6.6.3\| Huge pages: always\| DVFS: Performance \| ASLR: Off | | |

## 6.1 Data Profiling

We implement TierTrain in the PyTorch framework. Specifically, we exploit framework "hooks" that are triggered every time a layer execution begins and when tensors are saved in the forward pass. This information is passed to TierTrain, which is implemented as a daemon. We use the standard `forward_hook()` and `backward_hook()` available in PyTorch to implement these trigger points. Algorithm 1 shows a high level overview of hooks used by TierTrain.

## 6.2 Page Migration

TierTrain uses the standard `move_pages` [19] system to call available in Linux to move pages between fast and slow memory tiers (each tier is a NUMA node in Linux). To speed up eviction and prefetching, multiple threads are used to move pages across tiers. We experiment with different thread count and analyze the impact on the numbers of tensors selected or dropped for eviction. The sensitivity analysis with respect to number of threads is provide in Section 7.6.2. The implementation ensures that the eviction of identified tensors is performed immediately after a layer finishes its execution in the forward pass.

# 7 Evaluation

## 7.1 Experiment Setup

Table 2 lists the system configuration of our test beds. Table 3 shows different configurations and input data set used for benchmarking along with epoch runtime, average and peak memory usage in an all DRAM setup (all allocations on DRAM) without tiering. These values serve as our baseline. We use DRAM as fast memory tier and either Optane or CXL-attached memory as the slow memory tier.

**Table 3: Workloads used for evaluation.**

| Model | Model Type | DRAM (GB) | | Epoch (sec) | Dataset |
|---|---|---|---|---|---|
| | | Avg. | Peak | | |
| GAT | GNN | 79.5 | 120.4 | 344.6 | ogbn-products |
| GraphSAGE | GNN | 117 | 57.6 | 50.5 | |
| ViT | Transformer | 18.7 | 35.6 | 24.2 | CIFAR-10 |
| ResNet-34 | CNN | 21.7 | 31.3 | 35.5 | |
| ResNet-50 | CNN | 57.4 | 82.8 | 87 | |
| ResNet-152 | CNN | 110.7 | 170 | 170.1 | |

To evaluate TierTrain, we use the PyTorch Framework. Specifically, we use the *DGL* [80] library to run a *4-layered* Graph Attention Network (GAT), *4-layered* GraphSAGE Network, *torchvision* [63] library to run ResNet34, ResNet50, ResNet152, and the *HuggingFace* [82] library to run Vision Transformer (ViT).

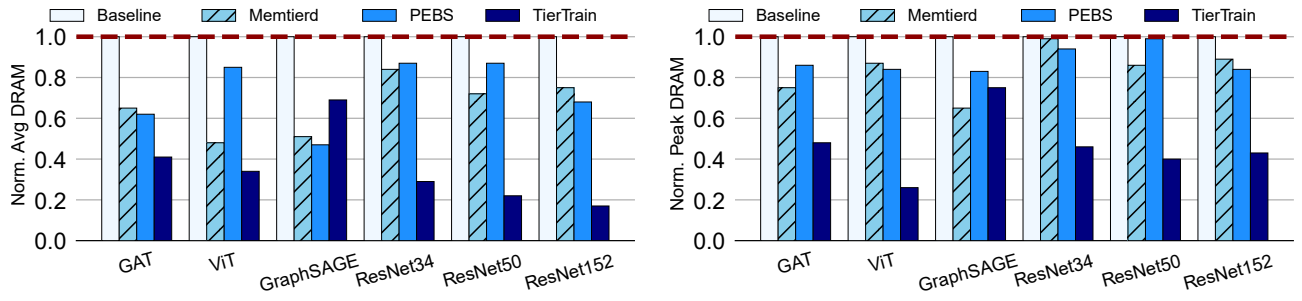
We compare the performance of TierTrain with memory tiering solutions based on idle-bit tracking and hardware performance counters. For the former, we use Memtierd [43], an open source tiering solution, with the default page migration policy. For the latter, we implemented a solution similar to HeMem [67] that builds a hotness profile based on hardware counters using Intel PEBS. Instead of a static threshold as used in HeMem, we perform tiering with a dynamic hotness threshold of 25-th percentile, i.e., in a profile window, all the data pages with hotness more than the 25th%lie stays in DRAM, and the rest is evicted to Optane.

We also compare TierTrain with AutoNUMA-based [16] tiering. AutoNUMA recently introduced a tiering mode where it can tier hot and cold data based on the hotness captured using NUMA faults. Unfortunately, prior works focusing on memory tiering for DNN training either have not made their code public [68] or have outdated and unmaintained dependencies [38] and hence we are unable to compare with them.

## 7.2 Evaluation Strategy

Our evaluation strategy is as follow:

(1) **Average and peak DRAM reduction:** We measure the capability of TierTrain to reduce average and peak DRAM consumption. Reduction in average and peak DRAM consumption demonstrates memory savings as such systems can be provisioned with less costly DRAM.
(2) **DRAM constraints:** We measure the tiering efficiency of TierTrain when the DNN training workload footprint




(a) Reduction in Avg. DRAM

(b) Reduction in Peak DRAM

**Figure 7: Reduction in average and peak DRAM memory for TierTrain compared with Memtierd, hardware counters (PEBS), and AutoNUMA based memory tiering solutions. In the baseline setting the entire workload runs in DRAM with tiering disabled**

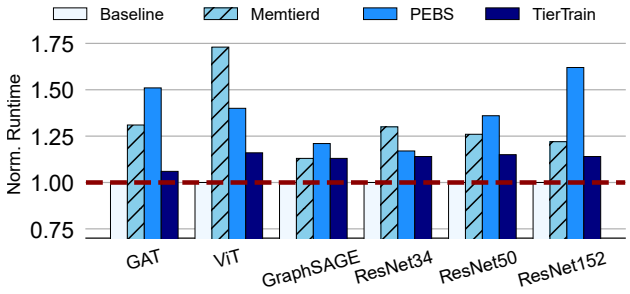
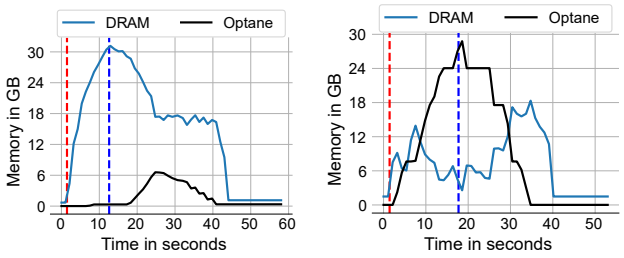



**Figure 8: Normalized runtime of TierTrain compared to other tiering solutions. In the baseline setting the entire workload runs in DRAM with tiering disabled**




(a) Memtierd

(b) TierTrain

**Figure 9: Memory footprint for a single epoch for ResNet-34 for Memtierd and TierTrain. The dashed red and blue vertical lines mark the beginning of forward and backward pass**

exceeds the DRAM capacity. This demonstrates the impact of tiering in DRAM constrained scenarios i.e., when DRAM is unable to cope with the capacity demands of DNN training workloads thus requiring memory to be expanded with slow memory.

(3) **CXL memory:** We evaluate on a real CXL-attached memory to future-proof our solution on emerging CXL memory technology. This also demonstrates the robustness of TierTrain on memory media with different latency and bandwidth characteristics.

In addition, our evaluation includes a deep-dive into the TierTrain's queuing system along with a sensitivity analysis with different number of threads for migration.

## 7.3 Average and Peak DRAM Reduction

As can be seen in Figure 7, TierTrain outperforms Memtierd by an average of 2× and 3.17× in terms of average DRAM and peak DRAM usage, respectively, while improving the overall epoch runtime by 66% (see Figure 8). TierTrain outperforms PEBS by an average of 2.5× and 4.5× in terms of average DRAM and peak DRAM usage, respectively, while improving the overall epoch runtime by 71% (see Figure 8).

**Discussion:** Memtierd tracks the idle bit in the page table entry of a data page periodically and builds a hotness profile based on that. However, it takes multiple scanning of the address space to build a profile – and a tiering solution based on that results in a delayed or sub-optimal tiering and hence, Memtierd migrates only a small amount of data to Optane (see Figure 9a). An aggressive address space scanning to profile the bits in the page table entry consumes a lot of compute resources and can severely impact the performance of the DNN training workload [58].

TierTrain outperform Memtierd and PEBS as it relies on hooks to extract data directly from the framework. Our measurement shows that hooks incur a negligible overhead of less than 0.05%. TierTrain's non-telemetry based aggressive

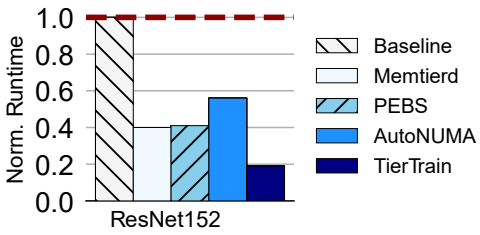



**Figure 10: Performance comparison of TierTrain with other tiering solutions in DRAM *constrained* setting for ResNet 152. Tiering is disabled in the baseline. DRAM capacity is 65 GB, Optane capacity is 1.5 TB, and peak memory footprint is 170 GB.**

eviction approach evicts saved tensors as soon as the completion of the execution of the layer. This can be seen by the increase in the Optane memory consumption with the execution of each layer in the forward pass (Figure 9b). During the backward pass, these tensors are moved from Optane to DRAM in a timely manner before they are accessed again – as seen by the drop in Optane consumption during the backward pass. As, TierTrain manages to prefetch all the tensors right before the corresponding layer in the backward pass starts, the overall impact on the performance is also limited to 2–16%.

*7.3.1 Deeper Layers.* As discussed in Section 4.2, as the network grows deeper, the memory footprint increases. The tiering opportunity also increases as ❶ the number of tensors that can be tiered increases and ❷ the idle time of tensors also increases with starting layers having the largest idle time. TierTrain efficiently leverages this increased tiering opportunity. As shown in Figure 7a, for ResNet models when the number of layers is increased from 18 to 34 and to 152, the reduction in average DRAM usage also improves from 71% to 78%, and to 83%.

## 7.4 DRAM-Constrained Scenario

Memory consumed by DNN training workloads scales with the complexity of the deep neural network and the size of the training data, in which case, the memory footprint of the workload might exceed the provisioned DRAM capacity. In such DRAM constrained scenarios, data pages spills into Optane memory. But as a hot data page placed in Optane severely impacts the performance (see Section 4.5), tiering solutions handle such scenarios by demoting cold data from DRAM to Optane, making room for ❶ promoting hot data from Optane to DRAM and ❷ future allocations in DRAM.

We demonstrates the efficiency of TierTrain in DRAM constrained scenarios by off-lining DRAM memory blocks. We bring down the total DRAM capacity to 65 GB which is less than the peak memory footprint of 170 GB for ResNet-152 model. Pages are allocated as per the default first-touch allocation policy in the Linux kernel, where pages are first allocated in DRAM before spilling over to Optane. Tiering is disabled in the baseline.

It can be observed from Figure 10 that TierTrain outperforms Memtierd, PEBS, and AutoNUMA by 35%, 37%, and 84%, respectively in terms of epoch execution time. Figure 11 shows the memory distribution across DRAM and Optane with baseline, AutoNUMA, Memtierd and TierTrain.

**Discussion:** We observe that AutoNUMA demotes ≈5 GB of data to Optane. However, increase in Optane memory consumption in Figure 11 for AutoNUMA is due to spill over from DRAM. Whereas TierTrain timely demotes ≈150 GB of data to Optane. Aggressive demotion of idle tensors by TierTrain results in tensors for the subsequent layers allocated in DRAM. As a result, the total number of Optane accesses come down by ≈ 58% for TierTrain compared to ≈ 16% reduction for AutoNUMA (see Figure 12). The cycles to serve L3 misses also drops by ≈ 62% for TierTrain compared to ≈ 28% for AutoNUMA as more requests are served from DRAM. The total number of dTLB page walk cycles also sees an improvement, mainly due to page table pages getting allocated in DRAM. As a result, the overall memory stall cycles comes down by ≈ 72% for TierTrain compared to ≈ 30% for AutoNUMA.

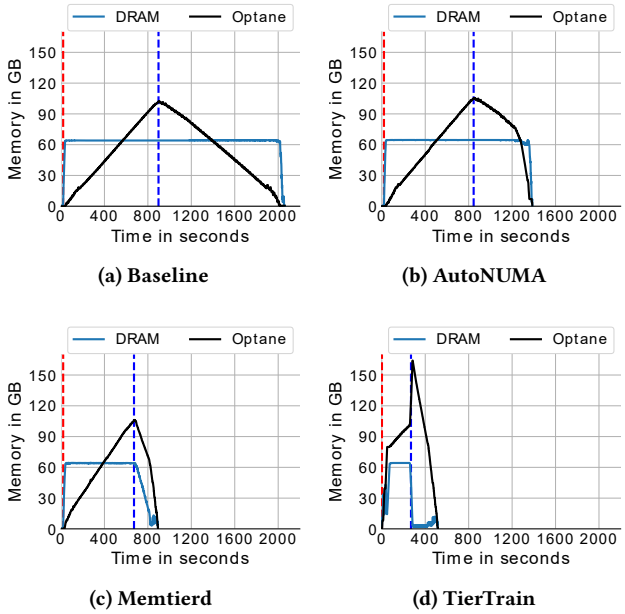
## 7.5 CXL-Attached Memory Tier

CXL or Compute eXpress Link [73] is an upcoming memory interconnect that allows the memory in a system to be scaled to terabytes. We evaluate TierTrain with a real CXL-attached memory as a slow memory tier and DRAM as a fast memory tier. As shown in Figure 13, the average and peak DRAM reduces by 48–59% and 45–51%, respectively with a performance overhead of 1–5%. In our test bed, the access latency to CXL-attached memory is better than Optane and hence results in a lower performance overhead as compared to 14–15% overhead with Optane.

These set of experiments demonstrates that TierTrain's queuing system is robust in handling both CXL and Optane-based slow memory tiers. TierTrain efficiently manages memory tiering, which is evident from significant reduction in average and peak DRAM consumption for both Optane and CXL-attached memory tiers.

## 7.6 Deep Dive

We perform a deep dive analysis into some of the system parameters of TierTrain including the performance impact of number of threads used for migrating idle tensors from DRAM to Optane and vice-a-versa.




(a) Baseline (b) AutoNUMA

(c) Memtierd (d) TierTrain

**Figure 11: Memory distribution across DRAM and Optane for a single epoch for ResNet-152 in a DRAM constrained setting. The dashed red and blue vertical lines mark the beginning of forward and backward pass.**

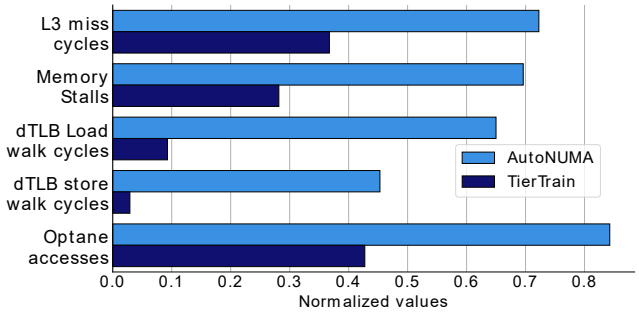
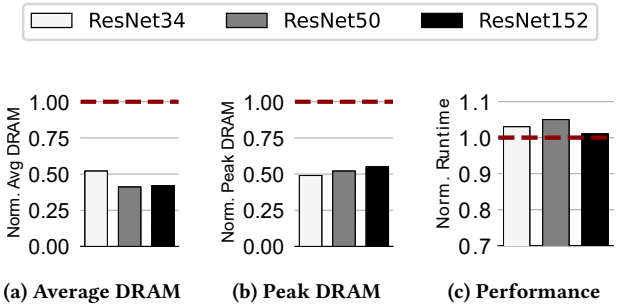



**Figure 12: Hardware performance counters in DRAM constrained setting normalized to baseline execution where tiering is disabled.**

*7.6.1 Queuing System.* As discussed in Section 5, we use a constant "stay time" or **ST** in order to ensure that there is no ping pong between fast and slow memory tier. **ST** determines the minimum time an evicted tensor should stay in the slow tier. We evaluate the impact of **ST** on DRAM usage and application performance using ResNet-34 as the representative workload. Table 4 shows how the size of total tensors migrated varies with different values of **ST**. A value of 0 for **ST** removes all the restrictions and hence, TierTrain migrated all the saved tensors to slow memory tier. As the




(a) Average DRAM (b) Peak DRAM (c) Performance

**Figure 13: Comparison of average and peak DRAM reduction along with epoch runtime for CXL-attached memory tier with TierTrain for ResNet models. Baseline is all DRAM with memory tiering disabled.**

**Table 4: Impact of Stay Time (ST) on number of eviction/prefetching requests for ResNet-34.**

| Stay Time (ST) | #Dropped requests | #Modified requests | Migrated data |
|---|---|---|---|
| 0 sec | 0 | 0 | 29.37 GB |
| 1 sec | 2 | 0 | 29.1 GB |
| 5 sec | 2 | 1 | 25.57 GB |
| 10 sec | 3 | 0 | 24.6 GB |
| 20 sec | 3 | 1 | 19.07 GB |

value of **ST** increases, TierTrain imposes a strict constraint on migrated tensors. As a result, TierTrain either starts dropping requests or modifying them as per the queuing system thus bringing down the amount of date migrated.

*7.6.2 Sensitivity Analysis.* Apart from **ST**, the amount of data migrated between fast and slow memory also depends on the number of threads used to migrated the data. Using more number of threads (upto a certain limit) increases the amount of data migrated in a given time window (i.e., migration rate in Equation 5). Hence, using more threads to migrate data can speed up the overall migration process, aggressively bringing down the average and peak DRAM usage. However, it also increases the performance overhead as the DNN training workload and tensor migration contend for memory bandwidth.

We use ResNet-34 as the representative workload to show the impact of different number of threads on the overall memory savings and performance overhead. Table 5 shows how varying the number of threads used to migrate idle tensors affects DRAM consumption and the runtime of each epoch. As the number of migration threads increase from 1 to 5, the amount of total migrated data increases from 13 GB to $\approx$ 30 *GB*. However, page migration increases the total number of dTLB flushes by 8.6×. This results in the total

**Table 5: Results showing sensitivity to the number of threads used for migrating idle tensors for ResNet-34 with TierTrain**

| #Th-reads | Data Migrated | DRAM Reduction | | Perf. | Norm. TLB flushes |
|---|---|---|---|---|---|
| | | Peak | Avg. | Ovh. | |
| 1 | 13.4 GB | 45.89% | 36.56% | 1.36% | 1 |
| 2 | 26.7 GB | 70.68% | 52.33% | 5.83% | 4.1 |
| 4 | 28.6 GB | 75.21% | 56.87% | 9.56% | 7.9 |
| 5 | 29.5 GB | 78.84% | 59% | 14% | 8.6 |

**Table 6: A summary of generic tiering solutions focusing on memory tiers supported and page placement policy.**

| Method | Stats | Trigger | Summary |
|---|---|---|---|
| TMO [81] | Stalls | LRU-based | Reduce memory TCO by evicting unused pages to swap. |
| Auto-NUMA [55] | LRU & NUMA | Faults-based | Aims for efficient tiering based on faults on DRAM and CXL devices. |
| HeMem [67] | PEBS | Events-based | Memory tiering based on a static hotness threshold |
| Memtis [52] | PEBS | Events-based | Memory tiering based on hotness reported by PEBS. |
| Memtierd [43] | Idle-bits | Events-based | Memory tiering based on the idle bits in PTE |

performance overhead to also increase from 1.36% to 14% due to higher number of dTLB misses.

## 8 Related Work

Several tiered memory systems have been proposed in recent years [9, 10, 12, 22, 30, 34, 47, 49, 50, 52, 53, 55, 67, 81, 83], along with data placement and migration policies to optimize performance and memory TCO. A typical setting is that a system is configured with a fast memory tier (DRAM or HBM) and a slow memory tier (NVMM, compressed memory, or CXL). The tiering solutions aim to better utilize the memory tiers present in a system by ensuring minimal performance loss. The tiering solutions can be tuned to perform tiering in a memory pressure condition [52, 55, 67] or reduce fast memory costs by keeping only necessary data in the fast memory [50, 81].

### 8.1 DNN-Specific Tiering

There is a plethora of work that looks at solving the issue of limited GPU memory [2, 39, 51, 65, 70, 72, 79] by leveraging CPU memory and doing efficient *tiering* so as to hide the cost of accessing data in CPU memory from GPU. Prior work in the space of memory tiering specific to DNN training (see Table 7 ) either uses high-overhead telemetry [68, 70], a rigid decision system bound to a particular model or set of memory tiers [68, 85], is not scalable due to the high computations associated with making the placement decision [38], has an all-or-nothing migration policy [38, 68, 70, 85], is on the critical path [38, 70], or cannot adapt to the dynamic nature of the system [38, 44, 68, 70, 85].

Prior work [44, 68] uses a page-table scanning approach to classify the data as hot cold. Any page table scanning approach requires modification to the bits in a page table entry and a TLB shootdown at a high frequency for the whole execution duration to capture a precise memory access pattern – resulting in a high performance ovheread [24, 58].

**Model agnostic:** A technique is scalable if the computational complexity of the data migration policy does not significantly go up with the model size. AutoTM[38] optimizes training runtime by mapping the network as an integer linear programming (ILP). The ILP formulation is designed to assign values to every tensor, deciding which memory tier they should be placed in. The ILP formulation is computationally expensive to solve and also becoming increasingly complex to compute with an increase in model size.

**Partial migration:** Several prior work [38, 68, 70, 85] in this space follows an *all-or-nothing* migration approach. AutoTM [38] assigns a value to each tensor (binary in case of a two-tier system), which dictates in which memory tier the tensor should be placed. AutoTM either migrates the whole tensor or none of it, as per the results from the ILP solver. It *does* not allow for partial migration of tensors.

**Off Critical Path:** AutoTM [38] performs tensor migration by inserting "`move nodes`" into the computational graph, which makes the tensor migration enter the critical path of the application and increases its runtime. vDNN [70] evicts tensors of a layer as soon as the layer's forward pass is done and prefetches it back before its backward pass begins. However, the subsequent layer in the computational graph cannot begin until the eviction and prefetch operations are completely done, which leads to a performance loss of 58% for their static DNN-based workloads.

**Holistic Policy and Dynamic Tuning**: A tiering solution should be aware of all the dynamics of a system, such as the characteristics of the memory tiers, the dynamic load on the system, memory bandwidth utilization, etc. and should dynamically adapt to the different execution conditions. Sentinel [68] performs profiling of the first epoch and uses the collected profile for the rest of epochs – a rigid approach that is unaware of any execution changes due to resource contention, remapping of data structures [8], or any other system-level changes.

TierTrain does not require any changes to adapt to different DNN training models, and the same queuing-based eviction and prefetching framework can be used across models. The

**Table 7: Comparison with prior solutions for data tiering for DNN training**

| Method | Telemetry Technique | Policy | Model Agnostic | Scalable? | Partial Migration | Holistic Policy | Off Critical Path | Dynamic Tuning |
|---|---|---|---|---|---|---|---|---|
| AutoTM | Graph Kernels | ILP solver | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Deep UM | Page Faults | Correlation Prefetching | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ |
| Sentinel | PTE poisoning | Adaptive Migration | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ |
| vDNN | Heuristics | Closest layer | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| CachedArrays | Custom APIs | User defined | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ |
| TierTrain | Hooks | Queuing model | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

queuing system in TierTrain does not require large computation resources and hence, the complexity of it does not grow with the model size. TierTrain also enables a key requirement of efficient tiering – support for partial migration of tensors. If eviction and prefetching of a whole tensor cannot be done in a timely fashion, then the queuing model calculates an optimal size, which is less than the original size that can be safely evicted and prefetched back in a timely manner. TierTrain executes the migration in the background through a daemon process, ensuring it stays out of the application's critical path. It also receives feedback on whether the tensors were evicted and prefetched back in a timely manner or not and adjusts the eviction and prefetch triggers accordingly in the next epoch.

## 9 Conclusion

With the increasing popularity of deep neural networks and the increase in the cost of training a large network, there is a thrust towards cost-efficient large network training. Due to the memory requirement of training such large models, the focus is shifting from computation to memory – which the latest generation of GPUs is unable to fulfill.

In this paper, we proposed an exploratory idea to address memory challenges by using CPU memory which can scale to terabytes. Our work is a step in the direction of a radical idea – training on CPUs, which opens it up to the general masses and not to a few wealthy key players.

TierTrain is a novel memory tiering solution aimed at addressing the memory wall problem for DNN training workloads with CPU memory. TierTrain demonstrates open opportunities in optimizing memory management for DNN training workloads with an efficient memory tiering solution based on a queuing system.

## References


[1] [n. d.]. corsix/amx: Apple AMX Instruction Set. https://github.com/corsix/amx. (Accessed on 07/30/2024).

[2] [n. d.]. cybertronai/gradient-checkpointing: Make huge neural nets fit in memory. https://github.com/cybertronai/gradient-checkpointing. (Accessed on 08/06/2024).

[3] [n. d.]. Decadal Plan for Semiconductors. https://www.src.org/about/decadal-plan/decadal-plan-full-report.pdf.

[4] [n. d.]. Intel® Optane™ DC Persistent Memory Product Brief. https://www.intel.in/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf. (Accessed on 08/01/2023).

[5] [n. d.]. Modular: AI's compute fragmentation: what matrix multiplication teaches us. https://www.modular.com/blog/ais-compute-fragmentation-what-matrix-multiplication-teaches-us. (Accessed on 07/30/2024).

[6] [n. d.]. NVIDIA NVLink and NVIDIA NVSwitch Supercharge Large Language Model Inference | NVIDIA Technical Blog. https://developer.nvidia.com/blog/nvidia-nvlink-and-nvidia-nvswitch-supercharge-large-language-model-inference/. (Accessed on 08/19/2024).

[7] [n. d.]. Tackling the Challenge of Bringing AI to HPC. https://www.hpcwire.com/2018/05/21/tackling-the-challenge-of-bringing-ai-to-hpc/. (Accessed on 10/18/2024).

[8] Martín Abadi, Paul Barham, Jianmin Chen, Zheng Chen, Andy Davis, Jeffrey Dean, Matt Devin, Sergey Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Berkeley, CA, USA, 265–283.

[9] Neha Agarwal and Thomas F. Wenisch. 2017. 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* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 631–644. doi:10.1145/3037697.3037706

[10] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *Data Management on New Hardware* (Philadelphia, PA, USA) *(DaMoN'22)*. Association for Computing Machinery, New York, NY, USA, Article 8, 5 pages. doi:10.1145/3533737.3535090

[11] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jin Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Gregory Frederick Diamos, Erich Elsen, Jesse Engel, Linxi (Jim) Fan, Christopher Fougner, Awni Y. Hannun, Billy Jun, Tony Xiao Han, Patrick LeGresley, Xiangang Li, Libby Lin, Sharan Narang, A. Ng, Sherjil Ozair, Ryan J. Prenger, Sheng Qian, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Anuroop Sriram, Chong-Jun Wang, Yi Wang, Zhiqian Wang, Bo Xiao, Yan Xie, Dani Yogatama, Junni Zhan, and Zhenyao Zhu. 2015. Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin. In *International Conference on Machine Learning*. https://api.semanticscholar.org/CorpusID:11590585

[12] Moiz Arif, Kevin Assogba, M. Mustafa Rafique, and Sudharshan Vazhkudai. 2023. Exploiting CXL-Based Memory for Distributed Deep Learning. In *Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP '22)*. Association for

Computing Machinery, New York, NY, USA, Article 19, 11 pages. doi:10.1145/3545008.3545054

[13] Ebubekir BUBER and Banu DIRI. 2018. Performance Analysis and CPU vs GPU Comparison for Deep Learning. In *2018 6th International Conference on Control Engineering & Information Technology (CEIT)*. 1–6. doi:10.1109/CEIT.2018.8751930

[14] Johannes Böhm, Michael Gschwind, Daniel Kossmann, et al. 2020. LIBXSMM: A High-Performance Library for Deep Learning. In *Proceedings of the 2020 IEEE International Conference on Big Data (Big Data)*. IEEE, Los Angeles, CA, USA, 1–8.

[15] Beidi Chen, Tharun Medini, James Farwell, Sameh Gobriel, Charlie Tai, and Anshumali Shrivastava. 2020. SLIDE : In Defense of Smart Algorithms over Hardware Acceleration for Large-Scale Deep Learning Systems. arXiv:1903.03129 [cs.DC] https://arxiv.org/abs/1903.03129

[16] Jonathan Corbet. 2012. AutoNUMA: the other approach to NUMA scheduling. *LWN. net* (2012).

[17] Intel Corporation. 2020. Efficient Deep Learning Inference on CPUs with OpenVINO Toolkit. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. IEEE, Seoul, South Korea, 1–7.

[18] Intel Corporation. 2021. Speed Up Deep Learning Framework Performance on Intel® Processors. In *Proceedings of the International Conference on Deep Learning*. Intel Corporation, Santa Clara, CA, USA, 1–6.

[19] Linux Kernel Documentation. 2023. Page Migration. https://docs.kernel.org/mm/page_migration.html Accessed: 2024-10-16.

[20] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2020. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *Proceedings of the 9th International Conference on Learning Representations (ICLR)* (Vancouver, Canada) *(ICLR '20)*. Neural Information Processing Systems Foundation, Vancouver, Canada, 1–12. https://arxiv.org/abs/2010.11929

[21] Shiv Ram Dubey. 2021. Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark. *arXiv preprint arXiv:2109.14545* (2021). https://arxiv.org/abs/2109.14545

[22] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, et al. 2023. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 727–741.

[23] Quchen Fu, Ramesh Chukka, Keith Achorn, Thomas Atta-fosu, Deepak R. Canchi, Zhongwei Teng, Jules White, and Douglas C. Schmidt. 2022. Deep Learning Models on CPUs: A Methodology for Efficient Training. *ArXiv* abs/2206.10034 (2022). https://api.semanticscholar.org/CorpusID:249888937

[24] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. BadgerTrap: a tool to instrument x86-64 TLB misses. *SIGARCH Comput. Archit. News* 42, 2 (sep 2014), 20–23. doi:10.1145/2669594.2669599

[25] Shivank Garg, Aravinda Prasad, Debadatta Mishra, and Sreenivas Subramoney. 2023. Motivating Next-Generation OS Physical Memory Management for Terabyte-Scale NVMMs. arXiv:2310.03370 [cs.OS]

[26] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. 2024. AI and Memory Wall. *IEEE Micro* 44, 3 (2024), 33–39. doi:10.1109/MM.2024.3373763

[27] Ashish Goel et al. 2022. An Architecture-Level Analysis on Deep Learning Models for Low-Impact Inference Workloads. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)* (Virtual Conference) *(AISTATS '22)*. Springer, New York, NY, USA, 1–10. doi:10.1007/s10462-022-10221-5

[28] Zhangxiaowen Gong, Houxiang Ji, Yao Yao, Christopher W. Fletcher, Christopher J. Hughes, and Josep Torrellas. 2022. Graphite: optimizing graph neural networks on CPUs through cooperative software-hardware techniques. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 916–931. doi:10.1145/3470496.3527403

[29] Google. 2024. Deep neural network models | Machine Learning | Google for Developers. https://developers.google.com/machine-learning/recommendation/dnn/softmax. (Accessed on 10/19/2024).

[30] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. 2023. Memory Pooling With CXL. *IEEE Micro* 43, 2 (2023), 48–57. doi:10.1109/MM.2023.3237491

[31] Nitesh Narayana GS, Marc Ordoñez, Lokananda Hari, Franyell Silfa, and Antonio González. 2023. ReuseSense: With Great Reuse Comes Greater Efficiency; Effectively Employing Computation Reuse on General-Purpose CPUs. arXiv:2311.10487 [cs.AR] https://arxiv.org/abs/2311.10487

[32] Yufeng Gu, Alireza Khadem, Sumanth Umesh, Ning Liang, Xavier Servot, Onur Mutlu, Ravi Iyer, and Reetuparna Das. 2025. PIM Is All You Need: A CXL-Enabled GPU-Free System for Large Language Model Inference. *arXiv preprint arXiv:2502.07578* (2025).

[33] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv preprint arXiv:2501.12948* (2025).

[34] Vishal Gupta, Min Lee, and Karsten Schwan. 2015. HeteroVisor: Exploiting Resource Heterogeneity to Enhance the Elasticity of Cloud Platforms. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Istanbul, Turkey) *(VEE '15)*. Association for Computing Machinery, New York, NY, USA, 79–92. doi:10.1145/2731186.2731191

[35] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS)* (Long Beach, California) *(NeurIPS '17)*. Neural Information Processing Systems Foundation, Long Beach, CA, USA, 1–11. https://cs.stanford.edu/people/jure/pubs/graphsage-nips17.pdf

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Las Vegas, NV, USA) *(CVPR '16)*. IEEE Computer Society, Los Alamitos, CA, USA, 770–778. https://arxiv.org/abs/1512.03385

[37] Wei He. 2023. The promise of training deep neural networks on CPUs: A survey. *Journal of Physics: Conference Series* 2649, 1 (nov 2023), 012017. doi:10.1088/1742-6596/2649/1/012017

[38] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 875–890. doi:10.1145/3373376.3378465

[39] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1341–1355. doi:10.1145/3373376.3378530

[40] Huili Huang and M. Mahdi Roozbahani. 2023. ScaleNet: An Unsupervised Representation Learning Method for Limited Information. *ArXiv* abs/2310.02386 (2023). https://api.semanticscholar.org/CorpusID:

49865868

[41] Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. 2023. VoxPoser: Composable 3D Value Maps for Robotic Manipulation with Language Models. *ArXiv* abs/2307.05973 (2023). https://api.semanticscholar.org/CorpusID:259837330

[42] Intel. 2023. PEBS (Processor Event-Based Sampling) Manual. https://xem.github.io/minix86/manual/intel-x86-and-64-manual-vol3/o_fe12b1e2a880e0ce-734.html

[43] Intel. 2024. GitHub - intel/memtierd. https://github.com/intel/memtierd.

[44] Jaehoon Jung, Jinpyo Kim, and Jaejin Lee. 2023. DeepUM: Tensor Migration and Prefetching in Unified Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 207–221. doi:10.1145/3575693.3575736

[45] Hyungyo Kim, Gaohan Ye, Nachuan Wang, Amir Yazdanbakhsh, and Nan Sung Kim. 2024. Exploiting Intel Advanced Matrix Extensions (AMX) for Large Language Model Inference. *IEEE Computer Architecture Letters* 23 (2024), 117–120. https://api.semanticscholar.org/CorpusID:270004187

[46] J. Kim et al. 2024. Scaling Beyond the GPU Memory Limit for Large Mixture-of-Experts Model Training. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*. PMLR, Vienna, Austria, 24342–24353.

[47] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 715–728. https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon

[48] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet classification with deep convolutional neural networks. *Commun. ACM* 60, 6 (may 2017), 84–90. doi:10.1145/3065386

[49] Sandeep Kumar, Aravinda Prasad, Smruti R. Sarangi, and Sreenivas Subramoney. 2021. Radiant: Efficient Page Table Management for Tiered Memory Systems. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management* (Virtual, Canada) *(ISMM 2021)*. Association for Computing Machinery, New York, NY, USA, 66–79. doi:10.1145/3459898.3463907

[50] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the 2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, Providence, RI, USA, 1–14. doi:10.1145/3297858.3304053

[51] Tung D. Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. 2019. TFLMS: Large Model Support in TensorFlow by Graph Rewriting. (2019). arXiv:1807.02037 [cs.LG] https://arxiv.org/abs/1807.02037

[52] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 17–34.

[53] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 574–587. doi:10.1145/3575693.3578835

[54] Youjie Li, Amar Phanishayee, Derek Murray, Jakub Tarnawski, and Nam Sung Kim. 2022. Harmony: overcoming the hurdles of GPU memory capacity to train massive DNN models on commodity servers. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2747–2760. doi:10.14778/3551793.3551828

[55] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 742–755. doi:10.1145/3582016.3582063

[56] MemVerge. 2024. CXL Use Case - Slash Memory Costs and Expand Capacity - MemVerge. https://memverge.com/cxl-use-case-slash-memory-costs-and-expand-capacity/. (Accessed on 10/19/2024).

[57] Sparsh Mittal, Poonam Rajput, and Sreenivas Subramoney. 2022. A Survey of Deep Learning on CPUs: Opportunities and Co-Optimizations. *IEEE Transactions on Neural Networks and Learning Systems* 33, 10 (2022), 5095–5115. doi:10.1109/TNNLS.2021.3071762

[58] Alan Nair, Sandeep Kumar, Aravinda Prasad, Ying Huang, Andy Rudoff, and Sreenivas Subramoney. 2024. Telescope: Telemetry for Gargantuan Memory Footprint Applications. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 409–424. https://www.usenix.org/conference/atc24/presentation/nair

[59] Prashant Nair, Chia-Chen Chou, and Moinuddin K. Qureshi. 2013. A Case for Refresh Pausing in DRAM Memory Systems. In *Proceedings of the 19th International Symposium on High-Performance Computer Architecture (HPCA)* (Shanghai, China) *(HPCA '13)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–12. doi:10.1109/HPCA.2013.6546270

[60] Nevine Nassif, Ashley O. Munch, Carleton L. Molnar, Gerald Pasdast, Sitaraman V. Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, Rafi Marom, Alexandra M. Kern, Bill Bowhill, David R. Mulvihill, Srikanth Nimmagadda, Varma Kalidindi, Jonathan Krause, Mohammad M. Haq, Roopali Sharma, and Kevin Duda. 2022. Sapphire Rapids: The Next-Generation Intel Xeon Scalable Processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. 44–46. doi:10.1109/ISSCC42614.2022.9731107

[61] Anant V. Nori, Rahul Bera, Shankar Balachandran, Joydeep Rakshit, Om J. Omer, Avishaii Abuhatzera, Belliappa Kuttanna, and Sreenivas Subramoney. 2021. REDUCT: Keep it Close, Keep it Cool! : Efficient Scaling of DNN Inference on Multi-core CPUs with Near-Cache Compute. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 167–180. doi:10.1109/ISCA52012.2021.00022

[62] SeongJae Park, Yunjae Lee, and Heon Y. Yeom. 2019. Profiling Dynamic Data Access Patterns with Controlled Overhead and Quality. In *Proceedings of the 20th International Middleware Conference Industrial Track* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 1–7. doi:10.1145/3366626.3368125

[63] Adam Paszke, Simon Gross, Francisco Massa, Adam Lerer, James Bradbury, Guy Chanan, Trevor Killeen, Zeming Lin, Nikhil Gimelshein, and Luca Antiga. 2016. Torchvision: Datasets, Transforms and Models for Computer Vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Las Vegas, NV, USA, 1–6.

[64] Adam Paszke, Simon Gross, Francisco Massa, Adam Lerer, James Bradbury, Guy Chanan, Trevor Killeen, Zeming Lin, Nikhil Gimelshein, Luca Antiga, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS)*. Neural

Information Processing Systems Foundation, Vancouver, Canada, 1–12.

[65] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 891–905. doi:10.1145/3373376.3378505

[66] Alec Radford, Luke Metz, and Soumith Chintala. 2016. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. arXiv:1511.06434 [cs.LG] https://arxiv.org/abs/1511.06434

[67] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 392–407. doi:10.1145/3477132.3483550

[68] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. 2021. Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 598–611. doi:10.1109/HPCA51647.2021.00057

[69] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. 2024. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems* (<conf-loc>, <city>Athens</city>, <country>Greece</country>, </conf-loc>) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 803–817. doi:10.1145/3627703.3650075

[70] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Article 18, 13 pages.

[71] SemiAnalysis. 2023. The Memory Wall: Past, Present, and Future of DRAM. In *Proceedings of the SemiAnalysis Conference* (New York, New York) *(SemiAnalysis '23)*. SemiAnalysis, New York, NY, USA, 1–10. https://www.semianalysis.com/p/the-memory-wall

[72] Xiaoyang Sun, Wei Wang, Shenghao Qiu, Renyu Yang, Songfang Huang, Jie Xu, and Zheng Wang. 2022. STRONGHOLD: Fast and Affordable Billion-Scale Deep Learning Model Training. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–17. doi:10.1109/SC41404.2022.00076

[73] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (<conf-loc>, <city>Toronto</city>, <state>ON</state>, <country>Canada</country>, </conf-loc>) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 105–121. doi:10.1145/3613424.3614256

[74] PyTorch Team. 2022. Enhancing PyTorch Performance on CPUs: Strategies and Techniques. In *Proceedings of the PyTorch Developer Conference*. Facebook AI Research, Menlo Park, CA, USA, 1–8.

[75] TensorFlow Team. 2020. Optimizing TensorFlow Performance on CPUs: Techniques and Best Practices. In *Proceedings of the TensorFlow Developer Summit*. Google LLC, Mountain View, CA, USA, 1–10.

[76] Michael Tschannen, Aran Khanna, and Anima Anandkumar. 2017. StrassenNets: Deep learning with a multiplication budget. In *International Conference on Machine Learning*. https://api.semanticscholar.org/CorpusID:30684700

[77] Petar Veličković, Guillem Cucurull, Alberto Casanova, Adriana Romero, P. Lio, and Yoshua Bengio. 2018. Graph Attention Networks. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)* (Vancouver, Canada) *(ICLR '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://arxiv.org/abs/1710.10903

[78] David Tawei Wang. 2005. Performance Analysis of Modern DRAM Memory Systems: Performance Analysis and a High Performance, Power-Constrained DRAM Scheduling Algorithm. College Park, MD, USA. https://user.eng.umd.edu/~blj/papers/thesis-PhD-wang--DRAM.pdf

[79] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: dynamic GPU memory management for training deep neural networks. *SIGPLAN Not.* 53, 1 (feb 2018), 41–53. doi:10.1145/3200691.3178491

[80] Minjie Wang, Zhiqiang Zhang, Jiayi Feng, Zonghan Liu, Jiong Chen, Yujia Wang, Wei Zhang, and Jian Huang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management (CIKM)*. Association for Computing Machinery, Beijing, China, 1–4.

[81] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 609–621. doi:10.1145/3503222.3507731

[82] Thomas Wolf, Julien Chaumond, Victor Sanh, Riccardo Moioli, Pierric Cistac, Clément Delangue, Lianhui Li, Antoine Girard, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45.

[83] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 331–345. doi:10.1145/3297858.3304024

[84] Ahmad Yasin. 2014. A Top-Down Method for Performance Analysis and Counters Architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (Austin, Texas) *(ISPASS '14)*. IEEE, New York, NY, USA, 35–44. https://www.semanticscholar.org/paper/A-Top-Down-method-for-performance-analysis-and-Yasin/6776ff919597ca4feccd413208dedc401f6e655d

[85] Wei Zhang, Hongyu Li, Xiaofei Wang, Minghua Chen, and Yunchuan Sun. 2023. Deep Learning-Based Intelligent Industrial Fault Diagnosis: A Comprehensive Review. *IEEE Transactions on Industrial Informatics* 19, 5 (2023), 3456–3471. doi:10.1109/TII.2023.3245678

[86] Wei Zhang, Xiaoyang Wang, and Jian Liu. 2021. Compiler Techniques for Accelerating CPU Performance in Deep Learning. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, San Diego, CA, USA, 1–10.

Received 2025-03-19; accepted 2025-05-03