Experiments for Predictable Execution of GPU Kernels

Flavio Kreiliger*, Joel Matějka*†, Michal Sojka† and Zdeněk Hanzálek†
Faculty of Electrical Engineering/Czech Institute of Informatics, Robotics and Cybernetics†*
Czech Technical University in Prague
Prague, Czech Republic
{kreilfla,matejjoe}@fel.cvut.cz, {michal.sojka,zdenek.hanzalek}@cvut.cz

***Abstract*—Multi-Processor Systems-on-Chip (MPSoC) platforms will definitely power various future autonomous machines. Due to the high complexity of such platforms, it is difficult to achieve timing predictability, reliability and efficient resource utilization at the same time. We believe that time-triggered scheduling in combination with PRedictable Execution Model (PREM) can provide strong safety guarantees, and our longer-term goal is to schedule execution on the whole MPSoC (CPUs and GPU) in time triggered manner.**

To extend PREM to GPUs, we compare two synchronization mechanisms available on the NVIDIA Tegra X2 platform: one based on pinned memory and another that uses a GPU timer (so-called *globaltimer*). We found that running the NVIDIA profiler (nvprof) reconfigures the resolution of the globaltimer from 1 µs to 160 ns. By using time-triggered scheduling with such a resolution, it was possible to reduce execution time jitter of a tiled 2D convolution kernel from 6.47% to 0.15% while maintaining the same average execution time.

***Index Terms*—predictable execution, gpu, nvidia, tx2, prem**

I. Introduction

Autonomous machines such as self-driving cars will certainly be a part of our future. Nowadays, both industry and researchers work heavily on various aspects of those machines. One aspect that is still not satisfactorily addressed is how to ensure their safe operation. Those machines require vast computational power to process all the sensor data, and reason about them in real-time, however, safety systems are traditionally implemented with slow, simple, but reliable computing elements. In contrast to that, autonomous machines are powered with heterogeneous computing architectures, where a multi-core CPU is accompanied by one or more accelerators such as GPUs or FPGAs, often on the same chip. These are called Multi-Processor Systems-on-Chip (MPSoC). In this work, we use a popular representative of these systems: NVIDIA Tegra X2 (TX2), which features a GPU.

While FPGAs can offer precise timing, GPUs seem to be more popular in these applications, perhaps due to their easier programmability. However, GPUs originate from industrial domains, where average-case performance was traditionally more important than real-time and safety guarantees.

This work was supported by the grant no. SGS19/175/OHK3/3T/13 and by the THERMAC project, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 832011.

To reason about safety properties, the functional safety standard for road vehicles ISO 26262 [1] defines the term "freedom from interference", as the absence of certain faults, one of them being "incorrect allocation of execution time". This means that predictable timing is a prerequisite for achieving safety according to this standard.

We believe (and many safety standards agree) that time triggered scheduling gives stronger safety guarantees than online event triggered scheduling. In the case of MPSoC platforms, time-triggered scheduling makes it easier to control contention on shared hardware resources (caches, buses, memories) and thus to control the inter-task interference. Our longer-term goal is to schedule execution on the whole MPSoC (CPUs and GPU) in time triggered manner. In our past work [2], we reduced interference between tasks on a multi-core CPU by time triggered scheduling. This paper is our starting step to doing the same for GPU tasks.

In this paper, we first evaluate synchronization mechanisms for GPU workload, with the conclusion that time-triggered synchronization has the potential of having much lower overhead than lock-based synchronization via so-called pinned memory. However, the overhead of time-triggered execution can be low only when estimates of worst-case execution time are tight. For this reason, we analyze the interference between tasks running on the GPU and try to reduce it by using two main techniques: 1) prefetching data from global memory to local shared memory [3] and 2) scheduling the GPU execution in time triggered manner. Our experimental evaluation shows that these techniques are able to reduce the interference and execution time jitter without significantly increasing total execution time.

More specifically, we adopt the concept of Predictable Execution Model (PREM) proposed by Pellizzoni et al. [4], where computation is split into memory and compute phases, and these phases are scheduled to not interfere with each other – for example, by not running two memory phases in parallel. For GPU workloads, this would result in severe underutilization of memory bandwidth. Therefore, we aim to allow multiple kernels to access memory simultaneously while preserving predictable execution time.

Our overall approach has two main assumptions: a) Executed workload is time-deterministic, meaning that the amount of computation can be determined ahead of time and does not

depend on processed data. This holds for many algorithms used for autonomous machines such as neural network inference [5] or visual object tracking [6]. b) Time-triggered scheduling is used for the whole MPSoC platform to ensure that interference between all on-chip processors can be controlled. On the other hand, it is known that time-triggered scheduling often lacks the required flexibility. For this reason, we envision the use of both time- and event-triggered approaches together. Time-triggered execution will be used for shorter, non-preemptive intervals (e.g., for processing of one camera frame), and multiple of those intervals will be executed in a more dynamic way based on online scheduling and synchronization mechanisms.

II. Related work

Similarly to our work, recent research by Cavicchioli et al. characterized interference on main memory and communication bus level between the CPU and GPU [7]. Other researchers [8], [9] developed various microbenchmarks to understand GPUs and their memory system. Our work differs from those by using time triggered scheduling.

The scheduling behavior of many GPUs is unknown in most cases due to a lack of publicly available and open documentation. Therefore, GPUs are mostly treated as black boxes, and different approaches for predictable execution of different workloads have been developed to bypass this uncertainty. An often used method is to ensure that only one process can access the GPU resources at a time by use of a locking mechanism [10]. The cost of this approach may be an underutilization of powerful GPUs. Dividing the workload into smaller preemptable chunks could reduce this problem [11], [12]. Others evaluated techniques to manage accesses to memory [13] to reduce contention between GPU and CPU applications.

Further Otternes et al. assessed the NVIDIA TX1 platform regarding real-time behavior concerning co-scheduling of multiple kernels [11] [12], and additionally, Amert et al. derived a set of the GPU scheduling rules used in the Jetson TX1 and TX2 platforms to brighten up the black box nature of those platforms [14]. They ran different experiments to understand how the GPU schedules the work if submitted from the same or different processes. They found that the GPU workload launched from different processes shares the GPU by the use of multiprogramming, where each kernel runs exclusively on the GPU during its assigned time slice and does not overlap other GPU computation. For GPU workload submitted from the same process, the computation can overlap and is scheduled according to the derived rules. Bakita et al. proposed a validation framework to validate those derived rules for future GPU generations [15].

Capodieci et al. [16] changed how the GPU workload is scheduled by using an EDF scheduler combined with a Constant Bandwidth Server. Their scheduler is implemented in a hypervisor and works by replacing the run list inside the GPU-host.

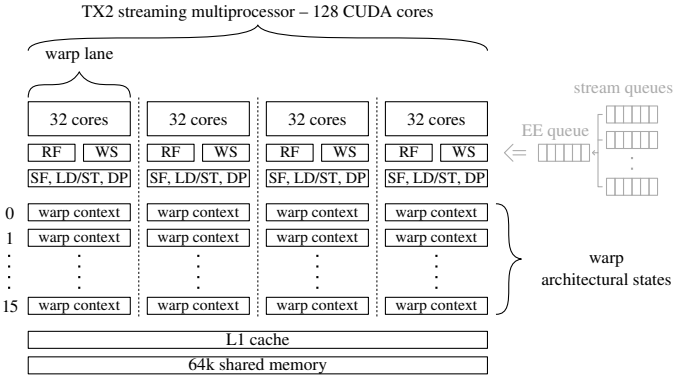

Figure 1. Estimated architecture of one SM of TX2

III. Background

A. GPU/TX2 architecture

NVIDIA Tegra X2 (TX2) is a high-performance embedded MPSoC consisting of two CPU clusters and one Pascal GPU with 256 CUDA cores. The memory bus is shared across the entire chip. However, each CPU cluster and GPU have a separate L2 cache. These caches are not coherent. The GPU is composed of two independent computing blocks called Streaming Multiprocessors (SM), each having its L1 cache, shared memory, and four warp lanes. Each warp lane executes a *warp*, i.e., a group of up to 32 threads performing the same instruction on different data. Since NVIDIA does not publish all details about their GPU architectures, it is difficult to estimate architecture details, and how GPU workload is scheduled on the available warp lanes. Based on publicly available documentation, previous work by Amert [14] and Capodieci [16], and our experiments, we assume the architecture of one streaming multiprocessor to be as depicted in Figure 1. The workload is inserted by CPUs into stream queues, then by rules revealed by Amert [14], put into the execution engine queue and assigned to an SM if enough resources are available. We assume that up to 16 warps can be assigned to a single warp lane. The warps from CUDA blocks (see III-B) are placed in the available *warp context* slots, which store their architectural state, and are run by the hardware warp scheduler (WS) as soon all their dependencies are satisfied. The warp scheduler issues and interleaves instructions from the associated warps, hiding latencies caused by waiting for shared resources. After all warps in a block have finished, the occupied warp context slots are freed and can be reused by warps from the next queued block. Warp scheduling is similar to hyperthreading used in CPUs. Multiple running warps share CUDA cores and other resources such as multiple load/store units, special function units (SF) and double precision units (DP) in one warp lane.

The GPU also features a clock source, called *globaltimer*, which provides synchronous time to all SMs.

In this paper, we do not consider graphics jobs and how the hardware is shared between them and compute jobs.

B. CUDA programming model

To offload computation to the GPU, NVIDIA offers the C/C++ API called CUDA. Programmers write so called *kernels*, i.e., functions that execute in parallel on the GPU. When a kernel is launched, the programmer specifies, with a special syntax, the kernel execution configuration: the number of threads and how those threads are organized into groups (CUDA-blocks or thread-blocks). Each block is executed independently without a built-in possibility to synchronize with other blocks or kernels. Only threads within a single block can be synchronized. Launched CUDA kernels are placed into queues called streams from where they are executed in FIFO order. By default, there is one stream per process. More streams can be created to execute kernels in parallel if enough resources are available. All kernel launches are asynchronous, meaning that if a CPU needs to wait for kernel completion, it has to invoke explicit synchronization operation.

IV. Evaluation goals

In this section, we explain the goals of this paper in more detail.

A. Synchronization mechanism

A precondition for applying PREM to GPU workloads is the availability of fast synchronization between all blocks running at the same time.

In our previous work, we used locks in shared memory to synchronize PREM phases on the CPU [2]. Shared memory offered a fast communication channel since multiple CPU cores share the same cache and the synchronization bypasses the main memory. On the TX2 GPU, an equivalent approach would be to use pinned memory, which is accessed in non-cached manner, to arbitrate accesses to the main memory.

An alternative approach would be to use time based synchronization using the *globaltimer*. We are interested in finding the overhead of the mechanisms and assessing their suitability for whole-GPU synchronization.

B. Benchmark selection

Polybench-ACC [17] is a collection of computational kernels such as matrix multiplication, 2D or 3D convolution, or linear equation solver, used to evaluate the performance of compilers and similar software. Mentioned algorithms are the core of many high-performance applications such as neural networks or image processing. To see the potential benefit of our interference reduction approach, we want to evaluate it on a benchmark highly sensitive to memory interference. Therefore, we evaluated the sensitivity of all polybench kernels to memory interference from CPU and selected 2D convolution as a good candidate.

C. Reduction of intra-GPU interference

As the Polybench 2D convolution kernel is accessing the global memory, it is hard to reduce the interference directly. Therefore, we first apply *tiling* – a technique commonly used to coalescence memory accesses in global memory to speed up the GPU execution [3], [18]. The tiling is done by splitting the input data into multiple tiles which fit into the shared memory segment within a CUDA block. The computation is then performed on the tile previously prefetched from the global memory into the shared memory. At the end, the processed tile is written back. This tiled implementation naturally maps to the three PREM-phases: *prefetch*, *compute* and *writeback*.

Further, we want to assess the interference between the individual phases of tile processing if scheduled synchronously in parallel.

V. Experimental evaluation

We ran all experiments on the Jetson TX2 board in NV Power Mode MAXN and with all frequencies configured to the maximum values by running *jetsonclock.sh* (a script provided by NVIDIA to configure board clocks). All source code we used for the experiments can be found in the git repository: *https://github.com/CTU-IIG/tt-gpu*

A. Pinned memory synchronization evaluation

We evaluated synchronization based on locks in pinned memory with two experiments. First, we measured the ping-pong round-trip time between two GPU kernels and later the experiment was repeated to collect the round-trip times between CPU and GPU since the synchronization mechanism should offer a possibility to be used for CPU to GPU synchronization. Both experiments have been repeated for 1000 times. We had to add the *membar* instruction to ensure that one GPU kernel sees the updates from the other GPU kernels.

Between GPU kernels the average round-trip time was 2.065 µs (min: 1.92 µs, max: 2.24 µs) and the CPU to GPU round trip time was in average 1.94 µs (min: 1.47 µs max: 2.56 µs). These times are not sufficient for synchronizing PREM phases on the GPU, because, as discussed later in Section V-E, the length of the phases is in the range of 1 to 4 µs and compared to this, the overhead of this synchronization mechanism would be too high.

B. GPU timer granularity

We evaluated the *globaltimer* as a synchronization mechanism between GPU tasks. According to the documentation [19], the globaltimer should have a resolution in the nanosecond level. The main criteria for the *globaltimer* to be used as a synchronization mechanism are its resolution and that it is running synchronously on both SMs. To evaluate these properties, we ran a kernel from Listing 1 with four blocks of one thread each. Each block retrieves the globaltimer timestamps in a for loop, storing them into its shared memory segment. The shared memory was selected for two reasons: 1) its access time is short enough to not influence timestamp precision much and 2) allocating shared memory segments to occupy half of the available shared memory on an SM ensures that two blocks execute on one SM and two on the other.

Figure 2 shows a zoom into the first few iterations of collected timestamps. Running the experiment in the default settings gives disappointing results. The measured resolution

Listing 1. Simplified kernel to retrieve global timer jitter

```
__shared__ uint64_t times[NOF_STAMPS];
for (int i = 0; i < NOF_STAMPS; i++)
  asm volatile("mov.u64 %0, %%globaltimer;" \
                       : "=l"(times[i]));
```

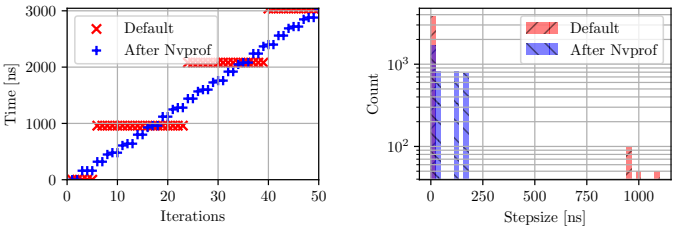

Figure 2. Timestamps and step sizes of the globaltimer after reboot and after one run of nvprof. The retrieved timestamps of the other blocks exhibited the same resolution.

was only 1 µs. The "Default" points on the left side show the timestamps collected by the first block. The right side of the figure shows the histogram of the differences between two subsequent timestamps. By coincidence, we found that running nvprof[1] once on an arbitrary kernel reduces the measured resolution of the globaltimer to 160 ns, as shown with "After Nvprof" points in Fig. 2. The use of nvprof seems to reconfigure the globaltimer on the GPU without reconfiguring it back at the end. Although this behavior is not documented and not really intuitive, it helped us to increase the resolution of the globaltimer.

It is important to highlight, that nvprof needs to run only once on an arbitrary kernel. After this run, the further kernels can run without the instrumentation with nvprof to still profit from the higher resolution.

C. Time triggered execution of tiled 2D Convolution

To see how the execution jitter occurs and if it can be reduced if multiple kernels (4 in our experiments) run in parallel, we compare the original 2D Convolution polybench benchmark (later denoted as *legacy* implementation) and our *tiled* version of it. Each kernel was run 1000 times then the average, minimum and maximum execution times have been calculated. Both implementations apply a 3x3 convolution mask on a dataset consisting of 1026x1022 float elements. The kernels were launched with a configuration of two blocks with 512 threads. The tiled implementation tiles the input data into 512 tiles of 4x512 elements. Each tile is processed in the following phases: first, the tile is prefetched from global memory into the CUDA shared memory segment, then the computation takes place, and in the end, the resulting data is written back to global memory. Since the streaming multiprocessor on the TX2 offers 64 kB of shared memory, we dimensioned our kernel blocks to use 16 kB of shared memory to allow the execution of 4 kernels in parallel. To investigate the possibility of interference reduction, we use the globaltimer to synchronize the running blocks and to control the start times of the tile processing. Figure 3 shows how

[1] *nvprof* is the profiling tool offered by nvidia to analyze traces and timings of called CUDA API and launched kernels

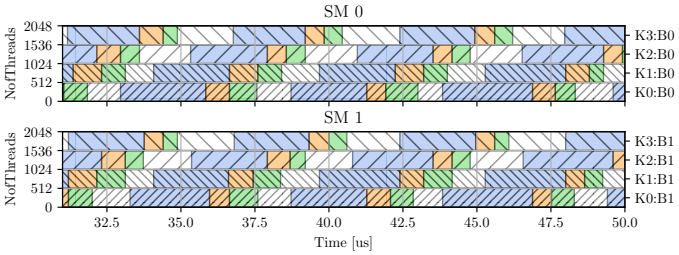

Figure 3. Zoom into the execution of 4 tiled 2D convolution kernels (K0–K3). The total execution time was 2.87 ms. The tiles are scheduled on both streaming multiprocessors with an offset of 1.4 µs against each other. Both blocks of the same kernel (B0–B1) are scheduled at the same time instance and are processing multiple tiles in sequence. Blue, orange and green colors represent *prefetch*, *compute* and *writeback* phases and blocks with the same hatch correspond to the same kernel. During the white phases the blocks are spinning on the *globaltimer* until they are allowed to process the next tile.

the tile processing start times are shifted with an offset of 1.4 µs against each other. The two blocks inside a kernel start processing their current tiles always at the same time, the white spaces between the tile processing phases represent the time a block is spinning on the globaltimer until it is allowed to start with the next prefetch phase.

To have a more elaborate overview of the influence of the tile scheduling offset to the observed execution jitter, we run four kernels of the tiled 2D convolution in parallel with different tile offsets. All kernels recorded their block start/end times using the globaltimer. The difference between the latest block end time and the earliest start time is called scenario execution time.

We can see in Figure 4 the average scenario execution time with the min-max execution jitter (blue) and the corresponding execution jitter compared to the scenario execution time in percentage (red). The dotted black line represents the average scenario execution time of the baseline (4 legacy kernels in parallel). As we can see, the scenario execution time and execution jitter remain relatively stable at 2.5 ms respectively 1.4% until the tile offset exceeds 1.2 µs. After this point, the scenario execution time increases and the execution jitter decreases. Based on these results, we classify the tile offsets of 1.3 µs and 1.4 µs as able to reduce the execution jitter while still having a acceptably low scenario execution time.

Further, the 2D convolution kernels were launched in the next scenarios: i) The original (legacy) implementation with 1 kernel running on the GPU, ii) the legacy implementation with 4 kernels running in parallel, iii) the tiled version with 4 kernels running in parallel but without synchronization and iv) the tiled version with the tile processing shifted by 1.3 µs and v) by 1.4 µs offset. Figure 5 shows the average execution time and execution jitter of the scenarios. The blue bars show the average scenario execution time. The minimum and maximum scenario execution times are represented by the small error bars on top of the blue bars. The red bars represent the min-max jitter in percentage relative to the average scenario execution time. It can be seen, that the legacy implementation suffers from high contention in the four kernel configuration. The worst-case observed execution time (WOET) is still slightly shorter than the WOET of the single

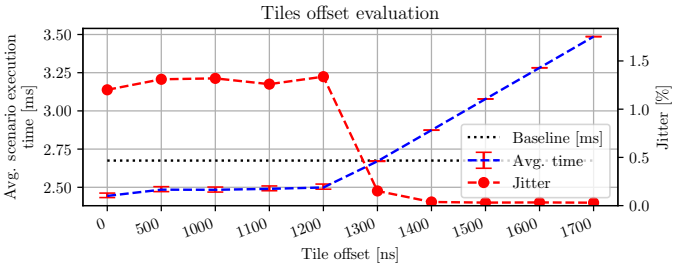

Figure 4. Influence of tile scheduling offset to the scenario execution time and the execution jitter.

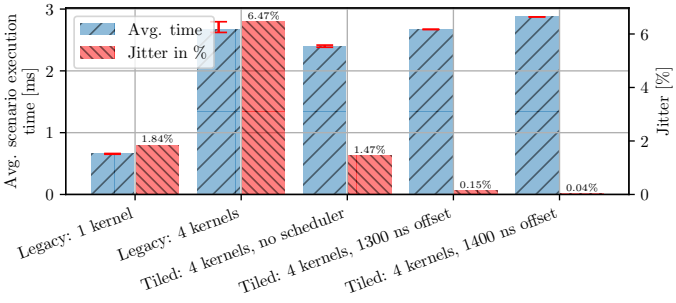

Figure 5. Comparison of scenario execution time. Tiles are scheduled against each other.

kernel version executed 4 times in a row, but the execution jitter is around 6.47% of the average execution time. The tiled implementation with 4 kernels already performed faster than the legacy implementation and its execution jitter is only 1.47%.

The tiling concentrates the accesses to the main memory of the kernels. Therefore, the kernels do not have to access the main memory in all phases, which leads to less contention and lower jitter. The scheduled tiled versions have a bit higher average scenario execution times than the legacy four-kernel version, but with the advantage of execution jitter reduced to 0.15% and 0.04% for the scheduling offset of 1.3 µs and 1.4 µs respectively. Still, one could argue that the WOET of the tiled version (2.42 ms) without scheduling is still shorter than the minimum execution time of the scheduled version (2.87 ms). However, the version without the scheduler offers no future possibilities to synchronize the GPU with the CPU, and the whole execution on the GPU would need to be treated as a single memory phase for CPU PREM scheduling.

D. Phase evaluation

In the tiled implementation, each block processes sequentially multiple tiles, each consisting of three PREM phases. To analyze in more detail how the phases interfere, we added another synchronization point, as shown in Figure 6, between compute and writeback phases to allow independent evaluation of phase interference. By shifting the phase start times, we measured the interference of: i) the prefetch and compute phases (WB is scheduled later not to run concurrently), and of ii) the writeback phases (PF and C are scheduled earlier not to run concurrently).

In Figure 7, we can see how the prefetch and compute phases interfere with each other. The average compute time bars are stacked on top of the average prefetch time bars. The

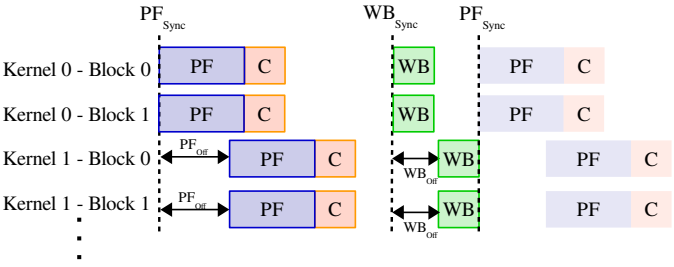

Figure 6. Synchronization points to schedule the PREM phases independently.

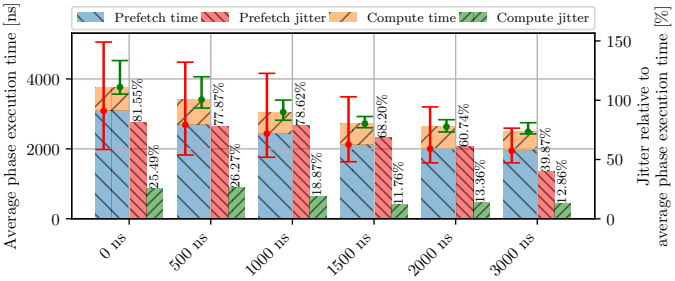

Figure 7. Only *prefetch* and *compute* phases are scheduled against each other (X-axis shows shift offset PF_{Off}). *Writeback* phases are moved away by the schedule and do not influence the previous two phases. In this experiment the two blocks running in a kernel are scheduled at the same time instance.

bars on the right represent the phase execution jitter of the two phases compared to the total average phase execution time (PF + C). One can see that the average phase execution time and the jitter are reduced the less the phases overlap. This effect is dominant in the *prefetch* phases. An interesting fact is that the *compute* phases have the biggest jitter when they overlap with other compute phases (no shift). This indicates some contention on the shared memory or other resources in the streaming multiprocessor. It also prevents the straightforward application of the PREM model, which assumes that compute phases do not interfere.

Figure 8 shows the interference of the *writeback* phases. Similarly to the *prefetch* phases, the less the *writeback* phases overlap, the more the phase execution time is reduced.

Even though the phase execution jitter appears to be high (e.g. 81% in Fig. 7 on the left), the kernel scenario execution jitter percentage is much smaller (1.2% in Fig. 5) since it is relative to longer scenario execution time.

E. Comparison of PREM scheduling on CPU and GPU

When we compare the above-described results with our previous application of PREM on the ARM CPUs of the Jetson

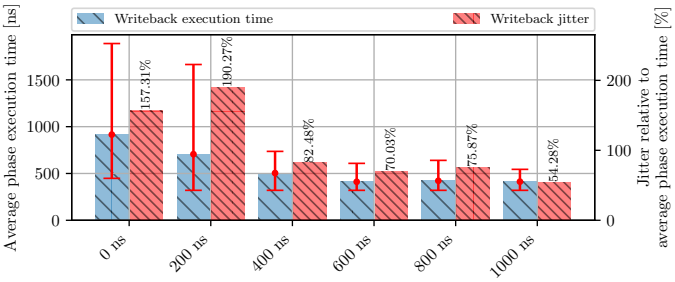

Figure 8. Execution time and jitter of *writeback* phases scheduled against each other (X-axis shows shift offset WB_{Off}). *prefetch* and *compute* phases are scheduled away to isolate the *writeback* phases

TX1 [2], the *prefetch* and *writeback* phases took around 100 and 400 µs respectively and *compute* phases up to 3 ms. This allowed to schedule a sequence of *memory* phases in parallel with one or more longer *compute* phases and the CPUs were efficiently utilized. On the GPU side, the phases execution times are much shorter and differently distributed. Namely, the *writeback* phase has the shortest phase execution time followed by the *compute* and the *prefetch* phases. Therefore, the approach used for CPU PREM scheduling, is not generally applicable to the GPU. When combined with the fact, that the execution time of *compute* phases is influenced by overlapping with other *compute* and *prefetch* phases, it is clear that the PREM scheduling rules need to be changed to be properly applicable to the GPU execution.

The experiment, where the whole tiles were scheduled against each other (Fig. 5), showed that the jitter could already be significantly reduced without introducing big increase of average execution time of all participating kernels. Therefore, a solution to predictable execution times on the GPU requires a different (less strict) set of co-scheduling rules than on the CPU. It remains to be seen whether/how such rules can be used as a proof for *freedom from interference*.

VI. Conclusion and future work

In this paper, we evaluated mechanisms for the low-overhead application of predictable execution model (PREM) to GPU kernels. We compared two synchronization mechanisms for synchronization of PREM phases. The memory-based synchronization achieves round-trip time of around 2 µs, which would result in too high overhead for short PREM phases on the GPU. Synchronization based on the globaltimer allows reaching lower overhead, but only after running *nvprof*, which magically increases the globaltimer resolution to 160 ns. Furthermore, we have shown that by using a tiled implementation of the 2D convolution kernel and tightly synchronizing execution all blocks across multiple kernels by using the globaltimer, we can reduce the execution time jitter from 6.47% to 0.15% while maintaining almost the same average execution time. We have also shown that the duration and interference of the PREM phases are different on the GPU compared to CPU. Namely, the phases are 100 to 1000 times shorter on the GPU and the execution time of *compute* phases can be influenced by other overlapping PREM phases. This and the short compute phase times make it impossible to execute a sequence of *memory* phases in parallel with a *compute* phase. On the other hand, simple scheduling the whole tiles with fixed offsets, investigated in this paper, resulted in sufficiently predictable execution with low jitter. Therefore, we believe that applying more advanced scheduling can lead to even more predictable execution, especially when combined with time-triggered CPU scheduling.

Since we performed the first experiments only on the 2D Convolution kernel, we plan to analyze in more detail how various execution phases influence each other in other kernels. Especially we would like to evaluate the behavior of the PREM phases of more compute intensive kernels. Based on such an evaluation, we want to come up with scheduling rules whose application will lead to low execution time jitter and acceptable performance at the same time. Later we plan to evaluate our scheduling concept on a real application commonly used in autonomous driving. Combining predictable GPU execution with PREM-based CPU execution is also planned.

References

[1] ISO, "ISO 26262 Road vehicles – Functional safety," 2011.

[2] J. Matějka, B. Forsberg, M. Sojka, P. Šůcha, L. Benini, A. Marongiu, and Z. Hanzálek, "Combining PREM compilation and static scheduling for high-performance and predictable MPSoC execution," *Parallel Computing*, 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167819118301789

[3] M. Harris, "Using shared memory in CUDA C/C++," NVIDIA, accessed: 2019-04-09.

[4] 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 *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2011, pp. 269–279.

[5] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[6] J. F. Henriques, R. Caseiro, P. Martins, and J. Batista, "High-speed tracking with kernelized correlation filters," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, no. 3, pp. 583–596, March 2015.

[7] R. Cavicchioli, N. Capodieci, and M. Bertogna, "Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms," in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2017, pp. 1–10.

[8] X. Mei and X. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, Jan 2017.

[9] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, March 2010, pp. 235–246.

[10] Y. Xu, R. Wang, T. Li, M. Song, L. Gao, Z. Luan, and D. Qian, "Scheduling tasks with mixed timing constraints in GPU-powered real-time systems," 06 2016, pp. 1–13.

[11] C. Basaran and K. Kang, "Supporting preemptive task executions and memory copies in GPGPUs," in *2012 24th Euromicro Conference on Real-Time Systems*, July 2012, pp. 287–296.

[12] J. Zhong and B. He, "Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1522–1532, June 2014.

[13] B. Forsberg, A. Marongiu, and L. Benini, "GPUguard: Towards supporting a predictable execution model for heterogeneous SoC," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, March 2017, pp. 318–321.

[14] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU scheduling on the NVIDIA TX2: Hidden details revealed," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, Dec 2017, pp. 104–115.

[15] J. Bakita, N. Otterness, J. H.Anderson, and F. D. Smith, "Scaling up: The validation of empirically derived scheduling rules on NVIDIA GPUs," 2018.

[16] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-based scheduling for GPU with preemption support," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, Dec 2018, pp. 119–130.

[17] University of Delaware, "Polybench-ACC," https://github.com/cavazos-lab/PolyBench-ACC, accessed: 2019-04-09.

[18] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: Optimizing GPU memory bandwidth via warp specialization," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2011, pp. 1–11.

[19] NVIDIA, "Parallel thread execution ISA version 6.4," https://docs.nvidia.com/cuda/parallel-thread-execution/#special-registers-globaltimer, accessed: 2019-04-09.