GPUguard: Towards supporting a predictable execution model for heterogeneous SoC

Funding acknowledgement:
688860 - High-Performance Embedded Real-time Architectures for Low-Power Many-Core Systems (SBFI)

GPUguard: Towards Supporting a Predictable Execution Model for Heterogeneous SoC

Björn Forsberg[1] Andrea Marongiu[1, 2] Luca Benini[1, 2]
[1] Swiss Federal Institute of Technology Zürich [2] University of Bologna
{bjoernf, a.marongiu, lbenini}@iis.ee.ethz.ch

***Abstract*—The deployment of real-time workloads on commercial off-the-shelf (COTS) hardware is attractive, as it reduces the cost and time-to-market of new products. Most modern high-end embedded SoCs rely on a heterogeneous design, coupling a general-purpose multi-core CPU to a massively parallel accelerator, typically a programmable GPU, sharing a single global DRAM. However, because of non-predictable hardware arbiters designed to maximize average or peak performance, it is very difficult to provide timing guarantees on such systems. In this work we present our ongoing work on GPUguard, a software technique that predictably arbitrates main memory usage in heterogeneous SoCs. A prototype implementation for the NVIDIA Tegra TX1 SoC shows that GPUguard is able to reduce the adverse effects of memory sharing, while retaining a high throughput on both the CPU and the accelerator.**

I. Introduction

In recent times it has become possible to perform heavy computations in a small power envelope through the integration of high-performing low-power accelerators on SoCs. As embedded SoCs are typically constrained in their area and power budgets, it is advantageous to share hardware components among several computational units. Thus, accelerated SoCs are typically designed as heterogeneous systems, sharing an off-chip (DRAM) memory. However, shared hardware resources are subject to contention when accessed by multiple devices, leading to variance in their access times.

In real-time systems, timing correct behavior must be ensured under all conditions. This leads to over-provisioning of the hardware to account for access time variability, causing poor utilization. On the other hand, the small-scale production of hardware capable of providing hard real-time guarantees typically drives costs well beyond those of their mass-produced general purpose counterparts. Because of this, the ability to deploy real-time workloads on commercial off-the-shelf (COTS) hardware becomes attractive.

As has been shown in several previous publications [1], the deployment of high-level software arbitration mechanisms can enable the execution of real-time workloads on COTS hardware. This paper presents our ongoing work in designing and implementing *GPUguard*, a high-level software abstraction layer which isolates access to shared DRAM in a heterogeneous GPU+CPU SoC (NVIDIA Tegra TX1) by enforcing time-separation between main memory access phases of CPU and GPU. To the best of our knowledge, *GPUguard* is the first scheme to apply such an approach in the context of commercial and large-volume heterogeneous SoC.

Experiments conducted on a prototype implementation of *GPUguard* show that the proposed scheme is able to deliver comparable throughput on both the CPU and GPU, while decreasing the memory latency variance.

II. Architectural Template

In the targeted heterogeneous SoC, the CPU is a multi-core processor with core-local private caches, and a shared last level cache (LLC). The GPU consists of one or more clusters of simple cores, with access to a per-cluster software managed scratchpad memory. Cores belonging to the same cluster can communicate via the shared scratchpad memory or the use of cluster-local barriers. Communication via cores in different clusters can only be done via the shared DRAM. All clusters within the GPU share a LLC. Within the cluster, cores may be sharing a single program counter, thus executing in lock-step.

Communication between the GPU and CPU complex can only be performed via the shared DRAM, thus the GPU is not capable of triggering CPU-side interrupts. Lastly, all hardware managed caches are managed by best-effort replacement policies, and their behavior inherently difficult to predict.

III. Background and Related Work

The Predictable Execution Model (PREM) [2] is designed to isolate access to shared main memory in multi-core CPUs. This is achieved by partitioning programs into *contention-sensitive* memory and *contention-free* computation phases, and scheduling these such that two memory phases are never executed in parallel. By scheduling only a single memory phase at a time, contention for memory is effectively avoided. This allows a system designer to tightly bound memory access latencies, leading to shorter worst-case execution times (WCET) and better use of the hardware. ***GPUguard is inspired by the PREM approach of separating programs into compute and memory phases, and uses the phase cycling of GPU programs as basis for system-wide memory scheduling.***

Extensions to the original PREM paper have mostly focused on multi-core CPUs [3]. PREM for heterogeneous SoCs has only been studied from the angle of WCET modeling [4].

A. Warp specialization

To realize PREM, applications must be divided into separate compute and memory phases. On GPUs, there exists a notion of division into PREM-like compute and memory phases in the staging of data through the local scratchpad. Typically, this

method is employed in programs with data re-use to avoid expensive re-fetching. It has been shown that GPU programs can be rewritten into two separate parts that individually handle these phases through *warp specialization* [5]. In the original work this is used to dedicate a subset of the GPU threads to emulating DMA engines to improve memory throughput. In the context of *GPUguard*, *warp specialization* provides a mechanism to further isolate the memory and compute phases into individually schedulable units. Furthermore, it also enables fine-tuning memory and compute phase lengths by decoupling the allocation of memory and compute threads.

B. MemGuard

PREM on heterogeneous architectures must also provide isolation between CPU and GPU. On multi-core CPUs, *MemGuard* [6], enforces per-core or per-task bandwidth budgets, and can be employed to prevent the CPU from using main memory during GPU memory phases. *MemGuard* uses performance counters to detect when a core overruns its memory budget, at which point a high-priority *throttle thread* is scheduled, thus pre-empting the running task. The *throttle thread* performs busy waiting, and thus ensures that no further memory accesses are generated. At preset intervals the memory budget is *renewed*, and the *throttle thread* is put back to sleep. Through the use of performance counters, tasks may continue to execute when the memory bandwidth is throttled, as long as they keep hitting in the local cache. Furthermore, no program changes are necessary to support legacy applications, as the budgeting mechanism is managed on a higher system level. *In this work, in combination with the PREM phase divisioning between CPU and GPU, we use the* MemGuard *bandwidth limiting mechanism for internal bandwidth control on the CPU*. To achieve this, we put the bandwidth budget renewal of *MemGuard* under the control of *GPUguard*.

IV. GPUguard

A. Time-sharing memory access between CPU and GPU

While it is possible to choose either the CPU or the GPU as the controlling party, GPUs generally do not contain event mechanisms such as interrupt lines. Their only way to react to external events, such as a CPU synchronization, is to resort to expensive polling. Therefore, *GPUguard* controls access to the shared global DRAM based on the natural shifting in phases of the GPU program, i.e., when the scratchpad memory has been filled or the computations on that data completed, respectively. It is beneficial to switch at these border regions, since interruptions in the memory phase would stall the execution of the program.

The ability to independently allocate compute and memory threads, as provided by the warp specialization approach, provides a mechanism to balance the length of the corresponding phases. To provide guarantees on available memory access windows the lengths of the intervals are kept constant by enforcing the upper bound on the execution time of the phases[1]. This is in line with previous PREM approaches.

B. Memory bandwidth control mechanism

The matching of PREM-phases to the memory-compute cycling of warp-specialized applications implicitly ensures that the GPU application does not perform memory accesses outside its memory phase. The CPU on the other hand has no such mappings, and thus another mechanism must be put in place to ensure that memory access isolation is upheld.

In the light of extending PREM, the CPU can locally use the traditional CPU-only implementation of PREM [2] to schedule tasks. Thus, during the GPU compute phases, the memory phases of CPU applications would be scheduled, and the computation deferred to the GPU memory phase, and the isolation properties would be upheld. This requires that the CPU-side programs conform to the PREM phase division. The full formalization of this approach is left as future work.

To support legacy applications without modification, *GPUguard* can be coupled with *MemGuard* by allowing the CPU to freely access global memory (i.e., no LLC miss limits) during the GPU compute phases, and once the GPU switches into memory mode, a bandwidth limitation is put in place using the *MemGuard* performance counter mechanism. This means that the CPU program can continue to execute as long as it keeps hitting in the local caches, while still supporting a preset amount of global loads until the bandwidth limit put in place by *MemGuard* is reached. By combining these two mechanisms, the isolation property is once again upheld. In the following sections, we assume that the *MemGuard* mechanism is used on the CPU, and that the bandwidth renewal process is under the control of *GPUguard*. This is reflected by *Sleep* in Fig. 1, which gives an overview of the *GPUguard* prototype.

C. Global memory-compute phase synchronization

To schedule access windows to the global memory, *GPUguard* employs a synchronization scheme: On every phase change the GPU writes a sync flag (*Sync*, Fig. 1) into a segment of memory visible to both CPU and GPU. After this, the GPU stalls until the flag has been unset, which signifies that the phase shift has been acknowledged by the CPU. To ensure that the CPU can execute jobs in parallel, the CPU is not polling for the GPU sync flag, but only acts on the synchronization once the preset length of the GPU phase has passed. This is achievable through the use of timer interrupts (*Timer*, Fig. 1). The CPU acknowledges the synchronization once the *MemGuard* bandwidth limitation is in place (*Wake*, Fig. 1), to ensure that the CPU is effectively hindered from overrunning its budget.

The interval during which a processing unit experiences one memory and one compute phase is referred to as the *GPUguard period*. The *period* is assumed to be defined at a per-kernel granularity, and thus does not change during the GPU-side execution.

[1]While *double buffering* is better for performance, *single buffering* provides the clearest division of memory and compute phases, and is the favored buffering technique for *GPUguard*. However, for compute bound applications, it may be possible to perform double buffering as well.

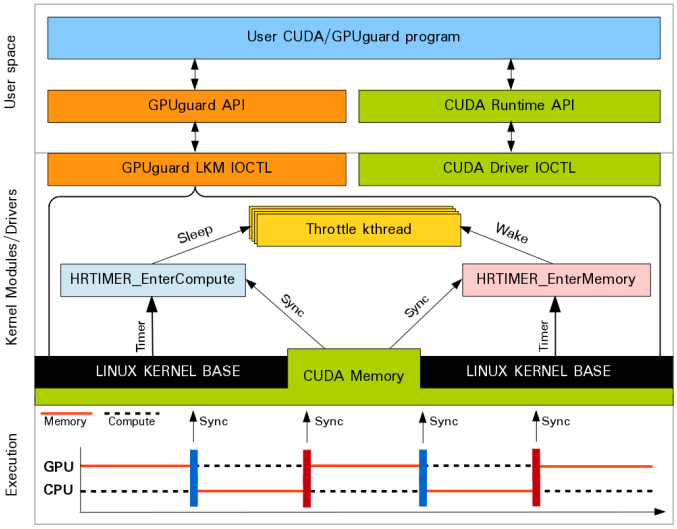

Fig. 1. An architectural overview of the implemented *GPUguard*.

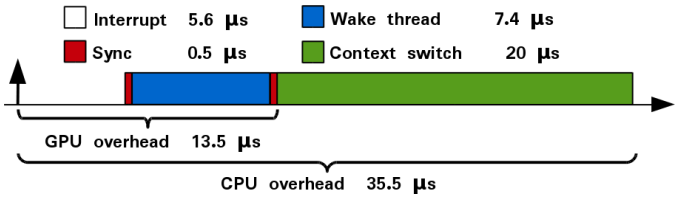

Fig. 2. Measured latencies of the different steps involved in performing a synchronization.

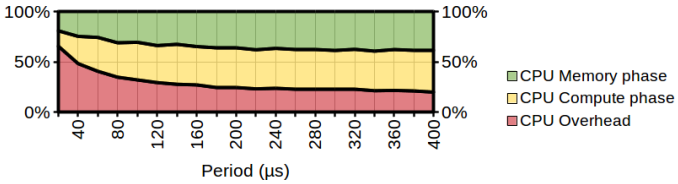

Fig. 3. The division of CPU phases under different *period* lengths in relation to the CPU synchronization overhead.

V. Evaluation

GPUguard is evaluated on the NVIDIA Tegra TX1 [7], a state-of-the-art COTS heterogeneous SoC. We quantify the different sources of overhead, and validate that *GPUguard* reduces the variability in memory access times.

A. Prototype implementation

We have implemented an evaluation prototype of *GPUguard*, split into a Linux loadable kernel module (LKM) and an API implemented in CUDA. Since *GPUguard* relies on warp specialization for the separation of compute and memory phases, the GPU-side API presents a similar interface as CudaDMA [5]. On the CPU, a call to the non-blocking `HostSync()` function starts the synchronization mechanism within the LKM. The non-blocking nature of the call allows the CPU to continue local execution while the kernel is running, as it would in any legacy CUDA program.

To ensure that the execution of the compute phases only rely on local resources, the memory phase will pre-fetch the required input data into the scratchpad memory, or *shared memory* in CUDA terminology. On the targeted platform, this amounts to $48KB$ of memory per *thread block*. *GPUguard* targets heterogeneous SoC without discrete memories, thus all copy operations can be instantiated from the GPU itself, and are subject to the *GPUguard* orchestrated memory access schedule. Fig. 1 shows the implemented prototype.

B. OS and hardware characterization

To gain an understanding of the overheads imposed by the synchronization scheme, we have characterized the three main sources of overhead: First, the *GPU synchronization overhead* is amount of time the GPU stalls while waiting for the synchronization acknowledgment from the CPU. Second, the *CPU synchronization overhead* is the degradation in performance experienced on the CPU due to the frequent interrupts induced by the synchronization. Third, the *period overhead* is the cost of stalls due to bad matches between the *GPUguard periods* and the actual execution time of the GPU phases.

As an initial characterization of both the CPU and GPU synchronization overheads, the time required for all synchronization steps were measured. This includes the Linux timer interrupt latency[2] (*interrupt*), the memory latency for writing the synchronization flags (*sync*), the time it takes to wake the *throttle thread* (*wake*), and the cost of context switching to the *throttle thread* once it has been scheduled (*context switch*). The results are presented in Fig. 2.

To evaluate the CPU overhead, we created a synthetic benchmark which only performs synchronizations, and measured the available processing and memory time on the CPU. For this experiment, we assign 50% of the memory bandwidth to the CPU. However, as the results in Fig. 3 shows, the actual un-throttled memory time available to the CPU is only about 40%, and decreases with the *period* length. This is because a shorter *period* implies more interrupts and more context switches, which is the source of CPU overhead. As *GPUguard* only throttles the memory bandwidth, a compute-intensive CPU task can continue execution during the entire *period*, as long as it does not require main memory access. On the other end of the spectrum, a memory-intensive CPU task will be subject to bandwidth throttling during the CPU compute phase, and in the extreme case it will be stalled until the next CPU memory phase. Thus, depending on the *compute-to-communication ratio* of the CPU tasks, and the bandwidth limit put in place through the *MemGuard* throttling mechanism, the available time for performing useful work for a specific task will be somewhere between these two extremes.

The GPU overhead, presented in Fig. 2, is smaller than that of the CPU, as the GPU does not perform any additional operations once the synchronization is done. However, as the waking of the *throttle thread* is done before the CPU

[2]Mainline Linux provides no guarantees on interrupt latency, although there are patches [8] that aim to provide this.

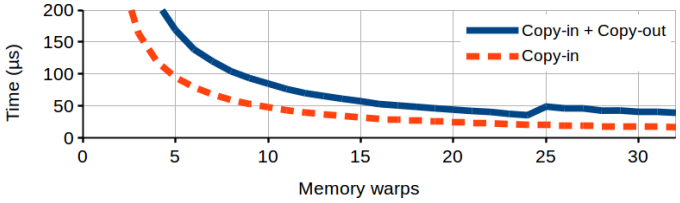

Fig. 4. The time required to fill, and fill + empty, the scratchpad memory on the GPU using different numbers of memory warps.

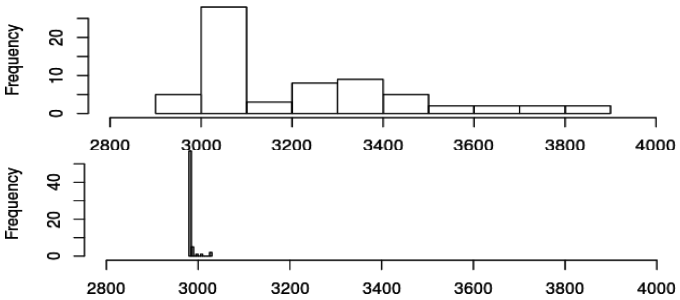

Fig. 5. The distribution of execution times for the Matrix Multiplication kernel under contention. The top histogram shows the variance in the baseline CUDA version, and the bottom one shows the variance when *GPUguard* is used to orchestrate memory accesses.

acknowledges the GPU, it is included in the GPU overhead.

The final source of overhead is due to the over-dimensioning of the *GPUguard period*. This overhead appears when the *GPUguard period* is longer than the actual execution time of the GPU phases, causing the GPU to stall while waiting for the CPU to enter the synchronization phase. To gain an insight into the optimal length for the GPU memory *period*, we measured the time required to populate the scratchpad memory using different amount of memory warps. We also measured the time to both populate and evict the scratchpad data, as this constitutes typical program behavior. The measured times are presented in Fig. 4. For most warp configurations, the time required to fill the shared memory is in the range where the CPU-side experiences heavy degradation due to synchronization overhead, see Fig. 3. This means that memory bound programs, i.e., programs that perform extensive searching or data traversal in relation to the length of the computation phase are susceptible to stalls. However, many classical GPU applications, such as BLAS (Basic Linear Algebra Subroutine) kernels, are compute intensive, and thus not affected by this limitation. We therefore employ a compute heavy program, matrix multiplication, a BLAS3-type kernel for the initial evaluation of *GPUguard*.

C. GPU Performance and Predictability characterization

To evaluate the achieved reduction in memory access time variance, we have implemented a *GPUguard* enabled version of matrix-matrix multiplication, a BLAS3 kernel. This is compared to a baseline version taken from the CUDA samples provided by NVIDIA. Matrix Multiplication has a compute to communication ratio of $2n^3/3n^2$, i.e., the computation part is cubic in the input size, while the communication is only quadratic. Thus, as the input size increases, the computational work increases asymptotically faster than the memory requirements, leading to fewer synchronizations due to the scratchpad memory size. Furthermore, we set the CPU bandwidth limit to zero during the GPU memory phases, thus stalling all memory accesses from the CPU until the end of the phase. This represents the maximum achievable isolation and the best case for increasing timing predictability.

We execute the baseline and *GPUguard*-enabled kernels over several iterations under memory contention generated by the CPU, and plot their execution time distributions in Fig. 5. As can be seen in the figure, the execution time variance in the *GPUguard* enabled versions of the BLAS kernels is near zero, in contrast to the baseline implementations. In addition to this, the execution time of the *GPUguard*-enabled kernel is within the range of execution times exhibited by the CUDA baseline implementation.

VI. Conclusion

We have presented *GPUguard*, a PREM-like method of dividing GPU programs into *contention-sensitive* memory and *contention-insensitive* computation phases, and mechanism for globally scheduling these to minimize memory interference between the GPU and CPU. We have characterized the main overheads as well as performed an initial validation of the reduction in memory access time variability. The source code will be available as part of the Hercules 2020 EU-project [9].

Our future work will focus on designing and implementing more lightweight synchronization schemes, more relaxed memory phase isolation policies, and the mathematical formalization of the approach. We will also implement a compiler to automate the creation of GPUguard-enabled programs.

Lastly, we are also looking into using hardware-level throttling in the memory controller, which promises much lower CPU-side overheads compared to Linux software solutions.

VII. Acknowledgment

This work has been supported by the EU H2020 project HERCULES (688860).

References

[1] S. Girbal et al. Deterministic platform software for hard real-time systems using multi-core COTS In Digital Avionics Systems Conference (DASC), 2015 IEEE/AIAA 34th, 2015.

[2] R. Pellizzoni et al. A Predictable Execution Model for COTS-Based Embedded Systems In *17th IEEE Real-Time and Embedded Technology and Applications Symposium, Chicago, IL*, pp. 269-279. 2011.

[3] A. Alhammad et al. Time-predictable execution of multithreaded applications on multicore systems Proceedings of the conference on Design, Automation & Test in Europe. 2014.

[4] P. Burgio et al. A memory-centric approach to enable timing-predictability within embedded many-core accelerators In Real-Time and Embedded Systems and Technologies (RTEST), 2015 CSI Symposium on. 2015.

[5] M. Bauer et al. CudaDMA: optimizing GPU memory bandwidth via warp specialization SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. 2011.

[6] H. Yun et al. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th, Philadelphia, PA, pp. 55-64. 2013.

[7] Jetson TX1 Embedded System Developer Kit http://www.nvidia.com/object/jetson-tx1-dev-kit.html

[8] Real-time Linux Wiki https://rt.wiki.kernel.org/index.php/Main_Page

[9] The Hercules Project, http://hercules2020.eu/