Understanding Oversubscribed Memory Management for Deep Learning Training

Mao Lin
University of California, Merced
Merced, CA, USA
mlin59@ucmerced.edu

Hyeran Jeon
University of California, Merced
Merced, CA, USA
hjeon7@ucmerced.edu

Abstract

This paper analyzes the performance impact of unified virtual memory (UVM) while running large deep learning (DL) workloads on a GPU with limited memory capacity. Due to the page fault handling overhead of UVM, DL frameworks have not officially supported UVM. However, given the rising GPU price, the global semiconductor shortages, and the ever-increasing DL model size, we cannot optimistically rely on adding more GPUs for DL processing. Various other solutions such as compression and quantization reduce memory footprint at the cost of performance and accuracy. Using UVM with memory oversubscription could tackle such concerns. However, UVM has not been actively leveraged in DL computing due to its performance overhead. In this study, we investigate the performance impact of UVM for DL computing to better understand the benefits and limitations. Our results show that while UVM enables training large models beyond GPU capacity, its effectiveness depends on the interplay between oversubscription factors and memory management strategies. We find that PCA significantly mitigates page fault overhead, improving performance at moderate oversubscription levels, but also increases migration traffic. These findings highlight the potential of integrating UVM with advanced memory management strategies to optimize DL workloads on limited-memory GPUs.

CCS Concepts: • **Computing methodologies** → **Machine learning**; **Parallel computing methodologies**; • **Computer systems organization** → **Parallel architectures**.

Keywords: GPU, Unified Virtual Memory, Performance Analysis, DNN, LLM

EuroMLSys '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1538-9/25/03
https://doi.org/10.1145/3721146.3721955

ACM Reference Format:
Mao Lin and Hyeran Jeon. 2025. Understanding Oversubscribed Memory Management for Deep Learning Training. In *The 5th Workshop on Machine Learning and Systems (EuroMLSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3721146.3721955

1 Introduction

Deep learning (DL) models have become the flagship application for general-purpose graphics processing unit (GPGPU) computing, driven by the growing prevalence of AI technologies. The memory requirements of emerging DL workloads are enormous and continue to increase alongside the rapid evolution of AI. For instance, Meta's latest large language model (LLM), Llama-3, contains 405 billion parameters [14], demanding approximately 1.9 TB of GPU memory [37]—a 5.8× increase compared to its predecessor, Llama-2, which has 70 billion parameters [38]. However, NVIDIA's state-of-the-art GPU, the H100, offers only 80 GB of device memory [7], the same capacity as the previous generation [5]. This significant gap between the demands of modern AI workloads and the limitations of device memory underscores the urgent need for improved memory designs to address performance bottlenecks in GPU computing.

To address the challenge, several solutions have been proposed, such as multi-GPU parallelism [20, 25, 30, 43], data offloading to the host memory [19, 24, 27, 41], intermediate result recomputation [13, 32, 33, 40], and memory compression and quantization [12, 22, 26, 42]. While these methods mitigate the problem to some degree, they require a deep understanding of the individual workloads or expensive multiple GPUs, and more importantly remain constrained by the GPU memory capacity.

On the other hand, NVIDIA's Unified Virtual Memory (UVM), which provides on-demand memory migration and eviction between CPU and GPU memories, effectively expands GPU memory to larger CPU system memory. Thus, UVM could be synergistically integrated with the aforementioned approaches to further mitigate the memory capacity issues. However, its page-fault-based data transfer is known to cause a significant performance overhead [29, 36]. Recent studies showed better performance with hand-tuned memory management over UVM [21, 36].

However, most of them did not exploit the full capacity of UVM. First, earlier studies considered zero to limited (e.g., up

to 1.5 - 2×) memory oversubscriptions. However, a more severe memory oversubscription becomes more practical given the recent efforts from both industry and academia about disaggregated memory systems. Most recent technologies (e.g., CXL) focus on CPU memory expansion by integrating multiple disaggregated memory devices. The memory space of accelerators including GPUs could be indirectly enlarged through CPU's expanded memory. In such disaggregated memory platforms, GPUs could run huge applications with severely oversubscribed memories. However, recent studies evaluate up to 2× oversubscription. Also, existing DL studies using UVM mostly leveraged object-level tensor allocations. However, modern DL frameworks such as PyTorch and Tensorflow support an advanced memory management method (namely PyTorch Caching Allocator (PCA)) to reduce the memory allocation and deallocation overheads for tens to hundreds of tensors. A large chunk of GPU memory is allocated as a shared memory pool, where tensors are mapped and unmapped to and from the pooled memory without separate GPU memory allocations and deallocations. The pool size increases gradually as more tensors are allocated up to the entire GPU device memory capacity. There has not been a study that leveraged PCA for DL computing, which potentially limits the performance of UVM.

In this paper, we aim to provide insights into the perils and opportunities of adopting UVM at its full capacity for DL computing. We break the boundary of the pool size by integrating PCA with UVM. As UVM supports memory oversubscription, technically the pool size can be as large as the combined memory capacity of CPU and GPU. To the best of our knowledge, there has not been a study that evaluated the performance impact of such a huge memory allocation and dynamic tensor mapping in the context of UVM. We characterize the behaviors of UVM under PCA by comparing them with more conventional usage of UVM by disabling PCA and replacing individual tensor allocations and deallocations with UVM APIs.

The contributions of this paper are as follows.

- To the best of our knowledge, this is the first study, which provides an in-depth analysis of the UVM performance for DL computing under various memory oversubscription.
- We analyze the impact of PCA on UVM performance and compare the performance with conventional tensor-level memory management.
- Our results show that PCA effectively reduces the page fault handling overhead, thereby providing a comparable performance without using UVM when memory is not oversubscribed. When GPU memory is oversubscribed, UVM shows a scalable performance while it is impossible without UVM.

2 Background

2.1 UVM Architectures

In discrete CPU-GPU systems, the CPU and GPU memories are physically distinct and connected via the PCI-Express bus. Data shared between the CPU and GPU must be allocated in both memories and explicitly copied between them by the program. To ease the programming burden, NVIDIA introduced Unified Virtual Memory (UVM) in CUDA 6.0, enabling access to the memory shared by CPU and GPU using a single pointer [10]. Later, NVIDIA's Pascal architecture [9]introduced advanced features such as on-demand migration and memory oversubscription. When GPU applications allocate data (e.g., using UVM allocation APIs like `cudaMallocManaged`), data pages are automatically copied to GPU memory on-demand when accessed by the GPU kernel. This process is managed by the UVM driver and GPU Memory Management Unit (GMMU) using *page fault* handling. Page faults occur when data pages are not present in the GPU memory, triggering on-demand allocation or migration. During a memory transaction, the GMMU checks whether the data page resides in the GPU memory. If not, a page fault triggers and the UVM driver copies the required page from CPU memory to GPU memory. Although UVM facilitates seamless page sharing, this does not always lead to performance gains due to the latency associated with page fault handling, including on-demand migrations. To reduce page fault handling overhead, UVM employs a prefetching algorithm. Migrations are handled in two coarse-grained units: 2 MB and 64 KB [16], effectively prefetching multiple pages for each individual 4 KB page fault. Prefetching occurs from the host CPU to the GPU. Eviction from the GPU to the CPU does not use prefetching and instead utilizes a single migration unit of 2 MB [2].

UVM enables CPUs and GPUs to share a flat virtual memory system, allowing zero-copy pointer sharing and seamless data transfers. It also extends GPU memory capacity by utilizing CPU system memory, enabling applications to handle larger data sets—a concept known as *memory oversubscription*. To support this, the UVM driver evicts the least recently used (LRU) pages from GPU memory to CPU memory when space is insufficient for new pages. The severity of memory oversubscription is often quantified by the *oversubscription factor* defined as the ratio of allocated UVM memory to the available GPU memory [4]. An oversubscription factor of $\leq$ 1.0 indicates no oversubscription, while $>$ 1.0 means oversubscription, requiring dynamic page evictions. For example, an oversubscription factor of 1.5 on a GPU with 80 GB memory means 120 GB is allocated, necessitating frequent page evictions to accommodate the workload.

2.2 DNN Architectures

DL workloads use Deep Neural Network (DNN) architectures, which include Convolutional Neural Networks (CNNs),

Table 1. Evaluation Platform

CPU	GPU	System	System Memory	GPU Driver	CUDA Toolkit	Nsight Systems
Intel(R) Xeon(R) Gold 5320	NVIDIA A100 80GB PCIe	Linux 5.14	128 GB	550.90.12	12.1	v.2023.1.2

Table 2. Evaluated DL models

Model	Type	Layers	Architecture	Batch Size	Memory Footprint (MB)
AlexNet	CNN	8	Convolutional Full Connected	128	5316
ResNet50	CNN	50	Residual Block	32	15952
ResNet101	CNN	101	Residual Block	32	22588
GPT-2	Transformer	12	Transformer (Decoder)	8	12008
BERT	Transformer	12	Transformer (Encoder)	16	12350
Whisper (small)	Transformer	12	Transformer (En/De-coder)	16	9824

Recurrent Neural Networks (RNNs) and Graph Neural Networks (GNNs), and Transformers. Due to their architectural differences, these models exhibit varying strengths across different tasks. For example, CNNs excel in image processing, while Transformers are mainly used for a wide range of natural language processing tasks. These architectural and task-specific differences also lead to distinct computational and memory behaviors, which are critical considerations for performance optimization on UVM systems.

CNNs: CNNs [23] typically consist of a combination of convolutional layers, pooling layers, and fully-connected layers. Each neuron in the convolutional layers takes multiple data points from the weights and feature maps and runs the convolution operation (dot-product). The results are summarized by various activation functions (e.g., ReLU or SigMoid) and then processed by pooling layers that produce the maximum or average value out of the convolution results. The fully-connect layers are used for producing the final image classification outputs.

Transformers: Transformers [39], which form the foundation of large language models (LLMs), are composed of a stack of Transformer blocks, each comprising an attention layer and a subsequent feed-forward layer, with input embedding layers, and the fully connected task-specific last layers. The attention layers take multiple input tokens and check the relations among the tokens in parallel. With the multiple layers of attention computations, Transformers extract the most important words from the input sentences and produce proper answers for various tasks such as question answering and sentiment analysis.

2.3 Memory Management on Prominent DL Frameworks

Memory allocation and deallocation are often time-consuming and resource-intensive. DL workloads, composed of computational layers and numerous tensors, suffer performance degradation from frequent calls to `cudaMalloc` and `cudaFree`.

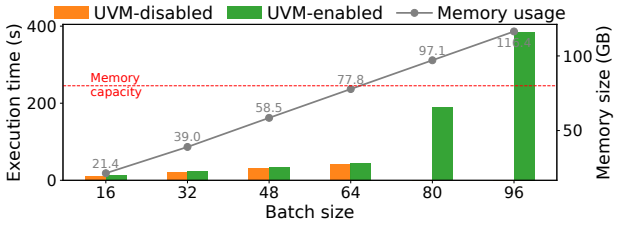


Figure 1. Comparison of GPT-2 performance with and without UVM and memory usage across varying batch sizes.

To address this issue, PyTorch [31] introduced the *PyTorch Caching Allocator* (PCA) [3], which leverages a pooling mechanism to reduce allocation and deallocation overhead. When a tensor is created, PCA requests a large memory chunk via `cudaMalloc` and allocates a portion to the tensor, storing the remainder in memory pools. For subsequent tensors, PCA first checks these pools; if memory is available, it assigns it to the tensor. Otherwise, it requests additional memory from the device via `cudaMalloc`. Instead of releasing memory with `cudaFree`, PCA recycles it back to the pools. Over time, memory usage stabilizes as no additional requests to `cudaMalloc` are needed. Users can bypass PCA by setting the `PYTORCH_NO_CUDA_MEMORY_CACHING` environment variable, where tensors are directly allocated and released memory via `cudaMalloc` and `cudaFree`.

While PyTorch focuses on memory pooling through dynamic memory allocation, TensorFlow [1] utilizes a slightly different approach. TensorFlow employs a similar pooling mechanism but pre-allocates all available GPU memory at the start of execution, managing it internally. Users can configure the initial GPU memory allocation size using the `per_process_gpu_memory_fraction` variable. This reduces allocation overhead but may result in inefficient memory utilization.

3 Methodology

3.1 Evaluation Setup

We collected the characteristics of DL workloads (detailed in Section 3.2) on a CPU-GPU discrete system described in Table 1. For each DL workload, we conducted 10 iterations of training. The oversubscription factor was varied from 1.25 to 10, by assuming a large disaggregated memory platforms. Note that the disaggregated memory devices are accessed through PCIe, which could be transparently managed by UVM as if those are a part of CPU system memory. To enable UVM support in the PyTorch framework, we replaced calls to `cudaMalloc` with `cudaMallocManaged`. Since PyTorch does not natively support UVM, the initialization of tensors with the attribute `device="cuda"` are frequently handled via `cudaMemcpyAsync` by the framework. We retained these `cudaMemcpyAsync` operations. Although UVM is conventionally designed to eliminate explicit memory copy operations,

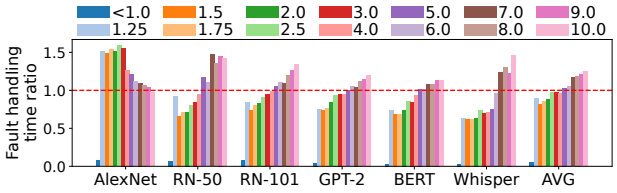


Figure 2. Ratio of page fault handling time with PCA to that without PCA.

we retained these memory copy APIs to reduce the performance overhead due to cold page misses. We applied UVM to the PyTorch framework using two approaches: pool-wise (with PCA) and tensor-wise (without PCA). One of the two options was selected for each experiment by switching the environment variable `PYTORCH_NO_CUDA_MEMORY_CACHING`. To adjust the memory oversubscription factor, we implemented a helper function that allocates a specified amount of device memory, effectively limiting the available memory size. This approach is a common method for controlling the UVM oversubscription factor, similar to techniques used in many other works [15–17]. Performance statistics were averaged over three measurements to mitigate the effects of system variance. UVM-related statistics were collected using the NVIDIA Nsight Systems profiling suite [8]. We adjust the batch size to regulate memory usage, maintaining a substantial memory footprint while ensuring acceptable execution time.

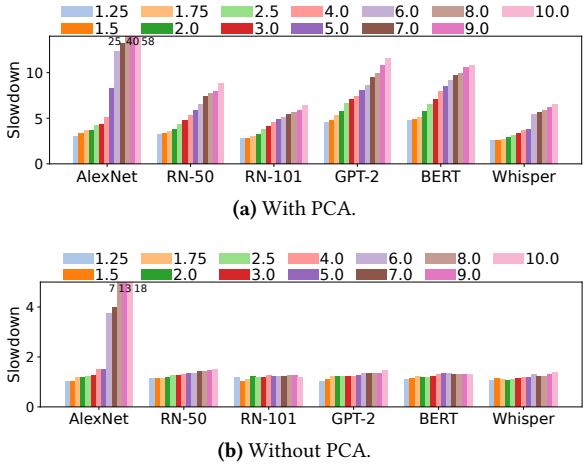
3.2 Workloads

We tested six widely used DL models, comprising AlexNet [23], ResNet50 [18] (denoted as RN-50), ResNet101 [18] (denoted as RN-101), GPT-2 [35], BERT [11], and Whisper [34], detailed in Table 2. The first three models—AlexNet, ResNet50, and ResNet101—are based on CNN architectures, while the latter three models—GPT-2, BERT, and Whisper—utilize Transformer architectures. These models were chosen to represent a diverse range of workloads, from image classification and text generation to context understanding and automatic speech recognition, highlighting the varying computational and architectural demands across domains.

4 Observations and Analysis

4.1 Overall DL performance with UVM

With the help of oversubscription feature, UVM is the only developer-agnostic solution for NVIDIA GPUs that enables deploying large workloads on the GPUs with limited memory capacity, even with its overheads. As a backend driver for emerging memory management interfaces (e.g., heterogeneous memory management (HMM) [28] and zero-copy transfer in grace-hopper architectures [6]), UVM is getting more widely leveraged to enable flexible and seamless data



(a) With PCA.

(b) Without PCA.

Figure 3. Performance under diverse oversubscription factors.

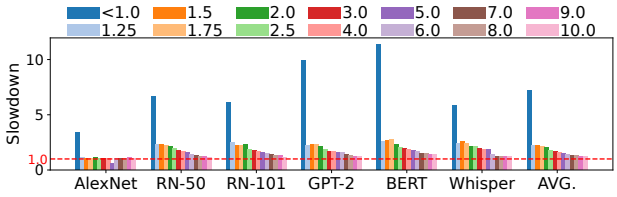


Figure 4. Performance under diverse oversubscription factors without PCA, compared to the baseline with PCA. (Values > 1.0 indicate that enabling PCA outperforms disabling PCA.)

transfer between CPU and GPU. Figure 1 shows the performance of a transformer model, GPT-2 [35], with and without UVM, under various batch sizes on the NVIDIA A100. When the batch size exceeds 80, the original PyTorch framework without UVM support fails to run GPT-2, resulting in an Out-of-Memory (OOM) error, which means that we should use multiple GPUs to run the model. With UVM enabled, the A100 is capable of training GPT-2 with larger batch sizes, even when the memory usage surpasses the 80 GB device memory capacity of the A100. Given the ever-increasing DL model size and the rising concerns about the affordability of high-end GPUs, UVM will be a more practical solution for DL studies.

Interestingly, when the memory is not oversubscribed, the UVM does not incur significant performance overhead compared to when UVM is not used, unlike conventional wisdom. This is thanks to PyTorch's PCA support, which significantly reduces page fault handling overhead, as shown in Figure 2. We will analyze the performance benefits and penalties newly introduced by PCA in the following sections in detail.

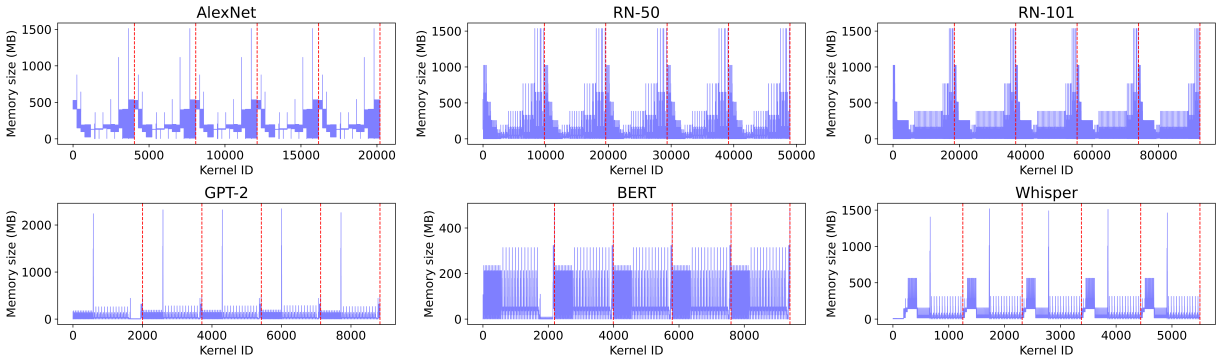
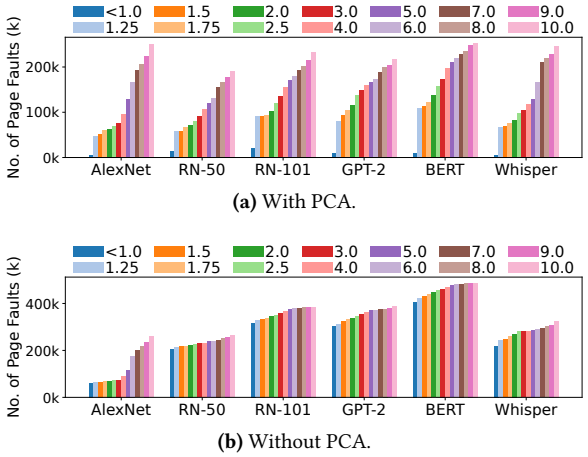


Figure 5. Memory usage per kernel over time for various models: x-axis represents kernel invocation IDs and y-axis shows memory consumption. Red dotted lines indicate iteration boundaries and only the first five iterations of each model are shown.

4.2 Performance under Various Oversubscription Factors

Figure 3 shows the performance slowdown of the DL workloads with and without PCA under diverse oversubscription factors ranging from 1.25 to 10. With PCA, the average performance slowdown is 7.29× (up to 57.74×), while without PCA, it is 1.75 × on average (up to 17.85×). When PCA is enabled, UVM is allocated to the memory pools, and tensor creation requests memory from these pools. Over time, the memory pools converge to a steady state where no additional UVM allocations are required in subsequent iterations. In contrast, without PCA, each tensor directly calls UVM APIs (e.g., `cudaMallocManaged`) to request memory and `cudaFree` to release it. These operations are repeated in every iteration. The repeated use of expensive memory operations, such as `cudaFree`, can become a bottleneck for the workload, thereby reducing the relative impact of UVM's performance penalty on overall performance. Notably, AlexNet experiences a significant performance drop when the oversubscription factor exceeds 5.0. This is because AlexNet's simple model architecture leaves little computation to hide UVM's memory access overhead, exacerbating performance losses. Similarly, Whisper suffers significant performance degradation when the oversubscription factor exceeds 6.0 due to extensive memory accesses that intensify page thrashing as device memory becomes increasingly constrained.

Although the performance impact without PCA is less severe compared to with PCA under memory oversubscription, enabling PCA still results in better overall performance. Figure 4 compares the performance of enabling PCA against disabling PCA under varying oversubscription factors. It demonstrates that enabling PCA consistently outperforms disabling PCA on average, although the performance gap decreases under higher oversubscription factors. Notably, in cases without oversubscription, disabling PCA is 7.22 × slower than enabling PCA on average. These results highlight that PCA remains an effective memory management strategy compared to directly using expensive memory APIs, such as `cudaMallocManaged` and `cudaFree`.



(a) With PCA.

(b) Without PCA.

Figure 6. Number of page faults.

> *Observation 1.* *LLMs, such as GPT-2 and BERT, benefit more from UVM than simpler CNNs under a high oversubscription factor, due to intensive computation overlapping with page fault handling.*

4.3 Kernel-wise Memory Requirements

While the performance declines sharply with higher oversubscription factors, it is worth noting that the model can still execute successfully even under extreme oversubscription conditions (e.g., an oversubscription factor of 10.0). This implies that a GPU is capable of running models that are significantly larger than its physical memory capacity, effectively enabling computations that would otherwise seem infeasible given the hardware constraints. This finding challenges the conventional wisdom of UVM, which generally recommends keeping the oversubscription factor below 1.25 [17].

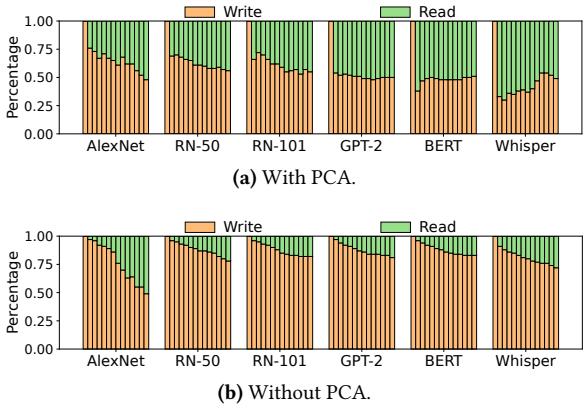
The ability to sustain execution at such high oversubscription factors can be attributed to the unique characteristics of deep learning (DL) model execution. Unlike traditional workloads, DL models comprise thousands of small computational kernels, each responsible for specific operations during training and inference. Although the overall memory footprint of these models can be exceedingly large, the actual working set required by the vast majority of these kernels remains relatively small at any given time. Figure 5 shows the memory usage per kernel over time for various models. It is evident that most kernels consume a very small amount of memory, and only a small fraction of kernels exhibit a significantly larger memory consumption. Further, the amount of memory consumed by the largest kernel is much smaller than the overall memory footprint of the entire model, as shown in Table 2. This observation suggests a promising opportunity for mitigating GPU memory limitations when deploying large-scale DL models with UVM. By leveraging this inherent sparsity in per-kernel memory demands, it becomes feasible to push beyond conventional oversubscription limits and efficiently execute large models on memory-constrained hardware.

> *Observation 2. Despite recommendations to limit the oversubscription factor to 1.25, our findings show acceptable overhead even at higher values for DL workloads.*

4.4 Statistics of Page Faults

4.4.1 Number of Page Faults. To reveal the underlying performance characteristics and the internal behaviors of DL workloads on UVM systems, we collected and analyzed more detailed statistics of UVM behavior. Figure 6 presents the number of page faults with and without PCA. Figure 6a shows a sharply increasing number of page faults under the higher oversubscription ratio, which indicates that the number of page faults is highly correlated with performance, and the overhead associated with page fault handling is the primary bottleneck for performance with PCA. In contrast, Figure 6b demonstrates a relatively flat trend without PCA. This flat trend can be attributed to the intensive memory operations, which dilute the intensity of memory access under high oversubscription factors.

Though the page faults are less steeply increasing, when PCA is not used, there are significantly more page faults than when using PCA. This is because, without PCA, memory is automatically reclaimed by `cudaFree` when a tensor reaches the end of its lifecycle. In contrast, with PCA enabled, the memory associated with a tensor remains resident on the device even after the tensor's lifecycle ends. PCA does not release memory but instead recycles it back into the memory pools, allowing subsequent output tensors to use it without incurring write page faults. This finding indicates memory



(a) With PCA.

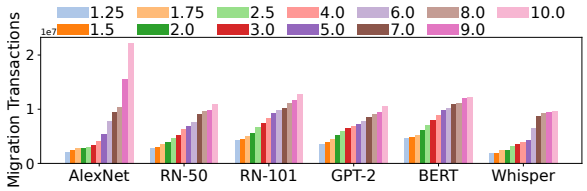
(b) Without PCA.

Figure 7. Breakdown of page faults caused by read/write transactions across different oversubscription factors.

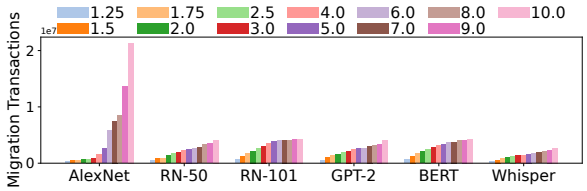
pooling mechanism effectively reduces page faults, as the mapped memory remains accessible for subsequent kernel executions. This reasoning is strongly supported by the detailed breakdowns of page faults and memory migration statistics, which will be described shortly.

4.4.2 Breakdown of Page Faults. Figure 7 presents the breakdown of page faults caused by read and write transactions. When there is no memory oversubscription, all page faults are caused by write memory transactions because we retained `cudaMemcpyAsync` in the PyTorch to reduce cold page misses, which proactively prefetches all input data into device memory, eliminating read page faults. However, with UVM, as the memory space used for output data is mapped on the GPU memory only when the corresponding pages are accessed by the GPU, for tensors that store results and do not require initialization, write transactions on these tensors trigger page faults and on-demand memory allocation. As the oversubscription factor increases, the proportion of page faults caused by read transactions gradually rises. This is because higher oversubscription factors lead to increasingly constrained device memory, resulting in tensors being evicted before all their read transactions are completed. Consequently, subsequent read transactions on evicted tensors introduce read page faults.

Interestingly, with PCA enabled, page faults are more evenly distributed between read and write transactions compared to cases without PCA. The reason is two fold. One reason is the efficient memory reuse. With PCA, multiple tensors can reuse a memory space that is already mapped on the device memory without releasing and reallocating the space, which helps reduce write page faults. Another reason is the delayed eviction. Because tensor memory space is not released even after their lifetime ends, new tensors that do not reuse the existing tensor's memory space will cause page eviction, which causes more read faults.

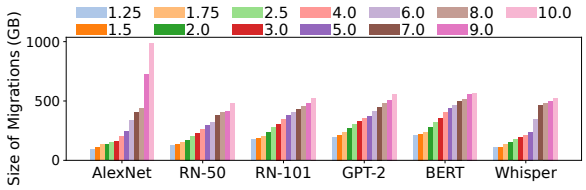


(a) With PCA.

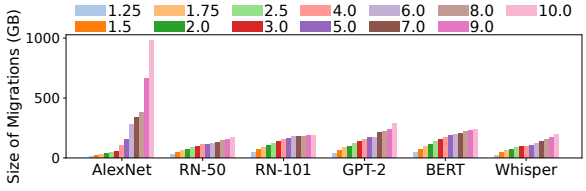


(b) Without PCA.

Figure 8. The count of migration transactions across different oversubscription factors.



(a) With PCA.



(b) Without PCA.

Figure 9. The size of migrations across different oversubscription factors.

Observation 3. PCA's pool-based memory management effectively reduces substantial page faults. Given that the main performance bottleneck of UVM is expensive page faults, pool-based memory management can be a solution.

4.5 Statistics of Page Migration Transactions

4.5.1 Number of Page Migration Transactions. Figure 8 plots the number of migration transactions in the unit of 10^7 transactions. Unlike the earlier statistics, PCA incurs almost 2× to 3× more migration transactions than when PCA is not used. This means that PCA enables more frequent memory transfer while reducing the page fault handling overheads. Note that Figure 6 shows that PCA causes only half the number of page faults than without using PCA. This is because the PCA reuses the memory space in the pools used by terminated tensors. These memory spaces are likely to be already mapped on the device memory. Therefore, only data contents are copied (migrated) without page faults. Also, as tensors do not release memory even after their lifetime ends, PCA causes more evictions, which also contribute to higher migration transactions.

As discussed in Section 4.4.1, delayed memory reclamation in PCA reduces page faults by allowing mapped memory reuse across subsequent kernels. Therefore, under high oversubscription, more memory migrations are triggered because of the limited device memory capacity. Despite more migrations, UVM's intrinsic prefetching and pre-eviction mechanisms [16] remove migration overhead from the critical path. In other words, PCA trades page fault overhead—always on the critical path—for migration overhead, thereby mitigating UVM overhead in DL workloads.

4.5.2 Size of Page Migrations. We also measured the amount of data traffic with and without PCA, as shown in Figure 9. This result aligns well with the migration transaction statistics shown in Figure 8. By incorporating PCA, we observe an increase in active data transfers while simultaneously reducing the number of page faults. Such highly correlated statistics between the number of transactions and the amount of data transfer imply that per-transaction traffic is similar in both cases. This is because of the UVM's uniform memory management mechanism; UVM manages memories in the unit of 2 MB blocks and migrates or prefetches data in the unit of 64 KB. The results also reveal that the migration size between the CPU and GPU is remarkably large, highlighting the extent of data movement required for execution under memory constraints. For instance, under an extreme oversubscription factor of 10.0, AlexNet exhibits a migration volume that is approximately 200× larger than its total memory footprint. This suggests that while UVM's on-demand migration mechanism simplifies CUDA programming, it also introduces substantial PCIe traffic overhead. These findings emphasize the pressing need for optimized UVM management strategies to accommodate the growing memory demands of large-scale GPU workloads, particularly for deep learning models.

Observation 4. PCA trades page fault overhead for memory migration overhead. As UVM's smart prefetching and pre-eviction mechanisms effectively remove memory migrations from the critical path, the cumbersome page fault overhead of UVM can be tackled by integrating the UVM with PCA.

5 Discussions

Our observations challenge several conventional wisdoms by demonstrating that PCA effectively mitigates the performance penalty of UVM, even in cases where GPUs are severely oversubscribed. This finding underscores the potential of PCA in enhancing UVM's usability for large-scale workloads, particularly in deep learning applications, where memory limitations are a persistent challenge. The effectiveness of UVM is further evident in deep learning computing due to its unique execution model. Deep learning workloads rely heavily on small computational kernels that execute in rapid succession, with extensive overlap between computations and memory transfers, thereby improving UVM's efficiency. This behavior is crucial for sustaining performance, even under extreme memory constraints. Additionally, UVM's ability to seamlessly manage memory transfers between host and device without explicit user intervention makes it highly attractive for deep learning researchers and practitioners.

However, a key limitation of UVM is its application context-agnostic nature, which may not always yield optimal performance. Since UVM is a general-purpose memory management system, it lacks specialized adaptation for deep learning workloads. This limitation suggests the need for further optimization to enhance its efficiency. Therefore, we encourage the research community to further explore the integration of PCA with UVM. A deeper investigation into this combination could enable more scalable deep learning computing without imposing significant programming overhead. Future work could explore adaptive memory management strategies that dynamically adjust UVM's behavior to specific deep learning workloads, paving the way for improved GPU memory efficiency in large-scale models.

6 Conclusion

In this paper, we characterize the performance of UVM for deep learning (DL) computing under severe memory oversubscriptions and investigate the feasibility of executing large DL models that would otherwise fail due to memory limitations. We also conduct a thorough analysis of the performance impact and behavioral characteristics of UVM when applied to DL workloads. Our observations indicate that while pool-based memory management effectively reduces page faults, it simultaneously increases data traffic, which can introduce additional performance bottlenecks. These findings emphasize the opportunities and challenges of using UVM for DL workloads, especially as models grow in size and complexity. They highlight the need for advanced optimization techniques and adaptive memory management to enhance efficiency and scalability for large-scale deep learning on GPUs.

Acknowledgement

This work was supported by NSF CCF-2341039.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: a system for large-scale machine learning. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (Savannah, GA, USA) (OSDI'16). USENIX Association, USA, 265–283.

[2] Tyler Allen and Rong Ge. 2021. Demystifying GPU UVM Cost with Deep Runtime and Workload Analysis. In 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). 141–150. doi:10.1109/IPDPS49936.2021.00023

[3] PyTorch Open-Source Community. Accessed February 2025. PyTorch CUDA Caching Memory Allocator. https://github.com/pytorch/pytorch/blob/main/c10/cuda/CUDACachingAllocator.cpp.

[4] NVIDIA Corporation. Accessed February 2025. Improving GPU Memory Oversubscription Performance. https://developer.nvidia.com/blog/improving-gpu-memory-oversubscription-performance/.

[5] NVIDIA Corporation. Accessed February 2025. NVIDIA A100 80GB PCIe GPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/PB-10577-001_v02.pdf.

[6] NVIDIA Corporation. Accessed February 2025. NVIDIA Grace Hopper Superchip Architecture Whitepaper. https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper.

[7] NVIDIA Corporation. Accessed February 2025. NVIDIA Hopper Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/.

[8] NVIDIA Corporation. Accessed February 2025. NVIDIA Nsight Systems: A system-wide performance analysis tool. https://developer.nvidia.com/nsight-systems.

[9] NVIDIA Corporation. Accessed February 2025. NVIDIA Tesla P100 Whitepaper. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[10] NVIDIA Corporation. Accessed February 2025. Unified Memory in CUDA 6. https://developer.nvidia.com/blog/unified-memory-in-cuda-6/.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018).

[12] Xianzhong Ding, Yunkai Zhang, Binbin Chen, Donghao Ying, Tieying Zhang, Jianjun Chen, Lei Zhang, Alberto Cerpa, and Wan Du. 2023. Vmr2l: Virtual machines rescheduling using reinforcement learning in data centers.

[13] Xianzhong Ding, Yunkai Zhang, Binbin Chen, Donghao Ying, Tieying Zhang, Jianjun Chen, Lei Zhang, Alberto Cerpa, and Wan Du. 2025. Towards VM Rescheduling Optimization Through Deep Reinforcement Learning. In Proceedings of the Twentieth European Conference on Computer Systems.

[14] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. arXiv preprint arXiv:2407.21783 (2024).

[15] Debashis Ganguly, Rami Melhem, and Jun Yang. 2021. An Adaptive Framework for Oversubscription Management in CPU-GPU Unified Memory. In 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE). 1212–1217. doi:10.23919/DATE51398.2021.9473982

[16] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay between hardware prefetcher and page eviction policy in CPU-GPU unified virtual memory. In Proceedings of the 46th International Symposium on Computer Architecture (Phoenix, Arizona) (ISCA '19). Association for Computing Machinery, New York, NY, USA, 224–235. doi:10.1145/3307650.3322224

[17] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2020. Adaptive Page Migration for Irregular Data-intensive Applications under GPU Memory Oversubscription. In 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). 451–461. doi:10.1109/IPDPS47924.2020.00054

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 770–778. doi:10.1109/CVPR.2016.90

[19] 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

[20] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24). USENIX Association, Santa Clara, CA, 745–760. https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng

[21] 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

[22] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. 2016. Bit-Plane Compression: Transforming Data for Better Compression in Many-Core Architectures. In 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). 329–340. doi:10.1109/ISCA.2016.37

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

[24] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24). USENIX Association, Santa Clara, CA, 155–172. https://www.usenix.org/conference/osdi24/presentation/lee

[25] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch distributed: experiences on accelerating data parallel training. Proc. VLDB Endow. 13, 12 (Aug. 2020), 3005–3018. doi:10.14778/3415478.3415530

[26] Yiwei Li and Mingyu Gao. 2023. Baryon: Efficient Hybrid Memory Management with Compression and Sub-Blocking. In 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA). 137–151. doi:10.1109/HPCA56546.2023.10071115

[27] Mao Lin, Keren Zhou, and Pengfei Su. 2023. DrGPUM: Guiding Memory Optimization for GPU-Accelerated Applications. 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, 164–178. doi:10.1145/3582016.3582044

[28] The linux kernel development community. Accessed February 2025. Heterogeneous Memory Management (HMM). https://www.kernel.org/doc/html/latest/mm/hmm.html.

[29] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. 2017. Training deeper models by GPU memory optimization on TensorFlow. In Proc. of ML Systems Workshop in NIPS, Vol. 7.

[30] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (St. Louis, Missouri) (SC '21). Association for Computing Machinery, New York, NY, USA, Article 58, 15 pages. doi:10.1145/3458817.3476209

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: an imperative style, high-performance deep learning library. Curran Associates Inc., Red Hook, NY, USA.

[32] Shishir G. Patil, Paras Jain, Prabal Dutta, Ion Stoica, and Joseph Gonzalez. 2022. POET: Training Neural Networks on Tiny Devices with Integrated Rematerialization and Paging. In Proceedings of the 39th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 162), Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (Eds.). PMLR, 17573–17583. https://proceedings.mlr.press/v162/patil22b.html

[33] 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

[34] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. 2023. Robust speech recognition via large-scale weak supervision. In Proceedings of the 40th International Conference on Machine Learning (Honolulu, Hawaii, USA) (ICML'23). JMLR.org, Article 1182, 27 pages.

[35] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. OpenAI blog 1, 8 (2019), 9.

[36] Jie Ren, Dong Xu, Shuangyan Yang, Jiacheng Zhao, Zhicheng Li, Christian Navasca, Chenxi Wang, Harry Xu, and Dong Li. 2024. Enabling Large Dynamic Neural Network Training with Learning-based Memory Management. In 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA). 788–802. doi:10.1109/HPCA57654.2024.00066

[37] Sam Stoelinga. Accessed February 2025. What GPUs can run Llama 3.1 405B? https://www.substratus.ai/blog/llama-3-1-405b-gpu-requirements.

[38] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288 (2023).

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In Proceedings of the 31st International Conference on Neural Information Processing Systems (Long Beach, California, USA) (NIPS'17). Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[40] 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. In Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Vienna, Austria) (PPoPP '18). Association for Computing Machinery, New York, NY, USA, 41–53. doi:10.1145/3178487.3178491

[41] 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

[42] Vinson Young, Sanjay Kariyappa, and Moinuddin K. Qureshi. 2019. Enabling Transparent Memory-Compression for Commodity Memory Systems. In 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). 570–581. doi:10.1109/HPCA.2019.00010

[43] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. Proc. VLDB Endow. 16, 12 (Aug. 2023), 3848–3860. doi:10.14778/3611540.3611569