# Challenges in Real-Time GPU Management*

Tanya Amert
The University of North Carolina at Chapel Hill
tamert@cs.unc.edu

## ABSTRACT

Existing work on response-time analysis for real-time task systems represented as processing graphs containing cycles does not properly handle GPU-using tasks when intra-task parallelism is restricted. For such graphs to be schedulable, the execution of GPU-using tasks within the cycles may need to be prioritized over that of other tasks. This paper presents possible approaches for managing tasks using NVIDIA GPUs while allowing some tasks to be prioritized over others, and explores the tradeoff that results between response times of various tasks.

## Keywords

GPUs, CUDA, real-time GPU management

## 1. INTRODUCTION

Advanced driver assist systems are enabled by a range of computer vision (CV) applications. These applications typically rely on cameras as an input modality. Processing of camera feeds for CV applications is greatly accelerated by graphics processing units (GPUs). Such safety-critical systems must be certified, so response-time bounds for CV applications executed on GPUs are a necessity.

However, bounding response times of tasks executed on a GPU is extremely challenging. NVIDIA's Drive-PX2 series of GPU-equipped embedded platforms are specifically designed for automotive applications, yet necessary scheduling details of NVIDIA GPUs are not available without a restrictive non-disclosure agreement, and their drivers are typically closed source. As a result, existing response-time bounds for such GPUs are not tight, and modifications to the internal scheduling policies are difficult to make. Although AMD provides open-source drivers for its GPUs, AMD is not as widely adopted by automotive companies.

**Contributions.** In this paper, we explore approaches to real-time GPU management that enable tighter response-time bounds for GPU-using tasks while allowing for prioritizing of some GPU-using tasks over others.

**Organization.** This paper is organized as follows. We provide an overview of NVIDIA GPU software and hardware in Sec. 2 and discuss related work in Sec. 3. We then explore various approaches to managing GPU execution in Sec. 4, and conclude in Sec. 5.

*Work supported by NSF grants CNS 1409175, CNS 1563845, CNS 1717589, and CPS 1837337, ARO grant W911NF-17-1-0294, and funding from General Motors.

**Listing 1** Vector Addition Pseudocode.

```
1: kernel vecAdd(A ptr to int, B: ptr to int, C: ptr to int)
       // Calculate index using built-in thread, block info
2:     i := blockDim.x * blockIdx.x + threadIdx.x
3:     C[i] := A[i] + B[i]
4: end kernel

5: procedure main
       // (i) Allocate GPU memory for arrays A, B, and C
6:     cudaMalloc(d_A)
7:     ...
       // (ii) Copy arrays A and B from CPU to GPU memory
8:     cudaMemcpy(d_A, h_A)
9:     ...
       // (iii) Launch the kernel
10:    vecAdd<<<numBlocks, threadsPerBlock>>>(
           d_A, d_B, d_C)
       // (iv) Copy results from GPU to CPU array C
11:    cudaMemcpy(h_C, d_C)
       // (v) Free GPU memory for arrays A, B, and C
12:    cudaFree(d_A)
13:    ...
14: end procedure
```

## 2. BACKGROUND

In this section, we provide an overview of the basics of NVIDIA GPUs.

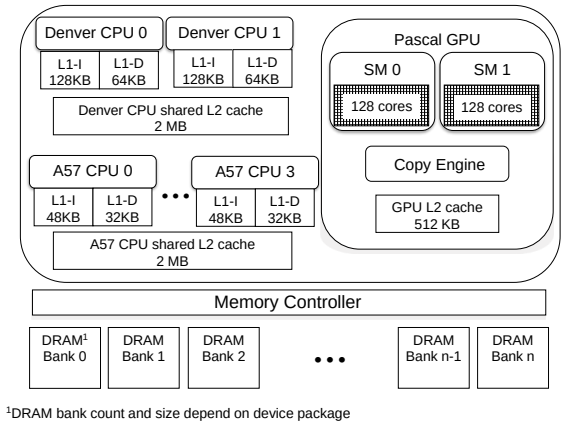
### 2.1 CUDA

CUDA is a C/C++ extension developed by NVIDIA to allow programmers to submit instructions to the GPU [8]. The CUDA API includes commands to copy data between the host CPU and the device GPU, as well as between multiple GPU devices, and to submit programs called *kernels* for execution on the GPU.

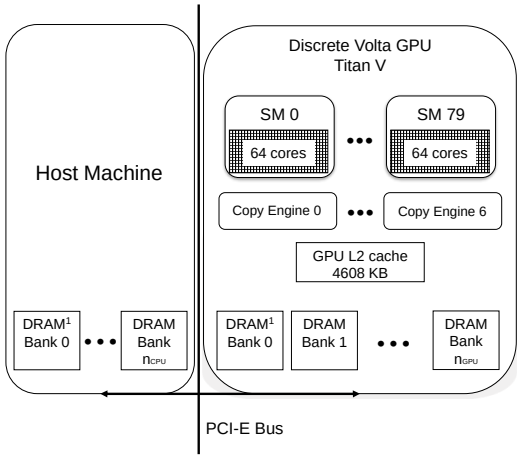
The general structure of a CUDA program is illustrated in Listing 1. This includes (i) allocating memory on the GPU, (ii) copying data to the GPU, (iii) executing one or more kernels on the GPU, (iv) copying the results back from the GPU, and (v) freeing any allocated memory on the GPU.

Kernel execution is divided into groups of 32 *threads*, called *warps*. Each thread in a warp executes in a SIMD fashion, using built-in CUDA variables to determine the data for that thread (Line 2). Warps are combined into *blocks*, which are further grouped into *grids*. As shown in Line 10, the command to issue a kernel requires the programmer to specify the layout of threads into grids and blocks (`numBlocks` and `threadsPerBlock`, respectively).

Each CUDA commands is issued to a *stream*, which is a FIFO queue of operations. By default, all CUDA commands are issued to the default stream, called the *NULL stream*.




(a) NVIDIA TX2




(b) NVIDIA Titan V

**Figure 1: Comparison of two GPUs: (a) NVIDIA TX2 (integrated) and (b) Titan V (discrete).**

Use of the NULL stream greatly reduces parallelism, and user-defined streams can be used to allow operations to execute in parallel, and thus better utilize powerful GPUs.

## 2.2 NVIDIA GPU Hardware

NVIDIA GPUs contain multiple copy engines (CEs), used to copy data to and from the GPU, and an execution engine (EE), comprised of multiple *streaming multiprocessors* (SMs). An SM can service up to 2048 threads at once. Integrated GPUs typically have an order of magnitude fewer SMs than discrete GPUs.

Example 1. *Details of two NVIDIA GPUs are depicted in Fig. 1. NVIDIA's TX2 system-on-chip (inset (a)) has a GPU comprised of one CE and two SMs. In contrast, the discrete Titan V (inset (b)) boasts seven CEs and 80 SMs.*

An SM is comprised of a number of hardware cores, serviced by a small number of warp schedulers. A warp scheduler hides memory latency by determining which of four warps should execute an instruction at a given time instant. For the rest of this paper, we consider scheduling at the thread and block level, and ignore warps, as their details are much more difficult to measure at runtime.

# 3. RELATED WORK

In this section, we discuss three groups of related work. In doing so, we consider the following different dimensions of GPU execution: whether the GPU is treated as a uniprocessor or multiprocessor device, if the approach uses locking protocols to synchronize GPU access or considers scheduling techniques, whether the GPU is accessed by one or multiple processes, and if one or multiple GPUs are available in a system. We consider related work first based on the choice of scheduling or synchronization as a management technique, and discuss the remaining dimensions later.

**Scheduling.** The NVIDIA documentation omits important details regarding GPU scheduling policies. To this end, Otterness *et al.* explored the behavior of kernels submitted from different processes (thus executed via timeslices) [9]. Amert *et al.* extended this exploration to GPU access from within a single process [1]; they described a set of rules dictating the order in which GPU operations (kernel executes or copy operations) execute on their respective GPU engines, and provided microbenchmark experiments to validate this behavior. They additionally studied the behavior when both user-defined streams and the NULL stream are used, as well as two different stream priority levels. Recently, Yang *et al.* extended this work to provide response-time bounds for GPU-using CV applications expressed as DAGs [10].

Capodieci *et al.* demonstrated a real-time scheduler for GPU operations [3]. They implemented a software scheduler module within the NVIDIA hypervisor's runlist manager, enabling preemptive earliest-deadline first (EDF) scheduling on the Drive-PX2. However, their approach requires the open-source drivers available only for NVIDIA's Drive-PX2 embedded platform, and some details and their source code were not made available due to non-disclosure agreements.

**Synchronization.** Access to the GPU can alternatively be arbitrated by a locking protocol. This approach was taken in developing GPUSync [4], a real-time GPU management framework. GPUSync utilizes $k$-exclusion locks to allow mutually exclusive access to a number of GPUs in a system. Implemented in the kernel, GPUSync is available as a fork [5] of the LITMUS$^{\mathrm{RT}}$ kernel [6].

**Remaining dimensions.** Of the described related work, both GPUSync and Capodieci *et al.* treat the GPU as a uniprocessor. For less-powerful embedded GPUs, this is a reasonable assumption. However, computations performed on a powerful GPU such as the Titan V might not require all of the GPU's many SMs, resulting in wasted GPU capacity. In that case, the multiprocessor-scheduling-based approaches of [1, 9, 10] might be more appropriate.

Simultaneous GPU access from different processes can greatly impact response times [9]. For discrete GPUs, the NVIDIA Multi-Process Service (MPS) allows the scheduling rules detailed in [1] to apply even when different processes access the GPU; without MPS, kernels submitted from different processes execute in timeslices based on the runlist, and thus do not truly execute concurrently on the GPU.

The above related work can also be applied for systems with multiple GPUs. The CUDA API allows the programmer to specify to which GPU an operation is submitted. The EDF approach from [3] applies if tasks are partitioned to the different GPUs, and GPUSync directly allows for multiple GPUs via its $k$-exclusion locking protocol to manage access.

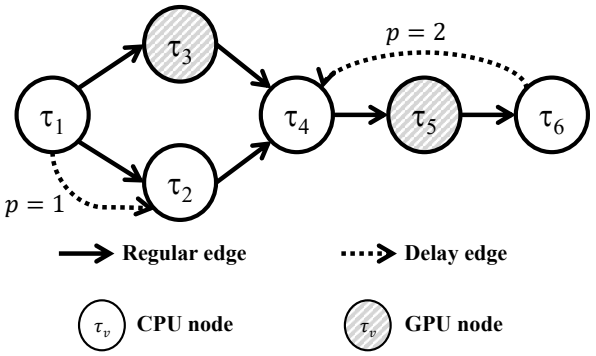



**Figure 2: A sporadic task graph with CPU and GPU nodes and one cycle.**

# 4. REAL-TIME GPU MANAGEMENT

In this section, we formalize our problem and present potential approaches to real-time GPU management that prioritize tasks within cycles.

## 4.1 Motivation

Processing graphs can be used to represent CV applications. Nodes represent tasks that perform some computation, and edges correspond to the data dependencies between tasks. *Delay edges* indicate that the data involves results from a prior time step (in CV applications, a time step typically corresponds to a video frame used as inupt). Corresponding to each delay edge is a value $p$ indicating the age of the historical data dependency.

EXAMPLE 2. *An example graph is depicted in Fig. 2. The edges from $\tau_1$ to $\tau_2$ indicate that a job of $\tau_2$ uses the outputs of the jobs of $\tau_1$ from the current and previous time steps.*

Cycles induced by the presence of delay edges can wreak havoc on existing real-time analysis. If some delay edge in a cycle has $p = 1$, then some task in that cycle depends on the output of the prior frame; as a result, no two jobs of any task in that cycle can possibly execute in parallel.

EXAMPLE 2. (cont'd) *In Fig. 2, the delay edge from $\tau_6$ to $\tau_4$ results in a cycle. As $p = 2$ for that edge, no more than two jobs of any task in $\{\tau_4, \tau_5, \tau_6\}$ can execute in parallel.*

The implicit-deadline sporadic task model is given as $\tau_i = (\Phi_i, T_i, C_i)$, where $\Phi_i$, $T_i$, and $C_i$ represent the task's phase offset, period, and worst-case execution time, respectively. The utilization of task $\tau_i$ is defined as $u_i = C_i/T_i$.

Assume multiple jobs of a task may execute concurrently. If the utilization of a cycle is greater than 1.0, then jobs of tasks in that cycle cannot be scheduled sequentially without incurring unbounded response times; if $u_{cycle}$ is the utilization of a cycle in the graph, then at least $\lceil u_{cycle} \rceil$ jobs of each task in the cycle must be allowed to execute simultaneously. However, the data dependency introduced by a cycle requires that no more than $p$ jobs of a given task may execute at once. Thus, tasks have *restricted intra-task parallelism*.

In an upcoming paper, Amert *et al.* [2] propose a new *rp-sporadic task model* and provide the corresponding analysis for sporadic task graphs containing cycles with utilization greater than 1.0. The rp-sporadic task model encodes the necessary intra-task parallelism as an additional task parameter: $\tau_i = (\Phi_i, T_i, C_i, P_i)$. In this model, $P_i$ is the maximum possible *intra-task parallelism* for the task.

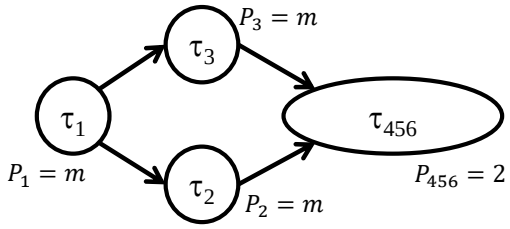



**Figure 3: The sporadic task DAG corresponding to the graph in Fig. 2.**

## 4.2 Proposed Approaches

The analysis presented in [2] reveals a circularity: to determine the response-time bound for CPU tasks, it is necessary to know the response-times of the GPU tasks, and vice versa. The GPU response-time analysis in [10] assumes full intra-task parallelism ($P_i = \infty$) for GPU tasks, so it does not apply to the rp-sporadic task model. Instead, Amert *et al.* break this circularity by assuming that access to the GPU is arbitrated with a simple FIFO mutex lock, and thus treat GPU blocking time as CPU execution time. They also combine cycles into *supernodes*; the resulting DAG corresponding to the graph from Fig. 2 is shown in Fig. 3. However, the response-time bounds derived in [2] are conservative for powerful GPUs such as the Titan V.

We now explore real-time GPU management approaches that enable more precise accounting of response times of GPU tasks under the rp-sporadic task model, and aim specifically to minimize the response times for GPU nodes that are part of cycles. We first assume each approach is taken individually; later we remove this assumption.

As in the graph in Fig. 2, we assume that each task executes on either the CPU or the GPU. For our purposes, we consider only the GPU tasks, which we divide into two sets: $\mathcal{H}$ and $\mathcal{L}$, corresponding to nodes that are (high priority) and are not (low priority) part of a cycle, respectively.

EXAMPLE 2. (cont'd) *In Fig. 2, $\mathcal{H} = \{\tau_5\}$ and $\mathcal{L} = \{\tau_3\}$.*

**Multiple GPUs.** In a system with many GPUs, each cycle can be assigned to its own GPU. If GPUs are not so abundant, a subset of the GPUs can be designated for use only by tasks in $\mathcal{H}$, with access arbitrated by a $k$-exclusion locking protocol, as in GPUSync [4]. If some task in $\mathcal{H}$ may access more than one GPU simultaneously, a locking protocol for *replicated resources* [7] can instead be used to manage access. This might be the case if multiple GPU operations are issued by a single task.

**Multi-Process Service.** In addition to allowing different processes' kernels to execute according to the rules in [1], MPS allows the programmer to specify a fraction of the GPU available to a given process.[1] Thus, MPS can be used to divide a GPU into two or more "virtual GPUs" (vGPUs); the scheduling rules of [1] apply to each vGPU.

EXAMPLE 3. *An example execution pattern is shown in Fig. 4. In this experiment, two processes each submitted a single kernel to a Titan V. Under MPS, each process was specified to utilize 40% of the GPU, i.e. 32 of the 80 total SMs. Each process submitted a kernel comprised of 160*

[1]MPS can only be used on discrete GPUs, and thus is not available for integrated GPUs, such as on NVIDIA's TX2 or Drive-PX2.

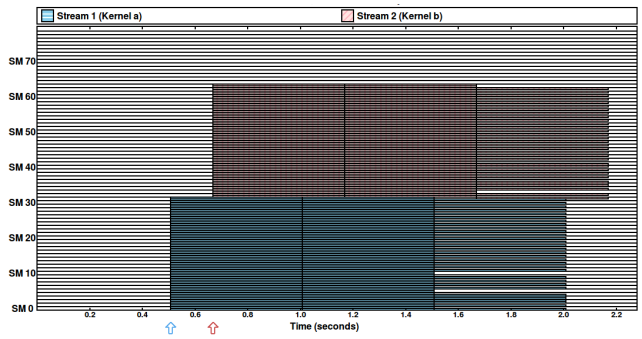



**Figure 4: Two processes, each using 40% of the GPU (all of the Titan V's 80 SMs shown).**

*blocks of 1024 threads each (exactly enough to utilize the entire GPU). Both kernels were able to run immediately using their 40% share, and both took three times as long to complete compared to their runtime in isolation. (Without MPS, one kernel would utilize the entire GPU, and the second would start upon completion of blocks of the first.)*

As demonstrated in Ex. 3, there is a tradeoff between the response time of a task and the fraction of the GPU it can utilize. One vGPU could service all tasks in $\mathcal{L}$, while the rest service tasks in $\mathcal{H}$. Alternatively, multiple processes could be configured with different GPU fractions, acting as servers of different execution power available to tasks in $\mathcal{H}$.

**Stream priorities.** One of the scheduling extensions explored in [1] is stream priorities. On existing NVIDIA GPUs, there are two priority levels for user-defined streams: priority-low (the default) and priority-high. As described by Amert *et al.*, the EE uses one scheduling queue per priority level. Rule A2 from [1] states that if a kernel is present at the head of the priority-high EE queue, then blocks from that kernel may execute on the EE. Otherwise, blocks of the kernel at the head of the priority-low EE queue, if any, may execute.

Example 4. *Fig. 5 depicts an experiment showing the starvation of a kernel submitted to a priority-low stream (Fig. 6 in [1], reproduced on a Titan V with only the first 8 SMs depicted for clarity). All 640 blocks of each of kernels K2 and K3 complete execution before the last 160 blocks of K1 execute. The execution of K3 is only delayed by the execution of K2, another kernel in a priority-high stream.*

Tasks in $\mathcal{H}$ could submit kernels only to priority-high streams, and tasks in $\mathcal{L}$ could submit kernels only to priority-low streams. Therefore, tasks in $\mathcal{H}$ would be delayed only by other tasks in $\mathcal{H}$ plus at most the longest block duration of any kernel submitted by a task in $\mathcal{L}$.

## 4.3 Combining Approaches

The approaches mentioned above can be combined. For example, in a system with multiple GPUs, allowing only tasks from a single cycle to execute on a GPU reduces blocking of those tasks, but might greatly underutilize the GPU. Instead, the prioritized stream approach could be used to allow tasks in $\mathcal{L}$ with short block durations to better utilize the GPU with only minimal delay for the tasks in $\mathcal{H}$ assigned to that GPU. Similarly, MPS and stream priorities could be combined to enable tasks in both $\mathcal{H}$ and $\mathcal{L}$ to execute on the same vGPU. The delay to tasks in $\mathcal{H}$ introduced by lower-priority jobs when using prioritized streams

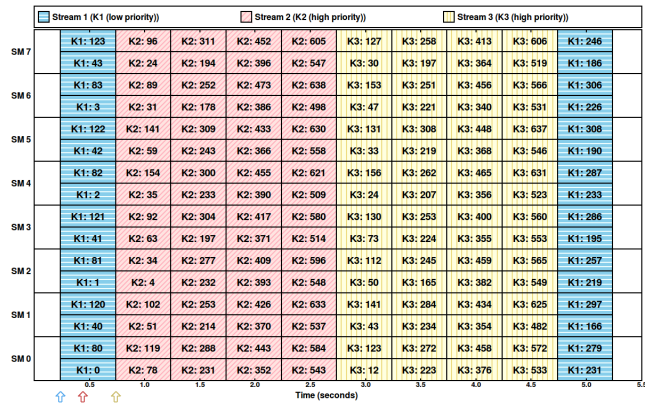



**Figure 5: Kernels in priority-high streams can starve kernels in priority-low streams (truncated to show only the first 8 SMs).**

suggests a design tradeoff between block duration and the number of blocks needed to complete computations.

# 5. CONCLUSION

In this paper, we presented multiple potential approaches for real-time GPU management, with a focus on prioritizing the operations of some GPU-using tasks over others. These techniques can enable tighter response-time bounds for GPU workloads. A deeper exploration of each approach, as well as combinations, is necessary in the future, with a case study of real automotive applications.

# 6. REFERENCES


[1] T. Amert, N. Otterness, M. Yang, J. Anderson, and F. D. Smith, "GPU scheduling on the NVIDIA TX2: Hidden details revealed," in *RTSS '17*.

[2] T. Amert, S. Voronov, and J. Anderson, "OpenVX and real-time certification: the troublesome history," in *RTSS '19, to appear*.

[3] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-based scheduling for GPU with preemption support," in *RTSS '18*.

[4] G. A. Elliott, "Real-time scheduling of GPUs, with applications in advanced automotive systems," Ph.D. dissertation, University of North Carolina at Chapel Hill, 2015.

[5] GPUSync Project, https://www.github.com/GElliott/litmus-rt-gpusync/.

[6] LITMUS$^{RT}$ Project, https://www.litmus-rt.org/.

[7] C. Nemitz, K. Yang, M. Yang, P. Ekberg, and J. Anderson, "Multiprocessor real-time locking protocols for replicated reources," in *ECRTS '16*.

[8] NVIDIA, "CUDA toolkit documentation v10.1.243," Online at http://docs.nvidia.com/cuda/, 2019.

[9] N. Otterness, M. Yang, S. Rust, E. Park, J. Anderson, F. Smith, A. Berg, and S. Wang, "An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads," in *RTAS '17*.

[10] M. Yang, T. Amert, K. Yang, N. Otterness, J. H. Anderson, F. D. Smith, and S. Wang, "Making OpenVX really 'real time'," in *RTSS '18*.