# DeepUM: Tensor Migration and Prefetching in Unified Memory

Jaehoon Jung*
jaehoon.jung@moreh.io
Moreh Inc.
Seoul, South Korea

Jinpyo Kim
jinpyo@aces.snu.ac.kr
Dept. of Computer Science and Engineering
Seoul National University
Seoul, South Korea

Jaejin Lee
jaejin@snu.ac.kr
Dept. of Data Science
Dept. of Computer Science and Engineering
Seoul National University
Seoul, South Korea

## ABSTRACT

Deep neural networks (DNNs) are continuing to get wider and deeper. As a result, it requires a tremendous amount of GPU memory and computing power. In this paper, we propose a framework called DeepUM that exploits CUDA Unified Memory (UM) to allow GPU memory oversubscription for DNNs. While UM allows memory oversubscription using a page fault mechanism, page migration introduces enormous overhead. DeepUM uses a new correlation prefetching technique to hide the page migration overhead. It is fully automatic and transparent to users. We also propose two optimization techniques to minimize the GPU fault handling time. We evaluate the performance of DeepUM using nine large-scale DNNs from MLPerf, PyTorch examples, and Hugging Face and compare its performance with six state-of-the-art GPU memory swapping approaches. The evaluation result indicates that DeepUM is very effective for GPU memory oversubscription and can handle larger models that other approaches fail to handle.

## CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; **Parallel programming languages**; • **Software and its engineering** → **Runtime environments**.

## KEYWORDS

deep learning, neural networks, CUDA, unified memory, data prefetching, runtime system, device driver

**ACM Reference Format:**
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 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3575693.3575736

*This work has been done when Jaehoon Jung was a Ph.D. student at Seoul National University.


*ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada*

ACM ISBN 978-1-4503-9916-6/23/03…$15.00
https://doi.org/10.1145/3575693.3575736

## 1 INTRODUCTION

Recent Deep Neural Network (DNN) models require tremendous GPU memory and computation power because deeper and wider layers generally provide better performance[7, 12, 19]. Moreover, a larger batch size can lead to shorter training time[14, 57]. As a result, hundreds to thousands of GPUs with a large amount of memory are used for training these models. Since general users are hard to access such an amount of expensive high-end GPUs, most of the research and training of large-scale DNNs are led by big tech companies.

Fortunately, training a large DNN model from scratch with a large-scale GPU system (pre-training) and then fine-tuning the model with a relatively small system for a specific task provides good performance. Thus, it is a common practice to get a pre-trained model and fine-tune it with a small system.

However, the problem is that the current state-of-the-art DNN models are so big that even fine-tuning the models is hard to be performed with a small system, especially with a single GPU system. For example, one of the state-of-the-art language models, GPT-3[7], has 175 billion parameters. One hundred seventy-five billion parameters require approximately 326 GB of memory space to load the model when saved in FP16. Such an amount of memory space cannot even be handled using a single NVIDIA H100 80GB GPU[41] that is the latest data center GPU from NVIDIA.

To solve such a memory capacity problem, many studies have been performed, such as data compression [6, 10, 18, 26, 34], mixed-precision arithmetic[11, 17, 28], data recomputation[8, 16, 55], and memory swapping[5, 6, 21, 24, 33, 45, 49–51, 55].

Among others, we focus on GPU memory swapping to solve the memory capacity problem of DNNs. Previous approaches in memory swapping are divided into two categories. One uses CUDA Unified Memory[38] with page prefetching[5, 35]. The other uses pure (i.e., non-UM) GPU memory with swapping-in/swapping-out memory objects[6, 21, 24, 33, 45, 49–51, 55].

Unified Memory (UM) provides a single address space shared between the CPUs and the GPUs. It exploits the GPU page fault mechanism to migrate pages between processors on-demand. The name could vary depending on the programming model (e.g., Shared Virtual Memory[31] in OpenCL[15] and Intel oneAPI[25]), but they all have very similar semantics.

Few GPU memory swapping studies are based on UM because of the significant overhead due to address translation and page fault handling[4]. Since UM utilizes virtual memory that requires address translation for every memory request in the GPU, it may impact

performance significantly. Also, handling page faults requires expensive I/O operations between the CPUs and GPUs for migrating pages between them.

Despite the disadvantages mentioned above, using UM can be superior to using pure GPU memory in some cases. First, pure (i.e., non-UM) GPU memory may fail to run a CUDA kernel when the total amount of the required memory by the kernel is larger than the GPU memory capacity. With UM, the CUDA kernel can execute even if the required pages are not in the GPU memory because they will be migrated on-demand by the page-fault mechanism. As a result, UM can handle large-scale DNNs that cannot be handled by using pure GPU memory. Second, using pure GPU memory may suffer from memory fragmentation. GPU memory allocation/deallocation occurs very frequently when training DNNs. Even though popular Deep Learning frameworks, such as TensorFlow[1] or PyTorch[44] manage their own GPU memory pool to minimize memory allocation/deallocation time and reduce memory fragmentation, they still have some memory fragmentation issues. Since UM is a kind of virtual memory, all memory objects are managed in the unit of a 4KB page. Thus, it suffers less from memory fragmentation and is more likely to run large DNN models without any problem.

In this paper, we propose a framework called DeepUM that exploits UM to allow oversubscribing GPU memory and implements optimization techniques to minimize the overhead caused by UM.

DeepUM modifies a correlation prefetching technique originally developed for cache-line prefetching to prefetch GPU pages. Among various prefetching techniques, we choose correlation prefetching[2, 3, 27, 53] because it is synergetic with UM. Since UM is based on the page fault mechanism, faulted accesses are monitored by the page fault handler. DeepUM can easily record the relationship between faulted pages using the faulted addresses obtained from the page fault handler. DeepUM exploits the fact that the kernel execution patterns and their memory access patterns are mostly fixed and repeated in the DNN training workload. Thus, memorizing the repeated patterns and exploiting the information through correlation prefetching is desirable. DeepUM's correlation tables record the history of the kernel executions and their page accesses during the training phase of a DNN. It prefetches pages based on the information in the correlation tables by predicting which kernel will execute next. While traditional correlation prefetching uses a single table to store history and records the relationship between the CPU cache lines, DeepUM correlation prefetching uses two different table structures.

To minimize the fault handling time, DeepUM proposes two optimization techniques for the GPU fault handling routines that are a part of the NVIDIA device driver. One is page pre-eviction based on the information from correlation tables. When GPU memory is oversubscribed, page eviction increases page fault handling time because page eviction logic lies on the critical path of the page fault handling routines[32]. The other optimization is page invalidation in the GPU memory when the page eviction victim is expected to be no longer used by PyTorch. This optimization removes unnecessary memory traffic between the CPUs and the GPUs.

DeepUM supports PyTorch, one of the most popular Deep Learning frameworks. Compared with the previous approaches, DeepUM requires very few code modifications in the original PyTorch source code (less than ten lines of the code) to change the behavior of the PyTorch memory allocator. Moreover, it requires no user code modification. The user code describes the DNN model and how it is trained.

The major contributions of this paper are summarized as follows:

- We propose DeepUM that exploits CUDA UM to allow GPU memory oversubscription for DNNs. It automatically prefetches data using correlation prefetching. It is fully automatic and transparent to users.
- We propose a correlation prefetching technique specialized for prefetching pages in DNNs. The two correlation tables record the history of the kernel executions and the page access patterns during the training phase of DNNs.
- We propose two optimization techniques for minimizing the GPU fault handling time. One is a new page pre-eviction policy coupled with correlation prefetching, and the other is invalidating the useless pages for PyTorch in the GPU memory when they are selected as victim pages.
- We evaluate DeepUM using nine large-scale DNNs from MLPerf[36], PyTorch examples, and Hugging Face[56]. We compare DeepUM with six previous GPU memory swapping approaches. The evaluation result shows that DeepUM achieves comparable performance to a previous GPU memory swapping approach that requires manual optimization. DeepUM achieves better performance than the remaining five swapping approaches. Moreover, DeepUM can run the models with a much larger batch size which previous approaches fail to run.

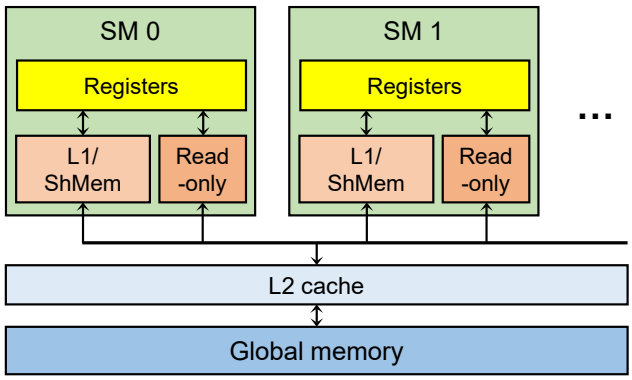



**Figure 1: Memory hierarchy of an NVIDIA GPU.**

## 2 BACKGROUND

This section briefly describes the GPU memory hierarchy and CUDA programming model. In addition, we introduce CUDA unified memory (UM) and the NVIDIA page fault handler used by CUDA UM.

### 2.1 GPUs and CUDA Programming Model

General-purpose computing on graphics processing units (GPGPU) allows programmers to process a tremendous amount of computation in a shorter time than the CPU. It is because hundreds to thousands of GPU threads perform computations simultaneously. To efficiently manage the GPU and offload computations to the

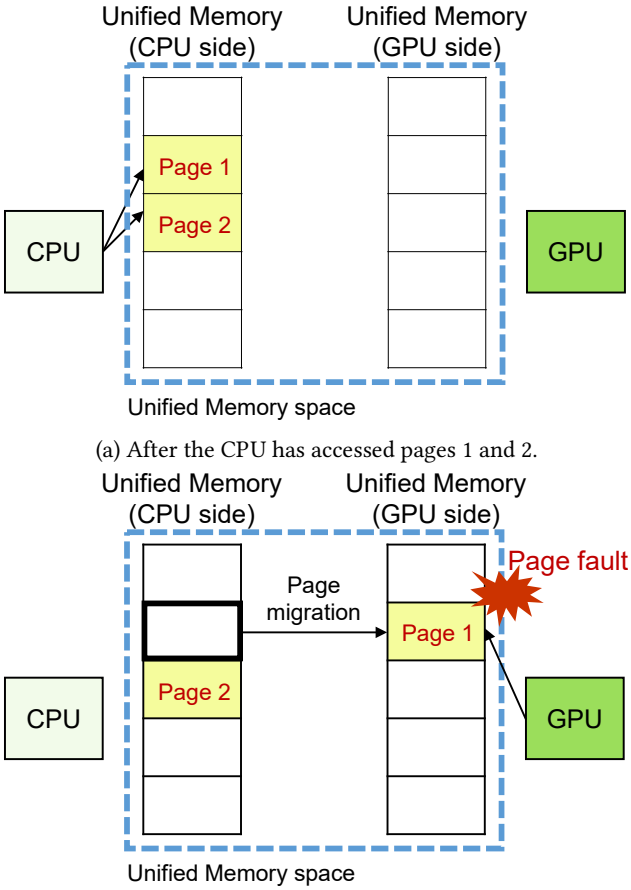
GPU, many programming models have been proposed: CUDA[40], oneAPI[25], openACC[43], and OpenCL[15]

In this paper, we focus on CUDA, a GPU parallel programming model developed by NVIDIA. A programmer must define a function-like CUDA kernel to offload computation to the GPU. Then, the CUDA kernel is executed on the GPU, $N$ times in parallel by $N$ different CUDA threads. Each CUDA thread has a unique thread ID that is used to decide the control flow and compute memory addresses to be accessed. CUDA threads are grouped into a thread block, and thread blocks are grouped into a grid. The programmer specifies the configuration of a grid when they launch a CUDA kernel.

Figure 1 shows a memory hierarchy of the latest microarchitecture of the NVIDIA GPU. A GPU consists of dozens of streaming multiprocessors (SMs). Each thread block in a grid is mapped to an SM in the GPU. An SM contains a few hundred CUDA cores. A thread in a thread block is mapped to a CUDA core, and an SM can handle several thread blocks concurrently.

Each SM has different types of memory units, such as registers, L1 cache, shared memory (scratchpad memory), and constant memory (read-only memory). While registers are private to each thread, other memory units are shared within an SM. Outside of an SM, there is an L2 cache shared across all SMs. The global memory is an off-chip DRAM, generally a few gigabytes to tens of gigabytes. As usual, a memory unit closer to CUDA cores (SMs) has a lower latency and smaller capacity. Even though the GPU offers tens of gigabytes of global memory, the DNN workload severely lacks the amount of global memory.

Since a GPU thread cannot directly access the main memory unless a programmer maps the main memory space to the GPU memory space, a programmer must manually move the data between the GPU global memory and the CPU main memory. In addition, access to the main memory mapped to the GPU memory space suffers performance because it incurs PCIe transfer on every memory access. To this end, NVIDIA proposes Unified Memory (UM).




(a) After the CPU has accessed pages 1 and 2.

(b) After the GPU has accessed page 1.

**Figure 2: Snapshot of Unified Memory when the CPU and GPU access pages.**

## 2.2 CUDA Unified Memory

CUDA Unified Memory (UM)[5, 29, 30, 38] is a component of the CUDA programming model. It provides a single memory address space that can be accessed by both the CPUs and GPUs in the same system. Beginning with the Pascal architecture[42], NVIDIA GPUs have a page migration engine. It allows GPUs to have a virtual memory system by exploiting the page fault mechanism. When a page accessed by a GPU does not reside in the GPU memory, the GPU raises a page fault interrupt signal, and the NVIDIA device driver migrates the faulted page from anywhere else in the system to the GPU. UM enables the GPU memory oversubscription without any intervention of the programmer. In other words, a CUDA program that requires a memory space larger than the GPU memory capacity can run seamlessly. Thus, it dramatically reduces the programmer's burden.

Figure 2 shows snapshots of UM when the CPU and GPU accessed pages. In Figure 2(a), a UM space is allocated by a programmer. Then, the host UM space exists in the main memory and the GPU UM space in the GPU memory. Suppose that the CPU has accessed pages 1 and 2. Then they reside in the host UM space. In Figure 2(b), now the GPU accesses page 1. Since page 1 resides in the host UM space, the GPU raises a page fault interrupt signal, and the NVIDIA device driver migrates page 1 from the main memory to the GPU memory. When the migration finishes, the GPU replays the faulted page access.

Despite its strength, the downside of UM is that the GPU page fault handling is costly. A translation lookaside buffer (TLB) exists for each streaming multiprocessor (SM) in the GPU. When a page fault occurs in the GPU, the TLB for the corresponding SM is locked and cannot handle any new translation until all faults from the SM are resolved[52]. Moreover, page faults require expensive I/O operations between the CPUs and GPUs for migrating pages and page evictions. Thus, it is highly recommended to insert CUDA prefetch API functions (e.g., `cudaMemPrefetchAsync()`) or CUDA user-hint API functions (e.g., `cudaMemAdvise()`) to reduce page faults.

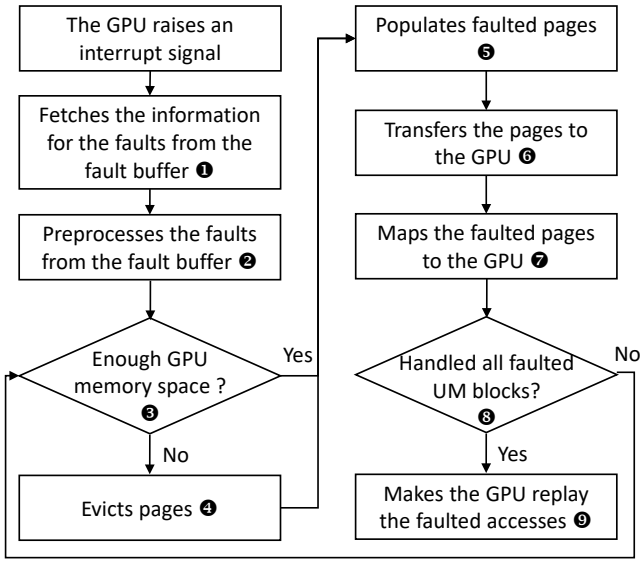



**Figure 3: The behavior of NVIDIA page fault handler.**

## 2.3 NVIDIA Page Fault Handler

When an NVIDIA GPU raises a page fault interrupt signal, the NVIDIA driver catches the interrupt signal and handles it. A fault buffer is a circular queue in the NVIDIA GPU. It stores faulted access information. The GPU can generate multiple faults concurrently, and there can be multiple fault entries for the same page in the fault buffer[52]. A UM block is a group of maximum 512 contiguous pages and a unit of management by the NVIDIA driver. The maximum size of a UM block is $4KB \times 512 = 2MB$, and all pages in the same UM block are processed together by the NVIDIA driver. Each UM block object contains the information of all pages in the UM block, such as which processor has the pages and whether the pages are mapped with read protection or write protection. If a UM block contains a faulted page, we call the UM block *the faulted UM block* hereafter.

Figure 3 shows the diagram of page fault handling. First, the NVIDIA driver fetches the page addresses and access types of the faulted accesses from the fault buffer in the GPU (❶). Then, the NVIDIA driver preprocesses the faults (❷). It removes the duplicate addresses and groups them according to their UM blocks. Next, the NVIDIA driver checks the available GPU memory space for each faulted UM block (❸). If no GPU memory space is available for the faulted UM block, it evicts some pages from the GPU to the CPU (❹). Then, it populates faulted pages in the GPU (❺) (i.e., it allocates GPU memory space to the faulted pages), and it transfers the pages to the GPU (❻). When the transfer is done, the faulted pages of the UM block are mapped to the GPU (❼). This process repeats until all faulted UM blocks are handled (❽). Finally, the NVIDIA driver sends a replay signal to the GPU, and the fault handler finishes (❾).

# 3 OVERALL STRUCTURE OF DEEPUM

In this section, we present the overall structure of DeepUM. DeepUM allows GPU memory oversubscription for DNNs by exploiting CUDA Unified Memory and using CPU memory as a backing store.

## 3.1 Structure of DeepUM

Figure 4 shows the overall structure of DeepUM. It consists of the DeepUM runtime and the DeepUM driver. The driver is a Linux kernel module. DeepUM targets PyTorch, one of the most popular deep learning frameworks, and PyTorch runs on top of the DeepUM runtime.

*DeepUM runtime.* The DeepUM runtime provides wrapper functions for CUDA memory allocation API functions to switch all GPU memory allocation requests to UM space allocation requests. It easily accomplishes GPU memory oversubscription by allocating all GPU memory objects in the UM space. Moreover, the DeepUM runtime provides wrapper functions for CUDA kernel launch commands and other CUDA library functions, such as those in cuDNN and cuBLAS. Note that CUDA library functions also launch CUDA kernels.

The DeepUM runtime manages a table called the *execution ID table*. The table holds kernel launch history and contains the hash value of each kernel's name and arguments. When a new kernel launch command comes to the DeepUM runtime, it computes the hash value of the kernel name and arguments. Then, it looks up the execution ID table to find the command of the same hash value. If it finds a matching command, it gives the same *execution ID* to the kernel. Otherwise, it assigns a new execution ID to the kernel and saves the information in the table. Finally, the DeepUM runtime enqueues a CUDA callback function to the CUDA runtime just before enqueueing the kernel launch command. The callback function passes the execution ID of the following kernel launch command to the DeepUM driver through the Linux `ioctl` command. The DeepUM driver uses the passed execution ID for correlation prefetching.

*DeepUM driver.* The DeepUM driver handles GPU page faults and prefetches pages to the GPUs. We observe that the kernel execution patterns and the memory access patterns within the kernels are fixed and repeated in the training phase of a DNN. Thus, memorizing the repeated patterns and exploiting the information for prefetching is desirable. Correlation tables managed by the DeepUM driver record the history of the kernel executions and their page accesses during the training phase of a DNN. The DeepUM driver prefetches pages based on the information in the correlation tables by predicting which kernel will execute next. In Section 4.2, we will describe the correlation prefetching mechanism used by the DeepUM driver.

There are four kernel threads in the DeepUM driver: *fault handling thread*, *correlator thread*, *prefetching thread*, and *migration thread*. The *fault handling thread* handles GPU page faults using the functions implemented in the NVIDIA driver, such as accessing the fault buffer and sending replay signals to the GPU. As mentioned in Section 2.3, the NVIDIA GPU has a hardware fault buffer that accumulates the information of faulted accesses. The DeepUM driver intercepts page fault interrupt signals to the NVIDIA driver, and

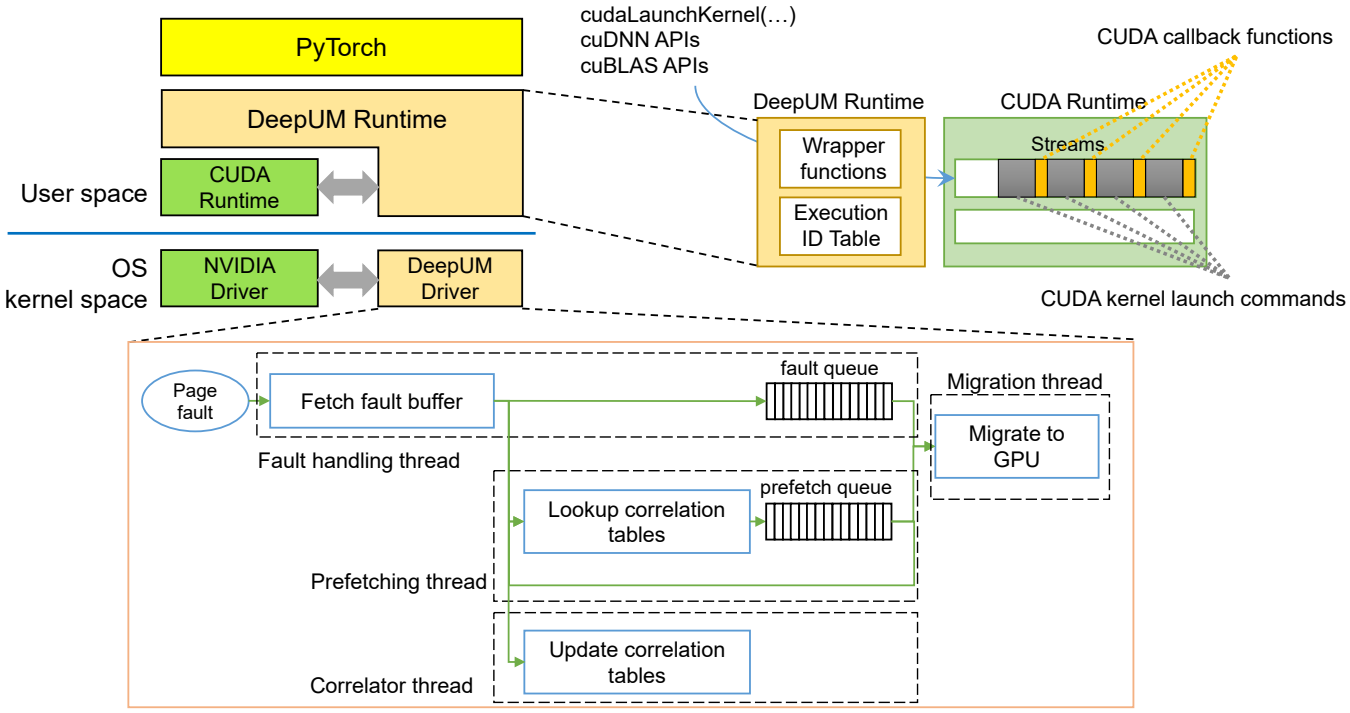



**Figure 4: Overall structure of DeepUM**

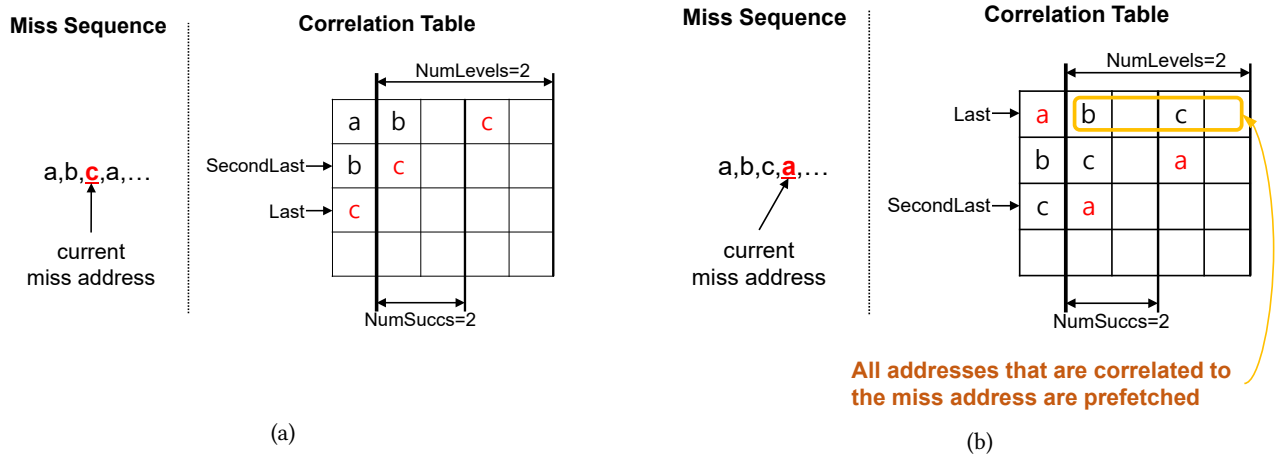



**Figure 5: Pair-based correlation prefetching.**

the fault handling thread reads the fault buffer. The fault handling thread passes the information of faulted accesses to the other three threads.

The *fault queue* is a single-producer/single-consumer queue that stores the UM block addresses of the faulted pages. It holds the highest priority items to be handled in the driver to make the GPU replay the faulted accesses as soon as possible.

The *correlator thread* manages correlation tables. It updates the correlation tables based on the fault information from the fault handling thread. We will discuss the structure of the correlation tables and how the correlator thread updates them in Section 4.2.

The *prefetching thread* looks up the correlation tables and calculates the UM block addresses for prefetching with the faulted block address. Then, it enqueues the prefetch commands to the *prefetch queue*, a single-producer/single-consumer queue. A prefetch command consists of a UM block address to prefetch and the execution ID for which the corresponding UM block is predicted to be used.

Finally, the *migration thread* migrates the UM blocks between the CPU and the GPU. The fault queue has a higher priority than the prefetch queue. It first handles commands from the fault queue managed by the fault handling thread. When the fault queue is

empty, it handles commands from the prefetch queue managed by the prefetching thread.

# 4 CORRELATION PREFETCHING FOR GPU PAGES

Correlation prefetching[2, 3, 9, 27, 53] was originally developed for cache-line prefetching. DeepUM modifies the original correlation prefetching to adapt it to prefetching pages for DNN workloads. There are two methods in the original correlation prefetching: *stride-based* and *pair-based*. The stride-based correlation prefetching[9] finds stride patterns in the sequence of missed addresses, while the pair-based correlation prefetching[2, 3, 27, 53] finds a correlation between missed addresses. DeepUM is based on the pair-based correlation prefetching technique.

## 4.1 Pair-Based Correlation Prefetching

The pair-based correlation prefetching records past sequences of missed addresses in a correlation table. When a cache miss occurs, it looks up the correlation table and prefetches all the addresses correlated with the missed address.

Figure 5 shows an example of pair-based correlation prefetching for cache lines. In Figure 5(a), we suppose that cache misses have occurred for addresses *a* and *b*. Following them, accessing address *c* causes a cache miss. A different missed address has a different row in the table. Each row of the correlation table has *N*-way associativity (Assoc) to reduce address conflict between UM blocks that map to the same row. In Figure 5, we assume associativity is one for easy understanding. For each set, there are NumLevels levels of successor miss addresses. In each level, NumSucc entries are MRU ordered from left to right. Last (points to *c*) and SecondLast (points to *b*) are pointers to the entries for the last and second last misses, respectively. The entry for *a* has *b* in the first level because the miss for *b* occurred right after *a*. It also has *c* in the second level because the miss for *c* indirectly follows the miss for *a* after the miss for *b*.

In Figure 5(b), we suppose that the cache miss for *a* occurrs again after the miss for *c*. The entries for *b*, *c*, and the Last and SecondLast pointers are updated. At this point, since there exist successor entries recorded for *a*, all the entries in the row of *a* (*b* and *c*) are prefetched from the memory in addition to accessing *a* in the memory.

## 4.2 Correlation Prefetching in DeepUM

Correlation prefetching in the DeepUM aims to reduce page faults by prefetching pages expected to be accessed by CUDA kernels. Unlike the original correlation prefetching, DeepUM uses two types of correlation tables: *execution ID* and *UM block*. Both types of tables have a single level ($NumLevels = 1$) because the prefetching thread does chaining. Moreover, there is no reason to maintain multiple levels because of the characteristics of the DNN workloads. Note that DeepUM's correlation prefetching works at the UM block level rather than cache-line or page level.

*Execution ID correlation table.* The execution ID correlation table (the execution table in short) records the execution history of the execution IDs of CUDA kernels, and only a single table exists. Execution IDs come from the DeepUM runtime. Figure 6 shows an example of an execution table. Each entry of the execution table holds sets of correlated execution IDs. Each set consists of four execution IDs. The first three IDs represent the previously executed kernels right before the last kernel (currently executing kernel when updating the table). Thus, the last ID represents the next kernel to execute when prefetching occurs. For example, the entry for the execution ID *0* has a record of (*7*, *9*, *92*, *75*). This record in the entry for the execution ID *0* implies that the kernels with execution IDs *7*, *9*, and *92* have been executed, and the kernel with execution ID *0* is currently executing. It predicts the execution ID of the kernel to be executed next as *75*.

| Execution ID | Record 0 | Record 1 | Record 2 | ... |
|---|---|---|---|---|
| 0 | (7, 9, 92, 75) | | | |
| 1 | (89, 53, 24, 10) | (34, 52, 22, 99) | (4, 3, 939, 2) | |
| 2 | (3, 53, 4, 8) | (33, 588, 34, 1) | | |
| ... | | | | |

**Figure 6: An execution ID correlation table.**

Correlation table for execution ID 0

| Block | Successor Block 0 | Successor Block 1 |
|---|---|---|
| a | b | p |
| b | e | q |
| c | d | |
| ... | | |

Start: a
End: q

Correlation table for execution ID 1

| Block | Successor Block 0 | Successor Block 1 |
|---|---|---|
| f | e | u |
| g | t | i |
| k | g | n |
| ... | | |

Start: k
End: u

**Figure 7: UM block correlation tables.**

The number of records each entry contains is variable, i.e., the number of the successor kernels of a kernel is variable. Thus, each entry can hold all history of successor kernels' execution IDs. DeepUM chooses this scheme to predict the next kernel to be executed as accurately as possible. Even though the incorrect result of predicting the next UM block to be accessed is not that costly, the inaccurate result of predicting the next kernel to be executed is expensive.

**UM block correlation table.** As mentioned in Section 2.3, the NVIDIA driver manages the pages in the CUDA UM space by grouping them into multiple UM blocks. A maximum of 512 contiguous pages are grouped into a UM block. Rather than recording the history of faulted page addresses, a UM block correlation table (a block table in short) records the history at the UM block level. There are two reasons for choosing this granularity. One is that there are too many page addresses to manage for large-scale DNNs. The other is that making DeepUM has the same page management granularity

as the NVIDIA driver is more efficient. Note that the NVIDIA driver manages the pages in the granularity of a UM block.

Figure 7 shows an example of block tables. A block table exists for each execution ID and records a history of UM block accesses within the corresponding CUDA kernel. It is similar to the table used in the original correlation prefetching. However, a block table contains the address of the *end* UM block that points to the last UM block prefetched, and the *start* UM block that points to the first faulted UM block in the corresponding kernel execution. These two pointers are used to implement chaining. Both the start UM block and end UM block are captured during the transition of the currently executing kernel. End UM block is the UM block where the page resides that lastly faulted right before execution ID transition. Start UM block is the UM block where the first faulted page resides that occurred right after the execution ID transition.

*Prefetching mechanisms and chaining.* When a page fault occurs, the DeepUM driver prefetches all pages in the UM blocks correlated to the faulted UM block by looking up the UM block correlation table of the currently executing kernel.

When the prefetching thread in DeepUM meets the UM block that is the same as the end block in the UM block correlation table, it ends prefetching for the kernel and predicts the kernel that will execute next by looking up the execution ID table. Then, it starts prefetching for the predicted kernel, beginning with the start UM block in the UM block correlation table of the predicted kernel.

Consider the block tables in Figure 7, suppose that DeepUM is prefetching the UM block $b$ for the kernel with execution ID *0*. Also suppose that the kernel with execution ID *1* will be executed right after the kernel with execution ID *0*. The successor UM blocks for $b$ are $e$ and $q$. Since block $q$ is the same as the end UM block for execution ID *0*, the prefetching thread stops prefetching for the kernel with execution ID *0*. Then it starts prefetching block $k$ (the start UM block of the block table for execution ID *1*).

This process is called chaining. It is the process of continuously calculating UM block addresses for prefetching after a page fault has occurred. The chaining ends when a new page fault interrupt signal is raised, or the prefetching thread fails to predict the next kernel to execute. The chaining pauses when the prefetching thread has enqueued all prefetch commands for the next $N$ kernels. The prefetching thread resumes after the currently executing kernel finishes.

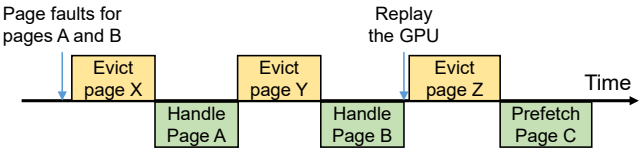



**Figure 8: A page eviction scenario.**

# 5 OPTIMIZATIONS FOR GPU PAGE FAULT HANDLING

In this section, we describe the optimization techniques for GPU page fault handling.

## 5.1 Page Pre-eviction

Page eviction occurs when the driver fails to allocate GPU memory space for migrating faulted pages. Figure 8 shows a scenario when page eviction occurs. Page eviction occurs when there is no GPU memory space available for the faulted pages to handle the faulted pages. Page eviction takes significant time, implying that the page eviction logic lies on the critical path of page fault handling. Thus, fault handling time increases when no more space is available on the GPU.

To minimize the fault handling time, the DeepUM driver pre-evicts pages when no free GPU memory space is left. A similar idea is proposed by Kim et al.[32]. The major difference is that DeepUM uses a different policy to select victim pages. The policy used by Kim et al.[32] is the same as the policy implemented in the NVIDIA driver. It evicts pages that are least recently migrated to the GPU. The DeepUM driver evicts pages that satisfy the following two conditions:

- Least recently migrated.
- Not expected to be accessed by the currently executing kernel and the next $N$ kernels predicted to execute.

Since the NVIDIA driver tracks the free spaces on the GPU side, the DeepUM obtains the available space information from the NVIDIA driver. It obtains the next executing kernel information from the execution ID correlation table.

## 5.2 Invalidating UM Blocks of Inactive PyTorch Blocks

PyTorch has different memory allocators for CPUs and GPUs. PyTorch's GPU memory allocator manages device memory pools to minimize memory allocation/free time and to reduce memory fragmentation. Two types of memory pools are managed by the GPU memory allocator: *large* and *small*.

A memory object in PyTorch is called a block. In this paper, we call it the *PT block* to distinguish it from a UM block. The large pool consists of PT blocks larger than 1MB, and the small pool consists of PT blocks less than or equal to 1MB. When a memory allocation request comes in, and the requested size is larger than 1MB, the memory allocator finds a PT block from the large pool. Otherwise, it finds a PT block from the small pool. When multiple PT blocks in the pool match the requested size, the allocator returns the smallest available PT block. In addition, the PT block is split when its size is much larger than the requested size. The selected PT block is removed from the memory pool and marked active. However, when no PT block is available in the memory pool, the GPU memory allocator allocates a new PT block by requesting a device memory space to the CUDA runtime.

After the PT block has been used by the DNN model and returned to the GPU memory allocator, the allocator inserts the PT block into an appropriate memory pool and marks it inactive. Inactive PT blocks, i.e., PT blocks in the memory pools, are freed to produce a new memory space only when no available memory space is left in the pool.

The problem arises when we use the PyTorch memory allocator with UM. When inactive PT blocks on the GPU memory are evicted to the CPU memory, unnecessary heavy data traffic occurs. In addition, they occupy CPU memory space. The problem worsens

**Table 1: System configuration.**

| | |
|---|---|
| CPU | 2 × AMD 2.35Ghz 32-core EPYC 7452 |
| Main memory | 512GB DDR4 for each node |
| OS | Ubuntu 18.04.4 LTS (kernel 4.15.0-72) |
| GPU | NVIDIA Tesla V100 PCIe 32GB<br>NVIDIA Tesla V100 PCIe 16GB |
| GPU driver | NVIDIA display driver 460.32 |
| CUDA version | 11.2 |
| cuDNN version | 8.1.1 |
| PyTorch version | 1.8.0 |

**Table 2: DNN models and datasets.**

| Model | Source | Dataset |
|---|---|---|
| GPT-2 XL[48] | Hugging Face[56] | Wikitext |
| GPT-2 L[48] | Hugging Face[56] | Wikitext |
| BERT Large[13] | Hugging Face[56] | Wikitext, GLUE CoLA |
| BERT Base[13] | Hugging Face[56] | Wikitext |
| DLRM[37] | MLPerf[36] | Cirteo Kaggle |
| ResNet200[20] | PyTorch examples[46] | ImageNet, CIFAR-10 |
| ResNet152[20] | PyTorch examples[46] | ImageNet |
| DCGAN[47] | PyTorch examples[46] | celebA |
| MobileNet[23] | PyTorch examples[46] | CIFAR-100 |

when the inactive PT blocks are marked as active and used by DNN models again. Since the pages in the PT blocks have been evicted to the CPU memory, they should be migrated to the GPU again, resulting in heavy data traffic.

To solve this problem, we add a few lines of code to the PyTorch memory allocator to tell the DeepUM driver when a PT block is marked inactive. If a victim page belongs to an inactive PT block, the DeepUM driver simply invalidates the corresponding UM block in the GPU memory.

# 6 EVALUATION

In this section, we compare the performance of DeepUM with previous approaches: LMS[33], vDNN[51], AutoTM[21], SwapAdvisor[24], Capuchin[45], and Sentinel[50].

## 6.1 Methodology

LMS (Large Model Support)[33] is an open-source project developed by IBM. It supports the PyTorch framework and automatic GPU memory swapping. We directly compare the performance of DeepUM and LMS by actually executing LMS. Since the other five approaches are based on TensorFlow[1] or other frameworks and are mostly closed-source, we indirectly compare their performance with DeepUM. Ren et al.[50] implement these approaches and measure the speedup of their training throughput over NVIDIA UM. Thus, we take the number from Ren et al. and compare them with the speedup of DeepUM over UM. We also obtain these approaches' maximum available batch sizes from the same paper. We use the same models, datasets, and GPU for a fair comparison.

*System configuration.* We use NVIDIA Tesla V100 GPUs[39] with different device memory sizes. Table 1 shows the detailed system configuration. We measure the energy consumption of the full system (including CPUs, GPU, DIMMs, motherboard, etc.) using a Hioki power meter (Hioki Model 3334 AC/DC Power HiTester[22]).

*DNN models and datasets.* We use six DNN models from various sources: Hugging Face[56], MLPerf[36], and PyTorch examples[46]. BERT and ResNet in Hugging Face and PyTorch examples are also included in MLPerf. The list of models and datasets are summarized in Table 2.

## 6.2 Comparison with Naïve UM and IBM LMS

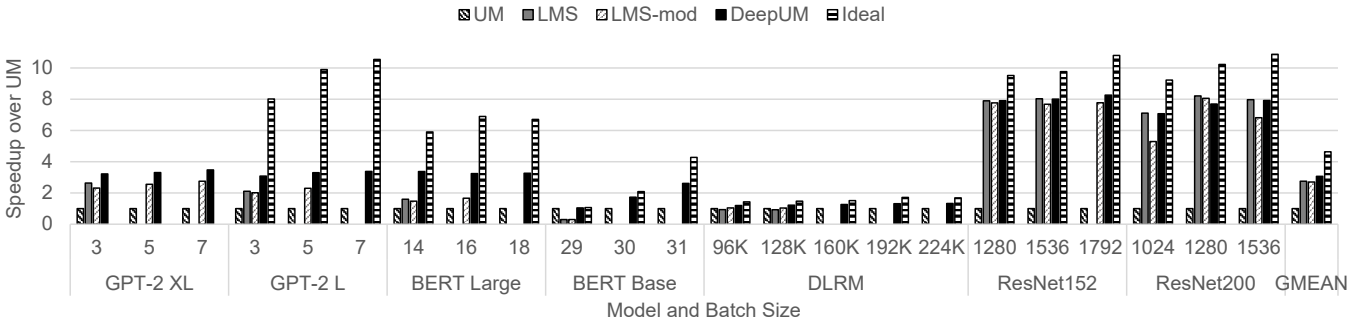
*Speedup.* Figure 9(a) shows the speedup of training throughput for each DNN model with various batch sizes and Figure 9(b) shows the elapsed time for running 100 training iterations. We use a single V100 32GB GPU. The speedup is obtained over UM that runs each DNN model using NVIDIA UM without prefetching. LMS shows the performance of the original LMS, and LMS-mod shows the performance of LMS that is modified to periodically free cached PT blocks in the PyTorch memory pool. By cleaning up cached memory objects periodically, we can reduce the occurrence of out-of-memory (OOM) errors caused by memory fragmentation. DeepUM uses UM block correlation tables, each containing 2048 rows, two-way associative, and four successors. Ideal shows the upper bounds of speedups. We obtain the upper bounds by measuring the execution times of the applications when there is no GPU memory oversubscription and scaling them up with the batch size. Also, we select the batch sizes close to the maximum batch size of LMS to enable the maximum GPU memory oversubscription. Some numbers of LMS and LMS-mod are missing because they fail to run the model in some batch sizes because of the OOM error.

Like other memory prefetching strategies, the performance depends on the application's memory access patterns and the ratio between the memory transfer time and the computation time. For example, DLRM shows almost no speedup over UM for both LMS and DeepUM. DLRM is a recommendation model introduced by Facebook, and its input data consist of users' preferences, buy lists, etc. The model looks up the embedding table for each input data item and transforms the input data item with an appropriate embedding vector. In DLRM, most of the memory space is used to store embedding tables. In addition, its memory access pattern is irregular because the embedding table lookups highly depend on the input data. This is why prefetching strategies of both LMS and DeepUM do not work well. On the other hand, ResNet consists of multiple building blocks called residual blocks. Once a memory object is prefetched for a residual block, the computation time dominates the processing of a residual block.

Note that LMS moves data at the whole tensor level while DeepUM moves data at the UM block level. While LMS is faster than LMS-mod on average, it fails to run with batch size that LMS-mod can run. DeepUM can run a larger batch size than LMS and LMS-mod because of the virtual memory system supported by DeepUM runtime.

On average, DeepUM shows the best performance. DeepUM is 3.06× faster than UM and 1.11× faster than LMS.

*Energy consumption.* Figure 9(c) shows the ratio of total energy consumption over UM. The lower the bar, the less energy consumption is over UM. The amount of energy consumption is highly

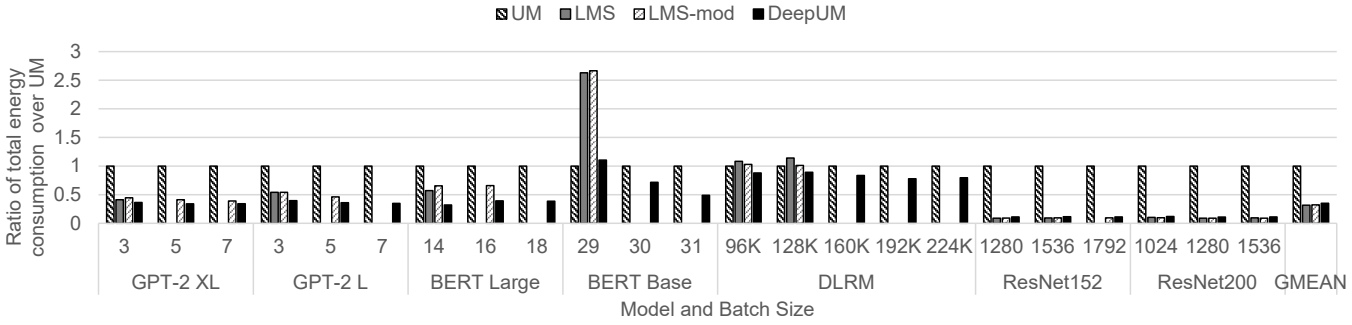



(a) The speedup of IBM LMS, DeepUM, and Ideal over the naïve UM implementation.

| Model and | GPT-2 XL | | | GPT-2 L | | | BERT Large | | | BERT Base | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Batch size | 3 | 5 | 7 | 3 | 5 | 7 | 14 | 16 | 18 | 29 | 30 | 31 |
| UM | 4597 | 7706 | 10981 | 1865 | 3839 | 5727 | 978 | 1307 | 1430 | 135 | 273 | 578 |
| LMS | 1747 | - | - | 885 | - | - | 611 | - | - | 450 | - | - |
| LMS-mod | 1990 | 3020 | 3997 | 927 | 1672 | - | 665 | 786 | - | 456 | - | - |
| DeepUM | 1429 | 2332 | 3163 | 605 | 1163 | 1695 | 290 | 403 | 438 | 129 | 158 | 222 |

| Model and | DLRM | | | | | ResNet152 | | | ResNet200 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Batch size | 96K | 128K | 160K | 192K | 224K | 1280 | 1536 | 1792 | 1024 | 1280 | 1536 |
| UM | 1203 | 1657 | 2123 | 2894 | 3318 | 31002 | 38173 | 49283 | 32420 | 44900 | 57302 |
| LMS | 1291 | 1789 | - | - | - | 3926 | 4754 | - | 4560 | 5470 | 7187 |
| LMS-mod | 1153 | 1602 | - | - | - | 3992 | 4972 | 6340 | 6124 | 5571 | 8407 |
| DeepUM | 1005 | 1363 | 1682 | 2201 | 2507 | 3922 | 4767 | 5965 | 4585 | 5835 | 7235 |

(b) Elapsed time (in seconds) for running 100 training iterations of the naïve UM implementation, IBM LMS, and DeepUM.




(c) The ratio of total energy consumption of IBM LMS and DeepUM over the naïve UM implementation.

**Figure 9: Comparison with naïve UM and IBM LMS.**

**Table 3: Maximum possible batch sizes.**

| Model | Dataset | LMS | DeepUM |
|---|---|---|---|
| GPT-2 XL | Wikitext | 3 | 16 |
| GPT-2 L | Wikitext | 3 | 24 |
| BERT Large | Wikitext | 14 | 192 |
| BERT Base | Wikitext | 29 | 256 |
| DLRM | Criteo Kaggle | 128k | 512k |
| ResNet200 | ImageNet | 1536 | 2304 |
| ResNet152 | ImageNet | 1536 | 1792 |

related to the speedup of each framework. On average, LMS requires 68% less energy than UM, and DeepUM requires 65% less energy than UM. While LMS has the best energy efficiency, the difference between LMS and DeepUM is very small.

*Maximum possible batch sizes.* Table 3 shows the maximum possible batch sizes of IBM LMS and DeepUM. DeepUM can run the models with the batch size that requires the peak memory usage to be almost the same as the total CPU memory size. The numbers in the table indicate that exploiting UM in DeepUM suffers fewer memory fragmentation issues and has a higher chance of running large DNN models without any memory problems.

*Correlation table size.* Table 4 shows the size of memory space used for storing correlation tables. Since DeepUM dynamically allocates a UM block correlation table when it finds a kernel with a new execution ID, the memory size for storing correlation tables differs for each DNN model and batch size. Note that the correlation tables are stored on the CPU side.

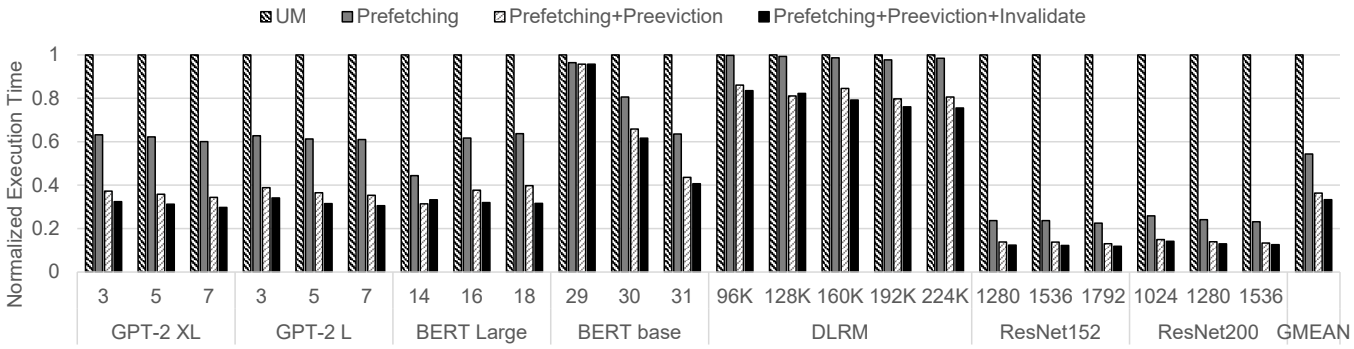



**Figure 10: Effects of prefetching and optimizations.**

**Table 4: Correlation table size.**

| Model | Batch size | Table size (MB) |
|---|---|---|
| GPT-2 XL | 3 | 308 |
| | 5 | 344 |
| | 7 | 348 |
| GPT-2 L | 3 | 169 |
| | 5 | 213 |
| | 7 | 232 |
| BERT Large | 3 | 78 |
| | 5 | 75 |
| | 7 | 74 |
| BERT Base | 3 | 19 |
| | 5 | 27 |
| | 7 | 33 |
| DLRM | 96k | 13 |
| | 128k | 19 |
| | 160k | 30 |
| | 192k | 31 |
| | 224k | 35 |
| ResNet152 | 1280 | 115 |
| | 1536 | 128 |
| | 1792 | 130 |
| ResNet200 | 1024 | 144 |
| | 1280 | 151 |
| | 1536 | 169 |

**Table 5: Average number of page faults per training iteration.**

| Model | Batch size | Fault count of UM | Fault count of DeepUM | Ratio |
|---|---|---|---|---|
| GPT-2 XL | 3 | 7437122 | 687 | < 0.1% |
| | 5 | 12395173 | 7612 | < 0.1% |
| | 7 | 17210705 | 2549 | < 0.1% |
| GPT-2 L | 3 | 2948920 | 235 | < 0.1% |
| | 5 | 6055304 | 476 | < 0.1% |
| | 7 | 8974631 | 884 | < 0.1% |
| BERT Large | 3 | 1171717 | 2913 | 0.2% |
| | 5 | 1777710 | 84 | < 0.1% |
| | 7 | 1834746 | 1355 | < 0.1% |
| BERT Base | 3 | 88459 | 1595 | 1.8% |
| | 5 | 349106 | 4536 | 1.3% |
| | 7 | 1077223 | 5531 | 0.5% |
| DLRM | 96k | 1263865 | 3706 | 0.2% |
| | 128k | 1712886 | 6912 | 0.4% |
| | 160k | 2583610 | 22624 | 0.8% |
| | 192k | 3471958 | 32139 | 0.9% |
| | 224k | 4278593 | 38437 | 0.9% |
| ResNet152 | 1280 | 121380940 | 34323 | < 0.1% |
| | 1536 | 144893625 | 72598 | < 0.1% |
| | 1792 | 182230994 | 144455 | < 0.1% |
| ResNet200 | 1024 | 126734315 | 107093 | < 0.1% |
| | 1280 | 173517031 | 68039 | < 0.1% |
| | 1536 | 207933814 | 118472 | < 0.1% |

*Number of page faults.* Original correlation prefetching is one method of cache-line prefetching for CPUs. In this case, the cache hit rate is typically used to show the effectiveness of a prefetching technique. However, the GPU we used does not have such a performance counter, and it is very hard to know whether the GPU has accessed the prefetched pages or not at a certain point. Even though there is an instrumentation tool, such as NVBit[54], it causes lots of instrumentation overhead, resulting in the prefetch timing difference. Therefore, we use the number of page faults to measure the accuracy of DeepUM prefetching technique.

Table 5 shows the average number of page faults per training iteration for each model and different batch sizes. The result indicates that DeepUM prefetches pages quite accurately and can significantly reduce page faults.

*Effects of prefetching and optimizations.* Figure 10 shows the effects of prefetching and optimizations. Prefetching shows the effect

**Table 6: Effect of parameters of the UM block correlation table.**

| Name | Assoc | NumSuccs | NumRows |
|---|---|---|---|
| Config0 | 2 | 4 | 128 |
| Config1 | 2 | 8 | 128 |
| Config2 | 4 | 4 | 128 |
| Config3 | 2 | 4 | 512 |
| Config4 | 2 | 8 | 512 |
| Config5 | 4 | 4 | 512 |
| Config6 | 2 | 4 | 1024 |
| Config7 | 2 | 8 | 1024 |
| Config8 | 4 | 4 | 1024 |
| Config9 | 2 | 4 | 2048 |
| Config10 | 2 | 8 | 2048 |
| Config11 | 4 | 4 | 2048 |
| Config12 | 2 | 4 | 4096 |

**Table 7: Maximum possible batch sizes of TensorFlow-based approaches and DeepUM.**

| Model (Dataset) | ResNet200 (CIFAR-10) | BERT Large (CoLA) | DCGAN (celebA) | MobileNet (CIFAR-100) |
|---|---|---|---|---|
| vDNN | 4.2K | not work | 1.4K | 1.2K |
| AutoTM | 5.6K | 27 | 2.5K | 3.2K |
| SwapAdvisor | 5.4K | 25 | 2.4K | 3.1K |
| Capuchin | 5.9K | 27 | 2.7K | 3.2K |
| Sentinel | 5.7K | 28 | 2.5K | 3.2K |
| DeepUM | 6.4K | 64 | 3.5K | 5.1K |

of the correlation prefetching only. Prefetching+Preeviction shows the effect of correlation prefetching and page pre-eviction described in Section 5.1. Finally, Prefetching+Preeviction+Invalidate shows the effect of all the optimization techniques mentioned in Section 5.1 and Section 5.2. Prefetching, Prefetching+Preeviction, and Prefetching+Preeviction+Invalidate reduce 45.6%, 63.7%, and 66.7% of the execution time on average.

As mentioned before, DLRM gets no benefit from prefetching due to its irregular memory access patterns. When the batch size is 29 in BERT-base, the effect of prefetching is very small. This is because a tiny portion of the total memory usage is oversubscribed (approximately 3% of the total memory usage).

Overall, the result indicates that the prefetching and optimization techniques in DeepUM are very effective.

*Sensitivity to the degree of prefetching.* DeepUM requires a few other CPU threads to update correlation tables and manage queues. Such tasks do not incur significant energy/power consumption or performance overhead because most of their work is table lookup/update and queueing. However, prefetching may contribute to more energy/power consumption. Moreover, aggressive prefetching may hurt performance because the prediction can be wrong and unnecessary pages can be migrated to the GPU. As a result, it may wastes memory bandwidth and evicts pages that will be accessed soon.

DeepUM prefetches pages predicted to be accessed by the following $N$ kernels. The user can statically control the degree of prefetching by modifying $N$. To verify the impact of aggressive prefetching, we measure applications' execution time and energy consumption while varying the degree of prefetching ($N$). Figure 11 shows the result for various values of $N$. Figure 11(a) shows the speedup and Figure 11(b) shows the ratio of total energy consumption of different values of $N$ over $N = 8$. They indicate that the speedup and energy consumption is inversely proportional. Also, there is a sweet spot ($N = 32$) where the speedup is the highest, and the energy consumption is the lowest.

## 6.3 Parameters of the UM Block Correlation Table

There are several configuration parameters of the UM block correlation table: the number of immediate successor blocks (NumSuccs), the number of rows in the table (NumRows), and the associativity of the table (Assoc). To find the optimal configuration, we perform a sensitivity analysis.

Table 6 shows different configurations for the UM block correlation table, and Figure 12 shows the speedups of the different configurations over Config0 using V100 32GB GPU for each configuration. We see that, on average, configuration *9* shows the best performance.

## 6.4 Comparison with TensorFlow-Based Approaches

We compare the performance of DeepUM with TensorFlow-based approaches: vDNN[51], AutoTM[21], SwapAdvisor[24], Capuchin[45], and Sentinel[50]. Figure 13 shows the speedup for each DNN model. The speedup is obtained on a V100 16GB GPU. Note that the numbers are obtained from Ren et al.[50], and they measure the speedup of the training throughput over NVIDIA UM without prefetching. Moreover, we obtain the upper bounds of speedups (Ideal) in the same way as in Figure 9.

Table 7 shows the maximum possible batch size of the TensorFlow-based approaches and DeepUM. To measure the maximum possible batch sizes, we limit the total CPU memory usage of DeepUM to 128GB to match the system configuration with the TensorFlow-based approaches.

Overall, DeepUM is faster than IBM LMS and other TensorFlow-based approaches than Sentinel. DeepUM shows comparable performance to Sentinel. Note that Sentinel's swapping mechanism is not transparent to the user while DeepUM's is fully automatic, as shown in Table 8. Moreover, DeepUM allows a much larger batch size than other previous approaches.

Note that previous approaches manage data at the DNN layer or tensor level. It implies that all data accessed in a layer or a tensor are moved together. The performance difference comes from the more fine-grained data movement of DeepUM, where memory objects are prefetched and evicted in a more fine-grained manner with accurate prediction through the correlation tables.

# 7 RELATED WORK

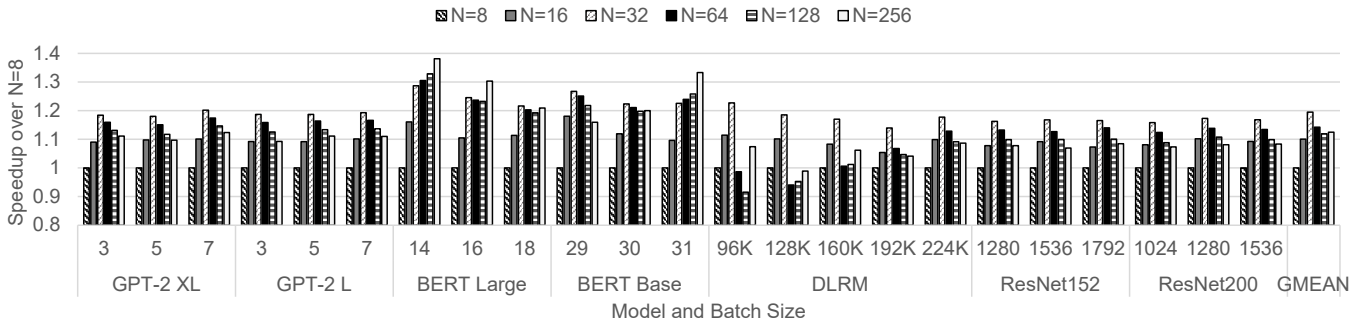
Many studies have been performed to overcome the GPU memory capacity problem by exploiting the swapping mechanism. There are two categories in the swapping approach. One swaps-in/swaps-out the whole GPU memory objects to the CPU memory or NVMe devices[6, 21, 24, 33, 45, 49–51, 55].

vDNN[51] is the first approach that introduces the GPU memory swapping for Deep Neural Network (DNN) workloads. However, the DNN models must use vDNN API functions, and it supports only convolutional neural networks (CNNs).

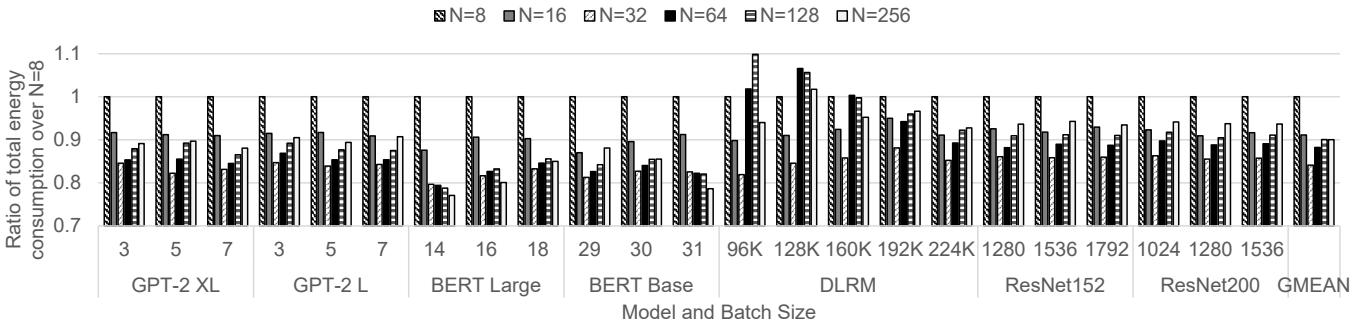
TFLMS[33] is an open-source project developed by IBM and included in the IBM Watson Machine Learning software package. It schedules swap-in/swap-out commands by modifying computational graphs of TensorFlow. It requires modifying both TensorFlow and TensorFlow user scripts. IBM also released the PyTorch version of TFLMS.

Both Superneurons[55] and FlashNeuron[6] take a DNN model as input and derive an optimal tensor offloading schedule. Superneurons offloads tensors in the GPU memory to the CPU memory while FlashNeuron[6] exploits NVMe SSDs to offload the tensors.

AutoTM[21] and SwapAdvisor[24] also take a DNN model as input to schedule its memory operations. AutoTM uses integer




(a) The speedup over $N = 8$, varying the degree ($N$) of prefetching.




(b) The ratio of total energy consumption over $N = 8$, varying the degree ($N$) of prefetching (the lower, the better).

**Figure 11: Sensitivity to degree of prefetching.**

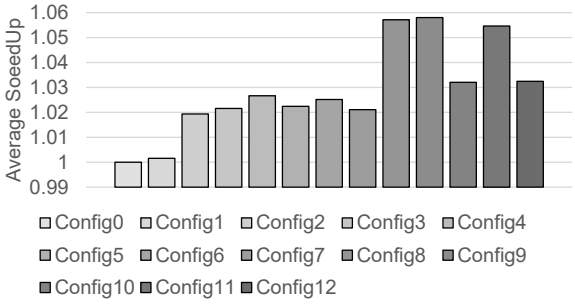



**Figure 12: Performance when varying the parameters of UM block correlation table.**

linear programming (ILP) not only to schedule data movement but also to reduce GPU memory fragmentation. On the other hand, SwapAdvisor[24] uses a genetic algorithm to schedule operators, memory allocations, and swap decisions. Capuchin[45] identifies the tensor access patterns at run time and schedules tensor eviction, prefetching, and recomputation.

Sentinel[50] is based on TensorFlow and dynamically gathers tensor access information from both the TensorFlow runtime and the operating system. It is similar to DeepUM because it exploits the page fault mechanism to profile and obtain the memory access patterns. However, the Sentinel uses the CPU page fault mechanism while DeepUM uses the GPU page fault mechanism of the GPU. In the profiling phase of the Sentinel, tensors are allocated in pinned memory of the CPU, and the GPU accesses it. Another difference between the Sentinel and DeepUM is that the Sentinel requires modifying TensorFlow and TensorFlow user scripts to insert Sentinel profiling API functions calls. The evaluation result of Sentinel shows that it outperforms vDNN, SwapAdvisor, AutoTM, and Capuchin in training throughput.

DeepSpeed[49] is a highly optimized and widely used deep learning framework for multiple GPUs. It keeps track of the sequence of DNN operations by hooking PyTorch APIs. Then, it provides the GPU memory swapping mechanism with the CPU main memory or NVMe SSD. However, DeepSpeed manages offloading model parameters, gradients, and optimizer states, not activations. The programmer should manage activations and temporary buffers manually through activation checkpointing. This implies that when the amount of activations exceeds the GPU memory size, the programmer must modify their code for activation checkpointing.

The other category of the swapping approach exploits CUDA UM with prefetching[5, 35]. OC-DNN[5] manually inserts prefetch commands in front of each DNN operation. DRAGON[35] uses an NVMe SSD as a backing store for UM. It targets general applications and uses a DNN workload as a showcase. It also requires user code modification and device driver modification.

Table 8 summarizes the differences between the previous approaches and DeepUM. In Table 8, the base Deep Learning (DL) framework is the underlying DL framework on which the proposal works. Some entries are left blank because they implement their idea from scratch. DL framework modification means that the base DL framework code should be modified to implement the proposal. User script modification means that the user script that specifies

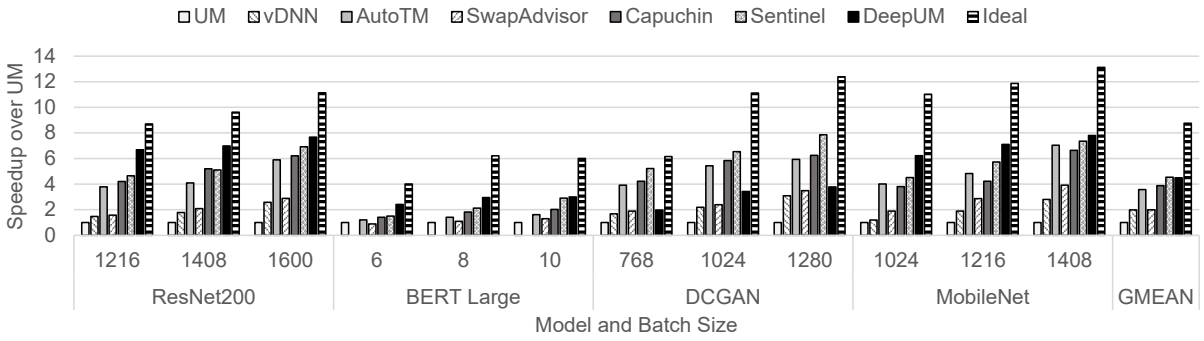



**Figure 13: Comparison with TensorFlow-based approaches.**

**Table 8: Comparison with previous approaches and DeepUM.**

| Name | Base DL framework | DL framework modification | User script modification | Run-time profiling |
|---|---|---|---|---|
| vDNN[51] | - | - | Y | N |
| TFLMS[33] | TensorFlow | Y | Y | N |
| Superneurons[55] | - | - | Y | N |
| FlashNeuron[6] | PyTorch | Y | N | N |
| AutoTM[21] | nGraph | Y | Y | N |
| Capuchin[45] | TensorFlow | Y | N | Y |
| SwapAdvisor[24] | MXNet | Y | Y | Y |
| Sentinel[50] | TensorFlow | Y | Y | Y |
| DeepSpeed[49] | PyTorch | N | Y | Y |
| DeepUM | PyTorch | Y | N | Y |

and describes DNN models and training process should be changed, or the user script should be modified with the special APIs proposed. Run-time profiling means whether the proposal performs profiling at run time to gather information.

DeepUM differs from the previous approaches in that it exploits UM to allow GPU memory oversubscription and uses a correlation prefetching technique to predict future memory accesses. DeepUM also proposes various optimization techniques to reduce the GPU page fault handling time. While DeepUM requires few modifications in the PyTorch framework, it neither requires user code modification nor operating system kernel/device driver modification. That is, it is fully automatic.

## 8 CONCLUSIONS

In this paper, we propose DeepUM, which allows GPU memory oversubscription for DNNs by exploiting CUDA Unified Memory and using the CPU memory as a backing store. While CUDA UM allows GPU memory oversubscription using a page fault mechanism, it introduces enormous overhead. We use a new correlation prefetching technique at the UM block level to hide the fault-handling overhead. We also introduce two optimization techniques to minimize the GPU fault handling time. One is a page pre-eviction technique, and a new page pre-eviction policy is coupled with correlation prefetching. The other is invalidating the unnecessary pages for PyTorch when they are selected as a page-eviction victim.

DeepUM driver hooks the page fault handler of the NVIDIA device driver so that it can monitor the page faults and prefetch the required pages. It also allows an easy and clean install of the Linux kernel module because the user does not need to change the code and recompile the NVIDIA device driver or OS kernel. DeepUM runtime can be loaded using the `LD_PRELOAD` environment variable. DeepUM can be easily turned on and off by setting the environment variable. While our approach requires few code modifications in the original PyTorch source code to change the behavior of the PyTorch memory allocator, no user Python code modification is required.

We evaluate the performance of DeepUM using nine large-scale DNN models from multiple sources: MLPerf, PyTorch examples, and Hugging Face and compare it with six state-of-the-art GPU memory swapping approaches. The evaluation result shows that DeepUM achieves comparable performance to one approach whose swapping mechanism is not transparent to the user. DeepUM achieves much better performance than the remaining five approaches. At the same time, DeepUM can handle larger models that other methods fail to handle.

## ACKNOWLEDGMENTS

This work was supported in part by the Institute for Information & communications Technology Promotion (IITP) grant (No. 2018-0-00581, CUDA Programming Environment for FPGA Clusters) and by the BK21 Plus program for BK21 FOUR Intelligence Computing (Dept. of Computer Science and Engineering, SNU, No. 4199990214639) through National Research Foundation of Korea (NRF), all funded by the Ministry of Science and ICT (MSIT) of Korea. ICT at Seoul National University provided research facilities for this study.

## REFERENCES


[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] T. Alexander and G. Kedem. 1996. Distributed prefetch-buffer/cache design for high performance memory systems. In *Proceedings. Second International Symposium on High-Performance Computer Architecture*. 254–263. https://doi.org/10.1109/HPCA.1996.501191

[3] An-Chow Lai, C. Fide, and B. Falsafi. 2001. Dead-block prediction dead-block correlating prefetchers. In *Proceedings 28th Annual International Symposium on Computer Architecture*. 144–154. https://doi.org/10.1109/ISCA.2001.937443

[4] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 136–150.

[5] Ammar Ahmad Awan, Ching-Hsiang Chu, Hari Subramoni, Xiaoyi Lu, and Dhabaleswar K. Panda. 2018. OC-DNN: Exploiting Advanced Unified Memory Capabilities in CUDA 9 and Volta GPUs for Out-of-Core DNN Training. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. 143–152. https://doi.org/10.1109/HiPC.2018.00024

[6] Jonghyun Bae, Jongsung Lee, Yunho Jin, Sam Son, Shine Kim, Hakbeom Jang, Tae Jun Ham, and Jae W. Lee. 2021. FlashNeuron: SSD-Enabled Large-Batch Training of Very Deep Neural Networks. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 387–401. https://www.usenix.org/conference/fast21/presentation/bae

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, T. J. Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[8] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[9] Tien-Fu Chen and Jean-Loup Baer. 1992. Reducing Memory Latency via Non-Blocking and Prefetching Caches. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, USA) *(ASPLOS V)*. Association for Computing Machinery, New York, NY, USA, 51–61. https://doi.org/10.1145/143365.143486

[10] Esha Choukse, Michael B. Sullivan, Mike O'Connor, Mattan Erez, Jeff Pool, David Nellans, and Stephen W. Keckler. 2020. Buddy Compression: Enabling Larger Memory for Deep Learning and HPC Workloads on GPUs. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 926–939. https://doi.org/10.1109/ISCA45697.2020.00080

[11] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 3123–3131.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/n19-1423

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *ArXiv* abs/1810.04805 (2019).

[14] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[15] Khronos group. 2022. OpenCL Overview - The Khronos Group Inc. https://www.khronos.org/opencl/.

[16] Audrūnas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. Memory-Efficient Backpropagation through Time. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) *(NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 4132–4140.

[17] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37* (Lille, France) *(ICML'15)*. JMLR.org, 1737–1746.

[18] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 1135–1143.

[19] 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. https://doi.org/10.1109/CVPR.2016.90

[20] 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. https://doi.org/10.1109/CVPR.2016.90

[21] 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. https://doi.org/10.1145/3373376.3378465

[22] HIOKI. 2022. AC/DC POWER HiTESTER 3334. https://www.hioki.com/global/products/power-meters/single-phase-ac-dc/id_6045.

[23] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *ArXiv* abs/1704.04861 (2017).

[24] 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. https://doi.org/10.1145/3373376.3378530

[25] Intel. 2022. oneAPI Programming Model. https://www.oneapi.com/.

[26] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. 2018. Gist: Efficient Data Encoding for Deep Neural Network Training. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, 776–789. https://doi.org/10.1109/ISCA.2018.00070

[27] D. Joseph and D. Grunwald. 1999. Prefetching using Markov predictors. *IEEE Trans. Comput.* 48, 2 (1999), 121–133. https://doi.org/10.1109/12.752653

[28] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Proteus: Exploiting Numerical Precision Variability in Deep Neural Networks. In *Proceedings of the 2016 International Conference on Supercomputing* (Istanbul, Turkey) *(ICS '16)*. Association for Computing Machinery, New York, NY, USA, Article 23, 12 pages. https://doi.org/10.1145/2925426.2926294

[29] Jaehoon Jung, Daeyoung Park, Youngdong Do, Jungho Park, and Jaejin Lee. 2020. Overlapping Host-to-Device Copy and Computation Using Hidden Unified Memory. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 321–335. https://doi.org/10.1145/3332466.3374531

[30] Jaehoon Jung, Daeyoung Park, Youngdong Do, Jungho Park, and Jaejin Lee. 2020. Overlapping Host-to-Device Copy and Computation Using Hidden Unified Memory. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 321–335. https://doi.org/10.1145/3332466.3374531

[31] Khronos. 2021. The OpenCL Specification. https://www.khronos.org/registry/OpenCL/specs/opencl-2.1.pdf#page=174.

[32] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. 2020. *Batch-Aware Unified Memory Management in GPUs for Irregular Workloads*. Association for Computing Machinery, New York, NY, USA, 1357–1370. https://doi.org/10.1145/3373376.3378529

[33] Tung D. Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. 2018. TFLMS: Large Model Support in TensorFlow by Graph Rewriting. *ArXiv* abs/1807.02037 (2018).

[34] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A Framework for Memory Oversubscription Management in Graphics Processing Units. 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, 49–63. https://doi.org/10.1145/3297858.3304044

[35] Pak Markthub, Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Satoshi Matsuoka. 2018. DRAGON: Breaking GPU Memory Capacity Limits with Direct NVM Access. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 414–426. https://doi.org/10.1109/SC.2018.00035

[36] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debo Dutta, Udit Gupta, Kim Hazelwood, Andy Hock, Xinyuan Huang, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. 2020. MLPerf Training Benchmark. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 336–349. https://proceedings.mlsys.org/paper/2020/file/02522a2b2726fb0a03bb19f2d8d9524d-Paper.pdf

[37] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Mikhail Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *ArXiv* abs/1906.00091 (2019).

[38] NVIDIA. 2021. Unified Memory Programming. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd.

[39] NVIDIA. 2022. Artificial Intelligence Architecture | NVIDIA Volta. https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/.

[40] NVIDIA. 2022. CUDA Parallel Computing Platform. https://developer.nvidia.com/cuda-zone.

[41] NVIDIA. 2022. NVIDIA H100 Tensor Core GPU Architecture. https://nvdam.widen.net/s/9bz6dw7dqr/gtc22-whitepaper-hopper.

[42] NVIDIA. 2022. Pascal GPU Architecture. https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/.

[43] OpenACC-standard.org. 2022. OpenAcc. https://www.openacc.org/.

[44] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary 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. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[45] 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. https://doi.org/10.1145/3373376.3378505

[46] PyTorch. 2022. PyTorch Examples. https://github.com/pytorch/examples.

[47] Alec Radford, Luke Metz, and Soumith Chintala. 2016. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1511.06434

[48] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[49] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. https://doi.org/10.1145/3394486.3406703

[50] 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. https://doi.org/10.1109/HPCA51647.2021.00057

[51] 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.

[52] Nikolay Sakharnykh. 2017. Maximizing Unified Memory Performance in CUDA. https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/.

[53] Y. Solihin, Jaejin Lee, and J. Torrellas. 2002. Using a user-level memory thread for correlation prefetching. In *Proceedings 29th Annual International Symposium on Computer Architecture*. 171–182. https://doi.org/10.1109/ISCA.2002.1003576

[54] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 372–383. https://doi.org/10.1145/3352460.3358307

[55] 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. https://doi.org/10.1145/3178487.3178491

[56] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Scao, Sylvain Gugger, and Alexander 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*. 38–45.

[57] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2019. Large batch optimization for deep learning: Training BERT in 76 minutes. *arXiv preprint arXiv:1904.00962* (2019).

Received 2022-07-07; accepted 2022-09-22