SiGAMMA: Server based integrated GPU Arbitration Mechanism for Memory Accesses

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

ABSTRACT

In embedded systems, CPUs and GPUs typically share main memory. The resulting memory contention may significantly inflate the duration of CPU tasks in a hard-to-predict way. Despite initial solutions have been devised to control this undesired inflation, these approaches do not consider the interference due to memory-intensive components in COTS embedded systems like integrated Graphical Processing Units. Dealing with this kind of interference might require custom-made hardware components that are not integrated in off-the-shelf platforms. We address these important issues by proposing a memory-arbitration mechanism, SiGAMMA ($Si\Gamma$), for eliminating the interference on CPU tasks caused by conflicting memory requests from the GPU. Tasks on the CPU are assumed to comply with a prefetch-based execution model (PREM) proposed in the real-time literature, while memory accesses from the GPU are arbitrated through a predictable mechanism that avoids contention. Our experiments show that $Si\Gamma$ proves to be very effective in guaranteeing almost null inflation to memory phases of CPU tasks, while at the same time avoiding excessive starvation of GPU tasks.

CCS CONCEPTS

• **Computer systems organization → Real-time system architecture**; *Single instruction, multiple data*; *Embedded software*; Multicore architectures;

KEYWORDS

GP-GPU, PREM, Memory-Centric Scheduling

ACM Reference format:
Nicola Capodieci, Roberto Cavicchioli, Paolo Valente and Marko Bertogna. 2017. SiGAMMA: Server based integrated GPU Arbitration Mechanism for Memory Accesses. In *Proceedings of RTNS '17, Grenoble, France, October 4–6, 2017,* 10 pages.
https://doi.org/10.1145/3139258.3139270

1 INTRODUCTION

GP-GPU (General Purpose GPU) computing is a very effective way to perform embarrassingly parallel computations in embedded devices, obtaining impressive performance at a limited power consumption [18]. Such computations are described through different APIs (Application Programming Interfaces) such as OpenCL, CUDA, DirectCompute, etc. These APIs assume a host initiates the computation and offloads parallel workload (i.e., computing kernels) to the GPU (also called device). As an example, see Figure 1, the CUDA architecture relies on a plurality of engines: the Execution Engine (EE) for computation and the Copy Engine (CE) for high throughput DMA transfers.

The Execution Engine reaches a high level of SIMD parallelism by exploiting hundreds of *CUDA cores* per Streaming Multiprocessor (SM). A hardware scheduler dispatches groups of 32 threads in lockstep to CUDA cores (these are called *warps* in CUDA terminology). In their turn, warps are grouped into *blocks*. The programmer can define a *launch configuration* for a *CUDA offload* (*CUDA kernel invocation*) by defining number of blocks and size of each block in terms of threads. Launch configurations can be logically spatially organized in three dimensions, and this is referred as *launch grid*. The CUDA kernel invocation configuration and respective launch grid determines how data to compute is accessed in terms of memory access patterns.

Both EE and CE, as well as the host system, may perform memory operations. On one side, EE cores that are not able to find the required data in local caches must access off-chip DRAM. On the other side, a CE can be used to perform copies of buffers from host visible memory to an area that is only visible to the GPU, or viceversa. When CPU and GPU have physically separated memories, such as in discrete GPU configurations, the memory contention between host and device is limited to DMA transfers through PCIe buses. On the other hand, in integrated Graphical Processing Units (iGPUs), CPU and GPU share the same DRAM, introducing a contention point that may affect the predictability of the system. In integrated solutions, contention happens on system DRAM, as GPU related DMA transfers (that still cause data movements in GPU L2 caches), do not affect data located in the CPU complex private caches (see Figure 1).

In integrated System-on-Chips (SoC), safety-critical tasks with tight deadlines are typically executed at the host side, offloading parallel kernels with a lower criticality to the GPU. Thus, memory contention may represent a significant threat for predictability and timing analysis of safety-critical CPU tasks. Concurrent memory accesses by CPU cores and other memory clients often undergo undisclosed, or non-priority-driven, arbitration policies at the memory controller level. The problem is magnified if the considered SoC features a high performance GPU able to saturate the available memory bandwidth. As will be shown in our experiments, the worst-case execution time of a real-time task at the host-side

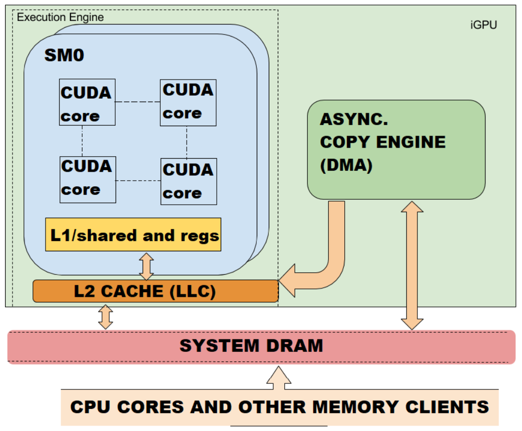


Figure 1: CUDA model and integrated GPU architecture. Focus on contention points and GPU engines

may increase up to 5 times due to the concurrent execution of a memory-intensive GPU application in modern integrated devices.

Such a significant latency increase motivates the need for a proper arbitration mechanism of memory accesses between host and device. In this paper, we present *SiGAMMA* (transl. *Si*Γ), a server-based mechanism that acts as a memory arbiter between CPU and GPU, moderating the penalties due to the concurrent memory accesses by GPU engines. CPU tasks are assumed to be compliant with a predictable execution model (PREM) that separates memory phases from purely computation phases, as explained in [17]. Such an execution model allows predictably bounding the delays due to the concurrent access to shared memory resources by real-time tasks in multi-core environments. Variants of this model are adopted at industrial level for automotive [13] and avionic [9] applications to decouple computation and communication phases of critical tasks and schedule them in a predictable way to avoid contention on shared resources.

While being inspired by the above mentioned memory arbitration mechanisms, *Si*Γ presents a set of distinguishing features:

- It uses a dynamic, event-based approach that allows for a better utilization of the memory bandwidth also for dynamic task sets.

- It does not rely on hardware counters, that, depending on the SoC implementation, may be too coarse grained to be used in real-time settings, especially for general-purpose embedded devices.

- It allows throttling GPU-side activities to limit their memory interference in heterogeneous SoCs using a novel and flexible mechanism, that is applicable also to closed-source architectures in which the possibility of modifying drivers is extremely limited.

The combination of these features will allow making the CPU-side memory request more predictable, without overly affecting performance at GPU side, as will be detailed in our experiments.

The paper is organized as follows. In section 2, we present a small survey of the literature regarding memory arbitration mechanisms in embedded devices and GPU arbitration strategies. In section 3, we provide a complete description of the boards used in our experiments, and a measure of the impact of memory-intensive GPU applications on the duration of CPU tasks. Section 4 formally presents *Si*Γ. Experiments and results are detailed in section 5. The limitations of our approach are discussed in section 6. Conclusive remarks and discussion regarding how to extend the presented research are included in section 7.

2 RELATED WORK

Related work covers mainly two topics: GPU scheduling and memory arbitration strategies. For the first topic, we provide a summary of the state-of-the-art regarding real-time scheduling on GPU, and existing mechanisms for preemption control at the GPU side. This will allow placing our GPU-side throttling mechanism within the existing literature. For the second topic, we detail existing works focusing on predictable memory-centric scheduling policies and execution models.

2.1 GPU Scheduling

The amount of literature regarding GPU scheduling is not as large as the contributions available for CPU or I/O scheduling. Kato et al. proposed TimeGraph [15], a real-time GPU scheduler that schedules GPU tasks characterized by priority levels. This is done by modifying an open-source GPU driver and monitoring the commands at driver level. Unfortunately, accessing GPU drivers is not always possible. For example, the NVIDIA based boards we adopted in our experiments rely on closed-source drivers, with a much better performance than competing open source solutions.

In a more recent publication, Schnitzer et al. [20] proposed a Reservation-based scheduling mechanism applied to GPU tasks, that still relies on open source GPU drivers. The goal is to schedule 3D-graphics tasks of an automotive application so that the real-time constraints are satisfied. The reservation-based approach proved to be very effective towards the GPU model in which the underlying assumption is the impossibility to preempt. Lower priority tasks are scheduled only if there is sufficient time to schedule higher priority GPU jobs before their deadline. These approaches, while obtaining good results in meeting deadlines and achieving predictability from the GPU perspective, have not considered the effects of interference operated by GPU tasks on CPU threads in systems that share the same DRAM.

Elliot at al. in [10] proposed a different model, in which GPU engines are seen as mutually-exclusive resources that can be accessed only by given real-time locking protocols. Based on this assumption, they developed GPUSync, a software framework for GPU management in multi-core real-time systems. The target is again obtaining real-time compliance from the GPU side, without considering the memory interference to the CPU. A limitation that is shared by all the previous approaches is that the GPU is always considered as a non-preemptable resource, reducing the scheduling strategies that can be applied in such scenarios. This may be an over-constraining assumption, since preemption of a CUDA kernel can be achieved by splitting a single kernel invocation into many different ones [4, 26]. While this might lead to a better control over the scheduled blocks (as we are able to take different scheduling decisions between kernel invocations), this solution adds a significant overhead at the CPU side.

The Persistent Thread programming model [12] is another attempt to "hijack" the undisclosed GPU scheduling policies by batching many kernel calls into a single invocation to then arbitrate the execution of blocks of GPU threads with user-defined scheduling and synchronization policies [23]. Reducing driver overhead, both performance and GPU utilization are proved to be significantly better than with traditional offloading models for various applications [6, 7]. Despite these performance improvements, persistent threads leave even less control to the CPU to preempt GPU activities. Moreover, transitioning to persistent threads implies changes at kernel code level that might be not so trivial to implement. In this respect, our *Si*Γ approach implies only minor host code refactoring, leaving GPU kernels untouched. By carefully reverse engineering the CUDA model, we were able to achieve user-space preemption of unmodified CUDA kernels in order to enforce our memory arbitration mechanism. This mechanism only requires the platform to support CUDA priority streams, allowing a fine-grained control over previously scheduled GPU warps, limiting the contention due to memory-intensive kernels for real-time applications.

2.2 Memory access arbitration

A pioneering work targeting the memory contention problem in embedded multi-core systems is presented by Pellizzoni et al. in [17]. In this work, the Predictable Execution Model (PREM) is proposed, along with a set of compiler extensions able to reformat existing application code dividing it into the following phases:

- Memory phase: a pre-fetching phase in which SPMs (Scratch Pad Memories) and/or CPU caches are loaded with the necessary data to be used in the following phase.
- Execution phase: the phase in which the CPU computes data pre-fetched in the local memory during the previous phase.
- Compatible intervals: phases accessing external devices (e.g., I/O storage), or executing code that cannot be reformatted in a PREM fashion.

This phase distinction operated on PREM-ized applications allows execution phases to compute pre-fetched data without resorting to main memory, while another client may access main memory without being interfered. In other words, this programming model can be easily coupled with proper scheduling techniques in order to prevent simultaneous memory phases coming from different cores to overlap in time, hence drastically reducing, or even eliminating, the contention on shared memory. A CPU multi-core extension of the PREM model and further refinements of the concept of memory-centric scheduling is presented in [3].

A more event-based approach has been recently proposed in *bwlock* [24], protecting critical memory-intensive sections of real-time applications by means of specifically designed system calls implemented through a kernel module. The authors show a better isolation and an improved performance with respect to PREM-based approaches.

While all of these works form the basis of our research, the PREM model was never applied in order to moderate the interference coming from a high performance iGPU. Since the impact of unregulated GPU-CPU memory accesses may be dramatic, as will be shown in section 3, this paper addresses this shortcoming. Our proposed server-based mechanism relies on having CPU real-time tasks complying with the PREM model, so to have distinct memory and execution phases. Such phases are previously estimated in terms of Worst Case Execution Time (WCET)[1].

A different way to arbitrate memory bandwidth between different cores in multi-core CPUs is given by the MEMGUARD mechanism presented in [25]. MEMGUARD is a kernel module that keeps track of the memory usage of cores or threads by using hardware performance counters. When a previously specified threshold of LLC (Last Level Cache) misses is reached, the module will spin the misbehaving core by forcing the scheduling of a higher priority process, which could be accomplished with a SCHED_FIFO process, whose priority is set to 99. This will prevent any other access to central memory for a given time window. We implemented the same kind of mechanism with two fundamental differences: (i) we do not use hardware counters, as they are known to provide unreliable measurements in many commodity SoCs [22]; and (ii) we spin a CUDA kernel on the GPU instead of a CPU thread.

3 MEASURING GPU TO CPU INTERFERENCE

In this section, we provide a complete description of the boards used in our experiments (NVIDIA TX1 and TK1), to then characterize them in terms of GPU-to-CPU memory interference. We are mainly interested in measuring the increase of memory access latencies of CPU tasks due to GPU activity.

3.1 Boards description

NVIDIA TK1 [1] and TX1 [2] are hybrid SoCs. Tegra K1 consists of a quad-core 2.3GHz ARM Cortex-A15 CPU (32kb I-cache + 32kb D-cache L1 per core, 2MB L2 cache common to all cores). An A15 shadow-core for battery-saving features is also present. The iGPU is a Kepler generation “GK20a" with 192 CUDA cores grouped in 1 SM (Streaming Multiprocessor). The development board is equipped with 2GB of LPDDR3 SDRAM working at 933MHz and 16GB of fast eMMC.

Tegra X1 consists of a quad-core 1.73GHz ARMv8 cluster with Cortex-A57 CPU (80kb L1 per core, 2MB L2 cache common to all cores) coupled with a low-power quad-core A53 CPU cluster and a Maxwell generation iGPU “GM20b" with 256 CUDA cores grouped in 2 SMs. The board features 4GB of LPDDR4 SDRAM and 16GB of eMMC.

The compute capability of a CUDA device is represented by a version number, also sometimes called “SM version". This version number identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available to the CUDA task. In our work, SM capability determines the support of discrete CUDA stream priorities, which is a necessary feature we exploit for enabling CUDA kernel pre-emption (see section 4.3). TK1 SM has capability 3.2 but it just supports a single priority value for its streams. TX1 SM has capability 5.3 and it supports two discrete stream priority levels (HIGH and LOW).

[1]We will use the term WCET also to denote the maximum time of memory phases, even if CPU execution is not involved.

Table 1: Platform-specific memory bandwidth [GB/s]

TK1			*TX1*		
CPU BW	**SYS BW**	**CE BW**	**CPU BW**	**SYS BW**	**CE BW**
3.5	14.9	6/12	4.5	25.6	10/20

3.2 GPU impact on latencies

In this section, we measure the impact of GPU interference w.r.t. CPU tasks in terms of latency increases when accessing system DRAM. We define a baseline scenario in which a single core sequentially accesses differently sized working sets and infer the average accessing time for a single memory access (i.e., the time it takes to access a word). Then, we repeat the same measurements co-running on the GPU an interfering CUDA application that accesses memory according to different paradigms:

- Executing a copy kernel that copies element-wise data between two buffers (*CUDA kernel*);
- Executing a copy kernel involving unified memory located buffers (*CUDA UVM*);
- Copying a buffer by means of the copy engine (*CUDA memcpy*); and
- Zeroing a buffer (using *cudaMemset*).

Table 1 shows the bandwidth for sequential reading operations of a single CPU core (CPU BW), the total memory bandwidth available by the specific DRAM configuration (SYS BW) and the bandwidth consumption of the GPU Copy Engine (CE BW). This latter value is split into two numbers: the first one refers to host to device copies (H2D) (i.e., the bandwidth used when copying a memory region from a host visible region to a GPU visible area, or viceversa), while the second one refers to the bandwidth used by the GPU when performing a copy (memcpy) between GPU-visible memory regions (D2D).

Figures 2a and 2b show the latency increase experienced by the CPU when interfered by GPU activities. The vertical lines correspond to the size of L1 and L2 caches. For working sizes larger than the last level cache (hence, accessing DRAM), the GPU may increase the CPU-side memory access delays by up to 300% for the TX1 board, and up to almost 500% for the TK1. A similar trend was noticed in several other experiments we conducted to highlight the latencies experienced at host and GPU side for different interfering patterns and alternative heterogeneous platforms. More accurate discussions and latency measurements have been published in [8]. Such an impressive slowdown motivated our research for more predictable and efficient mechanisms to arbitrate the access to shared memory resources in high-performance heterogeneous platforms.

4 THE SIGAMMA APPROACH

In this section, we introduce *Si*Γ, a mechanism to protect latency-critical CPU tasks from GPU-side memory interference. To better explain the underlying arbitration mechanism, we initially assume the time-critical application running on the CPU be composed of a single periodic PREM task. We will later show how the mechanism can be easily extended to work with multiple latency-sensitive tasks in section 6.

The PREM task is composed of three phases: load, compute and unload. During the first phase, the task loads all data required for the subsequent computation phase in LLC. The next phase computes the pre-fetched data from the local memory, without needing to access system DRAM. In the last phase, elaborated data is flushed into main memory and/or copied in output buffers. Critical tasks having working set sizes larger than the LLC may be conveniently split into multiple PREM-ized iterations.

Our mechanism will assure no interference is experienced by the critical task in the memory phases, allowing it to exploit its maximum bandwidth. Moreover, the absence of DRAM interference allows fully exploiting burst transfer features at memory controller level when accessing sequential chunks of data. Finally, the number of row buffer misses at memory bank level are significantly decreased due to the absence of concurrent memory requests.

On the GPU side, we define a GPU thread as a thread running on the CPU responsible for sending GPU jobs related commands to the GPU driver using CUDA as API, not a thread executing on the GPU. The CUDA application will use either the Copy Engine or the Execution Engine. A common programming paradigm in CUDA is to copy data from host visible region to device-only memory regions (H2D copy) using the Copy Engine; elaborate the data on the Execution Engine through a kernel invocation; and read back the computed data, copying it back from the device-only accessible region to host visible memory (D2H). One can mistakenly assume this execution model be PREM compliant. Unfortunately, this is not the case, since a CUDA kernel may often access shared DRAM even during its execution phase.

In embedded systems, this programming paradigm implies duplicating the memory consumed for GPU offloads. Still, there are several reasons why using this model instead of leveraging the coherency mechanisms of CPU/GPU caches:

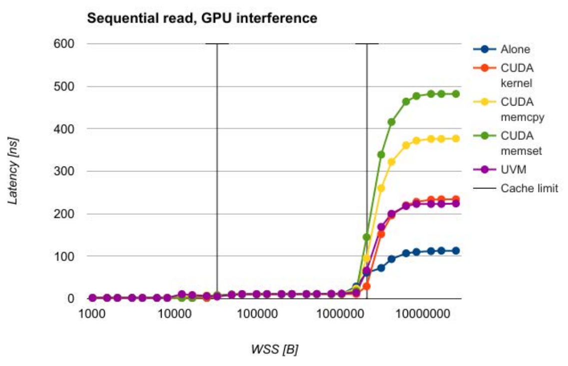
- Unified Virtual Memory (UVM) [19]) and similar CUDA based coherency mechanisms are not easy to profile and analyze in such closed and proprietary environments, preventing their adoption for real-time applications.
- Data transfer is a key aspect that needs to be explicitly considered by a programmer for maximizing the performance of GP-GPU computing applications [16]. Leaving the management of memory transfers to coherency mechanisms does not lead to better performance[2].
- CUDA is an API that exposes the same set of functions both to embedded iGPUs or to their high-end discrete counterpart. Porting a desktop application onto a mobile/embedded platform without significant code refactoring would then imply using a copy-based approach, as usually done for discrete GPU applications.

Having clarified the considered task model for CPU and GPU workloads, the next subsections describe the approach proposed to predictably limit the potential memory interference of uncoordinated CPU/GPU activities.

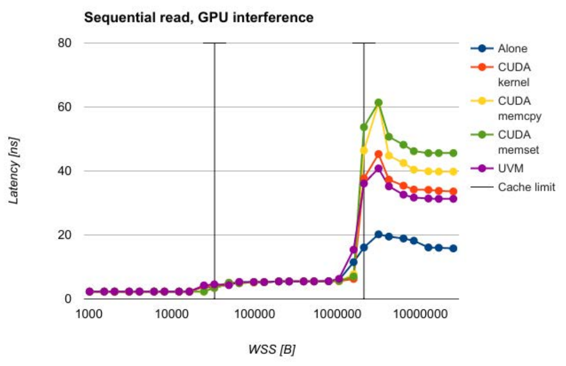
4.1 Memory arbitration mechanism

The *Si*Γ approach adopts a memory server that arbitrates the access to shared DRAM. Any host thread wanting to start a memory or computation phase needs to first access this server to notify the duration and nature of the following phase, namely, a memory phase (CPU_MEM), a computation phase (CPU_COM), or no operation

[2]https://github.com/Sarahild/CudaMemoryExperiments/tree/master/MemCpyExperiments



(a)



(b)

Figure 2: CPU-GPU interference latencies for NVIDIA Jetson TK1 (a) and TX1 (b). Vertical lines indicate CPU L1 and L2 cache boundaries and WSS refers to the CPU Working Set Size

(NO_OPER). The first one is used to signal to the server that a critical CPU thread is asking to exclusively access system DRAM; the second one informs the server that the CPU is about to undergo a compute phase and it will not resort to DRAM accesses; the latter one informs the server that CPU threads PREM iterations are over, hence releasing any locking privileges for system memory.

The GPU thread has read-only access to the server. Before performing any operation, it contacts the server to monitor the ongoing phase at the CPU side. If the CPU_MEM signal is active, the server does not allow any GPU operation. In this case, the server returns the remaining time of the memory phase, after which the GPU thread will contact the server again. If the host is instead in CPU_COM state, the server computes the remaining time of the computation phase, and allows the GPU thread to execute for a corresponding amount of time.

A graphical description of the protocol is depicted in figure 3, showing a time window with concurrently executing CPU and GPU tasks. At time t_0, the CPU thread acquires the lock for system DRAM by sending the CPU_MEM signal and the worst-case duration M_L of the memory loading phase. At time t_1, the GPU thread wants to execute a GPU task that might involve the copy or the execution engine, indicated simply with "GPU activity". The GPU thread queries the server, which replies with the amount of time the GPU thread will have to go in sleep mode before querying the server again. This happens at time t_2, when the CPU goes into CPU_COM and notifies the server about the length of this phase. This phase corresponds to the "grace period" in which the GPU thread can operate any kind of activity (from t_2 to t_3). How this activity is managed to be bounded within this grace period depends on which engine is going to be used by the GPU, as will be detailed in the next paragraph. The time interval between t_3 and t_4 is locked by the CPU thread for a consecutive load/unload memory phase, followed by the second grace period for the GPU until t_5. The lock is completely lifted after t_6, when the CPU thread communicates the NO_OPER signal, allowing unrestricted execution of the GPU task.

The cost of server communications will be experimentally evaluated in section 5, while further remarks on the impact of this server mechanism will be provided in section 6.

4.2 Split CUDA memory transfers

The above paragraph assumes GPU threads to well behave, i.e., not accessing shared memory resources while the server communicates a CPU memory activity. However, this is often not the case when addressing general-purpose GPU kernels. In case no guarantee can be given for a Copy Engine transfer or a CUDA kernel computation to complete within a grace period, we hereafter present an efficient mechanism to preempt their activities before interfering with critical CPU tasks.

For DMA-based activities, it is very easy to intercept any CUDA runtime calls to *cudaMemcpy* related functions, to split transfer commands according to a pre-determined granularity. This technique increases the overhead of copy tasks, establishing more DMA transfers and related host synchronizations. However, such an overhead can be easily estimated and optimized according to the desired transfer granularity, as shown in [14]. In our setting, we observed that a transfer granularity of 1 MB is sufficient to saturate the GPU Copy Engine bandwidth. A similar conclusion was reached in [4] for a similar scenario. The amount of time needed to transfer 1 MB is, therefore, the constant used to determine how many bytes can be transferred using the copy engine while the CPU thread is in a compute phase, as given by the following formula:

$$Q = \frac{t(C) - t(GPU_{req})}{T(Q_{CE})}, \tag{1}$$

where Q is the size (in MB) of the Copy Engine transfer allowed in the grace period; $t(C)$ is the end of the ongoing CPU_COM phase, as communicated by the CPU to the server; $t(GPU_{req})$ is the time at which the GPU thread contacted the server to start its activity; and $T(Q_{CE})$ is the time needed to complete a single 1 MB DMA transfer.

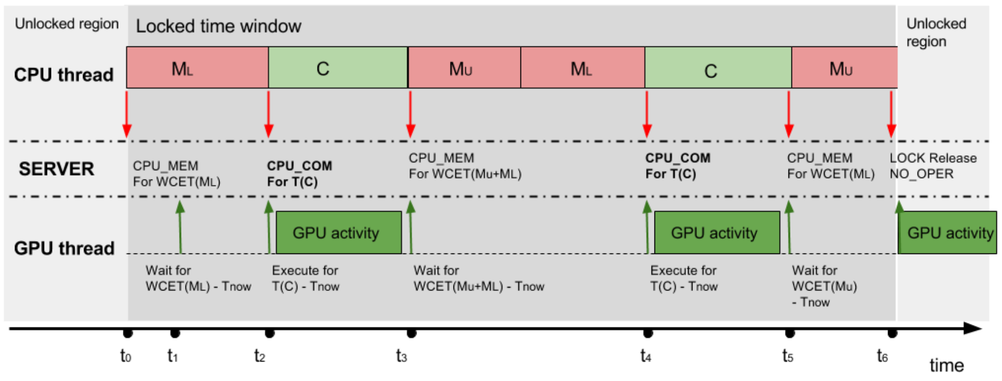


Figure 3: Timeline of an example sequence of interactions between a real-time CPU thread and a best effort GPU task

4.3 Kernel preemption

Preempting a CUDA kernel to split its execution between different CPU_COM phases is not a trivial task. However, we found a convenient solution for allowing kernel preemption, by exploiting the concepts of single CUDA context and stream priorities. The traditional model of CUDA applications involves the creation of as many CUDA contexts as the needed applications. While the driver has no problem handling these situations, this might lead to additional overhead. Hence, an alternative model in which a single CUDA model acts as a proxy to be shared among different applications represents a more suitable solution for reducing driver overheads, facilitating inter-process communication[3]. More than trying to reduce driver overhead or to exploit more efficient ways for process communication, having a single context gives us the ability to exploit priority streams in CUDA, as we can assign a different stream for each application. A CUDA stream is an abstraction of a queue of commands directed to the GPU: ideally, such commands are to be executed in parallel. However, such parallel execution can only take place if the commands in execution from each queue have reduced requirements in terms of GPU resources[4], i.e., registers, shared memory, block number and size. In most cases, however, kernels get executed concurrently rather than in parallel. This is the case especially for iGPUs, where the number of SMs is limited. Such concurrency occurs in a time sharing fashion, with the programmer having absolutely no control over the length of the time quanta assigned to each kernel. Starting from CUDA SDK version 7.5, the NVIDIA TX1 board can now create streams with an attached priority value. Only two discrete priority levels are available to the programmer: HIGH and LOW. A simple experiment is able to show that creating two streams, each with a different priority level, allows the kernel inserted in the higher priority stream to preempt a previously launched kernel inserted in the lower priority stream. Moreover, the high priority kernel runs to completion, hence it does not share any time slice with the lower priority kernel. This mechanism can be exploited for creating a *Spin Kernel* able to saturate the GPU occupancy level, hence preempting any other currently running kernel. The Spin Kernel does not make any memory access and can be tuned to last for an arbitrarily large duration.

[3]A detailed implementation is provided by CUDA MPS: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf
[4]These requirements are also known in CUDA terms as kernel occupancy

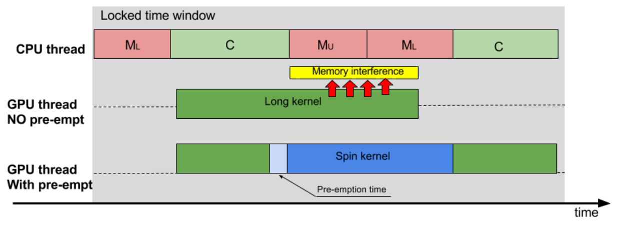
Figure 4a shows a Spin Kernel preempting a GPU thread that does not complete before the end of a CPU_COM phase, preventing the preempted kernel from accessing system DRAM during CPU_MEM phases. The server sets the duration of the Spin Kernel in order to match the length of the CPU memory phase. The proposed mechanism has some similarity with the MEMGUARD mechanism [25] mentioned in section 2 for throttling CPU-side memory accesses.

In order to spin the GPU, the server has to know the worst case preemption time, i.e., how long does it take for the Spin Kernel to evict a previously running kernel from the streaming multiprocessors. For measuring this timing window, we assign the Spin Kernel to the high priority stream, and a copy kernel, as the one described in section 3, to the lower priority stream. Both kernels have a known duration measured without interference. The lower priority kernel is executed first, waiting 100 μs before submitting the Spin Kernel. The result of the experiment is shown in figure 4b, where the nvprof output of kernel preemption has been analysed with NVIDIA NSIGHT visual profiler.

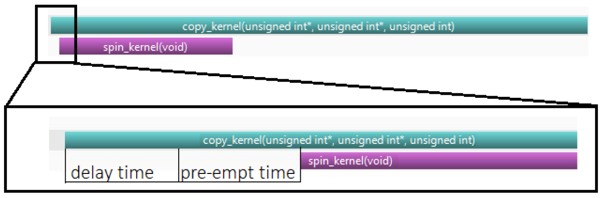
By subtracting our defined 100μs delay time from the total time between the start of the copy kernel and the start of the Spin Kernel, we obtain the actual preemption time. Out of the observed iterations, this time stabilizes between 120 and 170 μs. The variation depends on the number of stalling points in the lower priority kernel, i.e., instructions that cause a warp de-scheduling, such as memory accesses and barriers. A kernel with a reduced number of stalling points typically has a higher pre-emption time. Interestingly, it also means that the kernel is mostly computing without accessing memory, so that a larger preemption time does not constitute a significant threat to CPU memory phases.

5 EXPERIMENTS AND RESULTS

In compliance with the model described in the previous section, we consider a CPU real time thread pinned to one core. The CPU thread behavior is constant throughout its execution and for the different experiment iterations. Instead, different configurations are considered for the GPU task. Fixing the CPU thread varying the GPU computation does not imply a loss of generality in our



(a)



(b)

Figure 4: On the left side, a Spin Kernel preemption over a long lasting GPU activity. On the right side, nvprof output of kernel preemption analysed with NVIDIA NSIGHT visual profiler.

Table 2: Timings of a single PREM-ized iteration on the tested board. Times in μs

TK1			*TX1*			
$T(M_L)$	$T(C)$	$T(M_U)$	$T(M_L)$	$T(C)$	$T(M_U)$	
596	8704	1008	426	8278	687	**MIN**
722	8849	1223	642	9188	914	**MAX**
645	8761	1076	444	8784	716	**AVG**

conclusion. As will be later detailed, a key parameter for assessing *Si*Γperformance is given by the ratio between the CUDA kernel execution time and the time to completion of the CPU compute phase. In the loading phase, the CPU thread reads an amount of data equal to 85% of the LLC size. The remaining 15% of the LLC is reserved to server interactions and GPU driver activities.

The computation phase is a simple RGB-to-Y conversion operated on pre-fetched integer data. During the memory loading phase, previously computed data is flushed into system DRAM and then copied to an output buffer. The total size of data to be processed by the CPU thread amounts to 256 MB, corresponding to 145 iterations of PREM-ized phases for a single experiment. Table 2 shows the observed execution times for these phases in both tested boards with no interference.

The GPU thread is pinned to the remaining cores of the SoC. We define a single GPU iteration as a H2D copy of 50MB for the TK1, 100MB for the TX1, followed by a compute kernel of parametrized duration that alternates memory fetching instructions to purely compute instructions. The compute phase operates on variable subsets of the previously sent data. The size of this subset is always bigger than GPU L2 cache, so to be sure that the kernel phase will access system DRAM during its execution. Finally, a D2H copy phase is performed, having the same size as the H2D transfer. $T(Q_{CE})$ is estimated to be 122μs for TX1 and 250μs for TK1. We run the GPU task for 100 iterations per experiment. As opposed to the CPU thread (scheduled with FIFO 99 priority), the GPU thread is scheduled with CFS (linux default non real-time scheduler).
Each measure is repeated and analyzed for 1350 runs.

5.1 Tegra K1

Figure 5 reports the experimental results for the TK1 board, showing the measured latencies of each CPU phase with no interference (alone), un-arbitrated GPU interference (CPU+GPU interf), and with our memory arbitration mechanism (CPU+GPU arb). The kernel duration is initially set to 37% of $T(C)$, i.e., the minimum observed execution time of the CPU_COM phase with no interference.

The dramatic impact of GPU interference can be clearly seen on both memory phases, with a CPU-side memory latency that can be three times higher that in the non-interfered case, consistently with the latency impact measured in section 3. Our server mediation approach manages to keep the latency really close to the non-interfered case, with a negligible performance deterioration. Even if with a lesser extent, the CPU computing phase is also somewhat affected by the GPU activity, both with and without server arbitration. This is due to CUDA callback threads and other driver activities. How to mitigate such effects is currently under investigation.

The above results have been obtained only by splitting CE related memory transfers. Due to the CUDA SDK version installable on the TK1, it is impossible to exploit the kernel preemption mechanisms detailed in section 4.3. This implies allowing the GPU to perform a CUDA kernel invocation only if its kernel duration is smaller than $T(C)$. If this is not the case, the GPU thread may experience starvation. In order to estimate the impact of the server arbitration policies w.r.t. the GPU thread, we parametrize the GPU kernel duration as a function of $T(C)$, to then measure how many GPU task iterations are performed within the locked time window, i.e., the server-arbitrated time frame until the CPU thread sends a NO_OPER signal, as depicted in figure 3. We denote as *GCratio* the ratio between the worst case duration of a CUDA kernel and $T(C)$. Figure 6 shows the number of GPU iterations performed during the locked time window when varying *GCratio*.

A slow decrement of GPU iterations is visible increasing the *GCratio*, until reaching a starvation point, after which no GPU iteration is completed until the CPU thread becomes idle. In the TK1 case, starvation can only be avoided by splitting kernel executions into smaller phases, or recurring to substantial code modifications, as discussed in section 2, incurring additional overhead.

5.2 Tegra X1

The CUDA SDK version for the TX1 supports two discrete stream priorities, allowing us to test our Spin Kernel solution. GPU kernel duration is initially set to 34% of $T(C)$ for the case with no kernel preemption, and 140% of $T(C)$ with kernel preemption.

Figure 7 shows the results, which are fairly similar to the TK1 case. Once again we highlight how our server mediation strategy

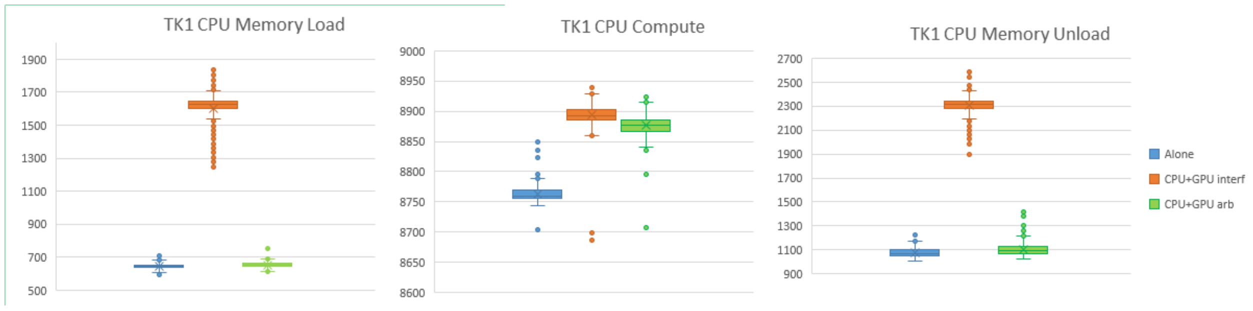


Figure 5: Results obtained on NVIDIA TK1. Times in μs.

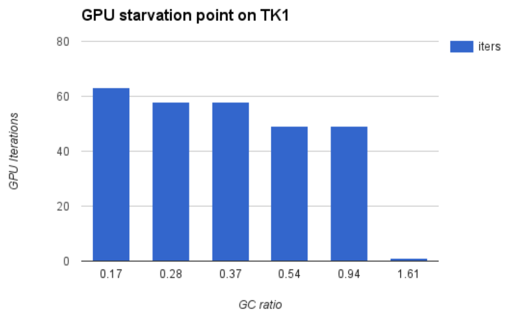


Figure 6: Starvation point on NVIDIA TK1. No kernel preemption strategies involved.

manages to drastically reduce the interference of the GPU on CPU memory phases, both in the worst-case and in the average-case. Our proposed kernel preemption mechanism (CPU+GPU pre) causes a limited performance loss due to the added driver overhead for invoking more kernels (the Spin Kernel has to be explicitly called). Still, it shows a significant improvement w.r.t. the non-arbitrated case.

To profile GPU starvation, we performed the same experiment described in the TK1 case, with and without kernel preemption. Results are shown in figure 8. Without kernel preemption, the same starvation point observed in the TK1 is reached when *GCratio* exceeds 1. With kernel preemption, a dramatic improvement in GPU efficiency is showed, allowing the execution of 34 GPU iterations during the locked time window. Experiments on both boards demonstrated the efficacy of our server mediation strategy for predictably bounding the interference experienced by CPU tasks due to GPU activities. However, depending on the size of GPU tasks and the availability of prioritized streams, the impact on GPU activities may significantly vary. The following section more formally analyzes these effects.

6 LIMITATIONS AND FURTHER ANALYSES

We have shown that our approach can safely protect CPU memory phases and guarantee predictability and freedom from interference due to GPU tasks. We hereafter discuss how to extend the bandwidth server mediation approach to multi-core real time task sets, and how to evaluate the loss of efficiency at the GPU side.

6.1 Extension to CPU multi-core

Our server approach can be extended to multi-core scenarios. The dynamic nature of *Si*Γ allows handling multi-core scenarios with minimal code modification. As an example, consider a setting with four cores, with a corresponding number of PREM tasks, as shown in figure 9.

We assume PREM-ized task in which the memory loading phase involves core private memory hierarchy. In the case of the considered boards, this corresponds to either using L1 data caches, or partitioning the L2 cache among the various cores. The GPU activity (either CE or EE), has to take place in the time window delimited by the end of the last CPU memory loading phase to the start of the first memory unloading phase. This restricts the applicability of our approach within such a smaller time window. If the time window is too small for a single GPU memory transfer or kernel preemption time, the GPU task might experience a long starvation. By properly dimensioning PREM phases, this latter scenario can be conveniently avoided.

It is now easy to realize that the case with multiple real-time tasks concurrently scheduled onto one or more cores can be easily treated with our *Si*Γ mechanism, by adapting the size of the memory/computation phase according to the requirements of the executing real-time task. The dynamically varying sizes of the server phases allow *Si*Γ to cope with the different requests of the tasks concurrently running on the host cluster. The delay imposed to GPU activities directly depends on the memory utilization of the CPU tasks, as well as on the granularity of their memory accesses.

6.2 Cost of server communication

In our experiments, the server was basically acting as a memory location accessible by all involved components. This was sufficient for this prototyping phase. However, we plan to integrate *Si*Γ as a kernel module on a RTOS or on a hypervisor.

The delay due to contacting the server is equal to the latency of accessing a variable located in system DRAM. Such delay can be easily inferred from the experiment of section 3. When implementing *Si*Γ as a kernel module, a system call has to be performed to access the server. System call overheads depend on the adopted OS and tested board features. TX1 with standard L4T O.S. has an estimated system call overhead of about 260 ns (calculated as round trip time). Hypercall overhead depends on the adopted hypervisor. Recently, support for TX1 on the Jailhouse hypervisor [21] has been

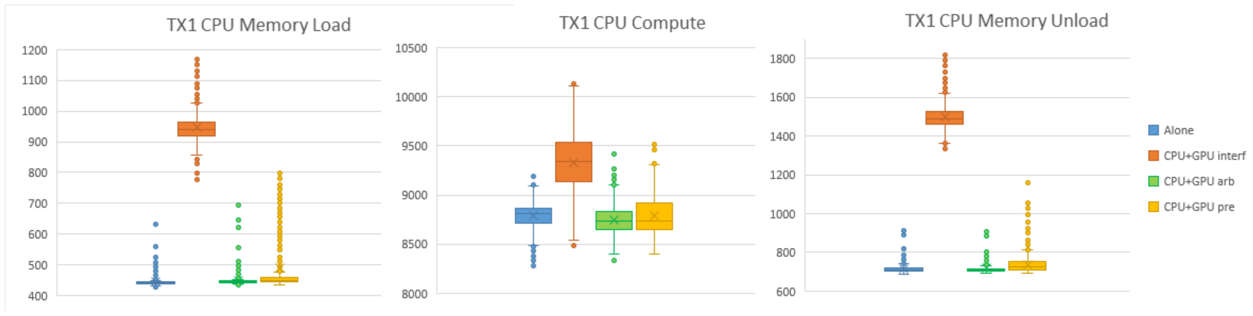


Figure 7: Results obtained on NVIDIA TX1. Times in μs.

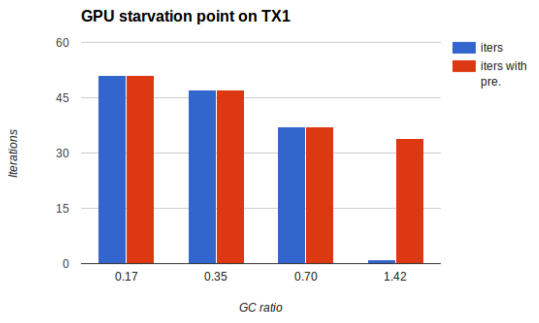


Figure 8: Starvation point on NVIDIA TX1. Blue: no kernel pre-emption, Red: kernel pre-emption enabled.

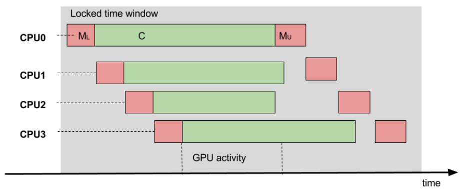


Figure 9: Example scenario with a multi-core real-time task set. A single PREM iteration per core is shown.

made available. A hypercall to a bare metal cell (implementing the *Si*Γ server as a Jailhouse lightweight proxy cell) is measured in a range between half a μs to 2 μs.

In section 6.3, this cost, referred to as ϵ, is considered to evaluate the potential starvation of GPU applications induced by *Si*Γ. This cost can be also used to infer an efficiency loss measure on the CPU side, considering that the CPU task has to pay an overhead of ϵ before starting a memory or a computation phase.

6.3 GPU efficiency loss analysis

In our model, GPU activities within a single GPU iteration can be divided into Memory phases M_G and Kernel phases K. Memory phases are memory transfers managed by the Copy Engine, either H2D or D2H. Kernels are computing phases scheduled in the GPU execution engine. We would like to find out how much each phase is delayed (worst case), according to its estimated duration ($T(M_G)$ or $T(K)$).

Being ϵ the server request overhead, each memory transfer M may finish its execution by

$$\begin{aligned} End(M_G) &= (\epsilon_M + T(M_U) + T(M_L)) \\ &+ \left\lfloor \frac{T(M_G)}{T(C)} \right\rfloor \cdot (3\epsilon + T(M_U) + T(M_L) + T(C)) \\ &+ \epsilon + T(M_G) \bmod T(C) \end{aligned} \tag{2}$$

where $\epsilon_M < \epsilon + T(Q_{CE})$.

For the kernel phases K we have to consider different cases:

- (A) $T(K) < T(C)$
- (B) $T(K) > T(C)$, with K pre-emptable
- (C) $T(K) > T(C)$, with K non pre-emptable

In the first and second case (A and B), it is always possible to execute the kernel, while different considerations have to be made for the third case (C). W.r.t kernel K, in the first case (A) we have:

$$End(K) = (\epsilon_K + T(M_U) + T(M_L)) + T(K) \tag{3}$$

where $\epsilon_K < \epsilon + T(K)$ represents the condition boundary for wich preemption is possible.

In the second case (B) we obtain an equation similar to the one inferred for the memory phase:

$$\begin{aligned} End(K) &= (\epsilon_P + T(M_U) + T(M_L)) + \\ &\left\lfloor \frac{T(K)}{T(C_P)} \right\rfloor \cdot (3\epsilon + T(M_U^{worst}) + T(M_L^{worst}) + T(C_P)) \\ &+ \epsilon + T(K) \bmod T(C_P) \end{aligned} \tag{4}$$

where $T(M_U^{worst})$ and $T(M_L^{worst})$ are the measured worst case execution time for M_U and M_L, $T(C_P) = [T(M_L) + T(C) + T(M_U)] - [T(M_U^{worst}) + T(M_L^{worst})]$ and $\epsilon_P < \epsilon + \textit{pre-empt time.}$

The term $T(C_P)$ of the equation 4 takes into account that the Spin Kernel used to pre-empt the GPU compute phase is sized according to M_U and M_L worst execution times.

In this case, in order to avoid GPU task starvation, the necessary condition is $T(C_P) > 0$.

For the last case (C) a non pre-emptable kernel phase can not execute during the locked time window without interfering with CPU memory phases; therefore the kernel phase must wait until the CPU thread releases the memory lock. This is the case in which starvation times are noticeably prolonged (see experiments in section 5.1).

7 DISCUSSION AND FUTURE WORK

After measuring the dramatic impact of unregulated CPU and iGPU parallel access to central memory in embedded devices, we presented a novel arbitration mechanism called *Si*Γ. *Si*Γ is able to efficiently orchestrate CPU and GPU memory accesses. By means of a mediation effect enabled by a memory arbitration server, the memory contention due to GPU engines is drastically reduced, allowing host-side real-time tasks to access memory with little to no performance loss, even when executing in parallel with GPU activities. In one of the two boards used in our experiments, we were able to demonstrate an improved mechanism to preempt execution kernels running on the GPU, allowing a reduction of the starvation of GPU tasks. This was enabled by exploiting CUDA stream priorities and related CTA level preemption, introducing a GPU implementation of memory bandwidth regulation mechanisms that have been previously introduced only at CPU-side (i.e., MEMGUARD). We also showed how the proposed approach can be easily extended to PREM tasks concurrently executing on a multicore host, leaving a more detailed schedulability analysis for this setting as a future work. We are also working on extending *Si*Γ to other GP-GPU API such as OpenCL, in order to be able to port it to architectures other than NVIDIA's: this port is relatively easy to achieve as every CUDA function call and features exploited for developing *Si*Γ have an OpenCL counterpart that work in a very similar way.

The promising results presented with our approach also call for further refinements in the server behavior. A promising research direction is related to more fine grained bandwidth arbitration between CPU and GPU. While the presented approach gives full priority to the CPU, the server can be easily modified to selectively allow GPU activities to overlap with CPU memory phase. By controlling such an overlap, less constraints may be imposed to both clients when accessing memory, while still retaining real-time guarantees. The final objective of our research is to implement system-wide server-based arbitration policies (possibly at hypervisor level) that are able to orchestrate CPU and GPU real time tasks with said methodology. A promising instrument to do so may be applying the PREM methodology also to CUDA kernels. This is achievable with CUDA warp specialization [5] as shown in preliminary experiments [11].

ACKNOWLEDGMENT

The work has been supported by the HERCULES Project, funded by European Union's Horizon 2020 research and innovation program under grant agreement No. 688860.

REFERENCES

[1] 2014. NVIDIA TK1 Development kit. http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html. (2014).

[2] 2015. NVIDIA TX1 Development kit. http://www.nvidia.com/object/jetson-tx1-dev-kit.html. (2015).

[3] Stanley Bak, Gang Yao, Rodolfo Pellizzoni, and Marco Caccamo. 2012. Memory-aware scheduling of multicore task sets for real-time systems. In *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 300–309.

[4] Can Basaran and Kyoung-Don Kang. 2012. Supporting preemptive task executions and memory copies in gpgpus. In *2012 24th Euromicro Conference on Real-Time Systems*. IEEE, 287–296.

[5] Michael Bauer, Henry Cook, and Brucek Khailany. 2011. CudaDMA: optimizing GPU memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 12.

[6] Jens Breitbart. 2010. Static GPU threads and an improved scan algorithm. In *European Conference on Parallel Processing*. Springer, 373–380.

[7] Nicola Capodieci and Paolo Burgio. 2015. 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, 6–12.

[8] Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. 2017. Memory Interference Characterization between CPU cores and integrated GPUs in Mixed-Criticality Platforms. In *22nd IEEE International Conference on Emerging Technologies And Factory Automation (ETFA 2017)*. IEEE, to appear.

[9] G. Durrieu, M. FaugÃl're, S. Girbal, D. Gracia PÃl'rez, C. Pagetti, and W. Puffitsch. 2014. Predictable Flight Management System implementation on a Multicore processor. In *Embedded Real Time Software (ERTS'14)*. TOULOUSE, France.

[10] Glenn A Elliott, Bryan C Ward, and James H Anderson. 2013. GPUSync: A framework for real-time GPU management. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*. IEEE, 33–44.

[11] Björn Forsberg, Andrea Marongiu, and Luca Benini. 2017. GPUguard: Towards supporting a predictable execution model for heterogeneous SoC. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 318–321.

[12] Kshitij Gupta, Jeff A Stuart, and John D Owens. 2012. A study of persistent threads style GPU programming for GPGPU workloads. In *Innovative Parallel Computing (InPar), 2012*. IEEE, 1–14.

[13] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, Falk Wurst, and Dirk Ziegenbein. 2017. WATERS Industrial Challenge 2017. http://ecrts.eit.uni-kl.de/forum/download/file.php?id=60&sid=fdaf5540bb967f3f923d96bc74ea2f7a. (2017).

[14] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan Rajkumar. 2011. RGEM: A responsive GPGPU execution model for runtime engines. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*. IEEE, 57–66.

[15] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. 2011. TimeGraph: GPU scheduling for real-time multi-tasking environments.

[16] Jose M Nadal-Serrano and Marisa Lopez-Vallejo. 2016. A Performance Study of CUDA UVM versus Manual Optimizations in a Real-World Setup: Application to a Monte Carlo Wave-Particle Event-Based Interaction Model. *IEEE Transactions on Parallel and Distributed Systems* 27, 6 (2016), 1579–1588.

[17] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. 2011. A predictable execution model for COTS-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 269–279.

[18] Nikola Rajovic, Alejandro Rico, James Vipond, Isaac Gelado, Nikola Puzovic, and Alex Ramirez. 2013. Experiences with mobile processors for energy efficient HPC. In *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 464–468.

[19] Amit Rao, Ashish Srivastava, KINI Yogesh, Alban Douillet, Geoffrey Gerfin, Mayank Kaushik, Nikita Shulga, Vyas Venkataraman, David Fontaine, Mark Hairgrove, et al. 2015. Unified memory systems and methods. (Jan. 20 2015). US Patent App. 14/601,223.

[20] Stephan Schnitzer, Simon Gansel, Frank Dỳrr, and Kurt Rothermel. 2016. Real-time scheduling for 3D GPU rendering. In *Industrial Embedded Systems (SIES), 2016 11th IEEE Symposium on*. IEEE, 1–10.

[21] Valentine Sinitsyn. 2015. Jailhouse. *Linux Journal* 2015, 252 (2015), 2.

[22] V. M. Weaver, D. Terpstra, and S. Moore. 2013. Non-determinism and overcount on modern hardware performance counter implementations. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 215–224. https://doi.org/10.1109/ISPASS.2013.6557172

[23] Shucai Xiao and Wu-chun Feng. 2010. Inter-block GPU communication via fast barrier synchronization. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 1–12.

[24] Heechul Yun, Santosh Gondi, and Siddhartha Biswas. 2015. Protecting memory-performance critical sections in soft real-time applications. *arXiv preprint arXiv:1502.02287* (2015).

[25] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. 2013. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 55–64.

[26] Jianlong Zhong and Bingsheng He. 2014. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1522–1532.