Work-in-Progress: NVIDIA GPU Scheduling details in Virtualized environments

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

ABSTRACT

Modern automotive grade embedded platforms feature high performance Graphics Processing Units (GPUs) to support the massively parallel processing power needed for next-generation autonomous driving applications. Hence, a GPU scheduling approach with strong Real-Time guarantees is needed. While previous research efforts focused on reverse engineering the GPU ecosystem in order to understand and control GPU scheduling on NVIDIA platforms, we provide an in depth explanation of the NVIDIA standard approach to GPU application scheduling on a Drive PX platform. Then, we discuss how a privileged scheduling server can be used to enforce arbitrary scheduling policies in a virtualized environment.

1 INTRODUCTION

Advanced Driver-Assistance Systems (ADAS) often feature an integrated GPU as a massively parallel programmable processor that has to be shared across a potentially large variety of applications, each having different timing requirements. We disclose and discuss the current NVIDIA approach to GPU scheduling for both graphic and compute applications on the Drive PX-2 “AutoCruise’ platform. The board features a single Tegra Parker SoC, which is composed by an exa-core CPU complex (a four-core ARM Cortex A-57 cluster, and a dual-core ARM-v8 compatible NVIDIA Denver cluster) and an integrated GPU. The GPU (gp10b) is an integrated version of the newly released Pascal Architecture, commonly featured in both consumer-level and HPC-level graphics cards, characterized by two Streaming Multiprocessors (SMs), each featuring 128 CUDA cores. Note that not all in-depth technical details can be revealed due to NDA restrictions. Still, we did an extensive effort to provide information on previously undisclosed technical details, whereas previous research contributions mostly involved reverse engineering the architecture [3], due to the closed-source nature of the NVIDIA software ecosystem [6]. Moreover, we describe how to enforce arbitrary scheduling policies at hypervisor level, as NVIDIA Pascal architecture allows for graphic shader/compute kernel preemption at pixel/thread granularity.

2 GPU SCHEDULING

The NVIDIA GPU scheduler features a hardware controller embedded in the GPU within a component called “Host"". 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. Channels are transparent to a user-space programmer, which specifies GPU workloads through API (CUDA, OpenGL, etc.) function calls. The workload consists of a sequence of GPU commands that are inserted in a Command Push Buffer, which is a memory region written by the CPU and read by the GPU. Channels are therefore related to an application’s Command Push buffer. A GPU application maps itself to one or more channels. Each channel is characterized by a 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*. The runlist is a list of established channels that may or may not have pending work to execute. The GPU Host implements a list-based scheduling policy that snoops each channel for work by browsing the runlist. Each application 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 (distributed with L4T, Linux For Tegra). [1]. 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 three high priority applications and one best effort (medium or low priority), each consisting of one channel.

Timeslice length, interleaving level and allowed preemption policy are the scheduling parameters that can be tuned by a user. The timeslice is the execution time assigned to a channel before being preempted. The interleaving level refers to the number of occurrences of a particular channel within a runlist. The rationale for allowing a channel to be replicated more than once in a runlist is to have higher priority channels checked for work more often than lower priority ones, allowing critical applications to be more resilient towards CPU-side delays when submitting commands, as the GPU scheduler polls more often higher priority applications

Conference’18, -
© 2019 ACM. 978-x-xxxx-xxxx-x/YY/MM...$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

[1] 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

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. Channels are established at application launch. In the NVIDIA runlist approach, the Host scheduler allows only one application to be resident within the GPU engines at a given time, and 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. 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. In our notation, a GPU task τ_i is characterized as having a requested GPU execution time C_i, with D_i being its relative deadline, and P_i is the period or minimum inter-arrival time between two job submissions. This model fits advanced automotive applications where critical jobs (both graphic and compute) such as pedestrian detection and speedometer rendering follow a recurring pattern. The computing platform 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. The execution time C_i may match the inference time for a DNN, or any other combination of CUDA kernel invocations, or the actual rendering time of the draw calls needed for displaying a graphic application. Such a scheduler is efficient for 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. Moreover, parameters' estimation (interleaving level and timeslice length) are difficult to optimize for complex task sets. It is trivial to prove that 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.

3 FUTURE WORK ON VIRTUALIZATION

NVIDIA GPU virtualization technology allows multiple guests to run and access the GPU engines. This is accomplished through a privileged hypervisor guest called RunList Manager, or RLM. The other guests wishing to access the GPU have to contact the RLM server through the inter-VM communication infrastructure of the hypervisor for operations such as channel allocations, scheduling parameters setting, memory management operations and obviously runlist construction. We are in the process of modifying the underlying GPU to RLM communication infrastructure so to have the RLM being able to intercept command submissions, to then define arbitrary SW scheduling policies and enforce them by simply constructing runlist having only the channels related to the applications we wish to schedule. By doing this, we are able to test and validate event-based approaches, that are known to provide stronger real-time guarantees compared to table driven approaches as the NVIDIA baseline interleaved scheduler. More specifically, preliminary results of a prototype Earliest Deadline First, augmented with a Constant Bandwidth Server (EDF+CBS), show a significant improvement in schedulability ratio over high utilization tasksets. This lead to no deadline misses and significant improvements over the Worst Case Response Time (WCRT) in our on board experiments (see Table 1). Changes in the GPU-RLM communication infrastructure means implementing signals at each scheduling event related to an application: new work has been submitted, previous work has been consumed by GPU engines and the event of server budget expiration (A, B and C respectively in figure 2). We are also investigating methodologies to mitigate memory interference between CPU and GPU in embedded SoCs [1, 2, 4, 5].

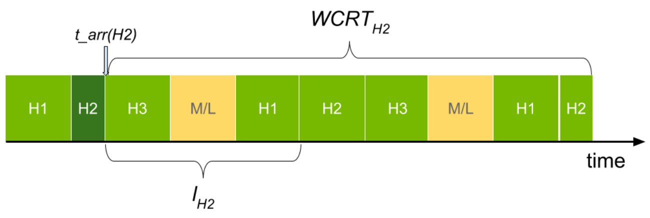

Figure 1: Worst-case scenario for a GPU task $H2$. The darker green $H2$ slot is shorter since the GPU Host had no work to dispatch.

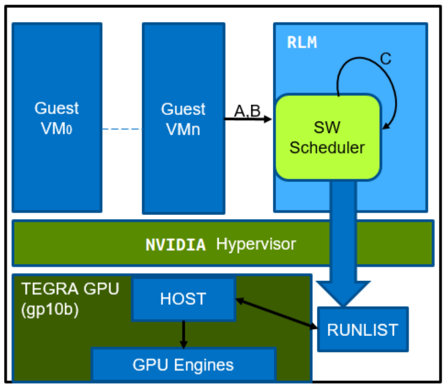

Figure 2: The main blocks of the prototype scheduler. A, B and C are event signals triggered for scheduling decisions.

Table 1

Task class	WCRT decrease [Interleaved vs. EDF %]	Deadline miss [Interleaved % | EDF %]
RT CUDA DNN Inference	64.56%	0.55% | 0%
RT Graphic app. (30 FPS)	93.35%	4.02% | 0%
Best Effort Graphic app. (60 FPS)	17.70%	-
Best Effort Graphic app. (unbounded)	(increase) 158.84%	-

ACKNOWLEDGMENT

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

REFERENCES

[1] Nicola Capodieci, Roberto Cavicchioli, Paolo Valente, and Marko Bertogna. 2017. 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)*. ACM, New York, NY, USA, 48–57.

[2] R. Cavicchioli, N. Capodieci, and M. Bertogna. 2017. 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)*. 1–10.

[3] Glenn A Elliott. 2015. *Real-time scheduling for GPUS with applications in advanced automotive systems.* Ph.D. Dissertation. The University of North Carolina at Chapel Hill.

[4] 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.

[5] Přemysl Houdek, Michal Sojka, and Zdeněk Hanzálek. 2017. Towards predictable execution model on ARM-based heterogeneous platforms. In *Industrial Electronics (ISIE), 2017 IEEE 26th International Symposium on*. IEEE, 1297–1302.

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