2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)

RT-Swap: Addressing GPU Memory Bottlenecks for Real-Time Multi-DNN Inference

Woosung Kang[1], Jinkyu Lee[2], Youngmoon Lee[3], Sangeun Oh[4], Kilho Lee[5], Hoon Sung Chwa[1*]

[1]*Dept. of Electrical Engineering and Computer Science, DGIST*, Republic of Korea
[2]*Dept. of Computer Science and Engineering, Sungkyunkwan University (SKKU)*, Republic of Korea
[3]*Dept. of Robotics, Hanyang University*, Republic of Korea
[4]*Department of Software & Computer Engineering, Ajou University*, Republic of Korea
[5]*School of AI Convergence, Soongsil University*, Republic of Korea

2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS) | 979-8-3503-5841-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/RTAS61025.2024.00037

***Abstract*—The increasing complexity and memory demands of Deep Neural Networks (DNNs) for real-time systems pose new significant challenges, one of which is the GPU memory capacity bottleneck, where the limited physical memory inside GPUs impedes the deployment of sophisticated DNN models. This paper presents, to the best of our knowledge, the first study of addressing the GPU memory bottleneck issues, while simultaneously ensuring the timely inference of multiple DNN tasks. We propose RT-Swap, a real-time memory management framework, that enables transparent and efficient swap scheduling of memory objects, employing the relatively larger CPU memory to extend the available GPU memory capacity, without compromising timing guarantees. We have implemented RT-Swap on top of representative machine-learning frameworks, demonstrating its effectiveness in making significantly more DNN task sets schedulable at least 72% over existing approaches even when the task sets demand up to 96.2% more memory than the GPU's physical capacity.**

I. Introduction

The increasing use of Deep Neural Networks (DNNs) in real-time systems has led to the evolution of ever larger and more intricate models to achieve superior accuracy on complex tasks [1]–[3]. This increase in complexity, in turn, accelerates the memory demands of these DNNs, introducing a significant hurdle, especially when deploying on GPUs. Despite the rapid advancements in GPU computational capabilities, GPU memory capacity has seen comparatively slower growth, leading to a severe bottleneck when accommodating the ever-increasing memory demands of DNN models [4]–[6]. This memory bottleneck not only constrains the adoption of sophisticated DNN models but also poses additional challenges in assuring worst-case timing guarantees in real-time systems, emphasizing the need for efficient memory management.

A number of prior studies [7]–[10] have made significant efforts to alleviate this GPU memory bottleneck by utilizing relatively larger CPU memory or SSDs. For example, GPUswap [8] allows GPU applications to allocate memory exceeding the physical GPU memory capacity by relocating data between GPU and CPU memories. A recent study [9] enables GPU memory oversubscription by utilizing direct I/O to SSDs targeting for embedded CPU–GPU integrated SoCs that share the physical memory between CPU and GPU. However, these approaches typically require the usage of their own device driver or modifications to existing GPU drivers, rendering them *incompatible* with popular machine learning (ML) frameworks such as PyTorch [11], TensorFlow [12], and Darknet [13], which are fundamental to the execution of DNN models.

Modern GPUs (e.g., NVIDIA RTX3090) support Unified Memory (UM), which enables a unified virtual memory space between CPU and GPU, and permits GPU memory oversubscription using hardware-based page fault handling and automatic data migration on UM. Some studies [14], [15] exploit UM to allow ML frameworks to execute a DNN model demanding a memory footprint larger than the physical GPU memory capacity. They prefetch the necessary data from CPU to GPU by predicting the DNN model's memory access pattern, reducing the page fault handling overhead. Additionally, other recent studies [16]–[20] offer direct (i.e., non-UM) GPU memory swapping mechanisms. These approaches profile the memory access pattern of a target DNN model and schedule data movement between CPU and GPU memories based on the memory access pattern. Despite the valuable contributions made by these approaches in overcoming the memory bottleneck tailored to DNN models, they primarily focus on training a single DNN model instead of inferencing multiple DNN models, and do not address the timing constraints associated with managing data movement, thus rendering them infeasible for real-time systems.

In this paper, we aim to bridge the gap between the limited capacity of GPU memory and the ever-increasing complexity of DNN models, while concurrently ensuring timely inference for multiple DNN tasks. To achieve this goal, we propose a new real-time memory management framework, named RT-Swap, specifically designed for DNN tasks. One of the key features of RT-Swap is a transparent and efficient swapping mechanism, which empowers an ML framework to perform inference on DNN tasks requiring a memory capacity larger than the physical GPU memory, without any alterations to the DNN models or modifications to the GPU device driver. Another key feature is its consistent virtual memory management mechanism, ensuring seamless data access for DNN tasks regardless of memory swapping. Building upon the above two features, RT-Swap develops a swap-aware real-time scheduling policy that determines both the maximum swap volume for each task and the schedule of swap-in/out operations in coor-

*Corresponding author: Hoon Sung Chwa (chwahs@dgist.ac.kr).

2642-7346/24/$31.00 ©2024 IEEE
DOI 10.1109/RTAS61025.2024.00037

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on July 12,2024 at 03:45:24 UTC from IEEE Xplore. Restrictions apply.

dination with the execution schedule of DNN tasks, ensuring their timely inferences. To this end, we derive a schedulability analysis to verify whether DNN tasks satisfy their timing constraints, taking into account the swapping overheads. We then formulate the swap volume assignment as an optimization problem with respect to the proposed schedulability analysis. We also develop a swap scheduling algorithm to minimize unnecessary swap operations and maximize the overlap between computation and data transfer, mitigating the overhead of memory swapping.

RT-Swap offers three distinct benefits. First, it is applicable to existing ML frameworks since it is implemented as a shared library that can work with a target ML framework. RT-Swap transparently intercepts all memory-related requests from the ML framework and manages the memory objects using the proposed swapping mechanism. Thus, it does not require any source code modification of DNN models running on a target ML framework. Second, it provides an efficient swapping mechanism. RT-Swap effectively addresses crucial fragmentation issues arising from frequent memory allocations and deallocations during swap operations. This ensures efficient GPU memory usage and avoids unnecessary swapping. Third, it guarantees predictable and deterministic swapping overheads and automatically coordinates the execution and swapping of DNN tasks at a system level to meet timing constraints.

We implement RT-Swap using Darknet [13] as the base ML framework and extend its compatibility to PyTorch [11][1]. We evaluate RT-Swap with four standard DNN models (YOLOv3 [21], ResNet [22], ResNext [23], and DenseNet [24]) on a GPU computing platform equipped with NVIDIA RTX3090 with 24 GB memory. The evaluation results demonstrate that RT-Swap can accommodate at least 72% more real-time DNN task sets compared to existing approaches even when requiring up to 96.2% more memory than the GPU's physical capacity, without violating any timing constraints. We also validate, via a runtime experiment, that RT-Swap guarantees predictable response times for DNN tasks with only marginal runtime overheads in comparison to NVIDIA's GPU memory oversubscription with UM.

Contribution. To the best of our knowledge, this paper presents a first approach that transparently expands the available memory capacity of a GPU, thereby addressing the memory bottleneck issue, while simultaneously ensuring timely inference for multiple DNN tasks. The contributions of this paper can be summarized as follows:

- We present a new GPU memory management framework that allows for the deployment of multiple DNN tasks that require more memory than physically available on a GPU, while still ensuring timing constraints are met (Sec. III).
- We design transparent and efficient swapping and virtual memory management mechanisms to reduce fragmentation overheads (Sec. IV)
- We develop swap-aware real-time scheduling and swap volume assignment algorithms that not only provide timing guarantees but also mitigate the swapping overhead (Sec. V).
- We implement an RT-Swap prototype on popular ML frameworks and demonstrate its capability in accommodating more real-time DNN tasks that demand more memory than the GPU's capacity, without violating any timing constraints (Secs. VI and VII).

[1] RT-Swap's source code is publicly accessible at https://rtcl.dgist.ac.kr/index.php/rtswap.

TABLE I: CUDA runtime APIs for memory management

Function	Description
`cudaMalloc()`	Allocates memory on the device
`cudaMallocManaged()`	Allocates memory as UM memory
`cudaFree()`	Frees memory allocated on the device
`cudaMemcpy()`	Copies data between the host and device
`cudaMemset()`	Initializes or sets device memory

II. Background

A. Target System

DNN tasks. A real-time DNN task involves a single DNN model, processing one inference request per job (task's instance) within a specific deadline. Each model consists of input, output, and multiple hidden layers, executed sequentially during inference. Essential model parameters like input data, feature maps, weights, and convolution space primarily occupy the GPU memory. Typically, DNN task inference in most ML frameworks is a two-step process: initialization, where each task's model parameters are preloaded into the GPU memory, followed by the inference stage where computations are performed using the preloaded parameters.

GPU memory management. We focus on a system composed of a GPU and a CPU with separate physical memories, shared by multiple DNN tasks for inference. Using CUDA runtime APIs, tasks allocate GPU memory as *memory objects*, representing contiguous virtual memory areas. Each object is mapped to the physical memory by the GPU driver, limited by the GPU's physical capacity. DNN tasks, once scheduled on the GPU, can access data only if it is stored on the GPU memory. Since the GPU lacks direct access to CPU (host) memory—unless deliberately mapped by a programmer into the GPU memory space[2]—it necessitates manual data migration between the GPU and CPU memories, facilitated by specific CUDA runtime APIs, as outlined in Table I.

B. Unified Memory and On-demand Paging

Modern GPUs offer Unified Memory (UM), enabling a shared virtual memory address space between the CPU and GPU with automatic data migration via on-demand paging. This allows the GPU to access pages in CPU memory, enabling DNN tasks to operate beyond GPU memory capacity, i.e., memory oversubscription. In CUDA, `cudaMallocManaged` allocates data accessible by both CPU and GPU via a single shared pointer. A page fault is triggered if the GPU accesses an unmapped virtual page, which is resolved by remapping and data copying, a process called on-demand paging. When the GPU's physical memory is full, the GPU driver moves a page from the GPU to the CPU to accommodate a new page, using the Least Recently Used (LRU) policy for page eviction. Although UM with on-demand paging enables GPU memory oversubscription, the LRU policy is imprecise, making the amount of evicted pages for each inference job unpredictable, rendering it infeasible for real-time multi-DNN systems.

[2] Note that accessing mapped CPU memory in the GPU space lowers performance due to PCIe transfers during each memory access.

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on July 12,2024 at 03:45:24 UTC from IEEE Xplore. Restrictions apply.

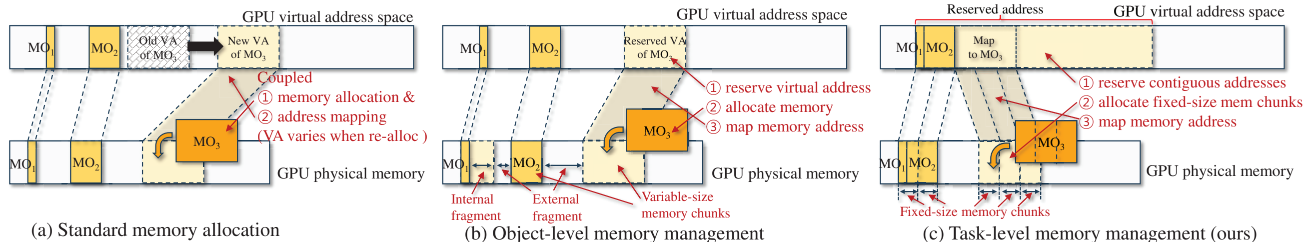

Fig. 1: Three different GPU memory management approaches

C. GPU Virtual Memory Management

The CUDA runtime offers low-level GPU virtual memory management (VMM) APIs. These differ from standard high-level functions such as `cudaMalloc`, where the GPU driver fully controls physical memory allocation and the mapping of virtual to physical addresses. The following three primary VMM APIs enable the separation of virtual and physical addresses of GPU memory, allowing for their independent management: `cuMemCreate` creates a physical memory chunk of a specific size; `cuMemAddressReserve` reserves a virtual address range; and `cuMemMap` maps a physical memory chunk to a virtual address range. Utilizing these APIs, we aim to provide transparent virtual GPU memory access to DNN tasks regardless of memory swapping.

III. Design Principle

RT-Swap is designed to bridge the gap between the limited capacity of GPU memory and the ever-increasing complexity of DNN models, also ensuring timely inference. It operates as a runtime memory management system, virtualizing the memory usage of DNN tasks across GPU and CPU memories. RT-Swap enables the execution of larger DNNs beyond the GPU's physical capacity by transparently managing the data's allocation and movement, while assuring timely inferences. To achieve this, RT-Swap tackles the following key challenges:

C1. How to allocate GPU memory exceeding the physical memory capacity?
C2. How to provide transparent GPU memory access for DNN tasks even when allocating more GPU memory than is physically available?
C3. How to provide an offline timing guarantee to DNN tasks in conjunction with the solutions to C1 and C2?

C1. All DNN task data must be pre-loaded into the GPU memory before execution. To tackle C1, a runtime data transfer support between the GPU and CPU memories is essential. When the available GPU memory falls short for a DNN task's GPU memory allocation request, some data associated with other tasks should be moved from GPU to CPU memory to make room (referred to as *swap out*). Conversely, when a DNN task requires access to data that is not currently in GPU memory, that data should be transferred back to GPU memory and reallocated (referred to as *swap in*).

C2. Addressing C1 involves implementing swap-in/out operations, necessitating the allocation and deallocation of GPU memories and data transfers between CPU and GPU. For C2, it is essential to make swap-in/out operations transparent to DNN tasks, maintaining consistent data access despite runtime data transfers. Particularly, reallocation after swapping back in assigns a new GPU memory space and virtual address, causing discrepancies with initial addresses and potential data access errors during execution. Hence, it is crucial to preserve transparency, ensuring consistent mapping of original virtual addresses to their corresponding data during swap operations.

C3. The implementation of swap-in/out operations introduces additional overhead, primarily due to the data transfers required between CPU and GPU memories. This extra overhead inevitably impacts the inference time of each DNN task. To tackle C3, we need to determine the maximum swap volume (the maximum amount of data to be swapped in/out) for each DNN task and the schedule of swap-in/out operations so as to ensure all timing constraints are met despite the added overhead of swap operations.

RT-Swap addresses C1–C3 with related design principles.

Task-level Predictable Memory Swapping. To ensure each task's timely inference under memory swapping, RT-Swap introduces a runtime swap manager that controls the amount of data to be swapped in/out at a task level and orchestrates each task's data movement, allocation, and deallocation. RT-Swap keeps track of all allocated memory objects and the cumulative GPU memory assigned to each DNN task. When the free GPU memory is insufficient to handle a memory allocation request, RT-Swap determines which memory objects should be swapped out to the CPU memory, thus freeing up GPU memory for the request. If a scheduled DNN task has any memory objects that have been previously swapped out, RT-Swap performs a swap-in operation, reallocating the memory objects to GPU memory before execution.

Transparent GPU Virtual Memory Management. RT-Swap utilizes CUDA-supported VMM APIs to provide a seamless DNN task experience, ensuring consistent memory access despite runtime GPU memory allocations and deallocations from swap operations. Unlike the standard `cudaMalloc`, which assigns a new virtual address during the reallocation of a swapped-out memory object (as shown in Fig. 1(a)), VMM APIs maintain virtual address consistency during memory object swapping. In Fig. 1(b), a virtual address (VA) range is typically reserved equivalent to the desired memory object size, paired with a *single* physical memory chunk of matching size. The VA remains reserved even when its physical chunk is unmapped during a swap-out, allowing a new physical chunk to be remapped to the original VA without inconsistency when swapped back in. An object-level VMM approach, however, leads to significant internal fragmentation issues. VMM APIs inherently use 2MB as the minimum physical chunk size. So, when allocating each memory object smaller than this

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on July 12,2024 at 03:45:24 UTC from IEEE Xplore. Restrictions apply.

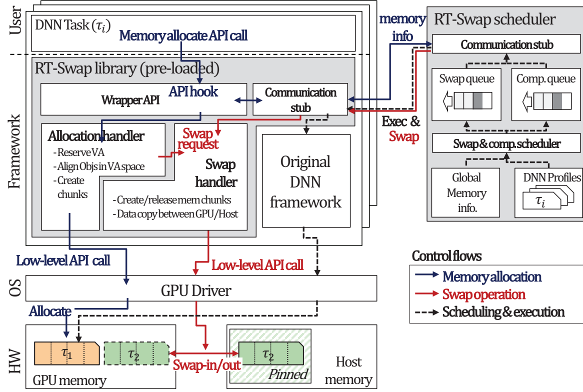

Fig. 2: System architecture overview

size, 2MB of memory is consumed, leaving a considerable portion unused (internal fragmentation), as illustrated by MO_1 in Fig. 1(b). Additionally, the object-level approach produces physical memory chunks in varied sizes for different memory objects, resulting in increased external fragmentation. To resolve these issues, RT-Swap adopts two strategies: maintaining a contiguous task-level VA range and sequentially mapping uniform physical chunks. As shown in Fig. 1(c), RT-Swap reserves a large, *contiguous task-level* VA range and sequentially allocates memory objects to consistent, *uniform* physical chunks. This method minimizes internal fragmentation within each physical chunk at a task level and prevents external fragmentation by using consistently sized physical chunks.

Swap-Aware Real-Time Scheduling. For a given taskset, RT-Swap determines each DNN task's maximum swap volume to ensure: 1) continual data retrieval from the GPU memory during execution, and 2) completion of execution before the deadline. We accomplish this by deriving a schedulability analysis, considering memory swapping overheads. We then formulate swap volume assignment as an optimization problem. This involves formulating swap volume assignment as an optimization problem, aiming to minimize the total swap volume while ensuring 1) and 2) with respect to the proposed schedulability analysis. Upon determining each task's maximum swap volume, RT-Swap identifies candidate memory objects to meet the swap volume. This list of candidates then informs the actual selection of memory objects to be swapped out during runtime. RT-Swap also coordinates the schedule of swap-in/out operations with the DNN tasks' execution schedule. It ensures that necessary memory objects are swapped in before a DNN task's execution, and sufficient free GPU memory is maintained by swapping out volumes from previously executed tasks. RT-Swap aims to minimize unnecessary swap-in/out operations and maximize the overlap between computation and data transfer, thereby efficiently mitigating the swapping overhead.

IV. System Design

Based on the aforementioned design principle, we design RT-Swap as a runtime memory management framework that can be integrated into popular ML frameworks. The system overview of RT-Swap is depicted in Fig 2. RT-Swap consists of two key components: RT-Swap Library and RT-Swap Scheduler.

A. RT-Swap Library

RT-Swap Library provides wrapper functions for relevant GPU memory API functions, some of which are exemplified in Table I. The rationale behind this is that each DNN task must use the API functions to allocate and access the GPU memory. This design enables the library to have control over the GPU management, leveraging the extensive CPU memory as a swapping device for facilitating swap-in/out operations during runtime. RT-Swap Library is implemented as a shared library, enabling pre-loading into a target ML framework by setting the `LD_PRELOAD` environment variable, without any source code modifications of DNN models and the GPU driver. This allows for the interception of target CUDA runtime API function calls, facilitating the execution of corresponding wrapper functions implemented in RT-Swap Library.

Initialization. In the initialization phase, RT-Swap Library is paired with each DNN task, holding essential information such as the maximum swap volume and a list of candidate memory objects to meet the swap volume. Detailed discussions on offline swap volume assignment and swapping candidate selection are in Sec. V-D. Aligned with our design principles, RT-Swap Library reserves a single, *contiguous* VA range, equivalent to the maximum swap volume, using `cuMemAddressReserve`. This reserved VA range is used for sequentially allocating memory objects in the swap candidate list and for dynamically mapping and unmapping physical memory chunks based on swap-in/out operation requests. It is beneficial to secure a contiguous VA range preemptively for swap candidates. This not only reduces fragmentation overhead but also ensures transparent GPU virtual memory access for all tasks (further details in subsequent sections discussing fragmentation overheads). During runtime, RT-Swap Library manages the GPU memory, handling allocations/deallocations and data transfers involved in swap operations.

Memory allocation/deallocation. RT-Swap Library operates at invocations of CUDA memory allocation API functions (i.e., `cudaMalloc`). Two scenarios exist when allocating a memory object: i) the memory object is a swap candidate, or ii) it is not. For non-swap candidates, RT-Swap Library simply executes the original API function call, ensuring that the memory object persistently resides in the GPU memory. For swap candidates, RT-Swap Library first identifies the placement of the memory object within the reserved VA range, aligning it sequentially with previous objects, as shown by MO_1–MO_3 in Fig. 1(c). It then creates multiple uniform physical chunks using `cuMemCreate`, ensuring their total size covers the memory object size, and maps these chunks to the object's VA range using `cuMemMap`. In both scenarios, RT-Swap Library communicates with RT-Swap Scheduler to confirm adequate free GPU memory to accommodate the new memory object. If the free GPU memory is scarce, RT-Swap Scheduler triggers a swap-out operation with another DNN task's RT-Swap Library, ensuring adequate GPU memory is released before allocating the new memory object. Memory deallocation requests (i.e., `cudaFree`) can be managed in a similar manner.

Swapping out/in. RT-Swap Library performs swap-in/out operations as directed by RT-Swap Scheduler at runtime, as shown in Fig. 3. RT-Swap adopts a uniform physical memory chunk as the basic swapping unit to avoid external fragmentation. This size is referred to as the *swap chunk size*. When there is a

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on July 12,2024 at 03:45:24 UTC from IEEE Xplore. Restrictions apply.

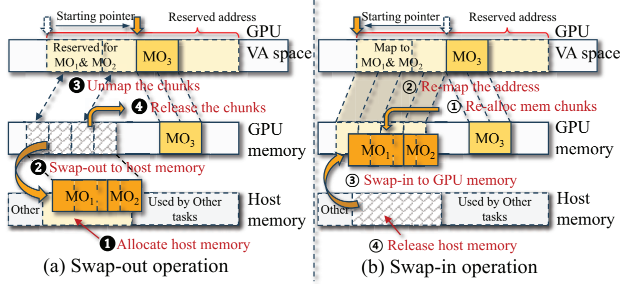

Fig. 3: Swap-out/in operation procedures

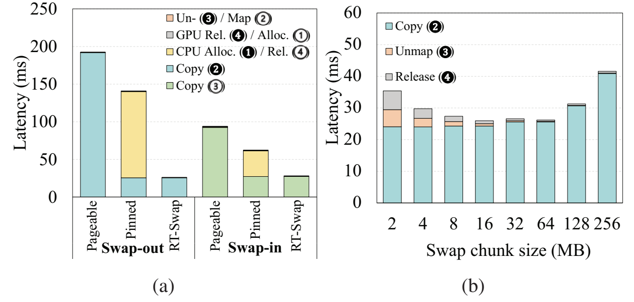

Fig. 4: Swap latency of 300MB memory: (a) latency breakdown of three methods; (b) impact of swap chunk sizes

request to swap out a specific amount of GPU memory, RT-Swap Library identifies *sequential* target physical chunks within the reserved VA range. This identification starts from the initial pointer and covers up to the size of the requested swap-out, which is rounded up to the closest multiple of the swap chunk size. RT-Swap Library then updates the starting pointer for future swap-out operations. The swap-out operation includes several steps as illustrated in Fig 3(a): ❶ allocating CPU memory, ❷ copying data from the target GPU physical chunks to the CPU memory, ❸ unmapping the target physical chunks from their corresponding GPU VA range, and ❹ releasing the physical chunks. RT-Swap Library also maintains a record of the GPU VA range and corresponding CPU VA range of the sequential target chunks, ensuring they can be accurately swapped back in later. The swap-in operation also involves several steps as illustrated in Fig 3(b): ① creating GPU physical memory chunks, ② mapping these chunks to the corresponding GPU VA range, ③ copying data from the CPU memory to the new GPU memory chunks, and ④ releasing the CPU memory.

Swap-in/out operations can incur noticeable performance overheads. Thus, it is essential to reduce these costs. The CUDA runtime offers two main types of CPU memory for swapping devices: *pageable* and *host-pinned*. Pageable memory, allocated using the standard `malloc` function, requires additional temporary pinned memory for data transfers between the CPU and GPU. In contrast, host-pinned memory, allocated using `cudaHostAlloc`, permits direct data transfers, omitting the need for temporary buffers and additional copying steps. Thus, utilizing host-pinned memory is a more efficient swapping method, reducing overall overheads. Additionally, steps ❶ and ④ can be omitted during the swap-out and -in operations, respectively, further enhancing efficiency. Given that RT-Swap pre-determines each task's maximum swap volume offline, it limits the maximum data transfer between the CPU and GPU to this volume. Consequently, RT-Swap Library pre-allocates host-pinned memory equivalent to the maximum swap volume during initialization, serving as a dedicated swapping space for each task throughout the inference stage.

Fig. 4(a) shows the latency in swapping out and in 300MB of GPU memory, with a 64MB swap chunk size, using three methods: Pageable, Pinned, and RT-Swap Library. Pageable and Pinned represent the swap-out and -in operations (i.e., ❶–❹ and ①–④), utilizing CPU pageable and host-pinned memory, respectively. RT-Swap Library mirrors Pinned, except it omits steps ❶ and ④ during the swap-out and -in operations, respectively. Pageable incurs the highest latency primarily due to the overheads of using temporary host-pinned memory in steps ❷ and ③, resulting in 192.6ms and 93.9ms for swap-out and -in operations, respectively. Pinned, despite using direct data transfers, still has significant latency due to overheads in other steps. Contrastingly, RT-Swap Library, omitting steps ❶ and ④, optimizes latency effectively, taking 26.2ms and 28.3ms for the swap-out and -in operations, respectively, and showing substantial improvements compared to the other two methods. Note that the swapping latency depends on the requested size and swap chunk size. In addition to minimizing swapping overheads, RT-Swap aims to hide the overheads as much as possible, maximizing the overlap between GPU computation and swapping, to be discussed in Sec. V.

TABLE II: Internal fragmentation overhead

	ResNet	DenseNet	ResNext	YOLOv3
Resolution	(416)	(416)	(416)	(416)
# of MO $<$ 2MB	1009	1689	962	372
# of MO $\geq$ 2MB	312	129	359	282
Total amount (GB)	1.4	0.9	2.2	2.6
MO-level VMM (GB)	3.5 (+2.1)	4.0 (+3.1)	4.2 (+2.0)	3.5 (+0.9)
RT-Swap (GB)	1.4 (+0.000)	0.9 (+0.001)	2.2 (+0.000)	2.6 (+0.001)

Mitigating fragmentation overheads. RT-Swap is designed to mitigate both internal and external fragmentation overheads inherent in the typical object-level VMM approach. Regarding internal fragmentation, we observe that DNN computation uses a wide range of parameter sizes, from a few KB to several hundreds of MB. Table II outlines the number of memory allocation requests from four DNN models, categorized by sizes. For instance, ResNet requests allocation for 1009 memory objects (MOs) (76.4%) under 2MB and 312 MOs (23.6%) larger than 2MB. Note that the memory allocation via CUDA VMM APIs is confined to a minimum physical memory chunk size of 2MB (as imposed by the GPU driver), the object-level VMM approach implies that every memory object smaller than 2MB will occupy a 2MB physical memory chunk. This approach leaves a significant portion of memory unused, leading to increased internal fragmentation. Such fragmentation triggers unnecessary swapping operations, making task sets hardly schedulable. Unlike the object-level VMM approach, RT-Swap Library pre-reserves a contiguous task-level VA range for potential swapping candidates, organizing the allocation sequentially. This ensures that each object's starting pointer aligns with the end of the preceding object. Therefore, RT-Swap bounds the internal fragmentation overhead within a given memory chunk size for each DNN task. This is made possible through task-level virtual memory management, as opposed to

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on July 12,2024 at 03:45:24 UTC from IEEE Xplore. Restrictions apply.

the conventional object-level approach. As shown in Table II, for ResNet, the object-level approach requires 3.5GB of GPU memory to allocate only 1.4GB of memory objects, yielding 2.1GB of GPU memory unusable. In contrast, RT-Swap requires only an additional 1MB beyond the total memory footprint.

Regarding external fragmentation, the object-level approach aggravates the issue by creating physical memory chunks of varied sizes, each corresponding to different memory object sizes. This variation in sizes results in inefficient memory usage and increased external fragmentation; for example, if a larger memory chunk is swapped out to make space for a smaller one, a significant portion of the freed GPU memory remains unused. Contrary to the object-level VMM approach, RT-Swap Library employs physical memory chunks of a consistent, uniform size for all memory objects. When a specific memory object size is requested for allocation, RT-Swap Library creates multiple uniform physical memory chunks, cumulatively sufficient to meet the memory object size. Moreover, RT-Swap uses a consistent physical memory chunk as the basic swapping unit, preventing external fragmentation.

Choosing an appropriate physical memory chunk size, which corresponds to the swap chunk size, is critical, as it involves a trade-off: managing a higher number of physical memory chunks or facing increased internal fragmentation overhead. We measure the swap-out latency of 300MB, varying the swap chunk size between 2MB and 256MB, as shown in Fig. 4(b). A larger swap chunk size leads to more copy time (❷) due to higher internal fragmentation overheads. Conversely, a smaller swap chunk size results in longer processing times in steps ❸ and ❹ due to the increased number of physical memory chunks needing to be unmapped and released. For instance, a 256MB swap chunk size allows RT-Swap Library to manage only two physical memory chunks, but necessitates the transfer of 512MB of data between the GPU and CPU memory, despite only 300MB being essential. In contrast, a 2MB swap chunk size necessitates managing 150 physical memory chunks but avoids internal fragmentation. The results in Fig. 4(b) show that the swap-out latencies for 2MB and 256MB swap chunk sizes are 35.3ms and 41.6ms, respectively. The optimal performance, 26.2ms, is achieved with a 64MB swap chunk size. A similar trend is also observed in the swap-in operation. Note that the optimal swap chunk size may vary based on the swap-in/out size; this is further discussed in Sec. V-D.

B. RT-Swap Scheduler

RT-Swap Scheduler prioritizes scheduling by accounting for the timing requirements of DNN tasks. RT-Swap Scheduler provides cooperative scheduling of GPU computations and memory swapping. This involves aligning swap-in/out operations with inference job schedules and vice versa, by managing *swap* and *computation* queues systematically. This approach efficiently hides swapping overhead while ensuring accurate and timely inference results for DNN tasks.

RT-Swap Scheduler operates as an independent process, maintaining system-wide information gathered from all running DNN tasks, and communicates with individual RT-Swap Library associated with each task. During initialization, RT-Swap Scheduler establishes a connection with each RT-Swap Library and interacts through an IPC-based *communication stub* interface that is implemented within each library and the scheduler. To develop offline swap volume and swap chunk size assignments, RT-Swap Scheduler maintains a detailed profile for each task's DNN model including its timing constraint, GPU execution time, swap-in/out latency, and set of memory objects.

During inference, RT-Swap Scheduler collaboratively manages each task's execution timing and the schedule of swap-in/out operations. RT-Swap Scheduler continuously monitors the available free GPU memory and maintains an updated list of physical memory chunks that have been swapped out for each task. Before the execution of a DNN task with swapped-out physical chunks, RT-Swap Scheduler proactively issues a swap-in request to the corresponding RT-Swap Library of that task. If there is limited free GPU memory available to process the swap-in request, RT-Swap Scheduler issues swap-out requests to other tasks ahead of the swap-in request. The details of scheduling and swap volume assignment will be in Sec. V.

V. Swap-Aware Real-Time Scheduling

So far, we have discussed the execution of DNN tasks by RT-Swap when the memory requirement exceeds the GPU's physical memory capacity. Now, we will discuss how RT-Swap schedules swap operations concurrently with the execution of DNN tasks, and determines the optimal swap chunk size and swap volume for each task to meet their timing requirements.

A. Task Model

We consider a CPU–GPU platform π with a GPU physical memory capacity denoted as m^D. The CPU physical memory capacity is assumed to be sufficiently large, as it can be easily expanded with multiple DRAMs. We denote the swap chunk size as δ, which represents the unit of memory swapping.

We represent real-time DNN inference tasks using a periodic task model, commonly utilized in various real-time systems. Each task is assumed to utilize one DNN model and handles one inference request per job. Each DNN task $\tau_i \in \tau$ can be specified as $\tau_i = (M_i, C_i, T_i, D_i)$. Here, M_i is the memory profile; C_i is the worst-case execution time (WCET) without swap operations; T_i is the period; and D_i, the relative deadline, is equivalent to T_i.

The memory profile M_i is further characterized by (m_i, $m_i^S, x_i, \Theta_i, O^{In}(x_i, \delta), O^{Out}(x_i, \delta)$), each of which is detailed below. m_i is the total GPU memory footprint of τ_i, assumed to remain within the GPU's memory capacity m^D. This footprint comprises *swappable* memory objects, denoted as m_i^S, which are allocated using CUDA runtime APIs such as `cudaMalloc` and are manageable by RT-Swap. Note that m_i^S is a subset of m_i, with the remaining memory being non-swappable components like static allocations and shared libraries. x_i denotes the maximum swap volume, a portion of m_i^S, determined offline by our proposed algorithm, ensuring each task τ_i does not exceed this swap limit to meet its timing constraint. Note that x_i is a multiple of the swap chunk size δ. Among all swappable GPU memory objects, we also denote a list of candidate memory objects for swapping as Θ_i. $O^{In}(x_i, \delta)$ and $O^{Out}(x_i, \delta)$ are the maximum times required for swapping in and out, respectively, estimated using linear regression, considering x_i and δ. All parameters in M_i can be determined offline (see Sec. VII for details).

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on July 12,2024 at 03:45:24 UTC from IEEE Xplore. Restrictions apply.

Each task τ_i potentially generates an infinite sequence of jobs every T_i time-units. Each job must complete execution within a relative deadline of D_i time-units. The priority ordering of DNN tasks is determined according to the Earliest Deadline First (EDF) policy. Note that GPU has separate computation and data copy engines, so the scheduler can schedule both computation and swap operations in parallel. When it comes to preemptiveness, we assume the execution of DNN tasks and each swap operation are non-preemptive.

B. Target Scheduling Problems

As we target a task set where the cumulative memory demand surpasses the GPU memory capacity, it is necessary to handle swap-in/out operations, which meet the following swap requirements.

- R1. Prior to the execution of τ_i, it is necessary to swap in the previously swapped-out memory volume, up to x_i.
- R2. Prior to τ_i's swap-in operation, a sufficient volume of free GPU memory must be ensured for τ_i. This is accomplished by swapping out a corresponding volume, up to x_i, from other DNN tasks.

Since the swap-in/out operations incur additional delay to each task's execution, the requirement for timing guarantees can be written as follows.

- R3. Adhering to R1 and R2, all jobs of all tasks $\tau_i \in \tau$ must meet deadlines for all possible legal job arrival sequences.

To satisfy the requirements R1–R3, we need to schedule not only the task executions on GPU but also the swap-in/out operations thereof, which entails the following two problems: (i) how to determine the execution order of the task executions on GPU *and* their swap-in/out operations, and (ii) how to determine the amount of swap-in/out operations *and* the swap chunk size, which can be formally stated as follows.

*Problem 1 (**P1**):* Given the swap chunk size δ and swap volume assignments $\{x_i\}$, design a swap-aware scheduling algorithm such that **G1)** the number of swap operations for each job of τ_i is bounded (as a very small number) and **G2)** the overlap between computation and swapping is maximized, both without compromising the swap requirements R1 and R2.

*Problem 2 (**P2**):* Given a task set τ with the proposed swap scheduling algorithm as a solution of P1, determine x_i for every $\tau_i \in \tau$ and δ such that the swap requirements R1–R2 and the timing guarantee requirement R3 are satisfied.

We will propose the solutions of P1 and P2 in Sections V-C and V-D, respectively.

C. Swap Schedule Generation

Since each of the inference tasks and swap-in/out operations is non-preemptive, the scheduler is invoked upon (i) request of a new inference job (`REQUEST`) or (ii) completion of a job or a swap operation (`COMPLETION`). The scheduler maintains two queues: one for inference requests (computation queue), and the other for swap requests (swap queue). In case of a `REQUEST` event, the newly requested job of τ_i is placed into the computation queue. Upon each invocation (either `REQUEST` or `COMPLETION`), both the jobs in the computation queue and the swap requests in the swap queue are scheduled under EDF. Let $\tau^{CPU}(t)$ denote the subset of τ for which *any* portion of their swap volumes resides in CPU memory at time t. When a job of τ_i is selected to be scheduled at time t, the scheduler checks whether τ_i belongs to $\tau^{CPU}(t)$. If $\tau_i \in \tau^{CPU}(t)$, its swap-in operation is placed into the swap queue to satisfy R1, and the execution of τ_i is blocked until its swap-in operation is completed. The scheduler also checks whether there is a sufficient amount of free GPU memory. If there is an insufficient amount of free GPU memory, the swap-out operations of other DNN tasks except τ_i are placed into the swap queue ahead of τ_i's swap-in operation to satisfy R2. Let $\tau^{GPU}(t)$ denote the subset of τ for which *any* portion of their swap volumes resides in GPU memory at time t. We choose the tasks in $\tau^{GPU}(t)$ to be swapped out in order of the latest future job release time first until enough volume is freed up to perform the swap-in operation of τ_i.

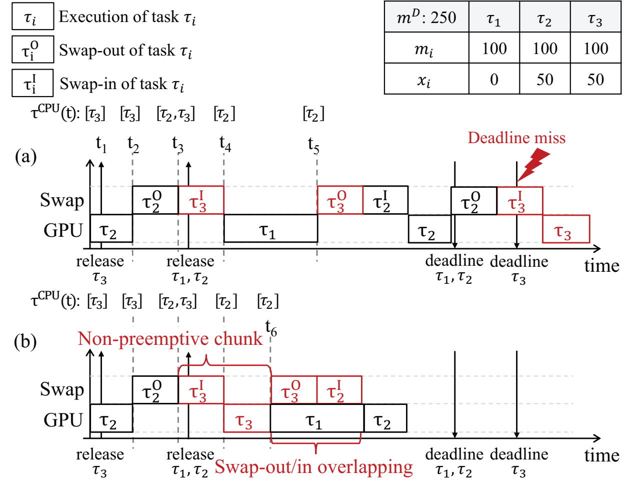

Fig. 5: Schedules for swap operations and normal executions: (a) unnecessary swap operations performed, (b) which is addressed by the RT-Swap Scheduler

Addressing G1 for P1. Although the basic mechanism of RT-Swap Scheduler satisfies both R1 and R2, the mechanism, as it is, may produce unnecessary swap operations, hindering timely inferences for DNN tasks, as described as follows. As shown in Fig. 5(a), three tasks (τ_1, τ_2, τ_3) are presented, each with m_1=m_2=m_3=100, and x_1=0, x_2=x_3=50, given a GPU memory capacity of m^D=250. τ_1 can be scheduled without any swap operation. To schedule τ_2 and τ_3, it is necessary to swap out one task and swap in the other. τ_3 requests its inference job at time t_1 and it is scheduled at t_2. Since $\tau_3 \in \tau^{CPU}(t_2)$ holds and no free GPU memory is available at t_2, the scheduler performs the swap-out operation of τ_2 at t_2 and then the swap-in operation of τ_3 at t_3. In the middle of the swap-in operation of τ_3, a new inference job of τ_1 and that of τ_2 whose priority is higher than τ_3 are released. Since the job of τ_1 has a higher priority than that of τ_2 and $\tau_1 \notin \tau^{CPU}(t_4)$ holds, the job execution of τ_1 is started at t_4. Once the job execution of τ_1 is finished at t_5, the job of τ_2 (whose priority is higher than that of τ_3) will be scheduled ahead of τ_3 at t_5. Consequently, the scheduler performs the swap-out operations of τ_3 to secure free GPU memory for τ_2's swap-in operation

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on July 12,2024 at 03:45:24 UTC from IEEE Xplore. Restrictions apply.

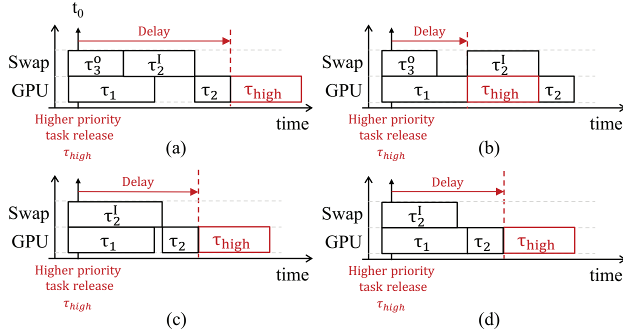

Fig. 6: Blocking scenarios: (a) undesirable long blocking delay, (b) which is addressed by RT-Swap Scheduler, (c) blocking delay upper-bounded by $O^{In}(x_i, \delta)+C_i$, and (d) by $C_{max,1}+C_{max,2}$

at time t_5. In this situation, the swap-in operation of τ_3 at t_3 is unnecessary since τ_3 is not scheduled due to the new release of the higher priority task τ_2. If this situation is repeated, the swapping overhead imposed on τ_3 is unnecessarily increased and unbounded.

To prevent this situation, RT-Swap Scheduler does not allow any preemption of higher priority tasks between the swap-in operation of a task and its normal execution. In Fig. 5(b), after the swap-in operation of τ_3 at t_3, RT-Swap Scheduler executes τ_3 by considering the swap-in operation of τ_3 and its execution as a non-preemptive region without allowing any preemption of higher-priority tasks between the swap-in operation of τ_3 and its execution. By the policy, RT-Swap Scheduler achieves G1 for P1, preserving the following property.

Property 1: Under RT-Swap Scheduler, every job of a task $\tau_i \in \tau$ performs at most one swap-in operation and one swap-out operation.

Addressing G2 for P1. RT-Swap Scheduler tries to maximize the overlap between computation and swapping to effectively hide the swap overhead. To this end, RT-Swap Scheduler performs the swap operations proactively in conjunction with the execution of inference jobs. Let $Q_c(t)$ denote the set of jobs in the computation queue at time t. If the highest priority job J_x in $Q_c(t)$ does not belong to $\tau^{CPU}(t)$ (i.e., no need to perform the swap-in operation), RT-Swap Scheduler schedules the swap-in operation of the highest priority job J_y in $Q_c(t) \cap \tau^{CPU}(t)$ (i.e., the job that requires the swap-in operation the earliest) to overlap it with the execution of J_x. Similarly, if the swap-out operation is required before J_y's swap-in operation, it is scheduled together with the execution of J_x. As shown in Fig. 5(b), at time t_6, τ_1 is the highest priority job in $Q_c(t_6)$ and does not belong to $\tau^{CPU}(t_6)$. τ_2 is the second highest priority in $Q_c(t_6)$ and belongs to $\tau^{CPU}(t_6)$, requiring the swap-out operation of τ_3 to swap in τ_2's memory. RT-Swap Scheduler schedules the swap-out operation of τ_3 and the execution of τ_1 in parallel, and subsequently schedules the swap-in operation for τ_2 in the middle of τ_1's execution. Therefore, τ_2 can be executed right after the execution of τ_1 without any delay caused by swap operations.

Although such proactive swap operations can effectively hide their overheads, they may impose high blocking delay on an inference job that does not require swap operations. Consider the situation in Fig. 6(a): the swap-in operation of τ_2 is not finished until the execution of τ_1 is finished, and therefore the execution of τ_2 cannot start right after the execution of τ_1. Under this situation, if a new highest-priority job of τ_{high} that does not require swap operations is released at t_0, its blocking delay by lower-priority jobs of τ_1 and τ_2 is longer than the sum of the execution time of τ_1 and τ_2 as shown in the figure; in fact, the blocking delay becomes longer as the swap-in operation time of τ_2 gets longer. This situation occurs due to the discrepancy between the highest-priority job in the swap queue and that in the computation queue when the highest-priority job in the computation queue does not require swap operations. To avoid this situation, we make the scheduler start the highest-priority swap-in operation *only if* the corresponding execution in the computation queue is the highest-priority, resulting in Fig. 6(b) for the same situation.

D. Swap Volume Assignment

We formulate **P2** as an optimization problem such that the total swap volume for a task set is minimized while R1–R2 (the swap requirements) and R3 (the timing guarantee requirement) are satisfied.

$$\underset{x_i,\,\delta}{\text{minimize}} \quad \sum_{\tau_i \in \tau} x_i \tag{1a}$$

$$\text{subject to} \quad 0 \le x_i \le m_i^S, \textit{ for all } \tau_i \in \tau, \tag{1b}$$

$$\sum_{\tau_i \in \tau} m_i - \sum_{\tau_j \neq \tau_i} x_j \ \le\ m^D, \textit{ for all } \tau_i \in \tau, \tag{1c}$$

$$\text{Eq. (3) in Theorem 1 holds.} \tag{1d}$$

Addressing the swap requirements. Constraint (1b) specifies that the maximum swap volume x_i for each task τ_i must be less than or equal to the amount of swappable memory m_i^S, which is an obvious constraint. Constraint (1c) represents a sufficient condition for a task set τ to fulfill R1 and R2 given the GPU memory capacity m^D. For each task τ_i, the left-hand side (LHS) of the constraint specifies the minimal necessary GPU memory allocation, which includes the maximum swap volume x_i required for the correct execution of τ_i (satisfying R1), while also considering the swap-out of the maximum swap volumes of all tasks other than τ_i. Then, to satisfy R2, the LHS value must be less than or equal to the GPU memory capacity m^D; this must hold for every task $\tau_i \in \tau$, which is Constraint (1c). To account for each task's internal fragmentation overhead, the total memory footprint m_i is adjusted by adding a margin up to δ, ensuring that m_i is rounded up to the nearest multiple of δ.

Addressing the timing guarantee requirement. To verify whether a task set associated with the swap chunk size δ and swap volumes $\{x_i\}$ meets all timing constraints under RT-Swap Scheduler (satisfying R3), we develop schedulability analysis by employing the existing utilization-based non-preemptive EDF schedulability analysis [25], [26]. We first upper-bound the total time for a job of each τ_i to occupy the computing system (either by its swap-in/out operations or GPU execution), by $O^{Out}(x_i, \delta) + O^{In}(x_i, \delta) + C_i$, which respectively represent the maximum time for swapping out to secure the memory space for τ_i, the maximum time for swapping in to transfer data for τ_i, and the maximum GPU execution time

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on July 12,2024 at 03:45:24 UTC from IEEE Xplore. Restrictions apply.

of τ_i; then, $\big(O^{Out}(x_i,\delta)+O^{In}(x_i,\delta)+C_i\big)/T_i$ is the task utilization for τ_i, to be used in Theorem 1. Second, we need to calculate B_{max}, the maximum blocking time for a higher-priority task's job to be delayed due to its lower-priority task(s) (either by their swap-in/out operations or GPU execution); we calculate B_{max} under RT-Swap Scheduler as follows.

Lemma 1: Under RT-Swap Scheduler, the maximum time a task is blocked by a lower-priority task's execution or swap operation (denoted by B_{max}) is calculated by

$$B_{max}=\max_{\tau_i\in\tau}\Big(O^{Out}(x_i,\delta),O^{In}(x_i,\delta)+C_i,C_{max,1}+C_{max,2}\Big), \tag{2}$$

where $C_{max,n}$ is the nth largest C_i for $\tau_i\in\tau$.

Proof: Suppose that a higher-priority task τ_{high} is released at t_0. We consider six cases depending on the existence/type of blocking by a lower-priority task τ_{low} (and another lower-priority task $\tau_{low'}$) at t_0.

(Case 1) At t_0, no swap-in/out operations and no GPU execution of any τ_{low}: $B_{max}=0$ holds trivially.

(Case 2) At t_0, the GPU execution of τ_{low} only: Since τ_{high} is blocked by the GPU execution of τ_{low} only, B_{max} is upper-bounded by $\max_{\tau_i\in\tau}C_i$.

(Case 3) At t_0, the swap-out execution of τ_{low} only: Since τ_{high} is blocked by the swap-out execution of τ_{low} only, B_{max} is upper-bounded by $\max_{\tau_i\in\tau}O^{Out}(x_i,\delta)$.

(Case 4) At t_0, the swap-in execution of τ_{low} only: According to RT-Swap Scheduler, the GPU execution of τ_{low} is followed by its swap-in execution without any higher-priority task preemption. Therefore, B_{max} is upper-bounded by $\max_{\tau_i\in\tau}O^{In}(x_i,\delta)+C_i$.

(Case 5) At t_0, the swap-out execution of τ_{low} and GPU execution of $\tau_{low'}$: According to Cases 2 and 3, B_{max} is upper-bounded by the maximum of $\max_{\tau_i\in\tau}C_i$ and $\max_{\tau_i\in\tau}O^{Out}(x_i,\delta)$. One may argue that if the swap-out of τ_{low} finishes before the GPU execution of $\tau_{low'}$, it is possible for another lower-priority task $\tau_{low''}$ to start its execution as shown in Fig. 6(a). However, this situation cannot occur as RT-Swap Scheduler starts the highest-priority swap-in operation *only if* the corresponding execution in the computation queue is the highest-priority, yielding the situation in Fig. 6(b).

(Case 6) At t_0, the swap-in execution of τ_{low} and GPU execution of $\tau_{low'}$: We consider two sub-cases. First, if the swap-in execution of τ_{low} is finishes later than the GPU execution of $\tau_{low'}$, B_{max} is the same as that for Case 4, upper-bounded by $\max_{\tau_i\in\tau}O^{In}(x_i,\delta)+C_i$, as shown in Fig. 6(c). Second, if the swap-in execution of τ_{low} finishes no later than the GPU execution of $\tau_{low'}$, the GPU execution of τ_{low} can start its execution only after that of $\tau_{low'}$ is finished. In this case, B_{max} is upper-bounded by the sum of two GPU executions of different tasks, which is $C_{max,1}+C_{max,2}$.

For all cases, B_{max} is upper-bounded by the right-hand-side of Eq. (2), which proves the lemma. ■

Once we apply B_{max}, we apply the schedulability analysis in [25], [26] as follows.

Theorem 1: If Eq. (3) holds for given δ and $\{x_i\}$, τ is schedulable by the proposed scheduling algorithm (i.e., no job deadline miss is guaranteed).

$$\frac{B_{max}}{\min_{\tau_i\in\tau}T_i}+\sum_{\tau_i\in\tau}\frac{O^{Out}(x_i,\delta)+O^{In}(x_i,\delta)+C_i}{T_i}\le 1.0 \tag{3}$$

Proof: The theorem holds by applying the schedulability analysis in [25], [26] to the fact that the total execution of each task τ_i is upper-bounded by $O^{Out}(x_i,\delta)+O^{In}(x_i,\delta)+C_i$ and the maximum blocking time is B_{max}. ■

Final solution. The optimization problem is solved using Gurobi [27], a well-known optimization solver, by restricting possible solutions for $\{x_i\}$ to multiples of δ. In addition, to determine a list Θ_i of candidate memory objects for swapping among all swappable memory objects, they are organized in descending order based on their sizes. They are sequentially added to Θ_i until the cumulative size of Θ_i meets or exceeds x_i. This approach allows RT-Swap to manage a reduced number of CPU VA to GPU VA mappings for swapping within Θ_i, thereby optimizing maintenance efforts.

VI. Implementation

We initially implement RT-Swap using Darknet [13] as the base ML framework and subsequently extend its compatibility to PyTorch [11]. RT-Swap consists of two main components: RT-Swap Library and RT-Swap Scheduler. RT-Swap Library, designed as a shared library, facilitates easy integration with any ML framework by simply preloading it through the `LD_PRELOAD` environment variable, avoiding source code changes. We implement the swap handler in RT-Swap Library using signals (`SIGUSR1` and `SIGUSR2` in POSIX) to execute swap-in/out operations based on RT-Swap Scheduler's requests. To avoid conflicts between inference and swap requests, the swap handler functions as a separate thread.

RT-Swap Scheduler, operating as a standalone process, manages communication with RT-Swap Library for each DNN task, requiring only minimal adjustments to the ML framework's code. For example, incorporating RT-Swap into Darknet requires fewer than 20 lines of code to establish an IPC interface. During the initialization phase of a DNN model within Darknet (in `parse_network_cfg` within *parser.c*), a dual IPC connection using named pipes is established for blocking I/O, consisting of one channel for the DNN task to send scheduling requests to RT-Swap Scheduler, and another for RT-Swap Scheduler to return scheduling decisions. In the inference phase, Darknet employs a specific function (`network_predict` in *network.c*) to execute a series of GPU kernels as per the DNN model's specifications. We integrate the communication stub at the beginning of this function, allowing the DNN task to forward a scheduling request to RT-Swap Scheduler upon initiating an inference job. The DNN task is then paused, awaiting a scheduling decision from RT-Swap Scheduler. RT-Swap Scheduler then takes charge of managing GPU computations and memory swapping for DNN tasks.

Compatibility. To showcase the compatibility of RT-Swap with other state-of-the-art ML frameworks, we extend its implementation to include PyTorch. PyTorch differentiates itself with a sophisticated *caching allocator* for GPU memory

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on July 12,2024 at 03:45:24 UTC from IEEE Xplore. Restrictions apply.

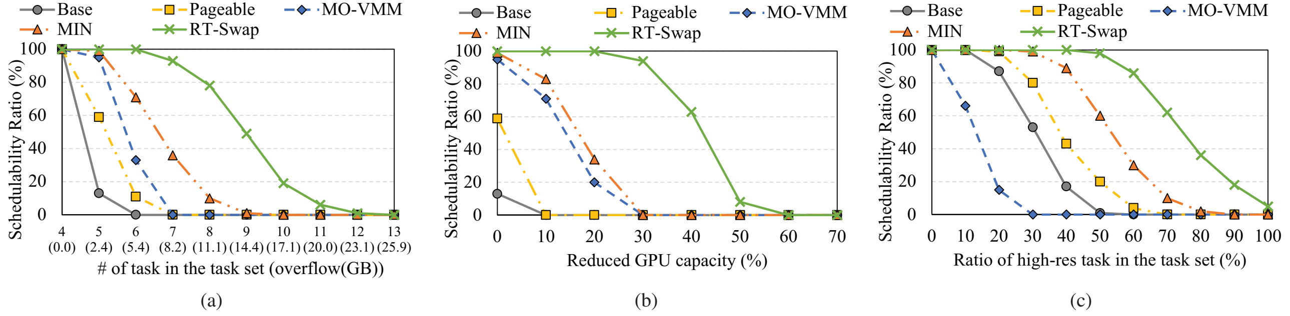

Fig. 7: Schedulability ratio comparison according to (a) the number of tasks in the task set, (b) reduced GPU capacity (%), and (c) ratio between low-resolution task and high-resolution task

TABLE III: DNN workload memory footprints and WCETs

	ResNet	YOLOv3	DenseNet	ResNext
Resolution	256, 416, 608			
GPU memory (GB)	1.4, 2.4, 3.8	2.3, 3.6, 6.2	1.0, 1.6, 2.7	1.6, 3.2, 5.7
Swappable (GB)	0.7, 1.4, 2.7	1.1, 2.6, 5.2	0.4, 0.9, 1.8	1.0, 2.2, 4.5
WCET (ms)	29, 36, 48	48, 51, 63	31, 34, 42	41, 45, 63

management, optimizing memory usage by estimating memory needs through an initial run, then allocating memory in large blocks. This approach minimizes runtime allocations and de-allocations, enhancing performance. Despite PyTorch's unique approach to memory management, it still relies on CUDA runtime APIs for its memory allocation processes. Thus, the implementation approach for integrating RT-Swap with PyTorch mirrors the methodology applied to Darknet. RT-Swap is compatible with ML frameworks that support CUDA 10.2 or higher. This compatibility depends on the availability of CUDA low-level GPU VMM APIs. Moreover, RT-Swap does not require DNN model changes, making it versatile for different DNN models.

VII. Evaluation

We demonstrate the capability of RT-Swap in optimizing GPU memory usage, utilizing CPU memory for swapping, while guaranteeing timely inference for multiple DNN tasks.

A. Experimental Setup

We evaluate RT-Swap on top of Darknet [13][3]. Our experiments are conducted on a server equipped with an NVIDIA RTX3090 GPU with 24GB of GPU memory, 128GB of CPU memory (32GB×4 DDR4 @ 2666MHz), and PCIe 3.0, operating on Ubuntu 18.04 with CUDA 12.1. We use four popular DNN models: YOLOv3 [21], ResNet [22], ResNext [23], and DenseNet [24], each with three input resolutions (256, 416, 608), as detailed in Table III. To profile each model, we utilize the NVIDIA nvprof tool, determining the total memory footprint while employing RT-Swap Library to obtain the amount of swappable memory. Computation time is measured by running each model 1000 times, taking the maximum value as the WCET. Swap-in/out times are estimated using linear regression, given their proportionality to the swap volume with a specified swap chunk size.

We compare four different approaches against RT-Swap:

- Base: on-demand paging with LRU replacement policy [28];
- MO-VMM: memory-object-level VMM;
- MIN: RT-Swap with the minimum swap chunk size (2MB);
- Pageable: RT-Swap with pageable CPU memory.

Base utilizes CUDA-supported unified memory with on-demand paging, employing the LRU replacement policy upon a page fault. Since on-demand paging cannot determine the swap amount offline, DNN executions are emulated using the same replacement policy and non-preemptive EDF scheduling to quantify individual DNN tasks' swap amounts for comparison. MO-VMM adopts the default memory-object-level virtual memory management method provided by CUDA with host-pinned memory. MIN and Pageable are variations of RT-Swap. MIN assigns the swap chunk size to a minimal value of 2MB, and Pageable utilizes pageable CPU memory as a swapping device, instead of using host-pinned memory. In comparison of the schedulability ratio, we use our proposed analysis in Eq. (3) for all approaches.

B. Extensive Simulations

With an increasing number of tasks. We generate a synthetic DNN task set τ as follows. For each task $\tau_i \in \tau$, its DNN model and resolution are randomly chosen from the four DNN models and three resolutions, respectively, as shown in Table III, and T_i is uniformly chosen in [3000, 5000]. Initially, 100 DNN task sets are created, each with 4 tasks and a total footprint of 23-24GB.[4] Additional tasks are incrementally added to each set, creating 100 new task sets each time, until reaching 13 tasks per set.

Fig. 7(a) compares the percentage of schedulable task sets by five approaches, varying the number of tasks from 4 to 13. RT-Swap consistently outperforms other approaches, rendering 72%, 139%, 221%, and 383% more task sets schedulable compared to MIN, MO-VMM, Pageable, and Base, respectively. This performance gap widens as the number of tasks (denoted as n) increases, coinciding with a growth in memory overflow from 0 to 25.9GB, causing more memory to be swapped. When $n \geq 6$, Base hardly finds any schedulable task sets due to its LRU replacement policy, which neglects schedulability, causing unpredictable page evictions. Similarly, Pageable struggles at $n \geq 7$, due to significant swapping overheads

[3] We choose Darknet for our RT-Swap experiment due to its straightforward structure, which facilitates easier comparison with existing approaches.

[4] We cap the footprint to ensure that adding any new task surpasses the 24GB GPU capacity, focusing on task sets exceeding GPU memory limits.

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on July 12,2024 at 03:45:24 UTC from IEEE Xplore. Restrictions apply.

TABLE IV: Task set information for a case study

Task	Deadline (ms)	DNN model	Max. swap volume (MB)
τ_1	600	DenseNet_416	0
τ_2	600	ResNet_256	0
τ_3	900	ResNext_608	576
τ_4	900	ResNext_608	576
τ_5	1200	ResNext_608	576
τ_6	1200	ResNext_608	576

from data transfers to pageable CPU memory. MO-VMM faces notable fragmentation issues, failing to optimize available GPU memory, leading to unschedulable task sets. MIN suffers from an exhaustive number of physical memory chunks, causing excessive swapping overheads and reduced schedulability. In contrast, RT-Swap optimizes the swap chunk size and swap volumes based on the proposed schedulability analysis, and employs efficient swapping mechanisms to minimize fragmentation overheads. Consequently, RT-Swap outperforms in scheduling more task sets, even with up to 12 tasks and an average overflow of 23.1GB (96.2% more memory).

With a reduced GPU capacity. We assess RT-Swap's memory efficiency by varying the GPU's physical capacity. Using the same 100 task sets from the previous simulation, each with 5 tasks, we reduce the original 24GB GPU capacity incrementally from 0% to 70%, as depicted in Fig. 7(b). As GPU memory capacity diminishes, swap amounts naturally increase. RT-Swap, however, manages to maintain the schedulability of task sets that were schedulable at the original GPU capacity, enduring an average GPU capacity reduction of 36.5%. In contrast, other approaches only sustain schedulability with an average capacity reduction of 5.2%. This enhancement indicates RT-Swap's efficient utilization of GPU memory, minimizing fragmentation and swapping overheads, thereby allowing for a more number of schedulable task sets even with limited GPU memory.

With an increasing resolution. We evaluate the effectiveness of RT-Swap in accommodating larger DNN models by varying the high-resolution task ratios. Among three resolutions of 256, 416, and 608, tasks utilizing 256 are considered as low-resolution tasks, while those utilizing 608 are deemed as high-resolution tasks. Starting with 100 task sets of ten low-resolution tasks each, we incrementally convert one random low-resolution task to high-resolution, generating 100 new task sets each time, until all tasks are high-resolution. For the four DNN models, high-resolution tasks require 2.9x more memory on average as compared to their low-resolution counterparts. In Fig. 7(c), RT-Swap successfully enables high-resolution inference in an average of 70.5% of tasks within the generated task sets, outperforming other approaches which only manage this in 29.4% of tasks. Consequently, RT-Swap significantly enhances the quality of inference results, effectively leveraging the available limited GPU memory capacity.

C. Runtime Experiments

To analyze the performance of RT-Swap in terms of inference latency and response time, we conduct a case study with six DNN tasks, detailed in Table IV. The cumulative memory footprint of these tasks is 25.6GB, exceeding the GPU's capacity by 1.6GB. The tasks τ_1 and τ_2 have deadlines of 600ms; τ_3 and τ_4 have deadlines of 900ms; and τ_5 and τ_6 have deadlines of 1200ms. τ_1 and τ_2 use DenseNet and ResNet,

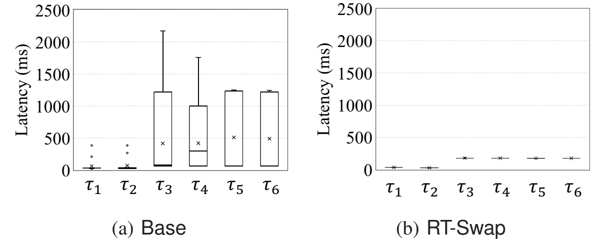

Fig. 8: Observed latency distribution using Base and RT-Swap

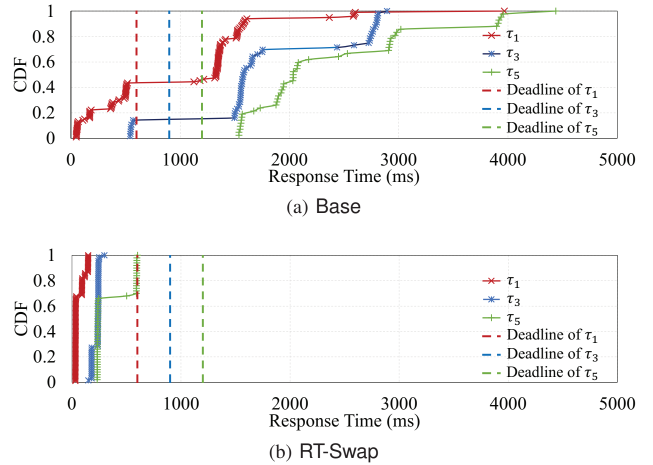

Fig. 9: Response Time CDF using Base and RT-Swap

respectively, whereas the remaining tasks use ResNext. The tasks are scheduled under EDF. Our proposed swap volume assignment algorithm ensures that the entire model parameters of τ_1 and τ_2, with shorter deadlines, reside in the GPU memory without swapping. Meanwhile, the other four tasks are each allocated 576MB of swap volumes to manage the excess GPU memory usage. RT-Swap sets the swap chunk size to 32MB.

Fig. 8 shows the average inference latency of six tasks using Base and RT-Swap (marked as a cross), and the distribution of observed latency (marked as a candlestick) depicting max/min values and 25 to 75 percentiles range as boxes. The latency includes the page fault handling time and the swapping time for Base and RT-Swap, respectively, in addition to the computation time. Note that outliers are excluded from the distribution for clarity and are instead plotted as dots. In the figure, we observe that Base adds high fluctuations in latency, on the order of seconds (up to 2.2secs). Under Base, although τ_1 with a shorter period exhibits fewer pages being evicted on average due to the LRU policy, its worst-case latency is 383ms, higher than the average value of 59ms by 6.5x. τ_3 with a longer period faces much higher and unpredictable page fault handling overhead, yielding a substantial increase in latency (up to 2172ms). On the other hand, RT-Swap shows consistent latency by limiting the maximum swap volume for each task. The worst-case latency for τ_1 is 33ms, and for τ_3 it is 184ms, imposing only a minor swapping overhead of up to 121ms.

To demonstrate RT-Swap's effectiveness in ensuring schedulability, Fig. 9 shows the measured end-to-end response time CDF of three tasks in the task set by Base and RT-Swap,

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on July 12,2024 at 03:45:24 UTC from IEEE Xplore. Restrictions apply.

displaying only three out of the six tasks as those with the same deadlines show similar results. Under Base, all tasks miss their deadline and show long tail latency with the maximum observed response time of 4438ms. On the other hand, RT-Swap achieves 14x reduction in the maximum observed response time on average over Base, making all tasks schedulable.

D. Runtime Overhead Analysis

We analyze the runtime overhead of RT-Swap for initialization, wrapper function calls, communications between RT-Swap Library and RT-Swap Scheduler, as well as memory usage and fragmentation, as detailed in Table V. These measurements are based on the task set used in a case study. To initialize RT-Swap, RT-Swap Scheduler sets up IPC channels and VMM configurations on each RT-Swap Library, resulting in an average delay of 120.5μs. Wrapper function calls in RT-Swap Library occur at the initialization stage, averaging at 3.5μs per memory allocation request. RT-Swap imposes scheduling overhead composed of IPC communication and swapping decision costs, averaging 11.6μs and 11.1μs (up to 12.1μs and 43.6μs, respectively). This delay is minor compared to DNN computation times and can be accommodated in the schedulability analysis. Additionally, RT-Swap needs extra CPU memory (up to 15.7MB) to manage mapping information of each task's CPU VA to GPU VA, along with swap and computation queues. However, it utilizes the abundant CPU memory available. RT-Swap shows an average increase of 6.3MB (up to 9.5MB) in GPU memory usage due to internal fragmentation.

TABLE V: Runtime overhead analysis for RT-Swap

Runtime overhead	Average	Min	Max
Initialization (one time) (μs)	120.5	80.1	194.2
Wrapper function calls (μs)	3.5	3.0	8.0
Scheduling (μs)	22.7	12.8	55.6
Extra CPU memory usage (MB)	14.9	14	15.7
Fragmentation (MB)	6.3	0	9.5

VIII. Related Work

GPU Memory Management for a Single Kernel. To process DNNs in real-time, GPU has become a primary computing device that works as a co-processor along with CPU and CPU memory. Since GPU memory is dedicated to the GPU device and managed directly by the GPU driver rather than the OS, there have been attempts to manage GPU memory on top of the GPU driver. Studies [29]–[32] have suggested GPU memory management by intercepting GPU library APIs when sending commands to the GPU driver. RSVM [29], for example, provides library-based memory management and allows transparent access to memory between the GPU and CPU using region-based allocation. VAST [30] utilizes a virtual address space for GPU memory based on OpenCL, dividing large memory tasks into smaller sub-kernels that fit within the available memory. RT-Swap leverages mechanisms for GPU memory management by intercepting GPU APIs to focus timely inference of *multiple* DNN tasks rather than a single task or kernel in existing approaches.

Driver-level GPU Paging. Various studies [7]–[10] attempt to overcome the limited GPU memory by providing a paging mechanism between GPU and CPU memory. Gdev [7] allows GPU contexts to allocate memory beyond the physical GPU memory size by swapping between CPU and GPU memory at the memory object granularity. GPUSwap [8] enables GPU memory oversubscription via direct access to CPU memory by ensuring the GPU always has access to the data. A recent study [9] provides GPU memory oversubscription by utilizing direct I/O to SSD on an integrated GPU. RT-Swap shares this vision to overcome the GPU memory wall in practice but there are several major differences that make RT-Swap more transparent, deployable, and timely for DNN tasks. First, they rely on proprietary GPU driver requiring modifications to the device driver or installing custom driver; in contrast, RT-Swap can be deployed on any NVIDIA GPU, either integrated or discrete, on top of the GPU device driver without modification by the proposed transparent GPU virtual memory management. Moreover, they do not deal with the overheads induced by swap operations falling short in ensuring timely inference for *multiple* DNN tasks.

Framework-level DNN Memory Management. Instead of a driver-level, DNN framework-level solution [16]–[18], [20] aims to efficiently map DNN layers across GPU and CPU memory. vDNN [16] aims to release or transfer feature maps generated between layers to CPU memory to address the large GPU memory footprint during the training step. Capuchin [17], unlike static approaches, offers a dynamic approach to select memory allocations and transfer them to CPU memory. Swapadvisor [18] reduces memory management overhead by overlapping layer execution time and memory transfer time through layer scheduling and determining which tensor to transfer. Sentinel [20] selects tensors to transfer based on their hotness, i.e., how frequently they are revisited. Demand layering [33] minimizes memory usage by loading and executing layers in a layer-by-layer manner. Despite these efforts, these framework-level solutions focus on a single DNN training/inference step rather than multiple DNN inferences. RT-Swap, on the contrary, provides transparent, efficient, and scalable system-wide memory management for multiple DNN frameworks running at the same time for real-time inferences.

IX. Conclusion

This paper presents a new real-time memory management framework, specifically designed to address the GPU memory bottleneck problem, while concurrently ensuring timely inference for multiple DNN tasks. We introduce transparent and efficient memory swapping and virtual memory management mechanisms with negligible fragmentation overhead, thereby expanding the available GPU memory. Moreover, the proposed swap-aware real-time scheduling algorithm guarantees the timely execution of DNN tasks and mitigates the overhead of memory swapping. Extensive evaluation and implementation on a popular ML framework prove the effectiveness of our approach in accommodating more real-time DNN tasks that demand more memory than physically available on the GPU, while preserving timing constraints.

RT-Swap is suitable for systems with distinct CPU and GPU memories, like NVIDIA DRIVE [34]. In integrated CPU-GPU SoCs, like NVIDIA Jetson [35], with shared memory, RT-Swap could enable transparent swapping between shared memory and secondary storage (e.g., SSD and ZRAM), a potential area for future exploration. Additionally, future enhancements may involve extending RT-Swap for layer-level GPU memory management to further boost memory efficiency.

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on July 12,2024 at 03:45:24 UTC from IEEE Xplore. Restrictions apply.

ACKNOWLEDGEMENT

This work was supported in part by (1) the National Research Foundation of Korea (NRF) grant (2018R1A5A1060031 (ERC), 2021R1F1A1063785, 2022R1A4A3018824, RS-2023-00248143, RS-2023-00213309) and (2) Institute of Information & communications Technology Planning & Evaluation (IITP) grant (IITP-2022-0-01053, RS-2022-00155885, IITP-2024-RS-2022-00156360) funded by the Korea government (MSIT).

REFERENCES

[1] H.-Y. M. L. Chien-Yao Wang, Alexey Bochkovskiy, "Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," 2022.

[2] Q. Zhao, T. Sheng, Y. Wang, Z. Tang, Y. Chen, L. Cai, and H. Ling, "M2det: A single-shot object detector based on multi-level feature pyramid network," in *AAAI*, 2019.

[3] Z. Zhou, M. M. R. Siddiquee, N. Tajbakhsh, and J. Liang, "Unet++: A nested u-net architecture for medical image segmentation," in *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support*, 2018.

[4] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "Superneurons: Dynamic gpu memory management for training deep neural networks," in *PPoPP*, 2018.

[5] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *SC*, 2020.

[6] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "Zero-offload: Democratizing billion-scale model training," in *ATC*, 2021.

[7] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt, "Gdev: First-class gpu resource management in the operating system." in *ATC*, 2012.

[8] J. Kehne, J. Metter, and F. Bellosa, "Gpuswap: Enabling oversubscription of gpu memory through transparent swapping," in *VEE*, 2015.

[9] J. Bakita and J. H. Anderson, "Enabling gpu memory oversubscription via transparent paging to an nvme ssd," in *RTSS*, 2022.

[10] P. Yu and M. Chowdhury, "Fine-grained gpu sharing primitives for deep learning applications," *MLSys*, 2020.

[11] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *NIPS*, 2019.

[12] M. A. et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://www.tensorflow.org/

[13] J. Redmon, "Darknet: Open source neural networks in c," 2013–2016. [Online]. Available: http://pjreddie.com/darknet/

[14] A. A. Awan, C.-H. Chu, H. Subramoni, X. Lu, and D. K. Panda, "OC-DNN: Exploiting advanced unified memory capabilities in CUDA 9 and Volta GPUs for out-of-core DNN training," in *HiPC*, 2018.

[15] J. Jung, J. Kim, and J. Lee, "Deepum: Tensor migration and prefetching in unified memory," in *ASPLOS*, 2023.

[16] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *MICRO*, 2016.

[17] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian, "Capuchin: Tensor-based GPU memory management for deep learning," in *ASPLOS*, 2020.

[18] C.-C. Huang, G. Jin, and J. Li, "Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping," in *ASPLOS*, 2020.

[19] M. Hildebrand, J. Khan, S. Trika, J. Lowe-Power, and V. Akella, "Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming," in *ASPLOS*, 2020.

[20] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning," in *HPCA*, 2021.

[21] A. Farhadi and J. Redmon, "Yolov3: An incremental improvement," in *CVPR*, 2018.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.

[23] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *CVPR*, 2017.

[24] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *CVPR*, 2017.

[25] T. Baker, "A stack-based resource allocation policy for realtime processes," in *RTSS*, 1990.

[26] W. Kang, S. Chung, J. Y. Kim, Y. Lee, K. Lee, J. Lee, K. G. Shin, and H. S. Chwa, "DNN-SAM: Split-and-merge dnn execution for real-time object detection," in *RTAS*, 2022.

[27] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2023. [Online]. Available: https://www.gurobi.com

[28] Nvidia, "Understanding on demand paging (odp)," 2022. [Online]. Available: https://enterprise-support.nvidia.com/s/article/understanding-on-demand-paging--odp-x

[29] F. Ji, H. Lin, and X. Ma, "RSVM: a region-based software virtual memory for GPU," in *PACT*, 2013.

[30] J. Lee, M. Samadi, and S. Mahlke, "VAST: The illusion of a large memory space for GPUs," in *PACT*, 2014.

[31] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: a GPU memory manager with application-transparent support for multiple page sizes," in *MICRO*, 2017.

[32] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, "Batch-aware unified memory management in GPUs for irregular workloads," in *ASPLOS*, 2020.

[33] M. Ji, S. Yi, C. Koo, S. Ahn, D. Seo, N. Dutt, and J.-C. Kim, "Demand layering for real-time dnn inference with minimized memory," in *RTSS*, 2022.

[34] NVIDIA DRIVE AGX Orin DevKit. [Online]. Available: https://developer.nvidia.com/drive/hyperion

[35] NVIDIA Jetson DevKit. [Online]. Available: https://developer.nvidia.com/embedded/jetson-developer-kits

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on July 12,2024 at 03:45:24 UTC from IEEE Xplore. Restrictions apply.