On the Cost of Freedom From Interference in Heterogeneous SoCs

Björn Forsberg, Luca Benini, Andrea Marongiu

Abstract

In heterogeneous CPU+GPU SoCs where a single DRAM is shared between both devices, concurrent memory accesses from both devices can lead to slowdowns due to memory interference. This prevents the deployment of real-time tasks, which need to be guaranteed to complete before a set deadline. However, freedom from interference can be guaranteed through software memory scheduling, but may come at a significant cost due to frequent CPU-GPU synchronizations. In this paper we provide a compile-time model to help developers make informed decisions on how to achieve freedom from interference at the lowest cost.

1 Introduction

In heterogeneous systems, the CPU and one or more accelerators are typically sharing a single DRAM. As memory accesses constitute a critical point for every process, heavy contention for the shared DRAM is bound to introduce significant delays in every process of the system. This makes the deployment of real-time workloads on such systems troublesome, as they must be dimensioned such that every task can finish before its *deadline*, even under the worst case interference. Budgeting for the worst case interference in a heterogeneous system is bound to introduce very pessimistic delays.

To enable freedom from interference in heterogenous systems, it must be guaranteed that the CPU and the GPU will not access the shared DRAM in parallel. However, as there is no possibility to interrupt the GPU during execution, the launching of a GPU kernel would by default block the CPU from accessing memory for a potentially prohibitive amount of time. Therefore, mechanisms to control the GPU memory accesses at a finer granularity are required.

A known software-based method for fine-grained memory scheduling in multi-core systems is the Predictable Execution Model (PREM) [1, 2]. PREM-style memory scheduling is enabled by separating tasks into fine-grained *Memory* and *Compute phases*. The memory phase is programmed to prefetch into local memory all data required by the Compute phase, guaranteeing that the latter will never experience a cache miss, and will therefore never access the shared DRAM. By further only allowing a single memory phase to execute at a time within the system, freedom from interference is effectively achieved.

Recently, PREM has been proposed for use in heterogeneous SoCs with integrated GPUs [3]. To enable PREM memory scheduling in heterogeneous systems, a CPU-GPU synchronization infrastructure is required, to be able to co-schedule the memory accesses of both devices. However, a possible drawback of the approach is the potentially high cost of the fine granularity of memory scheduling. On the GPU the available local storage is small. If the execution time of useful work that can be done on the local data is dominated by the cost of memory scheduling, a PREM-enabled kernel will experience large slowdowns.

On the other hand, executing the full kernel to completion has no fine-grained synchronizations and does not experience these slowdowns, but may block the CPU for a prohibitive amount of time. Thus, there exists a *tradeoff* between the costs of fine- vs. coarse-grained memory scheduling to achieve freedom from interference.

The creation of PREM-compliant code is a tedious and error-prone process, and is better left to a compiler [1, 2, 4]. Such a compiler was recently presented for GPU-equipped heterogeneous SoCs

[5], performing the separation into PREM Memory and Compute phases. As the compiler is able to analyze the characteristics of the GPU kernel, it could be used to predict the cost to achieve freedom from interference, and provide the developer with important information to make the correct *tradeoff* decision.

In this paper we present a model to determine the cost of freedom from interference based on kernel information extracted during compilation. We validate the results against measured execution times on the NVIDIA Tegra TX1 [6] heterogeneous SoC, which features a 4-core ARM A57 CPU and a 2-cluster 128-core Maxwell GPU.

The rest of the paper is structured as follows: Section 2 introduces the underlying compiler, followed by the model in Section 3. Section 4 discusses a possible use case of the presented model, and Section 5 concludes.

2 Enforcing PREM through compilation and Runtime support

The considered PREM compiler [5] operates on high-level programming languages, such as OpenMP, in which GPU kernels are described through the annotation of *parallel loops*. As parallel loops can be statically analyzed during compilation, it is possible to determine the memory accesses of the GPU kernel, and transform the loop to operate on *tiles* that only access so much data as fits in the local storage. This *tiling* forms the basis for the PREM memory and compute phases. All prefetches of the memory phase are directed to the software managed scratchpad memory (CUDA shared memory), as this does not suffer from the eviction effects that are common in hardware-managed caches.

To *PREM-ize* the code, each *tile* is cloned into three copies, which are then transformed to perform one of three functions: (i) **Load** – to prefetch all the data required for the execution of the tile from global DRAM to the local scratchpad memory; (ii) **Execute** – where all memory accesses of the original code are changed to use the data loaded to the scratchpad memory; (iii) **Store** – to write back all data to the global DRAM upon execution completion.

We refer to this compilation scheme as PREM LES. The **Execute** implements the PREM Compute phase, while the **Load** and **Store** codes together implement the PREM memory phase.

To ensure that the Memory phase of the GPU kernel is not executed while the CPU is accessing memory, the memory accesses of both CPU and GPU must be coordinated. This is achieved by using a CPU-GPU synchronization infrastructure, that allows a memory access token to be passed between the CPU and the GPU [3]. The device that is currently holding the token is allowed to execute memory phases, and thus access the global DRAM. The token is passed between the device in accordance with a time-triggered schedule. As the synchronization comes with a cost, kernels that are not able to perform a sufficient amount of work on the data that is loaded to the scratchpad will be dominated by synchronization and see a significant performance degradation.

3 Compile-time prediction of Runtime performance

This section describes the modeling of the runtime performance of the compiled kernels, and verifies the results against measured values. The evaluation is performed on benchmarks from the PolyBench-ACC benchmark suite [7] on both available GPU clusters.

3.1 Compiler analysis and hardware parameters

3.1.1 Understanding the hardware

Hardware-specific operation latencies considered in our approach consist of (a) arithmetic operations $l_{arithmetic}$, (b) scratchpad accesses l_{SPM}, and (c) DRAM accesses l_{DRAM}. For arithmetic operation and scratchpad access latencies we refer to data reported by the hardware vendor [8]. To determine the DRAM latency, a synthetic benchmark was executed under several configurations, and the average latencies extracted.

To determine the DRAM latency, two high-level classes were found to influence the latency: a) the

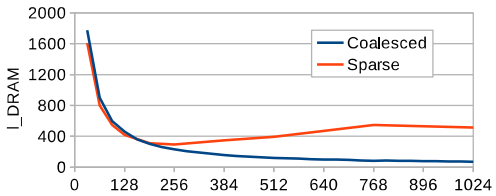

Figure 1: The difference in time it takes to load fully sequential and non-sequential data into the scratchpad (49152 bytes).

number of threads that are used, as this influences the bandwidth usage, and b) the memory access pattern.

For the memory access pattern two configurations are used, one in which all memory accesses are *coalesced*, i.e., threads executing concurrently will touch data on consecutive addresses, and one in which the accesses are sparse, i.e., every thread will need to fetch a unique cache line to satisfy its memory request. Based on these results, we further divide l_{DRAM} into the latency for coalesced memory accesses $l^{coal}_{DRAM}$and the latency of sparse memory accesses l^{sparse}_{DRAM}, both of which depend on the *blockDim*, i.e., the number of threads used for the kernel. The result for this experiment is presented in Figure 1.

3.1.2 Compile-time identification of instructions by latency

From the compiled code, the corresponding operations at each latency can be extracted. Due to the loop-based offloading in high-level languages, the compiler is able to distinguish coalesced from sparse accesses to arrays through scalar evolution analysis, as each value of the induction variable (IV) of the offloaded loop maps to a specific thread[1]. By analyzing the evolution of the IV it is thus possible to determine which thread will perform each access, and by extension, if the access is part of a coalesced pattern. By iterating over the instructions, the compiler can therefore count the number of arithmetic operations C, and the number of coalesced M_{coal} and sparse M_{sparse} memory accesses. These operations map to the identified latencies of $l_{arithmetic}$, $l^{coal}_{DRAM}$and l^{sparse}_{DRAM}. In LES, these accesses will also be performed in the scratchpad memory, at the corresponding l_{SPM} latency. Furthermore, due to the use of local storage, LES will only have to load the data to the scratchpad once, even if it is used multiple times during the computation. Thus, the compiler also keeps track of the unique memory accesses U_{coal} and U_{sparse}. While the M values can be found by iterating over the instructions, to determine the unique memory accesses U, the scalar evolution and alias analysis frameworks of LLVM are employed.

[1]Assuming static scheduling.

3.2 Estimating the Performance of PREM Load Execute Store

When estimating the performance of the LES transformations, both the cost of executing the PREM phases T_{Phases}, and the time required for the synchronization T_{Sync} must be taken into account. As the LES code consists of two parts, T_{Phases} is further split up into the individual execution times for the Compute and Memory phases such that $T_{Phases} = T_{Memory} + T_{Compute}$.

3.2.1 Execution time of PREM Phases

For the LES transformation, the memory phase execution time is dependent on both the data movements from DRAM, as well as the cost for accessing the scratchpad memory through which the data is staged. Because of this, the latency of both memories is taken into account in this phase. As data is only loaded once only the unique accesses U are considered. The execution time of the computation is only considered in the Compute phase, and as its accesses are performed on the scratchpad memory, only these latencies need to be considered. We model the execution time of each phase as shown in Equation 1.

$$\begin{aligned} T_{Memory} = {} & U_{coal} \times l^{coal}_{DRAM} + U_{coal} \times l^{coal}_{SPM} + \\ & U_{sparse} \times l^{sparse}_{DRAM} + U_{sparse} \times l^{sparse}_{SPM} \\ T_{Compute} = {} & C \times l_{arithmetic} + M_{coal} \times l^{coal}_{SPM} + \\ & M_{sparse} \times l^{sparse}_{SPM} \end{aligned} \tag{1}$$

3.2.2 Synchronization cost

The cost of performing the synchronization S with the host at the end of each PREM interval is a system-dependent parameter, in this case equal to

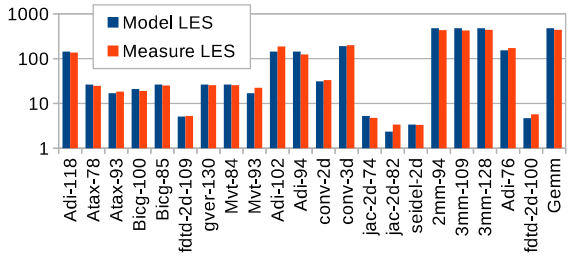

Figure 2: Validation of the modeled LES execution times against measurements on real hardware.

5.6μs. As both the Load and Store phases are executed in a PREM Memory phase, only two CPU-GPU synchronizations are required per interval, and therefore $T_{Sync} = 2 \times S$. Because the synchronization are triggered by CPU timers, there is also a minimum granularity $G_{sync} = 10\mu s$ at which the timers can be triggered, and thus each phase is forced to execute for at least this time (idling if the phase is shorter). The full LES interval execution time is as shown in Equation 2.

$$\begin{aligned} T_{LES} = & \, max(T_{Memory}, G_{sync}) + \\ & max(T_{Compute}, G_{sync}) + \\ & 2 \times S \end{aligned} \tag{2}$$

As the model calculates the execution time of a tile in LES the execution time of each tile needs to be added up to produce the full kernel execution time. This value is already determined by the compiler to be able to produce the tiling loops.

The modeled execution time is validated against measured values, and the results are presented in Figure 2. It can be seen that the predicted values follow quite accurately the measured ones (the error is on average below 10%).

3.3 Estimating Performance of the Original Kernel

To be able to estimate the cost of the LES transformation, the original kernel must be used as a baseline, and therefore its execution time must also be estimated.

To this aim, we rely on the model captured by Equation 3.

$$\begin{aligned} T_{Unmodified} = & \, R \times \\ & (M_{coal} \times l^{coal}_{DRAM} + \\ & M_{sparse} \times l^{sparse}_{DRAM}) + \\ & C \times l_{arithmetic} \end{aligned} \tag{3}$$

In contrast to the LES kernel, in which the memory phase accesses were organized into well-behaved pattern, and duplicate accesses are deferred to the scratchpad, the original kernel will perform every access through the cache hierarchy to global memory. This requires the model for the baseline to make predictions about the cache behavior of kernels. To capture this, a new parameter R is introduced, which provides a scaling factor for the experienced DRAM latency based on the cache performance. We empirically found that to capture this effect the benchmarks can be grouped in three classes, based on their access patterns.

- **Cache-friendly**: For kernels which perform most of their accesses in a coalesced pattern, the unmodified kernel may benefit from a higher degree of temporal data reuse compared to the average latency experiments (see Figure 1). These patterns can be easily detected in the compiler by looking at the indexing functions for the outermost loop nests (e.g., when thread IDs are used for indexing at this level the resulting pattern generates coalesced accesses).
- **Neutral**: For these kernels, no adjustments is necessary, as they conform to the average latencies from the original experiment.
- **Cache-unfriendly**: The average DRAM latencies l_{DRAM} only consider accesses from a single data structure, and thus fail to take into account the self-eviction that occurs between multiple data structures when accessed repeatedly (i.e., not only once as in LES). This pattern is identified in the compiler by determining the number of distinct data structures that are accessed sparsely, and already at three or more, the self eviction effect becomes significant.

Figure 3 shows the grouping of the various benchmarks in these three classes, and the

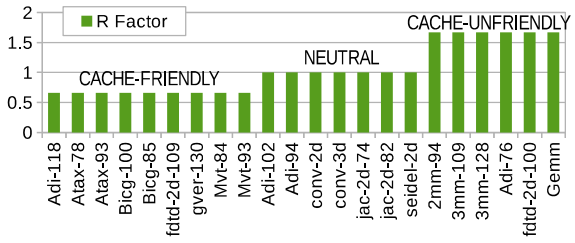

Figure 3: The memory adjustment factor R applied to each benchmark.

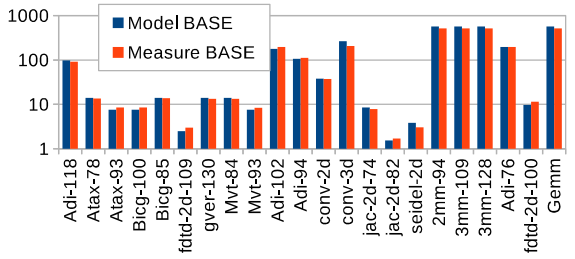

Figure 4: Comparison of the full execution time of the baseline program with the times estimated by the model.

empirical value for the R parameter for each class ($R_{cache-friendly} = 0.66$; $R_{neutral} = 1.0$; $R_{cache-unfriendly} = 1.67$).

As with LES, the model calculates the execution time of a tile-equivalent amount of work, and the execution time is added up to produce the full kernel execution time.

The modeled execution time is validated against measured values, and the results are presented in Figure 4. Also in this case, the predicted values follow quite faithfully the measured ones, the average error being below 10%.

4 Discussion

From the modeled results, we can see that the cost for LES varies heavily within the PolyBench-ACC suite, as illustrated by the LES-only column in Table 1. The column shows the slowdown factor of the LES kernel compared to the original (1.0 means the same execution time, less than 1.0 is an improvement). However, the kernels that show bad behavior under LES are also relatively short-lived, and with this knowledge it is possible to reserve the memory for the GPU during the full kernel execution. In this case, no LES transformation is applied, and the execution time is unchanged (1.0). In the two right-most columns of the table, we apply two different upper bounds on how long we are willing to reserve the memory for the GPU, and if the original kernel is shorter *and* LES shows a slowdown, we choose to go with the original kernel. We refer to this value as the $T_{max\_GPU\_reserve}$. In doing this, the average slowdown for the entire set of benchmarks is reduced from 44% for LES-only to 21% if $10ms$ of CPU blocking is allowed, and 0.5% if $20ms$ is allowed.

In the two left-most columns, the benchmarks that can be executed on the $T_{max\_GPU\_reserve}$ are marked by 1.0 (original execution time), and kernels that would block the CPU for longer than that are left blank. Only 7 kernels can be supported at $10ms$, and is increased to 12 at $20ms$. For all other kernels finer-grained memory scheduling, i.e., LES, is required.

Thus, these two approaches complement each other, and the information required for an informed tradeoff can be generated at compile-time. At both memory protection granularities, freedom from interference can be guaranteed through the synchronization infrastructure, enabling low-cost freedom from interference on heterogeneous CPU+GPU SoCs.

5 Conclusion

We have presented a compile-time model to estimate the cost of fine- and coarse-grained memory control in heterogeneous CPU + GPU systems. Through this information, a system developer can make informed decision on how to achieve freedom from interference with the lowest cost.

Acknowledgment

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

References

[1] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, “A

Kernel	Per-offload only $T_{max\_GPU\_reserve}$ $10ms$	$20ms$	LES only	Tradeoff $T_{max\_GPU\_reserve}$ $10ms$	$20ms$
Adi-118	–	–	1.50×	1.50×	1.50×
Atax-78	–	1.0×	1.84×	1.84×	**1.0×**
Atax-93	1.0×	1.0×	2.17×	**1.0×**	**1.0×**
Bicg-100	1.0×	1.0×	2.25×	**1.0×**	**1.0×**
Bicg-85	–	1.0×	1.86×	1.86×	**1.0×**
fdtd-2d-109	1.0×	1.0×	1.78×	**1.0×**	**1.0×**
Gemver-130	–	1.0×	1.93×	1.93×	**1.0×**
Mvt-84	–	1.0×	1.93×	1.93×	**1.0×**
Mvt-93	1.0×	1.0×	2.66×	**1.0×**	**1.0×**
Adi-102	–	–	0.96×	0.96×	0.96×
Adi-94	–	–	1.11×	1.11×	1.11×
conv-2d	–	–	0.90×	0.90×	0.90×
conv-3d	–	–	0.96×	0.96×	0.96×
jacobi-2d-74	1.0×	1.0×	0.61×	0.61×	0.61×
jacobi-2d-82	1.0×	1.0×	1.98×	**1.0×**	**1.0×**
seidel-2d	1.0×	1.0×	1.10×	**1.0×**	**1.0×**
2mm-94	–	–	0.86×	0.86×	0.86×
3mm-109	–	–	1.30×	0.86×	0.86×
3mm-128	–	–	0.86×	0.86×	0.86×
Adi-76	–	–	0.88×	0.88×	0.88×
fdtd-2d-100	–	1.0×	0.51×	0.51×	0.51×
Gemm	–	–	0.86×	0.86×	0.86×
		Average:	1.44×	1.21×	1.005×

Table 1: The cost of freedom from interference relative to the original unmodified program (with no freedom from interference guarantees) at different CPU blocking time thresholds.

predictable execution model for cots-based embedded systems," in *RTAS'11*. IEEE, 2011.

[2] A. Alhammad and R. Pellizzoni, "Time-predictable execution of multithreaded applications on multicore systems," in *DATE'14*. IEEE, 2014.

[3] B. Forsberg, A. Marongiu, and L. Benini, "Gpuguard: Towards supporting a predictable execution model for heterogeneous soc," in *DATE'17*, 2017.

[4] M. R. Soliman and R. Pellizzoni, "WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching," in *ECRTS'17*, 2017.

[5] B. Forsberg, L. Benini, and A. Marongiu, "Heprem: Enabling predictable gpu execution on heterogeneous soc," in *DATE'18*, 2018.

[6] "NVIDIA Jetson TX1 Developer Kit." [Online]. Available: http://www.nvidia.com/object/jetson-tx1-dev-kit.html

[7] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 Innovative Parallel Computing (InPar)*, 2012.

[8] NVIDIA, "Cuda c programming guide v9.1.85," March 2018. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/