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. For example, in the NVIDIA Tegra K1 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 software mechanism, which we call BWLOCK++, to protect the performance of GPU kernels from co-scheduled memory intensive CPU applications.**

I. Introduction

Graphic Processing Units (GPUs) are increasingly important computing resources to accelerate a growing number of data parallel applications, especially in the field of artificial intelligence that utilizes various forms of deep neural networks (DNNs). In recent years, DNNs are actively used in developing intelligent robots, such as UAVs and autonomous cars, and the GPUs have been key to efficiently handle the AI workloads. These intelligent robots are, however, resource constrained real-time embedded systems that not only require high computing performance but also must satisfy a variety of constraints such as cost, size, weight, and power consumption. This makes integrated CPU-GPU architecture based computing platforms, whichs integrate CPU and GPU in a single chip (e.g., NVIDIA's Jetson [1] series), an appealing solution for such robotics applicatiosn because of their high performance and efficiency [14].

Designing critical real-time applications on integrated CPU-GPU architectures is, however, challenging because of the contention in the shared hardware resources (e.g., memory bandwidth) which can significantly alter the application's timing characteristics. On integrated CPU-GPU architectures, such as NVIDIA Tegra K1, the CPU cores and the GPU typically share a single main memory subsystem. This enables the memory intensive batch jobs running on CPU cores to significantly interfere with the execution of real-time GPU tasks (e.g., vision based collision detection and avoidance) running in parallel; due to memory bandwidth contention.

To illustrate the significance of the problem stated above, we evaluate a vision based face detection benchmark [20] on a NVIDIA Tegra K1 platform (4 ARM CPU cores + 192 core GPU). The benchmark is single threaded w.r.t. the CPU but it uses the GPU to accelerate performance. That is, it uses at most one CPU core and the GPU, leaving three idle CPU cores in the system. Figure 1 shows the result of an experiment using the face detection benchmark. In *Solo*, we measure performance (frames/sec) of the benchmark in isolation. In *Corun-1*, *Corun-2* and *Corun-3*, on the other hand, we repeat the experiment after co-scheduling one, two and three memory intensive tasks respectively on the idle cores to observe the impact of the CPU co-runners on the execution of GPU benchmark. As can be seen in the figure, co-scheduling the memory intensive tasks on the idle cores significantly decreases the performance of this vision-based face detection benchmark—resulting in an approximately 3.3X slowdown; despite the fact that the benchmark has exclusive access to a CPU core and the GPU. The main cause of the problem is that, in the Tegra K1 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, no real-time execution guarantees can be provided if the system is left unmanaged.

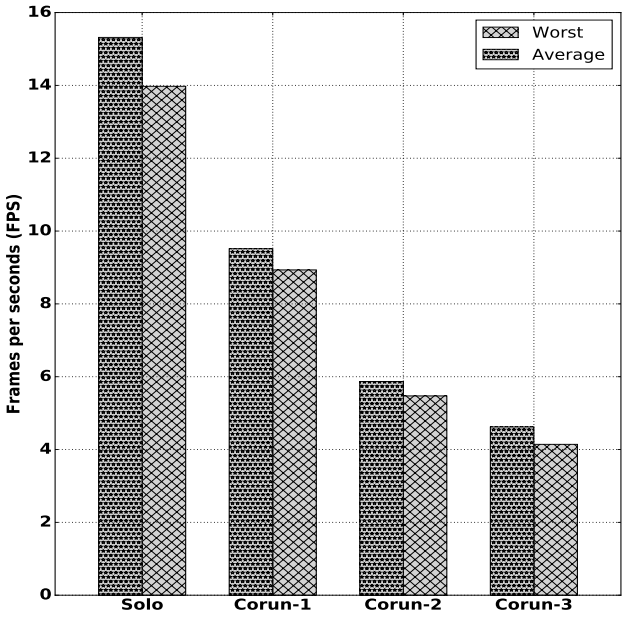


Fig. 1: Face-detection performance on NVIDIA Tegra K1 with CPU corunners

In this paper, we present a software based approach, which

we call BWLOCK++, to mitigate the memory bandwidth contention problem in integrated CPU-GPU architectures. More specifically, we focus on protecting real-time GPU tasks from the interference of non-critical but memory intensive CPU tasks. BWLOCK++ dynamically instruments GPU tasks at run-time and insert a *memory bandwidth lock* while critical GPU kernels are being executed on the GPU. When the bandwidth lock is being held by the GPU, the OS throttles the maximum memory bandwidth usage of the CPU cores to a certain threshold value to protect the GPU kernels. The threshold value is determined on a per GPU task basis and may vary depending on the GPU task's sensitivity to memory bandwidth contention. Throttling CPU cores inevitably negatively affect the CPU throughput. To minimize the throughput impact, we propose a throttling-aware CPU scheduling algorithm, which we call Throttle Fair Scheduler (TFS). TFS favors CPU intensive tasks over memory intensive ones while the GPU is busy executing critical GPU tasks in order to minimize CPU throttling. Our evaluation shows that BWLOCK++ can provide good performance isolation for bandwidth intensive GPU tasks in the presence of memory intensive CPU tasks. Furthermore, the TFS scheduling algorithm reduces the CPU throughput loss by up to 60%.

We make the following contributions in this paper:

- We present a software mechanism to ensure real-time performance of critical GPU kernels in the presence of memory intensive CPU applications on integrated CPU-GPU architecture platforms.
- We introduce an automatic GPU kernel instrumentation mechanism that eliminates the need of manual programmer intervention to protect GPU kernels.
- We identify a negative feedback effect of memory bandwidth throttling when used with Linux's CFS [13] scheduler. We present a throttling-aware CPU scheduling algorithm that solves the problem.
- We present detailed evaluation results of our framework on a NVIDIA Tegra K1 platform.

The remainder of this paper is organized as follows. We present necessary background and discuss related work in Section II. Section III provides details of the implementation of our software framework BWLOCK++ and the challenges involved in its design. In Section IV, we describe our evaluation platform and present evaluation results using a set of GPU benchmarks. We discuss the limitations of our approach in Section V and conclude in Section VI.

II. Background and Related Work

In this section, we provide necessary background and discuss related work.

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 systems community, GPUs have been studied actively in recent years because of their potential benefits in demanding data-parallel real-time applications [11]. As observed in [3], GPU kernels typically demand high memory bandwidth to achieve high data parallelism and if the memory bandwidth required by GPU kernels is not satisfied; it can result in significant performance reduction.

For discrete GPUs, which have dedicated graphic memories, researchers have focused on addressing interference among the co-scheduled GPU tasks. Many real-time GPU resource management frameworks adopt scheduling based approaches, similar to real-time CPU scheduling, that provide priority or server based scheduling of GPU tasks [9], [10], [23]. Elliot et al., formulate the GPU resource management problem as a synchronization problem and propose the GPUSync framework that uses real-time locking protocols to deterministically handle GPU access requests [6]. Here, at any given time, one GPU kernel is allowed to utilize the GPU to eliminate the unpredictability caused by co-scheduled GPU kernels. In [12], instead of using a real-time locking protocol that suffers from busy-waiting at the CPU side, the authors propose a GPU server mechanism which centralizes access to the GPU and allows CPU suspension (thus eliminating the CPU busy-waiting). All the aforementioned frameworks primarily work for discrete GPUs, which have dedicated graphic memory, but they do not guarantee predictable GPU timing on integrated CPU-GPU architecture based platforms because they do not consider the problem of the shared memory bandwidth contention between the CPU and the GPU.

Integrated GPU based platforms have recently gained much attention in the real-time systems community. In [14], [15], the authors investigate the suitability of NVIDIA's Tegra X1 platform for use in safety critical real-time systems. With careful reverse engineering, they have identified undisclosed scheduling policies that determine how concurrent GPU kernels are scheduled on the platform. In SiGAMMA [5], the authors present a novel mechanism to preempt the GPU kernel approach using a high-priority spinning GPU kernel to protect critical real-time CPU applications. Their work is orthogonal to ours as it solves the problem of protecting CPU tasks from GPU tasks while our work solves the problem of protecting GPU tasks from CPU tasks.

More recently, GPUGuard [7] provides a mechanism for deterministically arbitrating memory access requests between CPU cores and GPU in heterogeneous platforms containing integrated GPUs. They extend the PREM execution model [16], in which a (CPU) task is assumed to have distinct computation and memory phases, to model GPU tasks. GPUGuard provides deterministic memory access by ensuring that only a single PREM memory phase is in execution at any given time. Although GPUGuard can provide strong isolation guarantees, the drawback is that it may require significant restructuring

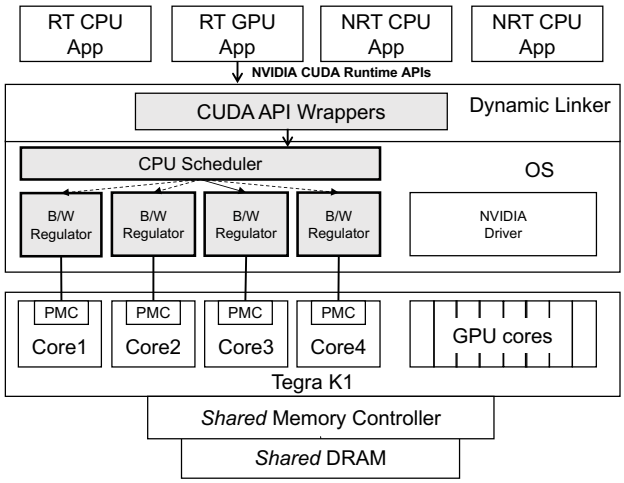


Fig. 2: BWLOCK++ System Architecture

of application source code to be compatible with the PREM model.

In this paper, we favor a less instrusive approach that requires minimal or no programmer invention. Our approach is rooted on a kernel-level memory bandwidth throttling mechanism called MemGaurd [22], which utilizes hardware performance counters of the CPU cores to limit memory bandwidth consumption of the individual cores for a fixed time interval on homogeneous multicore architectures. MemGuard enables a system designer—not individual application programmers—to partition memory bandwidth among the CPU cores. However, MemGuard suffers from system-level throughput reduction due to its coarse-grain bandwidth control (per-core-level control). In contrast, BWLOCK [21] is also based on a memory bandwidth throttling mechanism on homogeneous multicore architectures but it requires a certain degree of programmer intervention for fine-grain bandwidth control by exposing a simple lock-like API to applications. The API can effectively enable/disable memory bandwidth control in a fine-grain manner within the application source code.

Our work is also based on memory bandwidth throttling but it focuses on integrated CPU-GPU architectures and does not require any programmer intervention while minimizing throughput reduction of coarse-grain bandwidth throttling, as will be described in the following section.

III. BWLOCK++

In this section, we provide an overview of BWLOCK++, and discuss its design and implementation details.

A. Overview

BWLOCK++ is a software framework to protect GPU applications on integrated CPU-GPU architecture based SoC platforms. We focus on the problem of the shared memory bandwidth contention between GPU kernels and CPU tasks in integrated CPU-GPU architectures. More specifically, we focus on protecting real-time GPU tasks from the interference of non-critical but memory intensive CPU tasks.

As discussed in Section II, memory bandwidth is an important shared resource in an integrated CPU-GPU architecture. As more and more tasks, including real-time ones, are accelerated using integrated GPUs, a solution is needed to guarantee their performance without sacrificing too much efficiency and system utilization.

In BWLOCK++, 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 before a GPU application launches a GPU kernel, and after the kernel completes. While the GPU kernel is being executed, the GPU kernel holds a lock, which we call a *memory bandwidth lock*. While the bandwidth lock is being held, the OS regulates memory bandwidth consumption of the CPU cores to minimize bandwidth contention with the GPU kernel. Concretely, each core is periodically given a certain 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 suffers minimal memory bandwidth interference from the CPU cores. However, throttling CPU cores could significantly lower the overall system throughput. To minimize the negative throughput impact, we propose a new CPU scheduling algorithm, which we call the Throttling Fair Scheduler (TFS), to minimize the duration of CPU throttling without affecting memory bandwidth guarantees for real-time GPU applications.

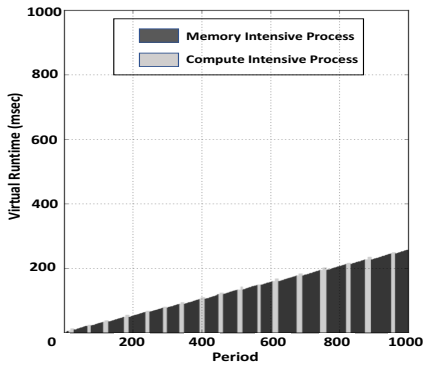
Figure 2 shows the overall architecture of the BWLOCK++ framework on an integrated CPU-GPU architecture (NVIDIA Tegra K1 platform). BWLOCK++ is comprised of three major components: (1) Dynamic run-time library for instrumenting GPU applications; (2) the Throttle Fair Scheduler; (3) Per-core B/W regulator. Working together, they protect real-time GPU kernels and minimize CPU throughput reduction. We will explain each component in the following sub-sections.

B. Automatic Instrumentation of GPU Applications

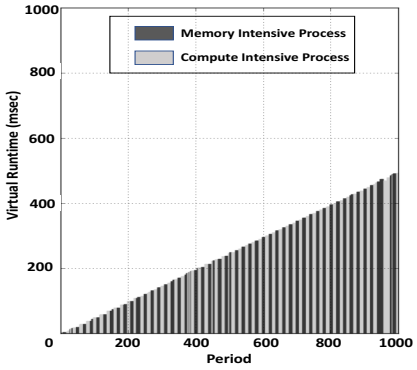
To eliminate manual programming efforts, we automatically instrument the program binary at the dynamic linker level.

We exploit the fact that the execution of a GPU application using a GPU runtime library such as NVIDIA CUDA typically follows fairly predictable patterns. Figure 4 shows the execution timeline of a typical synchronous GPU application that uses the CUDA API. The program code begins its execution on the CPU. It first allocates a memory block on the GPU and copies data from the host memory to the allocated GPU memory. Then it launches a GPU kernel, which is then executed on the GPU. In the meantime, the CPU is blocked (or executes something else) until the GPU kernel completes. Once the kernel completes, the CPU copies the resulting output from the GPU memory to the host memory.

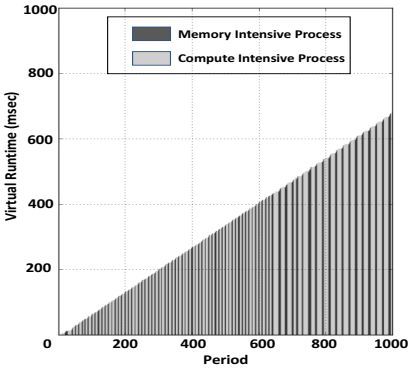
In order to protect the runtime performance of a GPU application from co-running memory intensive CPU applications,



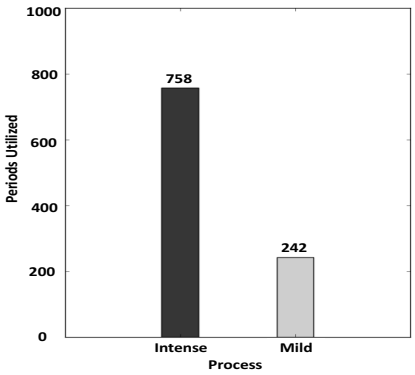
(a) Virtual Runtime Progression under Throttling (CFS)



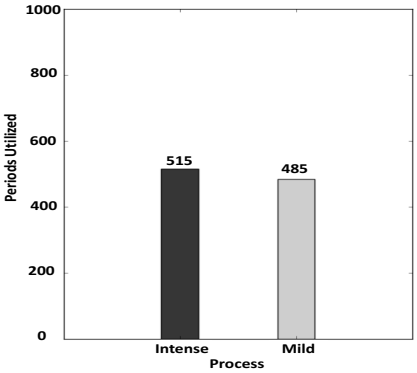
(b) Virtual Runtime Progression under TFS



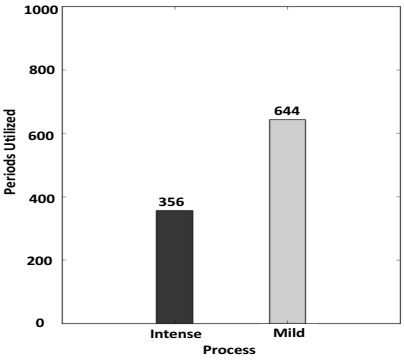
(c) Virtual Runtime Progression under TFS-3X



(d) Periods Utilized under Throttling (CFS)



(e) Periods Utilized under TFS



(f) Periods Utilized under TFS-3X

Fig. 3: Effect of throttling on CPU scheduling

API	Action	Description
cudaLaunch	Acquire BWLOCK++	Launch a GPU kernel
cudaDeviceSynchronize	Release BWLOCK++	Ascertain whether all the previously requested tasks on a specific GPU device have completed
cudaStreamSynchronize	Release BWLOCK++	Wait for all the tasks launched in a specific GPU stream to complete
cudaEventSynchronize	Release BWLOCK++	Wait until the completion of all work preceding the most recent call to cudaEventCreate in the appropriate compute stream

TABLE I: CUDA APIs instrumented via `LD_PRELOAD` for BWLOCK++

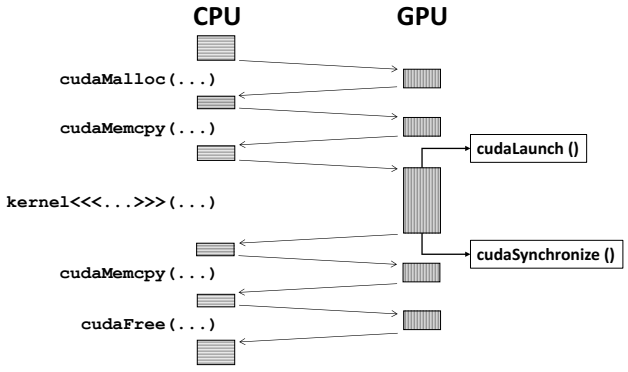
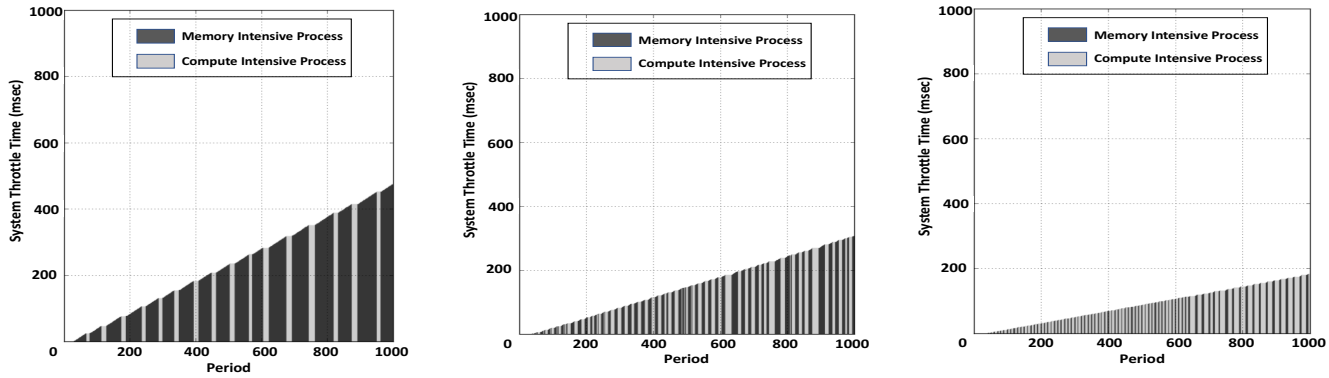


Fig. 4: Phases of GPU Application under CUDA Runtime

we need to ensure that the GPU application automatically holds the memory bandwidth lock while a GPU kernel is executing on the GPU. Upon the completion of the execution of the kernel, the GPU application again shall automatically release the bandwidth lock. This is done by instrumenting a small subset of CUDA API functions that are invoked when launching a kernel on GPU for execution or waiting for the kernel to complete its execution. These APIs are documented in Table I. More specifically, we write wrappers for these functions of interest which request/release bandwidth lock on behalf of the GPU application before calling the actual CUDA library functions. We compile these functions as a shared library and use Linux' `LD_PRELOAD` mechanism [8] to force the GPU application to use those wrapper functions whenever the CUDA functions are called. In this way, we automatically throttle CPU cores' bandwidth usage whenever real-time GPU kernels are being executed so that the GPU kernels' memory



(a) System Throttle Time with normal Throttling (b) System Throttle Time with TFS (c) System Throttle Time with TFS-3X

Fig. 5: System throttle time progression under different throttling schemes

bandwidth can be guaranteed.

A complication to the automatic GPU kernel instrumentation arises when the application uses asynchronous CUDA APIs to launch multiple GPU kernels in succession and then waits for those kernels to complete. In such a case, acquiring and releasing memory bandwidth lock on a per kernel basis is not effective. We circumvent this problem by using a nested locking scheme. The assumption under nested locking is that the application must wait separately for the completion of each individual GPU kernel that it launches. Upon launching the first GPU kernel, we acquire the memory bandwidth lock using our CUDA API wrappers as described above and increment the global nesting count. Each subsequent GPU kernel launch increments this count. When a GPU kernel completes its execution, the BWLOCK++ nesting count is decreased and the lock is released when the nesting count approaches zero. Using this scheme, we have been able to automatically instrument all the GPU benchmarks that we have used in evaluation of BWLOCK++.

The obvious drawback of throttling CPU cores is that the CPU throughput may be affected especially if some of the tasks on the CPU cores are memory bandwidth intensive. In the following sub-section, we discuss the impact of throttling on CPU throughput and present a new CPU scheduling algorithm that minimizes throughput reduction.

C. Throttle Fair CPU Scheduler

As described earlier in this section, BWLOCK++ uses a throttling based approach to enforce memory bandwidth limit of CPU cores at a regular interval. Although effective in protecting critical GPU applications in the presence of memory intensive CPU applications, this approach runs into the risk of severely under-utilizing the system's CPU capacity; especially in cases when there are multiple best-effort CPU applications with different memory characteristics running on the cores without holding the bandwidth lock. In the throttling based design, once a core exceeds its memory bandwidth quota and gets throttled, that core cannot be used for the remainder of the period. Let us denote the regulation period as T (i.e., $T = 1ms$) and the time instant at which an offending core exceeds its bandwidth budget as τ. Then wasted time due to throttling can be described as $T - \tau$ and the smaller the value of τ (i.e., throttled earlier in the period) the larger the penalty to the overall system throughput. The value of τ depends on the rate at which a core consumes its allocated memory budget and that in turn depends on the memory characteristics of the application executing on that core. To maximize the overall system throughput, the value of τ should be maximized—that is if throttling never occurs, $\tau = T$, or occurs late in the period, throughput reduction will be less.

1) Effect of throttling on CFS: One way to reduce CPU throttling is to schedule less memory bandwidth demanding tasks on the CPU while the GPU is holding the bandwidth lock. Assuming that each CPU core has a mix of memory bandwidth intensive and CPU intensive tasks, then scheduling the CPU intensive tasks while the GPU is holding the lock would reduce CPU throttling or at least delay the instant τ, which in turn improves CPU throughput. Unfortunately, Linux's default scheduler CFS [13] actually aggravates the possibility of early and frequent throttling when used with BWLOCK++'s throttling mechanism. The CFS algorithm tries to allocate fair amount of CPU time among tasks by using each task's weighted virtual runtime as the scheduling metric. The problem with throttling under CFS arises because the virtual run-time of a memory intensive process, which gets frequently throttled, increases more slowly than the virtual run-time of a compute intensive process which does not get throttled. This effect can be seen in Figure 3. Because of slower virtual run-time progression, the memory intensive process gets preferred by the CFS scheduler at each scheduling instance. This can be seen more clearly in the bottom part of Figure 3. In this figure, we plot the number of periods utilized by each task on a CPU core, over a course of one thousand sampling periods. Under CFS, out of all the sampling periods, 75% are utilized by the memory intensive process and only 25% are utilized by the compute intensive process. However, in each period that the memory intensive process gets to run in, it incurs throttling overhead which is captured by the throttle time metric as

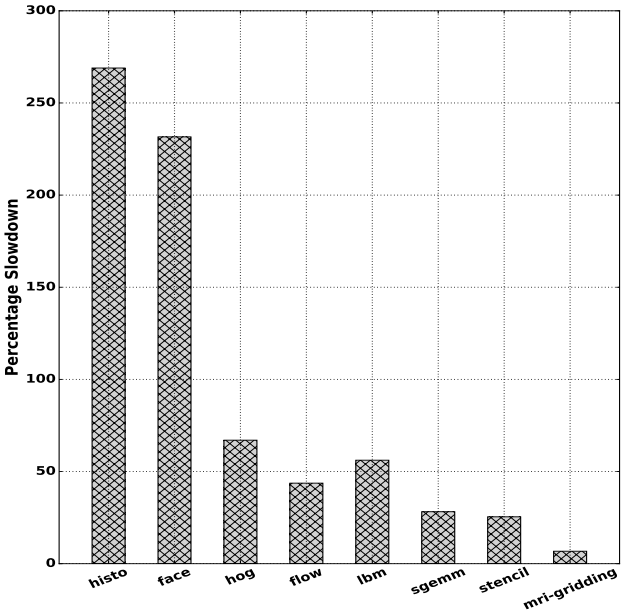


Fig. 6: Slowdown of the kernel execution time of GPU benchmarks due to three *Bandwidth* corunners

shown in Figure 5. This becomes a negative feedback loop which makes the system susceptible to severe capacity loss.

In order to circumvent this problem, we present a modification to the CFS algorithm in which the throttle time penalty that a process incurs is taken into account in its scheduling. This is done by simply adding the throttle time value of the process to its virtual run-time at the end of each BWLOCK++ sampling period. We call this modification to the CFS algorithm the Throttle Fair Scheduling (TFS) algorithm. Using this approach, we are able to significantly reduce CPU capacity loss while protecting the performance of memory critical applications. This design can be further extended to preferentially schedule compute intensive processes on the CPU while throttling is in place. Figure 3(f) shows the case where the throttling penalty, that a memory intensive thread incurs, is scaled 3X and then added to its virtual run-time. This makes the virtual run-time of the memory intensive thread increase sharply in the presence of throttling which makes it less suitable for scheduling. The net result of this is less throttling and improved overall system throughput.

D. Memory Bandwidth Regulator

In our framework, when the GPU kernels are busy running, CPU cores are throttled using a kernel-level memory bandwidth regulator module that utilizes hardware performance counters to trigger interrupts when memory bandwidth budgets of the cores are exhausted. The basic throttling mechanism of BWLOCK++ is the same as [21]. However, one main issue that we need to solve is to identify proper hardware performance counters because of the differences in the hardware architectures: Intel Haswell Xeon in [21] vs. ARM Cortex-A15 in this work. On Intel platforms, each core's last-level cache miss counter is mapped to Linux kernel's *PERF_COUNT_HW_CACHE_MISSES* event, which is used by the regulator kernel module. On ARM Cortex-A15, however, each core's L1 cache miss counter is mapped to the same event, which results in inaccurate bandwidth measurement. We fix this issue by using L2 cache line refill counter, *L2D_CACHE_REFILL* of ARM Cortex-A15, based on the processor's reference manual [4]. Because the L2 cache is the last-level cache in Cortex-A15, this counter is able to accurately measure the memory traffic generated from the CPU cores. We have confirmed that the counter indeed correctly counts memory traffic by conducting a set of experiments using a synthetic benchmark from the IsolBench benchmark suite [18].

IV. Evaluation

In this section, we present the experimental evaluation results of BWLOCK++.

A. Setup

We evaluate BWLOCK++ on NVIDIA Tegra K1 platform. We use the Linux kernel version 3.10.40, which is patched with the changes required to support BWLOCK++. The CUDA runtime library version installed on the platform is 6.5, which is the latest version available for Tegra K1. In all our experiments, we place the platform in maximum performance mode by maximizing GPU and memory clock frequencies and disabling the dynamic frequency scaling of CPU cores. We also shutdown the graphical user interface and disable the LLC prefetcher to avoid run to run variation in the experiments.

In order to evaluate BWLOCK++, we use a selection of representative GPU benchmarks from the parboil suite [17] and OpenCV [2]. Table II show the brief descriptions of the selected benchmarks.

B. Effect of Memory Bandwidth Contention

In this experiment, we investigate the effect of memory bandwidth contention due to co-scheduled memory intensive CPU applications on the evaluated GPU kernels.

First, we measure the execution time of each GPU benchmark in isolation. We then repeat the experiment after co-scheduling three instances of a memory intensive CPU application as corunners. We use the the *Bandwidth* benchmark from the IsolBench suite [18] as the memory intensive CPU benchmark, which updates a big 1-D array sequentially. The sequential write access pattern of the benchmark is known to cause worst-case interference on several multicore platforms including the Tegra K1 [19]. The results of this experiment are shown in Figure 6 and they demonstrate how much the GPU benchmarks suffer from memory bandwidth contention due to the co-scheduled CPU applications.

From this figure, it can be seen that the worst case slowdwon, in case of histo benchmark, is more than 250%. Similary, for face detection benchmark, the worst case slowdown is more than 200%. For all other benchmarks, the slowdown is non-zero and can be significant in affecting the real-time performance. These results clearly show the danger

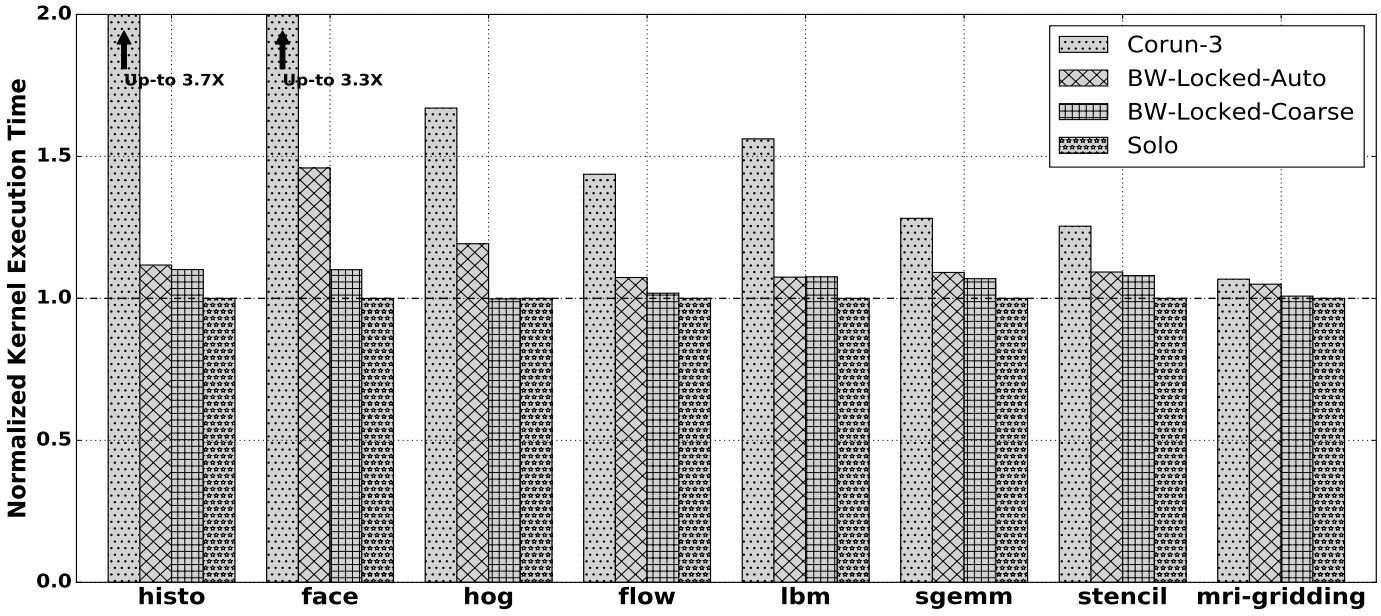


Fig. 7: BWLOCK++ Evaluation Results

Benchmark	Suite	Dataset	Description
Face	OpenCV	Video file (640 x 480 @ 25 fps)	Face detection using haar cascade classifiers
HOG	OpenCV	Video file (640 x 480 @ 25 fps)	Object detection using histogram of oriented gradients
Flow	OpenCV	Video file (640 x 480 @ 25 fps)	Detection of pattern of apparent motion of image objects
Histo	Parboil	Large	Computation of 2-D saturating histogram
LBM	Parboil	Short	Fluid dynamics simulation of an enclosed, lid-driven cavity, using the Lattice-Boltzmann Method
Sgemm	Parboil	Large	Register tiled matrix multiplication
Stencil	Parboil	Default	An iterative Jacobi stencil operation on a regular 3-D grid
MRI-Gridding	Parboil	Default	Computation of a regular grid of data representing an MR scan

TABLE II: Description of selected benchmarks

of uncontrolled memory bandwidth sharing in an integrated CPU-GPU architecture as GPU kernels may potentially suffer severe interference from co-scheduled CPU appliations. In the following experiment, we investigate how this problem can be addressed by using BWLOCK++.

C. Determining Memory Bandwidth Threshold

In order to apply BWLOCK++, we first need to determine safe memory budget that can be given to the CPU cores in the presence of GPU applications. However, an appropriate threshold value may vary depending on the characteristics of individual GPU applications. If the threshold value is set too high, then it may not be able to protect the performance of the GPU application. On the other hand, if the threshold value is set too low, then the CPU applications will be throttled more often that would result in significant CPU capacity loss. Therefore, we experimentally determine these threshold values on a per GPU application basis to find best trade-offs.

We calculate the safe memory budget for CPU cores by observing the trend of GPU application slowdown as the allowed memory usage threshold of CPU corunners is varied. Figure 8 shows this trend for the *histo* benchmark from the parboil suite. It can be seen that even a small change in the allowed memory usage threshold produces a significant change in the GPU benchmark execution; thus demonstrating that the performance of this application is extremely sensitive to the memory bandwidth. From this plot, we identify the threshold value which causes 10% slowdown of the GPU application and select that value as the allowed memory usage threshold for the CPU corunners. The 10% slowdown mark is arbitrarily selected and can be changed based on the application requirement. In this manner, we calculate the allowed memory usage thresholds for all the selected benchmarks. The selected threshold values, along with the corresponding GPU application slowdown, are mentioned in Table III.

Benchmark	Corun Threshold (MBps)	Slowdown @ Threshold
Histo	1	10%
Face	50	10%
LBM	50	8%
Stencil	100	9%
MRI-Gridding	100	5%
Flow	100	4%
Sgemm	200	7%
HOG	200	3%

TABLE III: Selected corun threshold values for GPU benchmarks

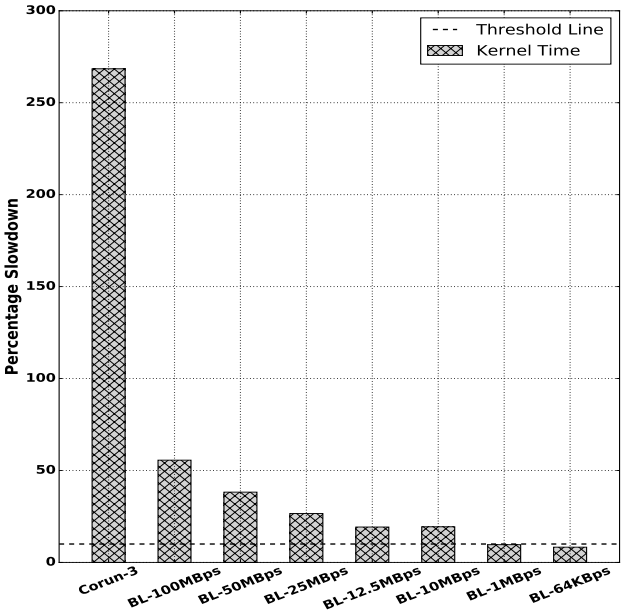


Fig. 8: Effect of corun bandwidth on the execution of histo benchmark

D. Effect of BWLOCK++

In this experiment, we evaluate the performance of BWLOCK++. Specifically, we record the corun execution of GPU benchmarks under two different versions of BWLOCK++. In *BW-Locked-Auto*, we use the automatic kernel protection mechanism explained in Section III-B. In *BW-Locked-Coarse*, we protect the entire execution of the GPU application by acquiring memory bandwidth lock before the application starts and releasing it once the application completes. We compare the performance under BWLOCK++ against the *Solo* and *Corun* execution of the GPU benchmarks which represent the measured execution times in isolation and together with co-scheduled memory intensive CPU applications, respectively.

To get the datapoints for *BW-Locked-Auto*, we configure BWLOCK++ according to the allowed memory usage threshold of the benchmark at hand and use our dynamic GPU kernel instrumentation mechanism to launch the benchmark in the presence of three *Bandwidth* benchmark instances (write memory access pattern) as CPU corunners. The final datapoints for *BW-Locked-Coarse* are obtained when the bandwidth lock is applied for the entire duration of the application. We use this final set as a baseline to determine how well our automatic instrumentation framework is working. As can be seen from the Figure 7, the execution under BWLOCK++ is within the acceptable performance margin (i.e., 10%) for all the GPU benchmarks. Also, the automatic instrumentation of BWLOCK++ provides almost near ideal performance when compared to coarse locking.

E. Throughput improvement with TFS

As explained in Section III-C, throttling under CFS results in significant system throughput reduction. In order to illustrate this, we conduct an experiment in which the GPU benchmarks are executed with six CPU corunners. Each CPU core, apart from the one executing the GPU benchmark, has a memory intensive application and a compute intensive application scheduled on it. For both of these applications, we use the *Bandwidth* benchmark with different working set sizes. In order to make *Bandwidth* memory intensive, we configure its working set size to be twice the size of LLC on our evaluation platform. Similarly for compute intensive case, we configure the working set size of *Bandwidth* to be half of the L1-data cache size. We record the total system throttle time statistics with BWLOCK++ for all the GPU benchmarks. We then repeat the experiment with our Throttle Fair Scheduling scheme. In *TFS-1*, we configure the TFS punishment factor as one for the memory intensive threads and in *TFS-3*, we set this factor to three. We plot the normalized total system throttle time for all the scheduling schemes and present them in Figure 9. It can be seen that TFS results in significantly less system throttling as compared to CFS.

V. Discussion

Our apprach has following limitations. First, we assume that CPU tasks are all best-effort and that no time critical real-time tasks can run on any of the CPU cores while the GPU is holding the bandwidth lock. Obviously, not all systems can satisfy this assumption. We claim that our approach is useful for situations where GPU accelerated tasks are critical, for example, vision-based automatic braking system. Also, the limitation can be mitigated, to a certain degree, by implementing a form of TDMA schedule among the CPU and GPU cores so that at any give time, either the CPU or the GPU can hold the bandwidth lock. This requires a mechanism to interrupt the execution of GPU kernels, which is possible in NVIDIA's recently release Pascal architecture based GPUs. We plan to explore such a coordinated scheduling mechanism as future work. Second, we assume that GPU applications are given a priori and they can be profiled in advance so that we can determine proper memory bandwidth threshold values. If this assumption can not be satisfied, an alternative solution is to use a single threshold value for all GPU applications, which eliminates the need of profiling. But the downside is that it may lower the CPU throughput because the memroy bandwidth threshold must be conservatively set to cover all types of GPU applications.

VI. Conclusion

In this paper, we presented BWLOCK++, a software based mechanism for protecting the performance of GPU kernels on platforms with integrated CPU-GPU architectures. BWLOCK++ automatically instruments GPU applications at run-time and insert a memory bandwidth lock, which throttling memory bandwidth usage of the CPU cores to protect performance of GPU kernels. We identified a side effect of memory bandwidth throttling on the performance of Linux default scheduler CFS, which results in the reduction of overall system throughput. In order to solve the problem,

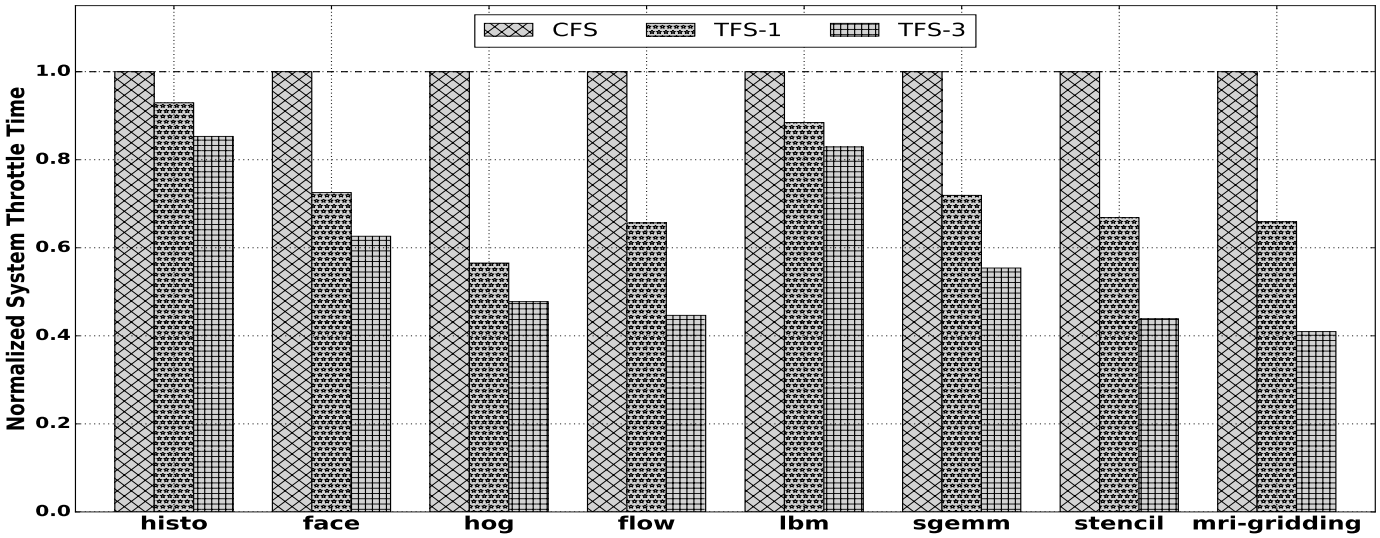


Fig. 9: Comparison of total system throttle time under different scheduling schemes

we proposed a modification to CFS, which we call Throttle Fair Scheduling (TFS) algorithm. Our evaluation results have shown that BWLOCK++ effectively protects the performance of GPU kernels from memory intensive CPU co-runners. Also, the results showed that TFS improves system throughput, compared to CFS, while protecting critical GPU kernels. In the future, we plan to evaluate BWLOCK++ on other integrated CPU-GPU architecture based platforms. We also plan to extend BWLOCK++ not only to protect critical GPU tasks but also to protect critical CPU tasks.

REFERENCES

[1] Nvidia jetson platforms. https://developer.nvidia.com/embedded-computing.

[2] Opencv. http://opencv.org.

[3] 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.

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

[5] N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna. SiGAMMA: Server Based GPU Arbitration Mechanism for Memory Accesses. In *International Conference on Real-Time Networks and Systems (RTNS)*, 2017.

[6] G. A. Elliott, B. C. Ward, and J. H. Anderson. Gpusync: A framework for real-time gpu management. In *Proceedings of the 2013 IEEE 34th Real-Time Systems Symposium*, RTSS '13, pages 33–44, Washington, DC, USA, 2013. IEEE Computer Society.

[7] B. Forsberg, A. Marongiu, and L. Benini. Gpuguard: Towards supporting a predictable execution model for heterogeneous soc. In *Design, Automation & Test in Europe (DATE)*, pages 318–321, 3001 Leuven, Belgium, Belgium, 2017. European Design and Automation Association.

[8] Greg Kroah Hartman. Modifying a Dynamic Library Without Changing the Source Code — Linux Journal. 2004.

[9] 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.

[10] 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.

[11] 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.

[12] H. Kim, P. Patel, S. Wang, and R. R. Rajkumar. A server based approach for predictable gpu access control. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017.

[13] I. Molnar. Modular scheduler core and completely fair scheduler. https://lwn.net/Articles/230501.

[14] N. Otterness, M. Yang, S. Rust, and E. Park. An evaluation of the nvidia tx1 for supporting real-time computer-vision workloads. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2017.

[15] N. Otterness, M. Yang, S. Rust, and E. Park. Inferring the scheduling policies of an embedded cuda gpu. In *Workshop on Operating Systems Platforms for Embedded Real Time Systems Applications (OSPERT)*, 2017.

[16] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '11, pages 269–279, Washington, DC, USA, 2011. IEEE Computer Society.

[17] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical report, University of Illinois at Urbana-Champaign, March 2012.

[18] 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.

[19] P. K. Valsan, H. Yun, and F. Farshchi. Addressing isolation challenges of non-blocking caches for multicore real-time systems. *Real-Time Systems*, 53(5):673–708, 2017.

[20] 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.

[21] 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.

[22] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[23] 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.