GPU-Accelerated Real-Time Path Planning and the Predictable Execution Model

Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 108C (2017) 2428–2432

Procedia Computer Science

International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland

GPU-Accelerated Real-Time Path Planning and the Predictable Execution Model

Björn Forsberg[1], Daniele Palossi[1], Andrea Marongiu[1,2], Luca Benini[1,2]

[1)] IIS – ETH Zürich, [2)] DEI – University of Bologna

Abstract

Path planning is one of the key functional blocks for autonomous vehicles constantly updating their route in real-time. Heterogeneous many-cores are appealing candidates for its execution, but the high degree of resource sharing results in very unpredictable timing behavior. The predictable execution model (PREM) has the potential to enable the deployment of real-time applications on top of commercial off-the-shelf (COTS) heterogeneous systems by separating compute and memory operations, and scheduling the latter in an interference-free manner. This paper studies PREM applied to a state-of-the-art path planner running on a NVIDIA Tegra X1, providing insight on memory sharing and its impact on performance and predictability. The results show that PREM reduces the execution time variance to near-zero, providing a 3× decrease in the worst case execution time.

Peer-review under responsibility of the scientific committee of the International Conference on Computational Science

Keywords: Heterogeneous Computing, GPGPU, Predictable Execution Model, Path Planning

1 Introduction

The interest in autonomous vehicles is growing constantly, with lots of practical applications appearing on the marketplace. Notable examples are unmanned aerial vehicles (UAVs) [1] and autonomous driving systems [2]. It is tempting to rely on integrated heterogeneous many-cores [1, 3] for the practical deployment of such workloads, given their high computational requirements, but their real-time requirements constitute an obstacle. Indeed, virtually all industrial real-time systems are based on single-core devices, for which it is much easier to derive solid guarantees about their timing behaviour. State-of-the-art timing and schedulability analyses rely on worst case execution time (WCET) [4] as a pessimistic upper-bound to task durations. In contrast to single-core processors, integrated heterogeneous devices share main memory (DRAM) between several actors, which greatly complicates WCET calculation. [5].

Custom-designed, real-time hardware [6] could tighten WCET bounds, but with lower performance and at considerably higher cost than general purpose systems. This has motivated research aimed at making the deployment of real-time systems on commercial off-the-shelf (COTS) hardware possible [7, 3]. Common approaches for doing this include enforcement of strict per-core budgets of a shared resource [8], or separation of the program into multiple *phases* that can be individually scheduled based on their use of shared resources [9, 10].

1877-0509
Peer-review under responsibility of the scientific committee of the International Conference on Computational Science
10.1016/j.procs.2017.05.219

The predictable execution model (PREM), originally proposed for single-core [9] and symmetric multi-cores [10], works by separating programs into memory and compute *phases* that can be independently scheduled, as to bound external interference. While its use in heterogeneous many-cores has been recently proposed [11] [12], its practical effectiveness in this context has only been discussed conceptually, or by experimenting with synthetic benchmarks.

In this paper we study the applicability and effectiveness of PREM when applied to a real-life, real-time workload, representative of a key functional block for autonomous navigation: a near-optimal parallel path planner [1] executing on the NVIDIA Tegra TX1. We show that for integrated devices, the GPU is highly susceptible to memory interference from the CPU, and that PREM is capable of greatly mitigating this, reducing the execution time variance to near-zero and providing a significant decrease in the WCET.

2 Path Planning and PREM

Reference path planner. The path planner used in this work relies on a non-deterministic algorithm with lock-free cost updates of the graph [1]. This enables a 3.7× decrease in execution time, at the cost of a small error in the path optimality ($< 1.2\%$). First, *automata synchronous composition* [13] merges a graph representing a discretized topology of the environment (i.e. the map) to a second graph representing the kinematics of the robot. Second, the composite graph is explored via Single Source Shortest Path (SSSP) (Dijkstra [14]). The main data structure is a sparse *state-transition matrix* used to represent all the vertices and the connecting edges. The matrix is stored in the *global memory*, that is mapped in system DRAM. The information about which nodes are "to be visited" is kept in an auxiliary array called *mask array*, while the cost to reach each node is stored in the *cost array*. At each iteration, a *reference node* is visited, and the cost to reach its *neighbors* updated.

In contrast to the original program where all load/stores are done on the DRAM, with PREM, on both the CPU and the GPU bring data from/to the DRAM to/from the L1 scratch-pad during the memory phase, while the compute phase operates on the local copies. This allows mutually exclusive access to the DRAM during *memory access windows*, which must be dimensioned to accomodate the worst case execution time. To get the PREM version, we first apply *warp specialization* [15], as shown in [12], as it provides a means for separating GPU programs into *memory* and *compute phases* that can be independently scheduled, and are identified at runtime via an *if* statement evaluating thread IDs. The entering of the *copy in*, *compute*, and *copy out* steps is protected by synchronization points, and this is the only point the PREM code differs from the warp-specialized code. In the warp specialized code these are regular barriers, as outlined in [15], while in the PREM code these are synchronization points with the CPU, as described in [12]. How to realize PREM on the CPU is discussed in [9].

Naive port experiments. We evaluate the effects of DRAM contention for the described path planning application on an NVIDIA Tegra TX1[1]. We evaluate the implementation on map sizes of 100×100. Results for the three versions of the path planner are shown in Figure 1 (A). Warp specialized code is significantly costlier for this implementation, due to memory copies executing according to a single-buffering scheme (i.e., we are only accessing memory during "half" the time). PREM-enabling synchronizations add marginal overhead.

To evaluate the effects of CPU interference on DRAM accesses, we repeat the test while the CPU executes the `stress` tool, capable of generating large amounts of memory requests. To produce the interference, we instruct `stress` to run 24 threads per core, which access a 32

[1] `http://www.nvidia.com/object/jetson-tx1-dev-kit.html`

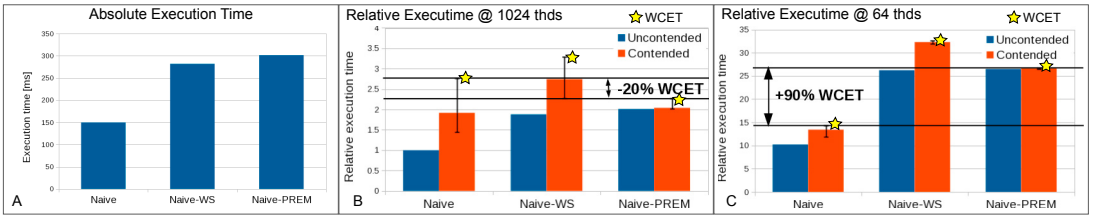

Figure 1: The absolute execution times (A), and the effect of DRAM interference (B) for 1024 threads on the three implementations of *Naive*, and the same for 64 threads (C).

MB array in strides of 129 bytes. Each test is executed 20 times, with and without memory contention. Figure 1 (B) shows that the PREM version has a much lower sensitivity to the interference from the CPU. For all measurements except one the execution time variance is near-zero, which is the expected result[2]. While the WCET of the baseline implementation is higher under contention than that of the PREM-enabled version, the gain is rather small. The reason for this is that the *Naive* port makes poor usage of the memory bandwidth. The *transition matrix* is stored in a large matrix, with each node's successors being stored at random offsets. In addition, the *cost array* is indexed by the vertex ID that does not reflect the order in which the vertices are visited, also leading to non-coalesced memory accesses, as shown in Figure 2 (A). To confirm that the intuition is correct, we run the experiment with 64 threads. As this corresponds to the number of physical processing elements, the GPU memory latency hiding features are prevented (there are no more threads to schedule when one is blocked on a memory transaction), leading to worst possible usage of the memory bandwidth. As expected, Figure 1 (C) shows that PREM has negative gain in WCET in this case.

Implementing Coalesced Memory Accesses. To overcome the poor memory performance of the *Naive* port, a preprocessing stage is introduced. This stage performs an offline exploration of the empty map, reordering the elements of the transition matrix such that they come in the order that they are explored by the sequential version. This change enables the streaming of the transition matrix to the GPU, which implies coalesced memory accesses and maximum use of the memory bandwidth[3]. As some vertices of the graph may be explored multiple times, to keep the streaming property of the transition matrix, these vertices must be added multiple times. We refer to this new version as *Coal*; a visual representation of its access pattern is presented in Figure 2 (B), which shows that each memory access now brings in multiple vertices at once. As the cost array is updated by multiple nodes, storing it in the visit order would introduce coherency issues due to the duplication, thus it is kept in the original format. The calculation of the cost to reach each node is greedy, as inherited from the original Dijkstra implementation: When nodes are explored, if the cost to reach the neighbor from the current reference node is lower than the previous cost, the cost is updated to that of the current node plus the edge. Thus, the algorithm breaks if a node which has not yet had its cost updated is explored, as this error would propagate to all its successors. Especially on a GPU, where thousands of nodes could be explored at once, a mechanism which prevents this must be implemented.

In the *Naive* implementation, this was addressed using the mask array, but for the streaming transition matrix, this is no longer feasible. Instead, the concept of *exploration frontiers* is introduced. The exploration frontiers is an enumeration of sets of vertices F, where all vertices

[2]We expect the outlier to disappear if real-time OS patches are used to bound the latency for the interrupts used to trigger the synchronizations, which we will look into in future work.

[3]Note that this is representative of a typical GPU optimization, and is not specific of PREM.

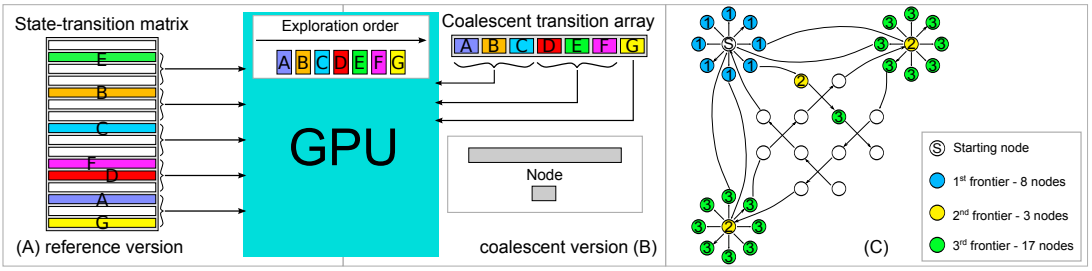

Figure 2: Left: A visualization on the memory access pattern of the *Naive* (A) and *Coal* (B) implementations. Right: An example on how the frontiers are constructed in *Coal* (C).

in F_n have been visited from at least one vertex in F_m for any $m : 0 \geq m < n$. As shown in Figure 2 (C), the base case is F_0 which contains only the source vertex. The next frontier is constructed by all the vertices that can be reached from the source vertex, and then the remaining frontiers are in turn populated by the vertices that can be reached by the previous frontier. The introduction of the frontier concept enables the insertion of breakpoints in the streaming transition matrix, at which point all previous vertices have to have been explored before the exploration can continue beyond that point in the stream, thus ensuring that nodes are not visited out of order. All of these operations are done offline and encoded into the streaming transition matrix. The warp-specialized and PREM codes are achieved in the same manner as for the *Naive* port.

Evaluation of Coalesced Memory Access Path Planner. The execution times for the different versions of the *Coal* path planner is presented in Figure 3 (A). The *Coal* path planner is 6× faster than the naive port, and the cost of warp specialization is somewhat lower than in the *Naive* port, which we attribute to the more efficient use of the memory bandwidth. However, as can be seen in Figure 3 (A), PREM implies a much larger increase in execution time. This is due to the breakpoints at frontier borders, as the current implementation will stop loading vertices when it encounters a frontier breakpoint. This means that at some iterations, the scratchpad buffers will not be completely filled. In the warp specialized implementation, this is not a problem as the iteration will finish more quickly. However, in the PREM-enabled implementation the worst case lengths of the phases are always enforced, which means that the operations on a near-empty and a full buffer will require the same execution time (if less work is available, a thread will idle until the timer for its phase has elapsed). This problem can be solved by always loading as many vertices as possible, and deferring the frontier boundary check to the computation phase. With such an optimization in place, the overhead for PREM is reduced to the same near-zero levels that we observed for the *Naive* implementation.

When we execute the 1024-threaded *Coal* implementation under DRAM interference, the unmodified version of the GPU application suffers a huge (8×) increase in execution time, as shown in Figure 3 (B). At this point the GPU is using the memory bandwidth most efficienctly, which means that it becomes extremely susceptible to interference. The PREM-enabled version remains at the original execution time, which means that the PREM-enabled implementation enables us to reduce the WCET by as much as 3×. In addition to this, the numbers reported for PREM are very pessimistic, because of the buffer-fill problems described previously. In a fully optimized GPU application, we believe the gains can be even larger. These results demonstrate that the isolation property of PREM holds, and that the WCET bounds calculated for the tasks in isolation, i.e., without interference, hold also in the contented case.

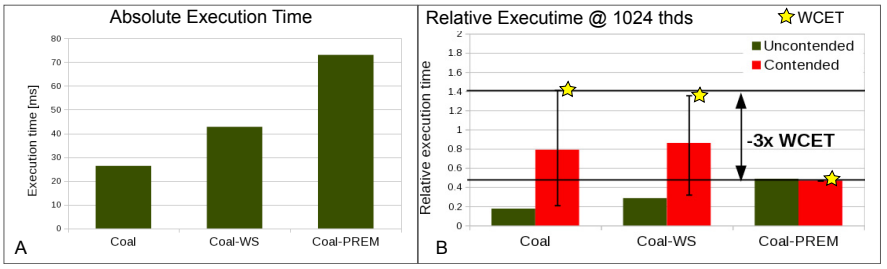

Figure 3: The absolute execution times (A), and the effects of DRAM interference (B) for 1024 threads execution on the three implementations of *Coal*, normalized to the *Naive* port.

3 Conclusion

This work studies the effects on PREM on heterogeneous COTS systems, applying it to a path planning application for autonomous vehicles. We show that integrated GPUs are subject to large increases in execution time under memory contention, that can be mitigated using a heterogeneous extension to PREM. We show that the WCET of the GPU application can be reduced by a factor of 3×, enabling better utilization of the hardware. We also show that the memory access pattern of the GPU application is a key factor to the effectiveness of PREM. This work has been supported by the EU H2020 project HERCULES (688860).

References

[1] D. Palossi *et al.*, "An energy-efficient parallel algorithm for real-time near-optimal uav path planning," in *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 2016.

[2] D. Watzenig *et al.*, *Introduction to Automated Driving*. Springer International, 2017.

[3] P. Burgio *et al.*, "A software stack for next-generation automotive systems on many-core heterogeneous platforms," in *Digital System Design (DSD)*. IEEE, 2016.

[4] S. Chattopadhyay *et al.*, "Worst case execution time analysis of automotive software," *Procedia Engineering*, 2012.

[5] A. Abel *et al.*, *Impact of Resource Sharing on Performance and Performance Prediction: A Survey*. Springer Berlin Heidelberg, 2013.

[6] M. D. Gomony *et al.*, "A globally arbitrated memory tree for mixed-time-criticality systems," *IEEE Trans. Computers*, 2017.

[7] L. M. Pinho *et al.*, "P-socrates: A parallel software framework for time-critical many-core systems," *Microprocessors and Microsystems*, 2015.

[8] H. Yun *et al.*, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *RTAS, 2013 IEEE 19th*. IEEE, 2013.

[9] R. Pellizzoni *et al.*, "A predictable execution model for cots-based embedded systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2011.

[10] A. Alhammad *et al.*, "Time-predictable execution of multithreaded applications on multicore systems," in *Design, Automation and Test in Europe (DATE)*. IEEE, 2014.

[11] P. Burgio *et al.*, "A memory-centric approach to enable timing-predictability within embedded many-core accelerators," in *Real-Time and Embedded Systems and Tech. (RTEST)*. IEEE, 2015.

[12] B. Forsberg *et al.*, "Gpuguard: Towards supporting a predictable execution model for heterogeneous soc," in *Design, Automation and Test in Europe (DATE)*, 2017.

[13] C. G. Cassandras *et al.*, *Introduction to discrete event systems*. Springer Science & Business Media, 2009.

[14] E. Dijkstra, "A note on two problems in connexion with graph," *Numerische Mathematik*, 1959.

[15] M. Bauer *et al.*, "Cudadma: optimizing gpu memory bandwidth via warp specialization," in *High performance computing, networking, storage and analysis*. ACM, 2011.