Work-In-Progress: Protecting Real-Time GPU Applications on Integrated CPU-GPU SoC Platforms

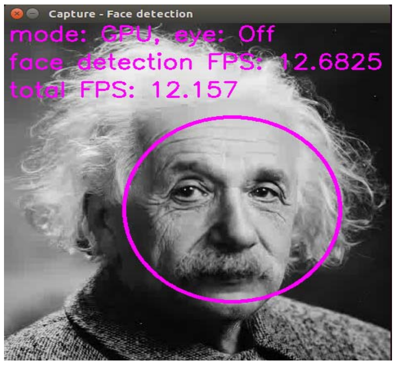
Waqar Ali, Heechul Yun
University of Kansas, USA.
{wali, heechul.yun}@ku.edu

***Abstract*—Integrated CPU-GPU architecture provides excellent acceleration capabilities for data parallel applications on embedded platforms while meeting the size, weight and power (SWaP) requirements. However, sharing of main memory between CPU applications and GPU kernels can severely affect the execution of GPU kernels and diminish the performance gain provided by GPU. In the NVIDIA Tegra TK1 platform which has the integrated CPU-GPU architecture, we noticed that in the worst case scenario, the GPU kernels can suffer as much as 4X slowdown in the presence of co-running memory intensive CPU applications compared to their solo execution. In this paper, we propose a kernel mechanism called BWLOCK++ which can be used to protect the performance of GPU kernels from co-running memory intensive CPU applications. Our preliminary investigations show that by using BWLOCK++, the performance slowdown of GPU kernels in the presence of memory contention can be decreased by up-to 63%.**

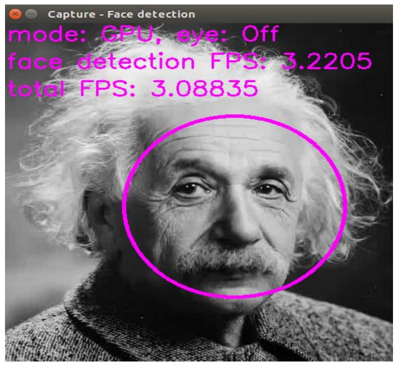
I. Introduction

Integrated CPU-GPU architecture based system-on-a-chip (SoC) platforms, such as NVIDIA's Tegra TK1 / TX1, are increasingly demanded for performance intensive cyber-physical systems (CPS) such as autonomous cars and unmanned aerial vehicles (UAVs). These CPS require high computing performance to process the vast amount of data flowing from a variety of sensors in real-time (e.g. real-time obstacle detection and avoidance) while satisfying a number of size, weight and power (SWaP) constraints. This makes systems with integrated CPU-GPU architectures an attractive option for CPS because such systems can provide high performance while meeting SWaP requirements [6].

Designing critical real-time applications on integrated CPU-GPU architectures is, however, challenging because contention in the shared hardware resources (e.g. memory bandwidth) can significantly alter the application's timing characteristics. For example, in the NVIDIA Tegra TK1 platform, the CPU cores and GPU use a single shared main memory subsystem. Therefore, memory intensive batch jobs running on CPU cores can cause significant delays to important real-time GPU tasks running in parallel due to memory bandwidth contention.



(a) Run-alone



(b) Co-run

Fig. 1: Face-detection performance[1]comparison on a NVIDIA Tegra TK1 (4 CPU + GPU): (a) Running alone on the system (the algorithm uses 1 CPU and the GPU); (b) Co-running with three memory-intensive tasks (each uses 1 idle CPU core)

To illustrate the significance of the problem, we evaluated a vision based face detection algorithm [8] on NVIDIA Tegra TK1 platform (4 ARM CPU cores + 192 core GPU). This

[1]The performance was measured with the dataset created by taking a screencast of the sample image for sixty seconds. The resolution of the resulting video file was scaled to 472x606 pixels. The screencast was used to ensure that both the solo and corun cases get the same data stream during the entire duration of the experiment.

algorithm is single threaded w.r.t. CPU but uses GPU to accelerate performance. That is, it uses at most one CPU core and the GPU, leaving three idle CPU cores in the system. In Figure 1(a), we measured performance (frames/sec) of the algorithm in isolation. In Figure 1(b), on the other hand, we co-schedule three memory intensive tasks on the idle system cores and measure the corun performance. As can be seen in the latter figure, co-scheduling the memory intensive tasks on the idle cores significantly decreases the performance of this vision-based face detection algorithm—resulting in an approximately 4X slowdown. The main cause of the problem is that, in the Tegra TK1 platform, both CPU and GPU share the main memory and its limited memory bandwidth becomes a bottleneck. As a result, even though the platform offers plenty of raw performance, if left uncontrolled, the system may fail to meet desired real-time performance.

In this paper, we present our software (OS) based approach to mitigate the memory bandwidth contention problem in integrated CPU-GPU architectures. We also present preliminary evaluation results using the NVIDIA Tegra TK1 platform.

II. Background and Related Work

We first provide a brief high-level description of how GPU works. GPU is an accelerator that executes some specific functions requested by a master CPU program. Requests to the GPU can be made by using GPU programming frameworks such as CUDA that offer standard APIs. A request to GPU is typically composed of the following four predictable steps:

- Copy data from host memory to device (GPU) memory
- Launch the function—called *kernel*—to be executed on the GPU
- Wait until the kernel finishes
- Copy the output from device memory to host memory

In the real-time system community, GPUs have been studied actively in recent years because of their potential benefits in demanding data-parallel real-time applications [5]. Several real-time GPU resource management frameworks focus on predictable scheduling of multiple GPU kernels on discrete GPUs [3], [4], [10]. In [1] the authors identify that the GPU kernels typically demand high memory bandwidth to achieve high data parallelism. They further show that if the memory bandwidth required by GPU kernels is limited or reduced, it can result in significant deterioration in the overall performance of the GPU application.

These prior works, however, focus on addressing the contention between multiple concurrent GPU kernels within a discrete GPU, while our paper focuses on the contention between CPUs and GPU in integrated CPU-GPU architecture. Unlike discrete GPU based platforms in which the GPU typically has its own dedicated high bandwidth (GDDR) memory, integrated CPU-GPU platforms usually have shared main memory between CPU applications and GPU kernels and the performance impact of sharing this memory can be significant as demonstrated in the previous section.

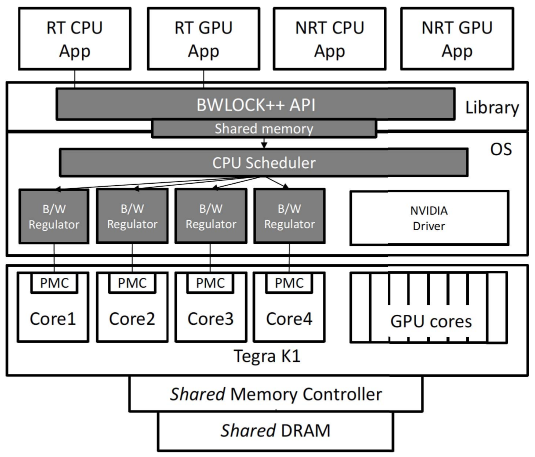


Fig. 2: BWLOCK++ on Tegra TK1

III. BWLOCK++ for Integrated CPU-GPU Architectures

In this work, we propose BWLOCK++ to protect GPU applications on integrated CPU-GPU architecture based SoC platforms. We exploit the fact that each GPU kernel is executed via explicit programming interfaces from a corresponding host CPU program. In other words, we can precisely determine when the GPU kernel starts and finishes by instrumenting these functions. To avoid memory bandwidth contention from the CPU, we notify the OS (Linux) before a GPU application launches the GPU kernel and after the kernel completes. While the GPU kernel is being executed, the OS regulates memory bandwidth consumption of the CPU cores to minimize bandwidth contention. Concretely, each core is periodically given a small amount of memory bandwidth budget. If the core uses up its given budget for the specified period, the (non-RT) CPU tasks running on that core are throttled. In this way, the GPU kernel suffer minimal memory bandwidth interference from the CPU cores.

Figure 2 shows the overall architecture of the BWLOCK++ framework on the Tegra TK1 platform. BWLOCK++ builds on our previous work BWLOCK [9], which target homogeneous multicore processors. In BWLOCK, the main goal is to protect soft real-time (SRT) CPU tasks from concurrent non-real-time tasks by throttling memory bandwidths of the non-real-time tasks. This is achieved by manually calling a lock-like API within the SRT task that wishes to protect its memory performance critical code region. In contrast, the goal of BWLOCK++ is to protect real-time GPU kernels from concurrent CPU tasks, on integrated CPU-GPU based SOC platforms, ideally without requiring manual programmer efforts.

IV. Prototype Implementation

In order to test the feasibility of BWLOCK++, we first ported our prior work [9]—which was originally developed for Intel Haswell-based platform—on a NVIDIA Tegra TK1

Benchmark	Performance (FPS)			
	Solo	Co-Run	BWLOCK++	Improvement (%)
Face	10.77	5.01	8.16	63
Hog	3.76	3.09	3.74	21
Flow	8.51	5.87	6.38	9

TABLE II: GPU Benchmark Performance under Bandwidth Contention

Event ID	Description
L2D_CACHE_REFILL_LD	Level-2 Data Cache Refill - Read
L2D_CACHE_REFILL_ST	Level-2 Data Cache Refill - Write

TABLE I: L2 Performance Counters on ARM Cortex-A15

platform. The main challenge we faced in this porting effort is the identification of the correct hardware performance counters to measure memory traffic generated by applications running on CPU cores. In the original x86 implementation, we use the *PERF_COUNT_HW_CACHE_MISSES* event in Linux, which is mapped to the LLC miss counter on Intel processors. On ARM processors, however, we found that the said event is not mapped to the last level cache miss counter. Instead, it is mapped to the L1-data cache miss counter (*L1_DCACHE_REFILL*), which cannot be used to measure the off-chip memory bandwidth traffic. From the Cortex-A15 [2] reference manual, we found two L2 (LLC on Cortex-A15) cache related performance counters—one for read and one for write refills, shown in Table I. Our current proof-of-concept implementation uses only the read counter in measuring off-chip memory traffic. In the future, we plan to investigate ways to use both read and write-refill counters to account for write-refill memory traffic as well. Note that write-intensive memory traffic can cause more severe interference than read traffic [7].

V. Preliminary Evaluation Results

To evaluate the feasibility of GPU kernel protection, we use three GPU-enabled OpenCV applications, which are included as samples in the OpenCV distribution, as workloads. Face is a face detection algorithm, which uses Haar Cascade classifiers; Hog is the Histogram of Oriented Gradients method based object detector; Flow is an Optical Flow algorithm implementation. All algorithms take the same video file as input. This video was created by recording real-time video feed from a camera. Note that this video file was different from the one used in Section I.

Because our current prototype does not implement automatic instrumentation, we manually added bandwidth lock system calls around GPU kernel launch/wait code in the test OpenCV applications. The basic experiment setup is as follows. We first measure the performance of each subject OpenCV benchmark in isolation (Solo). We then repeat the experiment with three synthetic memory read traffic generator programs—that is one core runs the subject benchmark and the rest of the cores run the memory traffic generators. Finally, we repeat the whole experiment with BWLOCK++ in the presence of the same corunners using our prototype implementation. Table II shows the results.

In case of Face, it can be seen that BWLOCK++ provides significant performance improvement over the unregulated corun case. For the other two benchmarks, the performance gain is not as significant but it is still observable (21% for Hog and 9% for Flow).

VI. On-going/Future Work

We have presented our preliminary investigation on the feasibility and effectiveness of the proposed BWLOCK++ approach. Our goal is to protect real-time performance of GPU tasks on integrated CPU-GPU architecture based SoC platforms. Our preliminary results are encouraging as we observe noticeable performance improvements (up to 63%) in the tested GPU enabled OpenCV applications in the presence of contending memory bandwidth intensive CPU applications.

Our future work include the followings: (1) We plan to implement runtime instrumentation to transparently insert bandwidth lock without manual code modifications; (2) We plan to develop CPU and GPU task co-scheduling algorithms to minimize memory bandwidth contention, which would reduce the amount of time to throttle the CPU cores; (3) We plan to evaluate BWLOCK++ on different integrated CPU-GPU platforms, including the Tegra TX1 platform.

References

[1] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler. Page Placement Strategies for GPUs within Heterogeneous Memory Systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 607–618. ACM, 2015.

[2] ARM. *Cortex-A15 Technical Reference Manual, Rev: r2p0*, 2011.

[3] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *USENIX Annual Technical Conference (ATC)*. USENIX, 2011.

[4] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-class gpu resource management in the operating system. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 37–37, Berkeley, CA, USA, 2012. USENIX Association.

[5] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, Nov 2015.

[6] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, and S. Wang. An evaluation of the nvidia tx1 for supporting real-time computer-vision workloads. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2017.

[7] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016.

[8] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition (CVPR)*, volume 1, pages I–511. IEEE, 2001.

[9] H. Yun, W. Ali, S. Gondi, and S. Biswas. BWLOCK: A Dynamic Memory Access Control Framework for Soft Real-Time Applications on Multicore Platforms. *IEEE Transactions on Computers (TC)*, PP(99):1–1, 2016.

[10] H. Zhou, G. Tong, and C. Liu. Gpes: a preemptive execution system for gpgpu computing. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 87–97, April 2015.