Project title:	High-Performance Real-Time Architectures for Low-Power Embedded Systems
Acronym:	HERCULES
Project ID:	688860
Call identifier:	H2020 – ICT 04-2015 – Customized and low power computing
Project coordinator:	Prof. Marko Bertogna, University of Modena and Reggio Emilia

D5.2: Scheduling Algorithm for Parallel Accelerators

Document title:	Scheduling Algorithm for Parallel Accelerators
Version:	1.1
Deliverable No.:	D5.2
Lead task beneficiary:	CTU
Partners involved:	UNIMORE, CTU, ETHZ
Author:	N. Capodieci, R. Cavicchioli, P. Valente, M. Bertogna, M. Sojka, P. Houdek
Status:	Final
Date:	2018-06-30
Nature:	Report
Dissemination level:	PU – Public

Purpose & Scope

The purpose of this deliverable is to describe scheduling strategies for deterministic execution on the GPU and for limiting the interference GPU puts on other parts of the system via shared resources, mainly via DRAM memory.

Revision History

Version	Date	Author/Reviewer	Comments
0.1	2017-12-20	CTU	Initial draft
0.2	2018-06-20	UNIMORE	Added Deadline-based scheduling for GPU
1.0	2018-06-22	CTU	First complete draft ready for review
1.1 Final	2018-06-30	CTU/ETHZ,MM	Incorporated comments from internal review

HERCULES project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement: 688860.

Contents

Acronyms

API	Application Programming Interface 8
CBS	Constant Bandwidth Server 3
CPU	Central Processing Unit 2
CTA	Cooperative Thread Array 5
CUDA	Compute Unified Device Architecture 11
DNN	Deep Neural Network 5
DRAM	Dynamic Random Access Memory 2
EDF	Earliest Deadline First 3
FPGA	Field Programmable Gate Array 16
FPS	Frame Per Second 11
GPU	Graphics Processing Unit 2
MC	Memory Controller 16
PCIe	PCI Express 16
PREM	Predictable Execution Model 2
RLM	RunList Manager 6
SM	Streaming Multiprocessor 8
SoC	System-on-Chip 3
USB	Universal Serial Bus 16
VM	Virtual Machine 6
WCET	Worst-Case Execution Time 11
WCRT	Worst-Case Response Time

1 Introduction

As we demonstrated in [D2.2], a significant source of interference causing non-predictable execution times is the main memory (Dynamic Random Access Memory (DRAM)), which is shared among many parts of the system-on-chip. Deliverable [D5.1] showed effective methods for mitigating this interference for the Central Processing Units (CPUs), using memory-centric scheduling algorithms and predictable execution models like Predictable Execution Model (PREM) [Pel+11]. However, in the heterogeneous computing platforms addressed by the HERCULES project, DRAM is not only accessed by the CPUs but also by other engines available in the system-on-chip, like integrated Graphics Processing Units (GPUs). In order for PREM to be effective, all major DRAM bandwidth consumers must be scheduled in coordination with the rest of the system not to access the main memory in an uncontrolled way. This topic has already been preliminarily discussed and evaluated in [D3.4], with a specific focus on the GPU. This deliverable describes two system-level techniques for scheduling execution on the GPU and to control its memory access patterns.

Scheduling the GPU is a difficult problem because current generation GPUs feature a hardware-based scheduler, whose function cannot be modified in an arbitrary way. We therefore need to cope with the existing hardware support, building our scheduling on top of it. We propose a GPU scheduler implemented on the CPU by intercepting all GPU kernel invocations.

In Section 2 we first describe how the current GPU scheduler works on NVIDIA embedded platform, to then detail how a prototype EDF scheduler might be implemented at hypervisor level. Do note that the described prototype will not be integrated with the rest of the HERCULES modules, but it is still useful for dissemination purposes.

Section 3 describes how to utilize the features of the NVIDIA memory controller to throttle memory accesses from the GPU, so to protect CPU tasks' response time.

The third method of dealing with GPU scheduling and controlling its DRAM access, which was pursued within the HERCULES consortium, has already been presented in deliverable [D3.4] under the name GPUguard.

1.1 Objectives

The objectives of Task 5.2 are to "*investigate predictable scheduling strategies to exploit the impressive computing power offered by the parallel acceleration engines available on the considered heterogeneous platforms*". Those scheduling strategies should "*avoid contention on shared resources, like memory and communication resources*".

2 Deadline based scheduling on the GPU

To investigate preemptive Earliest Deadline First (EDF) policies for scheduling GPU tasks, we utilize a recently released NVIDIA Tegra-based System-on-Chip (SoC) 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. 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:

1. Transitioning from a pre-existent table-based approach to an event-driven approach for GPU commands submission;
2. Prototyping an enhanced programming model for both CUDA and OpenGL with real-time extensions;
3. Prototyping an alternative SW (software) scheduler implementation that acts as a privileged guest in the hypervisor, improving over the currently implemented NVIDIA scheduler;
4. Integrating our prototype with the pre-existing model for handling dependencies between the NVIDIA computing platform sensors/actuators and the respective GPU tasks.

2.1 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 approach adopted by NVIDIA to GPU scheduling in current automotive-grade boards. It is important to highlight that Section 2.1.1 and 2.1.2 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 2.1.3.

2.1.1 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[1]". 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

[1] 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.

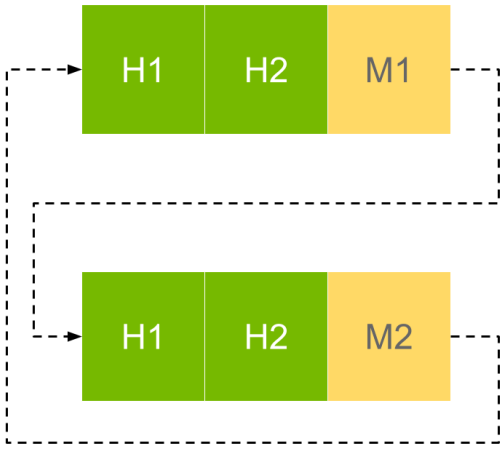

Figure 1: A reconstructed runlist composed of 2 High priority task and 2 Medium priority tasks.

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[2]. Synchronization operations such as acquiring and releasing a semaphore/syncpoint are commands enqueued within a Command Push Buffer.

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 kernel driver stack[3]. 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. Figure 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

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

[3]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`

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 Cooperative Thread Array (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 [Cap+17]. 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 Deep Neural Network (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

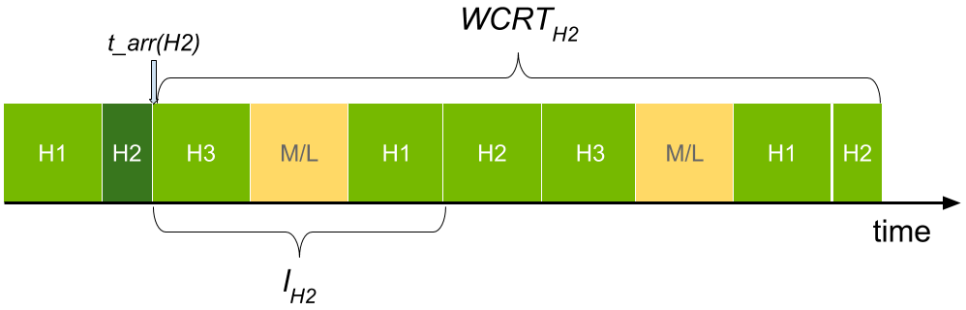

Figure 2: WCRT for a GPU task $H2$ as a function of its arrival time t_arr. Note that the darker green $H2$ is depicted as a short interval because the GPU Host did not find work to dispatch to the engines. Task initials H, M or L indicate their interleaving level. I_{H2} is the latency between $t\_arr(H2)$ and the beginning of H2 execution

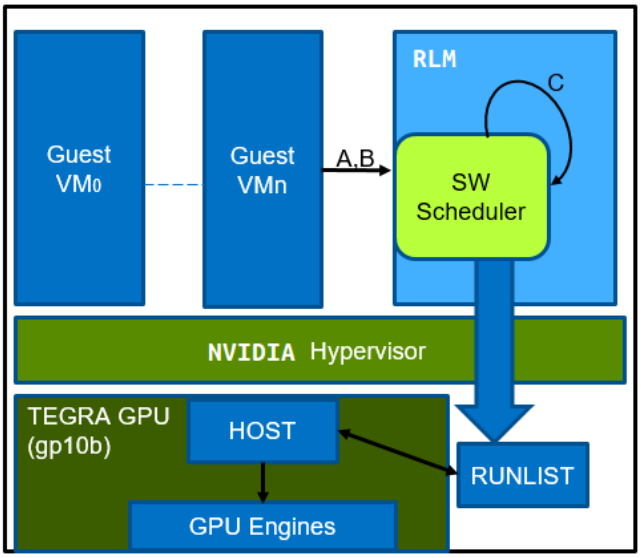

Figure 3: The main blocks characterizing our prototype scheduler implementation. A, B and C are event signals and messages used for scheduling decisions

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.

2.1.2 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-Virtual Machine (VM) communication and resource sharing. The NVIDIA hypervisor follows the bare-metal paradigm, statically assigning memory ranges and HW (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 (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.

2.1.3 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 Figure 3.

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 Figure 3).

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.

2.2 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 Streaming Multiprocessor (SM)s. 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 [Jog+15]. 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 CBS to enforce resource reservation 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.

2.2.1 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 Application Programming Interface (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 2.3.

Inter-process dependencies. Interprocess dependencies at GPU level 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 [16], 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. 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 2.1.3).

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.

2.3 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.

2.4 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.

2.4.1 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 2.1. 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 Frame Per Second (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 Worst-Case Execution Time (WCET) is 4 ms, with an average of 1.2 ms.
(2) A Compute Unified Device Architecture (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 [CMS12]. 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 2.1.1, 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.

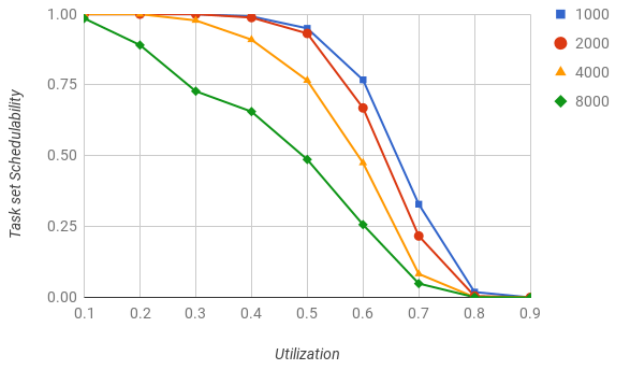

Figure 4: A

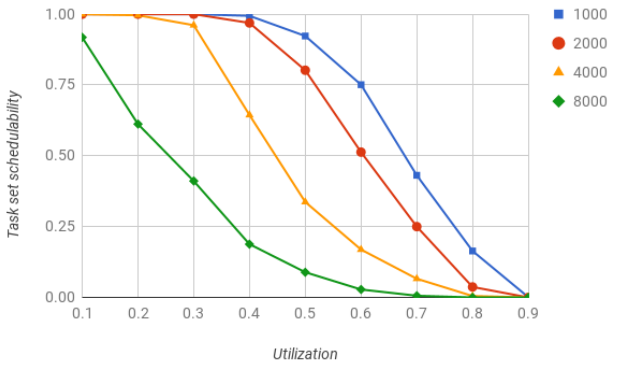

Figure 5: B

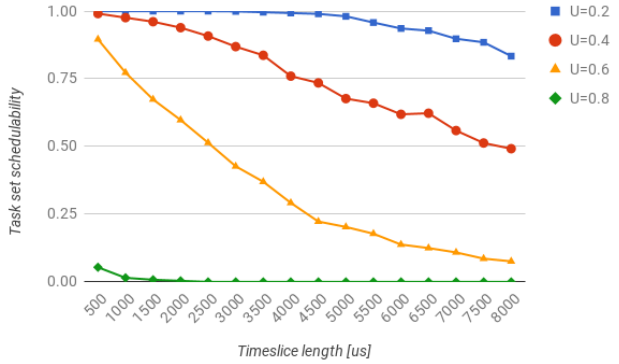

Figure 6: C

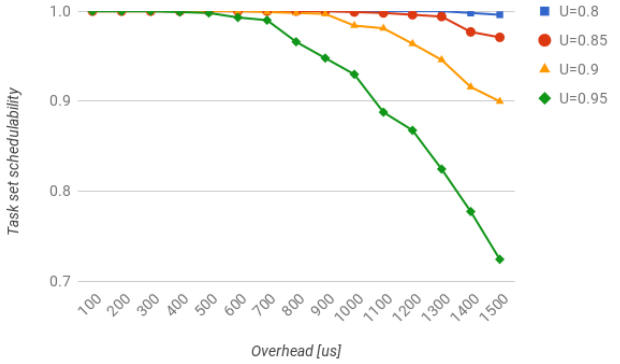

Figure 7: D

Figure 8: Schedulability ratio of NVIDIA scheduler as a function of task set utilization with 5 tasks (figure 4) and 20 tasks (figure 5) for different timeslice lengths *TS* in us; and as a function of timeslice length *TS* with 5 tasks (figure 6) for different utilizations; Inset (figure 7) 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).

2.4.2 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 [BB05] 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 2.1.1, the timeslice determines the maximum continuous ex-

ecution 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 Figure 8 detail the schedulability ratio of the considered algorithms, w here 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 (figure 4) 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 l arger blocking penalty that further deteriorates the schedulable utilization.

Figure 5 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 Figure 6 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 [BMR90], including preemption overhead and CPU-to-GPU communication delay.

A parameterized simulation is visible in Figure 7 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.

2.5 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 Figure 2, 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

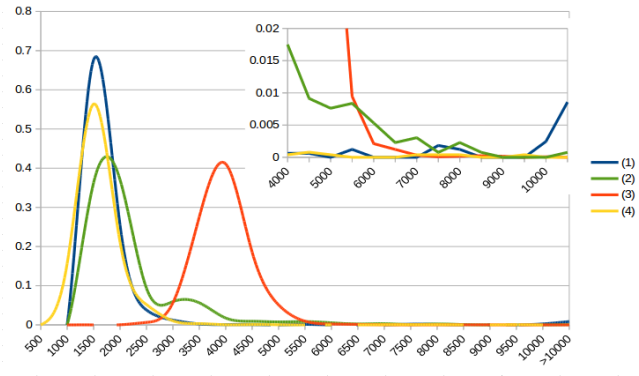

Figure 9: A

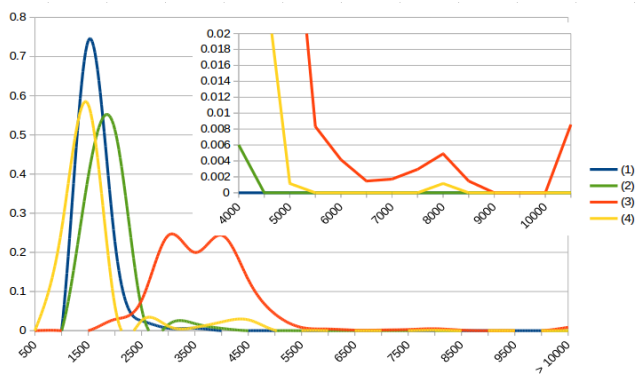

Figure 10: B

Figure 11: Detailed response times for tasks (1-4) described in Section 2.4.1 with (9) the interleaved scheduler and EDF+CBS (10). 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.

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

```
1   init_function (){
2     // Load data , geometries , textures
3     // And compile shaders ...
4     glSetFrameTarget ( framerate , budget_us );
5   }
6   Render_loop (){
7     // uniforms and attributes updates
8     // drawcalls so on ...
9     glSwapBuffers ();
10    // Kicks to GPU and waits as specified in
11    // init function .
12  }
```

Listing 2: CUDA API extension example

```
1  // CUDA ctx creation and data initialization
2    init_function (){
3     cudaStream_t s0, s1, ..., sn;
4     cudaStreamCreate (&s0 ); ...
5      cudaDeviceSynchronize ();
6   }
7  // CPU thread0
8  while ( wait_for_new_data ()){
9   cudaStreamDeadlineBegin (s0 ,Dr0 ,B0 ,P0 );
10 // cuda kernels , memcpy etc ... on stream s0
11 cudaStreamDeadlineEnd (s0 );
12 }
```

3 GPU throttling at memory controller

The *SiT* mechanism described in [Cap+17] throttles GPU memory accesses by either preempting GPU computation with high-priority spin kernel or by splitting longer CPU-GPU memory copies to multiple shorter ones and launching them according to an arbitration policy. Bak et al. [Bak+] propose to use Field Programmable Gate Array (FPGA)-based real-time PCI Express (PCIe) bridges for this purpose. But such a method cannot be used for a system-on-chip such as NVIDIA TX1. NVIDIA SoCs provide another mechanism that can be used to control how the GPU accesses memory. NVIDIA's memory controller includes several "knobs", which can modify certain aspects of how memory requests are arbitrated at the hardware level.

In this section, we analyze the arbitration mechanisms available in the NVIDIA memory controller and how they can be used to increase predictability of CPU task execution times by limiting the negative effect of the GPUs as presented in [Cap+17] and [CCB17].

3.1 Experimental platform

For the experiments in this section, we used a NVIDIA Jetson X1 board. NVIDIA ships TX1 with Linux kernel version 3.18 from Linux for Tegra (L4T) version 24.1. We used the kernel sources shipped by NVIDIA, but we recompiled them with a modified configuration to disable power-management features that can influence results of the experiments. We used 64-bit userspace from L4T 24.1 and CUDA v7.0 for L4T v24.1.

3.1.1 TX1 memory controller

NVIDIA's Memory Controller (MC) is a piece of hardware that receives requests from on-chip clients such as CPUs, GPU, Universal Serial Bus (USB) etc. and executes them according to a partially configurable policy. The memory controller, its clients and DRAM connection is depicted in Figure 12.

Operation of the memory controller can be configured via its registers. There are several hundreds of them. The most important registers for our purpose are:

- **THROTTLE** register specifies how many idle cycles are inserted between client requests and
- each bit in the **THROTTLE_MASK** register specifies whether a specific memory client takes part in throttling or not.
- Furthermore, each client has a corresponding **LATENCY_ALLOWANCE** register, which controls arbitration policy in so called *latency arbitration mode*, which is entered when any request waits in the memory controller longer than its configured deadline.

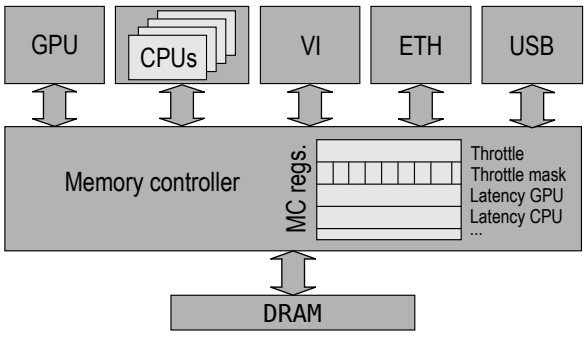

Figure 12: Simplified block diagram of Tegra X1 memory controller, its clients (at the top) and DRAM

Listing 3: Core part of the CPU benchmark

```
1  struct cacheline {
2    struct cacheline *next;
3    char gap[64 - sizeof(struct cacheline*)];
4  };
5
6  void workload(void)
7  {
8    struct cacheline *p;
9    int repeat = 1000000;
10   p = make_array(working_set_size, RANDOM);
11
12   start_measurement();
13   while (--repeat) {
14 #ifdef WRITE
15       p->gap[0]++;
16 #endif
17       p = p->next;
18   }
19   end_measurement();
20 }
```

Besides those registers, the memory controller also contains so called *statistics counters* that allow counting the accesses to the main memory from various clients including CPU cores, GPU, Ethernet controller and others. These counters are useful for analyzing behavior of the system as we have shown in [HSH17]. The limitation of statistics counters is their high power consumption for which, it is not recommended to use them in production. We are, therefore, not interested in throttling methods based on them.

3.2 Workload

In the experiments described later, we use synthetic workload to stress the memory controller.

On the CPU, we use a simple pointer chasing program [Dre07] as the workload. The core part of this program can be seen in Listing 3. Function `make_array` allocates a chunk of memory of the given size and creates a circular linked list in it. This workload can be parameterized with two binary parameters: WRITE and RANDOM. Without WRITE, the program only reads the memory, with it, memory writes are also performed. The RANDOM parameter specifies the ordering of list elements. Without RANDOM, list elements are ordered so that list traversal generates sequential memory accesses, with RANDOM, the memory accesses are random. We measure execution time needed to traverse a high number (1 million) of list elements. This structure of the workload ensures there is almost no CPU overhead during measurement, making the workload memory-bound.

On the GPU, we use basic operation from linear algebra called *saxpy* (product of scalar and vector is summed with another vector). We use vectors that consist of 10 million floating-point numbers because we need *saxpy* to be memory-bound. Also, when we need to adjust *saxpy* workload execution time we change the number of iterations steps. Because execution time of one *saxpy* computation is nearly non-measurable, we use a batch of 140 *saxpy* operations per iteration.

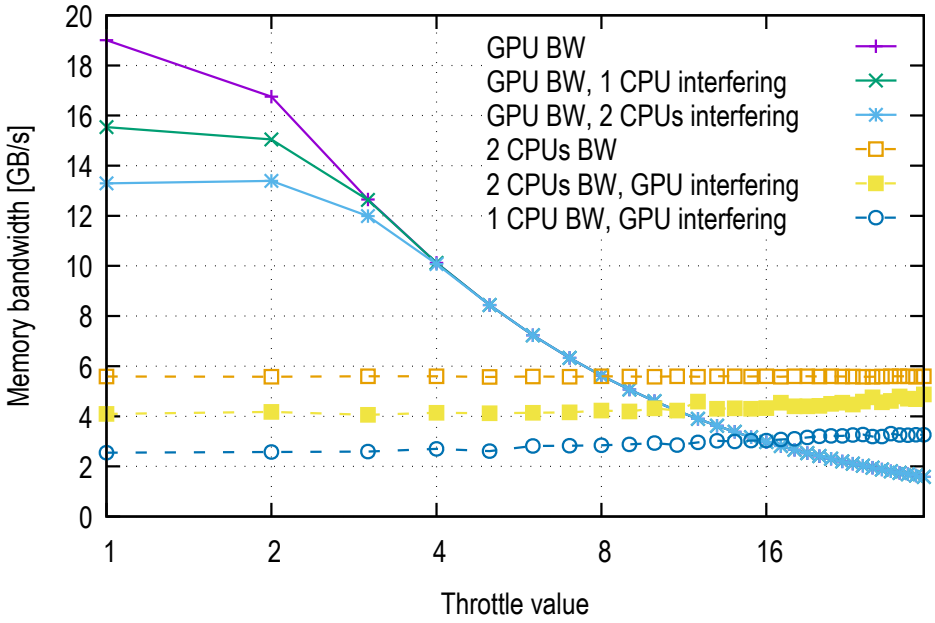

Figure 13: Effect of GPU memory bandwidth throttling in the memory controller

3.3 Throttling of GPU memory bandwidth

Before we analyze the effect of GPU throttling on CPU latencies, we first verify the functionality of throttling by measuring memory bandwidth. We run our pointer chasing program (sequential read and write version) on one or more CPUs together with a memory-bound *saxpy* kernel on the GPU. By measuring the time of executed operations, we obtained the memory bandwidth consumed by the CPUs and the GPU while changing the value of the *THROTTLE* register. Only throttling of the GPU was enabled in the *THROTTLE_MASK* register. Results can be seen in Figure 13. Solid lines (*GPU BW, ...*) show the memory bandwidth consumed by the GPU with or without interference from the CPUs. Dashed lines (*x CPU BW...*) show how much bandwidth was used by CPUs during the same test. Curves *GPU BW* and *2CPU BW* show the bandwidth without any interference from the CPU or GPU, respectively. We can see that the GPU can consume up to 19.5 GB/s, but its bandwidth can be throttled to almost any lower value. This proves that the throttling mechanism works as expected. At the same time, CPU bandwidth is almost the same. We can only see a small increase (about 1 GB/s) of the CPU bandwidth as the GPU bandwidth gets throttled, which is caused by smaller number DRAM bank collisions when the GPU is throttled.

3.4 Decreasing GPU-induced interference

With the GPU throttling mechanism in place, we are now interested in limiting the effect of the GPU on the execution time variations of CPU tasks.

To achieve predictable execution time, memory phases of PREM-compliant tasks running on different resources should not overlap, but to increase efficiency, we want the memory phases to overlap with computation phases on other resources. If a memory phase needs to be executed simultaneously with legacy code, we need a way of guaranteeing that the memory accesses in the legacy code do not interfere with the memory phase. Therefore, the goal of this experiment is to evaluate the effectiveness of MC throttling for limiting the GPU interference on CPU memory phase.

We construct an experiment as shown in Fig. 14. The long gray rectangles in the figure represent the memory-

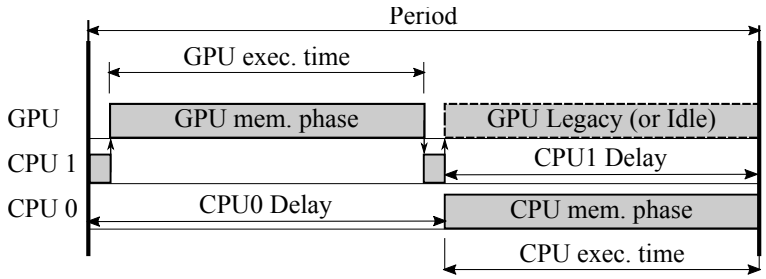

Figure 14: GPU interference experiment (Gantt chart of a single iteration)

bound workload described in Section 3.2 performing sequential memory accesses. We use them to mimic the behavior of PREM memory phases or to represent non-PREM legacy code (so called compatible intervals). Short rectangles on CPU 1 are tasks whose sole purpose is launching of the GPU kernel and measuring its length.

We perform a series of experiments with different parameters that evaluate different scenarios. The results of all experiments are shown in Figure 15. In the two leftmost experiments (CPU only and GPU only) we measure the execution time of the CPU and GPU tasks (denoted as *mem. phase* in Fig. 14) when they run alone in the system and we use the obtained times as a reference for subsequent measurements. In absolute terms, the measured execution times were 900 ms and 800 ms respectively. All results in Figure 15 are reported relative to these values. Therefore, the two leftmost bars in the figure are 100%; the error bars show minimum and maximum in 20 repetitions.

The next experiment, *Unsync.*, shows the results when both GPU and CPU memory phases were not synchronized. The tasks executed on CPU0 and CPU1 comprised the invocation of the *mem. phase* workload and a *delay*. As we can see from the error bars, execution time varies significantly depending on whether the memory phases overlapped or not. If we repeat the experiment, but synchronise (co-schedule) the execution on CPU 0 and 1 (*Sync.*), memory phases do not overlap and execution time is predictably low, almost 100% as in the first two experiments.

Next, we replace the GPU idle time by executing the same GPU-kernel as a legacy application (*Sync.+Legacy*). We observe 50% increase of CPU memory phase execution time. If we execute the legacy application with maximum throttling of GPU memory accesses, the increase of CPU memory phases execution time is only 10%

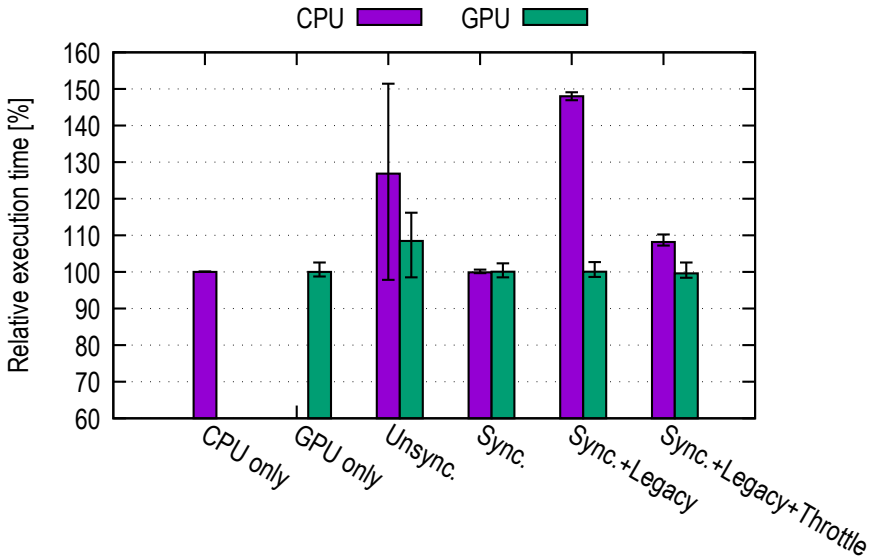

Figure 15: Execution times of memory phases with and without synchronization and GPU throttling

(*Sync.+Legacy+Throttle)*. This shows, that throttling at memory controller level can allow executing legacy applications on the GPU, without significantly influencing execution times of PREM memory phases on the CPU.

3.5 Limitations

As was shown above, the throttling mechanism can be used to mitigate the interference GPU puts on the CPUs, but the use of this kind of throttling also has some limitations.

1. Changing the throttle register in the memory controller is not for free. We measured that the setting of the throttle register takes around 1000 CPU clock cycles ($\approx$ 1 µs).
2. It is not possible to throttle different clients by different amount. This is due to the fact that there is only one *THROTTLE* register for multiple clients and one can only select with *THROTTLE_MASK* whether to switch throttling on or off.
3. Although throttling can be enabled also for the CPUs, doing so had no measurable effect on our benchmarks.
4. In addition to throttle registers, TX1 memory controller also contains latency registers that seem more suited for what we want to achieve. Their effect should be visible when the memory bandwidth is saturated and requests get queued in the memory controller. Despite our effort of saturating the memory bandwidth by multiple clients (CPU, GPU, Ethernet, Video Input), we were not able to measure the effect of the changes in these registers.
5. Limiting GPU memory bandwidth can prevent graphical output to display things correctly. When the GPU does not have enough bandwidth to prepare frame for the display, the display shows various artifacts such as missing parts of the screen etc. For this reason GPU throttling is only applicable in GPU-compute-only applications without graphical output.

4 Conclusion

In this deliverable we detailed how the scheduler for NVIDIA integrated GPUs work and how we can prototype a hypervisor privileged guest so to enforce arbitrary scheduling policies implemented in software. We demonstrated our approach by implementing EDF on a NVIDIA integrated GPU. In the second part of this deliverable, we utilize the features of NVIDIA memory controller to throttle memory accesses from the GPU, allowing the execution of legacy applications on the GPU together with PREM-compliant applications on the CPU.

The final objective is to implement system-wide server-based arbitration policies at hypervisor level (planned for deliverable D4.5) that are able to orchestrate CPU and GPU real-time tasks with both presented techniques. The HERCULES compiler and the GPUguard runtime presented in deliverable [**D3.2**] and [D3.4], respectively, complement the techniques presented in this deliverable to produce PREM-ized binaries of CPU and GPU tasks, and coordinate their execution between both engines.

References

[16] *Khronos EGLStream specification*. `https://www.khronos.org/registry/EGL/extensions/KHR/`. Khronos, 2016.

[Bak+] Stanley Bak et al. “Real-Time Control of I/O COTS Peripherals for Embedded Systems”. In: *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*.

[BB05] Enrico Bini and Giorgio C Buttazzo. “Measuring the performance of schedulability tests”. In: *Real-Time Systems* 30.1 (2005), pp. 129–154.

[BMR90] Sanjoy Baruah, Aloysius K. Mok, and Louis 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.

[Cap+17] Nicola Capodieci et al. “SiGAMMA: Server Based Integrated GPU Arbitration Mechanism for Memory Accesses”. In: *Proceedings of the 25th International Conference on Real-Time Networks and Systems*. RTNS ’17. Grenoble, France: ACM, 2017, pp. 48–57. ISBN: 978-1-4503-5286-4.

[CCB17] 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.

[CMS12] Dan Ciregan, Ueli Meier, and Jürgen 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.

[D2.2] Hercules consortium. *D2.2: Detailed characterization of platforms*. Deliverable of the HERCULES project. June 2017.

[D3.4] Hercules consortium. *D3.4: Preliminary runtime for parallel heterogeneous platform*. Deliverable of the HERCULES project. June 2017.

[D5.1] Hercules consortium. *D5.1: Power-Aware Scheduling Algorithm for Host*. Deliverable of the HERCULES project. Dec. 2017.

[Dre07] Ulrich Drepper. *What Every Programmer Should Know About Memory*. Red Hat, Inc. Nov. 21, 2007. URL: `https://www.akkadia.org/drepper/cpumemory.pdf`.

[HSH17] P. Houdek, M. Sojka, and Z. Hanzálek. “Towards predictable execution model on ARM-based heterogeneous platforms”. In: *2017 IEEE 26th International Symposium on Industrial Electronics (ISIE)*. June 2017, pp. 1297–1302. DOI: `10.1109/ISIE.2017.8001432`.

[Jog+15] Adwait Jog et al. “Anatomy of gpu memory system for multi-application execution”. In: *Proceedings of the 2015 International Symposium on Memory Systems*. ACM. 2015.

[Pel+11] Rodolfo Pellizzoni et al. “A predictable execution model for COTS-based embedded systems”. In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2011, pp. 269–279.