Deadline-based Scheduling for GPU with Preemption Support

Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna
Department of Physics, Mathematics and Informatics
University of Modena and Reggio Emilia Modena, Italy
{name.surname}@unimore.it

Aingara Paramakuru
NVIDIA Corporation
Toronto, Canada

***Abstract*—Modern automotive-grade embedded computing platforms feature high-performance Graphics Processing Units (GPUs) to support the massively parallel processing power needed for next-generation autonomous driving applications (e.g., Deep Neural Network (DNN) inference, sensor fusion, path planning, etc). As these workload-intensive activities are pushed to higher criticality levels, there is a stronger need for more predictable scheduling algorithms that are able to guarantee predictability without overly sacrificing GPU utilization. Unfortunately, the real-rime literature on GPU scheduling mostly considered limited (or null) preemption capabilities, while previous efforts in broader domains were often based on programming models and APIs that were not designed to support the real-rime requirements of recurring workloads. In this paper, we present the design of a prototype real-time scheduler for GPU activities on an embedded System on a Chip (SoC) featuring a cutting-edge GPU architecture by NVIDIA adopted in the autonomous driving domain. The scheduler runs as a software partition on top of the NVIDIA hypervisor, and it leverages latest generation architectural features, such as pixel-level preemption and thread-level preemption. Such a design allowed us to implement and test a preemptive Earliest Deadline First (EDF) scheduler for GPU tasks providing bandwidth isolations by means of a Constant Bandwidth Server (CBS). Our work involved investigating alternative programming models for compute APIs, allowing us to characterize CPU-to-GPU command submission with more detailed scheduling information. A detailed experimental characterization is presented to show the significant schedulability improvement of recurring real-time GPU tasks.**

***Index Terms*—GPU, Scheduling, real-time, ADAS**

I. Introduction

There is an increasing need in Advanced Driver Assistance Systems (ADAS) and Autonomous Vehicles (AV) technologies to support hybrid settings where highly-critical applications execute on the same embedded platform with less critical applications. High-performance embedded platforms are being proposed featuring multiple computing units, where shared resources across domains characterized by different criticality levels are usually managed by a hypervisor. To provide the required performance for complex ADAS/AV tasks, these systems often feature an integrated GPU as a massively parallel programmable processor that has to be shared across a potentially large variety of applications, each having different timing requirements. Being the component that provides the highest computing performance in the System of Chip (SoC), the GPU is becoming the most critical component to schedule in such heterogeneous systems. This led to the challenge of designing a real-time scheduler for GPU applications able to tackle both graphic and computing workloads that are typical of a vehicle capable of driving itself or to act as an intelligent assistant for the human driver. In such applications, GPU workloads may include DNN inference computing kernels, but also other generic embarrassingly parallel algorithms related to image processing, SFM (Structure From Motion), generic path planning and sensor fusion. Examples for graphic applications are speedometer and virtual cockpit rendering. The computing workloads potentially executed on high-performance embedded SoCs may therefore belong to different criticality domains. Some of them may be implicitly or explicitly constrained by deadlines, as it is typical for real-time tasks, whereas other applications may rather have weaker QoS requirements, henceforth called Best-Effort tasks.

In this paper, we propose and discuss the first (to the best of our knowledge) prototype implementation of a *deadline-based scheduler with preemption support* for GPU tasks in a virtualized environment. Previous efforts on GPU scheduling, as detailed in the next section, assumed little to no preemption capabilities for GPU tasks, or they did not take into account the possibility to leverage different programming models for GPU APIs in order to fully exploit event-driven scheduling algorithms. Deadline-based scheduling algorithms such as EDF (Earliest Deadline First) are known to be an optimal choice when scheduling systems composed of a single computing resource able to execute only one task at a time [1]. Since the single-instruction multiple-data (SIMD) execution paradigm of an integrated GPU fits the single-resource model for tasks working on parallel data in lockstep mode, EDF scheduling may be particularly useful to achieve a higher schedulable utilization for GPU tasks. Note that not all in-depth technical details of the setting can be revealed, nor the code be made freely available, due to NDA restrictions. Still, we did an extensive effort to provide information on previously undisclosed technical details, proposed implementation and API extensions, experimentally characterizing our solution over representative workloads and synthetic benchmarks. To investigate preemptive EDF policies for scheduling GPU tasks, we utilize a recently released NVIDIA Tegra-based SoCs able to expose shader/kernel preemption functionalities. Tests and experiments have been performed on an NVIDIA Drive-PX

"AutoCruise" platform featuring the Parker SoC. A notable feature of this SoC is the Pascal-based integrated GPU (gp10b) that allowed us to overcome some of the limitations assumed in previous papers dealing with real-time GPU scheduling. Since the introduction of the Pascal architecture, NVIDIA GPUs are graphic processing units able to support preemption at pixel-level for graphic applications, and at thread-level for CUDA compute workloads [2]. By leveraging this novel feature, and by having access to the NVIDIA software stack for embedded automotive systems, we were able to implement a prototyped version of an EDF-scheduler, which we then enhanced with a Constant Bandwidth Server (CBS) for providing task isolation in case of misbehaving applications. The implementation of such a scheduler implied: - Transitioning from a pre-existent table-based approach to an event-driven approach for GPU commands submission;
- Prototyping an enhanced programming model for both CUDA and OpenGL with real-time extensions;
- Prototyping an alternative SW (software) scheduler implementation that acts as a privileged guest in the hypervisor, improving over the currently implemented NVIDIA scheduler;
- Integrating our prototype with the pre-existing model for handling dependencies between the NVIDIA computing platform sensors/actuators and the respective GPU tasks.

II. Related Work

The GPU scheduling problem has been tackled by different research works. For graphic applications, Kato et al. proposed TimeGraph [3], a non-preemptive fixed-priority scheduler for only graphic GPU tasks, based on a modification of the Nouveau Open Source driver for NVIDIA GPU. The adopted event-driven approach is shown to outperform Best-Effort policies for scheduling real-time GPU tasks. More recently, Schnitzer et al. [4] proposed a Reservation-based scheduling mechanism for tasks, that also relies on open source GPU drivers. The goal was to schedule 3D-graphics tasks of an automotive application to meet frame-rate constraints, e.g., speedometer rendering rate as mandated by legal specifications. Lower priority tasks are scheduled only if there is sufficient slack to schedule higher priority GPU jobs before their deadlines, represented by a target framerate.

Our contribution modifies the proprietary NVIDIA driver to implement an event-driven scheduler, but it improves over both mentioned works in multiple aspect: (i) supporting not only graphic tasks but also GPU compute tasks; (ii) exploiting a much finer preemption granularity for GPU tasks, i.e., at pixel- and thread[1]-level; (iii) implementing a dynamic priority scheduler with resource reservation; and (iv) providing API extensions to consider tasks dependencies, deadlines, budgets and periods.

With relation to CUDA general-purpose kernels, Elliot et al. proposed in [5] system-wide lock mechanisms for GPU engines (Compute and Copy). In this solution, GPU engines are seen as mutually-exclusive resources that can be accessed only by given real-time locking protocols. Based on this assumption, the authors developed GPUSync, a software framework for GPU management in multi-core real-time systems.

Tasks are assumed to be composed of a CPU- and a GPU-part, where CPU activities are scheduled with EDF, while GPU kernels are scheduled with a non-preemptable FIFO algorithm. This latter choice is imposed by the fact that (differently from our presented work) no GPU preemption capabilities were considered in [5].

Other attempts to enhance the standard GPU hardware and software scheduler are related to the implementation of Persistent Threads and related applications [6], [7]. User-defined scheduling policies are obtained by batching many kernel calls into a single invocation to then arbitrate the execution of blocks of GPU threads within one or more persistently executing GPU threads [8]. All referred approaches consider the GPU as a *non-preemptable* resource, limiting the scheduling strategies that can be applied in such scenarios. Even if preemption of a CUDA kernel can be achieved by splitting a single kernel invocation into many different ones to have a better control over the scheduled blocks of threads [9], [10], this solution adds a significant overhead at the CPU side, while enabling only coarse grained preemptions, i.e., CTA (Cooperative Thread Array) level preemption within prioritized CUDA streams [11]. Moreover, these solutions imply significant changes both at CPU-host and GPU-kernel code level, preventing their adoption to closed-source tasks. Although recent contributions managed to mitigate this limitation by introducing source-to-source transformation at compiler level [12], [13], these solutions are still limited to a block- or task-level preemption granularity. For this reason, previous efforts on GPU scheduling cannot be used to implement resource reservation policies based on aperiodic servers, such as Constant/Total Bandwidth Servers [14], [15] and Deferrable/Polling Servers [16], as these mechanisms require preemption at a fine and known granularity.

Another common shortcoming of previously cited contributions is the lack of proper mechanisms to consider and arbitrate dependencies among different GPU applications. GPU-level dependencies are crucial to take into account, as they represent one of the most challenging aspects from a GPU scheduling point of view. Accounting for such dependencies implied a significant effort when developing a preemptive GPU scheduler. In order to fully exploit the proposed scheduling mechanisms, we also enhanced the programming model extending the API for real-time GPU applications. To our knowledge, previous contribution did not provide the possibility to specify deadlines, budgets and periods for GPU tasks at programming model level.

III. Prototype implementation

In this section, we detail our prototype implementation of a GPU scheduler at the software level. In order to do so, we first discuss some basic information related to the computing platform adopted in our implementation, disclosing the actual

[1]Thread-level preemption implies being able to preempt a CUDA kernel at compute instruction granularity

approach adopted by NVIDIA to GPU scheduling in current automotive-grade boards. It is important to highlight that sections III-A, III-B and III-C refer to the actual architectural solutions that are currently adopted within NVIDIA automotive boards. We then detail our prototype implementation for an EDF based scheduler with preemption support from section III-D.

A. Drive PX description

NVIDIA Drive PX is a high-performance embedded computing platform commonly used in Advanced Driving Assistance Systems (ADAS) and autonomous driving applications. The board features two versions: "AutoChauffeur" and "AutoCruise". The first one is a small scale supercomputer featuring two Tegra Parker SoCs, allowing external connection to up to two discrete GPUs connected through PCI-express. The second version, the one adopted in our experiments in Section VI, is a much simpler platform featuring a single Tegra Parker SoC, with a significantly smaller power consumption (10-15 W).

The Tegra Parker SoC is the latest embedded processing unit by NVIDIA, featuring a two-island CPU-complex and a high performance integrated GPU sharing the same LPDDR4 system memory. The CPU-complex is composed of a four-core A57 (ARMv8-A 64bit) island and by a dual-core NVIDIA Denver island. Denver is the NVIDIA proprietary design of a 64bit ARMv8-A compliant CPU architecture [2]. The GPU is an integrated scaled-down version of the newly released Pascal Architecture, commonly featured in both consumer-level and HPC-level graphics cards. The integrated GPU on Parker (gp10b) is characterized by two Streaming Multiprocessors (SMs), each featuring 128 CUDA cores.

B. GPU Scheduling and synchronization

With GPU scheduling, we refer to the arbitration mechanisms that regulate access to the GPU by the different applications. We do not consider the CTA (Cooperative Thread Array) hardware scheduling support within the same application, as it is out of the scope of this work. Recently disclosed information on NVIDIA GPU scheduler shows the presence of a hardware scheduler embedded in the GPU within a component called "Host[3]". The Host component is responsible for dispatching work to the respective GPU engines, such as the Copy, Compute and Graphics engines, in a Round-Robin way, and it is able to act in an asynchronous and parallel manner with respect to the CPU complex.

The Host scheduler fetches work related to channels, where a channel is an independent stream of work to be executed on the GPU on behalf of user-space applications. It is worth noticing that channels are transparent to a user-space programmer, which specifies GPU workloads through appropriate API (CUDA, OpenGL, etc.) function calls. Such a workload consists of a sequence of GPU commands that are inserted in a Command Push Buffer, which is a memory region written by the CPU and read by the GPU. Channels are therefore related to an application's Command Push buffer. Synchronization within a group of commands in the same channel or between different channels is implemented by means of semaphores and syncpoints, which are synchronization primitives able to be acquired and released by CPU, GPU engines and host1x[4]. Synchronization operations such as acquiring and releasing a semaphore/syncpoint are commands enqueued within a Command Push Buffer.

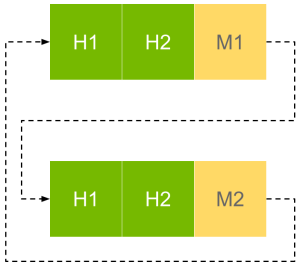


Fig. 1: A reconstructed runlist composed of 2 High priority task and 2 Medium priority tasks.

A GPU application maps itself to one or more channels. Each channel is characterized by a different timeslice value to timeshare the GPU execution among the different channels. Whenever all the work within a channel is consumed, or a preemption is needed for timeslice expiration, the currently running channel undergoes a context switch. Hence, the Host will start dispatching workloads related to the next channel from a list called *runlist*, and so on. The way in which applications map to one or more channels is application/API dependent and will not be discussed here.

The runlist is a list of established channels that may or may not have pending work to execute. It is important not to confuse the concept of runlist with the concept of Command Push Buffer. A runlist is not a list of pending GPU commands to be dispatched to the appropriate engines. It is simply a list of channels, each one pointing to a Command Push Buffer that contains the list of pending commands for that channel.

The GPU Host implements a list-based scheduling policy that snoops each channel for work by browsing the runlist. Each channel has a number of entries in the runlist that is proportional to its *interleaving level*. The scheduler browses the runlist, checking for each entry if the corresponding Command Push Buffer has workload to execute. If it does, the channel is scheduled until it either completes execution, or its *timeslice* expires. In the latter case, the channel is preempted, and it will be resumed in the next entry associated to that channel. If instead the application has no workload to execute, the scheduler skips its entries, proceeding to the channels related to the next application. An open source version of the runlist construction algorithm can be found in the NVIDIA

[2]www.tiriasresearch.com/downloads/nvidia-charts-its-own-path-to-armv8

[3]From now on, we refer to Host as the GPU component that dispatches work to the respective engines. Not to be confused with the term host in generic heterogeneous programming contexts, such as OpenCL or CUDA.

[4]The Tegra host1x module is the DMA engine for register access to Tegra's 2D graphics and multimedia-related modules.

kernel driver stack[5]. In general, all channels of a given priority level have an occurrence in the runlist before there is an entry for one lower priority slot. The next entry at that priority level will be after all channels of the higher priority level had another slot, and so on. Fig. 1 shows a sample runlist built with the mentioned algorithm for the case with two high priority applications and two medium priority ones, each consisting of one channel.

Timeslice length, interleaving level and allowed preemption policy are the scheduling parameters that can be tuned by a user. The timeslice is the execution time assigned to a channel before being preempted. The interleaving level refers to the number of occurrences of a particular channel within a runlist. The rationale for allowing a channel to be replicated more than once in a runlist is to have higher priority channels be checked for work more often than lower priority ones, allowing critical applications to be more resilient towards CPU-side delays when submitting commands. Replicating a channel within a runlist does not replicate its pending commands; it only increases the frequency in which the GPU Host will poll for work submissions related to that channel. Checking higher priority applications more often than lower priority applications is a design choice motivated by the asynchronous relation between GPU Host scheduler and CPU-side command submissions. Lacking direct CPU-to-GPU interrupt support to signal new command submissions, the GPU scheduler may poll more often higher priority applications to reduce their latency. Finally, the preemption policy allows labeling a channel to be non-preemptable, so that even if its timeslice expires, it may keep executing until it has no more pending work. Other preemption policies are CTA or thread-level preemption (for CUDA) and pixel-level preemption (for graphics workloads), allowing a channel to be preempted at the finest possible granularity when its timeslice is over. In older GPU architectures, such as Kepler and Maxwell, data movements operated by the copy engine were non-preemptable; however, preemption points might be easily inserted by splitting long copies into multiple smaller chunks [11]. In the considered GPU setting, i.e. Pascal GPU architecture, the hardware internally breaks up the copies into smaller chunks, so it is preemptable on this boundary.

Channels are established at context creation (i.e., at application launch). The Host keeps polling the command buffer related to the currently resident channel. Submitting new work or even adding/removing a channel does not have an immediate effect on Host scheduling. It is also worth mentioning that (i) the Host scheduler allows only one application to be resident within the GPU engines at a given time, and (ii) preemption is only initiated by a timeslice expiration event. If the executing channel is marked as preemptive, a timeslice expiration event triggers its preemption at pixel- or thread-level boundary, depending if it is a graphic or compute workload. Essentially, this scheduler performs a work-conserving TDMA (Time Division Multiple Access) between channels, and each channel can be assigned multiple slots within the runlist, which is the sequence of slots in the TDMA round.

We are interested in analyzing the response time of a GPU task, which is defined as a recurring set of commands sent to the Command Push Buffer associated to a channel. According to the standard notation for characterizing recurring real-time activities, a GPU task τ_i is characterized as

$$\tau_i \doteq (C_i, D_i, P_i), \tag{1}$$

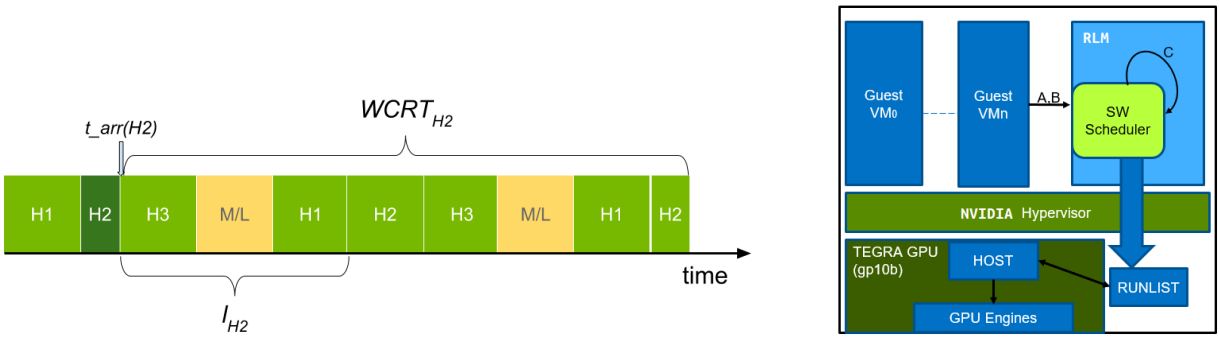
where C_i is the requested GPU execution time, D_i is the relative deadline, and P_i is the period or minimum inter-arrival time between two job submissions. This model fits perfectly an advanced automotive application where critical tasks (both graphic and compute) such as pedestrian detection and speedometer rendering follow a recurring pattern. The computing platform periodically acquires frames from one or more cameras at periodic rates, to feed them to Deep Neural Networks (DNNs) for object detection. Speedometer rendering must have a minimum target framerate that coincides with the periodic VBLANK signal, also known as vertical blanking interval, i.e., the signal triggered by the display refresh rate. The execution time C_i may match the inference time for a DNN, or any other combination of CUDA kernel invocations and copy operations, or the actual rendering time of the draw calls needed for displaying a graphic application.

NVIDIA's GPU scheduler is efficient for soft real-time requests and Best-Effort activities, but it shows some drawbacks in case of tighter real-time requirements. The scheduler allows only three priority levels (for interleaving), making this mechanism not sufficiently flexible for complex task sets. In addition, it is unclear how to estimate the optimal number of duplicated entries of a real-time task within the runlist, or how to properly select task timeslices. As will be shown in the experimental section, the list-based scheduling policy may lead to a very high latency between a job submission from the CPU and its actual execution on the GPU. Such a latency can be upper bounded using Theorem 1 in Appendix. Our analysis of the NVIDIA baseline scheduler proves that preemption alone (even at a fine granularity) is not sufficient for providing real-time guarantees to GPU task-sets. The focus of our work is to bypass the HW hardcoded arbitration policies, implementing scheduling algorithms at software level to improve real-time guarantees.

C. NVIDIA hypervisor

The Drive PX platform development kit includes hypervisor and GPU virtualization technology, allowing multiple guests to concurrently run and access the GPU engines. Each guest might be mapped to different virtual or physical CPU cores, and it can run different operating systems. The hypervisor is able to guarantee memory spatial isolation and it manages both inter-VM (Virtual Machine) communication and resource sharing. The NVIDIA hypervisor follows the bare-metal paradigm, statically assigning memory ranges and HW

[5]Available in the L4T (Linux For Tegra) kernel sources at https://developer.nvidia.com/embedded/linux-tegra and described in the official documentation available at https://docs.nvidia.com/drive/nvvib_docs/index.html



(a) (b)

Fig. 2: (a) Worst Case Response Time (WCRT) for a GPU task $H2$ as a function of its arrival time t_arr. Note that the darker green $H2$ is depicted as a short interval because the GPU Host did not find work to dispatch to the engines. Task initials H, M or L indicate their interleaving level. I_{H2} is the latency between $t\_arr(H2)$ and the beginning of H2 execution.
(b) The main blocks characterizing our prototype scheduler implementation. A, B and C are event signals and messages used for scheduling decisions.

(hardware) devices to the different guests in an exclusive manner. However, certain devices might be shared among different VMs. This is the case of the GPU, which can be concurrently accessed by different guests. This is accomplished through a privileged SW scheduler guest called RunList Manager, or RLM. The other guests wishing to access the GPU have to contact the RLM server through the inter-VM communication infrastructure of the hypervisor for operations such as channel allocations, scheduling parameters setting, and other memory management operations. In other words, regular guests have a para-virtualized GPU driver in which security-sensitive and resource-sharing operations are actually managed by the RLM. The only direct-access operation to the GPU allowed to the clients is pushing commands to the command buffer so to be fetched by the GPU Host. This happens in a completely transparent manner with respect to CUDA or OpenGL API calls. Hence, no user-space application code refactoring is needed.

D. Scheduler implementation

To implement our prototype scheduler, we enhanced the RLM with a software scheduling module. A block diagram of the SW stack featuring our scheduler is depicted in Fig. 2b.

The scheduler represents the interface towards the GPU HW, acting as a replacement of the current runlist-based approach. Our scheduling mechanism works by submitting to the GPU a runlist including only the channels mapped to the application selected by the scheduler, i.e., one application at a time. Since the GPU Host only polls channels included in the runlist, and the runlist is composed of a single resident application, the Host component is prevented from performing a context switch at timeslice boundaries. This allows implementing an internal RLM module able to take scheduling decisions of arbitrary complexity at SW side, without requiring modifications to the scheduling policy hardwired in the Host module. In order for our approach to work as expected, every application is marked as preemptable at pixel/thread granularity. Whenever our scheduler decides a new task is to be scheduled (i.e., enforcing a runlist update), a preemption signal is triggered on the currently running application.

On a design perspective, we had to modify the inner mechanisms of the currently implemented NVIDIA approach. As detailed in the previous section, the runlist-based arbitration has a list of pending work which is constantly and asynchronously polled with relation to the rest of the system. This mechanism is efficient to maximize throughput, but it may be not so appropriate to provide real-time guarantees to critical jobs, as scheduling decisions are not based on timing requirements, e.g., deadlines, periods and allowed budget. For this reason, we decided to implement an event-based approach relying on signals triggered by events such as new work submission, work batch completion and budget expiration (A,B and C in Fig. 2b).

Whenever a batch of commands is written by a client guest to the Command Push Buffer, a signal denoting new work submission is triggered to the RLM module. The existing virtualization support in the NVIDIA hypervisor did not trigger such a communication procedure, mainly for performance reasons: both graphic and compute applications might write into the Push Buffer at a very high frequency, and even a slightest delay might result in visible performance deterioration. In our prototype, we traded performance for real-time compliance, triggering a signal every time a new batch of commands is pushed.

A similar mechanism is needed to signal when work is completed, notifying the SW scheduler whenever GPU engines are idle. In order to do this, we decided to take advantage of synchronization procedures (semaphores and syncpoints) that the client driver inserts within the Command Push Buffer, using them to understand whether the previous commands have been consumed by the engines. Such synchronization data structures are passed each time a submission of new work is notified to the SW scheduler.

The last signaling event is triggered internally by the RLM. Every time a new runlist with the relevant set of channels

is pushed, a software timer keeps track of the time spent by the running application in the GPU engines. This is instrumental for developing scheduling algorithms based both on time-sharing and bandwidth reservation at application level. This latter signal is closely related to the implementation of the resource reservation scheduler detailed in the following section.

IV. EDF+CBS ALGORITHM

Our assumption to consider the GPU as a single computing resource may sound oversimplifying due to the massively parallel nature of a GPU. Newly released consumer/HPC level graphic cards featuring the same architecture as the one in the Parker SoC can scale up to a very large number of SMs, i.e., the computing cluster containing parallel executing CUDA cores. For example, an NVIDIA Tesla P100 scales up to 60 SMs. In these settings, mapping groups of SMs to different tasks might be instrumental for developing an efficient scheduling algorithm that still retains real-time properties.

In contrast, gp10b features only 2 SMs, as a completely different power consumption and die size is needed for embedded automotive scenarios. The GPU is sufficiently small to be considered as a single computing resource, where to schedule one GPU task at a time, hence taking advantage of thread-level parallelism within the task, but not among different tasks. The benefits of mapping multiple applications or tasks to different SM's would be neglected by GPU self-interference [17]. Considering an integrated GPU as a single resource suggested using EDF as a scheduling algorithm to maximize GPU resource utilization.

The absolute deadline d of a GPU task is computed as $d = D_i + t_a$, where t_a is the job arrival time. A scheduler based on absolute deadlines allows us to be independent from the clock skews of the different sensors and actuators utilized in the analyzed system. However, misbehaving tasks may still cause enqueued critical jobs to be scheduled too late. For this reason, we implemented a Constant Bandwidth Server (CBS) to enforce resource reservation [18] at task level. The budget B_i of the CBS server is assumed equal to the WCET of the corresponding GPU task C_i. Whenever a task overruns its budget, its deadline is proportionally postponed, potentially causing a preemption. CBS was selected for its design simplicity and limited implementation overhead, seamlessly integrating with the deadline-based scheduling support we prototyped at GPU level. Schedulability analysis with respect to EDF with CBS, also in the presence of shared resources can be found in [19] and [20].

A. Deadline-based GPU scheduling

Having defined the scheduling events that may modify the list of tasks, the EDF+CBS scheduler is implemented as a SW module on the RLM. However, there are a number of additional challenges that we had to consider. In particular we need to define the scheduling granularity, to detect and deal with inter-process dependencies and to handle Best-Effort applications.

Scheduling granularity. When designing a GPU scheduler, we had to decide at which granularity to take scheduling decisions. Doing it at command level would imply a heavy overhead due to the large number of commands that might compose a single API call. Setting deadlines only at application level would not provide the necessary flexibility, as an application may be composed of multiple jobs with different timing requirements. Therefore, we set the scheduling granularity at the level of command batches. A batch of commands is a group of commands that relates to a variable number of unsynchronized API calls. Such API calls use the same set of inputs and outputs related to a high-level definition of task. An example of how we define a batch of commands in a graphic application is represented by the set of commands for rendering the same frame. In a compute scenario involving DNN inference, we flag as a batch the set of commands related to the kernel invocations for each layer of the considered neural network, along with the data movements from CPU-GPU address space and vice versa. Graphic applications are batched by definition, as the swap buffers API call is used as frame delimiter. This cannot be applied for CUDA applications, hence our effort to propose an alternative programming model, as detailed in Section V.

Inter-process dependencies. Interprocess dependencies at GPU level have been only superficially considered in previous literature (see section II). However, this turned out to be one of the most challenging aspects when designing a real-time scheduler for the GPU. Dependencies between channels in the same application are trivially resolved by placing all the channels mapped to that application in the next runlist update. By doing so, the required channels for the considered application are available to be scheduled by the GPU to acquire and release the synchronization primitives for satisfying dependencies and enforcing the desired execution order. The hardware support at the GPU side is already optimized to sort out this kind of intra-application dependencies.

The same is not true when synchronization primitives are shared among command buffers kicked by different applications. An application - level example is given by the display server (Xorg or Weston) which is shared by multiple graphic client applications. Any kind of dependency graph can be established between an arbitrary number of GPU applications. This can be done by means of Khronos EGLStreams [21], which are sharable objects that allow sharing data across multiple contexts related to different APIs. This is how, for instance, a CUDA application might share a buffer with an OpenGL renderer, or how a video feed or a camera might share frames to be consumed by a CUDA application. EGLStreams act at user space level, flagging processes as data producer and consumer, but allowing these roles to switch over time[6]. At driver level, sharing of EGLStream objects translates into

[6]More information on EGL and CUDA interoperability can be found at: http://on-demand.gputechconf.com/gtc-eu/2017/presentation/53023-debalina-bhattacharjee-eglstreams-interoperability-for-camera-...cuda-and-opengl.pdf

pushing acquire and release syncpoint operations in the Command Push Buffer.

In our prototype, we implemented a deadline inheritance mechanism described as follows. Consider a task set τ, in which a task $\tau_i \in \tau$ might be a consumer or a producer. If τ_i is a consumer of τ_k, that implies τ_k being a producer of τ_i, we will denote it as $\tau_k < \tau_i$. Each task is mapped to one or more GPU channels. At every scheduling event, we need to decide how to fill the next runlist RL to submit to the GPU Host.

Once application τ_i is pulled from the ordered list of deadline-based batches, a recursive procedure fills the next runlist to be submitted by including all the channels mapped to the chain of dependencies of τ_i. The priority level of the channels added in this way is boosted to the same priority of application τ_i. Namely, all tasks that have a precedence constraint with τ_i have their priority boosted to that of τ_i. Once this new runlist is pushed to the GPU, every time a dependency is satisfied, the corresponding channels are disabled/removed from the current runlist. EGLStream shared objects are internally managed by GPU synchronization primitives like semaphores and syncpoints. As previously highlighted, this information is passed with each notification of new work submission (details in section III-D).

Best-Effort applications. We cannot expect Best-Effort applications to behave in a ideal manner, let alone to have them to communicate period, budget and deadlines to therefore send commands in a timely fashion. On the contrary, Best-Effort applications may flood the Command Push Buffer, potentially affecting the predictability of the system. In our prototype, we implemented a fixed-priority scheduler to arbitrate Best-Effort applications, that operates only when no real-time task is ready to execute.

V. API REAL-TIME EXTENSION

For enforcing the scheduling decision detailed in the previous sections, we need to allow the application developer to expose API functionalities for specifying task boundaries and respective timing parameters. We do this by prototyping API extensions for both CUDA and OpenGL. Ideally, considering the available support at API-side, a different programming model would be more suitable. The closest programming model able to fit our needs is represented by newly released APIs, such as Vulkan and Direct3D 12. These novel approaches to GPU programming involve preparing in advance pipeline state objects and command buffers to be then later submitted within a single (or limited set of) write operation inside the Command Push Buffer. We refer to these single submissions as a batch of commands.

The minimal CPU-to-GPU submission mechanism for these novel APIs involves minimal driver interactions and validation procedures, therefore minimizing the impact of CPU-side delays during command submission. This is in contrast with the traditional APIs (e.g., CUDA and OpenGL) and respective programming models, in which commands are constantly streamed from the CPU to the GPU, with each API call being validated at driver level. Such paradigm not only constitutes an additional threat to predictability, but it also makes it impossible for CUDA applications to define a concept of batched command submission. If we were able to complement these novel programming models with the possibility of sending scheduling parameters (period, budget and relative deadline) attached to each submission, the RLM guest would be informed about the most suitable scheduling decisions to take based on such parameters, properly sorting the queue of deadline batches, as well as setting the CBS with the appropriate budget and period.

Rather than exploiting Vulkan, our implementation extends the traditional APIs to become closer to such a newer generation of programming models. This allows us to fully exploit the maturity that characterizes traditional APIs, in terms of pre-existing libraries (such as cuDNN for CUDA) and available support. Our extensions are basically additional user space runtime OpenGL and CUDA functions that internally trigger appropriate messages and signals to the RLM. Graphic applications are intrinsically batched. On an application-level perspective, this resulted in the creation of an OpenGL API call able to inform the RLM about the rendering WCET and the desired target frame-rate. We do this before the rendering loop, i.e., during the graphic context initialization, as detailed in Listing 1 in the Appendix.

When the graphic application starts submitting commands related to the different frames, the RLM associates them to the scheduling parameters that have been previously specified. Such scheduling parameters are sent as a message to the hypervisor layer through the API-level function call that we introduced. This function is called *glSetFrameTarget* and takes two parameters as input: an unsigned integer for indicating the desired framerate, and another unsigned integer identifying the budget in μs to assign for the draw calls needed for rendering the subsequent frames.

For CUDA compute applications, instead, commands are not batched, as there is no equivalent concept of frame boundary. In order to create batches of commands, we introduce two additional CUDA runtime API calls: *cudaStreamDeadlineBegin* and *cudaStreamDeadlineEnd*. These API calls allow us to bind different batches of commands within different CUDA streams, where a CUDA stream is a software abstraction of a queue of commands which are executed in the order they are inserted into the stream. Therefore, our API extension allows us to identify task boundaries where to define scheduling parameters (period, budget and relative deadline) that will be then associated by the RLM to all the commands included between the code block of *cudaStreamDeadlineBegin* and *End*. Scheduling parameters are inserted as input arguments for *cudaStreamDeadlineBegin*. More specifically, the input arguments for the added function calls are:

- *cudaStream_t s* : the CUDA stream in which we want to enqueue the commands to schedule.

- *uint32_t* D_r : the relative deadline of the batch of commands [μs].

- *uint32_t* B : the budget of the batch of commands [μs]

- *int32_t* P : the period of the batch of commands [μs].

Work completion notification from CPU side to the GPU is implemented through *cudaStreamDeadlineEnd*, which is a wrapper to the CUDA standard runtime function *cudaStreamAddCallback*. This function registers an asynchronous callback to notify the RLM when the previously enqueued operations of the CUDA stream are completed. A simple pseudo-code sample is provided in Listing 2 in the Appendix.

In a typical setting, the CPU has multiple threads submitting batches of commands to the GPU. An initialization function creates the CUDA context and the CUDA streams that will be used to submit batches of commands. CPU threads are dynamically activated based on sensor inputs and external events. Each thread may then submit batches of commands to one or more of the created streams, associating a budget, deadline and period to each batch. We highlight that the insertion of these novel API calls for real-time tasks has to be done by the application developer, requiring only a minimal effort. No modification is instead needed for best effort applications.

VI. Evaluation and Testing

In order to validate the implementation of our scheduler and for providing a sound comparison analysis against the existing NVIDIA interleaved scheduler, we set up two different benchmarking scenarios. The first test environment evaluates the feasibility of our approach in a realistic scenario. We ran a set of experiments in the Drive PX board using a collection of both graphic and compute workloads that are representative of a real world ADAS application. Dependencies with display servers and output displays are taken into account.

Rather than showing other similar test benchmarks that would only characterize a limited portion of the schedulability space, we present a second test setting that provides an exhaustive characterization of the relative performances against a set of randomly generated task sets to evaluate the theoretical schedulability limits of the NVIDIA baseline approach. For each generated task set, we simulate the runlist construction using the existing NVIDIA algorithm.

A. Realistic benchmarks

In this evaluation scenario, different applications at different levels of criticality compete for GPU time. Applications run on a NVIDIA customized Ubuntu distribution using Weston display server. Such operating system runs on top of the NVIDIA hypervisor, as described in section III. Modified drivers and API implementation were applied to the latest version of the Tegra proprietary driver stack, both for CUDA and OpenGL. The experimental task set is composed as follows:

(1) An OpenGL real-time application with 30 FPS as a strict requirement (32 ms as deadline). This application renders a sphere built with 2500 dynamically displaced vertices. The sphere's reflective surface is rendered with cube environment mapping. This program runs with a resolution of 960x540 and its WCET is 4 ms, with an average of 1.2 ms.

(2) A CUDA real-time application running from a Weston command shell, submitting a CUDA stream of work for computing the inference of a 10-layer convolutional DNN. Its calculated WCET is 3 ms (1.5 ms on average) and its period is 40 ms. This network is a reduced version of an image processing inference kernel, that we trained on the CIFAR10 dataset [22]. Its relative deadline is set to 4 ms after the task release.

(3) A custom-built OpenGL Best-Effort application, featuring multi-texturing and dynamic lighting. This application has an average rendering time of 3.5 ms when running with a resolution of 960x540, and it is continuously submitting jobs.

(4) A Wayland-based porting of the known glxgears benchmark, which is a Best-Effort OpenGL application running in a 960x540 window with a framerate capped at 60 FPS, and an average rendering time of 1.1 ms. All these applications involve moving data from a CPU-managed address space to the GPU-address space as part of the work submission. All the graphic applications have a dependency on Weston. Even if WCETs and average GPU time for the previously described tasks are relatively short compared to a 16.67 ms window (corresponding to 60 FPS), the Best-Effort application (3) is continuously submitting commands, bringing the overall GPU theoretical utilization above 100%.

To evaluate the behavior of the existing NVIDIA interleaved scheduler detailed in section III-B, we assigned the two real-time applications the highest possible interleaving level (i.e., the highest priority), while maintaining the lowest interleaving level for the other two applications. The timeslices assigned to the high priority tasks are equal to their WCETs, whereas the lower priority tasks have a timeslice of 1 ms. For the EDF+CBS algorithm, we set CBS server budgets to the WCET of the real-time applications. With the interleaved approach, 5.6% of batch submissions from both real-time tasks experienced deadline misses. Instead, when adopting our proposed EDF+CBS approach, no deadline miss has been observed. The worst-case response times of real-time tasks (1) and (2) improve by 64% and 93%, respectively, at the expense of an increase of the response time of best-effort applications. This is in line with our scheduling target of privileging critical recurring real-time activities while limiting the interference due to best-effort jobs, consolidating the feasibility of our prototype implementation. Charts showing more detailed results can be found in the Appendix.

B. Simulated benchmarks

In order to provide a more detailed characterization of the considered schedulers for general task sets, we performed an exhaustive set of simulations with randomly generated recurring GPU workloads. The UUniFast algorithm presented in [23] was adopted to build task sets composed of a given number of real-time tasks with a desired overall Utilization U. Task periods were randomly generated from a uniform distribution in the range $[16ms, 125ms]$, corresponding to a framerate varying between 8 to 60Hz. WCETs were computed accordingly from the generated utilization and period. Deadlines were assumed to be equal to task periods. We considered

NR real-time tasks with the highest interleaving level, and a single Best-Effort task with low interleaving level continuously submitting work to the GPU. We assess the schedulability only in relation to the NR high priority applications. Each task is mapped to one GPU channel. The maximum preemption granularity level is enabled for all tasks.

The schedulability for NVIDIA's GPU scheduler has been characterized by invoking the runlist construction routine available in the referred open source driver, and simulating the resulting schedule, including the preemption and communication overhead, in the worst-case scenario outlined in Theorem 1 in the Appendix.

For the NVIDIA scheduler, we also characterized the behavior when varying the maximum allowed timeslice TS for all tasks, given in μs. As explained in section III-B, the timeslice determines the maximum continuous execution time allowed to a GPU task instance before being preempted. A larger timeslice implies a smaller number of preemptions, but also a larger blocking time. The experiments shown in Fig. 3 detail the schedulability ratio of the considered algorithms, where each point corresponds to 1000 randomly generated task sets. The real contribution of preemption and communication overhead is included in the simulated settings, as will be detailed later on. Inset (a) shows the behavior of NVIDIA's native scheduler with $NR = 5$ tasks. Clearly, the number of schedulable task sets decreases when increasing the overall utilizations due to the additional interference from concurrently executing GPU tasks. Even with a small timeslice (1ms), the schedulable utilization significantly drops for $U > 0.5$. Increasing the timeslice causes a larger blocking penalty that further deteriorates the schedulable utilization.

Fig. 3(b) shows the situation when increasing NR to 20. With a higher number of tasks, a larger blocking delay is imposed to real-time tasks, due to the higher number of entries associated to interfering tasks in the runlist. Indeed, the delay between two timeslices of the same channel is proportional to NR, so that increasing this value leads to a higher response time. To better understand how the timeslice length affects the schedulability of NVIDIA's scheduler, we performed a set of experiments varying TS within a $[500, 8000]$ μs, which are typical values adopted in the existing systems. The results with $NR = 5$ are shown in Fig. 3(c) for various GPU utilizations. Increasing the timeslice has again a significant impact on the schedulability. The last set of experiments is devoted to show the performances of our EDF+CBS scheduler. Since we consider the GPU as a single resource, the theoretical schedulable utilization of the EDF scheduler is 100%. To better characterize the improvement of our solution with respect to the native scheduler, we present the experimental results including the cost of CPU-to-GPU command submission, kernel driver-RLM interactions and GPU context switches. To this extent, we adopted the schedulability test for EDF presented in [24], including preemption overhead and CPU-to-GPU communication delay.

A parameterized simulation is visible in Fig. 3(d) within a representative overhead range. Almost all generated task sets are schedulable with our EDF scheduler even at very high utilization and with a large overhead, significantly improving over NVIDIA's existing approach. For large utilization values, the overhead starts affecting the schedulability when it exceeds 500 μs for task sets with 0.95 utilization, or $1ms$ for slightly smaller utilizations. While, for NDA reasons, we cannot include the detailed measurements we performed on each overhead component on the real platform, it suffices here to say that the actual overhead is way smaller than these values for typical CUDA kernels. The following section provides intuitive evidences behind this statement.

C. Overhead characterization

The experiments we presented factored in representative values of the overhead. E.g., the results shown for NVIDIA baseline scheduler considered a realistic preemption penalty whenever a timeslice expiration event is triggered while the executing channel has still pending work to be consumed. A similar overhead is added whenever our EDF-based scheduler triggers a preemption.

To better understand the magnitude of the overhead, consider the operations to be executed whenever a preemption is triggered. For CUDA kernels, the data context of the preempted task has to be saved, which basically implies storing all the local data for each SM to global memory, and flushing the GPU Last Level Cache (L2). Therefore, the amount of data to save in the worst-case amounts to around one Megabyte: 64KB of total constant memory, 48KB of shared/L1 for each SM, 32K 32bit registers for each SM, and 512KB as L2 size. Considering a 20GB/s bandwidth from GPU to system memory, saving such a context takes around 50 μs.

Instead, for graphic applications, pixel level preemption implies that the time to compute a pixel (and therefore the preemption latency) varies depending on the executing shader. We can bound this value to 750 μs, which is half the channel reset time, as documented in the Drive PX official guide. A channel reset occurs when a GPU application gets stuck in the engines, for example due to a long or infinite loop in a graphic shader or compute kernel. A reset implies evicting the context from the runlist/task-set, which is why we did not consider reset channels in our simulations.

For our prototype scheduler, in addition to preemption costs, it is necessary to add the communication cost between the guest and the RLM at each scheduling event, as detailed in section III-D. This cost could be neglected if the scheduler were implemented at hardware level, as is the case of NVIDIA baseline scheduler. While NDA reasons prevent us from providing exact figures on these communication signals, we parametrized the total overhead of our EDF+CBS simulation into the realistic range of [100, 1500] μs, with the real value being in the lower half of this range for typical workloads. For realistic overhead values within this range, Fig. 3(d) shows that our prototype scheduler is able to guarantee the schedulability of most generated task sets even at large utilizations: more than 70% of the generated task sets with utilization 0.95 are schedulable even with an overhead of $1500\mu s$.

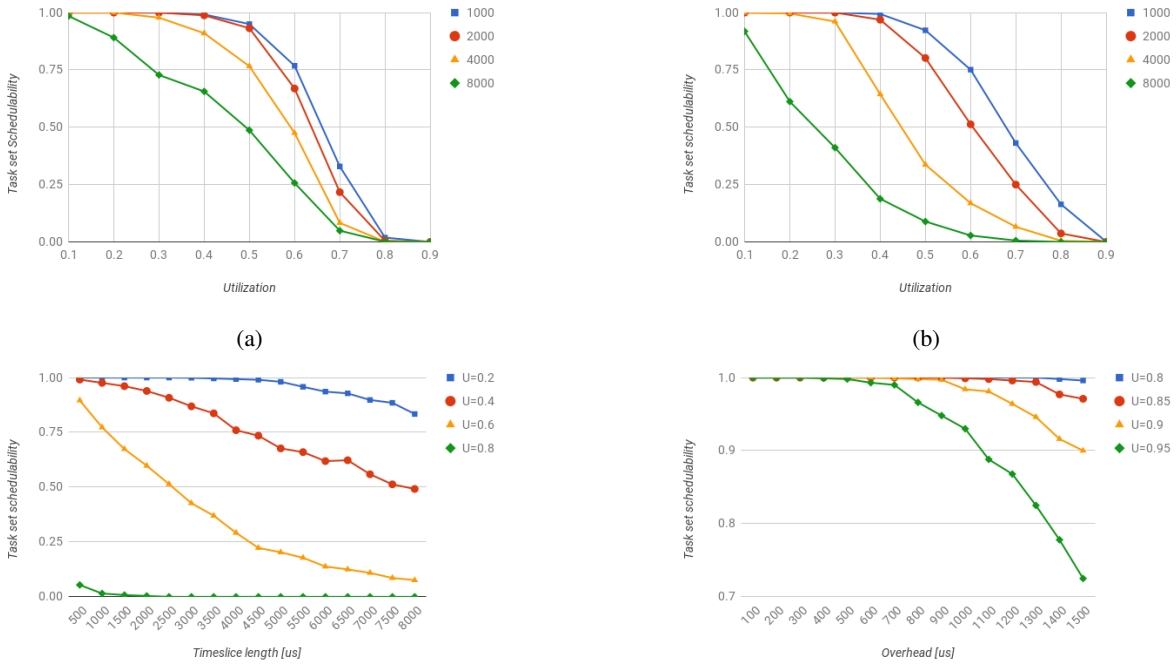


Fig. 3: Schedulability ratio of NVIDIA scheduler as a function of task set utilization with 5 tasks (a) and 20 tasks (b) for different timeslice lengths TS in us; and as a function of timeslice length TS with 5 tasks (c) for different utilizations; Inset (d) shows the schedulability ratio of our EDF scheduler as a function of preemption overhead with 5 tasks for different utilizations (note the different y-axis scale).

VII. Conclusion and Future work

This paper described a prototype implementation of a deadline-based scheduler with preemption support for integrated GPUs. The prototype runs as a software module on the RunList Manager partition of the NVIDIA hypervisor. It implements an EDF scheduling algorithm, enhanced with a CBS-based timing isolation mechanism for both compute and graphic workloads. On a software perspective, this work implied modifications at API and hypervisor level, in order to transition from NVIDIA's existing runlist-based approach to an event-driven one, hence providing more flexibility to system designers when defining the timing requirements of their task sets. The execution of high priority aperiodic activities can be easily accommodated in our scheduling support by assigning them a short, or null, relative deadline, allowing them to preempt any currently running job. This may be particularly useful in critical settings for promptly displaying critical messages to the user in a virtual interface, e.g., the instrument cluster in a driving cockpit, something that is difficult to achieve with the existing table-driven approach without wasting pre-reserved slots. We implemented our scheduler as a module within NVIDIA's proprietary hypervisor to allow the concurrent access to GPU resources by multiple guest VMs in a mixed-criticality environment. A similar solution could have been implemented at kernel driver level in a native non-virtualized solution using publicly available platforms. As a future work, we intend to enhance the implemented server with more advanced reclaiming mechanisms for GPU bandwidth left unused by real-time tasks [25], allowing a more flexible selection of GPU task budgets. In the current approach, the unused bandwidth is left to Best-Effort tasks. We also intend to address the more complex scheduling problem of GPUs composed of a larger numbers of streaming multiprocessors, or Multi-GPU settings within the same chip, as recently discussed in [26]). In these cases, considering the GPU as a single computing resource might not be an optimal choice, but more efficient algorithms may be designed following a multi-resource paradigm [27]. We are also working on mitigating the effects of memory contention in GPU-based embedded systems. As shown in [28], significant latencies may be experienced in heterogeneous embedded devices due to contention on shared memory, especially in case of memory-intensive GPU tasks. To overcome memory contention, several methodologies for arbitrating memory accesses between CPU and GPU are under investigation [11], [29]–[31]. Our final target is to tackle the coordinated scheduling problem of real-time tasks executing on heterogeneous embedded devices featuring multi-core host, GPU and a number of alternative accelerators[7] that are being integrated on next-generation devices for the autonomous driving domain.

[7]NVIDIA Deep Learning Accelerator http://nvdla.org/

ACKNOWLEDGMENT

The authors would like to thank NVIDIA Corporation for letting us work on the latest GPU architectures and for the useful information given to us. This work is also part of the Hercules and OPEN-NEXT projects, which are respectively funded by the EU Commission under the HORIZON 2020 framework program (GA-688860) and ERDF-OP CUP E32I16000040007.

REFERENCES

[1] 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, 1973.

[2] "Gpu scheduling and synchronization for adas," on-demand.gputechconf.com/gtc/2017/ presentation/s7105-venugopala-madumbu-adas-ad-challenges-gpu-scheduling -and-synchronization.pdf, NVIDIA, 2017.

[3] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," 2011.

[4] S. Schnitzer, S. Gansel, F. Dỳrr, and K. Rothermel, "Real-time scheduling for 3d gpu rendering," in *Industrial Embedded Systems (SIES), 2016 11th IEEE Symposium on*. IEEE, 2016, pp. 1–10.

[5] G. A. Elliott, B. C. Ward, and J. H. Anderson, "Gpusync: A framework for real-time gpu management," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*. IEEE, 2013, pp. 33–44.

[6] J. Breitbart, "Static gpu threads and an improved scan algorithm," in *European Conference on Parallel Processing*. Springer, 2010, pp. 373–380.

[7] N. Capodieci and P. Burgio, "Efficient implementation of genetic algorithms on gp-gpu with scheduled persistent cuda threads," in *Parallel Architectures, Algorithms and Programming (PAAP), 2015 Seventh International Symposium on*. IEEE, 2015, pp. 6–12.

[8] S. Xiao and W.-c. Feng, "Inter-block gpu communication via fast barrier synchronization," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–12.

[9] C. Basaran and K.-D. Kang, "Supporting preemptive task executions and memory copies in gpgpus," in *2012 24th Euromicro Conference on Real-Time Systems*. IEEE, 2012, pp. 287–296.

[10] J. Zhong and B. He, "Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1522–1532, 2014.

[11] N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna, "Sigamma: Server based integrated gpu arbitration mechanism for memory accesses," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, ser. RTNS '17. New York, NY, USA: ACM, 2017, pp. 48–57.

[12] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "Effisha: A software framework for enabling effficient preemptive scheduling of gpu," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2017, pp. 3–16.

[13] B. Wu, X. Liu, X. Zhou, and C. Jiang, "Flep: Enabling flexible and efficient preemption on gpus," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017.

[14] M. Spuri and G. C. Buttazzo, "Efficient aperiodic service under earliest deadline scheduling." in *RTSS*, 1994, pp. 2–11.

[15] M. Spuri and G. Buttazzo, "Scheduling aperiodic tasks in dynamic priority systems," *Real-Time Systems*, vol. 10, no. 2, pp. 179–210, 1996.

[16] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Transactions on Computers*, vol. 44, no. 1, pp. 73–91, 1995.

[17] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of gpu memory system for multi-application execution," in *Proceedings of the 2015 International Symposium on Memory Systems*. ACM, 2015.

[18] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Real-Time Systems Symposium, 1998. Proceedings. The 19th IEEE*. IEEE, 1998, pp. 4–13.

[19] L. Abeni, G. Lipari, and J. Lelli, "Constant bandwidth server revisited," *Acm Sigbed Review*, vol. 11, no. 4, pp. 19–24, 2015.

[20] G. Lipari, G. Lamastra, and L. Abeni, "Task synchronization in reservation-based real-time systems," *IEEE Transactions on Computers*, no. 12, pp. 1591–1601, 2004.

[21] "Khronos eglstream specification," https://www.khronos.org/registry/EGL/extensions/KHR/, Khronos, 2016.

[22] D. Ciregan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *Computer vision and pattern recognition (CVPR), 2012 IEEE conference on*. IEEE, 2012, pp. 3642–3649.

[23] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1, pp. 129–154, 2005.

[24] S. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *Proceedings of the 11th Real-Time Systems Symposium*. Orlando, Florida: IEEE, 1990, pp. 182–190.

[25] M. Caccamo, G. Buttazzo, and L. Sha, "Capacity sharing for overrun control," in *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*. IEEE, 2000, pp. 295–304.

[26] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "Mcm-gpu: Multi-chip-module gpus for continued performance scalability," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017.

[27] S. Baruah, M. Bertogna, and G. Buttazzo, *Multiprocessor Scheduling for Real-Time Systems*. Springer, 2015.

[28] R. Cavicchioli, N. Capodieci, and M. Bertogna, "Memory interference characterization between cpu cores and integrated gpus in mixed-criticality platforms," in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sept 2017, pp. 1–10.

[29] P. Houdek, M. Sojka, and Z. Hanzálek, "Towards predictable execution model on arm-based heterogeneous platforms," in *Industrial Electronics (ISIE), 2017 IEEE 26th International Symposium on*. IEEE, 2017, pp. 1297–1302.

[30] B. Forsberg, A. Marongiu, and L. Benini, "Gpuguard: Towards supporting a predictable execution model for heterogeneous soc," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017, pp. 318–321.

[31] A. Waqar and Y. Heechul, "Work-in-progress: Protecting real-time gpu applications on integrated cpu-gpu soc platforms," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2017.

APPENDIX

Theorem 1. *An upper bound on the response time R_i of a GPU task τ_i at the highest interleaving level scheduled with NVIDIA's scheduler can be found when (i) τ_i arrives right after one of its assigned slot elapsed, and (ii) all other tasks in the runlist are released as soon as possible after their execution.*

Proof. Consider the case shown in Fig. 2a, where a task τ_i with high interleaving level submits new commands to the Command Push Buffer right after the GPU Host checked its associated entry ($H2$) in the runlist. In this case, the task will have to wait until its next entry in the runlist. For tasks having the highest interleaving level, this means waiting one instance of each of the tasks having the same interleaving level, plus one instance of only one task with a lower interleaving level. If the considered job of τ_i does not complete its execution due to timeslice exhaustion, a further interfering contribution of the same amount will be experienced before the task can resume execution in a next slot. The interfering contributions are upper bounded considering a situation where each interfering task is re-released right after its execution. Moving τ_i's release earlier would allow it to catch the assigned slot, therefore

decreasing its response time. Moving it later would not change its schedule, also reducing the response time. □

We are interested in determining an upper bound on the response time of a task τ_i with high interleaving level. The maximum time interval l_i between two slots of τ_i's channel in the runlist can be computed as

$$l_i = \sum_{\substack{j=1 \\ j \neq i}}^{NR} \overline{TS_j} + TS_{M/L}$$

where $\overline{TS_j} = \min(TS, C_j)$, $TS_{M/L}$ is the timeslice assigned to the *medium* or *low* priority task and NR is the number of channels in the runlist. The response time R_i of τ_i given a timeslice TS can then be upper bounded as

$$R_i \leq \left\lceil \frac{C_i}{TS} \right\rceil \cdot l_i + C_i, \tag{2}$$

where $\left\lceil \frac{C_i}{TS} \right\rceil$ represents the number of preemptions due to timeslot expiration during the task execution. A simple way to include the preemption overhead contribution to the overall response time can then be derived by including the preemption cost ξ at each timeslot expiration event:

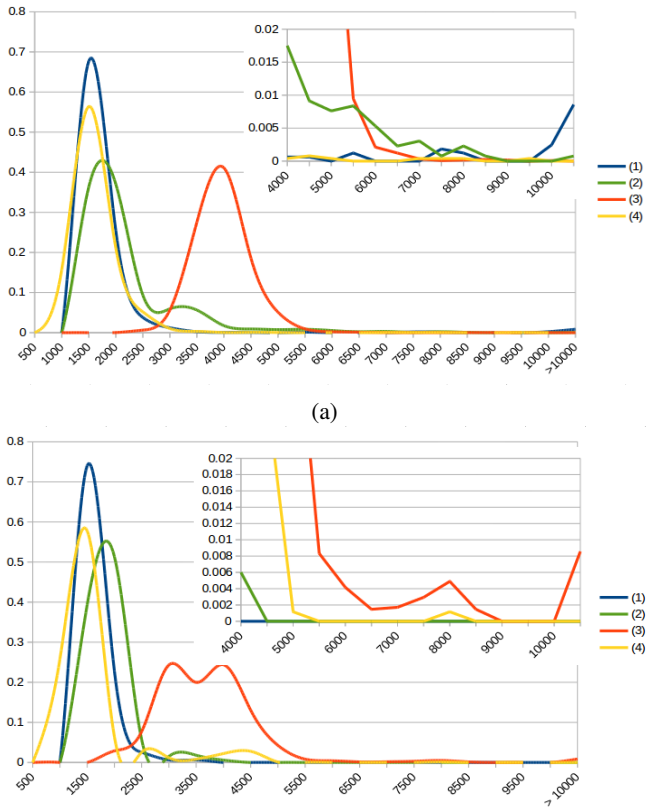
$$R_i \leq \left\lceil \frac{C_i}{TS} \right\rceil (l_i + \xi) + C_i. \tag{3}$$

Listing 1: OpenGL API extension example

```
init_function(){
  //Load data, geometries, textures
  //And compile shaders...
  glSetFrameTarget(framerate, budget_us);
}
Render_loop(){
  //uniforms and attributes updates
  //drawcalls so on...
  glSwapBuffers();
  //Kicks to GPU and waits as specified in
  //init function.
}
```

Listing 2: CUDA API extension example

```
//CUDA ctx creation and data initialization
 init_function(){
   cudaStream_t s0, s1, ..., sn;
   cudaStreamCreate(&s0); ...
   cudaDeviceSynchronize();
 }
//CPU thread0
while (wait_for_new_data()){
cudaStreamDeadlineBegin(s0,Dr0,B0,P0);
//cuda kernels, memcpy etc... on stream s0
cudaStreamDeadlineEnd(s0);
}
```



(a)

(b)

Fig. 4: Detailed response times for tasks (1-4) described in Section VI-A with (a) the interleaved scheduler and (b) EDF+CBS. Response time classes in the horizontal axis are in μs; vertical axis represents the relative frequency. The time interval where real-time task (2) can experience deadline misses (i.e., above 4000 μs) is magnified.