Compute Kernels as Moldable Tasks: Towards Real-Time Gang Scheduling in GPUs

Attilio Discepoli
Vrije Universiteit Brussel
Brussels, Belgium
attilio.discepoli@vub.be
0009-0001-7100-3909

Mathias Louis Huygen
Vrije Universiteit Brussel
Brussels, Belgium
mathias.louis.huygen@vub.be

Antonio Paolillo
Vrije Universiteit Brussel
Brussels, Belgium
antonio.paolillo@vub.be
0000-0001-6608-6562

***Abstract*—We present a preliminary evaluation of real-time scheduling policies for GPU kernels modeled as moldable tasks. Our framework maps periodic real-time jobs to CUDA kernels and uses `libsmctrl` to assign Texture Processing Clusters (TPCs), enabling gang scheduling with variable parallelism. A calibration tool provides per-kernel WCET estimates across different TPC counts, allowing the scheduler to trade execution time for resource usage.**

We compare several scheduling strategies, including CUDA's default concurrent kernel execution ("all-out"), a sequential EDF policy using all TPCs per job, and a moldable EDF scheduler that dynamically allocates just enough TPCs to meet each job's deadline. Using static per-task memory allocation, we eliminate prior sources of interference and achieve WCET predictability.

Our results show that deadline-aware scheduling outperforms the default CUDA strategy in scenarios with urgency mismatches. Moreover, moldable EDF improves over sequential EDF by reducing deadline misses under non-preemptive execution, especially when long-running jobs could block shorter urgent ones.

***Index Terms*—GPU Partitioning, Gang Scheduling, Moldable Tasks, WCET Estimation, Non-Preemptive Scheduling, CUDA.**

I. Introduction

GPUs are increasingly integrated into real-time and safety-critical systems, including autonomous vehicles, robotics, and industrial automation. While their parallelism offers significant performance benefits, GPUs remain challenging to integrate in systems requiring predictability. One key obstacle is the lack of fine-grained control over how GPU resources are shared and scheduled across concurrent workloads.

In response to this, recent efforts have explored ways to partition GPU resources, such as using NVIDIA's Streaming Multiprocessor (SM) partitioning feature, exposed via tools like `libsmctrl` [1]. Partitioning promises isolation between workloads by assigning subsets of SMs—called Texture Processing Clusters (TPCs)—to different jobs. This opens the door to applying real-time scheduling techniques such as Earliest Deadline First (EDF), gang scheduling, or moldable parallelism on GPUs.

Following the terminology of Goossens and Berten [2], a parallel job in a gang scheduling system is said to be *rigid* if its processor allocation is fixed externally and never changes, *moldable* if the scheduler decides the allocation at release time, and *malleable* if the allocation can change during execution. In our case, the allocated resource under consideration is the number of TPCs assigned to a kernel launch.

In this work, we explore the feasibility and benefits of deadline-aware *moldable scheduling* for GPU kernels, leveraging runtime TPC assignment to adapt to system load. We develop a custom CUDA scheduling framework that models periodic real-time task sets, where each job corresponds to a GPU kernel execution. Using `libsmctrl`, we assign TPC masks at runtime based on the job's urgency and its worst-case execution time (WCET) profile, measured as a function of TPC count.

We implement several schedulers, including CUDA's default concurrent launch policy ("all-out"), a sequential EDF policy that assigns all TPCs to one job at a time, and a moldable EDF scheduler that dynamically adapts job parallelism based on deadline constraints. Our evaluation shows two key findings. First, the "all-out" strategy, while aggressive in parallelism, is unaware of deadlines and can underperform even simple sequential EDF in deadline-sensitive workloads. Second, the moldable EDF scheduler improves over both baselines by assigning only the resources necessary to meet each job's deadline, thus avoiding deadline misses—particularly in non-preemptive scenarios where greedy jobs might otherwise block shorter, urgent ones.

These results demonstrate that combining SM partitioning with moldable real-time scheduling can outperform default GPU execution strategies, and offer a promising direction for predictable GPU integration in real-time systems.

II. Framework Overview

We developed a custom runtime framework for evaluating GPU scheduling strategies under a real-time task model. Each task corresponds to a periodic stream of GPU jobs, implemented as CUDA kernels, and is characterized by a period, relative deadline, and kernel type. The kernel type of a task defines the specific workload executed by its jobs. It implicitly determines the Worst-Case Execution Time (WCET) of a job as a function of the GPU resources (*e.g.*, TPCs) assigned to it, as detailed below. All jobs are released on a fixed periodic schedule and are executed non-preemptively. The scheduler is invoked on each job release and completion event to determine which jobs to run next and how to assign GPU resources.

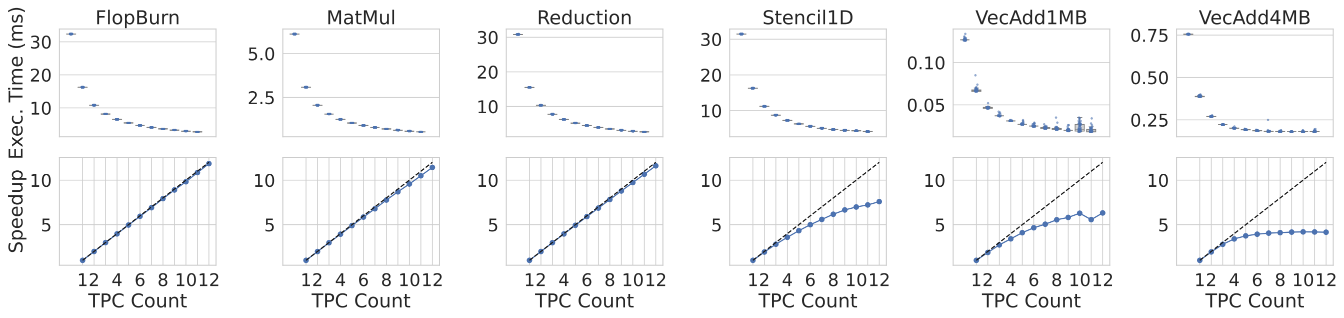


Fig. 1. Execution times (top row) and speedups (bottom row) of CUDA kernels under increasing TPC count (from 1 to 12). Compute-bound kernels (`FlopBurn`, `MatMul`, `Reduction`) show near-linear speedups, closely tracking the ideal scaling line (dashed). `Stencil1D` also scales well, but with diminishing returns due to memory reuse patterns (non-coalesced accesses). In contrast, memory-bound kernels (`VecAdd4MB`) saturate early, showing limited speedup beyond 4-6 TPCs. These results are used by the scheduler to estimate empirically the worst-case execution time as a function of TPC count.

SM Partitioning and TPC Masks. To control GPU resource usage, we use `libsmctrl`, a library that enables fine-grained partitioning of NVIDIA GPUs by restricting kernels to specific subsets of Streaming Multiprocessors (SMs). On our GPU model, a TPC corresponds to two SMs, and the TPC mask of a job defines the subset of SMs it is allowed to run on. This allows the scheduler to enforce isolation policies by selecting non-overlapping TPC masks for concurrent jobs.

Scheduler and Policy Interface. The framework supports pluggable scheduling policies. Schedulers implement a uniform interface and maintain their own internal state, such as the list of pending jobs. The interface consists of:

- `on_job_release(GpuJob*)`
- `on_job_complete(GpuJob*)`
- `std::vector<GpuJob*> select_jobs_to_run()`

For example, EDF-based schedulers use a priority queue internally, sorted on absolute deadlines, and allocate TPCs based on job urgency. All scheduling decisions are made online and are executed by a central loop that launches jobs using their assigned TPC masks. This design enables experimentation with various scheduling policies, including those supporting parallel jobs. We support:

- `seq-edf`: a sequential EDF policy where jobs run one at a time with all TPCs;
- `global-edf-1tpc`: each job runs with one TPC, allocated greedily;
- `moldable-edf`: each job is assigned the smallest TPC count that satisfies its WCET–deadline pair;
- `all-out` (baseline): every job is launched immediately, in a default-priority CUDA stream, with a full TPC mask (*i.e.*, no partitioning). On current NVIDIA GPUs, the Task Management Unit (TMU) and Work Distribution Unit (WDU) dispatch such kernels in *FIFO order within each priority level* [1]. Because all our streams use the default (equal) priority and no kernels spawn child kernels (*i.e.*, no CUDA Dynamic Parallelism), the dispatch order is reproducible across repeated runs of the same task set. This policy represents the default behavior of the CUDA runtime.

WCET Calibration. To estimate per-job execution time under different TPC configurations, we developed a WCET calibration tool. This tool runs each kernel type in isolation across 1 to N TPCs and records execution times. These measurements populate a per-kernel-type WCET vector made available to the scheduler. During execution, the scheduler can use this vector to reason about how many TPCs to assign to a job in order to meet its deadline.

Task-Based Memory Management. To ensure consistency and eliminate memory-related timing anomalies, each task acts as a "vessel" for its jobs, encapsulating both the kernel type and the memory buffers needed for execution. At program startup, each task allocates the required data structures in GPU device memory; these buffers are then reused by every job released by the task, avoiding any per-job allocation or deallocation during execution. This setup prevents host-to-device and device-to-host data transfers during execution, ensuring that all data resides in GPU memory throughout the experiment. It also isolates compute behavior from interference caused by shared memory traffic or copy engines—issues we leave to future work. This design significantly improves timing predictability. In earlier experiments, dynamic allocations and implicit data transfers introduced substantial timing noise and unintended synchronization effects, as also observed by Yang et al. [3]. With this simplification, execution times became remarkably more stable, and WCET overruns were largely eliminated.

Experimental Testbed. All experiments were conducted on an NVIDIA RTX 2000 Ada Generation Laptop GPU, a *discrete* GPU with 24 Streaming Multiprocessors (SMs), corresponding to compute capability 8.9. The system was configured with NVIDIA driver version 570.133.20 and CUDA toolkit version 12.8.

Frequency Control. To reduce variability and ensure repeatable WCET measurements, we disabled dynamic frequency scaling (DVFS) and fixed both the core and memory clocks using `nvidia-smi`. Specifically, we enabled persistence mode and locked the graphics and memory clocks to 2115 MHz and 7001 MHz, respectively: `nvidia-smi -pm 1, -lgc 2115, -lmc 7001,7001`. This eliminates one

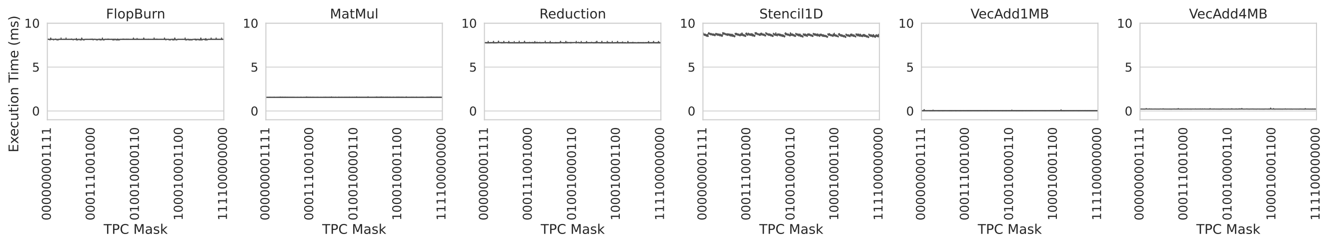


Fig. 2. Impact of TPC placement on execution time for a fixed TPC count ($k = 4$). Each subplot corresponds to a different CUDA kernel and shows execution time variation across all valid 4-TPC masks (out of 12 total). Only a few representative masks are shown on the x-axis for readability. Compute- and memory-bound kernels exhibit consistent runtimes regardless of which TPCs are selected, suggesting minimal sensitivity to placement.

major source of temporal jitter and ensures that observed WCET overruns are not artifacts of frequency throttling.

III. WCET Scaling and Placement Sensitivity

To support moldable scheduling and inform resource allocation decisions, we precompute the execution time of each CUDA kernel type across different levels of parallelism. Our WCET calibration tool executes each kernel in isolation using TPC masks ranging from 1 to 12 active TPCs, and records the corresponding execution times. The kernels used in this study span a range of compute and memory intensities:

- **FlopBurn**: a compute-bound kernel that performs repeated trigonometric operations to saturate the floating-point ALUs,
- **MatMul**: a tiled matrix multiplication using shared memory, representative of structured compute workloads,
- **Reduction**: a parallel reduction that combines shared memory and atomic operations, with moderate control divergence,
- **Stencil1D**: a 1D stencil operation with neighborhood data dependencies and moderate memory reuse,
- **VecAdd1MB / VecAdd4MB**: simple element-wise vector additions over 1MB or 4MB arrays, limited by global memory bandwidth.

Figure 1 shows the measured WCETs of these kernels and the corresponding speedups as a function of TPC count. Compute-bound kernels such as `FlopBurn` and `MatMul` scale nearly linearly with the number of TPCs, achieving close to ideal speedup. Memory-bound kernels such as `VecAdd1MB` and `VecAdd4MB`, however, exhibit diminishing returns beyond 4–6 TPCs. `Stencil1D` kernels fall in between: they benefit from parallelism but scale sub-linearly.

These WCET profiles are exposed to the scheduler at runtime and enable reasoning about the trade-off between parallelism and execution time of a job, based on the number of TPCs it is assigned. They serve our moldable scheduler's decision process, which aims to assign the minimum number of TPCs needed to meet a job's deadline.

To assess the robustness of these measurements, we conducted a sensitivity study on TPC placement. Even when the number of assigned TPCs is fixed, the specific SMs selected (*i.e.*, the bit pattern of the TPC mask) could, in principle, affect execution time due to undocumented architectural asymmetries or locality effects. We fixed the number of active TPCs to $k = 4$ and systematically evaluated each kernel across all valid TPC masks with exactly four bits set, resulting in 495 unique placements out of 12 TPCs. Note that the bit logic in our masks is inverted relative to the convention used by `libsmctrl`: a bit set to 1 denotes an active TPC in our encoding. Each mask configuration was executed 50 times following a warmup phase. Figure 2 summarizes the resulting distributions. We observe *no significant impact of TPC placement* on execution time. Compute-intensive kernels such as `FlopBurn`, `MatMul`, `Reduction`, and `Stencil1D` exhibit narrow and consistent distributions across all placements. Even memory-bound kernels like `VecAdd1MB` and `VecAdd4MB` do not exhibit systematic sensitivity to SM selection. These results confirm that, in isolation, WCET is primarily determined by the number of assigned TPCs rather than their physical placement.

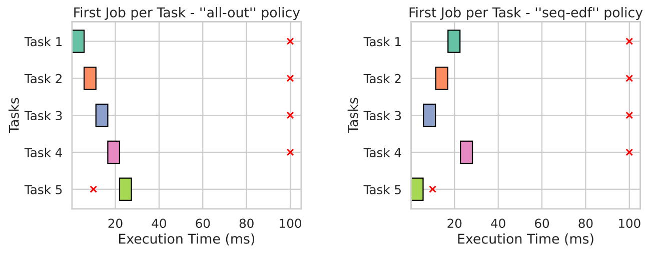
IV. Case Studies: When Scheduling Matters

To highlight the practical implications of deadline-aware scheduling on GPUs, we present two representative case studies drawn from our experiments. These scenarios illustrate how CUDA's default policy and simple sequential execution can both fail under real-time constraints, and how moldable scheduling can mitigate deadline misses.

A. Study 1: Deadline Awareness vs. Parallelism

This scenario compares `seq-edf` with the baseline `all-out` strategy. The taskset consists of five identical jobs. Four of them are tasks with a period and deadline of 100 ms, while one is an "urgent" task with a 10 ms deadline. All jobs execute the same kernel and share the same WCET profile, but the urgent task has significantly tighter timing constraints.

Under `all-out`, all five jobs are launched concurrently at each release. CUDA's default scheduling policy, as noted by Bakita et al. [1], admits kernels in FIFO order with limited prioritization. This means that once a kernel is submitted, all of its blocks must complete before newer kernels can begin execution—even if those newer kernels correspond to more urgent tasks. As a result, the urgent job may be delayed by earlier, less time-sensitive work, leading to deadline misses despite ample compute resources.



(a) `all-out`: jobs are launched concurrently. CUDA queues and schedules them in FIFO order. Urgent tasks are not prioritized.

(b) `seq-edf`: jobs are scheduled one at a time in EDF order. Urgent tasks are executed early and meet their deadlines.

Fig. 3. Case study 1: Comparison between deadline-unaware (`all-out`) and deadline-aware (`seq-edf`) scheduling. Each job uses all TPCs (12), and the taskset contains one urgent task (Task 5) with a tighter deadline. Red crosses mark job deadlines. Under `all-out`, Task 5 misses its deadline due to lack of prioritization. In contrast, `seq-edf` prioritizes it correctly, and all deadlines are met.

In contrast, `seq-edf` enforces a strict priority order based on absolute deadlines, running one job at a time using all TPCs. Although this strategy reduces concurrency, it ensures that the most urgent job is executed first, preventing unnecessary deadline violations.

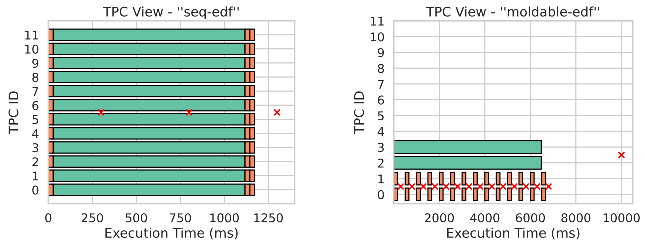
Figure 3 shows the resulting Gantt chart: while `all-out` suffers from misprioritized execution, `seq-edf` guarantees the timely completion of the urgent job.

B. Study 2: Resource-Aware Scheduling under Non-Preemption

This scenario compares `seq-edf` and `moldable-edf` on a two-task workload with non-preemptive execution. The first task is a "greedy" long-running job with a WCET of 1250 ms when using all 12 TPCs, and 7500 ms when limited to 2 TPCs. It has a period of 10 s and a relaxed deadline of 10 s. The second task releases short, urgent jobs with a WCET of approximately 265 ms on 2 TPCs and 25 ms on 12 TPCs, and with a period of 500 ms and a relative deadline of 300 ms.

Under `seq-edf`, jobs are executed sequentially using all 12 available TPCs. At the start of the hyperperiod, both tasks release their first jobs. Since the urgent job has an earlier deadline, it is correctly scheduled first. However, subsequent jobs of the urgent task arrive while the greedy job is executing and are blocked until it completes. Because execution is non-preemptive, the urgent jobs cannot reclaim GPU resources mid-execution and thus miss their deadlines. In contrast, `moldable-edf` dynamically estimates the number of TPCs required to meet each job's deadline based on the precomputed WCET table. It assigns only the necessary TPCs to the greedy job, leaving enough TPCs free to schedule the urgent job concurrently. As a result, `moldable-edf` avoids blocking and ensures both tasks remain schedulable.

Figure 4 shows the Gantt chart highlighting this difference: only `moldable-edf` successfully schedules both tasks without deadline misses.



(a) `seq-edf`: urgent jobs (orange) are delayed by the greedy job (green), leading to deadline misses for urgent jobs.

(b) `moldable-edf`: only minimal TPCs are assigned to the greedy task, enabling concurrent execution. All deadlines are met.

Fig. 4. Case study 2: Job execution timelines across TPCs. Each horizontal bar represents a kernel execution on a specific TPC. Red crosses mark job deadlines. Moldable scheduling avoids deadline misses by spatially multiplexing tasks.

These results show that moldable scheduling, when combined with SM partitioning, has the potential to enable higher schedulability even in non-preemptive systems.

V. Limitations

Our evaluation is preliminary and subject to several limitations that we plan to address in future work.

Resource isolation. While our framework allows fine-grained TPC partitioning, kernel executions are calibrated and evaluated in isolation, without concurrent memory traffic or host-side interference. Real-world deployments may experience shared resource contention—*e.g.*, across L2 cache slices, DRAM channels, or copy engines—which could impact both execution time and predictability, particularly for memory-bound kernels. As future work, we plan to extend our WCET analysis with interference-aware calibration, using co-scheduled background kernels to quantify slowdowns under contention. This will complement the results shown in Figure 2, where execution times were measured in isolation.

CUDA baseline behavior. We compare against CUDA's default kernel execution behavior, which typically launches kernels in FIFO order across streams. We do not explore alternative configurations enabled by the CUDA runtime, such as per-stream priorities or CUDA Graphs, though these could influence deadline satisfaction under certain workloads.

Application coverage. The current evaluation is limited to synthetic kernels with predictable structure and compute-dominated profiles. While useful to showcase scheduling behavior, real-world GPU workloads—such as DNN inference or sensor fusion—will be explored in future studies.

Formal analysis. Finally, no formal schedulability analysis is provided in this work. We focus on empirical evaluation; integrating analytical models (*e.g.*, response-time analysis) and characterizing worst-case interference under shared resource contention are important next steps.

VI. Related Work

Gang scheduling has long been explored in real-time systems to coordinate parallel workloads under predictable execution models. Nelissen et al. [4] introduced a response-time analysis framework for non-preemptive, periodic moldable gang tasks, establishing schedulability bounds under job-level fixed-priority (JLFP) policies.

Bakita et al. [1] proposed `libsmctrl`, a user-space mechanism for SM-level partitioning on NVIDIA GPUs, enabling spatial isolation via explicit control over the TPC mask. Our framework builds on this mechanism to support real-time schedulers that explicitly allocate GPU resources to individual jobs at runtime. Follow-up work [5] showed that SM partitioning alone does not ensure isolation, due to shared resource interference from memory controllers, copy engines, and internal arbitration mechanisms. These insights informed our framework design, particularly the need for static memory allocation. Ali et al. [6] recently introduced the Streaming Multiprocessor Locking Protocol (SMLP), which supports predictable intra-component GPU access by dynamically resizing GPU workloads to fit available SMs. SMLP is designed for component-based systems scheduled with JLFP inside time-sliced partitions, and offers analytical bounds on priority-inversion blocking. Their evaluation couples simulation-based blocking analysis with a brief hardware sweep—using `libsmctrl` to measure how one kernel's WCET scales with the number of SMs—to motivate the resizing model. Our work is complementary: we implement a moldable EDF scheduler on real CUDA hardware, calibrate WCETs for several kernels across TPC counts, and measure deadline behaviour under multiple scheduling policies. This empirical perspective lets us quantify the practical benefits of moldable gang scheduling for task sets with heterogeneous deadlines.

VII. Conclusion

We presented a preliminary evaluation of real-time scheduling strategies for GPUs using SM partitioning enabled by `libsmctrl`. Our framework models CUDA kernel executions as periodic tasks, performs WCET calibration across TPC counts, and supports a variety of scheduling strategies, including deadline-unaware (`all-out`), sequential EDF, and a moldable EDF policy. Through controlled case studies, we demonstrated that deadline-aware scheduling can outperform CUDA's default strategy, and that moldable EDF further improves schedulability by minimizing blocking under non-preemptive execution.

This work opens the path toward non-preemptive gang scheduling on GPUs by leveraging SM partitioning. As future work, we plan to develop a full-fledged moldable scheduler tailored to GPU architectures, along with formal schedulability analysis. This includes extending the moldable model to sporadic workloads by tracking per-task cooldown intervals—*i.e.*, known idle phases after job completion that allow safe, temporary reuse of reserved resources.

To further improve timing predictability, we will extend our WCET analysis to evaluate the impact of inter-kernel interference. Such analysis should also be conducted on real-world GPU applications; For instance, Bakita et al. [1] reported results for TPC partitioning on a YOLO workload.

A deeper investigation into memory-related effects—such as copy engine contention and device memory allocation overhead—is an important direction for future work.

Finally, we aim to explore the feasibility of true malleable scheduling, where TPC allocations can be updated dynamically during kernel execution—provided such capabilities are supported on current or emerging GPU architectures.

References

[1] J. Bakita and J. H. Anderson, "Hardware Compute Partitioning on NVIDIA GPUs," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)* . Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 54–66. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/RTAS58335.2023.00012

[2] J. Goossens and V. Berten, "Gang FTP scheduling of periodic and parallel rigid real-time tasks," Jun. 2010, arXiv:1006.2617 [cs]. [Online]. Available: http://arxiv.org/abs/1006.2617

[3] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, and F. D. Smith, "Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems," *LIPIcs, Volume 106, ECRTS 2018*, vol. 106, pp. 20:1–20:21, 2018, artwork Size: 21 pages, 880171 bytes ISBN: 9783959770750 Medium: application/pdf Publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[4] G. Nelissen, J. Marcè i Igual, and M. Nasri, "Response-Time Analysis for Non-Preemptive Periodic Moldable Gang Tasks," in *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Maggio, Ed., vol. 231. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 12:1–12:22.

[5] Bakita, Joshua and Anderson, James H., "Demystifying NVIDIA GPU Internals to Enable Reliable GPU Management," in *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2024, pp. 294–305.

[6] S. W. Ali, Z. Tong, J. Goh, and J. H. Anderson, "Predictable GPU Sharing in Component-Based Real-Time Systems," in *36th Euromicro Conference on Real-Time Systems (ECRTS 2024)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), R. Pellizzoni, Ed., vol. 298. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, pp. 15:1–15:22.