GCAPS: GPU Context-Aware Preemptive Priority-based Scheduling for Real-Time Tasks

Yidi Wang
University of California, Riverside, USA

Cong Liu
University of California, Riverside, USA

Daniel Wong
University of California, Riverside, USA

Hyoseung Kim
University of California, Riverside, USA

Abstract

Scheduling real-time tasks that utilize GPUs with analyzable guarantees poses a significant challenge due to the intricate interaction between CPU and GPU resources, as well as the complex GPU hardware and software stack. While much research has been conducted in the real-time research community, several limitations persist, including the absence or limited availability of GPU-level preemption, extended blocking times, and/or the need for extensive modifications to program code. In this paper, we propose GCAPS, a GPU Context-Aware Preemptive Scheduling approach for real-time GPU tasks. Our approach exerts control over GPU context scheduling at the device driver level and enables preemption of GPU execution based on task priorities by simply adding one-line macros to GPU segment boundaries. In addition, we provide a comprehensive response time analysis of GPU-using tasks for both our proposed approach as well as the default Nvidia GPU driver scheduling that follows a work-conserving round-robin policy. Through empirical evaluations and case studies, we demonstrate the effectiveness of the proposed approaches in improving taskset schedulability and response time. The results highlight significant improvements over prior work as well as the default scheduling approach, with up to 40% higher schedulability, while also achieving predictable worst-case behavior on Nvidia Jetson embedded platforms.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded and cyber-physical systems

Keywords and phrases Real-time systems, GPU scheduling

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2024.11

Acknowledgements This work is supported by the National Science Foundation (NSF) grants 1943265, 1955650, and 2312395.

1 Introduction

Real-time cyber-physical systems with GPU workloads have become increasingly prevalent in various domains including self-driving cars, autonomous robots, and edge computing nodes. This trend has been accelerated in recent years by the demand for learning-enabled components as most of their implementations heavily rely on the GPU stack. The scheduling problem of GPU-using tasks in these systems is therefore crucial to ensure timely execution and to meet stringent timing requirements. One of the key challenges here is effectively supporting prioritization and preemption, allowing higher-priority tasks to interrupt and supersede lower-priority GPU tasks whenever needed. This is particularly important in scenarios where critical high-priority tasks with stringent deadlines need to access GPU resources, while low-priority and best-effort tasks can tolerate such preemption to accommodate their execution.

As of yet, the default scheduling policy of commercial GPU devices provides little control over the prioritization and preemption of GPU access segments of tasks, causing unpredictable task response time and instability in real-time systems. The real-time research community has recognized this issue since the early era of GPU computing and has proposed several solutions. In particular, the use of real-time synchronization protocols, such as MPCP [20, 21] and FMLP+ [10], has been recognized as a promising way to manage GPU tasks in real-time systems with strong analyzable guarantees on the worst-case task response time. However, these approaches can suffer from long blocking time and priority inversion by lower-priority tasks since GPU segments are handled non-preemptively. There have been attempts to support priority-based GPU scheduling with preemption capabilities [6, 17, 31], but they require significant modifications to GPU access code, lack analytical support, and more importantly, may not work properly if the system has processes with unmodified GPU code or graphics applications due to the time-shared GPU context switching behavior of the device driver [4, 11].

In this paper, we address the aforementioned challenges and limitations by proposing GCAPS, GPU Context-Aware Preemptive Scheduling, for real-time GPU task execution in multi-core systems with analyzable guarantees. Our work focuses on Nvidia GPUs, especially those on Tegra System-on-Chips (SoCs) used in embedded platforms like Jetson Xavier and Orin running L4T (Linux for Tegra). The proposed approach works at the device driver level, and unlike existing techniques, they can protect the execution of real-time GPU processes from interference from best-effort non-real-time CUDA processes and graphics processes in the system. Specifically, compared to the existing approaches to enable GPU preemption, our approach requires minimum modifications to the user-level GPU access code, i.e., adding just one macro at the boundaries of GPU segments, but provides more fine-grained and efficient control of the GPU. This is particularly appealing to recent machine learning and computer vision applications as they are built on top of massive libraries that involve hundreds of different kernels. Thanks to the strictly preemptive and priority-driven GPU scheduling behavior, the proposed approach is analyzable and allows us to derive response-time tests for schedulability analysis.

In summary, the paper makes the following contributions:

- We propose a novel GPU context-aware preemptive priority-driven GPU scheduling approach for a multi-core system equipped with an Nvidia GPU. This approach not only enables GPU segments to be executed according to their task priority (especially important when task priority is assigned based on criticality), but also provides a way to assign different priorities to GPU segments, which yields a significant benefit in schedulability.
- We present a comprehensive analysis on the worst-case task response time under our proposed approach. In particular, our analysis considers both self-suspension and busy-waiting modes during GPU kernel execution, as well as the overhead caused by GPU context switching, which has been neglected in the literature. We also analyze the response time of a GPU-using task under the vanilla Nvidia Tegra GPU driver that follows a work-conserving round-robin policy.
- Our work is implemented on the latest Nvidia Tegra driver and will be open-sourced. [1]. Experimental results show that our approaches bring substantial benefits in taskset schedulability compared to previous synchronization-based approaches. A case study on Jetson Xavier and Orin platforms demonstrates the effectiveness of our work over the

[1] Available at `https://github.com/rtenlab/gcaps-super-repo`

default GPU driver and the applicability to various generations of GPU architectures.

2 Background on Tegra GPU Scheduling

Computational GPU workloads for Nvidia GPUs are often programmed using the CUDA library. These workloads are represented in *kernels* and user-level processes can launch kernels to the GPU at runtime. CUDA provides processes with *streams* to enable concurrent execution of kernels with a limited number of stream priority levels, e.g., only 2 in the Pascal architecture [30].

Since streams are bound to a user-level process that created them, the effect of stream scheduling and stream priority assignment is exerted only within each process boundary. The CUDA library is not a must for processes to access the GPU hardware. There are other low-level libraries for general-purpose GPU computing and graphics applications such as OpenCL and Vulkan. Programs built using different libraries co-exist in the system and they send GPU commands to the device driver.

Time-shared scheduling. At the device driver level, each process is associated with a *GPU context*, which represents a virtual address space and other runtime states on the GPU side. Any process accessing the GPU has a separate GPU context, regardless of whether it uses the CUDA library or not in the user space, and GPU contexts from different processes are time-sliced to share the GPU hardware.

To ensure fairness and prevent resource contention, the Tegra GPU driver uses a scheduling policy that assigns entries in the "runlist". [2] The entries of the runlist represent the allocation of time slices to TSGs (Time-Sliced Groups [5]) that are directly associated with processes. Each TSG has multiple "channels", each of which contains a stream of GPU commands received from its process. The runlist is populated with entries of TSGs, as depicted in Fig. 1. Each TSG entry maintains state attributes like the process ID, a list of channels, and the allocated time slice.

Runlist construction. The runlist is constructed by processes submitting commands to their respective TSG channels. Specifically, as commands are submitted, TSG entries are added to the runlist, which is managed under a mutex lock to prevent race condition. The device driver can assign priority to TSGs, and TSGs with higher driver-level priority are allocated larger time slices and more entries on the runlist. Following its construction, the runlist is scheduled by the GPU in a *round-robin* fashion, where each TSG entry's commands are executed for up to its time slice before moving to the next entry. This procedure continues until all commands of all active TSGs on the runlist have been executed.

As of this writing, there is no interface provided to the user to configure the length of time slices or the TSG priority setting. We have observed that the latest Tegra driver uses the same length of time slices for all TSGs, implying that the default scheduling policy of the driver focuses on ensuring fairness across different processes accessing the GPU.

GPU context switching. Moving from one TSG to another on the runlist causes GPU context switching. The major contributors to GPU context switching overhead are register file saving and cache flushing, the former taking much longer than CPUs due to the GPU's large register files [24].[3] In addition, extra delay may occur because Nvidia GPUs support

[2] In fact, there are multiple runlists but we refer to them as singular for simplicity. Each runlist corresponds to a specific hardware engine, such as copy engine or graphic engine. By default, every process maintains a single TSG entry on each runlist for storing the commands to be executed by that respective engine. This configuration does not impact the structure of our proposed design.

[3] Since the Pascal architecture, Nvidia GPUs use demand paging for GPU memory management. Thus,

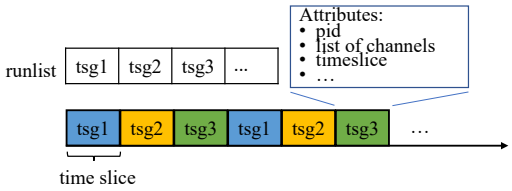


Figure 1 Runlist and time-sliced GPU scheduling

		No blocking	Task priority respected	Analyzable response time	Inter-GPU context
Prior Work	Unmanaged GPU (default driver)	×	×	✓	✓
	Sync.-based approaches [12–14, 20]	×	✓	✓	✓
	GPU partitioning [4, 16, 23, 26, 27, 29, 32]	✓	×	✓	×
	Preemptive GPU [6, 11, 15, 17, 31]	✓	✓	Unknown	✓
Ours	GCAPS	✓	✓	✓	✓

Table 1 Comparison of different GPU scheduling approaches

preemption at the pixel level for graphics tasks and the thread-block level for compute tasks [2]. In the case of data copy operations, data is divided into smaller chunks and preemption occurs at the boundary of each chunk [11]. Such delay is however very small compared to the length of GPU kernels, and for compute tasks, it can be separately measured or estimated by the maximum length of a single thread block among all kernels. Hence, we define the following term:

▸ **Definition 1** (GPU context switch overhead)**.** The GPU context switch overhead, θ, is the time required to switch from the GPU context of one process to that of another process, including all the aforementioned delay factors.

Prior work [11] reports that GPU context switching can take from 50 to 750 μs, which can be estimated by considering the GPU cache size and memory access latency. Our measurements in Sec. 7.2 show similar results, and our analysis in Sec. 6 accounts for this overhead with θ.

In summary, the Tegra GPU driver employs a *time-sliced round-robin* scheduling approach. This approach, however, does not respect the OS-level scheduling priority of processes, which is the main control knob to tune real-time performance in practice. This led to diminished responsiveness in high-priority real-time tasks whenever the system accommodated new low-priority or best-effort tasks. In addition, it is not easy for the user to observe such driver-level behavior because GPU profiling tools, such as Nvidia Nsight Systems, do not report GPU context switching events and each kernel execution time appears to be inflated with no time slice information. These issues contribute to difficulties in understanding and predicting the runtime behavior of GPU-enabled real-time systems.

the GPU memory of a context is not swapped out during GPU context switching, regardless of whether the GPU is integrated or discrete.

3 Related Work

Table 1 gives a summary of comparison between representative GPU scheduling approaches. Below we discuss prior work in various categories.

Synchronization-based GPU access control. Real-time synchronization protocols have played an important role in managing access to GPUs [12–14, 20]. With this approach, GPUs are modeled as mutually-exclusive shared resources and tasks are made to acquire locks to enter code segments accessing the GPUs, i.e., critical sections. MPCP [21] and FMLP+ [10] are prime examples for multi-core systems with GPUs and the use of such protocols enables analytically provable worst-case task response time bounds. However, as we discussed in Sec. 2, those works overlooked the inherent TSG context switching overhead. Another drawback is that the synchronization-based approach may suffer from blocking time from lower-priority tasks holding a lock and priority inversion caused by the priority boosting mechanism employed in these protocols [18]. This becomes particularly problematic when tasks busy-wait on long kernel execution, as discussed in [20].

Preemptive GPU scheduling. Several previous studies [6, 17, 31] have proposed software-based mechanisms to enable preemptive scheduling of real-time GPU tasks. These approaches introduce the concept of decomposing long-running GPU kernels into smaller blocks, allowing preemption to occur at the boundaries of these blocks. By enabling preemptive scheduling, the waiting time of high-priority tasks can be significantly reduced, improving responsiveness and offering a better chance to meet timing requirements. However, the cost of utilizing these mechanisms is not trivial as they necessitate a significant rewriting of user programs [6] or an implementation of a custom CUDA library with device driver modifications [6, 31]. Capodieci et al. [11] proposed a hypervisor-based technique to support preemptive Earliest Deadline First (EDF) GPU scheduling of virtual machines (VMs) in a virtualized environment. This approach achieves GPU performance isolation among VMs and shares some similarities with our work, in terms of controlling GPU context switching at the device driver level. However, it lacks consideration of the end-to-end response time of tasks involving CPU and GPU interactions, which is a specific focus of our work. Recently, Han et al. [15] proposed REEF, which enables microsecond-scale, reset-based preemption for concurrent DNN inferences on GPUs. This approach proactively kills and restarts best-effort kernels leveraging the idempotent nature of most DNN inference, but it is not applicable to a wide range of applications.

GPU partitioning. As a GPU is composed of multiple compute units, e.g., Streaming Multiprocessors (SMs) on Nvidia GPUs, there have been attempts to spatially partition the GPU and make them accessible by multiple real-time tasks in parallel [16, 23, 26, 27, 32]. They use SM-centric kernel transformation [29] to run kernels on their designated SMs/partitions. As this involves extensive program modifications and may suffer from misbehaving tasks, Bakita and Anderson [4] recently proposed a user-space library that minimizes program changes and offers much better usability and portability. With GPU partitioning, task performance is greatly affected by partitioning results, e.g., a high-priority task may suffer performance degradation due to the small number of SMs assigned to it or experience blocking if its SMs are shared with other tasks. In addition, all these approaches work within a single GPU context, i.e., one process; hence, multiple processes with separate contexts will still time-share the GPU, as discussed in Sec. 2. Note that our work does not compete with GPU partitioning techniques. They can be used within each process and our work enables predictable scheduling of GPU processes.

4 System Model

We consider a multi-core system with a GPU, which is common in today's embedded hardware platforms like Nvidia Jetson. The CPU has ω identical cores and the GPU is yet another processing resource used by compute-intensive tasks. The GPU consists of internal resources including Execution Engines (EEs) and Copy Engines (CEs). The EE and CE operations of a single process can be done asynchronously at runtime, and during pure GPU execution, the process can either busy-wait or self-suspend on the CPU. However, different processes cannot use the GPU at the same time because of the time-sharing scheduling of GPU contexts at the GPU device driver, as discussed before.

Task Model. We consider a taskset Γ consisting of n sporadic tasks (processes) with fixed priority and constrained deadlines.[4] Out of these, n^g tasks require GPU operations. Each task is assumed to be preallocated to one CPU core with no runtime migration, i.e., *partitioned multiprocessor scheduling*. The execution of a task is an alternating sequence of CPU segments and GPU segments. CPU segments run entirely on the CPU and GPU segments involve GPU operations such as memory copy and kernel execution. A task τ_i can be characterized as follows:

$$\tau_i := (C_i, G_i, T_i, D_i, \eta_i^c, \eta_i^g, \pi_i)$$

- C_i: the cumulative sum of the worst-case execution time (WCET) of all CPU segments of task τ_i.
- G_i: the cumulative WCET of GPU segments (including memory copies and kernels) of τ_i.
- T_i: the minimum inter-arrival time of each job of τ_i.
- D_i: the relative deadline of each job of τ_i, assumed to be smaller than or equal to the period, i.e., $D_i \leqslant T_i$.
- η_i^c: the number of CPU segments in each job of task τ_i.
- η_i^g: the number of GPU segments in each job of task τ_i; if τ_i does not use the GPU, $\eta_i^g = 0$.
- π_i: the priority of task τ_i.[5]

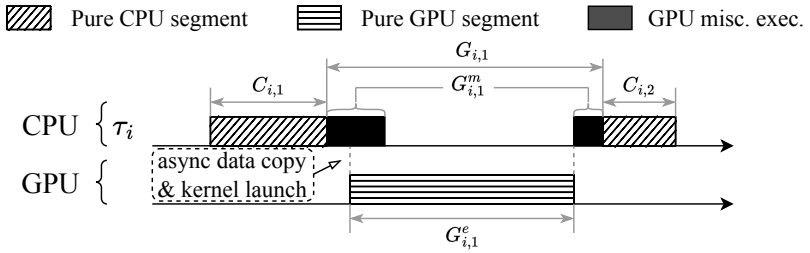


Figure 2 Task model

Fig. 2 depicts these parameters, and by default, in each task, all the segments have the same priority. Each GPU segment $G_{i,j}$[6] can be characterized as follows:

$$G_{i,j} := (G^m_{i,j}, G^e_{i,j})$$

[4] We assume tasks are processes and use them interchangeably in this paper.

[5] Our work allows the GPU segments of τ_i to run with a separate priority from its OS-level process priority to improve schedulability (explained later in Sec. 5.3). We will use π_i^c for CPU segment priority and π_i^g for GPU segment priority. If not specified, $\pi_i = \pi_i^c = \pi_i^g$.

[6] Although Tegra GPU supports zero-copy memory transfer, we still include the memory copy in the diagrams to illustrate the complete process. It is important to note that the choice of using zero-copy or not does not impact the analysis or the experimental results of this work.

Listing 1 Example usage of GCAPS macros

```
int task_function() {
    ...
    gcapsGpuSegBegin(fd, getpid()); /* GCAPS: GPU segment begin */
    cudaMemcpyAsync(d_in, h_in, mem_size_in, cudaMemcpyHostToDevice,
    stream));
    MyKernel<<<grid, threads, 0, stream>>>(d_in, d_out);
    cudaMemcpyAsync(h_out, d_out, mem_size_in, cudaMemcpyHostToDevice,
    stream));
    gcapsGpuSegEnd(fd, getpid()); /* GCAPS: GPU segment finish */
    ...
}
```

- $G^m_{i,j}$: the cumulative WCET of miscellaneous CPU operations in the j-th GPU segment of task τ_i, $G_{i,j}$.
- $G^e_{i,j}$: the WCET of GPU workload in $G_{i,j}$ that requires *no* CPU intervention such as data copy and kernel execution; and we call it a *pure GPU segment*.

$G^m_{i,j}$ is the time for launching a CUDA kernel, overhead for communicating with the GPU driver, and miscellaneous CPU operations for issuing other GPU commands. $G^e_{i,j}$ is the time for GPU data copy and kernel execution, during which task τ_i can either busy-wait or self-suspend on the CPU. Note that $G_{i,j} \leqslant G^m_{i,j} + G^e_{i,j}$ because the worst-case of G^m and G^e are not necessarily happening on the same control path and they may execute in parallel in asynchronous mode [20].

5 GCAPS: Priority-based Preemptive GPU Context Scheduling

This section presents our priority-based preemption GPU context-aware scheduling approach. It involves a set of user-level runtime macros that notify the GPU driver to update the runlist, and it provides fine-grained control over GPU segments.

5.1 GCAPS Algorithm

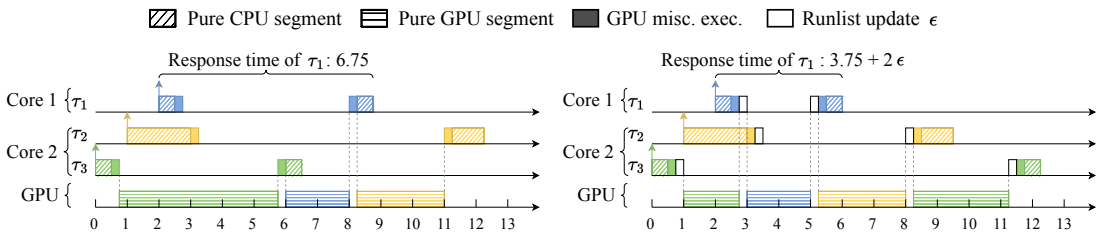
We first introduce the high-level scheduling procedures of GCAPS. To implement the approach, we add two macros that allow user programs to indicate the beginning and completion of a GPU segment. When the macro is called, it generates an IOCTL command and sends it to the GPU driver through a file descriptor, and requests the driver to update the runlist accordingly. While we chose IOCTL as a way to interact with the driver, other methods could be used as well, such as system calls and `procfs`/`sysfs` interfaces.

The macros introduced are `gcapsGpuSegBegin()` and `gcapsGpuSegEnd()`, which are wrappers to our IOCTL syscalls. A sample user program is listed in Listing. 1. The code between them is a GPU segment. With the help of these two macros, we can let our driver-level approach know the boundaries of GPU segments and make GPU scheduling decisions at the right time.

An IOCTL command issued by the macro triggers our TSG scheduler shown in Alg. 1. To keep track of which tasks are in the runlist and which tasks are pending, two bitfield lists, *task_running* and *task_pending*, are maintained in the GPU driver. When a caller task τ_i notifies that it begins its GPU segment through `gcapsGpuSegBegin()`, the scheduler first checks whether τ_i is a real-time task by checking whether the `rt_priority` field of the task's `task_struct` is set. If it is not, the scheduler checks whether there is any real-time task that is currently running and decides whether to add τ_i to the runlist or add it to the pending list (line 6 to 10). If τ_i is a real-time task, the scheduler checks the priority of τ_i relative to the currently-running highest-priority task τ_h ($\tau_h \in task\_running \wedge \tau_h \neq \tau_i$). If the priority

Algorithm 1 Priority-based TSG scheduling

```
1: task_pending = ∅
2: task_running = ∅
3:                                   ▷ Note that a task exclusively exists in one of these two lists
4: procedure TSG_SCHEDULER(τ_i, add)
5:     if add then                                                    ▷ τ_i requests to be added
6:         if τ_i is not a real-time task then
7:             if no real-time task is in task_running then
8:                 Add τ_i to task_running
9:             else
10:                Add τ_i to task_pending
11:        else                                                       ▷ τ_i is a real-time task
12:            τ_h ← the highest-priority task in task_running
13:            if τ_i.rt_priority > τ_h.rt_priority then
14:                Add τ_i to task_running
15:                Move τ_h to task_pending
16:            else
17:                Add τ_i to task_pending
18:    else                                                           ▷ τ_i requests to be removed
19:        τ_k ← the highest-priority RT task in task_pending
20:        if τ_k exists then
21:            Move τ_k to task_running
22:            Remove τ_i from task_running
23:        else                                                       ▷ no pending real-time task
24:            task_running ← task_pending
25:            task_pending ← ∅
26:    Add all TSGs of tasks in task_running to the runlist
```

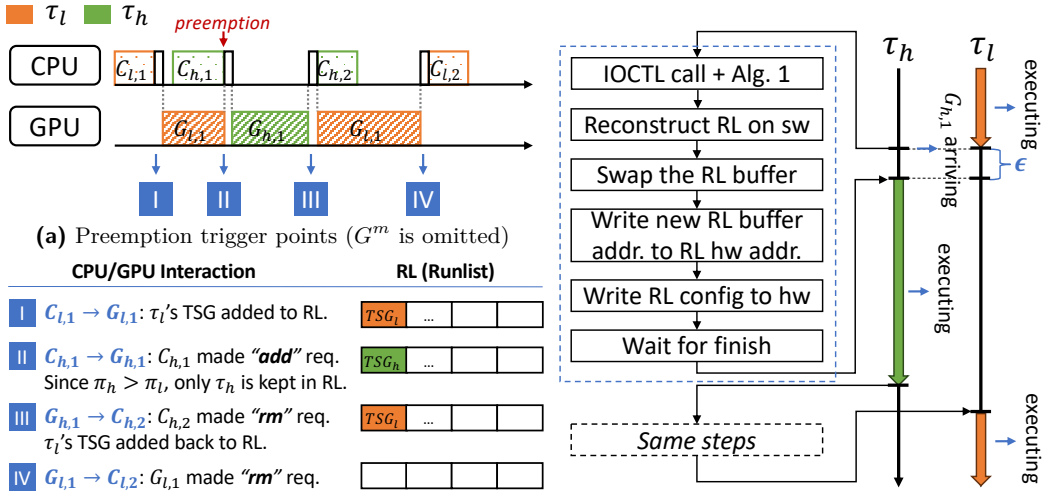


(a) Synchronization-based approach **(b)** Proposed approach

Figure 3 Example schedule of three tasks under different approaches (priority $\tau_1 > \tau_2 > \tau_3$)

of τ_i is higher than τ_h, the scheduler preempts the GPU execution of τ_h and moves it to the pending list, and τ_i is added to the runlist. Otherwise, τ_i is added to the pending list (line 11 to 17). If τ_i notifies the driver about the completion through `gcapsGpuSegEnd()`, the scheduler first finds the highest-priority task τ_k in the pending list. If τ_k exists, it is added to the runlists. Otherwise, if there are only best-effort tasks, the scheduler adds all of them to the runlist to resume their progress in a time-shared manner (line 18 to 25). At the end of the scheduler, it directly updates the driver's runlist based on *task_running* such that the TSGs of tasks are dispatched and GPU context switching takes place immediately. To implement the scheduler, we only added about 300 lines of code in the driver, and 10 lines of code in each userspace macro.

▸ **Example 1** (Motivational example)**.** Figs. 3 compares task schedules under the conventional synchronization-based approach and our proposed approach. τ_1 is running on Core 1, while τ_2 and τ_3 are running on Core 2. The synchronization-based approach shown in Fig. 3a treats the entire execution of a GPU segment as a critical section. Tasks are serviced in order based on their task priorities. This approach ensures that each task completes its GPU segments in a deterministic and predictable manner. However, τ_1 is delayed by the GPU segments of all of its lower-priority tasks and gets a response time of 6.75.



(a) Preemption trigger points (G^m is omitted)

(b) GPU/CPU Interaction of Fig. 4a

(c) Context Switching Details

Figure 4 Preemption by GPU segments and GPU context switching

On the other hand, our approach avoids this delay by allowing preemption during GPU segment execution. Fig. 3b shows an example schedule under the proposed approach, and the overhead introduced by our approach is denoted as ϵ, defined as follows.

▸ **Definition 2** (Runlist update delay)**.** The runlist update delay, ϵ, is defined as the sum of the time it takes to complete our TSG scheduler (represented by α, including the cost for IOCTL system call, TSG scheduling algorithm, and runlist update) and the resulting GPU context switching overhead (θ). Hence, $\epsilon = \alpha + \theta$.

Unlike the synchronization-based approach, at $t = 3 - \epsilon$, τ_1's GPU segment issues an IOCTL syscall to notify the driver that its GPU segment is ready to run. It causes preemption of τ_3's GPU segment by removing its associated TSGs from the runlist, and the GPU is solely occupied by the highest-priority task in the system, τ_1, until it completes. The response time of τ_1 is $3.75 + 2\epsilon$, much smaller than that of the synchronization-based approach. This strategy is followed in the remaining schedule.

5.2 GPU Context Switching Details

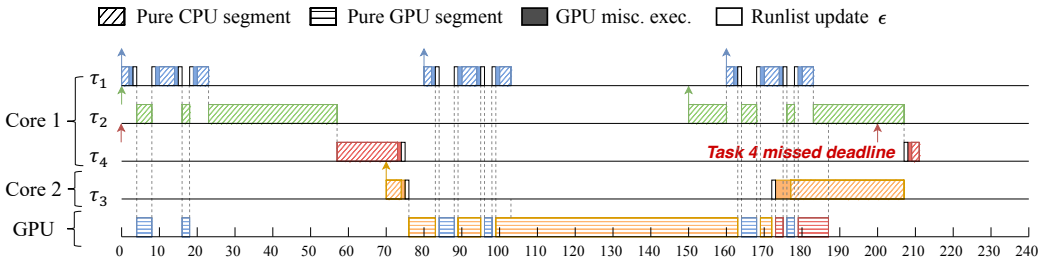
This section details the context switching of GPU segments when preemption is triggered. Fig. 4a shows a schedule of two real-time tasks, where preemption points are marked in red and the CPU and GPU interactions are marked in blue. Fig. 4b illustrates the runlist status at each CPU/GPU interaction. Note that, even if the active TSGs of a task are removed from the runlist, they are kept in the scheduler data structure of the GPU driver and won't be lost; hence, we can add those TSGs back to the runlist, e.g., $G_{h,1} \rightarrow C_{h,2}$ in Fig. 4b, to resume their execution.

Fig. 4c illustrates the detailed steps of GPU context switching between τ_l and τ_h. The procedures outlined within the dashed blue block represent the entire cost of preemption (ϵ). After the IOCTL system call is invoked, Alg. 1 is executed in the driver and it identifies the ids of all the TSGs for the runlist. To manage concurrent system calls efficiently in our proposed approach, we replace the default mutex lock in the driver with a real-time mutex, `rt_mutex` [22], to reduce the blocking time as well as prevent priority inversion.

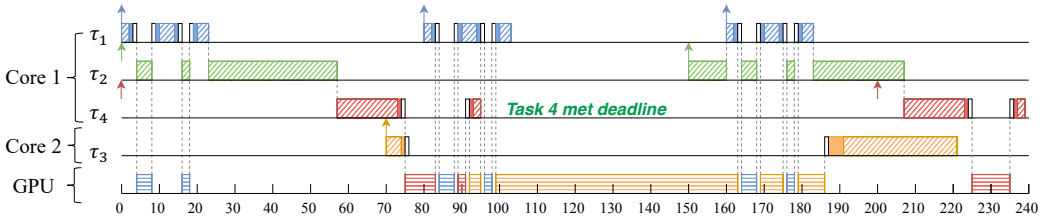
Once the new runlist is constructed, the scheduler swaps it with the one currently held by hardware. This is similar to the well-known double buffering technique. The scheduler allocates runlist buffers in DMA memory during the driver initialization phase. The size of

Task	CPU	$T_i = D_i$	CPU Segments	GPU Segments
τ_1	1	80	$C_{1,1} = 2, C_{1,2} = 4, C_{1,3} = 3$	$G^m_{1,1} = 2, G^e_{1,1} = 4,\ G^m_{1,2} = 2, G^e_{1,2} = 2$
τ_2	1	150	$C_{2,1} = 40$	-
τ_3	2	190	$C_{3,1} = 4, C_{3,2} = 30$	$G^m_{3,1} = 5, G^e_{3,1} = 80$
τ_4	1	200	$C_{4,1} = 16, C_{4,2} = 2$	$G^m_{4,1} = 2, G^e_{4,1} = 10$

Table 2 Taskset used in Fig. 5



(a) CPU priority = GPU priority: $\tau_1 > \tau_2 > \tau_3 > \tau_4$. RT test failed.



(b) CPU priority: $\tau_1 > \tau_2 > \tau_3 > \tau_4$; GPU priority: $\tau_1 > \tau_2 > \tau_4 > \tau_3$. RT test passed.

Figure 5 Example schedule of assigning separate GPU priority under self-suspension mode

each buffer corresponds to the product of the entry size and the number of entries, which depend on hardware capabilities. For instance, on Nvidia Jetson Xavier NX, each entry is 16 bytes with 65535 entries in total. Then the runlist buffer to use is submitted to the hardware. It involves writing the new runlist buffer address to the runlist's hardware address and writing the runlist configurations to the hardware registers. During the submission, the driver polls the hardware and waits until it finishes. After these operations, the new runlist only contains the TSGs of τ_h in Fig. 4c, and τ_h can run in isolation without interference from τ_l.

5.3 Separate Priority for GPU Segments

Under the proposed approaches, GPU segments are executed following their OS-level task priorities by default, and the preemption can occur at segment boundaries.

To improve taskset schedulability, we can assign separate priority to the GPU segments of a task, different from its CPU priority. In this case, a task τ_i's CPU segments and GPU segments may have different priorities, denoted as π^c_i and π^g_i respectively. Note that in our approach, the segments of the same type have the same priority (either π^c_i or π^g_i).

We adopt Audsley's approach for this purpose [3]. Hence, if the schedulability test given in Section 6 determines a taskset is unschedulable, we iterate through all tasks from the lowest to the highest CPU priority and check whether each priority level can be assigned to the GPU segments of a task without causing the taskset to fail the schedulability test. Allowing different priorities for CPU and GPU segments may cause a deadlock if their priorities are not coordinated. To prevent deadlocks, we maintain the relative priority order of GPU segments identical to their corresponding CPU segments (i.e., OS-level priorities) for tasks executing on the same CPU core. For instance, consider two tasks τ_1 and τ_2 assigned to the same CPU, with CPU priority $\pi^c_1 > \pi^c_2$. If our algorithm suggests a GPU priority

order where $\pi_1^g < \pi_2^g$, we treat this assignment as infeasible.

▸ **Example 2** (Effect of separate priority for GPU segment)**.** Consider the taskset in Table 2. Task priority is assigned by the Rate-Monotonic (RM) policy, and CPUs are assigned by Worst-Fit-Decreasing (WFD) heuristic. The lower the task index, the higher its priority. The first job of τ_3 arrives at time 70, and the other tasks' jobs arrive at time 0. In Fig. 5a, each task uses the same priority for its CPU and GPU segments. During the time window $t = [73, 74]$, both $G^e_{3,1}$ and $G^e_{4,1}$ were ready to begin. Since $G^e_{3,1}$ has a higher priority, it gained GPU access. When $G^e_{4,1}$ finished at $t = 187$, $C_{4,2}$ could not start because τ_2 is running on the same CPU. Those delays caused τ_4 to miss the deadline.

In Fig. 5b, the priorities of tasks' CPU segments remain unchanged but we swapped the GPU priority of τ_3 and τ_4. Unlike the schedule in Fig. 5a, at $t = 75$, $G^e_{4,1}$ obtained GPU access first and τ_4 is able to meet the deadline.

This timeline illustrates only one possible scenario. We evaluated this taskset using the proposed response time analysis detailed in Sec. 6.3. The results show that the priority assignment in Fig. 5a failed the test while the priority assignment in Fig. 5b passed the test.

To implement this approach, we allow our macros to take one extra argument, which is the user-defined GPU segment priority: `gcapsGpuSegBegin(fd, getpid(), gprio)` and `gcapsGpuSegEnd(fd, getpid(), gprio)`. In Alg. 1, we change line 12, line 13 and line 19 to decide whether a task's TSG should be in the runlist or not based on tasks' GPU segment priorities, i.e., `gprio`.

6 End-to-End Response Time Analysis

This section presents a comprehensive analysis of the end-to-end response time of tasks involving CPU and GPU interactions for both the round-robin approach of the default Nvidia Tegra driver and our proposed priority-based GPU context scheduling approach. The applicability of the proposed analysis is not limited to integrated GPUs with shared memory architecture such as Jetson series. As explained in Sec. 5.2, only the runlist buffer is swapped during context switches and the GPU memory of a preempted task is not swapped out.

6.1 Response Time Breakdown

Before proceeding to the individual analysis for each scheduling approach, we first present the components that account for the overall response time. For a task τ_i, its worst-case response time can be upper-bounded by:

$$R_i := C_i + G_i + I_i^C + I_i^G \tag{1}$$

C_i and G_i stand for the computation requirement of τ_i's CPU and GPU segments. I_i^C and I_i^G are the interference τ_i can experience due to CPU segments and GPU segments, respectively. The interference due to CPU segments, I_i^C, consists of two main components: (i) preemption from higher-priority tasks on the same core (P_i^C), and (ii) blocking due to runlist updates in our approach (B_i^C). These components will be explained later in Sec. 6.3. The total CPU interference is expressed by:

$$I_i^C := P_i^C + B_i^C \tag{2}$$

As for the interference from GPU segments, I_i^G, we break it down into the following components:

Direct Preemption (I_i^{dp}). *Direct preemption* occurs under the proposed approach when a task's GPU segment execution gets preempted by a higher-priority GPU segment, regardless of whether the two tasks are running on the same CPU core or not. As an example, Fig. 6a shows that τ_2 gets direct preemption from τ_1 under the proposed approach, since they are contending for the GPU resource.

Indirect Delay (I_i^{id}). *Indirect delay* refers to the delay imposed on the CPU segments of a task τ_i due to busy-waiting GPU segments.[7] Such busy-waiting is caused by higher-priority tasks on the same CPU as τ_i, and the amount of indirect delay imposed on τ_i is also affected by GPU-using tasks on different CPU cores. However, indirect delay cannot exist stand-alone. It is contingent on the presence of direct interference (i.e. direct CPU or GPU preemption) from a higher-priority GPU-using ($\eta_h^g > 0$) task τ_h running on the same core as τ_i.

▸ **Example 3** (Indirect delay). For the proposed approach, the CPU segment of τ_3 in Fig. 6a shows an example: τ_1 preempts τ_2's GPU execution, making τ_2's busy waiting period longer, and it further delays τ_3's execution. In other words, τ_1 indirectly delays τ_3 since they are not directly competing for the same resource. For the default scheduling approach, consider the taskset in Fig. 6b, with priority of $\tau_2 > \tau_1 > \tau_3$. In this case, τ_3 is delayed by the GPU interleaved execution of τ_1 and τ_2; thus τ_3 is indirectly delayed by τ_1 since τ_1 and τ_3 are not competing for the same resource.

Interleaved Execution (I_i^{ie}). The default round-robin scheduling approach adopts a scheduling strategy of *interleaved execution* where multiple GPU kernels are executed in an alternating, overlapped manner, instead of completing one before starting the next, as discussed in Sec. 2. With this approach, a TSG of a task has to wait for the completion of the preceding other tasks' TSGs before it starts, leading to extended execution time of GPU segments, which are perceived as the prolonged boxes on the timeline in profiling tools. In the worst case, all the TSGs are active and fully utilize the time slices, and each GPU segment of a task τ_i must wait for prior TSGs to finish. This procedure repeats throughout the entire execution of τ_i. To model this delay, we consider a TSG time slice length of L, and $G_{i,j}^e$ requires at most $\lceil \frac{G_{i,j}^e}{L} \rceil$ times of TSG slices and GPU context switches. Assuming there are ν GPU-using tasks in the system, the maximum delay for each GPU segment of τ_i due to interleaved execution is upper bounded by:

$$\mathcal{I}(\nu, G_{i,j}^e) := (L + \theta) \cdot \nu \cdot \lceil \frac{G_{i,j}^e}{L} \rceil \tag{3}$$

By considering all the above factors, the overall GPU interference of a task τ_i can be computed as follows:

$$I_i^G := I_i^{dp} + I_i^{id} + I_i^{ie} \tag{4}$$

6.2 Analysis for Default Round-Robin TSG Scheduling

In the prior literature, the tasks' response times are typically considered unpredictable under the default scheduling policy in the GPU driver ([12–14, 18, 20]), and also the tasks' execution is graphed as large overlapped boxes in the GPU profiling tools. However, as we have revealed

[7] I_i^{id} is not about the context switching overhead θ (Def. 1), which includes register saving/restoring cost and extra delay due to pixel-level or thread block-level preemption granularity as discussed in Sec. 2. The overhead θ (and $\epsilon = \alpha + \theta$ in our approach) will be discussed separately.

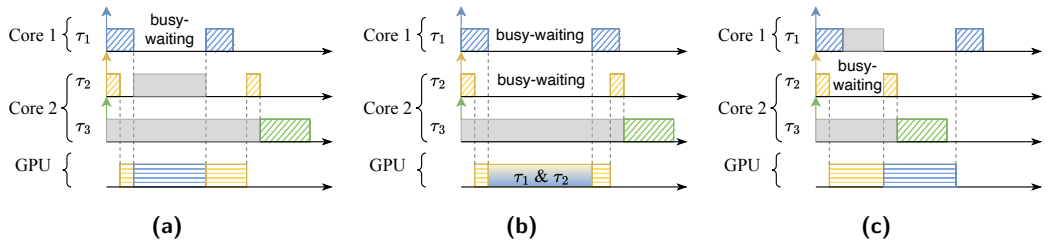


Figure 6 GPU Segment Interference with busy-waiting (G^m and ϵ are omitted for simplicity). The shaded area indicates the amount of preemption or blocking. (a) Proposed preemptive scheduling: $(\pi_i^c = \pi_i^g)$ $\pi_1 > \pi_2 > \pi_3$. (b) Default round-robin scheduling: τ_1 indirectly delays τ_3 regardless of the relative priority of τ_1 and τ_3. (c) Proposed preemptive scheduling with separate GPU segment priority assignment: $\pi_1^c > \pi_2^c > \pi_3^c$; $\pi_2^g > \pi_1^g > \pi_3^g$.

that the Tegra GPU driver employs a time-sliced round-robin scheduling approach in Sec. 2, this makes this approach analyzable. In the following analysis, we assume that the default behavior of the Tegra driver that equally treats all GPU requests from different processes. This is a reasonable assumption given that the TSG slice and priority configurations are not exposed to the user and we have not observed changes in those settings, as we discussed before.

▸ **Lemma 1** (GPU interleaved execution). *Under the default Tegra GPU driver, the worst-case interference from GPU interleaved execution for a task τ_i is bounded by:*

$$I_i^{ie} = \sum_{j=1}^{\eta_i^g} \mathcal{I}(|\{k \mid \tau_k \neq \tau_i \wedge \eta_k^g > 0\}|, G_{i,j}^e) \tag{5}$$

Proof. This equation captures the total amount of interference due to the interleaved execution of any other tasks τ_k and τ_i itself. ◂

▸ **Lemma 2** (GPU direct preemption). *Under the default Tegra GPU driver, the GPU direct preemption delay imposed on a real-time task τ_i is zero, i.e., $I_i^{dp} = 0$.*

Proof. This follows our definition of direct preemption in Sec. 6.1. With the default driver's approach, GPU segments are not preempted, but interleaved. ◂

▸ **Lemma 3** (CPU blocking time). *Under the default Tegra GPU driver, the worst-case CPU blocking time for a task τ_i is zero, i.e., $B_i^C = 0$.*

Proof. This is obvious since the default driver does not require tasks to explicitly request for the runlist update. ◂

6.2.1 Busy-Waiting Mode

▸ **Lemma 4** (GPU indirect delay). *Under the default Tegra GPU driver with busy-waiting, the worst-case interference from GPU indirect delay for a task τ_i is bounded by:*

$$I_i^{id} = \sum_{\tau_h \in hpp(\tau_i) \wedge \eta_h^g > 0} \lceil \frac{R_i}{T_h} \rceil \cdot \sum_{j=1}^{\eta_h^g} \mathcal{I}(|\{k \mid \tau_k \notin hpp(\tau_i) \wedge \eta_k^g > 0 \cup \tau_h\}|, G_{h,j}^e) \tag{6}$$

where $hpp(\tau_i)$ is the set of higher-priority tasks running on the same CPU core as τ_i.

Proof. A task τ_i experiences indirect delay from each higher-priority task τ_h busy waiting on the same CPU during its GPU segment execution ($\tau_h \in hpp(\tau_i) \wedge \eta_h^g > 0$ in the outer summation). For each τ_h, we need to bound the maximum busy-waiting period during which its GPU execution can interleave with any other GPU-using tasks. This can be done by the summation of Eq. (3) for each GPU segment of τ_h, $G_{h,j}^e$. However, since the busy-waiting periods of tasks in $hpp(\tau_i)$, other than τ_h, have already been accounted for iteratively in the outer summation, we exclude $hpp(\tau_i)$ from the cardinality of the interleaving taskset considered by Eq. (3), i.e., $|\{k \mid \tau_k \notin hpp(\tau_i) \wedge \eta_k^g > 0 \cup \tau_h\}|$, to prevent double counting. ◂

▸ **Lemma 5** (CPU preemption). *Under the default Tegra driver with busy-waiting, the worst-case interference from CPU preemption is bounded by:*

$$P_i^C = \sum_{\tau_h \in hpp(\tau_i)} \lceil \frac{R_i}{T_h} \rceil \cdot (C_h + G_h^m) \tag{7}$$

Proof. A task τ_i can only experience interference from CPU preemption from the higher-priority tasks running on the same CPU core for a duration of $C_h + G_h^m$. ◂

6.2.2 Self-Suspension Mode

▸ **Lemma 6** (GPU indirect delay). *Under the default Tegra GPU driver with self-suspension, the worst-case interference from GPU indirect delay for a task τ_i is zero, i.e., $I_i^{id} = 0$.*

Proof. A task with self-suspension would not experience GPU indirect delay as explained in Sec. 6.1. ◂

▸ **Lemma 7** (CPU preemption). *Under the default Tegra driver self-suspension, the worst-case interference from CPU preemption is bounded by:*

$$P_i^C = \sum_{\tau_h \in hpp(\tau_i)} \lceil \frac{R_i + J_h^c}{T_h} \rceil \cdot (C_h + G_h^m) \tag{8}$$

where $J_h^c = R_h - (C_h + G_h^m)$.

Proof. Each job of τ_h imposes a delay of up to $C_h + G_h^m$ on τ_i and the self-suspending behavior of τ_h can be captured by a jitter term J_h^c as reported in [9]. ◂

6.3 Analysis for Proposed GPU Context Scheduling

Our approach introduces the runlist update delay of ϵ (Def. 2). In the worst case, runlist updates are required both before and after each GPU segment of τ_i, since the associated TSGs need to be added and removed by IOCTL calls as shown in Listing 1. This leads to a cumulative cost of $2\epsilon \cdot \eta_i^g$ for the entire job of τ_i. For ease of presentation, we define G_i^*, G_i^{e*}, and G_i^{m*} to incorporate the two times of runlist updates into the execution requirements.

$$G_i^* = G_i + 2\epsilon \cdot \eta_i^g \text{ , } G_i^{e*} = G_i^e + 2\epsilon \cdot \eta_i^g \text{ and } G_i^{m*} = G_i^m + 2\epsilon \cdot \eta_i^g$$

Also, due to the use of `rt-mutex` in our approach, each GPU segment of τ_i can experience blocking time of ϵ when there is an ongoing runlist update initiated by a lower-priority task.

▸ **Lemma 8** (Blocking time). *Under the proposed approach, the worst-case blocking time for a task τ_i is bounded by:*

$$B_i^C = (\eta_i^g + 1) \cdot \epsilon \tag{9}$$

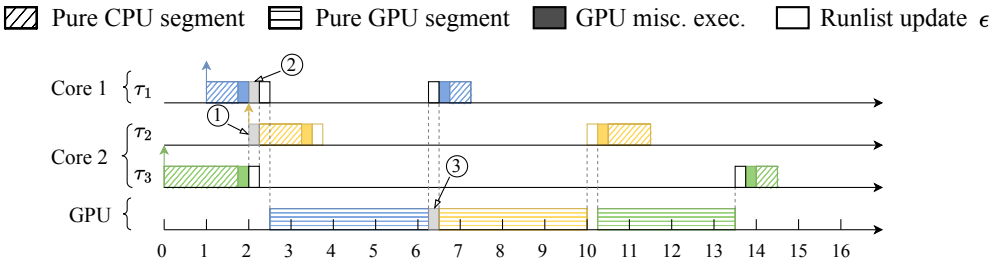


Figure 7 Example schedule of three tasks ($\pi_1 > \pi_2 > \pi_3$) with runlist update delay

Proof. First, at least one time of the blocking of ϵ applies to every τ_i, regardless of whether it is a GPU-using or CPU-only task. This is because it can experience blocking from GPU-using lower-priority tasks at the very beginning of its job instance, as illustrated by ① in Fig. 7. If τ_i is a GPU-using task, each GPU segment requires up to ϵ for potential blocking from lower-priority tasks, which results in $\eta_h^g \cdot \epsilon$. Therefore, the total amount of blocking imposed by lower-priority tasks is bounded by $(\eta_i^g + 1) \cdot \epsilon$. ◂

▸ **Lemma 9** (Interleaved execution). *Under the proposed approach, the interference from interleaved execution of a real-time task τ_i is zero, i.e., $I_i^{ie} = 0$.*

Proof. This is obvious since the GPU segments of real-time tasks are not allowed to execute in an interleaved manner with our proposed approach. ◂

▸ **Example 4** (Runlist update delay). Fig. 7 can help better understand all types of runlist update delay under the proposed approach. The task of interest here is τ_2 which runs with medium priority. τ_3 triggers runlist update first, and this blocks to τ_2's CPU segment at ① and τ_1's runlist update at ②. Before τ_1 finishes GPU execution, τ_2 cannot start its GPU segment as τ_1 is actively using the GPU with higher priority than τ_2. Then, the start time of τ_2's GPU segment is delayed by the runlist update at ③ triggered by τ_1 to remove τ_1's TSG from the runlist.

Based on these observations, we derive the response time analysis of busy-waiting and self-suspending GPU tasks in the following.

6.3.1 Busy-Waiting Mode

▸ **Lemma 10** (GPU direct preemption). *Under the proposed approach with busy-waiting, the worst-case interference from GPU direct preemption for a task τ_i is bounded by:*

$$I_i^{dp} = \sum_{\substack{\tau_h \in hpp(\tau_i) \\ \wedge \eta_h^g > 0 \wedge \eta_i^g > 0}} \lceil \frac{R_i}{T_h} \rceil \cdot G_h^{e*} + \sum_{\substack{\tau_h \in hp(\tau_i) \wedge \tau_h \notin hpp(\tau_i) \\ \wedge \eta_h^g > 0 \wedge \eta_i^g > 0}} \lceil \frac{R_i + J_h^g}{T_h} \rceil \cdot G_h^{e*} \tag{10}$$

where $J_h^g = R_h - G_h^e$.

Proof. A task τ_i can experience direct GPU preemption from a higher-priority task τ_h only when both τ_i and τ_h are GPU-using tasks ($\eta_i^g > 0 \wedge \eta_h^g > 0$). We consider the preemption from: (i) τ_h on the same core as τ_i ($\tau_h \in hpp(\tau_i)$), and (ii) on different cores ($\tau_h \in hp(\tau_i) \wedge \tau_h \notin hpp(\tau_i)$). In both of the cases, the duration of preemption equals to G_h^{e*}, which includes the pure GPU execution and the runlist update cost of 2ϵ for each of τ_h's GPU segment.

In the case of (ii), a release jitter J_h^g is considered to account for the carry-in effect of τ_h's GPU execution. This is because τ_h's job releases are not synchronized with the τ_i's job

release, causing carry-in jobs to the window of τ_i's response time. It is known that in an arbitrary time window t, the number of arrivals of a higher-priority task τ_h with a carry-in job can be upper-bounded by $\lceil \frac{t+J_h}{T_h} \rceil$, where the jitter $J_h = D_h - C_h$ if τ_h's response time is unknown [7], and $J_h = R_h - C_h$ otherwise [19]. ◂

▸ **Lemma 11** (GPU indirect delay). *Under the proposed approach with busy-waiting, the worst-case interference from GPU indirect delay for a task τ_i is bounded by:*

$$I_i^{id} = \sum_{\substack{\tau_h \in hp(\tau_i) \wedge \tau_h \notin hpp(\tau_i) \\ \wedge \eta_h^g > 0 \wedge \eta_i^g = 0}} \lceil \frac{R_i + J_h^g}{T_h} \rceil \cdot G_h^{e*} \tag{11}$$

Proof. Under the proposed approach with busy-waiting, a task τ_i gets GPU indirect delay when it is preempted by a higher-priority task τ_h on the same core when τ_h is busy waiting on the CPU during its GPU segment execution. During this busy-waiting period, the higher-priority task τ_h can experience GPU direct preemption from any higher-priority GPU-using tasks on different CPUs ($\tau_h \in hp(\tau_i) \wedge \tau_h \notin hpp(\tau_i) \wedge \eta_h^g > 0$), and such τ_h further increases indirect delay imposed on τ_i. This happens regardless of whether τ_i is a CPU-only or GPU-using task. However, the last term in Eq. (10) has already bounded all the delays from such τ_h that τ_i may experience when τ_i is a GPU-using task. Therefore, in this lemma, we only consider the case that τ_i is a CPU-only task, to prevent double counting. ◂

▸ **Lemma 12** (CPU preemption). *Under the proposed approach with busy-waiting, the worst-case interference from CPU preemption of a task τ_i is bounded by:*

$$P_i^C = \sum_{\tau_h \in hpp(\tau_i)} \lceil \frac{R_i}{T_h} \rceil \cdot (C_h + G_h^m) \tag{12}$$

Proof. The proof directly follows Lemma 5. ◂

6.3.2 Self-Suspension Mode

▸ **Lemma 13** (GPU direct preemption). *Under the proposed approach with self-suspension, the worst-case interference from GPU direct preemption for a task τ_i is bounded by:*

$$I_i^{dp} = \sum_{\substack{\tau_h \in hpp(\tau_i) \\ \wedge \eta_h^g > 0 \wedge \eta_i^g > 0}} \lceil \frac{R_i + J_h^g}{T_h} \rceil \cdot G_h^e + \sum_{\substack{\tau_h \in hp(\tau_i) \wedge \tau_h \notin hpp(\tau_i) \\ \wedge \eta_h^g > 0 \wedge \eta_i^g > 0}} \lceil \frac{R_i + J_h^g}{T_h} \rceil \cdot G_h^{e*} \tag{13}$$

Proof. This is a variant of Lemma 10. The difference lies in the first term accounting for the interference of GPU direct preemption from higher-priority GPU-using task τ_h on the same core. τ_h imposes interference of its pure GPU execution, G_h^{e*}, to τ_i only when τ_i uses the GPU ($\eta_i^g > 0$). Here, from the perspective of τ_i with $\eta_i^g > 0$, the runlist update delay on the CPU and GPU overlaps, and thus using G_h^e safely bounds GPU preemption from τ_h. The self-suspending behavior of τ_h can be captured by a jitter term, as reported in [9]. The second term remains the same as in Lemma 10. ◂

▸ **Lemma 14** (GPU indirect delay). *Under the proposed approach with self-suspension, the worst-case interference from GPU indirect delay of a task τ_i is zero, i.e., $I_i^{id} = 0$.*

Proof. This phenomena does not exist under self-suspension mode as explained in Sec. 6.1. ◂

▸ **Lemma 15** (CPU preemption). *Under the proposed approach with self-suspension, the worst-case interference from CPU preemption of a task τ_i is bounded by:*

$$P_i^C = + \sum_{\substack{\tau_h \in hpp(\tau_i) \\ \wedge \eta_h^g = 0}} \lceil \frac{R_i}{T_h} \rceil \cdot C_h + \sum_{\substack{\tau_h \in hpp(\tau_i) \\ \wedge \eta_h^g > 0}} \lceil \frac{R_i + J_h^c}{T_h} \rceil \cdot (C_h + G_h^{m*}) \tag{14}$$

Proof. This is a variant of Lemma 12. If τ_h is a GPU-using task running on the same core ($\tau_h \in hpp(\tau_i) \wedge \eta_h^g > 0$), each job of τ_h imposes a delay of up to $(C_h + G_h^{m*})$ and the self-suspending behavior of τ_h is accounted for by the jitter term, J_h^c [9]. ◂

6.4 Analysis for GPU Priority Assignment

When the GPU priority assignment given in Sec. 5.3 is used, the amount of preemption due to higher-priority GPU tasks, i.e., *hpp*() and *hp*(), needs to be revisited. Recall that our assignment preserves the same relative priority order for GPU segments as the CPU priority order for tasks running on the same core. The meaning of *hpp*() therefore remains unchanged. However, *hp*() needs to be redefined such that it means the set of tasks with higher "GPU segment" priorities in the system. This is because any interference due to GPU segment execution (I_i^{dp} and I_i^{id}) of Lemma 10, 11 and 13 are now governed by GPU segment priorities. When computing the release jitter J_h^x, R_h needs to be replaced with D_h since the worst-case response time of higher-priority tasks is unknown when applying our GPU priority assignment method. With these simple modifications, our analysis in the previous section can analyze the effect of the GPU priority assignment.

Besides the benefits introduced in Example 2, the use of separate GPU segment priority assignment is particularly effective in mitigating the scheduling inefficiency of busy-waiting as shown in Example 5. Our evaluation results in Sec. 7.1 will confirm this claim.

▸ **Example 5** (Separate GPU priority assignment under busy-waiting mode). In Fig. 6a, τ_3 is indirectly preempted by τ_1 as explained in Sec. 6.1. However, if we consider Fig. 6c where τ_2 is assigned a higher GPU priority than τ_1, τ_3 no longer experiences the delay from τ_1's GPU segment, thereby achieving a shorter response time.

7 Evaluation

We conduct schedulability experiments to compare the proposed approaches against prior work and assess the effect of the GPU priority assignment. Then, we present a case study on two Nvidia embedded platforms.

7.1 Schedulability Experiments

We generated 1,000 random tasksets for each experimental setting based on the parameters in Table 3. The parameter selection is inspired by the prior work [20], with slight modifications to increase the system load. Based on the measurement in Sec. 7.2, we aggressively set ϵ to 1 ms for our approaches, while assuming zero overhead for synchronization-based approaches and setting θ as low as 200μs for TSG context switching in the default round-robin scheduling. For each task in a taskset, the number of tasks on each CPU is first chosen randomly within the range, and the utilization per CPU is generated based on the UUniFast algorithm [8]. Then for each task, its period and the number of GPU segments are uniformly randomized within the given range. Then the parameters for each segment are determined. Task priority is assigned by the Rate Monotonic (RM) policy. After this, we re-allocate the tasks to the CPUs for load balancing purpose with WFD (worst-fit decreasing) heuristic.

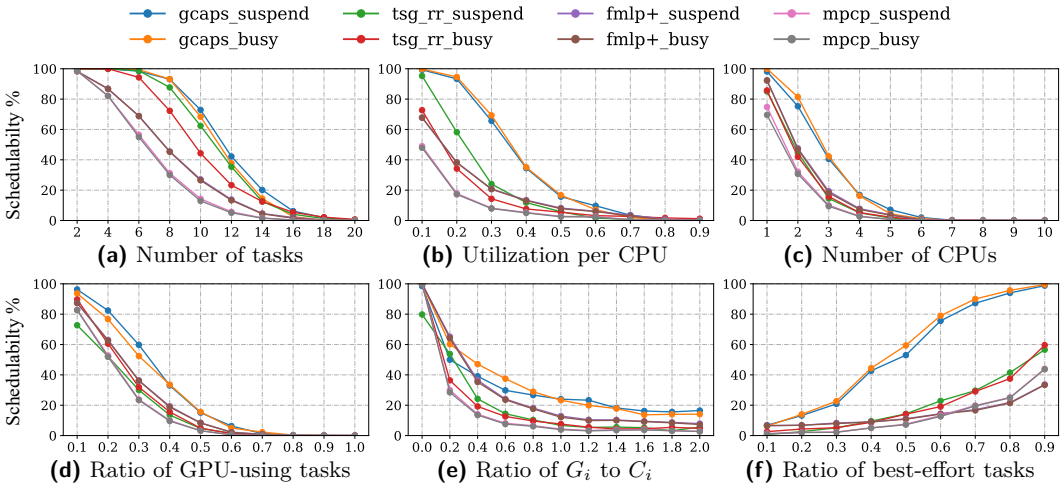


Figure 8 Schedulability of approaches with different experimental settings

7.1.1 Comparison with Prior Work

We first compare our proposed approach, GCAPS, with the default TSG round-robin scheduling as well as two well-known synchronization-based methods, MPCP [20] and FMLP+ [10], both of which offer suspension-aware and busy-waiting analyses. For default TSG scheduling, we use the analysis in Sec. 6.2 and set the length of the time slice to 1024 μs since it is the default value set in the driver. For our approach, we use the analysis with the GPU priority assignment in Sec. 5.3. Hence, we first run the response time test for a taskset with the default RM priorities, and if the test fails, try again with separate priorities for GPU segments.

We investigate the impact of varying the number of tasks in the taskset, the number of CPUs, the utilization per CPU, and the ratio of GPU-using tasks in Figs. 8a, 8c, 8b and 8d, respectively. The results show that, in general, the `gcaps_busy` and `gcaps_suspend` approaches outperform previous methods.

Fig. 8e examines the effect of changing the ratio of G_i/C_i. When the ratio of G_i/C_i is small, the proposed approach underperforms `fmlp+` and `tsg_rr_suspend`, because `fmlp+` can efficiently schedule tasks when GPU load is light, and the delay caused by TSG context switching is not significant when the duration of GPU segments are relatively short. The advantages of our approaches are mitigated by the critical section of runlist updates, but this trend does not continue as the ratio increases.

Lastly, we explore the impact of best-effort tasks running with the lowest priority in the system. After generating the tasks using the aforementioned method, we randomly designate a specific percentage of tasks as best-effort tasks in this experiment. Fig. 8f depicts the percentage of schedulable tasksets as the ratio of best-effort tasks increases. The rest of

Parameters	Value
Number of CPUs	4
Number of tasks per CPU	[3, 6]
Ratio of GPU-using tasks	[40, 60] %
Utilization per CPU	[0.4, 0.6]
Task Period	[30, 500] ms
Number of GPU segments per task	[1, 3]
Ratio of GPU exec. to CPU exec. (G_i/C_i)	[0.2, 2]
Ratio of GPU misc. in GPU exec. (G_i^m/G_i)	[0.1, 0.3]
Runlist update cost (ϵ)	1 ms

Table 3 Parameters for taskset generation

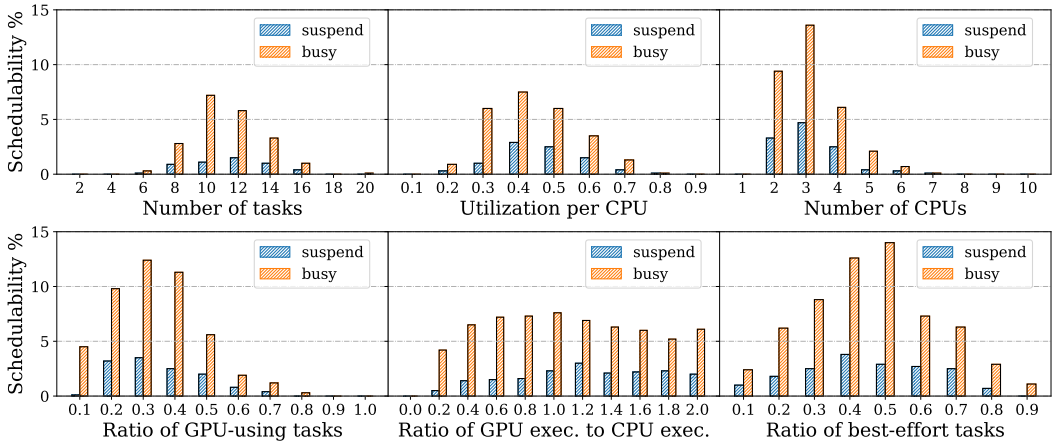


Figure 9 Schedulability gain in GCAPS by GPU priority assignment

the tasks are all real-time tasks with constraint deadlines. The best-effort tasks contribute to blocking time in the analysis of `mpcp` and `fmlp+`, and share the time-sliced GPU with real-time tasks in `tsg_rr`. Since GPU preemption is enabled in our proposed approaches, they significantly outperform the prior methods.

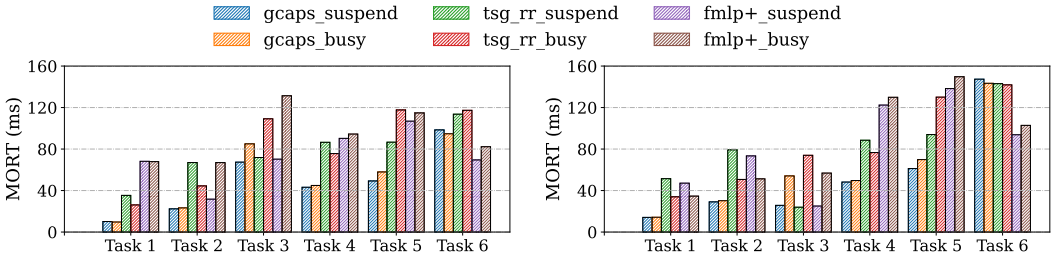
7.1.2 Effect of GPU Priority Assignment

In this experiment, we evaluate the impact of GPU priority assignment on taskset schedulability. We compare baseline analyses of `ioctl_busy` and `ioctl_suspend` with and without separate GPU priorities, using the taskset generation parameters from Table 3. Fig. 9 illustrates the advantages of GPU priority assignment. Busy-waiting approaches tend to benefit more from this assignment, as explained in Section 6.4 of the manuscript. Additionally, both busy-waiting and self-suspending approaches benefit from it since assigning GPU priorities independently of CPU priorities makes GPU resource allocation more efficient. E.g. Tasks with shorter GPU segments or higher GPU urgency can be prioritized appropriately, reducing resource wastage.

7.2 System Evaluation

We implemented our preemptive GPU scheduling approaches on two Nvidia platforms: the Nvidia Jetson Xavier NX Development Kit running L4T R35.2.1 with Jetpack 5.0.2 and the Nvidia Jetson Orin Nano Developer Kit running L4T R35.4.1 with Jetpack 5.1.2. The first platform features a 6-core 64-bit Carmel ARMv8.2 processor and a Volta architecture GPU. For our experiments, we configured it to run at its highest frequencies in the 6-core 15W mode. The second platform is equipped with a 6-core Arm Cortex-A78AE v8.2 64-bit CPU and an Ampere architecture GPU, and we operated it at its peak frequencies under its default power mode.

Case Study. We conducted a case study on the aforementioned platforms to evaluate the performance and effectiveness of the proposed preemptive GPU scheduling mechanism. Table 4 provides a summary of the taskset employed in this study, and we show the tasks' WCET collected on Jetson Xavier NX. The benchmarks are from Nvidia CUDA Samples [1]. The execution requirements of the tasks (C_i and G_i) are obtained by profiling the benchmarks and rounding them up. T_i ($= D_i$) is chosen based on C_i and G_i to ensure task utilization falls between 0.05 and 0.35. The CPU assignment is based on the consideration of load balancing.



(a) Jetson Xavier NX **(b)** Jetson Orin Nano

Figure 10 Maximum observed response time on two platforms

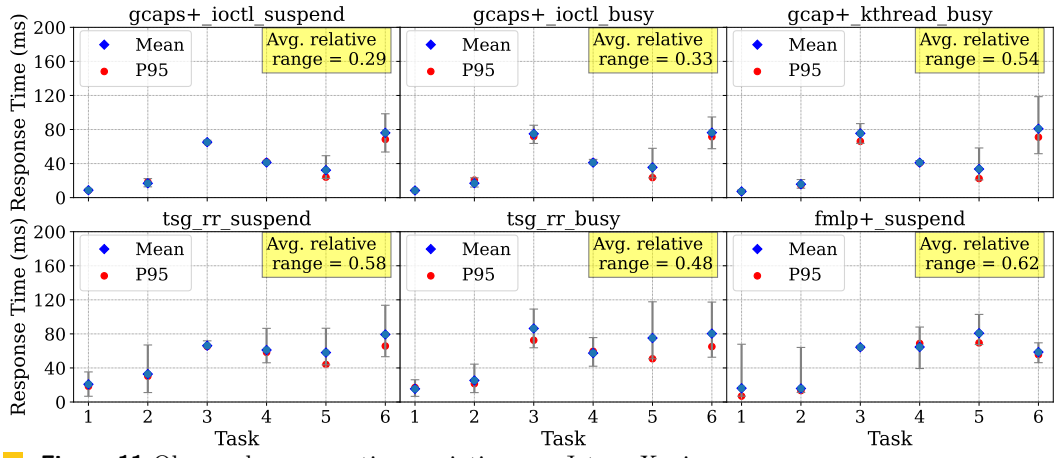


Figure 11 Observed response time variations on Jetson Xavier

The tasks in the table are arranged in descending order of priority, and each task's GPU segments use the same OS-level priority as its CPU segments. Tasks 3 is a CPU-only task with $G_i = 0$, while the remaining tasks involve GPU computations. Tasks 6 and 7 are categorized as best-effort tasks, as they are not assigned real-time priority. Task 7 is a graphic application running at 16 FPS to stress the GPU. To suspend a task during its GPU execution, we used CUDA events with the `cudaEventBlockingSync` flag. We compared our approaches against `tsg_rr` (default round-robin scheduling in Nvidia GPU driver) and `fmlp+` (synchronization-based approach).

We released the tasks at the same time and executed them for a duration of 30s during which we measured the maximum observed response time (MORT) for each real-time task. The results are depicted in Fig. 10a. Generally, the proposed approach can provide low MORT for real-time tasks, particularly for higher-priority tasks such as Task 1 and Task 2. This indicates that the IOCTL approach prioritizes these tasks effectively. However, it is apparent that the best-effort tasks, such as Task 6, experience a trade-off, displaying higher MORT under the proposed approaches than `fmlp+`. This is also due to the reason that the low-priority tasks can benefit from the blocking effect under `fmlp+` approaches. We omit the results for task 7 since it is a graphic application and its performance is measured through

Task	Workload	C_i	G_i	$T_i = D_i$	CPU	Priority
1	histogram	1	10	100	1	70
2	mmul_gpu_1	2	12	150	2	69
3	mmul_cpu	67	0	200	2	68
4	projection	12	15	300	1	67
5	dxtc	2	16	400	1	66
6	mmul_gpu_2	4	44	200	4	0
7	simpleTexture3D (graphic app)	4	27	67	4,5	0

Table 4 Taskset used in case study

Task	tsg_rr_suspend		tsg_rr_busy		gcaps_suspend		gcaps_busy	
	MORT	WCRT	MORT	WCRT	MORT	WCRT	MORT	WCRT
1	45.33	60	26.13	60	10.15	16	9.68	16
2	66.97	73.6	44.47	73.6	22.36	32	23.28	32
3	71.84	76	109.14	129.2	67.39	75	85.01	111
4	86.50	98.2	75.64	192.2	43.17	59	44.91	59
5	86.62	127.8	117.68	Failed	49.24	79	57.93	79

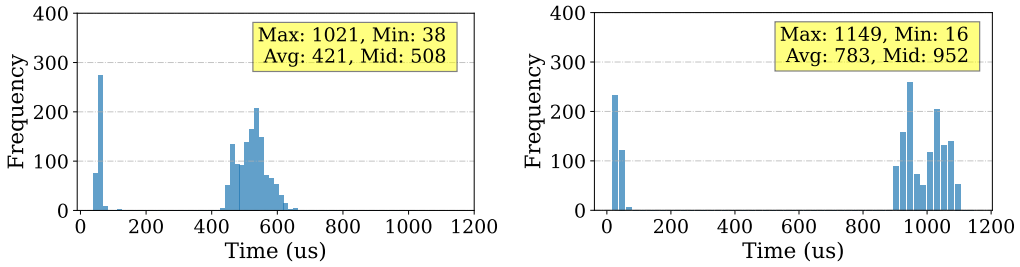
Table 5 Comparison of MORT (ms) and WCRT (ms) on Jetson Xavier

FPS. According to our observation, under each scheduling approach, an average FPS of around 15 can be maintained.

Fig. 11 illustrates the observed response time of each task, with error bars representing the deviation from the mean; above for "Max-Mean" and below for "Mean-Min" respectively. The "Average relative range" is calculated as "(Max-Min)/Max" to reflect variability. Both `ioctl_suspend` and `ioctl_busy` show a tendency to more consistent response times for real-time higher-priority tasks, as evidenced by more compact error bars in Fig. 11 and small variability compared to `tsg_rr` and `fmlp+`. `fmlp+` exhibits higher variability in the observed response time primarily due to blocking which has significantly increased the response time for the real-time tasks. Meanwhile, `tsg_rr` displays medium-sized error bars across the tasks, due to the use of a fairer allocation of GPU resources but without introducing blocking.

Table 5 lists the comparison of MORT and the worst-case response time (WCRT) bounds computed using our analysis given in Sec. 6 for the default round-robin scheduling and our proposed approaches. `tsg_rr_busy` failed the response time test at Task 5 due to the conservative nature of the analysis for busy-waiting tasks. The results of `fmlp+` are omitted since the tests failed at Task 1, while we observed no deadline misses for this taskset running on the real systems. This proves that our proposed preemptive GPU scheduling approach can offer tighter WCRT bounds for higher-priority tasks.

We run the same experiments on Nvidia Jetson Orin Nano, an embedded GPU platform with the latest Ampere architecture, and the similar trends of MORT are shown in Fig. 10b.



(a) Jetson Xavier NX **(b)** Jetson Orin Nano

Figure 12 Histogram of runlist update overhead

Runlist Update Overhead. We also measured the overhead of runlist update, ϵ (Def. 2), while running the taskset in the case study. The data and the distribution is shown in Fig. 12. The lower mode in the distribution indicates requests that do not necessarily require runlist updates, and it mainly includes the cost of accessing the IOCTL system call. In our experimental settings, these two boards have similar CPU frequencies at about 1.5GHz while Jetson Xavier NX has a much higher GPU frequency of 1.1GHz than Jetson Orin Nano's 625MHz. Both platforms exhibit a maximum overhead of about 1 ms, which is higher than the range reported in prior work [11]. We suspect this is due to the relatively lower frequency of our GPUs and it could be optimized in future generations of GPU architectures, as can be seen with Orin's case (10% higher overhead despite half the frequency). Nonetheless,

we consider the cost acceptable based on our schedulability experiments conducted with a similar overhead.

TSG Context Switching Overhead. We designed a separate experiment to measure the TSG context switching overhead, θ (defined in Sec. 2). The main idea is to use Eq. (3). To accurately gauge the number of TSGs, we incorporated a dummy loop within each kernel to extend the kernel duration so that it ensures that multiple timeslices are required to complete the execution. We launched different numbers of instances simultaneously of each workload given in Table. 4. In an ideal scenario, running ν identical kernels concurrently would lead to a slowdown factor of ν, and we can estimate the TSG context switching overhead based on the difference between the ideal and the actual slowdown. Specifically, we first measured the independent completion time of a kernel as E_1. We then launched ν identical kernels simultaneously and recorded the completion time as E_ν. This approach allows us to compute the TSG switching overhead as follows:

$$\theta = \frac{E_\nu - \nu \cdot E_1}{\nu \cdot E_1} \cdot L \tag{15}$$

where L is set to 1000μs in the default Tegra driver. The task slowdown and the estimated TSG context switching times are listed in Fig. 13, where both platforms demonstrate an average TSG context switching overhead greater than 200μs which is not trivial, especially for long-running kernels. It is worth noting that the overhead on Jetson Orin is lower than that one Jetson Xavier, which is opposite to the case of runlist update delay (ϵ). This might be attributed to several factors related to architectural differences that Jetson Orin has a better pipeline management and a more efficient context-saving mechanism.

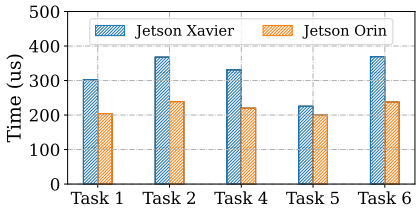


Figure 13 Avg. Overhead per TSG Context Switching

8 Conclusion

In this paper, we present a preemptive priority-based scheduling approach for GPU-using tasks in a multi-core real-time system equipped with an Nvidia GPU. We first discussed how the Nvidia Tegra GPU driver works and presented the design of GCAPS, our priority-based preemptive GPU context scheduling approach. Then, we provided a comprehensive response time analysis for both the default round-robin scheduling of the device driver and our proposed approach. To the best of our knowledge, this was the first attempt to formally analyze the worst-case response time under the default GPU driver's time-shared GPU context scheduling mechanism. Through empirical evaluations, we have demonstrated the effectiveness of our approach in enhancing schedulability compared to synchronization-based approaches and the default driver. Additionally, our case study shows the benefits of our approach in predictability and responsiveness over the default GPU driver and prior work.

Future work can focus on further optimizing and refining the proposed approach and exploring additional scheduling strategies such as dynamic priority. Combining our device-driver level approach with GPU partitioning mechanisms will also be an interesting direction.

References

1. Nvidia CUDA samples. https://github.com/NVIDIA/cuda-samples.
2. AnandTech. The NVIDIA GeForce GTX 1080 & GTX 1070 Founders Editions Review. https://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review.
3. Neil C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. 2007.
4. Joshua Bakita and James H. Anderson. Hardware Compute Partitioning on NVIDIA GPUs. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2023.
5. Joshua Bakita and James H. Anderson. Demystifying NVIDIA GPU Internals to Enable Reliable GPU Management. *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2024.
6. C. Basaran and K. Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 287–296, 2012. doi:10.1109/ECRTS.2012.15.
7. Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on parallel and distributed systems*, 20(4):553–566, 2008.
8. Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Syst.*, 30(1–2):129–154, may 2005. doi:10.1007/s11241-005-0507-9.
9. Konstantinos Bletsas, Neil C. Audsley, Wen-Hung Huang, Jian-Jia Chen, and Geoffrey Nelissen. Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions. *Leibniz Transactions on Embedded Systems*, 5(1):02:1–02:20, May 2018. doi:10.4230/LITES-v005-i001-a002.
10. Björn B Brandenburg. The FMLP+: An asymptotically optimal real-time locking protocol for suspension-aware analysis. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 61–71. IEEE, 2014.
11. Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based scheduling for GPU with preemption support. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 119–130. IEEE, 2018.
12. Glenn Elliott and James Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48:34–74, 05 2012. doi:10.1007/s11241-011-9140-y.
13. Glenn Elliott and James Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems. *Real-Time Systems*, 49(2):140–170, 2013.
14. Glenn Elliott et al. GPUSync: A framework for real-time GPU management. In *IEEE Real-Time Systems Symposium (RTSS)*, 2013.
15. Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association. URL: https://www.usenix.org/conference/osdi22/presentation/han.
16. S. Jain, I. Baek, S. Wang, and R. Rajkumar. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 29–41, 2019. doi:10.1109/RTAS.2019.00011.
17. S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 57–66, 2011. doi:10.1109/RTSS.2011.13.

18 H. Kim, P. Patel, S. Wang, and R. R. Rajkumar. A server-based approach for predictable GPU access control. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, 2017. doi:10.1109/RTCSA.2017.8046309.

19 Jinkyu Lee. Improved schedulability analysis using carry-in limitation for non-preemptive fixed-priority multiprocessor scheduling. *IEEE Transactions on Computers*, 66(10):1816–1823, 2017.

20 Pratyush Patel, Iljoo Baek, Hyoseung Kim, and Ragunathan Rajkumar. Analytical enhancements and practical insights for MPCP with self-suspensions. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.

21 Ragunathan Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings., 10th International Conference on Distributed Computing Systems*, pages 116–117. IEEE Computer Society, 1990.

22 Steven Rostedt. Rt-mutex. https://docs.kernel.org/locking/rt-mutex-design.html, 2009.

23 S. Saha, Y. Xiang, and H. Kim. STGM: Spatio-temporal GPU management for real-time tasks. In *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–6, 2019. doi:10.1109/RTCSA.2019.8864564.

24 I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on GPUs. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 193–204, 2014. doi:10.1109/ISCA.2014.6853208.

25 Yidi Wang. *Advancing Real-Time GPU Scheduling: Energy Efficiency and Preemption Strategies*. PhD thesis, University of California, Riverside, 2023.

26 Yidi Wang, Mohsen Karimi, and Hyoseung Kim. Towards Energy-Efficient Real-Time Scheduling of Heterogeneous Multi-GPU Systems. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 409–421. IEEE, 2022.

27 Yidi Wang, Mohsen Karimi, Yecheng Xiang, and Hyoseung Kim. Balancing energy efficiency and real-time performance in GPU scheduling. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 110–122. IEEE, 2021.

28 Yidi Wang, Cong Liu, Daniel Wong, and Hyoseung Kim. Unleashing the power of preemptive priority-based scheduling for real-time gpu tasks, 2024. arXiv:2401.16529.

29 Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 119–130, 2015.

30 Yecheng Xiang and Hyoseung Kim. Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 392–405. IEEE, 2019.

31 H. Zhou, G. Tong, and C. Liu. GPES: a preemptive execution system for GPGPU computing. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 87–97, 2015. doi:10.1109/RTAS.2015.7108420.

32 An Zou, Jing Li, Christopher D Gill, and Xuan Zhang. RTGPU: Real-time GPU scheduling of hard deadline parallel tasks with fine-grain utilization. *IEEE Transactions on Parallel and Distributed Systems*, 2023.