ZeroSwap: Minimizing Swap Overhead for Real-Time Multi-DNN Inference via SSD-based GPU Memory Extension

2026 IEEE 32nd Real-Time and Embedded Technology and Applications Symposium (RTAS) | 979-8-3315-8070-4/26/$31.00 ©2026 IEEE | DOI: 10.1109/RTAS68450.2026.00030

Woosung Kang[1], Filippo Muzzini[2], Gianluca Brilli[2], Jong-Chan Kim[3], Jinkyu Lee[4], Hoon Sung Chwa[1*]

[1]*Daegu Gyeongbuk Institute of Science and Technology (DGIST)*, Republic of Korea
[2]*University of Modena and Reggio Emilia*, Italy
[3]*Kookmin University*, Republic of Korea
[4]*Yonsei University*, Republic of Korea

***Abstract*—The feasibility of real-time scheduling with multiple deep neural networks (DNNs) is constrained by the limited GPU memory. One solution is extending the GPU memory by leveraging the CPU memory as a swap device. However, in integrated GPU systems, where the CPU and GPU share a unified physical memory, such approaches are deceptive. Instead, SSDs can be used as a swap device. This SSD-based swapping, however, introduces substantial overhead due to PCIe transfers between RAM and SSD and additional memory management complexity. To overcome these challenges, we present ZeroSwap, a memory swapping framework that minimizes the swap overhead for real-time multi-DNN inference on embedded systems with integrated GPUs. ZeroSwap introduces three key mechanisms: (i) semantic-aware selective swapping that swaps only inference-critical data to minimize swap volume, (ii) shared pinned allocation that eliminates runtime allocation overhead through physical memory sharing, and (iii) segment-level overlapping that hides PCIe transfer latency by overlapping swap operations with computation. We implement ZeroSwap on top of a state-of-the-art machine learning framework and demonstrate its effectiveness in making up to 101.7% more DNN task sets schedulable and achieving up to 3.2× faster response time compared to existing approaches.**

I. Introduction

Memory demand has emerged as a critical constraint for real-time DNN execution. This challenge has originated from the rapid advancement of deep neural networks (DNNs), which has led to increasingly complex and large-scale models designed to deliver higher accuracy and performance across diverse applications [1]–[3]. At the same time, there is a growing demand for on-device inference in resource-constrained environments such as embedded boards, where multiple DNNs often run concurrently [4]. This trend further exacerbates the memory issue, making it one of the most pressing bottlenecks in real-time execution.

To overcome this memory bottleneck, researchers have explored both model- and system-level approaches. Model compression techniques such as pruning [5], quantization [6], and knowledge distillation [7] can reduce memory demand, but often degrade accuracy. System-level approaches, in contrast, aim to expand GPU capacity without altering model structure, primarily by utilizing CPU memory as a swap device, and have thus gained attention for mitigating memory limits without compromising accuracy. Early studies [8]–[10] modify GPU drivers to offload data to CPU memory, while later framework-based methods [11]–[14] selectively transfer tensors based on layer dependency or runtime profiling to reduce peak memory usage. More recently, RT-Swap [15] introduces a runtime framework that enables predictable memory swapping between CPU and GPU memory for real-time multi-DNN inference. These approaches are effective on discrete GPUs by offloading GPU data to CPU memory, temporarily relieving GPU memory pressure and enabling workloads larger than device capacity.

However, in integrated GPU systems, which dominate embedded and edge platforms, such CPU-assisted swapping is infeasible. Since the CPU and GPU share a unified physical memory space, migrating data between them does not free additional capacity. A natural alternative is to leverage secondary storage, such as solid-state drives (SSDs), which now offer sufficient bandwidth and latency to support DNN workloads exceeding available RAM [16], [17]. As a result, SSD-based swapping has emerged as a practical approach to extend memory capacity in integrated GPU environments. A recent study [16] achieves GPU oversubscription through driver-level paging of GPU memory to SSD. Framework-level methods such as Demand Layering [17] and TASO [18] adopt layer-wise weight loading, overlapping I/O with computation to reduce peak memory usage. Hardware–software co-design efforts, including ZnG [19] and scheduling-aware prefetching [20], integrate GPUs with SSDs by exploiting flash storage as an extension of global GPU memory.

Although SSD-based swapping extends usable memory capacity, it incurs significant overheads that hinder timely execution. Our in-depth analysis on an integrated GPU system shows that swap-induced operations dominate end-to-end latency, up to 5.3× longer than GPU execution alone, accounting for 81% of total inference time. This overhead arises mainly from two physical domains: (i) data movement across the PCIe channel between SSD and memory, and (ii) internal operations within unified memory. However, a closer look reveals that this overhead is heavily fragmented across three distinct bottlenecks: storage I/O, memory management (e.g., allocations and releases), and redundant data copies. All three components are substantial and increase linearly with the amount of data swapped. This reveals a strict interdependency: optimizing a single stage merely shifts the bottleneck rather than resolving it, making partial mitigation insufficient for timely execution.

We aim to minimize swap overhead and ensure timely inference for DNN tasks. To this end, we propose ZeroSwap, a memory swapping framework specialized for real-time multi-

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

2642-7346/26/$31.00 ©2026 IEEE
DOI 10.1109/RTAS68450.2026.00030

DNN workloads on integrated GPU systems. ZeroSwap is the first to address this fragmented overhead by tightly integrating three key mechanisms into a unified framework. First, it performs *semantic-aware selective swapping*, exploiting the semantics of DNN data to drastically reduce PCIe data movement by swapping out and restoring only data essential for inference. Second, it employs *shared pinned allocation*, which eliminates runtime memory management overhead by enabling physical memory sharing across DNN tasks and thereby avoids the complex swapping pipeline otherwise required in integrated GPUs. Finally, it adopts *segment-level overlapping*, partitioning each DNN task into two segments and overlapping the data movement of the latter with the execution of the former, thereby hiding transfer latency behind computation. Tight integration of these three mechanisms is both novel and essential; the absence of any single component would leave the remaining overheads to dominate, inevitably leading to a schedulability breakdown.

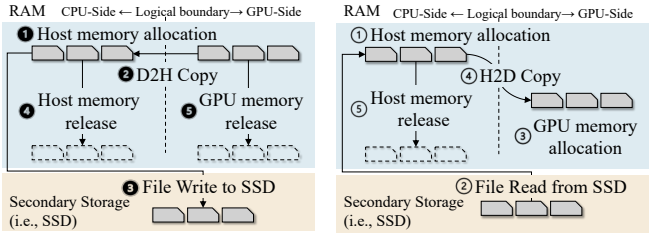
While these mechanisms form the foundation of ZeroSwap, achieving minimal swap overhead and timely inference further depends on how policies are coordinated, particularly for DNN partitioning and swap volume sharing. We therefore develop a segment-level overlap-aware scheduler that determines how each DNN should be partitioned to maximize overlap between data transfer and computation, and a shared swap volume assignment policy that allocates swap capacity across tasks to efficiently mitigate both sources of overhead while ensuring schedulability.

We implemented ZeroSwap[1] on PyTorch [21], one of the most widely used ML frameworks. ZeroSwap serves as a middle layer between the ML framework and the operating system's memory subsystem, transparently enabling swapping with minimal overhead and without requiring any modification to the GPU driver or the OS. Our evaluation employs four representative DNN models, on an integrated GPU platform, NVIDIA Jetson Orin Nano [22] with a SK hynix P41 SSD [23]. ZeroSwap achieves near-zero swap overhead, with only a 3.6% increase in latency even when total memory demand exceeds the GPU's physical capacity by 54.2%, without violating timing constraints. This enables up to 101.7% more real-time DNN task sets to be schedulable compared to existing approaches. Furthermore, through a case study on a four-camera object detection system, we demonstrate that ZeroSwap achieves a 3.8× higher frame rate and 3.2× faster response time compared to the state-of-the-art real-time swapping system [15].

This paper makes the following main contributions:

- We propose ZeroSwap, a memory swapping framework for integrated GPU systems that enables real-time multi-DNN execution beyond GPU capacity with near-zero swap overhead;
- We design three complementary techniques: semantic-aware selective swapping, shared pinned allocation, and segment-level overlapping, together reducing swap volume, management cost, and transfer latency;
- We develop an overlap-aware schedulability test and co-optimization algorithm that jointly determine swap volumes and segment boundaries to ensure system-wide timing guarantees; and
- We implement ZeroSwap on PyTorch and evaluate it using real DNN workloads on an integrated GPU platform, demonstrating over 2× improvement in schedulable real-time DNN task sets and reduced inference latency compared to existing approaches.

[1]ZeroSwap source code is publicly accessible at https://rtcl.dgist.ac.kr/index.php/zeroswap.



(a) Swap-out operation (b) Swap-in operation

Fig. 1: Swap-out/in procedure under integrated GPU systems

II. Background

A. Memory Swapping in Integrated GPU Systems

The limited physical memory of embedded systems, particularly those with integrated GPUs, makes it infeasible to keep all tasks resident simultaneously. To overcome this constraint, systems employ memory swapping, which relocates inactive memory regions to external storage and restores them when needed. A swap consists of two stages: (1) swap-out, where memory is evicted and stored externally, and (2) swap-in, where evicted data is reloaded for execution. This mechanism enables workloads that exceed physical capacity to proceed.

Unlike discrete GPUs, integrated GPUs share a unified physical memory space. Thus, swapping requires transferring data to and from SSDs rather than moving it between GPU and CPU memories. This process introduces an additional pipeline involving host buffers and file I/O. As illustrated in Fig. 1a, a swap-out involves: ❶ allocating a host buffer, ❷ copying data from GPU memory to the buffer, ❸ writing the buffer to the SSD via PCIe, ❹ releasing the buffer, and ❺ freeing the original GPU memory. Conversely, Fig. 1b shows a swap-in: ① allocating a host buffer, ② reading data from the SSD into the buffer, ③ allocating GPU memory, ④ copying the buffer contents to GPU memory, and ⑤ releasing the host buffer. We refer to this swapping procedure as the Vanilla throughout the paper.

TABLE I: CUDA Virtual Memory Management APIs

Function	Description
`cuMemCreate`	Create a physical memory handle
`cuMemAddressReserve`	Reserve a virtual address range
`cuMemMap`	Map physical memory to a VA range
`cuMemExportToShareableHandle`	Export memory as a shareable handle
`cuMemImportFromShareableHandle`	Import memory from a shareable handle

B. GPU Virtual Memory Management

To gain finer control over GPU memory, CUDA provides low-level virtual memory management (VMM) APIs that decouple virtual and physical memory. Unlike `cudaMalloc`,

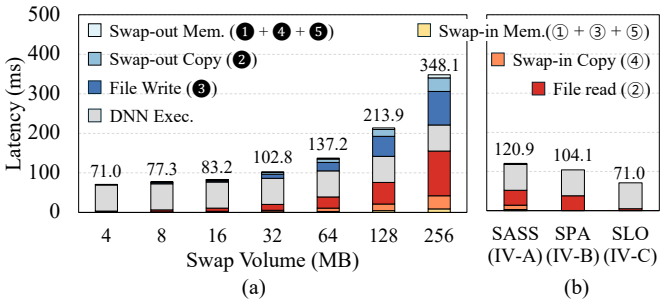


Fig. 2: Latency breakdown of YOLOv7 under swap overhead (a) Vanilla (4-256MB), (b) Optimized with SASS, SPA, and SLO at 256MB.

which abstracts placement through the driver, VMM primitives allow explicit allocation, reservation, and mapping of memory regions, as well as inter-process memory sharing via export/import handles as shown in Table I. These features form the basis of our design, enabling task-level memory sharing and efficient swapping across processes.

III. Swap Overhead Analysis

Memory swapping allows systems to manage workloads exceeding physical capacity, but it introduces significant latency from management and data movement overheads, hindering timely execution of DNN tasks. To quantify swap-induced delays on an integrated GPU, we conducted fine-grained latency measurements for each stage of the swap pipeline on a Jetson Orin Nano equipped with an SK Hynix P41 SSD [23].[2] As a representative workload, we executed YOLOv7 [3] at 640×640 resolution, which consumes 1224 MB of memory and runs in 66.1 ms. From this footprint, we varied the swap volume, the amount of data swapped in and out, from 4 MB to 256 MB and measured corresponding overheads.

Latencies for memory allocation and release (❶ + ❹ + ❺ in swap-out, ① + ③ + ⑤ in swap-in operations), CPU-GPU data copies (❷, ④), and RAM-SSD I/O operations (❸, ②) were measured separately for both swap-in and swap-out processes along with the DNN execution time. As shown in Fig. 2, under Vanilla which follows the swapping mechanism described in Sec. II-A, swap operations dominate end-to-end latency, reaching up to 348.1 ms at 256 MB, 5.3× longer than YOLOv7 execution, and accounting for up to 81% of total latency. At 256 MB swap volume, these three overhead sources—memory allocation and release, data copying, and storage I/O—account for 6%, 24%, and 70% of the total swap overhead, respectively, revealing a heavily fragmented structure in which no single source dominates.

Since all three sources contribute comparably to the total overhead, optimizing any single source alone is insufficient, as the remaining two still impose substantial delays. Moreover, all three sources grow noticeably with the swap volume, showing high sensitivity to data size. Consequently, system-level optimization that jointly addresses all three sources is required to guarantee timely inference. Based on these observations, two strategies are required to minimize swap overhead and satisfy the timing requirements of DNN tasks:

[2]Other representative SSDs exhibited similar trends.

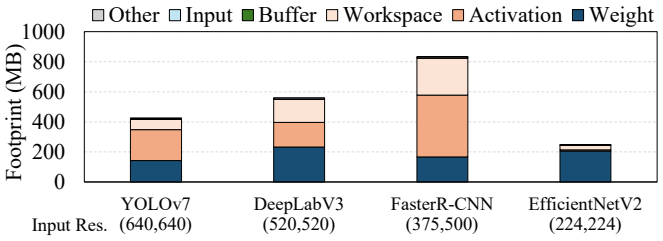


Fig. 3: Peak GPU memory breakdown across DNN workloads

S1. Reducing the amount of data swapped to lower overall overhead, particularly PCIe I/O latency; and
S2. Optimizing internal memory operations during swapping to eliminate allocation and data copying overhead.

IV. Design Principles of ZeroSwap

To minimize swap overhead and satisfy the timing requirements of DNN tasks, we propose three design principles aligned with two strategies. For **S1**, *semantic-aware selective swapping* reduces the swap volume by excluding non-essential data. For **S2**, *shared pinned allocation* removes redundant copies and repeated allocation overhead in the swap pipeline, while *segment-level overlapping* hides PCIe transfer latency by executing data transfers concurrently with DNN computation.

A. Semantic-Aware Selective Swapping

The goal of a swap system is to enable tasks whose memory demand exceeds physical capacity to execute while allowing suspended tasks to resume correctly. Conventional systems save and restore entire memory regions, regardless of necessity. This raises a fundamental question: must all data be preserved to resume DNN inference? DNN tasks exhibit a distinctive memory usage pattern due to their structured and sequential computation model. Each task processes an input through a fixed sequence of layers, applying pretrained parameters (i.e., weights) to generate intermediate results (i.e., activations) and temporary buffers (i.e., workspaces). Activations and workspaces are input-dependent and overwritten once a new input arrives, whereas weights remain constant across inputs. Therefore, when swap-in and swap-out operations are performed only at the input boundaries, temporary data such as activations and workspace buffers require neither swap-out nor swap-in. Their contents are valid only during the processing of a single input and are discarded once the inference for that input completes. Weights, in contrast, persist across all inferences and are stored on the SSD as read-only parameters. Since they never change during execution, saving them during swap-out is unnecessary; instead, they only need to be loaded once during swap-in to restore the model state. This distinction reveals a key insight: by recognizing the semantics of each data type, the system can substantially reduce swap volume. Building on this, the swap procedure is optimized to eliminate all data movement for activations and workspaces, while restoring only constant *weight* parameters during swap-in.

The benefit, however, depends on the footprint of weights. If weights already dominate, the potential reduction is limited. To quantify this, we analyze the memory footprint of representative DNN tasks, YOLOv7 [3], DeepLabV3 [24], Faster R-CNN [25], and EfficientNet [26]. As shown in Fig. 3,

weights account for 19.9–83.3% of the footprint, with an average share of 44.6%. Activations contribute 3.2–49.3%, while workspaces occupy 13.5–30.7%. These results highlight that by concentrating swap operations solely on weights, the swap volume can be reduced by nearly half on average. To estimate the expected benefit of semantic-aware selective swapping, we consider YOLOv7, whose weights constitute 33.7% of total memory footprint. As shown in Fig. 2, under SASS (Semantic-Aware Selective Swapping) with 256 MB swap volume, handling only weight data reduces the swap volume to 33.7% of the original, accelerates all stages of the swap pipeline, and omits swap-out copy (❷) and file-write step (❸). The effective swap volume decreases from 256 MB to 87 MB, lowering end-to-end latency from 348.1 ms to 120.9 ms and yielding a 65.3% reduction in overhead, which decreases its portion of the end-to-end latency from 81% to 45.3%.

Realizing this requires a semantic-aware selective swap procedure. First, the system must identify the exact locations of weights within each task's memory layout so that swap operations target only the necessary data. Second, the swap process must be redesigned to leverage this knowledge: during swap-out, only memory allocation and release are performed without saving contents, while during swap-in, the system restores weight segments and reconstructs the original layout. Finally, the memory regions selected as swap targets must be chosen carefully. Even with the same swap volume, the overhead of swap-in varies depending on how much of that volume consists of weights. Thus, selecting targets that minimize the fraction of weight data is essential to reducing swap overhead.

B. Shared Pinned Allocation

In integrated GPU systems, swap-out and swap-in operations follow a complicated pipeline of allocation, data copy, and release, introducing significant overhead. Each step exists for specific reasons. CPU–GPU copies are required because GPUs cannot directly access host memory; once data is read from the SSD into host memory, it must still be copied into GPU-accessible memory. Similarly, repeated allocation and release for the swap region arise because ML frameworks manage GPU memory independently, with each task operating in its own isolated space. This enforces a one-to-one mapping between a task's virtual memory and its physical allocation, thereby requiring repeated allocation and release whenever memory is reclaimed for swap operations.

Zero-copy access in CUDA optimizes this pipeline by allowing GPUs to directly access host-pinned memory allocated through APIs such as `cudaMallocHost`. Once allocated, this memory is transparently accessible by both CPU and GPU, eliminating explicit data copies between them. However, as shown in Fig. 4, host-pinned memory incurs substantially higher management costs, as it requires coordination between the OS and the GPU driver [27], [28]. Allocating 256 MB takes 26.8 ms, about 1.6× slower than pageable memory allocation and equivalent to 40.5% of YOLOv7's inference latency.

To further optimize the swap pipeline, allocation and release costs must also be minimized. This limitation stems from the one-to-one mapping that ties each task's virtual memory exclusively to its physical allocation. When memory is reclaimed from one task and reassigned to another, the system must first release the old mapping and then allocate a new one, making such operations unavoidable. This motivates a new memory management scheme that breaks framework-level isolation and enables memory sharing across DNN tasks. Whereas standard allocators within ML frameworks typically optimize intra-task reuse [29], [30], we tackle the distinct challenge of inter-task physical memory sharing. By supporting many-to-one mappings between task virtual memory and physical memory, the system allows tasks to reuse physical memory without repeated allocation and release. As a result, the swap pipeline becomes more efficient: CPU–GPU copies are eliminated through host-pinned memory, while allocation overhead is removed through physical memory sharing.

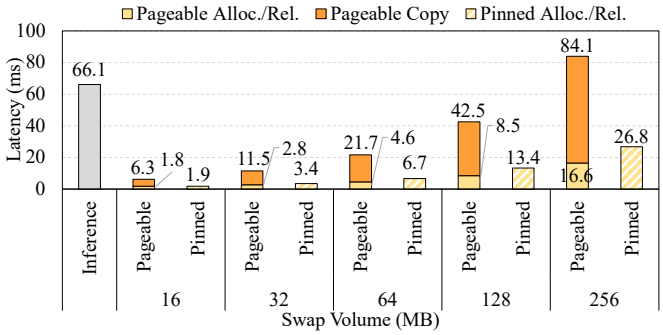


Fig. 4: Swap overhead of pageable and pinned memory

To evaluate how much the proposed shared pinned allocation can reduce swap overhead, we estimated the end-to-end latency of the swap pipeline after excluding the costs of CPU–GPU copies and memory allocation/release. SPA (Shared Pinned Allocation) in Fig. 2 shows this estimated latency. In combination with SASS, applying SPA further eliminates the swap-in copy (④) and memory-allocation (❶, ❹, ❺, ①, ③, and ⑤) steps of the conventional pipeline, reducing the end-to-end latency to 104.1 ms with a 256 MB swap volume and lowering the swap overhead to 36.5% of the total latency. This demonstrates that SPA successfully removes all in-RAM memory management overheads, making the swap pipeline more efficient and leaving only the file-read overhead from the SSD to restore weight data.[3]

C. Segment-Level Overlapping

In integrated GPU systems, PCIe transfers and DNN execution have opportunities to overlap since they are controlled by different processing units: PCIe I/O is managed by the CPU and SSD controller, whereas computation is executed on the GPU. This independence allows the two operations to proceed concurrently, enabling PCIe transfer latency to be hidden behind DNN execution time, reducing the overhead. To maximize overlap, it is essential to determine which computations proceed concurrently with which memory transfers. Since DNNs are organized as sequential layers, each layer depends only on its own weights. This property enables swap targets to be selected from later layers in the execution sequence, whose weights are not immediately required. Consequently, their weights can be prefetched in parallel with the execution of earlier layers, effectively hiding transfer latency. Although overlap

[3]With a 256 MB swap volume, SASS reduces the effective amount to 87 MB, making SPA's improvement appear smaller than it actually is.

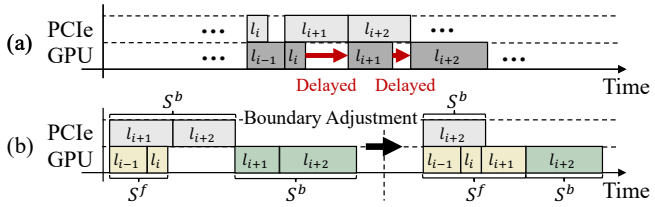


Fig. 5: (a) Layer-level and (b) segment-level overlapping

between two consecutive layers (e.g., fetching the weights of layer l_{i+1} while executing layer l_i) is also possible, such fine-grained overlap can suffer from delays whenever transfer time and computation time are imbalanced, as illustrated in Fig. 5. To avoid these stalls, we group the DNN model into front S^f and back S^b segments, overlapping the execution of the front segment with the transfer of the back segment weights of the same DNN model. Unlike consecutive-layer overlap, this segment-level approach provides flexibility in adjusting segment boundaries for better compute-I/O alignment, thus reducing imbalance-induced stalls.

To estimate the benefit of overlap, we consider the SPA case in Fig. 2, where YOLOv7 execution is divided evenly: the first half forms the front segment, and the second half the back segment. File reads for the back segment overlap with the execution of the front segment. The resulting latency is shown as SLO (Segment-Level Overlapping) in Fig. 2. For a 256 MB swap volume, 87.0% of the weight-read overhead is overlapped with DNN execution, reducing the end-to-end latency to 71.0 ms under SLO. This corresponds to a 31.8% and 79.6% reduction compared to SPA and Vanilla, respectively. SLO reduces the portion of swap overhead in the total latency to 6.9%, showing the effectiveness of segment-level overlapping.

Realizing this strategy requires carefully balancing the two segments. If the front segment is too large, the overlap window widens, but leaves little memory in the back segment available for swapping, making it difficult to accommodate the memory demands of all tasks. Conversely, if the back segment is too large, memory capacity is easily satisfied due to the larger swap volume, but the overlap window shortens. Finally, correctness requires that the back segment executes only after its weights have been fully restored, necessitating precise coordination between swap operations and DNN execution.

V. System Design

To realize the three design principles, we develop ZeroSwap, which is composed of three complementary components as shown in Fig. 6. Shared Pinned Allocator (SPA) unifies memory allocation across tasks by combining host-pinned memory with physical memory sharing, thereby eliminating redundant copies as well as repeated allocations. Since sharing physical memory across independent tasks introduces concurrency risks absent in single-task scenarios, SPA is co-designed with the scheduler to enforce mutually exclusive access to shared regions without runtime remapping overhead. Building on this foundation, Semantic-Aware Selective Swapper (SASS) identifies and restores only essential data (i.e., weights) while excluding non-critical data, thereby reducing the amount of data transferred over PCIe I/O. Beyond simple data filtering, SASS is co-designed with job-level non-preemptive scheduling to confine

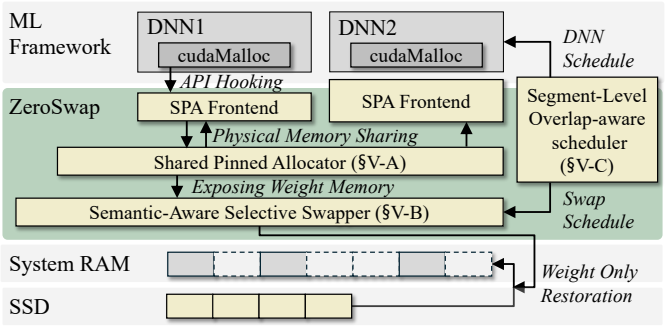


Fig. 6: Overview of ZeroSwap

the lifespan of input-dependent activations to a single job, allowing them to be safely discarded at job boundaries without incurring swap-out overhead. Moreover, SASS directly enables segment-level overlapping by prioritizing the restoration of back-segment weights, ensuring that the swap target aligns with the front-segment's execution window. Finally, Segment-Level Overlap-aware scheduler (SLO) coordinates swap operations with DNN execution, enabling data transfers to proceed in parallel with independent computation and thereby hiding I/O latency. Together, these tightly coupled components form a unified system that minimizes swap overhead while maintaining the timing requirements of DNN inference.

A. Shared Pinned Allocator

Shared Pinned Allocator (SPA) serves as the sole authority for GPU memory allocation and mapping, with all DNN tasks delegating memory management to it. By handling all GPU memory requests on behalf of DNN tasks, SPA enables physical memory regions to be shared across multiple tasks, thereby eliminating per-task runtime allocation and reducing swap overhead. Here, *sharing physical memory* means that a given physical memory region is mapped into the virtual address (VA) spaces of multiple DNN tasks, allowing all of them to access the same physical memory region. Achieving this requires precise control over which physical regions are mapped into each task's virtual memory space. To enable such control, ZeroSwap leverages CUDA Virtual Memory Management (VMM), which decouples virtual and physical memory in GPU allocations, allowing independent management. Using this VMM capability, SPA controls which physical memory regions are mapped into each task's virtual address space. For the VA ranges corresponding to swap regions, SPA maps them to the same physical memory regions across tasks, thereby enabling physical memory sharing. This design first eliminates the need to copy data between CPU and GPU memory by leveraging host-pinned allocation, and second removes the need to release or reallocate memory during swap-out or swap-in. Instead, swap operations are accomplished simply by overwriting data in the shared pinned memory. As a result, tasks can seamlessly resume inference without any additional runtime memory management. Access to shared memory is coordinated by Segment-Level Overlap-aware scheduler to avoid concurrency hazards, such as race conditions or data corruption, as detailed in Sec. V-C. For non-swap regions, however, SPA assigns distinct physical memory to each task, preventing data corruption and ensuring correct execution.

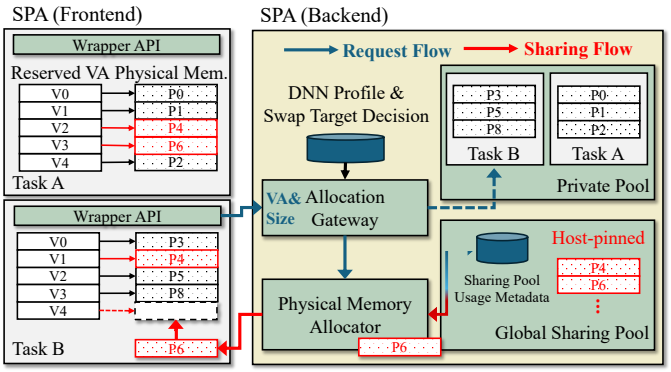


Fig. 7: Overview of Shared Pinned Allocator

To realize this design, SPA is composed of two coordinated components as shown in Fig. 7:

Frontend. The frontend of SPA is implemented as a preloaded shared library within the ML framework, where it controls each task's virtual address space and establishes mappings between the VA space and physical memory chunks. To achieve this, it hooks CUDA memory APIs (e.g., `cudaMalloc`, `cudaFree`, `cudaMemcpyAsync`) to intercept and redirect allocation requests issued by the ML framework.

At initialization, it reserves a single contiguous VA region via CUDA VMM, sized to the task's profiled memory demand. On allocation requests, it forwards the requested VA range and size to the backend of SPA, which returns physical memory chunks, either host-pinned memory for swap regions or standard GPU memory for non-swap regions. These chunks are mapped sequentially into the reserved VA region, and the corresponding virtual address is returned to the framework, preserving conventional semantics. Since memory is provisioned in coarse-grained units (i.e., 2MB)[4], memory requests that do not align with this granularity may leave parts of physical memory unused, causing fragmentation. To mitigate this, the frontend tracks both allocated ranges and mapped chunks so that subsequent allocations can proactively reuse the remaining space, thereby reducing fragmentation while preserving the expected contiguous layout. Importantly, once all physical memory has been mapped into each task's VA space at initialization, no further runtime memory management is required. The VA–physical mappings remain fixed throughout execution, so even when tasks undergo swapping.

Backend. The backend of SPA operates as an independent process that allocates physical memory through VMM and supplies it to DNN tasks upon allocation requests. It maintains two types of physical memory pools: per-task private pools for non-swap regions, isolated to each task, and a global sharing pool for swap regions, shared across all tasks.

Upon receiving a request, the backend determines, based on the virtual address range and profiling information, whether the allocation requires a private physical chunk or a shareable one. This classification follows the swap-target decision, which will be described in Sec. VI. For private chunks, the backend allocates standard GPU memory chunks through VMM, records them in the task's private pool, and returns the chunks to the frontend. These private chunks are dedicated exclusively to their owning task and are never shared with others.

For shareable chunks, the backend enables reuse of host-pinned physical memory across tasks. To this end, it maintains a global sharing pool that stores previously allocated host-pinned chunks. If the pool contains a chunk not yet used by the requesting task, that chunk is reassigned, allowing allocation to proceed without consuming additional physical memory. If all chunks in the pool are already in use by the task, a new host-pinned chunk is allocated, added to the pool, and subsequently made available for reuse by other tasks. When a task releases a memory region, it notifies SPA to stop using that space. If the physical memory is still in use by other tasks, the chunk is retained; the physical memory is released only after all tasks stop referencing the chunk.

Through the coordinated mapping of each task's VA space and physical memory by SPA frontend and backend, ZeroSwap establishes a design in which every task's virtual address space remains continuously backed by physical memory, even when the aggregate demand exceeds GPU capacity. Unlike conventional approaches, this mapping is never altered at runtime, which completely eliminates repeated allocation and release overhead. Furthermore, by allocating shared chunks as host-pinned memory, ZeroSwap avoids redundant CPU–GPU copies and enables direct PCIe transfers, ensuring that swap operations are carried out with minimal overhead.

B. *Semantic-Aware Selective Swapper*

Semantic-Aware Selective Swapper (SASS) reduces swap overhead by tailoring swap operations to essential data, leveraging the semantics of different data types. As discussed in Sec. IV-A, when accounting for the semantics of the DNN task, a swap-out operation requires only memory release, while a swap-in involves memory allocation and reading weight data from the SSD. Since SPA eliminates allocation and release in swap regions, SASS reduces swapping to restoring only weight data during swap-in, while no data is stored during swap-out.

Profiling. To achieve this, SASS must identify the VA ranges where weight data reside. Here, *weight* data refer not only to pretrained parameters but also to any immutable values, such as hyperparameters, that must be read as fixed constants. Since DNN tasks expect these data to reside at specific VA ranges determined at allocation time, the swapper must restore them to the same locations during swap-in to guarantee correct access. A key challenge is that weight data are often fragmented across the memory layout, appearing at layer granularity or even finer divisions. Thus, the system must know the exact positions of these weight segments to write them back correctly. Because ML frameworks assume a disjoint CPU/GPU memory model, such weight data are always transferred from CPU to GPU memory during initialization. In practice, weights are copied layer by layer using APIs such as `cudaMemcpyAsync`. By hooking these calls, our profiler records the destination GPU virtual addresses along with their sizes, thereby identifying the VA ranges corresponding to weights across layers. Restoring the weights at these same VA ranges during swap-in preserves the original layout expected by the DNN task, enabling seamless recovery without remapping or restructuring. The VA-range information

[4]The allocation granularity may vary across GPU architectures.

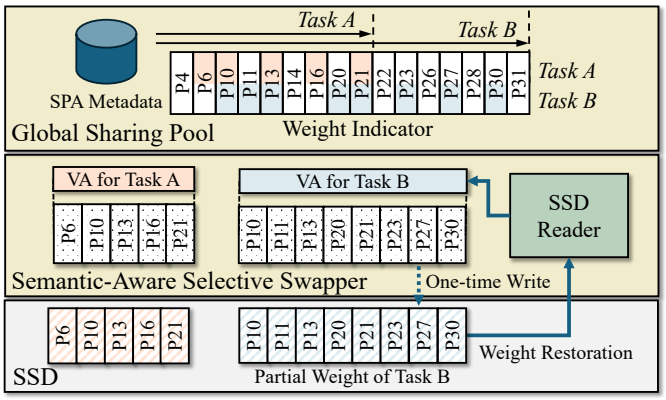


Fig. 8: Overview of Semantic-Aware Selective Swapper

is collected offline via API hooking and provided to the SPA backend as configuration data. At runtime, SPA passes this information to SASS, which uses it to restore the weights to their original locations.

Profiling alone, however, is insufficient. If the swapper restores weight segments individually for each scattered VA range, the system must maintain detailed metadata for every segment, significantly increasing complexity. Moreover, SSDs can only serve reads over continuous regions, meaning that fragmented weights would inflate the number of read operations and overhead. To overcome this inefficiency, SASS adopts a centralized, one-time weight read as shown in Fig. 8. The key observation is that writing weight data into a VA range is equivalent to writing it into the physical memory chunk mapped to that range. Since SPA manages all physical memory allocations, SASS queries it to retrieve the physical chunks corresponding to each task's weight ranges in the swap region based on the profiling. For every task, SASS allocates a new dedicated contiguous VA space for weight restoration and maps the relevant physical chunks into this space. This consolidation transforms scattered weights into a contiguous view from the swapper's perspective. During initialization, the collected weight data for each task are stored on the SSD in a contiguous layout, which can then be reloaded with a single read during swap-in. The mapped physical chunks ensure that, from the DNN's perspective, the original scattered layout is faithfully preserved without additional remapping.

Some physical memory chunks may be shared across multiple tasks' weight ranges. In such cases, the chunks are mapped into multiple restoration allocations, but at runtime they hold only the weights of the task that most recently completed a swap-in. Access to these regions is strictly governed by the scheduler, ensuring that no concurrency hazards arise. Details are discussed in Sec. V-C.

C. Segment-Level Overlap-aware scheduler

Segment-Level Overlap-aware scheduler coordinates DNN execution with selective weight restoration to ensure that transfers are properly overlapped with computation while preventing potential concurrency hazards. To this end, the scheduler communicates with both the ML framework and SASS through Inter-Process Communication (IPC) channels. Inference jobs are received from the framework and queued by priority to meet task deadlines. When a job is dispatched, the scheduler triggers SASS to restore its weights in the background, overlapping data transfer with the execution of the early segment.

Two classes of hazards can arise in this environment. *Inter-task hazards* occur if multiple DNN tasks attempt to access shared memory concurrently, potentially leading to race conditions. *Intra-task hazards* occur when a task begins executing a layer whose weights have not yet been fully restored, risking access to stale data from another task's weights or temporary state. To eliminate inter-task hazards, the scheduler enforces job-level exclusivity so that only one task executes at a time, ensuring serialized access to shared memory without the latency of runtime VMM remapping. Crucially, this non-preemptive execution intrinsically bounds the lifespan of temporary data to a single job, providing the fundamental safety guarantee for SASS to discard activations without swap-out. To eliminate intra-task hazards, when the execution of the DNN task reaches the start of the back segment, the framework queries the scheduler for permission to proceed, which is granted only after the corresponding weight transfer has completed. By orchestrating execution and weight restoration in this manner, the scheduler guarantees exclusive memory access across tasks, enforces correct intra-task synchronization, and ensures that weight transfers are effectively overlapped with computation.

VI. Segment-Level Overlapping Decision

Through ZeroSwap, swapping is performed selectively based on data semantics, eliminating the need for runtime memory management while allowing swap operations to overlap with computation and effectively hide their latency. However, minimizing swap overhead requires more than an efficient swapping mechanism—it also depends on how each DNN is partitioned to define its segment boundaries, how sharing targets are determined, and how the sharing volume is determined. The degree of achievable overlap is dictated by how the execution is segmented, and the sharing volume must be chosen with explicit consideration of this overlap; otherwise, excessive data may remain uncovered, leading to unhidden latency. Moreover, even with the same sharing volume, the composition of shared data critically affects performance: since ZeroSwap performs semantic-aware selective swapping, the proportion of inference-critical weight data within the shared region directly determines the actual amount of swap traffic. These design choices must also satisfy two key requirements:

R1. The sharing volume must be sufficiently large so that the total memory footprint of all tasks fits within the available physical memory.
R2. The swap overhead must be small enough for each task to meet its timing constraints, which requires maximizing the overlap between I/O and GPU computation.

To jointly satisfy these requirements, ZeroSwap relies on a three-stage hierarchical optimization in which each stage builds upon the outcome of the previous one: (i) model the end-to-end execution time under overlapping to quantify the swap-induced delay (Sec. VI-B), (ii) determine the segment boundaries and sharing targets that maximize overlap and minimize this delay (Sec. VI-C), and (iii) decide the sharing volume, based on (ii), that satisfies both R1 and R2 (Sec. VI-D).

A. System Model

We consider an integrated GPU with unified CPU–GPU memory, where m^D denotes the available capacity, supported by a secondary storage device. Since secondary storage can be easily expanded, we assume it always provides sufficient capacity for offloading data.

Task Model. We model real-time DNN inference tasks using the periodic task model, a standard abstraction in real-time systems. Each task corresponds to a DNN model and executes one inference per job instance. A task $\tau_i \in \tau$ is defined as (C_i, T_i, D_i, M_i), where C_i is the worst-case execution time (WCET) excluding swap operations, T_i is the period, D_i is the relative deadline (assumed equal to T_i), and M_i denotes the task's memory profile. Tasks are scheduled under the Earliest Deadline First (EDF) policy. We assume job-level non-preemptive scheduling for two reasons. First, it ensures that only one task accesses GPU memory at a time, eliminating inter-task memory hazards. Second, preemption is allowed only at job boundaries, restricting swap operations to occur exclusively at these points. This guarantees that input-dependent data, such as activations and workspaces, naturally expire after each job, allowing the system to avoid saving and restoring them across jobs, thereby reducing swap overhead.

Memory Model. The memory profile M_i represents the system memory usage of task τ_i, including both CPU and GPU allocations in the unified DRAM of the integrated GPU system. Formally, $M_i = (m_i, m_i^S, m_i^{wt}, m_i^{tmp}, X_i, x_i)$. Here, m_i denotes the total memory footprint, while m_i^S refers to the shareable portion of task τ_i. We define m_i^S as the amount of dynamically allocated GPU memory, identified by intercepting CUDA runtime allocation APIs (e.g., `cudaMalloc`). Within m_i^S, m_i^{wt} represents memory used for model weights, and m_i^{tmp} represents input-dependent temporary data such as activations and workspaces. The remainder of m_i consists of non-shareable regions, such as static buffers, shared library allocations, and host memory usage. The sharing volume X_i denotes the portion of m_i^S that is designated for sharing across tasks. Within X_i, the effective read volume $x_i \subseteq X_i$ represents the fraction corresponding to weight data that must be restored during the swap-in phase.

B. Segment-Level Overlapping Model

We model the execution of a DNN task τ_i as being divided into two consecutive segments, as shown in Fig. 9: the front segment S_i^f and the back segment S_i^b. The partitioning between the two segments depends on the selection of x_i, where layers requiring the weights in x_i form S_i^b, and all preceding layers form S_i^f, which will be discussed later in more detail. l_i denotes the first layer of S_i^b, serving as a synchronization point at which execution must stall until the corresponding weight data have been restored. The execution costs of the two segments are denoted $C_{i,f}$ and $C_{i,b}$, respectively, such that $C_i = C_{i,f} + C_{i,b}$.

The restoration of weights x_i requires $R(x_i)$ time in the worst case, which we estimate offline through profiling and regression as a function of the data size. When a job of τ_i is released, the scheduler immediately triggers the swapper to restore x_i in the background, thereby overlapping the I/O transfer with the computation of S_i^f. While the read operation

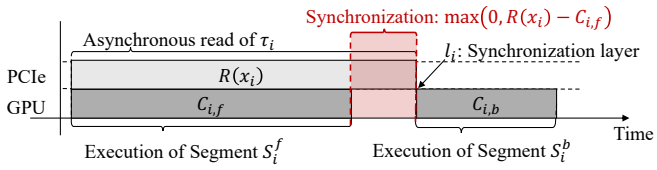


Fig. 9: Schedules for read operation and task execution

proceeds for $R(x_i)$ time, the task simultaneously executes its front segment for $C_{i,f}$ time units until reaching l_i. Note that S_i^f requires no weight reads, as its weights always reside in memory and x_i that need to be read belong to the back segment S_i^b. If the read is not yet complete, the task must stall at this synchronization point before continuing with S_i^b, while preemption is not permitted.

Under this model, the execution time of τ_i is given by

$$C_i^{exec} = \max(C_{i,f}, R(x_i)) + C_{i,b}. \quad (1)$$

and the additional swap-induced delay at the synchronization point can be quantified as $\max(0, R(x_i) - C_{i,f})$. Thus, overhead occurs only if the weight restoration extends beyond the computation of S_i^f. To minimize such delay, two conditions are essential. (i) The effective read volume x_i, and thus its read time $R(x_i)$, must be small enough to fit within the available window. (ii) The front segment S_i^f must provide a sufficiently long execution window, meaning $C_{i,f}$ should be enlarged by placing the synchronization point l_i as deep as possible in the execution. When both conditions are satisfied, swap operations are fully hidden behind computation, achieving effective I/O-computation overlap.

C. Segment Boundary Decision

Given the overlap conditions, ZeroSwap determines the boundary between the front and back segments by constructing sharing targets that simultaneously (i) minimize the effective read volume x_i and (ii) defer the synchronization point l_i for a given sharing volume X_i.

To minimize x_i, the system first includes memory chunks associated with temporary, input-dependent data, such as activations and workspaces. Although these chunks are designated as part of the sharing target, they require no restoration because their contents are invalidated after each inference. This enables X_i to be partially or even fully satisfied without introducing any swap-in latency. The sharing volume can be expressed as

$$X_i = m_i^{tmp} + x_i, \quad 0 \le x_i \le m_i^{wt}, \text{ for all } \tau_i \in \tau \quad (2)$$

Note that X_i may be smaller than m_i^{tmp}, but this does not affect correctness because temporary data require no restoration and therefore incur no swap overhead. If X_i cannot be filled by temporary data alone, weight memory chunks are incrementally added until the remaining volume is satisfied.

To defer l_i, weight chunks are selected in reverse order of layer execution, prioritizing those used in later layers. This strategy pushes the synchronization point deeper into the execution timeline, thereby enlarging $C_{i,f}$ and providing a wider window in which data transfers can be overlapped with computation. This process is subject to two constraints.

First, chunks that contain both weights and activations are excluded, since restoring them would corrupt valid activations with outdated data. Second, when a weight chunk is referenced by multiple layers, the synchronization boundary is determined by the earliest layer that requires it. The back segment S_i^b is thus formed by aggregating weight chunks in decreasing order of layer index until their cumulative size reaches x_i. The earliest layer requiring any of these weights becomes the synchronization point l_i, while all preceding layers constitute the front segment S_i^f. Through this boundary decision process, minimizing x_i via temporary data selection and deferring l_i via late-layer prioritization, ZeroSwap reduces the effective read latency while maximizing I/O-computation overlap, thereby minimizing swap-induced delays.

D. Sharing Volume Assignment

Even if segment boundary decisions maximize overlap, assigning a sharing volume without considering how much of its read overhead can be hidden by overlap can still introduce delays that prevent timely inference. Therefore, the sharing volume of each task must be chosen sufficiently large to ensure that the memory footprint of all tasks can be mapped into the physical capacity (R1), yet carefully bounded so that, given the degree of overlap achievable for each task, the remaining overhead still allows all tasks to meet their deadlines (R2).

To satisfy R1, we first define the total memory footprint required by the task set τ. Each task τ_i consumes a private memory region of size $(m_i - X_i)$, which cannot be shared with others, in addition to contributing to the shared region of size X_i. Since the shared memory is common to all tasks, its actual physical occupancy is determined by the maximum swap volume among tasks, $\max_{\tau_i \in \tau}(X_i)$, rather than the sum. The system can accommodate all tasks in physical memory only if this total footprint does not exceed the available capacity m^D. Hence, R1 is satisfied when:

$$\sum_{\tau_i \in \tau} (m_i - X_i) + \max_{\tau_i \in \tau}(X_i) \leq m^D. \tag{3}$$

To satisfy R2, we must determine whether a task set τ with assigned sharing volumes X_i can meet its timing requirements under ZeroSwap. For this purpose, we develop a schedulability analysis based on utilization-based non-preemptive EDF analysis [31], [32] that explicitly accounts for segment-level overlapping. For each task τ_i, the effective execution time under overlap is given by Eq. (1), with X_i determined according to Eq. (2). This definition accounts for both the synchronization delay at l_i and the subsequent computation. The utilization of τ_i is therefore expressed as $U_i = C_i^{exec}/T_i$. Because ZeroSwap adopts non-preemptive execution, including at synchronization points, a higher-priority task may be delayed by the complete execution of a lower-priority task. Consequently, the maximum blocking time is bounded by the longest execution among all tasks, i.e., $\max_{\tau_i \in \tau} C_i^{exec}$.

Using these definitions, we obtain the following schedulability condition.

Theorem 1: A task set τ with assigned volumes X_i is schedulable under ZeroSwap if

$$\frac{\max_{\tau_i \in \tau} C_i^{exec}}{\min_{\tau_i \in \tau} T_i} + \sum_{\tau_i \in \tau} U_i \leq 1.0 \tag{4}$$

Proof: The proof follows directly from the non-preemptive EDF schedulability analysis [31], [32], using C_i^{exec} as an upper bound on task execution and $\max_{\tau_i \in \tau} C_i^{exec}$ as the maximum blocking time. ■

Bringing together the conditions established for R1 and R2, we formulate the sharing volume assignment as the following optimization problem. The objective is to minimize the total sum of effective read volume for minimized overhead across all tasks, subject to both memory and schedulability constraints:

$$\underset{X_i}{\text{minimize}} \quad \sum_{\tau_i \in \tau} x_i \tag{5a}$$

$$\text{subject to} \quad \text{Eq. (2), Eq. (3), Eq. (4)} \tag{5b}$$

Solution. We solve the optimization problem using Gurobi [33], a well-known solver. Since ZeroSwap relies on CUDA VMM for memory management, we restrict the decision variable x_i to discrete values in 2 MB increments, matching the minimum granularity of VMM-managed memory chunks. Additionally, the parameters required for optimization, such as $R(x_i)$, $C_{i,f}$, and $C_{i,b}$, can be efficiently obtained through linear regression and offline profiling.

VII. Evaluation

A. Experiment Setup

We implement and evaluate ZeroSwap on top of PyTorch [21], a widely used ML framework. Since ZeroSwap can be easily integrated into various ML frameworks, we used PyTorch for its implementation simplicity. Experiments are conducted on an NVIDIA Jetson Orin Nano [22], an integrated GPU platform equipped with a 6-core Arm Cortex processor, 8GB LPDDR5 memory, CUDA 12.6, and an SK Hynix Platinum P41 SSD (PCIe 3.0x4) [23]. Since part of the memory is reserved for system services, the available capacity for DNN execution, m^D, is about 7.5 GB. Our evaluation uses four representative DNN models: YOLOv7 [3], DeepLabV3 [24], Faster R-CNN [25], and EfficientNetV2 [26]. This diverse combination reflects realistic perception workloads in autonomous driving and robotics, where multiple cameras and varied vision tasks (e.g., object detection and segmentation) must execute concurrently. To investigate the impact of the compute–I/O performance balance on overlapping, we vary the dynamic voltage and frequency scaling (DVFS) modes of the Orin Nano. The device supports three DVFS configurations, `15W`, `25W`, and `MaxN`, that scale the GPU frequency. Execution-time and memory profiles of the four models under each mode are summarized in Table II. Each model is executed

TABLE II: Memory footprints, input resolution, and WCETs of DNN workloads under different DVFS modes

Model	Resolution	Memory Footprint (MB)			WCET (ms)		
		m_i	m_i^{wt}	m_i^{tmp}	15W	25W	MaxN
YOLOv7	640×640	1224	148	278	99.3	70.4	66.1
DeepLabV3	520×520	1270	240	320	371.6	259.3	241.6
Faster R-CNN	375×500	1812	174	660	455.6	321.9	313.4
EfficientNetV2	224×224	962	218	30	86.5	94.3	74.0

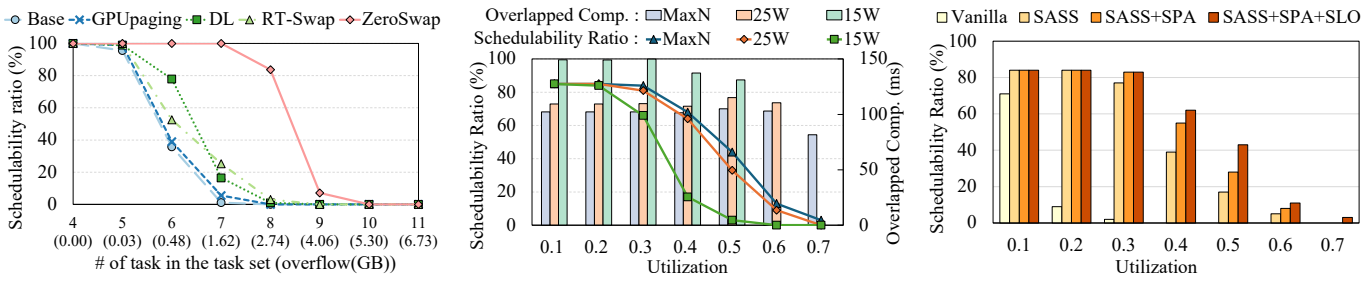


Fig. 10: Schedulability ratio comparison for (a) task-set size, (b) DVFS settings, and (c) ablation of ZeroSwap components

1,000 times, and the maximum latency is taken as the worst-case execution time (WCET). Memory usage is monitored via `/proc/meminfo`, with page caches dropped before each run. The model parameters m_i^{wt} and m_i^{tmp} are obtained by intercepting CUDA allocation APIs through API hooking. Unless otherwise specified, all experiments are conducted under the `MaxN` DVFS mode.

We compare the following four approaches with ZeroSwap:

- Base: no swapping capability;
- GPUpaging [16]: GPU-SSD paging via driver modification, with LRU eviction;
- Demand-Layering (DL) [17]: weight-buffer reuse with consecutive-layer overlapping; and
- RT-Swap [15]: CPU-GPU discrete memory-based swapping;

Base executes DNN tasks only when all models fit in RAM, without SSD swapping. GPUpaging enables GPU–SSD paging via driver modifications, evicting pages on demand with an LRU policy. Since per-task swap volume is unknown, we estimate it offline through emulation and add the corresponding overhead to execution time for schedulability analysis. Demand-Layering reduces memory usage by reusing weight buffers and loading weights layer by layer from SSD while overlapping I/O with computation. When reads exceed computation time, the resulting delays are incorporated into the execution time. RT-Swap, originally designed for discrete GPUs with CPU–GPU swapping, is adapted for integrated GPUs by using host-pinned memory and redirecting the swap device from CPU memory to the SSD. Schedulability ratios are computed using Eq. (4), except when comparison approaches define their own analysis.

B. Simulation Results

Schedulability with an increasing number of tasks. To evaluate schedulability under varying memory pressure, we construct multi-DNN task sets of increasing size. Each task τ_i is a DNN inference randomly drawn from the four models, with periods assigned by the UUniFast algorithm [34] to fix total utilization at 0.3. We first generate 100 task sets with four tasks each, which fit entirely within the available GPU memory m^D. Memory demand is then increased by adding one task per set, producing 100 new task sets at each step. Task periods are adjusted to preserve total utilization, so that only memory pressure varies. This process continues up to 11 tasks per set, spanning memory-sufficient to memory-constrained workloads. As the number of tasks grows, the total memory footprint exceeds m^D, increasing the average overflow, as indicated in parentheses along the x-axis of Fig. 10a.

Fig. 10a compares schedulability ratios as the number of tasks increases from 4 to 11. ZeroSwap consistently outperforms all baselines, with improvements of 111.1%, 101.7%, 66.8%, and 74.7% over Base, GPUpaging, Demand-Layering, and RT-Swap, respectively. Base, which lacks swapping, supports at most six tasks before exceeding memory capacity. GPUpaging yields only slight gains since its LRU eviction ignores task deadlines despite enabling GPU–SSD paging. Demand-Layering effectively reduces parameter memory by reusing weight buffers; however, because it does not manage activations or workspace buffers, the overall memory footprint grows rapidly as the number of tasks increases. Furthermore, because it operates reactively at runtime without offline analysis, it suffers from severe execution stalls when encountering layers with heavy I/O demands. RT-Swap enables swapping and schedules up to seven tasks, yet its per-task memory management and discrete memory-oriented optimizations fail to address SSD–RAM bottlenecks in integrated GPUs, capping its performance.

In contrast, ZeroSwap uses semantic-aware selective swapping with shared pinned allocation to suppress memory demand growth as tasks increase. For instance, while independent management incurs an overflow of 5.3 GB at 9 tasks, ZeroSwap reduces this to 1.6 GB, substantially lowering the swap volume. Furthermore, swap overhead is restricted to read operations, which are effectively hidden behind computation by leveraging offline optimization to pre-determine optimal segment boundaries, thus preventing the I/O stalls seen in reactive approaches. This increases the DNN execution time by only 3.6% on average. Consequently, ZeroSwap schedules up to nine tasks even when memory demand exceeds physical capacity by over 4 GB, with near-zero swap overhead.

Schedulability across DVFS modes. The exposed overhead of weight reads is determined by the degree of overlap between PCIe transfers and computation, governed by the ratio between processing capability and storage throughput. To examine how this balance affects performance, we vary the DVFS modes of the Jetson Orin Nano. Adjusting DVFS scales GPU frequency, and thus DNN execution time, while the SSD's read throughput remains constant, revealing how changes in compute–I/O balance influence overlap and overall performance. 100 task sets consisting of 8 DNNs are generated under the `MaxN` mode, with task periods scaled to achieve utilizations from 0.1 to 0.7. Task sets with utilization above 0.7 are excluded, as no schedulable configurations were found.

TABLE III: Task configuration for the 4-camera case study

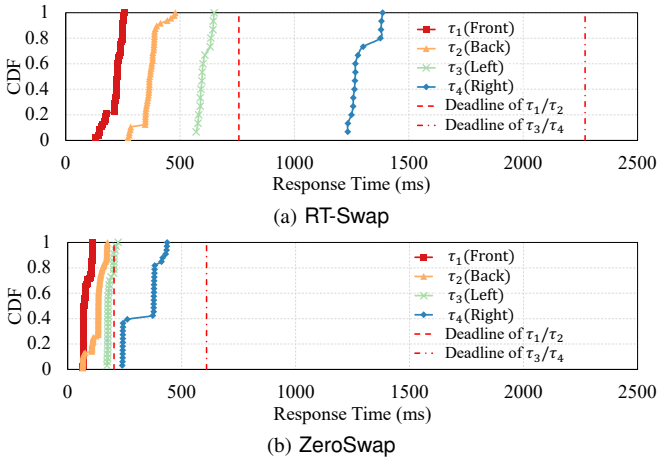
Task	Camera	Period (ms)		Swap Vol. (x_i) (MB)	
		RT-Swap	ZeroSwap	RT-Swap	ZeroSwap
τ_1	Front	757	203	192	186 (54)
τ_2	Back	757	203	192	186 (54)
τ_3	Left	2271	609	192	188 (56)
τ_4	Right	2271	609	192	188 (56)

These sets are then fixed, and schedulability is evaluated while varying the DVFS configuration.

Fig. 10b plots the schedulability ratios across DVFS modes (lines) together with the average amount of computation overlapped with PCIe transfers for a single task (bars). As expected, the `MaxN` mode achieves the highest schedulability ratio due to its superior computational performance. Interestingly, the `25W` mode, despite offering lower computational capability, delivers a schedulability ratio comparable to that of `MaxN`. The reason is that the longer execution time under `25W` enlarges the overlap window for hiding I/O latency, thereby compensating for slower computation. This effect is confirmed by the measured overlap: from utilization 0.1 to 0.6, where both modes find schedulable task sets, the average overlapped computation is 110.3 ms under `25W` compared to 102.7 ms under `MaxN`, resulting in an additional 7.6 ms per task, or approximately 60.8 ms across an 8-task set. In contrast, `15W` mode further extends the overlap opportunity to 149.9 ms but suffers from excessively long execution times that dominate system behavior, resulting in the lowest overall performance. These results demonstrate that the degree of computation–I/O overlap is a key determinant of performance and that ZeroSwap, by explicitly optimizing this overlap, achieves robust schedulability across diverse system settings.

Ablation Study. We conduct an ablation study to quantify the contribution of each component in ZeroSwap. Starting from Vanilla, described in Sec. II-A, we incrementally apply semantic-aware selective swapping (SASS), shared pinned allocation (SPA), and segment-level overlapping (SLO). The study uses 100 task sets of eight tasks each, with utilization varied from 0.1 to 0.7.

Fig. 10c presents the results of the ablation study. Vanilla manages all DNN data in pageable memory and performs swapping without overlap, causing schedulability to drop sharply as utilization rises due to unamortized swap overhead. With SASS, non-essential data are excluded from the swap pipeline, substantially reducing overhead and improving schedulability. Nonetheless, residual costs from memory management and swapping of essential data still lead to degradation at higher utilization. Adding SPA (SASS+SPA) provides little benefit at low utilization, where SASS alone is effective, but at higher levels it reduces repeated allocation and host–device copy overhead, stabilizing performance. Finally, with SLO (SASS+SPA+SLO), even the latency of reading essential weights is hidden behind computation, allowing the system to find a schedulable configuration under heavy load. This ablation study validates the strict necessity of our unified approach: removing any single component causes the remaining fragmented overheads to dominate, inevitably leading to a schedulability breakdown.



(a) RT-Swap

(b) ZeroSwap

Fig. 11: Response time CDF under RT-Swap and ZeroSwap

C. Runtime Experiments

To evaluate how effectively ZeroSwap reduces swap overhead, we conduct a runtime case study comparing ZeroSwap with RT-Swap in a four-camera object-detection pipeline representative of autonomous perception, deployed on the NVIDIA Jetson Orin Nano. [5] The workload comprises four periodic DNN tasks (Front, Back, Left, Right), each running YOLOv7 with 480×480 input resolution on the nuScenes dataset [35]. Each task consumes 1.36GB of system memory, yielding a total footprint of 5.44GB.[6] The system provides 5GB of usable memory for DNN execution out of 8GB physical capacity, with the remainder reserved for the operating system and concurrent CPU workloads. This results in approximately 440 MB overflow, thereby inducing swap operations.

The Front and Back (camera) tasks operate at a higher sampling rate, while the Left and Right (camera) tasks run at one-third of that rate to emulate asymmetric sensing demands. For both ZeroSwap and RT-Swap, we vary the input sampling period T_i while preserving this ratio, and apply schedulability analysis to determine the minimum feasible T_i, corresponding to the maximum achievable FPS that satisfies all task deadlines. The resulting configurations are then deployed on hardware, where we measure end-to-end response times at runtime.

Table III summarizes the resulting periods, and swap volumes with effective read amounts x_i for RT-Swap and ZeroSwap. RT-Swap yields feasible configurations only at 757 ms (1.3 FPS) for Front/Back and 2271 ms (0.4 FPS) for Left/Right, whereas ZeroSwap sustains schedulability at 203 ms (4.9 FPS) and 609 ms (1.6 FPS), achieving a 3.8× higher sampling rate. These sampling rates are identical to those attainable with sufficient memory, showing that ZeroSwap enables the system to behave as if no memory overflow occurred. This improvement arises from ZeroSwap's efficient swap pipeline: whereas RT-Swap manages 192 MB per task without semantic awareness, ZeroSwap restricts reads to 54–56 MB of essential data and further minimizes exposed swap delay through PCIe transfer overlapping.

[5] Demo videos illustrating the experiments are available at https://rtcl.dgist.ac.kr/index.php/zeroswap.

[6] Although reduced-resolution inputs are used, residual memory demand remains due to nuScenes data handling and detection processing.

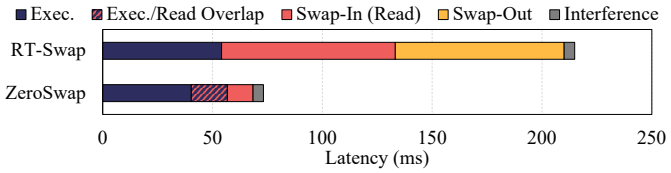


Fig. 12: Response time breakdown of τ_1

Fig. 11 presents the CDFs of end-to-end response times for the 4 object-detection tasks under RT-Swap and ZeroSwap. Both approaches complete all tasks within their respective deadlines; however, ZeroSwap consistently achieves significantly lower response latencies than RT-Swap. For τ_1, RT-Swap exhibits a maximum response time of 254.9 ms, whereas ZeroSwap reduces it to 110.5 ms, corresponding to an improvement of approximately 2.3×. Across all tasks, the maximum observed response time under RT-Swap is 1385.9 ms, compared to 436.2 ms with ZeroSwap, again demonstrating a speedup of 3.2×.

To understand the source of response-time reduction, we analyze the average response time of τ_1 under RT-Swap and ZeroSwap, as shown in Fig. 12. Other tasks differ only by higher-priority interference and are omitted. With RT-Swap, the system swaps the full 192 MB during both swap-in and swap-out, incurring latencies of 79.1 ms and 76.9 ms, respectively. By contrast, ZeroSwap restores only 54 MB of essential data, eliminates swap-out, and performs a single swap-in with a total latency of 28.1 ms. Moreover, 58.3% (16.4 ms) of the latency is hidden by overlapping with YOLOv7 execution. Consequently, the average response time improves from 214.9 ms under RT-Swap to 73.2 ms under ZeroSwap, a 2.9× reduction.

D. Overhead Analysis

We measured the profiling, initialization, communication, and metadata overhead of ZeroSwap, based on the previous case study. The memory layout of weights is captured once by hooking weight copies during initialization. After several inference runs, the total memory footprint is profiled via CUDA hooks and system monitoring. Initialization time rises from 460.4 ms to 1634.9 ms due to VMM-based Shared Pinned Allocator allocation, but occurs only once. IPC communication between the scheduler, ML framework, and swapper costs 8.3 µs. Metadata for the global sharing pool is 2.6 KB and scales linearly with task count. Fragmentation is limited to 2 MB, given the 2 MB chunk size. Finally, when GPU physical memory is sufficient, ZeroSwap incurs negligible runtime overhead, mainly from lightweight IPC, as swap operations are inactive.

VIII. Related Work

Discrete GPU Swapping. Early systems [8]–[10] extend GPU memory by paging data to CPU memory through driver-level modifications. For example, Gdev [8] swaps GPU memory objects into host memory, and GPUSwap [9] enables direct host access to reduce transfer overhead. Framework-based approaches [11]–[14] achieve similar goals by selectively offloading tensors to CPU memory based on layer structure or runtime profiling. vDNN [11] transfers intermediate feature maps during training to reduce GPU footprint, Capuchin [12] dynamically selects which allocations to offload, SwapAdvisor [13] overlaps tensor transfers with execution through scheduling, and Sentinel [14] prioritizes transfers based on tensor hotness. While effective on discrete GPUs, they are not applicable to integrated GPUs, where CPU and GPU share a unified memory. More recently, RT-Swap [15] leverages VMM to provide predictable swapping between CPU and GPU for real-time multi-DNN inference, but still depends on runtime allocation, ignores tensor semantics, and targets discrete GPUs. In contrast, ZeroSwap provides a swapping framework specialized for integrated GPUs that eliminates runtime allocation, minimizes swap overhead through semantics-aware selective swapping, and coordinates swap scheduling with task timing to ensure predictable and deadline-aware inference.

Integrated GPU Swapping. In contrast to discrete GPUs, integrated GPUs share a unified physical memory, making CPU-assisted swapping ineffective since transferring data to CPU-visible regions does not release additional capacity. To address this, research has explored SSD-based extensions. Bakita and Anderson [16] leverage direct SSD I/O on integrated GPUs to enable paging to SSD. Framework-level solutions such as Demand Layering [17] and TASO [18] adopt layer-wise weight loading, overlapping I/O with computation to mitigate memory usage while hiding transfer latency. Hardware–software co-design efforts, such as ZnG [19] and scheduling-aware prefetching [20], integrate GPUs more tightly with SSDs, exploiting flash storage as an extension of global GPU memory. While these approaches enable integrated GPUs to exploit SSDs for expanded capacity, they lack deadline-aware memory management and leave swap operations unoptimized, incurring substantial overhead. In contrast, ZeroSwap minimizes swap overhead and explicitly incorporates task timing into its swapping and overlapping mechanisms, enabling timely inference in real-time multi-DNN workloads.

IX. Conclusion

This paper introduced ZeroSwap, a memory swapping framework designed for real-time multi-DNN inference on integrated GPUs to mitigate swap-induced overhead. To achieve this goal, ZeroSwap proposes three complementary mechanisms that collectively minimize swap overhead by reducing data movement, eliminating runtime management costs, and overlapping data transfers with computation. In addition, ZeroSwap determines segment boundaries and sharing volumes that jointly minimize swap overhead while ensuring the timely execution of DNN tasks. Implementation and extensive evaluation on a state-of-the-art machine learning framework demonstrate that ZeroSwap effectively accommodates larger real-time DNN workloads within limited GPU memory, maintaining timing guarantees and achieving faster response. As future work, we plan to extend ZeroSwap to discrete GPU platforms and further explore multi-GPU configurations that share a single SSD to enable coordinated swap management across devices.

Acknowledgement

This work was supported in part by (1) the National Research Foundation of Korea (NRF) grant (RS-2023-00213309, RS-2024-00438248, RS-2025-17622972) and (2) Institute of Information & communications Technology Planning & Evaluation (IITP) grant (RS-2024-00398157, RS-2025-02219277(AI Star Fellowship Support Project(DGIST))) funded by the Korea government (MSIT).

REFERENCES

[1] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *ICLR*, 2021.

[2] Z. Li, W. Wang, H. Li, E. Xie, C. Sima, T. Lu, Q. Yu, and J. Dai, "Bevformer: Learning bird's-eye-view representation from multi-camera images via spatiotemporal transformers," in *ECCV*, 2022.

[3] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," in *CVPR*, 2023.

[4] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," in *ICML*, 2023.

[5] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," in *ICLR*, 2019.

[6] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *CVPR*, 2018.

[7] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge Distillation: A Survey," *International Journal of Computer Vision*, vol. 129, pp. 1789–1819, 2021.

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

[9] J. Kehne, J. Metter, and F. Bellosa, "GPUswap: Enabling oversubscription of GPU memory through transparent swapping," in *VEE*, 2015.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] W. Kang, J. Lee, Y. Lee, S. Oh, K. Lee, and H. S. Chwa, "RT-Swap: Addressing GPU Memory Bottlenecks for Real-Time Multi-DNN Inference," in *RTAS*, 2024.

[16] J. Bakita and J. H. Anderson, "Enabling GPU memory oversubscription via transparent paging to an NVMe SSD," in *RTSS*, 2022.

[17] 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.

[18] Y. Wen, A. Anderson, V. Radu, M. F. O'Boyle, and D. Gregg, "TASO: Time and space optimization for memory-constrained dnn inference," in *SBAC-PAD*, 2020.

[19] J. Zhang and M. Jung, "ZnG: Architecting gpu multi-processors with new flash for scalable data analysis," in *ISCA*, 2020.

[20] T.-Y. Wang, C.-F. Wu, C.-W. Tsao, Y.-H. Chang, and T.-W. Kuo, "Scheduling-aware prefetching: Enabling the pcie ssd to extend the global memory of gpu device," in *NVMSA*, 2021.

[21] 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.

[22] NVIDIA Jetson Orin Nano Super DevKit. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/nano-super-developer-kit/

[23] SK hynix Platinum P41 SSD. [Online]. Available: https://ssd.skhynix.com/platinum_p41/

[24] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking atrous convolution for semantic image segmentation," in *arXiv*, 2017.

[25] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *NIPS*, 2015.

[26] M. Tan and Q. Le, "EfficientNetV2: Smaller models and faster training," in *ICML*, 2021.

[27] S. Bateni, Z. Wang, Y. Zhu, Y. Hu, and C. Liu, "Co-optimizing performance and memory footprint via integrated cpu/gpu memory management, an implementation on autonomous driving platform," in *RTAS*, 2020.

[28] J. Choi, H. You, C. Kim, H. Y. Yeom, , and Y. Kim, "Comparing unified, pinned, and host/device memory allocations for memory-intensive workloads on Tegra SoC," *Concurrency and Computation: Practice and Experience*, vol. 33, 2021.

[29] PyTorch Contributors, "PyTorch Documentation: CUDA Memory Management," https://docs.pytorch.org/docs/stable/notes/cuda.html#memory-management, accessed: 2026-03-17.

[30] TensorFlow Contributors, "BFC (Best-Fit with Coalescing) Allocator Implementation," https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/common_runtime/bfc_allocator.cc, 2024, accessed: 2026-03-17.

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

[32] 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.

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

[34] E. Bini and G. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, May 2005.

[35] H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom, "nuscenes: A multimodal dataset for autonomous driving," in *CVPR*, 2020.