Work-in-Progress: Impacts of Critical-Section Granularity When Accessing Shared Resources

Tanya Amert
Department of Computer Science
Carleton College
tamert@carleton.edu

Catherine Nemitz
Department of Mathematics and Computer Science
Davidson College
canemitz@davidson.edu

***Abstract*—The prevalence of computer-vision applications in autonomous vehicles necessitates the use of graphics processing units (GPUs), which must be shared among tasks due to size, weight, power, and cost constraints. Such sharing is typically managed via locking protocols. However, experiments reveal a trade-off in the granularity of such sharing; GPU operations' execution times are reduced if multiple GPU accesses are grouped in a single lock request (*i.e.*, form a single *critical section*) rather than assuming only one access per critical section. This grouping exposes a broader trade-off between the extra lock overhead (and blocking) incurred by an individual task, and the analytical blocking experienced by all other tasks in the system. This trade-off is expressed herein via an extended resource model and explored using example task systems, demonstrating the impact of different access-grouping heuristics.**

***Index Terms*—real-time systems, synchronization, graphics processing units**

I. Introduction

With the ever-growing array of driver-assist features available on new vehicles and full autonomy looming in the future, computer-vision (CV) applications are needed to perceive the surrounding world. Such CV applications typically require specialized hardware accelerators, such as graphics processing units (GPUs). Due to stringent constraints including size, weight, power, and cost, these GPUs must be shared among multiple CV applications on the same platform.

Unfortunately, managing shared hardware resources is challenging in general, and is especially problematic for GPUs. Preempting computations is often impractical, and the built-in scheduling behaviors of NVIDIA GPUs can lead to unexpected delays [11]. Locking protocols can be used to arbitrate access to these shared GPUs [8].

Requiring mutually exclusive GPU accesses gives rise to a trade-off: the more computations a task performs while holding the lock (*i.e.*, in a *critical section*), the sooner it completes, as it does not incur the additional overhead of multiple lock and unlock requests; however, the additional computations induce longer blocking times experienced by other tasks. Before describing our contributions regarding this trade-off in more detail, we first briefly overview related work on GPU sharing and trade-offs in computation granularity.

This work was supported by NSF grants CPS 1837337, CPS 2038855, and CPS 2038960, and ONR grant N00014-20-1-2698.

Related work. Amert *et al*. [1] presented TimeWall, a framework that provides temporal isolation for CPU+GPU applications. Although TimeWall allows multiple GPU accesses to be performed as part of a single critical section, an exploration of the impact of doing so was left to future work. In Sec. II, we present the results of such experiments.

Other approaches in GPU management include modifications to open-source portions of the NVIDIA scheduler to enable deadline-driven scheduling [7] or treating GPU computation and copy engines as mutually exclusive resources [8]. However, to our knowledge, none of the prior work explored varying the groupings of computations into critical sections.

Voronov *et al*. [10] provided heuristics to merge nodes in graph-based applications, aiming to reduce overall analytical response-time bounds. Others have considered limited preemptive scheduling [5], [9], which reduces overhead by only allowing preemption at certain points in tasks' execution. Although preemption points effectively group accesses into larger non-preemptive regions, such non-preemptivity precludes the possibility of concurrent CPU+GPU execution by different tasks. In prior work on both limited-preemption scheduling and graph node merging, blocking time due to shared resources was assumed to be already included in per-task worst-case execution times. On the contrary, we explore the trade-off that arises when the worst-case blocking is not assumed as a given.

Contributions. In this paper, we present three contributions. First, we detail our preliminary experiments grouping GPU accesses within a single critical section. In our experiments, grouping multiple GPU accesses in one critical section reduced 99.9^{th}-percentile access durations by up to 68%.

Second, we provide an extended resource model that enables us to explore the trade-off in resource-access granularity and blocking time; we note that this model applies generally to any shared resources, not just GPUs.

Finally, we use our model to explore this trade-off in a case-study evaluation of example task systems, demonstrating the differences in tasks' repsonse times when different heuristics are used to determine the critical-section granularity.

Organization. The remainder of this paper is organized as follows. We detail our motivating GPU experiments in Sec. II, describe our extended resource model in Sec. III, illustrate its implications in Sec. IV, and conclude in Sec. V.

II. Motivating Experiment

We now describe the setup and discuss the results of our GPU-access-grouping experiments.

A. Experimental Setup

We now describe our experimental platform and the workload we executed upon it.

Platform. Our experiments were performed on a machine with two eight-core 2.10-GHz Intel Xeon Silver 4110 CPUs and one NVIDIA Titan V GPU. Each CPU core has a pair of 32-KB L1 instruction and data caches, and a 1-MB L2 cache; all eight CPU cores on a socket share an 11-MB L3 cache. To mitigate interference, we disabled hyperthreading and graphics output.

Workload. Our workload was comprised of a GPU-based CV application called *Histogram of Oriented Gradients* (HOG). The `hog` workload, based on the CV library OpenCV [2], performs pedestrian detection by processing images at 13 different resolution levels. For each image (a single frame of a video) and for each level, there are five GPU computations,[1] K1–K5, as depicted in Fig. 1.

We executed two instances of the `hog` program for 5000 frames each, and measured the duration of each GPU computation on the CPU using `clock_gettime()`. To ensure that only one of the two instances could use the GPU at any given time, we arbitrated GPU access using a locking protocol [1]. We considered three different locking configurations, separately grouping GPU copy and computation operations, which we collectively refer to as *accesses*:

- `GROUP-NONE`: No accesses were grouped together. Processing one image required 78 lock requests (one copy-in, $(13 \cdot 5) - 1$ computations, and 13 copy-out operations).
- `GROUP-MINOR`: Each K3–K5 sequence occurred within a single critical section; copy-out operations were minorly grouped. Thus, processing one image involved $1 + 13 \cdot 3 + \lceil 13/3 \rceil = 45$ lock requests.
- `GROUP-MAJOR`: All computations were grouped into one critical section (K2–K5 for the first level, K1–K5 for the others); all copy-out operations were grouped. Processing one image required $1 + 13 + 1 = 15$ lock requests.

We considered the workload within the context of a larger system, and therefore executed our workloads as real-time tasks within LITMUS$^{\text{RT}}$ [6]. We allocated only four of the CPUs to our `hog` tasks and distributed the 11-MB L3 cache such that the GPU-using tasks were allotted only 4.4-MB of the total L3 cache. For the purpose of measuring GPU access times, however, we did not execute any additional workloads. Each `hog` task executed with a period of 20 ms.

B. Results

The 50th, 99th, and 99.9^{th}-percentile GPU-access-duration timings are given in Tbl. I for the computations performed in each of the three configurations.[2]

[1] For the original image resolution, computation K1, which resizes the image, is skipped.

[2] The worst-case measurements include extreme outliers; these outliers were the subject of prior work [1], so they are not discussed here.

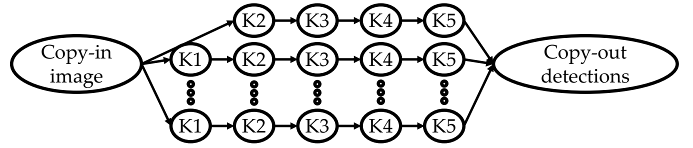

Fig. 1. Simplified structure of the `hog` application, comprising copying image data onto the GPU, computations (labeled K1–K5), and copying detection results off of the GPU.

TABLE I
GPU-access durations (in microseconds) measured on the CPU for each of the access-grouping configurations we considered.

Configuration	Statistic	K1	K2	K3	K4	K5
GROUP-NONE	50th	120	125	150	123	140
	99th	145	146	177	145	171
	99.9^{th}	150	152	191	150	182
GROUP-MINOR	50th	120	125	151	24	41
	99th	146	146	184	34	65
	99.9^{th}	153	152	191	48	73
GROUP-MAJOR	50th	121	25	46	23	40
	99th	144	140	73	31	63
	99.9^{th}	153	150	90	48	76

Observation 1. Grouping GPU accesses reduces the durations of later accesses within the same critical section.

Comparing `GROUP-NONE` and `GROUP-MINOR`, 99.9^{th}-percentile measurements of K4 and K5 were reduced by up to 68% when they occurred immediately following K3 without lock and unlock calls in between. Median measurements under `GROUP-MAJOR` were even more drastic: K2's duration was reduced by 80%.

Observation 2. The number of prior accesses within a critical section does not impact the reduction in duration of subsequent grouped GPU accesses.

This can be seen in comparing measurements for K4 and K5 using the `GROUP-MINOR` and `GROUP-MAJOR` configurations.

III. System Model

We now describe our task model and resource model, as well as our enhancements to express the grouping of multiple accesses within a single lock request.

A. Task Model

We consider a set $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$ of n tasks, where each task τ_i releases a (possibly infinite) sequence of jobs, $J_{i,1}, J_{i,2}, ...$; we refer to a generic job of task τ_i as $J_{i,j}$.

Task τ_i is *periodic* if it releases a new job every T_i time units, and *sporadic* if T_i is a lower bound on the time between job releases. Each job $J_{i,j}$ must complete execution by a deadline $D_i \leq T_i$ time units after its release. Ignoring shared resources, we represent a task as the three-tuple $\tau_i = (T_i, C_i, D_i)$, where C_i is the worst-case execution time of each job $J_{i,j}$. If $T_i = D_i$ we can simplify this to $\tau_i = (T_i, C_i)$.

B. Resource Model

Although we consider independent tasks, they may need to access a shared resource.[3] When job $J_{i,j}$ requires access to a shared resource, it makes a *lock request* for the lock protecting that resource. It *acquires* the lock once the request is satisfied, and it then *holds* the lock (and is thus guaranteed mutually exclusive access to that resource) until it *releases* the lock via an *unlock request*. The instructions executed while a job holds the lock comprise a *critical section*; each task can thus be represented as an alternating sequence of *unprotected sections* and *critical sections* of code, as illustrated in Fig. 2 (a).

C. Request Model

A critical section is typically comprised of a single access to a shared resource. We extend this by allowing a critical section to consist of multiple individual accesses. We let α_i^k refer to the kth shared-resource access (for $k \geq 1$) made by a job of task τ_i. We define γ_i^k as the corresponding kth non-access segment, where γ_i^0 is the sequence of instructions executed before the first access; γ_i^k immediately follows α_i^k. A critical section is thus a contiguous sequence of access segments, with interleaving non-access segments, as depicted in Fig. 2 (b).

Given this access-focused viewpoint, a task τ_i can be represented as the alternating sequence $\gamma_i^0, \alpha_i^1, \gamma_i^1, \alpha_i^2, \gamma_i^2, \ldots$. Any contiguous sequence of segments $\alpha_i^p, \gamma_i^p, \ldots, \alpha_i^q$ for $p < q$ can be grouped together into a single critical section, as long as no segment is part of two critical sections.

We now expand our task representation to the four-tuple $\tau_i = (T_i, \Gamma_i, A_i, D_i)$, where Γ_i is a list of the durations of non-access segments $\gamma_i^0, \gamma_i^1, \ldots$, and A_i is a list of the durations of access segments $\alpha_i^1, \alpha_i^2, \ldots$. If $T_i = D_i$, we can simplify this representation to $\tau_i = (T_i, \Gamma_i, A_i)$. We assume C_i is an upper bound on the summed durations of all segments in Γ_i and A_i, assuming a given grouping of accesses into critical sections.

Example 1. Consider a task system comprised of two tasks, which make one and three accesses, respectively: $\tau_1 = (140, [30, 30], [10])$ and $\tau_2 = (250, [20, 10, 20, 20], [10, 10, 10])$. These two tasks are illustrated in Fig. 3, along with two possible groupings of each task's accesses into critical sections. ◊

D. Access Model

Motivated by the experimental results given in Tbl. I, we consider an access itself to be decomposed into multiple intervals, including pre-access and post-access operations as well as the fundamental access operation itself.

For example, on a GPU, some pre-access operations (e.g., loading instructions or data from memory) may be faster if multiple accesses are performed back-to-back within a single critical section. The observed duration of an access is then closer to only the actual computation, for example, the instructions executed to convert an image to grayscale.

[3]We assume there is only one shared resource, such as a single GPU, and leave the extension to multiple resources as future work.

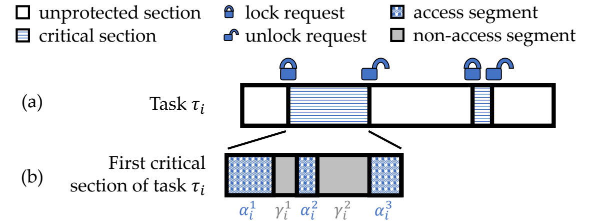

Fig. 2. The decomposition of a task and a critical section.

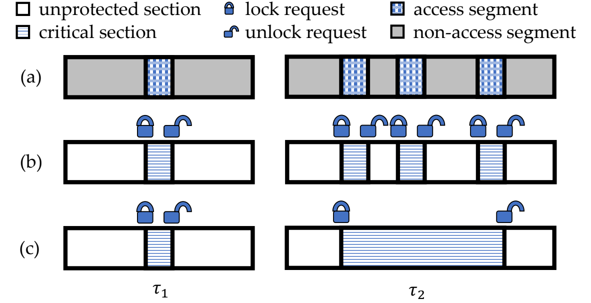

Fig. 3. The task system described in Ex. 1: (a) divided into segments, (b) with one access per critical section, and (c) with all accesses grouped.

IV. Case Study: Non-Preemptive Protocol

To consider the trade-offs in critical-section granularity, we begin by focusing on a single system type and then describe how different choices of lock granularity for our example from Sec. III would impact response times and ultimately, schedulability. Finally, we present an example in which access times themselves are impacted by critical-section granularity.

A. Scheduling Algorithm and Resource Access Protocol

For our case study, we focus on the use of a fixed-priority scheduling algorithm, deadline monotonic (DM), in which task priorities are assigned by non-increasing deadline. Let $hp(\tau_i)$ (resp., $lp(\tau_i)$) be the set of tasks with priority higher (resp., lower) than that of τ_i. Under DM scheduling, the worst-case response time, R_i, of a task τ_i is given by:

$$R_i = B_i + C_i + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j$$

Here, B_i is the worst-case priority-inversion blocking incurred by task τ_i, which depends on the maximum critical-section duration of tasks in $lp(\tau_i)$. For simplicity, we assume the Non-Preemptive Protocol (NPP) is used to protect access to shared resources [4]. Under the NPP, a task executes non-preemptively during its entire critical section. Thus, if a higher-priority job is released immediately after a lower-priority job acquires the lock,[4] the higher priority job is blocked for the remaining duration of the critical section.

As described in Sec. III-C, a critical section may be comprised of multiple resource accesses. Each critical section begins with acquiring the lock and ends with releasing the lock. A given locking protocol adds overhead for both the

[4]Although the NPP inherently prevents any contention for such a lock, we describe its execution in terms of lock and unlock calls to reflect how other resource-access protocols function.

lock and *unlock* procedures, represented by $\mathcal{O}_{\text{lock}}$ and $\mathcal{O}_{\text{unlock}}$, respectively. For simplicity, we assume that $\mathcal{O}_{\text{lock}}$ and $\mathcal{O}_{\text{unlock}}$ contribute to blocking of other tasks, and thus include these terms in the duration of a given critical section. In the future, we will explore more precise overhead accounting [3].

B. Impact of Critical-Section Granularity

We now explore the impact of access grouping on schedulability using the example tasks introduced in Sec. III-C.

Example 1 (cont'd)**.** Tasks τ_1 and τ_2 share a resource. Under DM scheduling, τ_1 has higher priority. For this system, we assume that the overhead of the locking protocol, the NPP, is fixed (*i.e.*, does not depend on the task acquiring or releasing the lock), with $\mathcal{O}_{\text{lock}} = 2$ and $\mathcal{O}_{\text{unlock}} = 1$.

Task τ_1 makes one access to the shared resource and thus has one critical section with duration $\mathcal{O}_{\text{lock}} + |\alpha_1^1| + \mathcal{O}_{\text{unlock}} =$ 13 time units. Thus, $C_1 = |\gamma_1^0| + 13 + |\gamma_1^1| = 73$ time units.

If no accesses by task τ_2 are grouped, as in Fig. 3 (b), then each of the three critical sections has duration $\mathcal{O}_{\text{lock}} + |\alpha_2^k| + \mathcal{O}_{\text{unlock}} = 13$ time units and thus $C_2 = \sum_k |\gamma_2^k| + 3 \cdot 13 = 20 + 10 + 20 + 20 + 39 = 109$ time units.

If all three accesses of τ_2 are grouped, as in Fig. 3 (c), the critical section has duration $\mathcal{O}_{\text{lock}} + |\alpha_2^1| + |\gamma_2^1| + |\alpha_2^2| + |\gamma_2^2| + |\alpha_2^3| + \mathcal{O}_{\text{unlock}} = 2 + 10 + 10 + 10 + 20 + 10 + 1 = 63$ time units and $C_2 = |\gamma_2^0| + 63 + |\gamma_2^3| = 20 + 63 + 20 = 103$. ♢

However, the choice of lock granularity does not occur in isolation. This choice can have a large impact on critical-section duration, and in turn, blocking and schedulability.

Example 1 (cont'd)**.** For τ_1, with the most granularity, we solve $R_1 = 13 + 73 = 86$. Instead, with the least granularity, $R_1 = 63 + 73 = 136$. As $R_1 < D_1 = 140$, τ_1 is schedulable.

As τ_2 is the lowest priority task, $B_2 = 0$. Thus, with the most granularity, we solve $R_2 = 109 + \left\lceil \frac{R_2}{140} \right\rceil \cdot 73 = 255 > 250 = D_2$; task τ_2 is not schedulable. If instead we choose to group all accesses, we have $R_2 = 103 + \left\lceil \frac{R_2}{140} \right\rceil \cdot 73 = 249 < D_2$, and task τ_2 is schedulable. ♢

We now consider an example that is only schedulable if accesses are not grouped.

Example 2. Consider tasks $\tau_3 = (130, [30, 30], [10])$ and $\tau_4 = (260, [20, 10, 20, 20], [10, 10, 10])$. If all accesses are grouped, then $R_3 = 136 > D_3$ and $R_4 = 249 < D_4$ and the task system is not schedulable. If no accesses are grouped, $R_3 = 86 < D_3$ and $R_4 = 255 < D_4$; the task system is schedulable. ♢

The above examples illustrate the complexity in this trade-off. We propose the following heuristics: (1) highest-priority tasks should have all accesses grouped, and (2) lower-priority tasks should have accesses grouped such that the system remains schedulable. We leave to future work determining optimal access groupings, *e.g.*, via a linear program.

C. Granularity-Dependent Resource Access Times

We now consider the implications of the previous examples of the GPU-using task as described in Sec. II. We look at a single task processing one image resolution.

Example 3. Consider a task with access durations based on the 99.9^{th} percentile of K1–K5 observations under `GROUP-NONE` in Tbl. I. Let $\tau_5 = (20000, [20, 20, 20, 20, 20, 20], [150, 152, 191, 150, 182])$. The values in Γ_5 represent short interleaving computations. Assuming negligible overheads, any access grouping results in $C_5 = 945$.

If we instead group accesses using the `GROUP-MAJOR` approach and apply the 99.9^{th}-percentile `GROUP-MAJOR` results for K1–K5, we have $C_5 = 637$. This significant time reduction is independent from any time saved by reducing the number of lock/unlock calls. ♢

Ex. 3 supports the idea in Sec. III-D that GPU-based tasks may have some minimum baseline of execution, along with additional operations when interleaved with other work. An exploration of such effects will be a focal point of future work.

V. Conclusion

In this paper, we presented an extended request and access model for shared resources, which was motivated by critical-section-granularity experiments using a GPU with a real CV application. We demonstrated, via an example using DM+NPP scheduling, the trade-off between critical-section granularity and the analytical blocking imposed upon other tasks.

Our next steps include extending the resource model to include multiple resources, and extending proposed heuristics to other schedulers and other locking protocols. We also plan to further explore the access-duration reduction due to grouping, and express the grouping of accesses into critical sections as a linear program.

References

[1] T. Amert, Z. Tong, S. Voronov, J. Bakita, F.D. Smith, and J. Anderson. TimeWall: Enabling time partitioning for real-time multicore+accelerator platforms. In *RTSS 2021*.

[2] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[3] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-time Operating Systems*. PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 2011.

[4] G. Buttazzo. *Hard real-time computing systems: Predictable scheduling algorithms and applications*. Springer US, 3rd edition, 2011.

[5] G. Buttazzo, M. Bertogna, and G. Yao. Limited preemptive scheduling for real-time systems: A survey. *IEEE transactions on Industrial Informatics*, 9(1):3–15, 2012.

[6] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\text{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *RTSS 2006*.

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

[8] G. Elliott, B. Ward, and J. Anderson. GPUSync: A framework for real-time GPU management. In *RTSS 2013*.

[9] M. Nasri, G. Nelissen, and B. Brandenburg. Response-time analysis of limited-preemptive parallel DAG tasks under global scheduling. In *ECRTS 2019*.

[10] S. Voronov, S. Tang, T. Amert, and J. Anderson. AI meets real-time: Addressing real-world complexities in graph response-time analysis. In *RTSS 2021*.

[11] M. Yang, N. Otterness, T. Amert, J. Bakita, J. Anderson, and F.D. Smith. Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems. In *ECRTS 2018*.