curtSCHED: Architecture-Independent Real-Time GPU Scheduling via Statistical Deferrable Servers

Hao Zhang[00000−0002−2541−0140] and Frank Mueller[0000−0002−0258−0294]

Department of Computer Science, North Carolina State University,
Raleigh, NC, United States
hzhang47@ncsu.edu
mueller@cs.ncsu.edu

Abstract. Computation-intensive tasks such as deep neural network (DNN) training and inference utilize graphics processing units (GPUs) to accelerate computation. However, when autonomous driving leverages the DNN model for real-time object detection and on-board continuous learning simultaneously access the same embedded GPU, the scheduling policy on a GPU is static and cannot be altered, which presents a problem as their non-preemptive intra-context kernel execution lacks real-time guarantees. NVIDIA and prior work provide space partitioning techniques, e.g., Multi-Process Service (MPS) and Multi-Instance GPU (MIG), for better utilization of GPU resources and asymmetric partitioning of streaming multiprocessors (SMs). But MPS requires specific GPU architecture support (Volta and above) and lacks support for embedded GPUs such as NVIDIA AGX. MIG pre-allocates partial GPU resources for instances in predefined coarse ratios but lacks the flexibility required for fine-grained scheduling. Furthermore, both MPS and MIG also lack any notion of real-time constraints termed time partitioning.

We present curtSCHED, a real-time scheduling framework based on a statistical model and a dynamic deferrable server mechanism, which is task-aware, yet architecture- and application-independent. It provides real-time guarantees for one class of GPU kernels while supporting best-effort execution for another class in a transparent manner and without GPU architecture support, non-portable driver modifications or even changes to the application binaries. This capability is demonstrated for a combination of real-time inference and best-effort continual learning tasks in an autonomous driving scenario. We demonstrate the ability to characterize the execution behavior of kernels execution time via model prediction for a workload that is itself composed of DNN kernels in the profiling and learning phase, which is subsequently leveraged by curtSCHED to ensure deadlines of GPU kernels with real-time priority during the online scheduling phase. A curtSCHED prototype has been implemented to control GPU execution from the CPU-triggered invocations of kernels. We also investigate the root cause of abnormally long execution times and solve this problem on the CPU side. Experiments show that jobs of real-time tasks meet their deadlines with negligible utilization overhead on CPU, GPU and overall application execution.

1 Introduction

GPUs are increasingly used in practice for accelerating computational kernels ranging from numerical kernels to machine learning (ML) algorithms. Recently, such GPU acceleration has also been exploited under real-time constraints, such as autonomous driving (AD). However, GPUs are inherently hardware-controlled with non-preemptive kernel execution, which is a mismatch to preemptively scheduled task sets under real-time static priority or dynamic deadline scheduling. This work tries to bridge this divide between non-preemptive GPU kernels and preemptive CPU scheduling generically for GPU kernels, which is demonstrated specifically for real-time tasks related to AD.

AD relies significantly on perception with the objective of understanding the surroundings by detecting and tracking objects in everyday driving scenarios. The faster and more accurate information the AD system obtains from a perception subsystem, the safer and more comfortable AD can be. Perception is an ML concept based on DNNs integrated as a subsystem of the onboard self-driving system. For example, Intel Mobileye [26], NVIDIA Drive OS [11], Volkswagen and Daimler VWOS [36], Tesla AutoPilot [4], and Comma.ai openpilot [29] all integrate perception in their AD systems and leverage the on-vehicle acceleration, mostly via GPUs, to process large real-time sensing data, such as camera images (2D pixel arrays) and Lidar (Light Detection and Ranging) images (3D point coordinates).

GPUs, initially developed for video editing and gaming, have become widely utilized for ML applications. NVIDIA offers the Jetson product line, specifically designed for embedded or edge devices, delivering substantial computational power while maintaining low energy consumption. E.g., the Jetson Orin delivers 200 trillion operations per second (TOPS) while consuming only 40W.

The training (aka. learning) and inference tasks involved in perception for AD are highly computationally intensive. Perception DNN models require a massive amount of parallel calculations, such as convolution, multiplication, and other matrix-based arithmetic computations. Model training is typically performed in cloud data centers using a cluster of high-end GPUs, such as NVIDIA A100/H100 devices, over days or even weeks. Once trained, these inference models are deployed on vehicular devices to enable AD. Models periodically receive over-the-air (OTA) updates via cellular networks every few months as training is repeated with additional real-world scenarios. However, these updates can lag behind, leaving gaps in handling newly encountered challenges.

An alternative to late OTA updates would be to enable on-board self-driving systems to learn from "near-mistakes" with current hardware capabilities via "continual learning" (CL). For example, by reviewing image frames prior to a near-mistake that led to a hard braking action, a self-driving car could learn roadside features associated with similar scenarios much sooner (within minutes or hours), such that repeated mistakes can be avoided when entering the same intersection on consecutive days. The benefit of continual on-board learning is apparent. Instead of waiting for periodic OTA updates from the cloud, the con-

tinually learned model improves its handling of similar driving scenarios with much shorter turn-around time.

At first glance, it appears practical to perform training tasks on-board, given that the volume of data required for learning is significantly smaller than the vast datasets used for training with GPU clusters (tens of images versus hundreds of thousands). Additionally, these training tasks are not time critical, meaning that best-effort completion is sufficient, with model updates occurring within minutes or hours. However, due to the nature of DNN-based training, which involves forward and backward propagation, as well as data augmentation, *continual learning* is inherently far more computationally intensive than any online *inference* task. Moreover, inference tasks are subject to stringent real-time constraints, where missing a deadline could lead to severe consequences, including accidents that may result in damage or even loss of life. For instance, if an object detector fails to recognize a pedestrian in time, serious bodily injury could occur. Furthermore, when CL and real-time inference share the same GPU, a training kernel could interfere with and delay the real-time inference kernel, compromising the vehicle's ability to detect objects promptly. Hence, the deadline of real-time tasks executing on GPUs cannot be guaranteed when non real-time tasks also use the same GPU.

To address the challenge of scheduling mixed real-time and best-effort GPU tasks, we present curtSCHED, a statistical model-based runtime scheduling framework that supports mixed real-time and background (non real-time) execution of GPU kernels with only minimal software modifications. We propose a time-partitioning model for GPU sharing, grounded in real-time scheduling theory, that offers finer granularity and higher precision than existing GPU real-time scheduling approaches. We further demonstrate curtSCHED for real-time inference and online CL tasks in AD. curtSCHED guarantees that inference tasks meet their real-time constraints while the CL tasks make best-effort progress, even for scenarios where only one kernel executes on a GPU at a time. The key idea is to divide kernel execution into two phases. During the first phase, a statistical model is leveraged to automatically "learn" the execution time of kernels at a low-level CUDA driver interface, which is demonstrated for DNN execution profiled on a commodity GPU. curtSCHED enables kernel-specific execution time profiling, which covers forward and the backward propagation of a DNN. The second phase features online GPU sharing between real-time and background kernels, i.e., inference and training tasks in our running example. Our proposed priority scheduling policy is based on slack management to guarantee a lower bound for inference speed. Experimental results show that our framework is GPU architecture independent. With its profile-based timing analysis, inference tasks meet their deadlines while training tasks progress in their training. In this approach, we retain the original DNN architecture, i.e., do not apply pruning, dropout shortcuts, weights approximation, or any other techniques to modify inference. Hence, our method does not suffer from performance accuracy trade-offs, i.e., the original inference tasks remain unaffected.

Contributions: First, to the best of our knowledge, this is the first model-based time-partitioning scheduling at the kernel invocation level, designed specifically for real-time AD tasks concurrently accessing a single GPU. Second, we develop a technique employing a statistical model capable of dynamically predicting kernel execution times, which are managed by the low-level CUDA driver. Third, our framework is applicable to arbitrary applications and even closed-source GPU software stacks, as it neither requires support from the hardware nor relies on non-portable modifications of GPU drivers, in contrast to prior work. In fact, not even application changes are required. We implement curtSCHED based on C/C++ and ELF (executable and linkable format) binary interception to conduct an extensive evaluation on various GPU architectures. Fourth, we investigate the root cause of abnormally long end-to-end kernel execution times and solve this problem by leveraging CPU-side memory pools. Lastly, we perform a comprehensive evaluation of state-of-the-art space-partitioning techniques for GPUs [7]. The results demonstrate that our framework successfully meets real-time constraints with a guarantee rate of 99.85%. In contrast, the related work achieved only a 98.4% success rate, corresponding to a frame-drop rate of 1.6% for object detection tasks.

2 Related Work

Several methods for GPU sharing among concurrent applications have been explored by multiplexing multiple kernels either temporally [19, 32, 31, 38, 2] or spatially [3, 30, 20, 40, 27, 7]. Such techniques have been also applied for multi-DNN workloads [42, 41, 46]. Cusched [33] proposes hardware extensions to Kepler-like GPUs by adding thread-block–level preemption and in-GPU schedulers (FCFS, priority queues, token) to enable concurrent multi-process execution and improve throughput/fairness in multiprogramming settings. In contrast, our curtSCHED is a user-space framework that intercepts cuLaunchKernel() calls, predicts per-kernel runtimes, and enforces real-time guarantees via a deferrable server—without hardware or driver changes—whereas CUsched even frames soft-RT only through hardware preemption at thread-block granularity.

GPU scheduling problem for real-time tasks has been explored with different solutions. Fractional GPUs [14] proposes a software-based mechanism to allow multiple applications to run in parallel on a single GPU while still maintaining isolation by space partitioning compute and memory resource among these applications. In this way, memory interference among applications is reduced and runtime predictability is increased for the real-time application. Another approach [28] is to reverse engineer the closed scheduling mechanisms deployed in NVIDIA GPUs to infer the scheduling hierarchy of GPUs by analyzing runtime resource occupation, which enables schedulability for latency-sensitive applications. Applying dynamic scheduling (EDF) enables preemption on GPU but requires non-portable NVIDIA driver modifications [9].

Another approach is to adapt the DNN application to meet real-time constraints. Kang et al. [18] propose a fine-grained resource allocation model that

schedules the DNN task at the level of layers on both CPUs and GPUs. It compromises on bit accuracy to speed up processing time of the DNN task on the CPU. To meet a deadline, Subflow [21] dynamically drops neurons within the DNN to reduce the volume of the computation, thereby completing the DNN task in time. Jiang et al. [15] present a flexible framework for high-resolution object detection on edge devices. This framework splits the image into disjoint areas and applies different object detection models to associated partitions. However, this incurs high memory consumption when multiple DNN models need to be loaded. Furthermore, the fast but reduced accuracy model to meet timing constraints compromises inference accuracy. Hence, this approach is not suitable for the autonomous driving platforms where on-board resources are limited and safety cannot be compromised. Kang et al. [17] also split the DNN task, but unlike [15], they divide the task into subtasks with different safety critical levels and schedule these subtasks based on priority levels. All of these, including [13], sacrifice accuracy to meet timing constraints of real-time tasks by modify the DNN application. In contrast, our work retains the original DNN architecture/application code without loss of accuracy for inference tasks while ensuring that online training tasks make progress.

Han et al [12] leverage the idempotency of DNN kernel calls, i.e., a DNN call for some input produces the same output if called repeatedly without any side effects on global state. They devise a preemption scheme that launches a real-time kernel on the GPU, which aborts a currently running GPU kernel mid way and reissues the aborted one at a later time. However, this mechanism requires patches on the binary representation of the GPU driver, i.e., it is not portable across driver versions and across different hardware generations. Furthermore, it incurs overhead when stopping the kernel execution mid way. In contrast, our approach does not require any software or driver modifications, nor application changes, and is not constrained by idempotency of kernels.

Yi et al. [43] design a pseudo-preemption mechanism on a single GPU for concurrent DNN-based augmented reality (AR) applications. RT-mDL [24] is a framework to support mixed real-time deep learning (DL) tasks on edge platforms with heterogeneous CPU and GPU resources. It scales down DL models to reduce small storage cost, which enables fast detection of traffic sign classes but loses 1.7% accuracy. Both works' scheduling unit is layer-wise or partitions the DNN. In contrast, we schedule at the granularity of kernel functions, which is application independent, i.e., generalizes beyond DNNs. Prior works [43, 24] aim at scheduling multiple DNN *inference* tasks combined with offloading partial workloads to CPU cores, while our work targets computationally more demanding online DNN *training* combined with real-time inference. The training time on each image is typically 5X longer than that of inference [45].

Continual Learning as a Background Workload As continual learning (CL) gains momentum in deep learning, its industrial and real-world applications, including autonomous driving (AD), are being actively explored. For object detection, [45] investigates online, on-vehicle CL and proposes an ML ar-

chitecture tailored to AD. For perception beyond detection, [16] adapts semantic segmentation to a CL setup and evaluates it on the CLAD benchmark [35]. Additional AD use cases studied under a CL paradigm include road-surface classification [10], velocity control [39], and traffic forecasting [34]. Complementary work such as [45] further explores continual adaptation in AD settings.

These contributions develop CL algorithms and pipelines. However, they are largely orthogonal to our focus on runtime predictability when CL (treated here as a best-effort, event-driven workload) shares a single GPU with real-time perception. curtSCHED addresses this systems problem via kernel-level interception, execution-time prediction, and a deferrable-server policy for time partitioning.

PipeSwitch [5] leverages the layering structure of DNN models with its layer-by-layer computation pattern to pipeline model transmission over the PCIe bus by executing and tasks on the GPU. This work aims to improve the GPU cluster utilization and reduce the context switch overhead of multiple DNN applications (learning and inference). Yet, it lacks support for real-time constraints of DNN tasks on GPUs, which is required by real-time inference and, in our case, combined with online learning on board a self-driving vehicle. Prophet [25] proposes a timing model on per-image processing granularity and coordinates multiple DNN tasks (object and lane detection) to estimate real-time perception execution time. The timing information is profiled per DNN layer and requires application adaption.

3 Background and Motivation

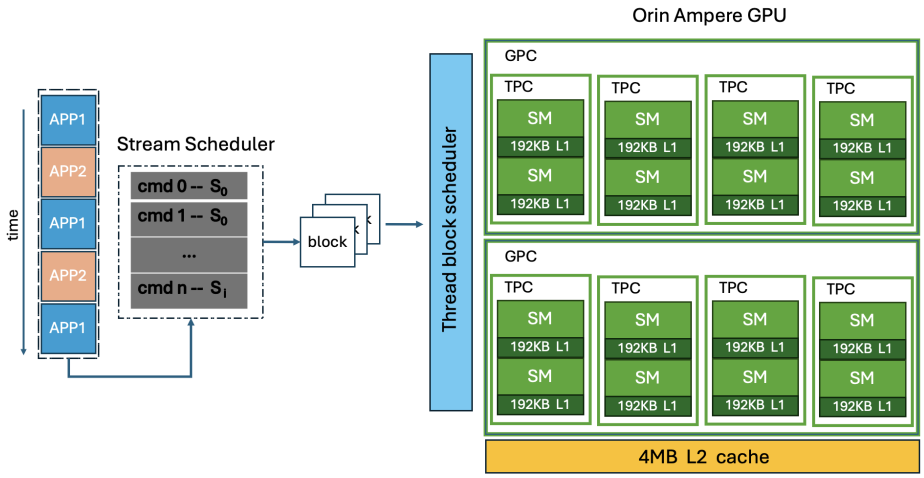


Fig. 1: AGX Orin Ampere Scheduling Hierarchy

3.1 Non-preemptive and Black-box Scheduling on GPUs

A GPU kernel, once invoked, is executed non-preemptively until completion. Stream priorities via CUDA library API *cudaStreamCreateWithPriority*() do not enable true preemption of running kernels (e.g., forcibly interrupt a kernel mid-execution). The related work [44] establishes a scheduling order for multiple jobs by suspending execution at warp boundaries with block execution. It cannot support runtime interruption due to lack of hardware support, i.e., their technique is non-preemptive up to warp execution and will incur sometimes shorter, sometimes longer delays until a long-latency memory reference is encountered. This is a mismatch for contemporary preemptive real-time scheduling policies. While there is research focused on enabling preemption on GPUs [9] to support dynamic priority task scheduling or non-preemption GPU execution [19] to support static priority, these works are either poised by proprietary code restrictions, or they require an open-source driver or a modified proprietary GPU driver, generally without vendor support and guaranteed support for future software releases.

Scheduling in NVIDIA GPUs is a hierarchical arbitration mechanism. We use the latest embedded GPU platform AGX Orin as an example to illustrate GPU scheduling. This Ampere GPU consists of two Graphic Processing Clusters (GPCs), eight Texture Processing Clusters (TPCs), 16 Streaming Multiprocessors (SMs), 192 KB of L1-cache per SM, and 4 MB of L2 Cache. There are 128 CUDA cores per SM. As shown in Figure 1, from left to right, the hierarchical order involves an application scheduler, stream scheduler, and thread block scheduler. The application scheduler is at the top of the hierarchy. A detailed description of the scheduling policy at the highest level of the GPU arbitration mechanism has been presented in [9]. The remaining schedulers including the warp scheduler are at the level of the GPU firmware or hardware. At the driver-level, each application seeking GPU resources opens a number of channels, which are inserted in a ready queue. Each entry in the ready queue is characterized by a timeslice length and a priority value. This mechanism is work-conserving using TDMA (Time Division Multiple Access) among channels, where each channel can be assigned multiple slots within the queue, according to its priority value, thus shaping the sequence of slots for the TDMA round.

However, how these firmware schedulers divide application streams, arrange execution orders, and execute on the warp is not open sourced and remain entirely proprietary. Previous research work [28, 1] has unveiled some underlying details by performing black-box testing with GPU micro benchmarks, e.g., indicating that the stream scheduler is in the FIFO queue based on the stream operation submission time.

The characteristics of the stream scheduler provide the opportunity of breaking down CUDA operations as scheduling units before sending them to the GPU. CUDA's API is divided into two layers: the high-level runtime API and the low-level driver API. Both are in user space and provide an interface for managing the GPU. The low-level driver API interprets the high-level runtime API, subdivides the kernel defined in the runtime API, and launches kernel execution on

the GPU. The crux of the problem is to *find a portable mechanism that allows us to schedule the execution order of low-level APIs on the CPU* before it is sent to the FIFO queue of the stream scheduler on GPU, which is beyond user/CPU host control.

3.2 Background Workloads Sharing the GPU (Case: Continual Learning)

Many deployments co-locate *non–real-time* background workloads on the same GPU as *real-time* (RT) perception or control. Examples include continual/training-style updates, mapping/SLAM back-ends, data compression, or batch analytics. In our case study, *continual learning* (CL) is representative as it is event-driven, compute-intensive (forward+backward passes, augmentation), and lacks hard deadlines, which fits the best-effort (BE) class. Prior work has surveyed CL methods [37], but the specifics of catastrophic forgetting are orthogonal to our contribution.

The key systems problem is **concurrency on a single GPU**: BE kernels can delay RT kernels absent explicit time partitioning and priority control. This motivates curtSCHED's design (Section 4): intercept kernel launches, predict per-call execution time, and allocate BE work only within slack via a deferrable server, which preserves RT guarantees while allowing BE progress. We evaluate this with CL as one BE instance (Section 5), but the mechanism applies to any BE workload sharing the GPU.

4 Design and Implementation

4.1 curtSCHED Design

The goal of curtSCHED is to provide a generic, application-agnostic GPU scheduling framework capable of enforcing real-time guarantees for selected tasks while ensuring best-effort progress for others. The framework operates entirely in user space and requires no modifications to GPU hardware, firmware, or driver binaries. While our evaluation in Section 5 uses an autonomous driving (AD) scenario as a case study, the underlying mechanisms apply to any CUDA-based workload that launches GPU kernels through the CUDA driver API.

Figure 2 illustrates the high-level architecture of curtSCHED, and Figure 3 shows its anchoring point in the GPU driver layer. Regardless of the application, all CUDA kernels are ultimately launched through the driver-level function `cuLaunchKernel()`. curtSCHED intercepts these calls in user space, without modifying application binaries, GPU drivers, or hardware.

curtSCHED operates in two distinct phases: *Profiling & Model Fitting* and *Online Scheduling*. In the first phase, only the timing model is enabled, learning GPU kernel execution times from collected data samples. In the second phase, the real-time scheduler and deferrable server leverage the timing model's predictions to schedule multiple RT and BE jobs. Upon interception, the framework extracts

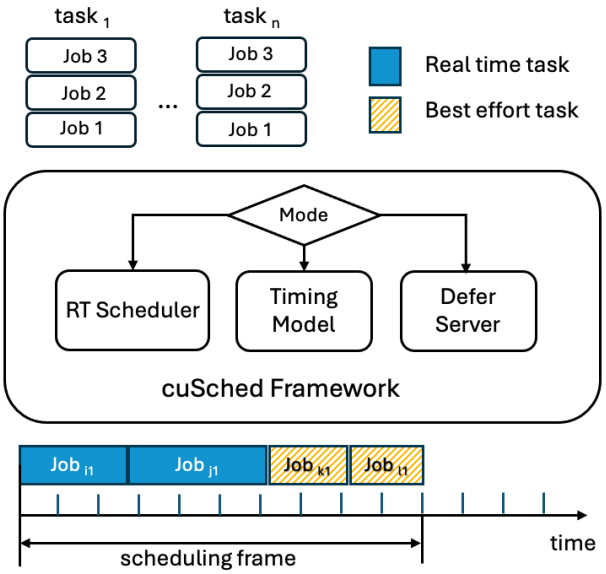


Fig. 2: curtSCHED: A Statistical Model-driven Deferrable Server-based GPU scheduling framework

a kernel call signature (function ID, launch configuration, parameters) and passes it to the timing model. In Phase 1, the recorded execution time is used as ground truth for model training. In Phase 2, the execution time is predicted by the timing model. The scheduler then decides whether to

1. launch RT kernels immediately based on priority, and to
2. place BE kernels in a deferrable server queue before dispatching them if the remaining execution budget in the current frame can accommodate their predicted runtime.

This design offers several benefits. First, fine-grained scheduling at the kernel-call level maximizes GPU utilization and reduces the probability of RT deadline misses. Second, the interception mechanism applies uniformly to all CUDA applications and libraries (e.g., cuDNN) without requiring application source code changes, which makes the framework entirely application-independent. Third, the design is portable across GPU architectures and does not require architecture-specific features or driver modifications [3]. Finally, because all logic resides on the CPU side, curtSCHED imposes negligible GPU overhead. The CPU-side interception adds only minimal CPU cycles per kernel launch, which remain fully preemptive under the host OS scheduler.

4.2 Task Model and Scheduling:

curtSCHED adopts a mixed workload model of *periodic real-time (RT)* tasks and *aperiodic best-effort (BE)* tasks. This model supports multiple tasks requesting

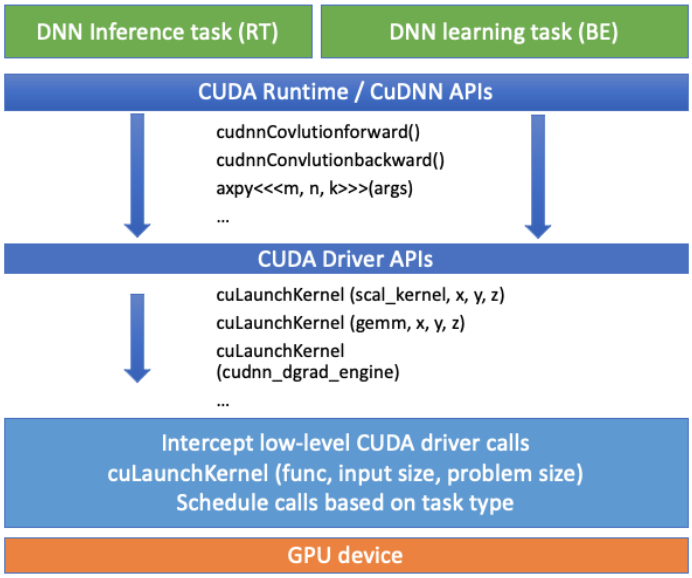


Fig. 3: curtSCHED interception workflow for DNN Tasks

concurrent access to the GPU and is generic to many domains. Our evaluation (Section 5) uses an autonomous driving (AD) scenario as a case study.

A periodic RT task T_i is defined by the tuple (p_i, e_i, d_i), where p_i is the period, e_i is the worst-case execution time (WCET), and d_i is the relative deadline. An execution instance of T_i is denoted as job j_i. In contrast, BE tasks are aperiodic (non-real-time) and have no fixed period or strict deadline. They are triggered by events such as the availability of new data for processing.

Time is divided into *frames* following the cyclic executive model [6], adapted here for GPU kernel scheduling. In a cyclic executive, the system operates in a repeating sequence of fixed-length time slots, with each job assigned a specific slot. Scheduling decisions are made only at frame boundaries, so there is no preemption within a frame.

As shown in the top half of Figure 4, in the theoretical model the RT tasks (blue) T_1 and T_2 are released periodically, and the deferrable server task T_s (orange) reserves a portion of the frame's execution budget for BE tasks. Multiple BE jobs can share the reserved bandwidth within the frame. The bottom half of Figure 4 illustrates a real-world execution trace, where kernel execution times vary between invocations.

The frame size f must be large enough so that every job can start and complete within the frame, yet small enough to keep the schedule table compact. As per [6], f should satisfy:

$$2f - \gcd(p_i, f) \leq D_i. \tag{1}$$

In our case study, to meet a 30 FPS RT requirement, the frame size is set to 1/30 s ($\approx$ 33.3 ms). The schedule is static and clock-driven, i.e., RT jobs are released and scheduled at fixed times regardless of events.

At runtime, synchronization ensures orderly GPU access. Each RT job processes one data unit (e.g., an image) per period. RT jobs are always given priority to guarantee deadlines, and any slack time within the frame is used to execute BE jobs from the aperiodic queue via the deferrable server. This design assumes that GPU capacity is sufficient to meet RT demand in isolation. Under this assumption, executing RT jobs first in each frame ensures their deadlines are met (see Equation 3). In practice, actual execution times vary between kernel invocations, even for the same kernel signature, due to runtime factors such as GPU memory contention. The next section describes how curtSCHED addresses this variability.

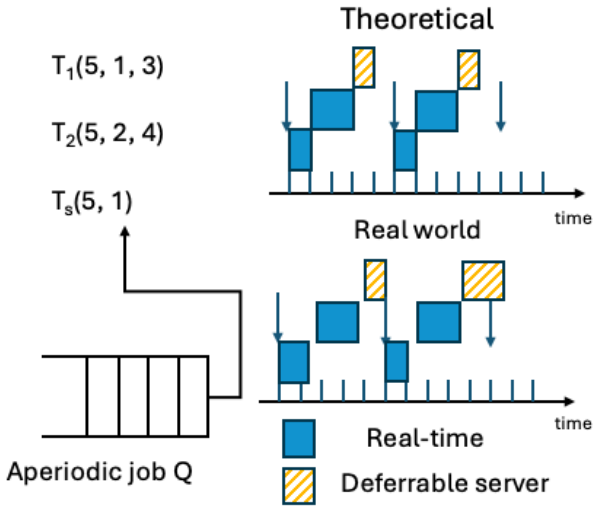


Fig. 4: Theoretical vs. real world scheduling

In our autonomous driving case study, we instantiate the task model with two real-time tasks of different criticality levels. Task T_1 (RT_high) performs detection of vehicles and vulnerable road users (VRUs) including pedestrians and cyclists, while task T_2 (RT_low) handles traffic sign recognition. Given that collision avoidance is more time-critical than sign recognition for immediate safety, T_1 is assigned higher priority ($P_1 > P_2$). This priority ordering aligns with the ASIL (Automotive Safety Integrity Level) requirements in ISO 26262, where VRU detection typically requires ASIL-D (highest) while traffic sign recognition may be classified as ASIL-B. Both tasks must meet their respective deadlines

D_1 and D_2 within each frame period, with Time-Demand Analysis (Equation 3) ensuring schedulability under the fixed-priority scheme.

4.3 Two-Phased Framework

Having established the task model and scheduling constraints in Section 4.2, we now describe how curtSCHED implements its two-phased approach to enforce these real-time guarantees. curtSCHED divides kernel execution interception into two distinct phases: (1) *Profiling & Model Fitting* and (2) *Online Scheduling*. In the first phase, the framework collects and summarizes GPU kernel execution times as descriptive statistics. From these collected samples, a statistical model learns to predict execution times for individual kernel calls at the low-level CUDA driver interface. Profiling is performed on a per-kernel *call signature* basis, which includes the caller's address and launch parameters (e.g., grid/block dimensions, shared memory usage, and problem size). This allows curtSCHED to distinguish between calls to the same kernel with different characteristics and to associate each with its own profiling record.

In the second phase, curtSCHED uses the trained model to make per-call execution time predictions and schedules tasks based on priority and available execution budget. Real-time (RT) kernels are guaranteed precedence, while best-effort (BE) kernels are dispatched only if their predicted runtime fits within the current frame's remaining budget. The real-time priority policy ensures a bounded interference from background execution.

The mode switch, illustrated in Figure 2, enables the timing model only during the profiling phase, while all modules (including the scheduler and deferrable server) are enabled in the online phase.

First Phase: Profiling & Model Fitting The framework records execution times of intercepted `cuLaunchKernel()` calls for each unique call signature and summarizes these in a fixed-size sliding window. Continuous operation could generate unbounded memory requirements for timing data and impose significant CPU overhead for statistical updates. curtSCHED addresses this by maintaining a rolling set of samples using a sliding window with an LRU replacement policy. When the window is full, the oldest sample is removed to accommodate the newest one. Within each window, the framework tracks

- minimum and maximum execution times,
- rolling mean and variance, and
- histogram of observed times.

Each profiling sample records the tuple (f, gx, gy, gz, bx, by, bz, t), where f identifies the intercepted low-level kernel function, (gx, gy, gz) and (bx, by, bz) represent the grid and block dimensions respectively, and t is the measured execution time in microseconds. For instance, profiling Yolo yields samples such as (scale_kernel, 512, 22, 1, 201, 1, 1, 32), (upsample_kernel, 512, 57, 1, 241, 1, 1, 1444), and (add_bias_kernel, 512, 21, 1, 344, 1, 1, 361), where the final

value represents execution time. The multiple linear regression (MLR) model treats the first seven components as feature vectors and learns the mapping to execution time, enabling runtime prediction for scheduling decisions in the online phase.

Profiling data are stored in an in-memory hash map with $O(1)$ access time cost. The primary table maps a kernel function ID to a secondary table keyed by launch parameters, which holds the associated timing statistics. Within each histogram bin, a binary search tree (BST) stores the samples in sorted order, enabling efficient calculation of confidence intervals during scheduling.

Once sufficient samples are collected, curtSCHED fits a MLR model,

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n + \epsilon, \quad \epsilon \sim N(0, \sigma^2),$$

where the x_i are feature vectors derived from the kernel call signature and problem dimensions. The training process takes only seconds on modern CPUs, as the model complexity is intentionally kept low for runtime efficiency. The trained model is then serialized and loaded during system initialization for the online phase. The fitted model provides execution time predictions with configurable confidence bounds.

Second Phase: Online Scheduling The online phase uses the timing model to drive a deferrable server [22] for BE task execution while preserving RT deadlines. The deferrable server maintains a replenishable execution budget per frame for BE jobs without impacting the schedulability of RT jobs.

When a BE job reaches the head of its queue, curtSCHED

1. retrieves the predicted execution time from the model,
2. compares it to the remaining budget in the current frame, and
3. executes the job if it fits; otherwise, defers it to the next frame.

The prediction interval at confidence level $(1 - \alpha)$ is

$$\hat{y} \pm t_{\frac{\alpha}{2}, df} \cdot s_e \sqrt{1 + \mathbf{x}_0^T (X^T X)^{-1} \mathbf{x}_0}, \tag{2}$$

where $t_{\frac{\alpha}{2}, df}$ is the t-distribution critical value, s_e is the standard error, $df = N - n - 1$, and $\mathbf{x}_0 = [1, x_1, x_2, \ldots, x_n]^T$.

When the deferrable server has the lowest priority in a fixed-priority system, higher-priority RT tasks are unaffected. The Time-Demand Analysis (TDA) for an RT task T_i is

$$W_i(t) = e_i + \sum_{j \in \mathrm{HP}(i)} \left\lceil \frac{t}{p_j} \right\rceil e_j, \tag{3}$$

where e_i is the WCET of T_i, $\mathrm{HP}(i)$ is the set of higher-priority tasks, and p_j and e_j are the period and WCET of a higher-priority task T_j. TDA ensures schedulability for all RT tasks under the assumption that GPU capacity meets their demand.

Algorithm 1 deferrable server thread vs. main thread

```
procedure DEFERRABLE SERVER
    while true do
        Lock defer_mutex
        Wait on defer_cond with defer_mutex held
        while defer_queue is not empty do
            args ← DEQUEUE(defer_queue)
            if time_remaining > model_predict(args) then
                REAL_CULAUNCHKERNEL(args)
                defer_queue.pop()
            else
                Wait on defer_cond with defer_mutex held
    Unlock defer_mutex
```

curtSCHED.config:

```
mode ← model_train | schedule
period ← 1/30 second
real_time_task_ids ← RT_THREAD
best_effort_task_ids ← BE_THREAD
confidence ← 0.95
CFP ← CFP ON GPU
```

```
procedure CURTSCHED(Task_Type, func_args)
    if Task_Type = Higher_RT then
        while ¬turn_for_high do
            Wait on rt_sem
        REAL_CULAUNCHKERNEL(func_args)
        high_done ← 1
        turn_for_high ← 0
        Post rt_sem
    else if Task_Type = Lower_RT then
        while turn_for_high do
            Wait on rt_sem
        REAL_CULAUNCHKERNEL(func_args)
        low_done ← 1
        turn_for_high ← 1
        Post rt_sem
    elseBest_Effort
        Lock defer_mutex
        while QUEUE_IS_FULL(defer_queue) do
            Wait on defer_cond with defer_mutex held
        ENQUEUE(defer_queue, func_args)
        Unlock defer_mutex
    if high_done ∧ low_done then
        high_done ← 0
        low_done ← 0
        Lock defer_mutex
        time_remaining ← GET_FRAME_IDLE
        Broadcast defer_cond
        Unlock defer_mutex
        wait_for_next_frame ← 0
```

In the case study, inference tasks (RT) always take precedence over training tasks (BE). This priority assignment reflects the safety-critical nature of real-time perception, while training proceeds only when spare GPU capacity is available.

4.4 Implementation

We utilize synchronization to control and coordinate concurrent accesses from disjoint tasks to the GPU and employ a frame-based schedule to accommodate the non-preemptive execution of GPU kernels. As shown in Algorithm 1, for each frame in our schedule, an intercepted low-level CUDA kernel call corresponding to any real-time or best-effort job (e.g., the CL training thread) is being blocked at the kernel level. Any higher priority RT job executes before lower priority jobs, the latter of which only run after all RT jobs have completed within the current frame. This is accomplished by putting concurrent BE jobs into the defer queue of fixed size, which also blocks BE jobs when queue is full.

After all the RT jobs have completed within a frame, the remaining time in this frame is calculated and the deferred server thread is woken up. On the deferred server side, fine-grained jobs of only BE tasks are executed conditionally

considering their model-predicted execution time vs. the remaining time budget in the frame. If sufficient slack remains within the frame, the lower priority BE jobs issues the call on GPU immediately; otherwise, it defers until the next frame. We employ a simple parser at the initial stage to make our framework user friendly and to provide flexibility for experimental setups. The parser reads the curtSCHED.config file and extracts values that define the characteristics of tasks and scheduling information. These include the mode (indicating whether the framework is used for model fitting or online scheduling), the scheduling period/frame for all tasks, the confidence score for model prediction, and the call function path (CFP) ID corresponding to the specific GPU on which the experiment is running. (The CFP ID refers to a function in the code, which is automatically identified later.)

The use of a deferrable server enhances GPU utilization by capitalizing on idle time and offers flexible management of aperiodic tasks. While challenges may include potential budget inefficiencies (small unused idle periods due to conservative predictions of WCET) and minor interference with real-time tasks, these are effectively mitigated by assigning the lowest priority to aperiodic tasks and employing a statistical model for precise execution time predictions. This approach ensures minimal disruption, thereby realizing a balanced and optimized scheduling strategy.

5 Evaluation

The objectives of the experiments are threefold. First, we evaluate whether the curtSCHED framework meets real-time constraints, i.e., if it ensures each image frame is processed within the deadline imposed by the frame rate (e.g., 30 FPS), while concurrently allowing the best-effort training tasks to progress by utilizing the available deferrable server. Second, we assess the general applicability of curtSCHED across various NVIDIA GPUs. Third, we perform a comparative evaluation of curtSCHED against related scheduling frameworks.

Experimental Setup: We evaluated curtSCHED on various commodity GPUs (see Table 1). curtSCHED is responsible for scheduling both real-time and best-effort tasks, which involves executing long sequences of GPU kernel calls to implement higher-level ML functionality using the Yolov4 object detection model [8], based on Darknet and written in C/C++. A real-time task involves object detection inference, where each job processes every single image frame. During the profiling phase, we observed that the number of low-level kernel calls varies across GPUs, CUDA versions, and DNN structures. (as shown in the last column of Table 1). **Allocator optimization.** Before running any experiments reported in this section, we enabled a memory pool optimization in the application to eliminate sporadic long latencies from dynamic allocation/free (see Section 5.3 for root-cause analysis). Unless otherwise stated, all results were obtained *with* the memory pool enabled. The optimization does not change application semantics and reduces timing variance observed during preliminary profiling.

Table 1: CPU+GPU Hardware Platforms and Generations (gen.), CPF=calls_per_frame (CUDA 11.2 w/ YOLOv4)

CPU architecture	cores	CPU Memory	NVIDIA GPU	GPU Memory	GPU gen.	GPU cores	CUDA gen.	CPF
Intel Sandy bridge	16	16GB DDR3 1600	RTX 2070	8GB GDDR6	Turing	2,304	SM_75	586
Intel Cascade Lake	16	192GB DDR4 2666	RTX 3060Ti	8GB GDDR6	Ampere	4,864	SM_86	554
AMD Epyc Rome	16	128GB DDR4 3200	RTX A4000	16GB GDDR6	Ampere	6,144	SM_86	554
AMD Epyc Rome	16	128GB DDR4 3200	A100	80GB HBMe2	Ampere	6,912	SM_80	637
ARM v8.2 64bit	8	32GB LPDDR4x (shared)	AGX Jetson Xavier	32GB LPDDR4x	Volta	512	SM_72	646
ARM Cortex-A78AE	6	32GB LPDDR5 (shared)	AGX Jetson Orin	32GB LPDDR5	Ampere	1,024	SM_87	555

Evaluation metrics: The inference speed measures if the inference task can predict objects within an image before the deadline, while the training speed indicates how quickly the training task makes progress. We measure these two metrics at the application side based on end-to-end latency. E.g., if the deadline is 50 ms, the inference speed should be greater than or equal to 20 FPS.

Additionally, we calculated the deadline miss rate of the real-time task, which indicates how many real-time jobs processing a single image failed to complete before their deadline.

The curtSCHED framework records and calculates the number of missed cases and their rates during the online scheduling phase. Equation 4 shows the calculation for the miss rate:

$$miss\_rate = \frac{\sum missed\_cases}{\sum processed\_image}. \tag{4}$$

5.1 curtSCHED Evaluation

We evaluate both the logical and temporal correctness of our framework. Since curtSCHED does not apply pruning, dropout shortcuts, weights approximation, nor other techniques to modify higher-level inference, ML metrics for logical correctness remain the same as the origin application. This work focuses on the real-time characteristics of ML execution. Nonetheless, we use these metrics to check logical correctness on all testing devices (see Table 1).

The ML application measures the inference and training speed. By analyzing the source code of the application, the rate is calculated as an average, i.e., counting the total number of images completed over a constant time interval. The average speed indicates a coarse end-to-end latency for the task. It also checks if the majority of real-time jobs finished their computation on time.

In addition to application-side speed monitoring, we measure the number of missed deadlines per scheduled frame within the framework by leveraging the system-wide real-time clock (CLOCK_REALTIME). This fine-grained metric provides detailed timing analysis for our framework.

The purpose of the real-time inference task is to provide object detection outputs per image frame, subsequently used in autonomous driving scenarios. curtSCHED schedules image processing tasks on a per-frame basis, adhering to the specified target frame rate. In contrast, the training task processes images

in batches cached in memory without real-time deadlines; its performance is measured by throughput rather than latency.

Experiments without curtSCHED We compare the inference speed between the real-time task exclusively accessing a single GPU in the presence of both the real-time and the best-effort (training) tasks that are concurrently accessing the same GPU. We do so for a variety of GPUs listed in Table 1 to also expose hardware capabilities and limitations.

Figure 5(a) shows result samples of the inference task with key statistical measures over each x-axis data point (GPU architecture) when the task exclusively occupies the GPU. The computational ability of a GPU directly impacts the median inference speed, with higher computational ability resulting in faster speeds. The results for the NVIDIA AGX Xavier are intentionally omitted here, as its speed of 22.1 FPS does not meet the minimum real-time requirement of 30 FPS. The embedded GPU AGX Orin has the lowest speed among all tested GPUs, with 33.9 FPS and a standard deviation of 5.52. The high-end A100 GPU has an average inference speed of 95.6 FPS with a standard deviation of 11.08.

Figure 5(b) represents the FPS rate for the inference task when the training task is concurrently accessing the same GPU without using curtSCHED. In this case, the average speed slows down to less than half speed compared to Figure 5(a). We observe that the standard deviation becomes lower across all GPU devices compared to the former data series. We also observe a trend of more powerful GPUs resulting in higher standard deviation. Initially, we suspected that the reason for this was due to context switch overhead at the application side, which incurs a worst-case spike when the CPU switches between threads. However, after thorough analysis on both the CPU and GPU side, we located the root cause as originating from the application. We enhanced the application's performance by employing a sophisticated memory pool optimization (discussed in 5.3).

On the A100, running both the inference and training tasks simultaneously still results in a inference speed meeting the minimum requirements of 30 FPS. However, it is questionable if such an expensive and power-intensive GPU (300W peak) will ever be deployed on a vehicle, although future hardware generations may provide such computational capabilities at lower power, as has been the case in the past. Nonetheless, we also observe that when two or more best-effort training tasks are deployed in the background, real-time constraints of 30 FPS for the inference task can no longer be guaranteed.

Experiments with curtSCHED We evaluate curtSCHED with the multi-task configuration described in Section 4.2, where RT_high (vehicle and VRUs detection) and RT_low (traffic sign recognition) execute concurrently with best-effort continual learning tasks. Figure 6a depicts in bars both inference (RT tasks) and training (BE tasks) speeds in terms of frame rate.

Deadline misses primarily result from inaccuracies in WCET predictions. Although our model achieves high-confidence predictions, perfect (100%) accuracy

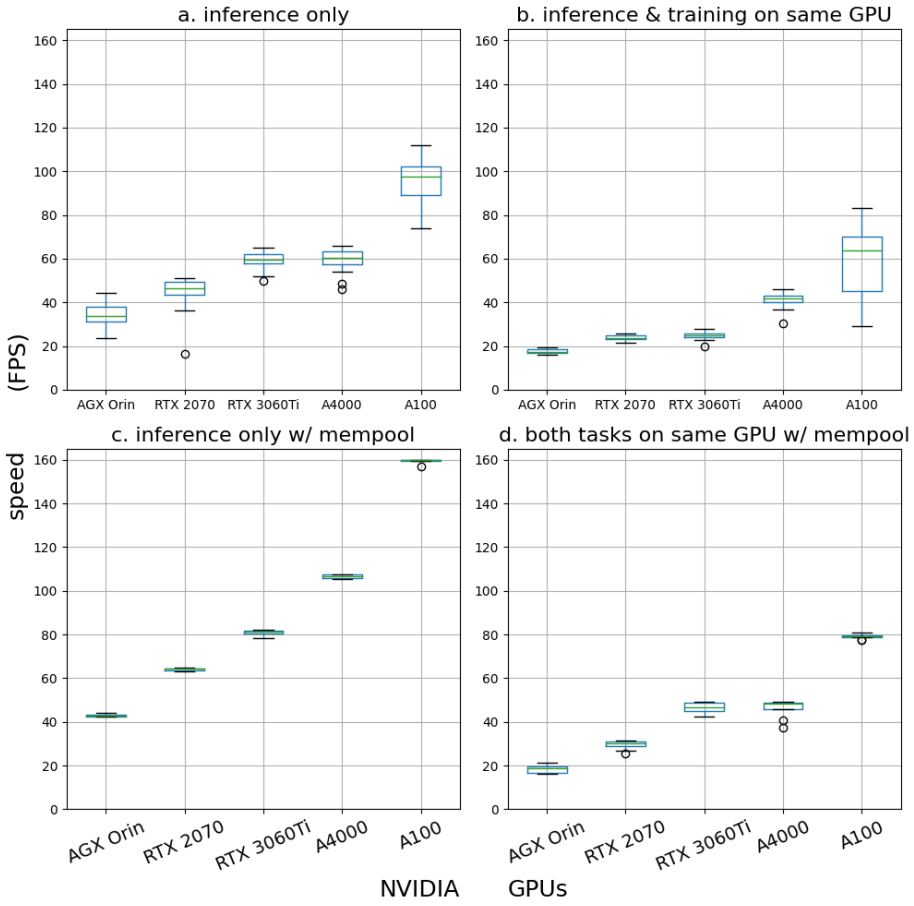


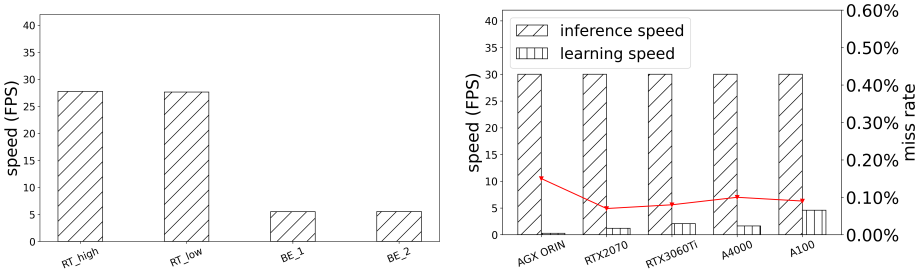
Fig. 5: Inference speed comparison

is practically not achievable, but the method suffice if high accuracy with occasional overruns are acceptable. Under the same stipulation, another source of deadline misses occasionally arises from resource contention due to mutexes and locks shared across multiple tasks. Such contention can introduce unexpected delays, causing tasks to exceed their deadlines.

We start from single RT and BE tasks. Figure 6b depicts both inference and training speeds as bars in terms of frame rate (left y-axis) as well as a red line for the deadline miss rate (right y-axis). Compared to the multiple tasks for both RT and BE categories, the converged frame speed matches the target value incurring a maximum missed rate of 0.15%.

The occasional deadline misses can be tolerated in autonomous driving, as each frame is highly correlated with its neighbor frames. As a result, the real-time frame rate is only reduced slightly (e.g., from 30 to 27.6 frames per second) when frames are dropped due to missed deadlines. As long as these misses are

bounded, i.e., by ensuring at most k frames are dropped within a sliding window of n frames, this behavior can be explicitly accommodated within the safety requirements. We use the COCO [23] dataset to perform inference on RT tasks with around 11k images. Based on the results, the minimum detection speed is 27.085 FPS with an average speed of 27.64 FPS. Thus, a $k = 3$ out of $n = 30$ fraction is guaranteed in practice. That is, this framework ensures safety standards remain satisfied despite minor inaccuracies and task contention in practice.



(a) Multiple RT and BE w/ curtSCHED (b) Speed and miss rate w/ curtSCHED

Fig. 6: Timing results

5.2 Comparison with Space Partitioning and MIG

We aimed to compare curtSCHED with the time-partitioning approach in [2], but the public code repository lacks the specific commit and key implementation details needed for replication. As an alternative, we reproduced a space-partitioning method [7] and use it as a baseline in our evaluation.

As shown in Figure 7, we reproduced the results of a space partition method [7] on an RTX2070 GPU. This requires modifications of Nvidia header files and exposing undocumented control of the mask bits of SMs. Applications then have the capability to select SMs before running GPU kernels. Figure 7(a) depicts results where only the real-time task accesses the GPU, while Figure 7(b) shows results for both types of tasks on the same GPU. The x-axis indicates how many SMs are used for the real-time task. In Figure 7(b), the training task uses the remaining disjoint SMs on the same GPU. We observe interference on the real-time task when multiple tasks share the GPU seen by comparing Figures 7(a) and (b). This aligns with the results shown in Figure 14 in [7], where both inference speeds fluctuated.

Nvidia introduced the MIG mode in the A100, which enables space partitioning of GPU resources. The A100 offers fixed configurations to select from, namely 1g/10gb, 2g/20gb, 3g/40gb, and 4g/40gb (both compute/memory). The entire GPU has 7g compute and a total of 80gb memory. We tested the real-time

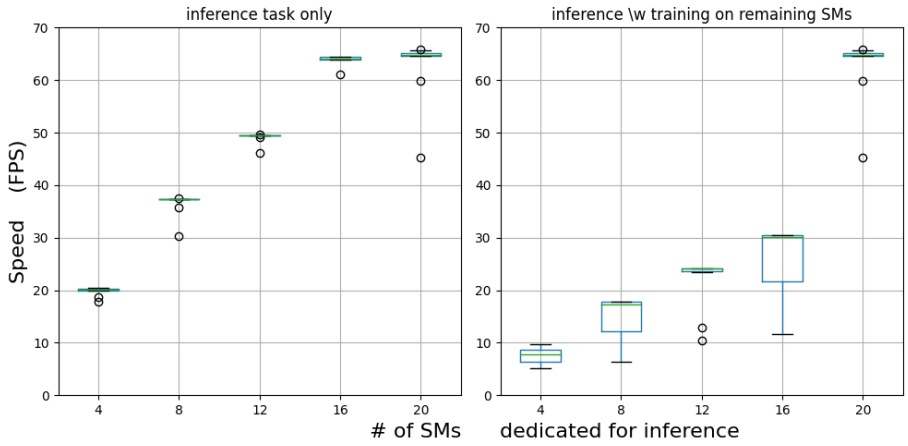


Fig. 7: Space partition results

task both individually and with both tasks (real-time and best-effort) running on the A100. The inference speed (real-time task) in both cases is consistent and does not show any interference when the two tasks are co-scheduled (Figure 8(a)). Since MIG slices both compute and memory, it provides full isolation for each partition.

curtSCHED, based on the cyclic executive time partition, is more suitable for real-time tasks than space partitioning and MIG. Compared to space partition, curtSCHED guarantees real-time constraints via time partitioning. Additionally, curtSCHED provides flexibility in tuning the deadline. The results in Figure 8(b) show that the configured deadline is met on the RTX 2070 GPU with significantly lower variability (tight boxes) in execution time and only few outliers.

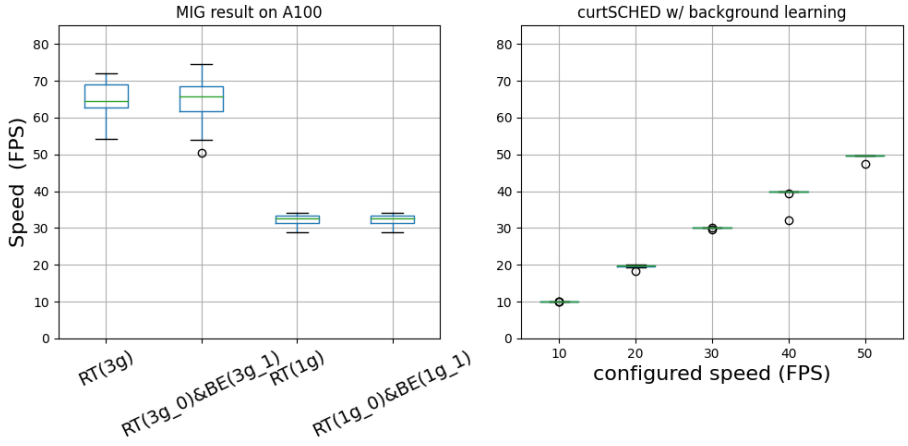


Fig. 8: MIG (left) and curtSCHED (right) results

5.3 Root Cause of Missed Deadline Cases

In the early stage of evaluation, we found that the missed deadline rate ranged from 1% to 3.2%, varying by GPU. We investigated the root cause of deadline misses using an empirical method: We obtained execution times by adding instrumental code to the framework, application, and GPU kernel execution. The cause was not related to GPU glitches incurring long execution times, as observed and analyzed in related work [2].

We located the root cause on the application side. The running example involves loading images from files, copying them to memory, and dynamically allocating and freeing memory during runtime. As shown in Algorithm 2, the original procedure of the Darknet-based Yolo loads images from files, copies them to allocated memory, and frees memory after image processing completes. Upon

Algorithm 2 Yolo Image Loading Procedure

```
1: i ← 0
2: while i < n_images do
3:     thread_join
4:     X ← get_next_image
5:     start_LoadImageThread(next_file → memory)
6:     Y ← predict(X)
7:     i ← i + 1
8:     free(X)
```

iterating over testing for n images on the Yolo Model, the image loading thread is started prior to the while loop. Within the loop, the program waits for the image matrix X to be extracted from the file and predicts the result Y based on X, while starting another loading thread in the meantime. When prediction is done, the program frees the image. We observed that the C library function free() could sometimes have exceptionally long execution times (due to free list traversals and TLB misses on page walks), which are 2-3 orders of magnitude longer than the average running time.

To address this issue, we implemented our own memory pools to avoid frequent calls to free(). Memory pools are commonly used in production software systems such as performance-critical, embedded, and real-time systems. The application is optimized by pre-allocating a fixed-sized memory pool and associating memory locations with a bitmap, indicating whether the corresponding memory space is available or not. Figures 5(c) and (d) plot the results for inference only and both inference+training tasks on various GPUs without the curtSCHED framework. Compared to (a) without the memory pool, the inference speeds are higher and incur less variation in time over all GPUs.

6 Conclusion

This work contributes curtSCHED, a runtime GPU scheduling framework designed for managing real-time and best-effort tasks. The framework employs

fine-grained scheduling strategies tailored specifically to these task types. By leveraging user-space interception, curtSCHED achieves task awareness while remaining application- and GPU architecture-independent, ensuring seamless forward portability as new hardware and software versions emerge. We have validated the curtSCHED framework with high-level machine learning workloads relevant to autonomous driving across a range of GPU devices. Furthermore, the static priority scheduling provided by curtSCHED ensures that real-time tasks reliably meet soft real-time constraints, while simultaneously enabling best-effort tasks to continue progressing effectively.

Acknowledgment

This work was funded in part by NSF grants CISE-2521121, CISE-1747555, and CISE-1813004.

References

1. Amert, T., Otterness, N., Yang, M., Anderson, J.H., Smith, F.D.: Gpu scheduling on the nvidia tx2: Hidden details revealed. In: 2017 IEEE Real-Time Systems Symposium (RTSS). pp. 104–115. IEEE (2017)
2. Amert, T., Tong, Z., Voronov, S., Bakita, J., Smith, F.D., Anderson, J.H.: Timewall: Enabling time partitioning for real-time multicore+ accelerator platforms. In: 2021 IEEE Real-Time Systems Symposium (RTSS). pp. 455–468. IEEE (2021)
3. Ausavarungnirun, R., Miller, V., Landgraf, J., Ghose, S., Gandhi, J., Jog, A., Rossbach, C.J., Mutlu, O.: Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency. ACM SIGPLAN Notices **53**(2), 503–518 (2018)
4. Tesla autopilot. https://www.tesla.com/autopilot (2015)
5. Bai, Z., Zhang, Z., Zhu, Y., Jin, X.: Pipeswitch: Fast pipelined context switching for deep learning applications. In: 14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20). pp. 499–514 (2020)
6. Baker, T., Shaw, A.: The cyclic executive model and ada. In: Proceedings. Real-Time Systems Symposium. pp. 120–129 (1988). https://doi.org/10.1109/REAL.1988.51108
7. Bakita, J., Anderson, J.H.: Hardware compute partitioning on nvidia gpus. In: 2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 54–66. IEEE (2023)
8. Bochkovskiy, A., Wang, C.Y., Liao, H.Y.M.: Yolov4: Optimal speed and accuracy of object detection. arXiv preprint arXiv:2004.10934 (2020)
9. Capodieci, N., Cavicchioli, R., Bertogna, M., Paramakuru, A.: Deadline-based scheduling for gpu with preemption support. In: 2018 IEEE Real-Time Systems Symposium (RTSS). pp. 119–130. IEEE (2018)
10. Cudrano, P., Bellusci, M., Macino, G., Matteucci, M.: Continual cross-dataset adaptation in road surface classification. In: 2023 IEEE 26th International Conference on Intelligent Transportation Systems (ITSC). pp. 4278–4284. IEEE (2023)
11. Nvidia drive os. https://developer.nvidia.com/drive/driveos (2019)

12. Han, M., Zhang, H., Chen, R., Chen, H.: Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). pp. 539–558. USENIX Association, Carlsbad, CA (Jul 2022), https://www.usenix.org/conference/osdi22/presentation/han
13. Heo, S., Cho, S., Kim, Y., Kim, H.: Real-time object detection system with multi-path neural networks. In: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 174–187. IEEE (2020)
14. Jain, S., Baek, I., Wang, S., Rajkumar, R.: Fractional gpus: Software-based compute and memory bandwidth reservation for gpus. In: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 29–41. IEEE (2019)
15. Jiang, S., Lin, Z., Li, Y., Shu, Y., Liu, Y.: Flexible high-resolution object detection on edge devices with tunable latency. In: Proceedings of the 27th Annual International Conference on Mobile Computing and Networking. pp. 559–572 (2021)
16. Kalb, T., Roschani, M., Ruf, M., Beyerer, J.: Continual learning for class-and domain-incremental semantic segmentation. In: 2021 IEEE Intelligent Vehicles Symposium (IV). pp. 1345–1351. IEEE (2021)
17. Kang, W., Chung, S., Kim, J.Y., Lee, Y., Lee, K., Lee, J., Shin, K.G., Chwa, H.S.: Dnn-sam: Split-and-merge dnn execution for real-time object detection. In: 2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 160–172. IEEE (2022)
18. Kang, W., Lee, K., Lee, J., Shin, I., Chwa, H.S.: Lalarand: Flexible layer-by-layer cpu/gpu scheduling for real-time dnn tasks. In: 2021 IEEE Real-Time Systems Symposium (RTSS). pp. 329–341. IEEE (2021)
19. Kato, S., Lakshmanan, K., Rajkumar, R., Ishikawa, Y., et al.: {TimeGraph}:{GPU} scheduling for {Real-Time}{Multi-Tasking} environments. In: 2011 USENIX Annual Technical Conference (USENIX ATC 11) (2011)
20. Lee, M., Song, S., Moon, J., Kim, J., Seo, W., Cho, Y., Ryu, S.: Improving gpgpu resource utilization through alternative thread block scheduling. In: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA). pp. 260–271. IEEE (2014)
21. Lee, S., Nirjon, S.: Subflow: A dynamic induced-subgraph strategy toward real-time dnn inference and training. In: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 15–29. IEEE (2020)
22. Lehoczky, J.P., Sha, L., Strosnider, J.K.: Enhanced aperiodic responsiveness in hard real-time environments. In: Unknown Host Publication Title, pp. 261–270. IEEE (1987)
23. Lin, T.Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., Zitnick, C.L.: Microsoft coco: Common objects in context. In: European conference on computer vision. pp. 740–755. Springer (2014)
24. Ling, N., Wang, K., He, Y., Xing, G., Xie, D.: Rt-mdl: Supporting real-time mixed deep learning tasks on edge platforms. In: Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems. pp. 1–14 (2021)
25. Liu, L., Dong, Z., Wang, Y., Shi, W.: Prophet: Realizing a predictable real-time perception pipeline for autonomous vehicles. In: 2022 IEEE Real-Time Systems Symposium (RTSS). pp. 305–317. IEEE (2022)
26. Intel mobileye. http://www.mobileye.com (1999)
27. Nvidia: Nvidia multi-processes service (mps) (2021), https://docs.nvidia.com/deploy/mps/index.html

28. Olmedo, I.S., Capodieci, N., Martinez, J.L., Marongiu, A., Bertogna, M.: Dissecting the cuda scheduling hierarchy: a performance and predictability perspective. In: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 213–225. IEEE (2020)
29. openpilot. https://www.comma.ai/openpilot (2023)
30. Pai, S., Thazhuthaveetil, M.J., Govindarajan, R.: Improving gpgpu concurrency with elastic kernels. ACM SIGARCH Computer Architecture News **41**(1), 407–418 (2013)
31. Park, J.J.K., Park, Y., Mahlke, S.: Chimera: Collaborative preemption for multitasking on a shared gpu. ACM SIGARCH Computer Architecture News **43**(1), 593–606 (2015)
32. Tanasic, I., Gelado, I., Cabezas, J., Ramirez, A., Navarro, N., Valero, M.: Enabling preemptive multiprogramming on gpus. ACM SIGARCH Computer Architecture News **42**(3), 193–204 (2014)
33. Tanasic, I., Gelado Fernandez, I., Cabezas, J., Navarro, N., Ramírez Bellido, A., Valero Cortés, M.: Cusched: multiprogrammed workload scheduling on gpu architectures (2013)
34. Tsai, M.J., Cui, Z., Liu, C., Yang, H., Wang, Y.: An incremental learning-based framework for non-stationary traffic representations clustering and forecasting. In: 2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC). pp. 3237–3242. IEEE (2022)
35. Verwimp, E., Yang, K., Parisot, S., Hong, L., McDonagh, S., Pérez-Pellitero, E., De Lange, M., Tuytelaars, T.: Clad: A realistic continual learning benchmark for autonomous driving. Neural Networks **161**, 659–669 (2023)
36. Volkswage vw.os. https://www.volkswagenag.com/en/news/fleet-customer/2021/01/transformers.html (2021)
37. Wang, L., Zhang, X., Su, H., Zhu, J.: A comprehensive survey of continual learning: Theory, method and application (2023). arXiv preprint arXiv:2302.00487 **1**(5)
38. Wang, Z., Yang, J., Melhem, R., Childers, B., Zhang, Y., Guo, M.: Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 358–369. IEEE (2016)
39. Wei, D., Xing, J., Yang, S., Lu, Y., Huang, Y.: Continual reinforcement learning for autonomous driving with application on velocity control under various environment. In: 2023 7th CAA International Conference on Vehicular Control and Intelligence (CVCI). pp. 1–8. IEEE (2023)
40. Wu, B., Chen, G., Li, D., Shen, X., Vetter, J.: Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In: Proceedings of the 29th ACM on International Conference on Supercomputing. pp. 119–130 (2015)
41. Xiang, Y., Kim, H.: Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference. In: 2019 IEEE Real-Time Systems Symposium (RTSS). pp. 392–405. IEEE (2019)
42. Yang, M., Wang, S., Bakita, J., Vu, T., Smith, F.D., Anderson, J.H., Frahm, J.M.: Re-thinking cnn frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 305–317. IEEE (2019)
43. Yi, J., Lee, Y.: Heimdall: mobile gpu coordination platform for augmented reality applications. In: Proceedings of the 26th Annual International Conference on Mobile Computing and Networking. pp. 1–14 (2020)

44. Zahaf, H.E., Lipari, G.: Design and analysis of programming platform for accelerated gpu-like architectures. In: Proceedings of the 28th International Conference on Real-Time Networks and Systems. pp. 1–10 (2020)
45. Zhang, H., Mueller, F.: Claire: Enabling continual learning for real-time autonomous driving with a dual-head architecture. In: 2022 IEEE 25th International Symposium On Real-Time Distributed Computing (ISORC). pp. 1–10. IEEE (2022)
46. Zhou, H., Bateni, S., Liu, C.: S^ 3dnn: Supervised streaming and scheduling for gpu-accelerated real-time dnn workloads. In: 2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 190–201. IEEE (2018)