Work-in-Progress: Enabling Transparent Priority Scheduling for NVIDIA GPUs

Noah Weaver and Joshua Bakita
Department of Computer Science, Mohamed bin Zayed University of Artificial Intelligence, UAE
Email: {noah.weaver, joshua.bakita}@mbzuai.ac.ae

***Abstract*—Modern embedded systems increasingly rely on GPUs for safety-critical tasks such as object detection in autonomous vehicles and data visualization in medical diagnostics. These applications require running high-priority workloads alongside lower-priority tasks on shared GPU hardware due to cost, power, or space constraints. However, NVIDIA GPUs employ round-robin scheduling that treats all tasks equally, causing unpredictable delays for critical workloads. Existing GPU priority scheduling solutions require application modifications, introduce high overhead, or depend on specialized embedded hardware. We present a GPU-level static priority scheduler built on the `nvsched` framework that requires no application changes and runs on commodity GPUs. Our scheduler maintains tasks in a priority-ordered queue and uses a parameter communication system to enable dynamic priority updates from userspace. Experiments with MobileNetV3 inference under GPU contention demonstrate 32–41% lower maximum latency for high-priority tasks when using our scheduler compared to NVIDIA's default scheduler. Our approach provides practical priority control for GPU workloads while maintaining low scheduling overhead.**

I. Introduction

Embedded real-time systems increasingly perform complex GPU-accelerated tasks, from perception and planning in autonomous vehicles to natural language processing in intelligent assistants. Many of these systems must run safety-critical workloads alongside lower-priority tasks on shared GPU hardware to reduce cost and power consumption. However, current GPU scheduling mechanisms provide no priority control—a high priority task scheduled on the CPU may be delayed behind lower-priority work on the GPU, creating a fundamental disconnect in system-wide resource management.

By default, NVIDIA GPUs employ time-sliced round-robin scheduling where all tasks receive equal treatment regardless of importance or deadlines. This approach creates unpredictable timing behavior unsuitable for real-time systems: a critical object detection task must wait its turn behind background processing.

We address these limitations by implementing a preemptive static priority scheduler that operates entirely on the GPU without requiring application changes. Building on the `nvsched` framework, we also provide a parameter communication system for dynamic priority updates from userspace.

To illustrate the improvement of our system, consider an autonomous vehicle running two tasks on a single GPU: Task 1 is a high-priority object detection process, and Task 2 is a low-priority HD map generation process. Consider the schedule when both tasks release simultaneously, as shown in Fig. 1.

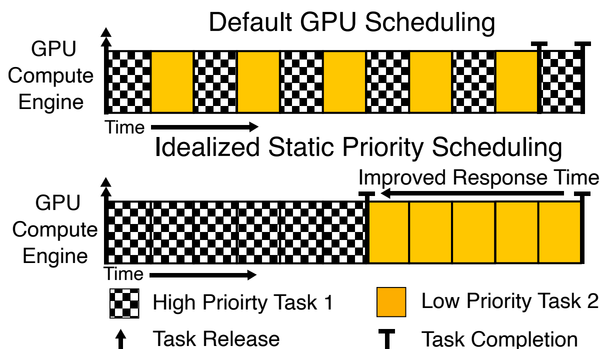

Fig. 1. Our scheduling technique ensures the complete execution of the prioritized task before the low priority task.

With the default scheduler, Task 1 must wait for alternating time-slices of Task 2. With our scheduler, Task 2 is preempted and forced to wait, halving Task 1's response time.

Previous work on GPU priority scheduling is insufficient due to its frequent inapplicability. Capodieci et al. [1] implemented deadline-based GPU scheduling with preemption support, though their work was developed with NVIDIA and their implementation has remained secret. GCAPS [2] introduced static priority GPU scheduling using OS-level task priorities but their system has high overheads. Further, both approaches exclusively target embedded GPUs and require application modifications, creating deployment barriers for existing systems. Our work provides priority scheduling on commodity GPUs without requiring changes to GPU applications.

Contributions. We contribute the following:

1) A preemptive static priority scheduler running entirely on the GPU using the `nvsched` framework.
2) A parameter communication system enabling userspace applications to dynamically update GPU task priorities.
3) An experimental evaluation demonstrating 31.7–40.7% latency reductions for high priority ML inference workloads under GPU contention compared to NVIDIA's default round-robin scheduler.

II. Background

A. GPU Scheduling Mechanisms

NVIDIA GPUs employ a time-sliced round-robin scheduling approach at the firmware/hardware level. Each GPU-using process is associated with a GPU context representing a virtual address space. The GPU maintains "runlists" (typically one for each engine [3]) of Time-Sliced Groups (TSGs). Each TSG corresponds to one CPU process and encapsulates multiple channels containing streams of GPU commands. The runlist scheduler operates in a work-conserving, preemptive manner, cycling through TSG entries in fixed order. Each TSG receives equal time slices (typically 2 ms) regardless of the importance or deadlines of their associated tasks. When a TSG's time budget expires or a TSG completes early, the scheduler automatically switches to the next enabled TSG in the runlist. GPU context switching between TSGs incurs overhead of 50–750 μs [2] due to the large amount of GPU state that must be saved [1]. This default approach treats all tasks equally—a critical safety function requiring immediate attention may wait behind lower-priority background tasks, creating undesirable timing behavior for real-time systems.

B. nvsched Background

`nvsched` is the working name for the scheduling framework we introduced in a prior work-in-progress paper [4].[1] `nvsched` enables custom schedulers to run entirely on the GPU, eliminating CPU-GPU synchronization costs. `nvsched` operates by controlling which tasks are enabled in the runlist, implicitly controlling what tasks will be executed on the GPU. `nvsched` ensures only the scheduler task and its selected next task are in the runlist at any point. This allows the scheduler to ensure that a specific task executes on the GPU without interference from other tasks. The framework supports pluggable scheduler interfaces through callback functions for task arrivals, departures, blocking, and wake-up events. `nvsched` schedulers configure how often they are run by setting a *quantum* value that specifies the minimum frequency at which the scheduler is invoked. It detects scheduling events by monitoring runlist changes and task completion timing, while using privileged GPU memory mappings to access control registers. The framework supports Turing, Ampere, and Ada-generation discrete NVIDIA GPUs. `nvsched` works with CUDA, OpenCL, and OpenGL tasks without requiring modification of the application. Prior approaches require costly CPU interrupts and context switches, adding around 421 μs of overhead per scheduling decision [2].

C. The Static Priority Scheduler

With static priority scheduling, tasks have fixed priorities that remain constant until changed by a user. In preemptive static priority systems, higher priority tasks immediately preempt lower priority tasks upon arrival and continue running until completion, ensuring predictable response times for critical tasks. For real-time systems, static priority scheduling provides several advantages: predictable worst-case response times through established analysis techniques [5] and simple implementation with low runtime overhead.

III. Implementation

We build upon the `nvsched` framework, extending it with (1) a preemptive static priority scheduler and (2) a parameter communication system that enables dynamic task priority updates from the CPU to GPU-resident scheduler plugins.

A. SPRIO Scheduler Design

Our SPRIO (Static Priority) scheduler implements preemptive priority-based scheduling on the GPU. Tasks are organized in a circular doubly-linked list ordered by priority, with higher-priority tasks positioned earlier in the list. Within each priority level, tasks are scheduled in round-robin order to ensure fairness among equal-priority workloads. Our implementation currently uses higher numerical values for higher priorities.[2] We chose a quantum of 2 ms as it results in the 75 μs scheduler overhead only taking up 3.75% of the quantum.

When a new task arrives, the scheduler inserts it into the priority-ordered list at the appropriate position based on its priority value (0 by default). Upon receiving a priority update message, the scheduler removes the task from its current position and reinserts it according to its new priority, ensuring the list remains properly ordered. This maintains O(1) removal and O(1) scheduling costs but requires O(n) insertion in the worst case, where n is the number of tasks.

B. System Architecture

When a CPU process wishes to update the priority of a GPU task, it sends a parameter message through a named pipe. A dedicated reader thread monitors this pipe and places messages into a circular buffer allocated in CUDA host pinned memory, making it accessible to both CPU and GPU code. During each scheduling iteration, the GPU-resident scheduler checks this buffer for pending parameter updates and processes them by reordering tasks in its priority queue.

This architecture decouples the parameter update mechanism from the scheduling logic, allowing arbitrary scheduling policies to receive parameter updates. The use of a pipe-based interface provides a simple, standard inter-process communication mechanism that any process can use to influence GPU scheduling.

IV. Evaluation

We evaluate our SPRIO scheduler against NVIDIA's default scheduler by recording the response time improvement for a high priority task under GPU contention and measuring scheduling overheads. MobileNetV3-Small [6] is a lightweight convolutional neural network used in production mobile applications for real-time image classification. Our experiments

[1] The full paper describing `nvsched` is progressing through the publication process and will appear prior to 4Sany full-length manifestation of this work on priority scheduling.

[2] We plan to reverse this convention in future versions to match Linux nice values, where lower numbers indicate higher priority.

demonstrate that applying GPU-level priority scheduling reliably improves the performance of high priority tasks.

We set up three different scenarios. In each, MobileNetV3-Small runs alongside the `constant_cycles_kernel` benchmark from prior work [3]. We only vary the MobileNetV3-Small batch size to be 32, 64, or 128 images large. We then benchmark the time per MobileNetV3-Small inference when running under the default scheduler, and when running under our SPRIO scheduler. We use a quantum size of 2 ms, configure our scheduler to use a higher priority (10) for MobileNetV3-Small, and take 1,000 samples in each test. Additional system properties are shown in Table I. Our results are shown in Table II, with statistics computed over samples 150-1000 to allow for a warm up period.

TABLE I
EXPERIMENTAL SPECIFICATIONS

Component	Specification
GPU	NVIDIA Quadro RTX 4000 (Turing)
CPU	AMD Ryzen Threadripper PRO 3955WX
RAM	64GB
Operating System	Ubuntu 20.04.6 LTS
CUDA Version	12.2
PyTorch Version	2.0.1
Framework	`nvsched` (latest pre-release)

CPU priority validation. To validate that GPU-level scheduling is necessary, we separately tested setting CPU process priority (via `nice -20`) while using the default GPU scheduler. This configuration showed no significant difference, confirming that OS-level priority controls do not affect GPU scheduling behavior.

A. Performance Results

TABLE II
OBSERVED MAXIMUM LATENCY COMPARISON OF BASELINE ROUND-ROBIN SCHEDULING VERSUS SPRIO PRIORITY SCHEDULING UNDER GPU CONTENTION.

Batch	Baseline (ms)	SPRIO (ms)	Improvement (%)
32	12.03	8.22	31.67
64	21.39	13.67	36.09
128	42.06	24.95	40.67

Reviewing Table II, observe that the improvements range from 31.67% to 40.67%, with larger batch sizes showing the greatest improvement. Larger batches save more absolute time—a batch size of 128 reduces latency by 17.11 ms compared to 3.81 ms for a batch size of 32. This is because larger batches have longer time periods between blocking events, resulting in fewer scheduler invocations and greater efficiency.

B. Scheduling Overhead Analysis

We measured scheduling overhead using NVIDIA's Nsight Systems profiler to capture context switch timings and scheduler execution duration. A comparison between the overhead in the default scheduler and our SPRIO scheduler is illustrated in Fig. 2.

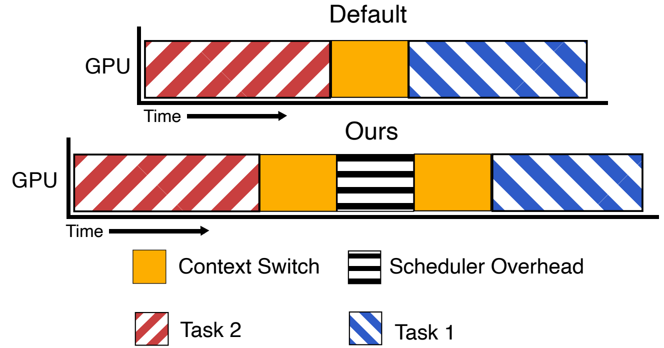

Fig. 2. Our scheduler incurs an additional context switch and scheduling overhead compared to NVIDIA's default scheduler.

The baseline round-robin scheduler incurs context switching overhead and advances to the next task in the runlist at fixed 2 ms intervals. SPRIO also uses a 2 ms quantum but incurs additional scheduling overhead from context switching and scheduler execution as shown at the bottom of Fig. 2.

Context switches take approximately 66 μs in both baseline and SPRIO configurations, while the SPRIO scheduler itself executes in roughly 75 μs. Since each task switch requires two context switches (one to switch to the scheduler, one to switch to the next task) plus scheduler execution time, each SPRIO scheduling iteration adds approximately 207 μs of overhead.

Even after accounting for scheduling overhead, we observe a net 3.81 ms improvement for a batch size of 32, with similar improvements across batch sizes. This indicates that the benefits of prioritization substantially outweigh the scheduling costs. The ability to preempt low-priority work at the next quantum when high priority tasks are ready more than compensates for the increased scheduling overhead.

V. FUTURE DIRECTIONS

While our evaluation demonstrates substantial performance improvements for an ML inference workload under GPU contention, several avenues remain for extending this work.

Experimental extensions. Our current evaluation focuses on a single high priority inference task competing against a background compute task. We plan to evaluate multiple concurrent inference streams at different priority levels to better reflect realistic multi-tenant GPU environments. Additionally, we plan to test across diverse workloads—including models with different compute and memory characteristics such as ResNet and BERT—this would demonstrate the generality of our approach beyond MobileNetV3. To understand the impact of scheduling decisions on low-priority tasks we plan to measure background task throughput to quantify the cost/benefit trade-off of priority enforcement. Furthermore, we plan to explore the relationship between quantum size and performance across different workload types to reveal optimal configurations. Finally, we plan to demonstrate dynamic priority updates during

execution to showcase the full capabilities of our parameter communication system, showing that priority changes take effect within a single quantum period.

OS integration. Our current parameter communication system requires explicit priority updates through our command-line utility or API. We intend to explore automatic synchronization between CPU-level and GPU-level task priorities. This would create a unified notion of task priority across the CPU and GPU, allowing priority management without GPU-specific knowledge.

Scheduler performance. Our current SPRIO implementation could be optimized to reduce scheduling overhead. We plan to profile the scheduler to identify any performance bottlenecks and explore improvements to data structures or validation logic.

Advanced scheduling and formal analysis. We plan to extend our parameter communication framework to support more sophisticated schedulers and demonstrate the extensibility of our design. We also plan to conduct rigorous formal worst-case response time analysis to enable the deployment of our scheduler in safety-critical real-time systems.

VI. Conclusion

Modern embedded systems require fine-grained control over GPU resource allocation to meet the timing requirements of safety-critical workloads. However, the lack of flexible, programmable GPU scheduling mechanisms has hindered both practical deployment and research exploration of GPU scheduling policies. We present a preemptive static priority scheduler that demonstrates the usefulness of the `nvsched` framework in practice. Our SPRIO scheduler shows that explicit priority control running on the GPU is possible, and our parameter communication system enables experimentation with arbitrary scheduling policies. Experimental evaluation demonstrates that GPU-level priority scheduling provides substantial performance improvements compared to NVIDIA's native scheduler, reducing the maximum latency of a high priority task by 31.7–40.7%. In future work, we plan to experiment with a variety of workloads, scheduling algorithms, and OS priority integration in order to further demonstrate the viability of our scheduler. We also plan to perform formal response-time analysis of our SPRIO scheduler to demonstrate its relevance to safety critical real-time systems.

Acknowledgments

Claude Sonnet 4.5 was used to draft Sec. V and Sec. II. These sections were then carefully edited and rewritten by hand.

References

[1] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-based scheduling for GPU with preemption support," in *Proceedings of the 39th IEEE Real-Time Systems Symposium*, Dec 2018, pp. 119–130.

[2] Y. Wang, C. Liu, D. Wong, and H. Kim, "GCAPS: GPU context-aware preemptive priority-based scheduling for real-time tasks," in *Proceedings of the 36th Euromicro Conference on Real-Time Systems*, Jul 2024, pp. 14:1–14:25.

[3] J. Bakita and J. H. Anderson, "Demystifying NVIDIA GPU internals to enable reliable GPU management," in *Proceedings of the 30th IEEE Real-Time and Embedded Technology and Applications Symposium*, May 2024, pp. 294–305.

[4] ——, "Work in progress: Increasing schedulability via on-GPU scheduling," in *Proceedings of the 31st IEEE Real-Time and Embedded Technology and Applications Symposium*, May 2025, pp. 422–425.

[5] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, Jan 1973.

[6] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for mobilenetv3," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 1314–1324.