GPU Sharing for Image Processing in Embedded Real-Time Systems*

Nathan Otterness[1], Vance Miller[1], Ming Yang[1], James H. Anderson[1], F. Donelson Smith[1], and Shige Wang[2]
[1]Department of Computer Science, University of North Carolina at Chapel Hill
[2]General Motors Research

Abstract

To more efficiently utilize graphics processing units (GPUs) when supporting real-time workloads, it may be beneficial to allow multiple tasks to issue GPU computations without blocking one another. For such an option to be viable, it is necessary to know the extent to which concurrent GPU computations interfere with each other when accessing hardware resources. In this paper, measurement data is presented regarding such interference for several image processing routines motivated by automotive use cases. These measurements were taken on NVIDIA Jetson TK1 and TX1 boards. The presented data suggests that currently available real-time GPU management frameworks should evolve to enable the option of co-scheduling GPU computations.

1 Introduction

Vision-based sensing through cameras is being widely used in automobiles today to support advanced driver assistance systems (ADASs). Common capabilities of current ADASs include forward collision detection with automatic braking, lane departure warnings, and adaptive cruise control. Envisioned capabilities include advanced obstacle-tracking features, sign recognition, and 360-degree sensing.

Such capabilities give rise to workloads that can be challenging to support for three reasons. First, individual tasks may be subject to real-time constraints. Second, such tasks may be computationally intensive. Third, the overall workload must be supported on a hardware platform that operates within an acceptable size, weight, and power (SWaP) envelope and also is not too expensive.[1] In light of these needs, multicore+GPU platforms have been suggested as a promising way forward. Such a platform consists of several general-purpose CPUs augmented with one or more graphics processing units (GPUs) that can accelerate computations typically required in automotive settings.

Prior foundational work: GPUSync. Unfortunately, efficiently utilizing GPUs in contexts where real-time constraints exist requires sifting through many tradeoffs involving how GPUs are allocated at runtime and how GPU computations and related overheads are analyzed when checking real-time schedulability. To enable such tradeoffs to be systematically studied, our research group developed a real-time GPU allocation framework called GPUSync [15]. In GPUSync, the management of GPU-related hardware resources is viewed as a *synchronization* problem and thus real-time multiprocessor locking protocols are used to acquire and release such resources. GPUSync is highly configurable: options exist to control how tasks are scheduled on CPUs, how data is copied to and from GPUs, how GPU-related computations are queued and prioritized, *etc.*

Beyond GPUSync. In recent work, we have been attempting to evolve our work on GPUSync to more directly meet the needs of automotive use cases. The consideration of such use cases has caused the nature of our work to change in two significant ways. First, GPUSync is implemented primarily in LITMUS$^{\mathrm{RT}}$, and the code base is large, approximately 15,000 lines. Automotive manufacturers would likely be highly resistant to allowing such extensive operating system (OS) modifications. Due to this, we have shifted our attention to a simplified variant of GPUSync called GPUSyncLite that implements only a few GPUSync configurations (one currently) and requires only minimal OS modifications (none currently). Second, our prior GPUSync-related experimental work was conducted on an Intel platform that provides 12 CPU cores augmented with eight high-end GPUs. At present, it is hard to imagine such an expensive, energy-hungry platform being used in a production automobile. As a result, we have shifted our attention to less-expensive ARM-based platforms that provide a single less-costly, less-capable GPU.

Efficient GPU utilization through co-scheduling. This shift in hardware platform has created a new dilemma: when using a single, less-capable GPU, any waste of the GPU's capacity becomes untenable. Unfortunately, when using most previously proposed real-time GPU management frameworks [7, 8, 9, 12, 16, 25, 26, 49, 47, 48, 54, 55], including GPUSync, such under-utilization may be common. In particular, these frameworks disallow concurrent GPU execution by different tasks, so a task that under-utilizes the GPU's hardware resources can waste much of its capacity. Other prior work [10, 11] has considered co-scheduling GPU workloads, but in this work, several simplifying assumptions are made that preclude applicability on real-world GPUs. Notably, GPU instructions are assumed to always require only

*Work supported by NSF grants CPS 1239135, CNS 1409175, and CPS 1446631, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and funding from General Motors.

[1]In contrast to various "one-off" implementations of autonomous or semi-autonomous features, as seen for example in the Google car [1] and various DARPA challenge vehicles [45], affordability is a serious limitation with respect to production automobiles.

a single clock cycle, and cache misses and memory latency are not considered. Furthermore, this prior work includes no evaluation using real hardware.

To combat GPU under-utilization, we are beginning to investigate a new variant of GPUSyncLite that allows GPU computations issued by different tasks to be concurrently co-scheduled. When considering multi-threaded workloads scheduled on conventional multicore platforms, Jain *et al.* [22] observed that some co-scheduling choices are constructive and some are destructive. This is true in our context as well. In particular, it is *constructive* to co-schedule GPU computations issued by different tasks if the resulting GPU execution times and blocking times (*i.e.*, times spent waiting to access a GPU) yield real-time schedulability improvements. In contrast, such co-scheduling is clearly *destructive* if it causes a large inflation in GPU execution times or blocking times. Any such inflation is a sign that the co-scheduled GPU computations are adversely interfering with each other with respect to the hardware resources they access.

Contributions of this paper. To get a sense of the nature of such interference, we conducted experiments involving several common image-processing routines motivated by automotive use cases. For each of the considered routines, we obtained execution-time data via a measurement process under various co-scheduling scenarios. These measurements were taken on NVIDIA Jetson TK1 and TX1 boards. The obtained data suggests that certain co-scheduling choices are indeed constructive, while others are clearly destructive. The main contribution of this paper lies in presenting this data and discussing its implications as far as the future evolution of real-time GPU management frameworks is concerned.

Organization. In the rest of the paper, we provide needed background on GPUs (Sec. 2), describe the image-processing benchmarks under consideration (Sec. 3), present our experimental data (Sec. 4), and conclude (Sec. 5).

2 Background on GPUs

In this section, we provide a brief introduction to GPU hardware and programming fundamentals.

GPU hardware. GPUs may be either discrete or integrated. *Discrete GPUs* are packaged on adapter cards that plug into a host computer bus. Such a GPU has its own local DRAM memory that is completely independent from the DRAM memory used by the host processor. Discrete GPUs commonly draw between 150 and 250 watts, need active cooling, and occupy substantial space. *Integrated GPUs* are commonly found in system-on-chip (SOC) designs. The SOC typically combines a multicore machine with a GPU and uses DRAM memory that is tightly shared between the GPU and CPU cores. Integrated GPUs commonly draw between 5 and 15 watts, require minimal cooling, and add virtually no space requirements. These attributes make integrated GPUs the *de facto* choice in many embedded computing domains.

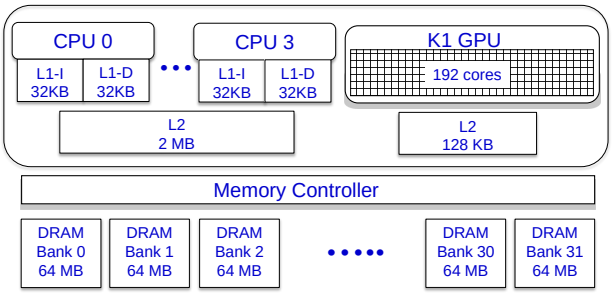

Figure 1: Jetson TK1 architecture.

Several SOC implementations with integrated GPUs capable of running sophisticated image-processing programs are on the market, including options from AMD [5], Intel [21], NXP [41] and NVIDIA [38]. In this work, we are using NVIDIA Jetson TK1 [39] and TX1 [40] boards, which retail for $200 and $600, respectively. These are likely acceptable price points in many automotive settings.

As illustrated in Fig. 1, the TK1 employs an SOC design that incorporates a quad-core 2.32 GHz 32-bit ARM machine and an integrated Kepler GK20a GPU. The CPUs share a 2-MB L2 cache. The GPU has 192 cores and a 128-KB L2 cache and provides up to 365 32-bit GFLOPS. The TK1 is a "big-little" platform in which an additional low power, low performance ARM CPU (not shown in Fig. 1) is provided on chip. The ARM CPUs and the GPU share 2 GB of 930 MHz DRAM memory partitioned into 32 banks.

The TX1 is a higher-end platform with a similar design. It consists of a quad-core 1.91 GHz 64-bit ARM machine, a 2-MB L2 cache shared by all CPUs, 4 GB of 1600 MHz DRAM, and an integrated Maxwell GM20B GPU. The GPU has 256 cores and a 256-KB L2 cache, and provides up to 512 32-bit GFLOPS. The TX1 is also a "big-little" platform.

As Fig. 1 suggests, GPU-using tasks may compete for many hardware resources. These resources include caches, DRAM memory banks, the memory bus and memory controller, and GPU cores. In prior work on real-time multicore computing, issues related to shared-hardware interference have received considerable attention [2, 3, 4, 6, 13, 14, 17, 18, 19, 20, 23, 27, 29, 28, 30, 31, 33, 35, 42, 44, 46, 50, 51, 52, 53]. However, we are aware of no such work that considers hardware interference with respect to GPU computations.

Obviously, concurrent GPU computations by different tasks may directly interfere with each other. Additionally, such computations can also interfere with programs running on CPU cores. For example, on both the TK1 and TX1, requests to load new lines into the GPU's L2 cache require accesses to the DRAM banks and may interfere with accesses by CPU cores. Further, so that GPU programs may be easily ported between discrete and integrated GPUs, CUDA (see below) explicitly treats memory as being either CPU-local (host memory) or GPU-local (device memory) and provides operations for copying data between the two. Such copy op-

erations run concurrently with programs running on both the GPU cores and the CPU cores, potentially creating additional DRAM interference.[2] With integrated GPUs, explicit data copying can be avoided by using the *zero-copy* functions of CUDA (see below).

GPU programming in CUDA. The following is a high-level description of GPU programming in CUDA [37]. A GPU is fundamentally a co-processor that performs operations requested by CPU programs. CUDA programs use a set of C or C++ library routines to request GPU operations that are implemented by a combination of hardware and device-driver software. The typical structure of a CUDA program is as follows: **(i)** allocate GPU-local (device) memory for data; **(ii)** use the GPU to copy data from host memory to GPU device memory; **(iii)** launch a program—called a *kernel*—to run on the GPU cores to compute some function on the data; **(iv)** use the GPU to copy output data from the device memory back to the host memory; **(v)** free the device memory. On integrated GPUs, CUDA provides a zero-copy option where programs can simply pass a pointer to shared memory where data used for a kernel is located—that is, explicit copying from CPU-local memory to GPU-local memory is avoided.

By default, copy operations are synchronous with respect to the CPU program: they do not return until the copy is complete and will not start until any prior kernels have finished. However, kernel launches are always asynchronous, and asynchronous copy operations are also available. These operations require the CPU process to explicitly wait for GPU operations to complete, using a configurable synchronization mechanism. We configured our experiments to block the CPU process while synchronizing.

CUDA operations pertaining to a given GPU are ordered by associating them with a *stream*. By default, there is a single stream for all programs that share a GPU, but multiple streams can be optionally created. Operations in a given stream are executed in FIFO order, but the order of execution across different streams is determined by the GPU scheduling in the device driver. They may execute concurrently (or out of request order with respect to other streams).

Each GPU operation from a CUDA program is represented internally by a command string that is written to a command buffer (queue) managed by the device driver. The driver then schedules these commands for execution on the GPU. Programmers can think of a GPU as being abstractly composed of one or more *copy engines (CEs)* that implement transfers of data between device memory and host memory, and an *execution engine (EE)* that executes GPU kernels. Both the TK1 and TX1 have one CE that moves data both ways.

EEs and CEs operate concurrently. When there are multiple streams, multiple kernels and one or two copy operations can operate concurrently depending on the GPU hardware. When a kernel is scheduled, it may not require all EE resources, in which case the GPU scheduler may co-schedule more than one kernel (from different streams only) to execute concurrently and increase GPU occupancy. Concurrent kernel execution can create more interference in the GPU L2 cache and for DRAM accesses. To the best of our knowledge, complete details of kernel attributes and policies used by NVIDIA to co-schedule kernels are not available.

[2] With discrete GPUs, only the GPU data-copy operations may cause DRAM interference with respect to CPU usage and then typically in the form of DMA operations over a bus.

3 Benchmark Programs

In the study presented herein, we considered both GPU programs and CPU-only benchmark programs.

GPU programs. We chose three CUDA programs as representative of typical image-processing computations, and a fourth to represent a general class of programs that create stress on GPU resources:

- **stereoDisparity (SD)**: Extracts 3D depth information from 2D images taken with a stereo camera. The input consists of left and right 640×533 color images; the output is a 640×533 grayscale image.
- **fastHOG (HOG)**: Detects objects in an image using histograms of oriented gradients. The input is a 640×480 color image; the output is a matrix of bounding-box coordinates and object-detection probabilities.
- **Convbench (CONV)**: Executes convolutional neural-network layers as used in image recognition. The input is a 227×227 color image; the output is a matrix of neural-network parameters.
- **matrixMul (MMUL)**: Multiplies two square matrices of 32-bit floats (16 MB each).

SD and MMUL were taken from CUDA samples distributed by NVIDIA [36], HOG was downloaded from Oxford University [43], and CONV was constructed using code from AlexNet [32], implemented in Caffe [24]. All programs were adapted to run as iterative tasks, with a short random sleep between iterations. Each iteration corresponds to processing one image (SD, HOG, and CONV) or performing one matrix multiplication (MMUL). The programs were instrumented to log total execution time and the time required for performing data copies and executing kernels in every iteration. Even though our experiments were conducted using fixed images as inputs, we still verified that none of the benchmarks exhibited different runtime characteristics based on the content of the input images.

Each CUDA program was executed in a stream of its own, with all memory copies performed asynchronously and placed in the stream along with kernel launches in the intended FIFO order. After each kernel launch or group of memory copies, the CPU execution of each program was blocked while waiting to synchronize with the GPU. Each program was structured to ensure that all memory allocation

and freeing operations were done outside the iteration loop and all memory accesses within each iteration were to pinned memory, as is common practice in real-time systems. Display operations for the visualization of input or output images were removed. Image input data was read from memory buffers as would happen with camera-driven input. Two versions of each program were constructed, one with zero-copy memory and one without.

CPU-only benchmark. We used this program as a CPU-only workload:

- **vectorAdd (VADD)**: Adds two vectors of 32-bit floats (16 MB each).

VADD was based on the CUDA samples [36] and instrumented in an identical fashion as the GPU programs, but launches no GPU kernels.

4 Experiments

We are interested in supporting automotive image-processing workloads on a multicore+GPU platform such as the TK1 or the TX1. We assume that such workloads have soft real-time constraints: missing a deadline (occasionally) does not have catastrophic consequences, as long as an incomplete frame can be dropped and the system as a whole can use redundant or historical data processed by hard real-time components as a fail-safe mechanism. Given this assumption, our tasks can be provisioned by determining their execution times via measurement. Such a provisioning could be based on a task's average-case execution time, its worst-case execution time, or some intermediate value between the two. A measurement-based approach is further justified by the lack of adequate static timing analysis tools for multicore+GPU platforms. Even if such tools did exist, they would probably produce execution-time estimates that are so pessimistic that virtually no interesting workload could be supported.

The issue being considered in this paper is whether allowing GPU co-scheduling might have schedulability benefits. To get a sense of any potential benefits, we conducted experiments on both the TK1 and TX1 in which the various benchmark programs described in Sec. 3 were used as surrogates for real application code. These experiments were designed to assess whether GPU co-scheduling can be constructive from a schedulability point of view. We assessed this by running different combinations of the benchmark programs and recording execution-time data. We call each experiment involving such a combination of programs a *scenario*. In each scenario, execution-time data was recorded for a set amount of time (typically 10–15 minutes) under the default Linux scheduler with the considered programs pinned to separate CPUs. We present our obtained execution-time data by plotting cumulative distribution functions (CDFs), as such functions provide a sense of the best-case, average-case, and worst-case recorded times. We denote a given scenario by simply listing the combination of programs that were run.

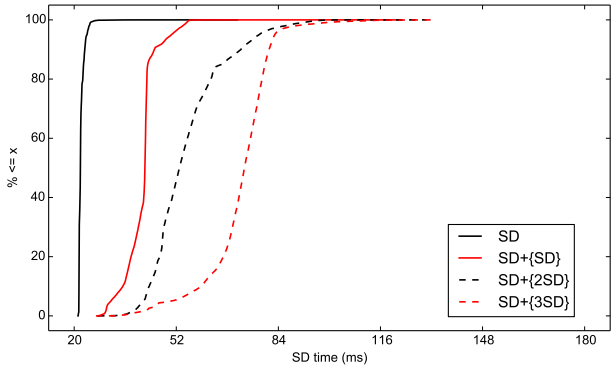

Figure 2: CDF of execution times of SD in scenarios only involving multiple SD instances.

For example, in the scenario HOG+{2SD,HOG}, execution-time data was obtained on one CPU for the HOG program in the presence of two instances of SD and another instance of HOG running on the other three CPUs.

In total, we tested 52 scenarios, each both with and without the zero copy feature of CUDA and on both the TK1 and TX1. Unless otherwise noted, the scenarios presented here were measured on the TK1 and did not use the zero-copy feature. Results for all considered scenarios can be found in an online appendix[3].

Typical observed trends. We begin by commenting on general trends seen in our collected data.

Obs. 1. GPU co-scheduling was always constructive in scenarios consisting of multiple instances of a single benchmark.

Fig. 2 supports this observation for the case of the SD benchmark. In this case, GPU co-scheduling is mildly constructive. While SD execution times do increase with more competition, they do not increase to the point of eliminating any benefit due to co-scheduling. In particular, the addition of one competitor yields execution times that are somewhat better than simply doubling the execution time of a single instance, and this trend continues to apply as more competition is introduced.

Obs. 2. GPU co-scheduling was so constructive in some scenarios that any introduced interference was practically negligible.

Fig. 3 supports this observation. Note that the execution times for SD remain virtually unaffected when instances of MMUL are introduced. This low impact is probably due to MMUL having short kernel execution times (approx. 1ms), which would rarely prevent SD from accessing the GPU.

Obs. 3. In some scenarios, particularly those involving HOG, GPU co-scheduling proved to be rather destructive.

Fig. 4 supports this observation. Note that the most destructive interference occurs when two instances of HOG and two instances of SD are run together, given by the curve for

[3] https://cs.unc.edu/~anderson/papers/ospert16_long.pdf

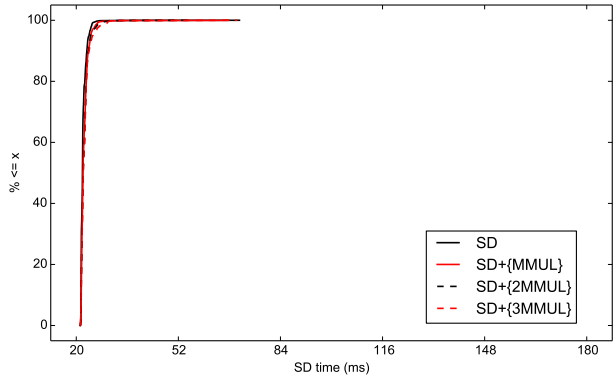

Figure 3: CDF of execution times of SD in scenarios involving MMUL competitors.

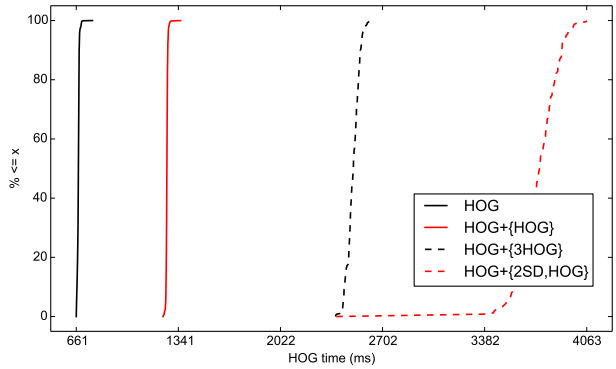

Figure 4: CDF of execution times of HOG in scenarios involving multiple instances of other benchmarks.

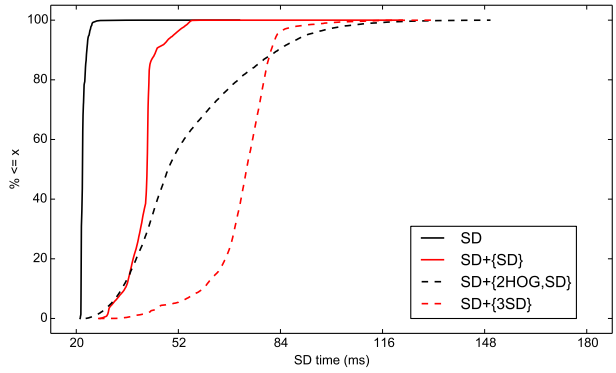

Figure 5: CDF of execution times of SD in scenarios involving multiple instances of other benchmarks.

the scenario HOG+{2SD,HOG}. Fig. 5 presents execution-time data for SD that allows us to examine this same scenario from the perspective of SD. In particular, note the curve labeled SD+{2HOG,SD} in Fig. 5.

In our TK1 experiments, the worst-case execution time of SD running in isolation was 71.1ms, and the worst-case execution time of HOG running in isolation was 768.9 ms. However, the *median* execution time of HOG in the HOG+{2SD,HOG} scenario was 3747.0ms. Had the scheduler simply treated the GPU as an exclusive resource when running two instances of HOG and two instances of SD, we could expect HOG's worst-case execution time to be closer to 1680.0ms, which is the sum of each instance's worst-case execution time in isolation. By examining the curve for SD+{2HOG,SD} in Fig. 5, we see that SD in this scenario has a median execution time only approximately 30ms worse than its execution time in isolation. Since a single iteration of HOG performs over 180 kernel invocations of varying sizes, and an iteration of SD performs only one, the plots support the hypothesis that a large portion of the effect on HOG is due to HOG's multiple kernels being interleaved with SD's single kernel at multiple points in each HOG iteration. While one may argue that this scheduling in SD's favor is beneficial in some applications, the significantly increased execution time for HOG may result in an overall net loss in terms of schedulability.

Obs. 4. The TX1 platform exhibited similar trends to those observed on the TK1.

The TX1, with greater resources, unsurprisingly exhibited improved execution times. Most interference patterns, however, applied to both platforms. This is shown in Fig. 7, which shows similar patterns to Fig. 4, and Fig. 8, which is analogous to Fig. 6 (discussed next).

An anomalous result. We conclude this section by discussing an anomalous result that suggests that further study of sources of interference among GPU-using tasks is needed.

Obs. 5. In rare cases, a benchmark program exhibited *better* performance when executing in the presence of a competing workload rather than in isolation.

We were very surprised to find that in some cases, increasing the concurrent workload unintuitively led to slight *improvements* in observed benchmark execution times. We observed such improvements in two sets of scenarios, shown in Figs. 6 and 8, where instances of HOG exhibited execution-time improvements with additional competition. This behavior was noticed in HOG with one or two VADD competitors on the TK1, and with up to 3 VADD competitors on the TX1. The only other scenarios where we observed such behavior involved the CONV benchmark competing against additional CONV instances.

Our current hypothesis is that this behavior is due to DRAM or CPU L2 cache activity. This hypothesis is based on the observation that, in Fig. 6, the VADD benchmark runs

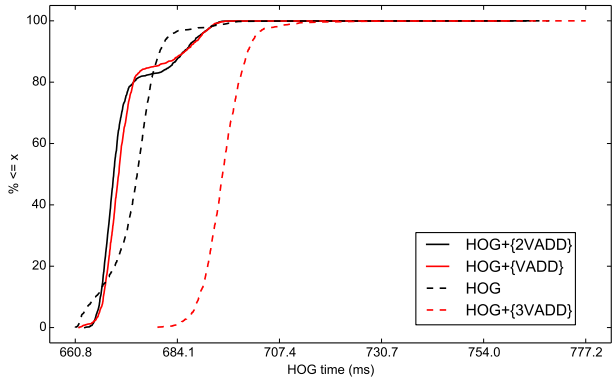

Figure 6: CDF of execution times of HOG in scenarios involving VADD competitors (which are CPU-only).

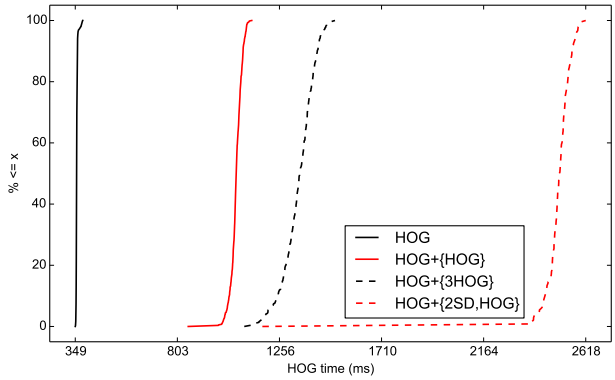

Figure 7: The same scenarios as Fig. 4 running on the TX1.

solely on the CPU. This fact eliminates GPU contention as the source of the anomaly in Fig. 6, leaving only hardware resources shared by the two benchmarks as potential causes: the CPU, its L2 cache, and the DRAM banks. We still, however, do not have a concrete explanation of this anomalous behavior, and plan to continue investigating it in hopes of identifying specific causes.

5 Conclusion

In order to effectively use GPUs in automotive settings, it is imperative to not waste GPU capacity. Such waste can lead to the necessity of introducing additional hardware, which can have a detrimental impact with respect to SWaP and monetary cost. Unfortunately, most prior GPU management frameworks proposed for real-time systems [7, 8, 9, 12, 16, 25, 26, 49, 47, 48, 54, 55] preclude multiple tasks from executing GPU kernels concurrently. If such a kernel requires only a relatively small fraction of a GPU's processing cores, then much of that GPU's capacity will be wasted. In this paper, we have explored the possibility of allowing multiple kernels to be co-scheduled in the context of image-processing applications. Our results suggest that, in some cases, allowing multiple kernels to be co-scheduled can have a positive impact on real-time schedulability. Allowing such functionality will require new extensions to prior real-time GPU management frameworks.

In future work, we plan to introduce such extensions to the frameworks developed by our group, GPUSync and GPUSyncLite. These extensions will require the use of real-time locking protocols that sometimes allow multiple tasks to hold locks simultaneously. Blocking analysis will be required for these protocols as well. We believe that the needed protocols can be obtained by using ideas found in recently proposed multiprocessor real-time locking protocols for managing replicated resources [34]. Our idea here is to abstractly view a single GPU as a replicated resource and require a task to lock only the replicas it needs. In other future work, we intend to conduct more in-depth experimental studies to try to discern the root sources of interference that cause some kernels to perform poorly when co-scheduled. Additionally, we plan to consider other GPU-based hardware platforms that might be viable in automotive use cases.

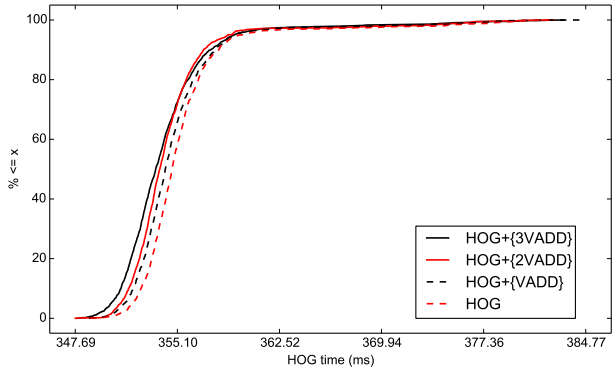

Figure 8: The same scenarios as Fig. 6 running on the TX1.

References

[1] Google self-driving car project. Online at `https://www.google.com/selfdrivingcar/`, 2016.

[2] A. Alhammad and R. Pellizzoni. Trading cores for memory bandwidth in real-time systems. In *RTAS '16*.

[3] A Alhammad, S. Wasly, and R. Pellizzoni. Memory efficient global scheduling of real-time tasks. In *RTAS '15*.

[4] S. Altmeyer, R. Douma, W. Lunniss, and R.I. Davis. Evaluation of cache partitioning for hard real-time systems. In *ECRTS '14*.

[5] AMD. Amd embedded g-series system-on-chip product brief. Online at `https://www.amd.com/Documents/AMDGSeriesSOCProductBrief.pdf`.

[6] N. Audsley. Memory architecture for NoC-based real-time mixed criticality systems. In *WMC '13*.

[7] J. Aumiller, S. Brandt, S. Kato, and N. Rath. Supporting low-latency CPS using GPUs and direct I/O schemes. In *RTCSA '12*.

[8] C. Basaran and K. Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *ECRTS '12*.

[9] K. Berezovskyi, , L. Santinelli, K. Bletsas, and E. Tovar. WCET measurement-based and extreme value theory characterisation of CUDA kernels. In *RTNS '14*.

[10] K. Berezovskyi, K. Bletsas, and B. Andersson. Makespan computation for GPU threads running on a single streaming multiprocessor. In *ECRTS '12*.

[11] K. Berezovskyi, K. Bletsas, and S. Petters. Faster makespan estimation

for GPU threads on a single streaming multiprocessor. In *ETFA '13*.

[12] A. Betts and A. Donaldson. Estimating the WCET of GPU-accelerated applications using hybrid analysis. In *ECRTS '13*.

[13] M. Campoy, A. Ivars, and J. Mataix. Static use of locking caches in multitask preemptive real-time systems. In *IEEE/IEE Real-Time Embedded Sys. Workshop '01*.

[14] M. Chisholm, B. Ward, N. Kim, and J. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *RTSS '15*.

[15] G. Elliott. *Real-Time Scheduling of GPUs, with Applications in Advanced Automotive Systems*. PhD thesis, University of North Carolina at Chapel Hill, 2015.

[16] G. Elliott, B. Ward, and J. Anderson. GPUSync: A framework for real-time GPU management. In *RTSS '13*.

[17] G. Giannopoulou, N. Stoimenov, P. Huang, and L.Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *EMSOFT '13*.

[18] M. Hassan and H. Patel. Criticality- and requirement-aware bus arbitration for multi-core mixed criticality systems. In *RTAS '16*.

[19] M. Hassan, H. Patel, and R. Pellizzoni. A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *RTAS '15*.

[20] J. Herter, P. Backes, F. Haupenthal, and J. Reineke. CAMA: A predictable cache-aware memory allocator. In *ECRTS '11*.

[21] Intel. Intel atom processor series product brief. Online at `http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/atom-x3-c3000-brief.pdf`.

[22] R. Jain, C. Hughs, and S. Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *RTSS '02*.

[23] J. Jalle, E. Quinones, J. Abella, L. Fossati, M. Zulianello, and P. Cazorla. A dual-criticality memory controller (DCmc) proposal and evaluation of a space case study. In *RTSS '14*.

[24] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *ACMMM '14*.

[25] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *RTSS '11*.

[26] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX Annual Technical Conference '11*.

[27] H. Kim, D. Broman, E. Lee, M. Zimmer, A. Shrivastava, and J. Oh. A predictable and command-level priority-based DRAM controller for mixed-criticality systems. In *RTAS '15*.

[28] H. Kim, D. de Niz, B. Anderson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *RTAS '14*.

[29] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *ECRTS '13*.

[30] N. Kim, B. Ward, M. Chisholm, C.-Y. Fu, J. Anderson, and F.D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *RTAS '16*.

[31] Y. Krishnapillai, Z. Wu, and R. Pellizzoni. ROC: A rank-switching, open-row DRAM controller for time-predictable systems. In *ECRTS '14*.

[32] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*.

[33] R. Mancuso, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time colored lockdown for cache-based multi-core architectures. In *RTAS '13*.

[34] C. Nemitz, K. Yang, M. Yang, P. Ekberg, and J. Anderson. Multiprocessor real-time locking protocols for replicated resources. In *ECRTS '16*.

[35] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity environment. In *ECRTS '14*.

[36] NVIDIA. Cuda sample programs. Online at `http://docs.nvidia.com/cuda/cuda-samples`.

[37] NVIDIA. Cuda zone. Online at `http://www.nvidia.com/object/cuda_home_new.html`.

[38] NVIDIA. Jetson tx1 system-on-module data sheet. Online at `https://developer.nvidia.com/embedded/downloads`.

[39] NVIDIA. Whitepaper: NVIDIA Tegra K1. Online at `http://www.nvidia.com/content/pdf/tegra_white_papers/tegra-k1-whitepaper.pdf`.

[40] NVIDIA. Whitepaper: NVIDIA Tegra X1. Online at `http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf`.

[41] NXP. i.mx 6dual/6quad automotive and infotainment applications processors data sheet. Online at `http://cache.freescale.com/files/32bit/doc/data_sheet/IMX6DQAEC.pdf`.

[42] R. Pellizzoni, A. Schranzhofer, J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *DATE '10*.

[43] V. Prisacariu and I. Reid. fastHOG–a real-time GPU implementation of HOG. *Department of Engineering Science*, 2310, 2009.

[44] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. Phatak, R. Pellizzoni, and M. Caccamo. A real-time scratchpad-centric OS for multi-core. In *RTAS '16*.

[45] S. Thrun. Toward robotic cars. *Communications of the ACM*, 53:99–106, 2010.

[46] P. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *RTAS '16*.

[47] U. Verner, A. Mendelson, and A. Schuster. Batch method for efficient resource sharing in real-time multi-GPU systems. In *ICDCN '14*.

[48] U. Verner, A. Mendelson, and A. Schuster. Scheduling periodic real-time communication in multi-GPU systems. In *ICCCN '14*.

[49] U. Verner, A. Mendelson, and A. Schuster. Scheduling processing of real-time data streams on heterogeneous multi-GPU systems. In *SYSTOR '12*.

[50] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *ECRTS '13*.

[51] M. Xu, S. Mohan, C. Chen, and L. Sha. Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *RTAS '16*.

[52] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicoore platforms. In *RTAS '14*.

[53] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *ECRTS '12*.

[54] J. Zhong and B. He. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *TIEEE Transactions on Parallel and Distributed Systems*, 25:15221532, 2014.

[55] H. Zhou, G. Tong, and C. Liu. GPES: A preemptive execution system for GPGPU computing. In *RTAS '15*.