Work in Progress: Increasing Schedulability via on-GPU Scheduling*

Joshua Bakita and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill, USA
Email: {jbakita, anderson}@cs.unc.edu

***Abstract*—GPUs are increasingly needed to run a variety of tasks in embedded systems, from object recognition to conversational chat. Some of these tasks are *safety-critical, real-time* tasks, where completing each by its deadline is essential for system safety. To meet the practical constraints of real-world systems, these tasks much also be run efficiently. Unfortunately, current techniques to schedule GPU-using tasks onto a single GPU while respecting deadlines impart high overheads, leading to inefficiency and substantial capacity loss during formal analysis. We address this problem by moving GPU scheduling from the CPU to the GPU. Our approach limits overheads, increasing the proportion of CPU tasks which can meet their deadlines by as much as 12.1% while increasing available GPU capacity.**

I. Introduction

Embedded real-time systems are being called upon to perform increasingly complex tasks, from perception and planning in a self-driving car to natural language processing for an intelligent assistant. GPUs are currently the most readily available and widely used processor for these types of computations. Due to practical constraints on commercial embedded systems (*e.g.* size, weight, power, and cost), it has become necessary to maximize efficiency by sharing a single GPU among multiple tasks. In many of these systems, it is essential to ensure that *safety-critical* tasks—those for which failure could result in death or destruction—always complete by their deadlines.

Unfortunately, current techniques to schedule multiple GPU-using tasks onto a single GPU impart high overheads. Such overheads consume valuable processor time that could be used to execute tasks. We find that this cost easily exceeds 11% of system capacity when schedulers must execute frequently, This limits the range of task sets in which all tasks can be guaranteed to meet their deadlines.

Our solution. We eliminate the overheads inherent in prior approaches to GPU scheduling by moving GPU scheduling off of the CPU and onto the GPU. Fig. 1 illustrates how our approach (bottom) differs from that of prior work (top) during an invocation of the GPU scheduler. Note how the CPU timeline is entirely absent from the bottom of the figure—our solution does not rely on any CPU computations to execute a scheduling decision. Instead, our scheduler runs in-between each GPU task's execution (similar to how CPU schedulers are conventionally run between CPU tasks).

*Work supported by NSF grants CPS 2038960, CPS 2038855, CNS 2151829, and CPS 2333120, and ONR contract N0001424C1127.

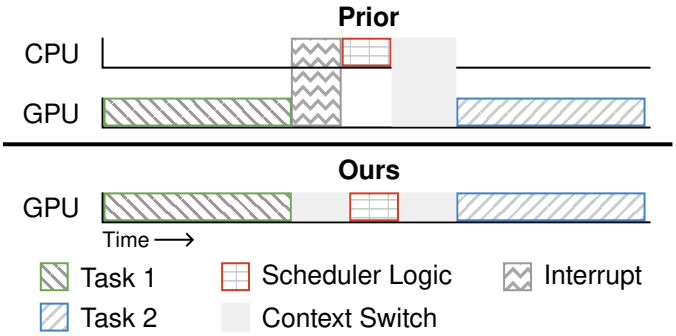

Fig. 1. Our scheduling technique removes high-overhead CPU–GPU synchronization operations by moving GPU scheduling off of the CPU.

Prior work. Most prior work has focused on eliminating a need for GPU scheduling by either allowing only one task to use the GPU at a time, or by subdividing the GPU into pieces that are each used only by one task at a time. GPUSync [1] is an exemplar of the earlier approach, whereas the SMLP [2], built on hardware [3], [4] or software-based GPU partitioning techniques [5]–[7], represents the latter approach. These approaches are limited in that they exclusively grant some portion of the GPU to a single task until that task yields access to the GPU; they have no way of evicting an overrunning task from the GPU. Without an eviction mechanism, an urgent GPU-using task may have to wait for less-important tasks to complete before gaining access to the GPU. This can make it impossible to guarantee that deadlines will be met. In contrast, preemptive GPU schedulers such as TimeGraph [8], Capedioci *et al.*'s EDF scheduler [9], and GCAPS [10] have the ability to preempt tasks on the GPU, ensuring that urgent GPU-using tasks will not have to wait for lower-priority ones before running.[1] These prior works only have acceptable overheads for scheduling algorithms that run infrequently, and cannot support the needs of frequently-run schedulers such as Pfair [11] or EEVDF [12].

Contributions. In this work, we:

1) Demonstrate that GPU scheduling from the CPU is fundamentally high-overhead.
2) Build a GPU scheduler supporting NVIDIA GPUs that runs entirely on the GPU.
3) Evaluate our scheduler, finding that it eliminates on-CPU overheads and reduces on-GPU overheads.

[1]TimeGraph cannot always preempt overrunning tasks due to contemporary hardware limitations, but does stop such tasks when possible.

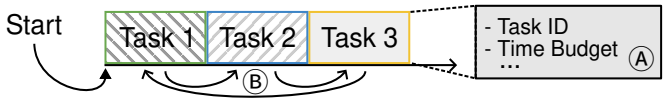

Fig. 2. Illustration of a GPU runlist of three tasks.

4) Test the analytical benefits of on-GPU scheduling, showing a 11.6–12.1% capacity improvement.

II. Background

A. Task Model

Each task i in our system has a *period* T_i and *worst-case execution time* C_i. Tasks release *jobs* every T_i time units, and each job may take up to C_i time units to execute. We assume implicit deadlines, such that each job must complete within T_i time units. The utilization of each task is $U_i = \frac{C_i}{T_i}$, and the total system utilization is $U = \sum_{i=0}^{n} U_i$, where n is the number of tasks. In this work, we treat the GPU and CPU as separate systems that communicate asynchronously. Each system has its own total utilization and set of tasks.

B. Built-in GPU Scheduling

Both NVIDIA and AMD GPUs support running multiple tasks on the GPU by rapidly switching between them, *i.e. time-slicing* them. On NVIDIA GPUs, this time-slicing is done by a built-in round-robin scheduler [9]. The set of tasks it runs is defined by a *runlist* (Fig. 2) that is provided by the GPU driver. Each entry in the runlist represents a task, a budget, and other state (Ⓐ in Fig. 2). The built-in scheduler processes the list in order, running each task for its specified budget before preempting it and switching to the next task (Ⓑ in Fig. 2). The driver sets each budget to 2 ms by default. A task forfeits the remainder of its budget if it finishes early. After reaching the end of the runlist, the built-in scheduler returns to the beginning of the runlist and repeats the scheduling process.

C. Fundamental Overheads of on-CPU GPU Scheduling

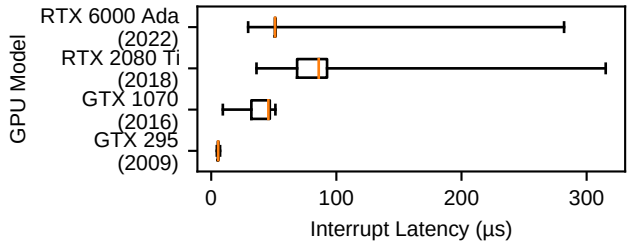

Fig. 3. 0, 25, 50, 75, and 100th percentile time between triggering an interrupt on the GPU and receiving it on the CPU (100 samples).

For a GPU scheduler such as TimeGraph [8], a principal overhead is the time required for the GPU to signal the CPU of the completion of a task via an interrupt, *i.e.*, the *interrupt latency* (squiggle in Fig. 1). We benchmarked this overhead on several generations of GPUs (methodology in Appx. A), and plot the results in Fig. 3.

Note how interrupt latencies have increased with time. On the GTX 295 (a variant of the GTX 285 used by TimeGraph [8]), the mean latency is 5 μs, but on the 13-years-newer RTX 6000 Ada, the mean latency is 56 μs. For schedulers which use interrupts to trigger the scheduler, these overheads are *fundamental*; it requires at least as much time as the interrupt latency for the CPU to be informed of an event on the GPU. These costs can compound quickly. For a scheduler signaled every 2 ms (*e.g.* Pfair [11] or EEVDF [12]), 56 μs would represent a 2.8% overhead cost, just for interrupts.

III. Our Solution: On-GPU Scheduling

Our solution eliminates the expense of CPU–GPU synchronization by making GPU scheduling decisions via a scheduling task running directly on the GPU. Fig. 1 shows a timeline comparing the two approaches during a task switch.

With the prior approach (top), the CPU must be signaled that the GPU has completed Task 1, must execute the scheduling algorithm, and must signal the GPU to switch to Task 2. On the RTX 6000 Ada, signaling the CPU via an inturrupt takes 50 μs on avg (Fig. 3), and switching between tasks tasks about 65 μs (in our tests). This implies that no matter how fast a GPU scheduler is at making scheduling decisions, it will always require at least 115 μs to switch between tasks. Assuming 50 μs to execute the scheduling algorithm and a 2 ms scheduling frequency (matching the built-in GPU scheduler), for every 2000 μs of CPU time, 165 μs (8.25%) would be lost to the scheduler while the GPU sits idle.

We instead use the time that the GPU would have sat idle to perform GPU scheduling directly on the GPU (bottom in Fig. 1). This eliminates the need for an interrupt, but introduces the need to switch to and from the GPU-scheduling task on the GPU. With some scheduler optimizations to reduce switching time,[2] the time required for this additional switch is about the same as the cost of an interrupt, meaning that it continues to take 165 μs to switch between tasks on the GPU.[3] However, all on-CPU overheads are eliminated, freeing up capacity for other CPU tasks.

IV. Implementation

We implemented our on-GPU scheduling framework on the NVIDIA RTX 2080 Ti and RTX 6000 Ada. We had to ensure that: **(1)** only tasks selected by our scheduler are run, **(2)** events such as task completion, creation, or destruction trigger our scheduler, **(3)** the driver cannot preempt our scheduler, and **(4)** our scheduler has equivalent privileges as the driver in accessing scheduling data structures and registers.

Controlling task selection. Our framework forces the build-in round-robin scheduler to run our selected task by only enabling at most two tasks in the runlist at a time: our scheduler task, and the task we would like to schedule. We then control how long the task we are scheduling may run for by setting its hardware-enforced budget (to, *e.g.*, 2 ms). After this budget expires, the build-in scheduler will switch back to our scheduling task. Our scheduler can then enable a different

[2]Thread-block-level preemption, also known as SM draining [13], allows for lower context switching times if tasks manually insert preemption points.

[3]This assumes that the GPU scheduling algorithm runs sufficiently fast on the GPU; we have observed this in practice.

task and repeat the process. This implements task selection and ensures that our scheduler is invoked on task completion.

Enabling event receipt. When a GPU task is created or destroyed, we want our scheduler to be triggered to handle the event, rather than allowing the built-in scheduler to take over. We discover that the GPU driver always writes tasks into the runlist in the order in which they were created, and always resets the built-in scheduler to the first entry of the runlist after a GPU task is created or destroyed. By always launching our GPU-scheduling task before any others, we can ensure it is at the start of the runlist, and that it will always be the first task to run after a GPU task is created or destroyed. We can then detect and handle such events in the GPU scheduler by scanning and detecting additions or removals from the runlist.

Running our scheduler non-preemptively. We prevent the driver or built-in scheduler from interrupting our scheduler by disabling instruction-level preemption for our scheduler task on the GPU. Instead, we use a preemption mode[2] which can only take effect at predefined points in a tasks' execution. We then insert said points between iterations of our scheduler, making each iteration of our scheduler execute non-preemptively.[4]

Accessing privileged registers and memory. To implement our scheduling framework, we need to be able to modify and resubmit the runlist from within a GPU task. These are privileged operations, as they require access to GPU control registers and physical memory. We address this problem by creating a Linux kernel module to modify the GPU page tables for our scheduling task, mapping in all of GPU physical memory and the GPU control registers.[5]

V. Evaluation

We evaluate the absolute overheads of on-GPU scheduling, and consider how the move from on-CPU to on-GPU scheduling effects the overall capacity of the system.

A. Absolute Overheads

To measure the absolute overheads of our approach, we compare how long it takes to complete a task set with NVIDIA's built-in scheduler vs. with our on-GPU scheduler configured to mimic the round-robin scheduling policy of the built-in scheduler. (We verified visually via the context-switch-tracing feature of NVIDIA's Nsight Systems profiler that both produce identical schedules.) The built-in scheduler uses a dedicated hardware unit and has no overhead cost, so any increase in execution time observed when running a task set under our scheduler is our scheduler's absolute overhead.

[4]Non-preemptive execution appears as a hang to the driver, and can trigger an attempted GPU reset. We disable the context-switch timeout watchdog to ensure that the GPU is not reset while our scheduler is running.

[5]GPU control registers are normally only visible to the CPU, and have no corresponding physical address on the GPU, making it impossible to map them into a GPU virtual address space normally. However, the GPU has a special type of virtual memory mapping used for accessing shared regions of CPU memory. We use this mapping type to map a set of CPU addresses which does not correspond to CPU physical memory, but corresponds to the PCIe control region for the GPU, enabling access to the GPU control registers.

TABLE I
Overhead of on-GPU Scheduling on RTX 6000 Ada

# Tasks	Time w/ Built-in (baseline) (ms)	Time w/ on-GPU (ours) (ms)	Overhead (% increase)
1	45,442	49,208	8.3%
2	94,218	98,578	**4.6%**
3	141,334	147,922	**4.6%**
4	188,450	198,021	5.0%
5	235,540	248,463	5.5%

We tested with task sets of 1–5 tasks on the RTX 6000 Ada, and present the results in Table I. This table shows how long each task set took to complete under each scheduler, and the percent increase in execution time incurred when our scheduler was enabled. Observe how the overhead of our approach is 4.6% at best, with a slow increase as the number of tasks increases. The increase is because our scheduling framework has a runtime complexity of $O(n)$; future implementations should be able to achieve $O(1)$. For one task, the overhead is large as it includes both running our scheduler and switching between tasks (the built-in scheduler need not switch between tasks when only one is present). For more than one task, tasks are already being switched between, so enabling our scheduler only adds the cost of our scheduler itself.

Returning to the cases with 4.6% overheads, since our scheduler is run once every 2 ms (to match the built-in one), a 4.6% overhead equates to 91 μs per scheduler invocation on average. This is significantly less than the 421-783 μs average per invocation in recent work [10]. Our overhead is even less than the mean interrupt latency on some GPUs (such as the 96 μs for the RTX 2080 Ti in Fig. 3)—on such GPUs our scheduler could *finish* before an on-CPU scheduler would be able to start.

B. Analytical Benefits to Schedulability

Eliminating the on-CPU overhead of GPU scheduling has an analytical benefit on both the CPU and GPU, but the benefit is greatest on the CPU. For a global scheduler, such as the global earliest-deadline-first scheduler (G-EDF), all tasks must have their execution times extended to account for the maximum time that they could be interrupted by an on-CPU GPU scheduler. We analyze how this affects the total capacity of the CPU via a *schedulability study*, a simulation-based way to evaluate how scheduler and overhead changes effect the overall ability of a system to guarantee met deadlines. Each experiment works by generating a set of tasks with a given total system utilization, and then testing if each scheduling approach can guarantee met deadlines for that task set.

In our study, we utilized the `schedcat` framework and included schedulability tests,[6] and considered both four- and eight-core systems of tasks with moderate periods and utilizations (as defined in prior work [14]). We charged each task 165 μs every 2 ms to represent the overhead of on-CPU GPU scheduling. We also compared against the cost of dedicating an entire CPU core to GPU scheduling (representing an on-CPU spin-based alternative to interrupt-based signaling), and

[6]https://github.com/brandenburg/schedcat; descendant from [14].

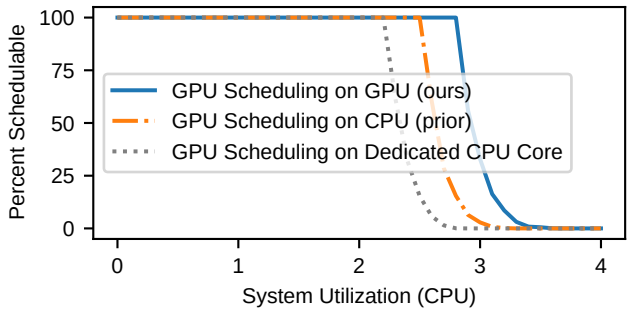

Fig. 4. Ratio of task sets that are schedulable for a given total CPU utilization on a four-core system. When the overhead of GPU scheduling is added, less capacity is available to execute CPU tasks.

to our approach (which does not incur any on-CPU overheads). We generated 2,000 task sets for every 0.25 increase in system utilization, and tested what proportion of task sets were schedulable using G-EDF under each scheduling approach. The results are shown in Fig. 4 for four cores. Each data point represents what portion of task sets were schedulable (vertical axis) for a given total system utilization (horizontal axis).

The area under each line is known as the *schedulable-utilization area*, and represents the set of task systems that each approach can schedule. By looking at the difference in these areas, we can estimate the total system capacity gained or lost by switching between approaches. In the case of four cores, switching to on-GPU scheduling increases total system capacity by 11.6% over on-CPU GPU scheduling, and by 26.0% over GPU scheduling with a dedicated core. In the case of eight cores, the gains are 12.1% and 12.6% respectively.

These results indicate that switching to on-GPU scheduling can increase the capacity on the CPU by 11% or more over the best alternative approach. We believe that the benefits of our approach may be substantially greater in real-world systems, as we purposefully assumed charitable overheads for on-CPU GPU scheduling—actual overheads for such competing approaches may be much higher [10].

VI. Conclusion

In this work, we presented on-GPU scheduling for NVIDIA GPUs and demonstrated how it can increase schedulability on the CPU by 11% or more without incurring high overheads on the GPU. Outstanding issues include a need to implement a real-time scheduler for the GPU using our framework, and a need to test how such changes effect average-case performance. Further, our scheduling framework does not use parallelism in its internal algorithms; as the GPU has many cores, a more-parallel implementation could reduce the rising overhead costs that we observe for higher task counts.

Appendix A
Methodology of Interrupt Experiments

All experiments were performed on an AMD 3950X-based Linux system, with kernel and driver versions as specified in Table II, and power management disabled.[7] We used a monitoring task, and a custom Linux hard-interrupt handler to record timestamps. Both were pinned to the same core, and used the same clock (`CLOCK_MONOTONIC_RAW`). Interrupts were generated by dereferencing a NULL-pointer on the GPU to trigger a page fault. Immediately before the dereference, the on-GPU task would flip a bit in shared memory to signal the monitor task to record a timestamp. Once the interrupt was received by the kernel, our custom handler would immediately record a timestamp, and then pass control onto the normal interrupt-handling path. To compute the interrupt latency, we took the difference between the two timestamps.

[7]Linux's power management was disabled by writing "n/a" to the file `/sys/devices/system/cpu/cpuX/pm_qos_resume_latency_us`.

TABLE II
Configurations Used for Interrupt Latency Experiments

GPU	Year	Kernel Version	Driver Version
GTX 295	2009	5.4.224-litmus+	340.108
GTX 1070	2016	5.4.224-litmus+	535.216.03
RTX 2080 Ti	2018	5.4.224-litmus+	535.216.03
RTX 6000 Ada	2022	6.8.0-52-generic	550.142

References

[1] G. A. Elliott, B. C. Ward, and J. H. Anderson, "GPUSync: A framework for real-time GPU management," in *RTSS*, Dec 2013.

[2] S. W. Ali, Z. Tong, J. Goh, and J. H. Anderson, "Predictable GPU sharing in component-based real-time systems," in *ECRTS*, Jul 2024.

[3] N. Otterness and J. H. Anderson, "AMD GPUs as an alternative to NVIDIA for supporting real-time workloads," in *ECRTS*, July 2020.

[4] J. Bakita and J. H. Anderson, "Hardware compute partitioning on NVIDIA GPUs," in *RTAS*, May 2023.

[5] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style GPU programming for GPGPU workloads," in *InPar*, May 2012.

[6] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, "Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations," in *ICS*, Jun 2015.

[7] C. Yu, Y. Bai, H. Yang, K. Cheng, Y. Gu, Z. Luan, and D. Qian, "SMGuard: A flexible and fine-grained resource management framework for GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 12, Jun 2018.

[8] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "TimeGraph: GPU scheduling for Real-Time Multi-Tasking environments," in *USENIX ATC*, Jun 2011.

[9] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-based scheduling for GPU with preemption support," in *RTSS*, Dec 2018.

[10] Y. Wang, C. Liu, D. Wong, and H. Kim, "GCAPS: GPU context-aware preemptive priority-based scheduling for real-time tasks," in *ECRTS*, Jul 2024.

[11] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.

[12] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *RTSS*, Dec 1996.

[13] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," in *ISCA*, Jun 2014.

[14] B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, UNC Chapel Hill, 2011.