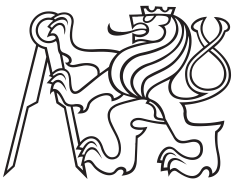
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Time-predictable GPU execution

Flavio Kreiliger

Supervisor: Ing. Joel Matějka
Field of study: Cybernetics and Robotics
May 2019

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Kreiliger Flavio** Personal ID number: **473095**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Cybernetics and Robotics**

Branch of study: **Cybernetics and Robotics**

II. Master's thesis details

Master's thesis title in English:

Time-predictable GPU execution

Master's thesis title in Czech:

Časově deterministické vykonávání kódu na GPU

Guidelines:

1. Make yourself familiar with ongoing efforts and results of making the GPU code execution predictable i.e. suitable for hard real-time embedded applications.
2. Evaluate existing approaches such as PREM-compiler and GPUguard on NVIDIA Tegra X2 SoC.
3. Investigate sources of unpredictability of GPU execution such as cache contention, memory bus contention, GPU scheduling and operating system scheduling.
4. Propose and implement a mechanism for time-triggered scheduling of GPU execution with the goal of reducing timing unpredictability.
5. Document and test all the developed techniques

Bibliography / sources:

[1] J. Bakita, N. Otterness, J. Anderson, and F.D. Smith, "Scaling Up: The Validation of Empirically Derived Scheduling Rules on NVIDIA GPUs", Proceedings of the 14th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications, pp. 49-54, July 2018.
[2] Matejka, Joel & Forsberg, Björn & Sojka, Michal & Sucha, Premysl & Benini, Luca & Marongiu, Andrea & Hanzálek, Zdeněk. (2018). Combining PREM Compilation and Static Scheduling for High-Performance and Predictable MPSoC Execution. Parallel Computing. 10.1016/j.parco.2018.11.002.
[3] B. Forsberg, A. Marongiu and L. Benini, 'GPUguard: Towards supporting a predictable execution model for heterogeneous SoC,' Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, Lausanne, 2017, pp. 318-321.
[4] NVIDIA: CUDA C PROGRAMMING GUIDE, v10.0.130, Oct. 31, 2018.

Name and workplace of master's thesis supervisor:

Ing. Joel Matějka, Department of Control Engineering, FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **06.02.2019** Deadline for master's thesis submission: **24.05.2019**

Assignment valid until: **20.09.2020**

Ing. Joel Matějka	prof. Ing. Michael Šebek, DrSc.	prof. Ing. Pavel Ripka, CSc.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

. _____
Date of assignment receipt

_____
Student's signature

Acknowledgements

First, I would like to thank my supervisor Joel Matějka for his immense support, helpful advice, and productive discussions.

Further, I express my thanks to Michal Sojka, for contributing with his deep knowledge during our weekly meetings.

I would like to express my gratitude to Professor Zdeněk Hanzálek for the opportunity to write my thesis in his group.

Their help and support during the writing process of the paper "Experiments for Predictable Execution of GPU Kernels" made it possible to publish parts of this thesis to the OSPERT19 conference.

Finally, I would like to say thank you to my parents for supporting me during my studies.

Declaration

I hereby declare that the submitted thesis is exclusively my own work and that I have listed all used information sources in accordance with the Methodological Guideline on Ethical Principles for College Final Work Preparation.

Prague, 24. May 2019

Abstract

In this thesis, we evaluate the interference between multiple GPU (Graphics processing unit) kernels running in parallel based on artificial random and sequential walks, the 2D Convolution benchmark provided by polybench and the KCF (Kernelized Correlation Filter) tracker implemented by Vít Karafiát and Michal Sojka. To achieve a reduction of the interference between the running kernels and to reduce the resulting execution jitter, we used a time-triggered execution on the GPU. To enable the synchronization, we assessed two synchronization mechanisms available on Tegra X2 platform: one based on zero-copy memory and one based on the globaltimer. We found that the NVIDIA profiler (nvprof) reconfigures the resolution of globaltimer from 1 µs to 160 ns. With this resolution, we were able to reduce the execution time jitter of a tiled 2D convolution kernel from 6.47% to 0.15% while maintaining the same average execution time by use of a time-triggered GPU execution.

Keywords: predictable execution, gpu, nvidia, tx2, prem, interference

Supervisor: Ing. Joel Matějka

Contents

Figures

Tables

Chapter 1

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, but 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 Central Processing Unit (CPU) is accompanied by one or more accelerators such as Graphics processing units (GPU)s or Field-programmable gate arrays (FPGAs), often in the same chip. These are called Multi-Processor Systems-on-Chip (MPSoC).

While FPGAs can offer precise timing, GPUs seems 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. In this work, we use NVIDIA Tegra X2.

To reason about safety properties, the functional safety standard for road vehicles ISO 26262 defines the term "freedom from interference". Freedom from interference between software elements in the system allows those elements to be analyzed independently, simplifying the whole safety process. Elements are free from interference when certain faults, for example "incorrect allocation of execution time", are not present.

We believe (and safety standards agree) that time triggered scheduling gives stronger safety guarantees. One reason is that it is easier to control contention on shared hardware resources (caches, buses, memories) and thus 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 [1], we reduced interference between tasks on a multi-core CPU by time triggered scheduling.

Another reason for time triggered scheduling is that traditional synchronization schemes such as semaphores and spinlocks are more expensive on larger platforms. Therefore we aim to use time triggered scheduling within shorter time intervals and then occasionally (re)synchronize these intervals using traditional synchronization techniques.

Correct execution time allocation must be derived from the Worst case execution time (WCET) analysis. If the results of WCET analysis are too pessimistic, more time must be allocated and the computational power provided by the MPSoC is wasted and may not be sufficient for required autonomous operation.

In this thesis, we concentrate on the GPU execution only, and we try to characterize the interference between GPU kernels running in parallel and the resulting increase of their average execution time. Based on these findings, we want to show under which circumstances the resulting execution jitter is low or high.

To describe the interference between GPU kernels, we analyze the interference between kernels performing sequential and random walks in parallel. Further, we evaluate how interference induced from the CPU affects the average element access times during those walks.

In a next step, we analyzed this contention with a less artificial kernel than the sequential and random walk kernels. We analyzed memory sensitivity of the available polybench [2] kernels and found the 2D Convolution an appropriate to show how the launch of multiple instances running in parallel influences the execution jitter.

Later we reimplemented the 2D Convolution kernel to split the processed dataset into multiple tiles which fit into the GPUs shared memory. This change allowed us to break the tile processing into three phases: 1) prefetch data into shared memory, 2) perform the computation on the shared memory, and 3) write back the results from shared memory to global memory.

More specifically, we adopt the concept of Predictable Execution Model (PREM) proposed by Pellizzoni et al. [3], 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.

To be able to synchronize the PREM phases, we assessed two synchronization mechanisms available on the Tegra X2 platform: One based on Zerocopy memory and one based on the globaltimer residing on the GPU.

Finally, we performed multiple experiments by scheduling the processed tiles and the PREM phases with different offset to determine how the reduced interference affects the resulting execution time and execution jitter.

To evaluate our findings on an application closer to reality, we measured the interference between the complex matrix operation kernels present in the Kernelized Correlation Filter (KCF) tracker [4] to understand which kernels interfere the most and where the reduced interference might provide the biggest benefits.

Parts of this thesis have been accepted as the paper *Experiments for Predictable Execution of GPU Kernels* by OSPERT19 [1].

[1] `https://ospert19.tudos.org/`

Chapter 2

Background

2.1 Real time applications

In many technical applications, such as process or engine control applications, it is compelling that the systems react in a given deadline. This might be less of a problem for temperature controlling applications, but for safety-critical systems, where serious harm as injuries, death or material damage can happen. The main task of such a system is to deliver the correct result of a job within a given deadline. This deadline can have an arbitrary length from multiple hours to a few us/ns. Depending on the task to solve. There are different definitions of deadlines in real-time systems:

Hard deadline
A hard deadline is defined that the system has to deliver the correct results between a specified time interval. If the result is delivered too early or too late, serious harm can happen to the system or users. A typical example of a hard deadline is an airbag in a car.

Soft deadline
A soft deadline is defined that the results are delivered in average in the specified time interval. If the result is provided earlier or later in some cases, no consequences happen. An example might be a video stream application.

Firm deadline
No immediate harm happens if a deadline is missed, but the result of the job cannot be used in this interval. To achieve this goal a real-time system must run deterministically and predictable and needs to provide enough resources to serve the events correctly even though they arrive in high frequencies. In real-time applications, most commonly no operating system or a real-time operating system is used to ensure that the real-time jobs finish within their deadline requirements. The main difference between real-time operating systems and standard operating systems is that a less important task can be preempted immediately and high priority tasks are not delayed by other work that needs to be performed first. Further, the used platform requires to provide predictable access and execution times (worst case guarantees) on shared resources as, CPU, memory (cache, bus,

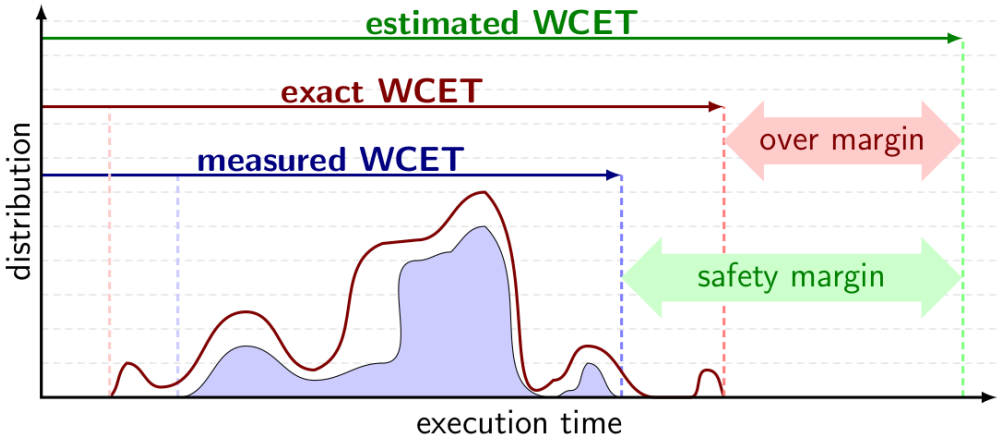


Figure 2.1: Estimation of the Worst-Case Execution Time, and the over-estimation problem [5]

memory controller), IO peripherals and other resources.

2.1.1 WCET

To ensure that a job can fulfill a deadline requirement every time, it is necessary to retrieve its WCET. The WCET specifies the longest time a task takes to finish its assigned goal. To get the WCET of a program on a modern multiprocessor system, it is necessary to include all possible branches, used data and all possible contention with other active components in the system since no guarantee of exclusive execution can be given. This is done by measurement methods (start to end measurement with stressing benchmarks running in parallel) and static methods (instruction counting, detailed hardware model) [5]. Both approaches have their limitations, and an error margin needs to be taken into account. Especially since systems are getting more and more complex, it is difficult to estimate the error margin which leads to conservative WCET estimation and therefore low utilization of the system (see Figure 2.1).

2.1.2 Predicatbility problems in COTS

Today, it is more and more common to build real-time system using Commercial Off-The-Shelve (COTS) components, since computation demand on real-time systems increases (illustrated in Figure 2.2) and costs still need to be in a competitive range. These systems are designed to achieve high average throughput and perform therefore often much better than custom built real-time Hardware (HW) platforms. As a trade-off, the design did not consider worst-case timing guarantees required by real-time systems to ensure the hight average throughput. COTS systems often include optimizations as pipelining, out-of-order execution, data and branch speculation to parallelize instruction execution as much as possible in their cores. Additionally, they consist of multiple active components (CPU, IO-peripherals) that can independently initiate access to shared resources as the

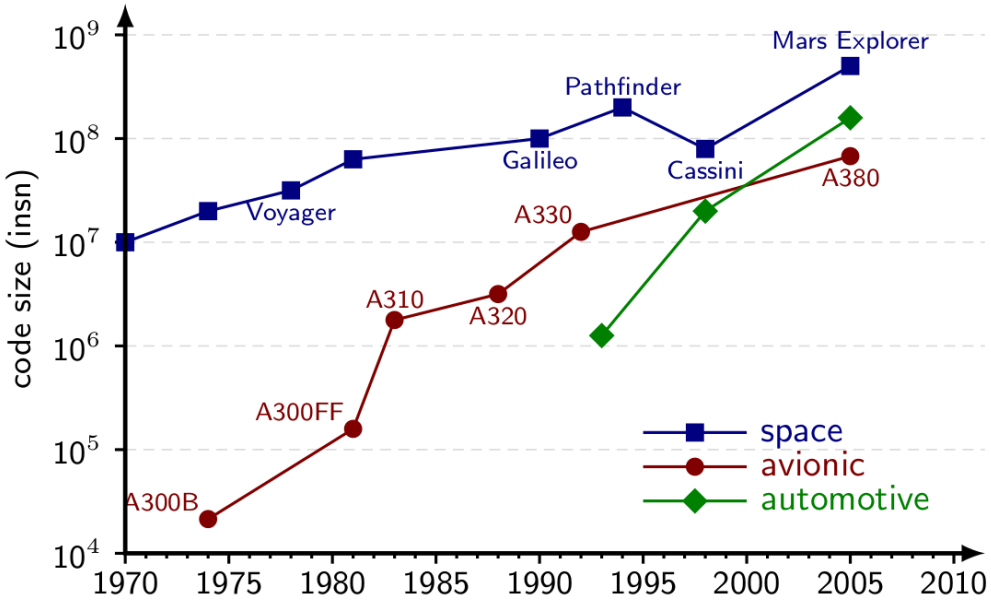


Figure 2.2: Evolution of code size in space, avionic and automotive embedded systems [5]

main memory. In the worst case, this can lead to contention on communication buses, memory devices, and caches and therefore to degraded access times. Due to these optimizations, the WCET is estimated very pessimistically. As an example R. Pellizzoni et al.,2010 showed that the computation time of a task could increase linearly with the number of suffered cache misses due to contention in main memory. This means that if there are three active components in the system, the WCET can nearly triple.[6]

2.2 PREM

To cancel the contention problem and constantly achieve short and predictable response times for real-time task COTS components, it is necessary to control the operation-point of each shared resource and maintain it below saturation limits [3] to ensure no unpredictable processing delays occur. To tackle the contention on the level of bus communication and shared memory, the PRedictable Execution Model (PREM) is introduced [3]. Code of real-time tasks is analyzed and scheduled among all other active components in the system so that contention on accesses to shared resources is resolved and accesses to shared memory resources is scheduled without exceeding their saturation level. This means each task has exclusive access to the resources which allows a much more optimistic estimation of the WCET.

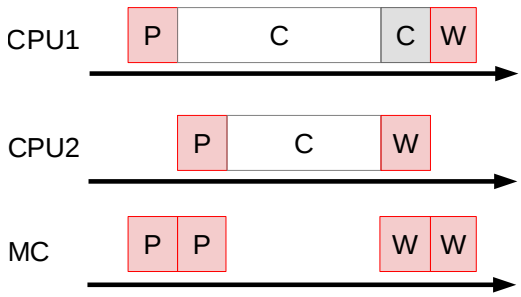


Figure 2.3: Predictable interval [1]

2.2.1 Concept

To apply PREM, a real-time task is scheduled as a sequence of intervals of two different types: *predictable* and *compatible*.

Compatible interval

No transformations are performed for compatible intervals. During their executions, caches misses may happen any time, and the code is allowed to use OS calls. Compatible intervals can be preempted by interrupt handlers of associated peripherals. To reduce pessimism on the WCET, it is recommended to minimize the number of compatible intervals, and to keep them as short as possible. Therefore blocking calls performed within a compatible interval need to have some reasonable timeout.

Predictable interval

Predictable intervals are split into three phases: *prefetch phase*, *compute phase*, and *writeback phase*. The *prefetch phase* and *writeback phase* are also referred as memory phases. Figure 2.21 visualizes a predictable interval consisting of the mentioned phases. During the initial prefetch phase, the CPU accesses the main memory and prefetches the data used in the Compute phase into the last level cache. Since only one task in the system is allowed to be in a memory phase at once, no contention on the shared main memory happens. Since all the cache lines are already prefetched, the following execution of the compute phase experiences almost zero cache misses. Therefore the CPU which is currently running a compute phase does not access the main memory and another memory phases can be scheduled during this time. This allows the next predictable interval to enter its memory phase. To ensure that other active components in the system have enough time to perform their memory phases, the compute phase has a constant execution time. If it finishes earlier, it busy-waits until its deadline. At the end of a compute phase, another memory phase, called writeback phase, takes place to write back the memory. During the execution of a predictable interval, the CPU cannot be preempted by the OS or interrupt handlers.

PREM - transformation

To transform written code to PREM intervals, a particular compiler is used which is built on top of the LLVM Compiler infrastructure. The compiler is developed by ETHZ and is accessible under Hercules-public [1]. The compiler selects PREM-intervals and transforms them into the three phases *prefetch*, *compute* and *writeback*. First, the predictable interval is duplicated three times. On each instance of the duplicated code different transformations are applied that correspond to the single phases. [1]

Prefetch/Writeback
All instructions not related to memory access or calculating memory addresses, including control-flow, branches, and calculations are removed. Load instructions are replaced with prefetch instructions. After this transformation, the prefetch phase is reduced to its minimum. The writeback phase experiences a similar transformation, but load instructions are replaced with cache flush and invalidate instructions.[1]

Compute
The compute phase is taken as is. Since data is prefetched in the prefetch phase, these phases should not experience any cache misses during its execution. Except if the cache replacement policy has evicted some cache lines during the prefetch phase.[1]

PREM - scheduling

After the PREM intervals are determined and transformed, they need to be scheduled to meet the requirements specified by PREM. There are two possibilities to schedule the PREM phases: With an ordered schedule, which is computed offline or using an event-based schedule where each task tries to acquire a memory phase online and waits until no other tasks are in memory phase. The first approach was described and used by Joel Matějka et al. [1], where the gains of PREM have been shown nicely on the application of a GEneralized Matrix Multiplication (GEMM), a Fast Fourier Transform (FFT) and a binary search tree. This approach allowed to reduce the variance of completion times by the factor of 9 for optimal scheduling solutions and by the factor of 5 for scheduling solutions generated by use of a heuristic.

2.3 GPU introduction

In the early 1992 3D graphic cards were still very rare and a topic of scientific research and high-tech development. By the end of this decade, first 3D-graphic cards came to marked to fill this cap in personal computers. With the first graphic cards, the demand for video-games, advanced 3D-modeling in industry could be satisfied. The first available graphic cards on the marked were strictly

[1] `https://iis-git.ee.ethz.ch/H2020-Compiler/HerculesCompiler-public`

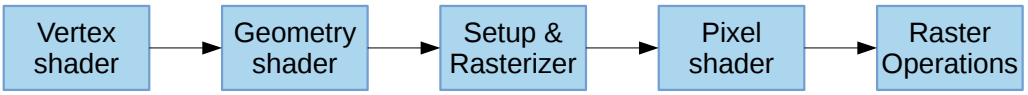


Figure 2.4: GPU rigid pipeline

designed to target the problem of 3D rendering and visualization. They had to output a rendered 3D model up to 60 times a second. Therefore they consisted of a rigid 3D-pipeline shown in figure 2.4. This pipeline consisted of five main stages: *Vertex shader*, *Geomtry shader*, *Setup & Rasterizer*, *Pixel shader* and *Raster operation*. The input was a 3D model of the scene using vertexes and edges and the output was a rendered 3D scene. In the first stage the vertexes of the model are manipulated (transformation, rotation, scaling) and vertex lightening is added, then the geometry shader removes hidden vertexes. In the pixel shader stage texture of colors and pixels is defined and added as pixel fragments. In the last stage those pixel fragments are rendered to pixels and aggregated to the final output image.[7]

Later GPUs have evolved from the rigid hardware pipeline to a more multiple purpose implementation. Fixed function units have been transformed to a grid of unified processors (called Streaming Multiprocessor (SM) for NVIDIA hardware) that can perform those tasks and much more. This transformation has been done through the iteration of many GPU generations. This lead to the GPU-architecture presented in Figure 2.5 where the GPU consists of several general multiple purpose processors. This changes in the graphic pipeline opened completely new possibilities to use the GPU for different compute intensive tasks and not only for graphics related problems. Since in the last years the throughput and performance of GPUs (See Figure 2.6) has increased to a multiple of the performance offered by CPUs, industry and research use GPUs increasingly to parallelize compute intensive tasks. Due to the pipelined and parallelized architecture (Single Instruction Multiple Data (SIMD)), not all algorithms can efficiently utilize the cores of a GPU. Applications which are not well parallelizable and use incoherent and unpredictable memory accesses are not applicable to the streaming multiprocessors of a GPU. Compute intense tasks or streaming jobs on data sets map well to the GPUs streaming architecture. If the algorithms are chosen wisely the GPU can be used as a very efficient accelerator.[7]

2.3.1 GPU on the Jetson TX2

NVIDIA Tegra X2 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 SM, Figure 2.7 shows a high level overview of such an SM in the of a GP100 Pascal GPU. In contrast to the GP100 GPU, the TX2 offers 128 CUDA-cores each having its L1 cache, instruction cache, shared memory, and four warp schedulers. For each warp scheduler the SM offers an instruction buffer, two dispatch units, a register file, 32 CUDA cores, 16 Double-Precision (DP)

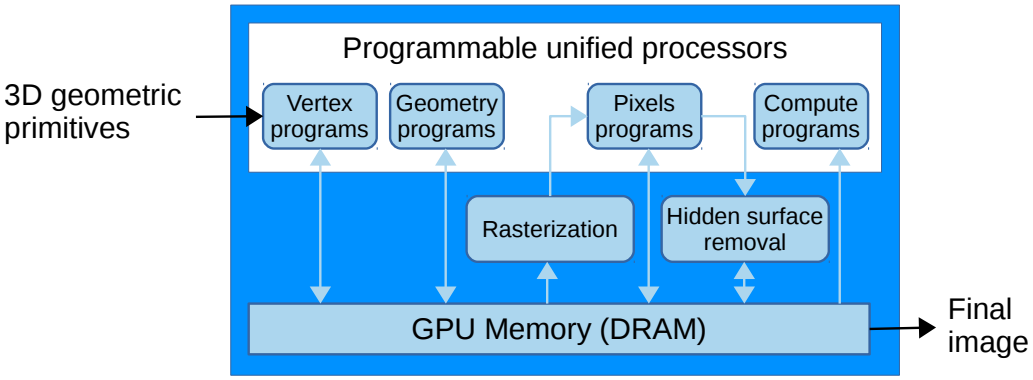


Figure 2.5: "Graphics pipeline evolution.The NVIDIA GeForce 8800 GPU replaces the traditional graphics pipeline with a unified shader architecture in which vertices, triangles, and pixels recirculate through a set of programmable processors.The flexibility and computational power of these processors invites their use for general-purpose computing tasks." [7]

units, 8 Load/Store (LD/ST) units and 8 Special Function Units (SFUs) (sqrt, sin, cos...). Since one SM on the TX2 has 128 CUDA-cores and is partitioned into 4 warp schedulers, the threads are executed in groups of 32 threads, called a warp. If a workload is assigned to the SM the instructions are placed in the Instruction Cache. From there the workload is partitioned and distributed to the four warp schedulers Instruction Buffers. Unfortunately NVIDIA did not publicate how the partitioning is performed but it might be based on the assigned warp-id. The instructions in the Instruction Buffer are marked as ready if all dependencies are satisfied. The warp scheduler tries to fill both dispatch units with instructions marked as ready. Each cycle the warp scheduler gets the number of free Functional Units (FUs) and Dispatch Units. If there are not enough FUs available for the next ready instruction the warp-scheduler stalls or tries to issue another independent instruction to hide the latency. In the Dispatch Unit the warp-id is used to calculate the absolute address for each thread in the register file and the thread mask to decide which threads need to be executed in the next cycle. Then it sends the instruction to the free FUs. If not enough FUs are available the remaining instructions are queued for the next cycle and the dispatch unit remains busy for more than one cycle.

As previously mentioned the thread mask is used to mask the execution of threads for an instruction if they have diverged inside the warp. This happens if the warp has to execute a conditional branch as shown in Figure 2.8. In this case the first 4 threads take one branch and the other 4 threads the other branch. First, the thread mask is set to mask the latter 4 threads to execute the instructions A and B. If those threads reach the reconverge point the mask is inverted to mask the first 4 threads for the execution of instructions X and Y. After this point the unmasked execution takes place again for all threads to execute instruction Z [9]. This needs to be considered since this branching behaviour can reduce the parallelization in the GPU and nearly double the execution time in the worst case.

To visualize the warp-scheduling with more clarification, the Instruction Buffer can be imagined as a storage containing 16 warp contexts (instruction, warp id

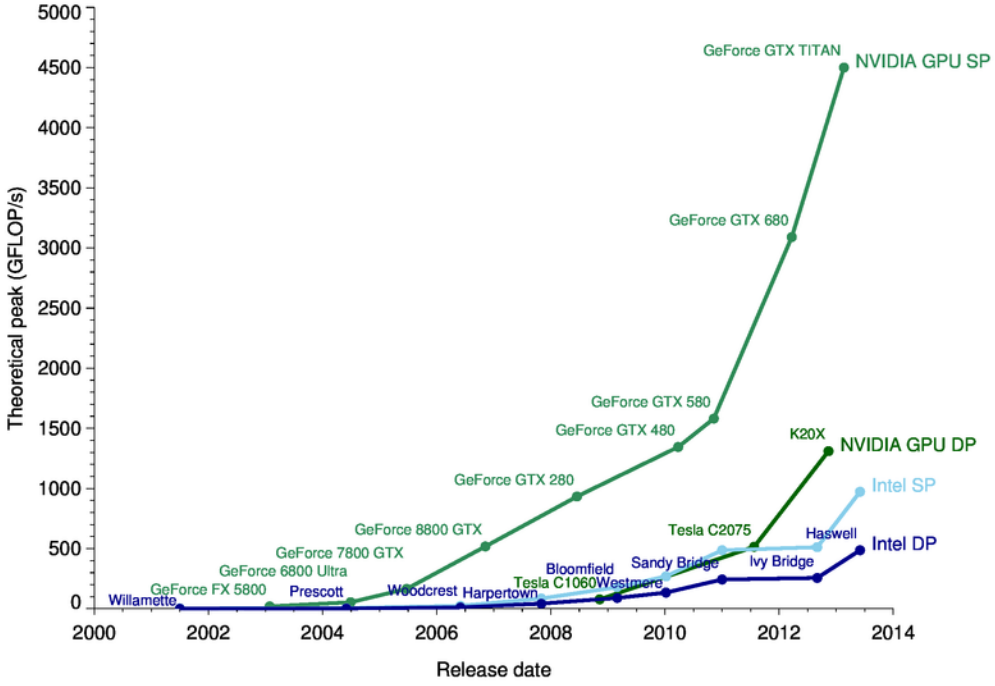


Figure 2.6: CPU vs GPU performance[8]

and the thread mask). The warp scheduler checks if one of those warp contexts is ready and schedules on the designated warp lane or dispatch unit. 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 [10] and Capodieci [11], and our experiments, we assume the architecture of one streaming multiprocessor to be as depicted in Figure 2.9. The workload is inserted by CPUs into stream queues, then by rules revealed by Amert [10], 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 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 dependencies of the threads 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 block. Warp scheduling is similar to hyperthreading used in CPUs. Multiple running warps share CUDA cores and other resources such as multiple LD/ST units, SFUs and DP units in one warp lane. Latencies generated by instruction or data cache misses or other unsatisfied dependencies can be hidden by scheduling another warp-context. This way the GPU is busy as long there are enough threads ready to be executed.

Further the two SMs feature a shared nanosecond timer, called globaltimer, which runs synchronously on the GPU.

Figure 2.7: GP100 Pascal micro-architecture [9]

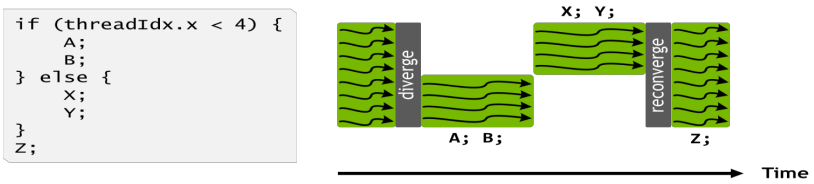


Figure 2.8: Diverged Warp execution [9]

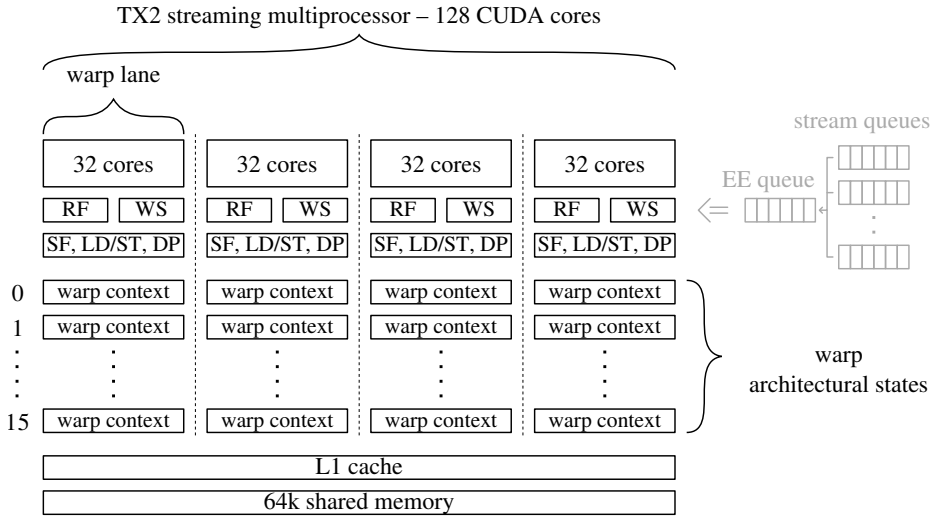


Figure 2.9: Estimated architecture of one SM of TX2

2.4 CUDA

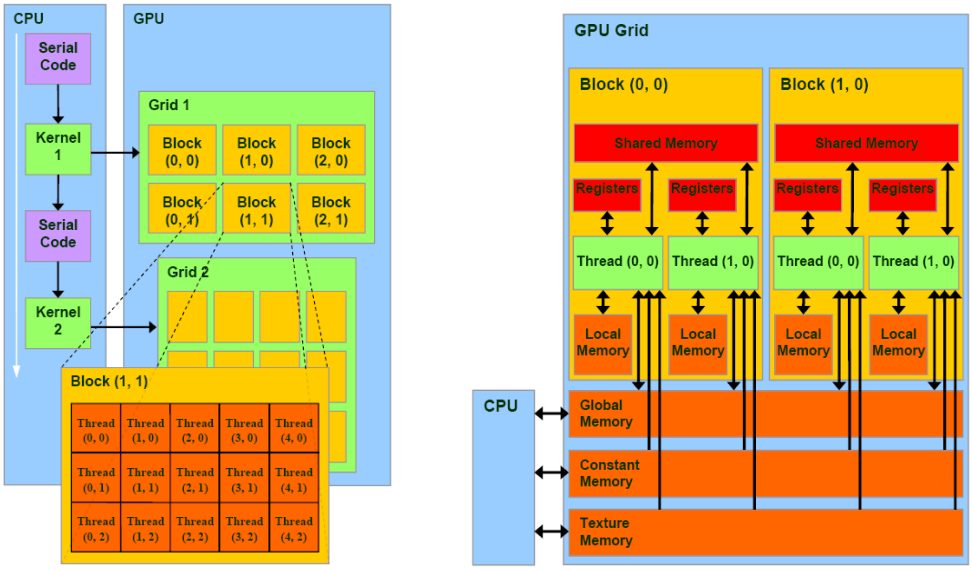
To allow an easy and straight forward implementation of computational work on the GPU NVIDIA introduced the CUDA framework [1]. To offload computation to the GPU, programmers write so called *kernels*, i.e., functions that execute in parallel on the GPU. When a kernel is launched the programmer specifies, 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). Launched CUDA kernels are placed into queues called streams 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. A good starting point to familiarize with the CUDA framework is the CUDA C Best Practices Guide[2].

2.4.1 Programming model

The CUDA-Programming model splits a program into host and device code. Host code runs on the CPU and can only access data available on the CPU side. Device code on the other side runs on the GPU and can access data stored on the GPU. CUDA offers a framework to write programs that are able to run on both platforms. The code segment performing the transition of the execution from the CPU to the GPU is named kernel and can be called with a special syntax from the CPU side. To differentiate the code segments between host, kernel and device code CUDA offers two important keywords: `__global__` and `__device__`. The `__global__` keyword marks a function as kernel code. Those functions can be called from the CPU and issue a kernel launch on the GPU. The `__device__` marks device functions. Those functions can not be called from the CPU but only from the GPU. Further this keyword allows to create global variables on the device accessible inside the running kernels.

Listing 2.1 shows the code of a simple CUDA program. First, the data is allocated and initialized on the CPU (host). Then, if the traditional memory model (see 2.4.2 Memory model) is used, memory is allocated on the GPU and the data is copied into this device environment using the GPUs copy engine. This data transfer step can be omitted sometimes if a different memory model is used. After the data is available on the GPU the kernel can be launched. To launch a kernel, the programmer specifies the number of threads to be used in the kernel and how those threads are grouped together into thread blocks. Further, the size of the used shared memory in a kernel and the stream, where the kernel launch is enqueued, can optionally be specified. Since all kernel launches are asynchronously, the programmer needs to place a `cudaDeviceSychronize()` call to issue an explicit synchronization before copying back the results. After the synchronization point the resulting data is copied back to the CPU and is available for further processing.

[1] `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`
[2] `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`



(a) : CUDA programming model [13] **(b) :** CUDA memory model[13]

Figure 2.10: CUDA models

Figure 2.10a shows how a kernel is launched in a grid (collection of CUDA-blocks) of thread blocks. The grid and the blocks can have up to 3 dimensions where each thread can be identified according to his X and Y block and thread ids, stored in the `threadIdx.x`, `threadIdx.y`, `blockIdx.x`, `blockIdx.y` registers.

During the kernel execution the thread and block ids indicate the running thread which element to access. If more elements need to be processed than threads are available the kernel implementation can be changed to use grid-stride loops to instruct threads to process more than one element.

Threads in one block can synchronize by use of barriers or communicate through the shared memory. One streaming multiprocessor can execute multiple blocks until some register, memory, thread limitations occur. But a CUDA block can not be split between two streaming multiprocessors since threads could not longer share their memory or synchronize.

The CUDA programming model has the disadvantage, that the offloading of algorithms to the GPU need some rewriting of the code. There are other possibilities, as for example OpenMP and OpenACC, to offload kernels by simple compiler directives. Those frameworks use the CUDA driver API behind the scenes and the algorithms are implicitly transformed to the CUDA language. Chapter 2.5 *OpenMP offloading* shows how GPU offloading using OpenMP works in more detail.

2.4.2 Memory model

On the GPU, memory can be accessed in different locations. Figure 2.10b gives a short overview of the CUDA memory model. The red blocks represent fast, on-chip memory whereas the orange blocks represent slower memory. The registers are available per thread and CUDA-core and can be used for local storage. The second

Listing 2.1: Example kernel in CUDA: saxpy [12]

```
#include <stdio.h>

__global__ void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
  int N = 1<<20;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

  // Perform SAXPY on 1M elements
  // kernel<<<gridDim, threadDim, sMem, stream>>>(ptr_data);
  saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
  cudaDeviceSychronize()

  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = max(maxError, abs(y[i]-4.0f));
  printf("Max error: %f\n", maxError);

  cudaFree(d_x);
  cudaFree(d_y);
  free(x);
  free(y);
}
```

on-chip memory is given by the shared memory (scratchpad memory) which gives the threads the possibility to communicate with each other and to exchange data since it is shared inside a CUDA-block. Further there is local, global, constant and texture memory. If data is loaded from global memory, data is by default cached in the L2 cache only. The L1 cache is only used for local memory to ensure that local data, which was stored in local memory due to register spilling, is not evicted by global loads cached in L1 cache. This behavior can be changed by the compiler flag `-dlcm=ca` to instruct NVIDIA CUDA Compiler (NVCC) that global loads should be cached in L1 cache.

Traditional memory

In the traditional memory model the host and device manage separate memory locations. Before a kernel can process a dataset located in host memory, the programmer needs to allocate the memory on the GPU and issue a copy operation to copy the data to the GPU. After the kernel execution the resulting data needs to be copied back to the host.

```
// Allocate Matrix A
// Allocate B and D on the device
// and copy B to the device
// Run kernel using A, B and C
// Copy back B and D to the host



// Allocate C
// Run kernel using A and storing results in
// Copy back C to the host



// Free all device memory
```

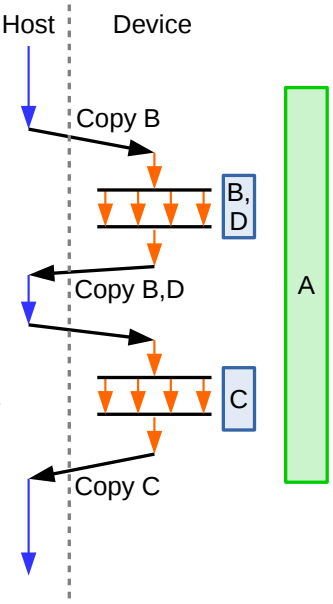


Figure 2.11: Traditional memory model.

To avoid a lot of copy operations, already allocated device memory can be reused to share memory between multiple kernels. This enables a more fine grained control about the data movement during the execution. The use of shared memory segments between multiple kernel calls is shown in Figure 2.11. It can be seen that matrix A is allocated in the device data environment. All kernels launches afterwards can access this data and work with it. The first kernel copies data B to the device environment and allocates D in it. Then it performs some work with AB and D. At the end of the kernel B and D are copied back to the host environment. The second kernel allocates C on the target and performs again with A and C. At the end C is copied to the host environment. This way the

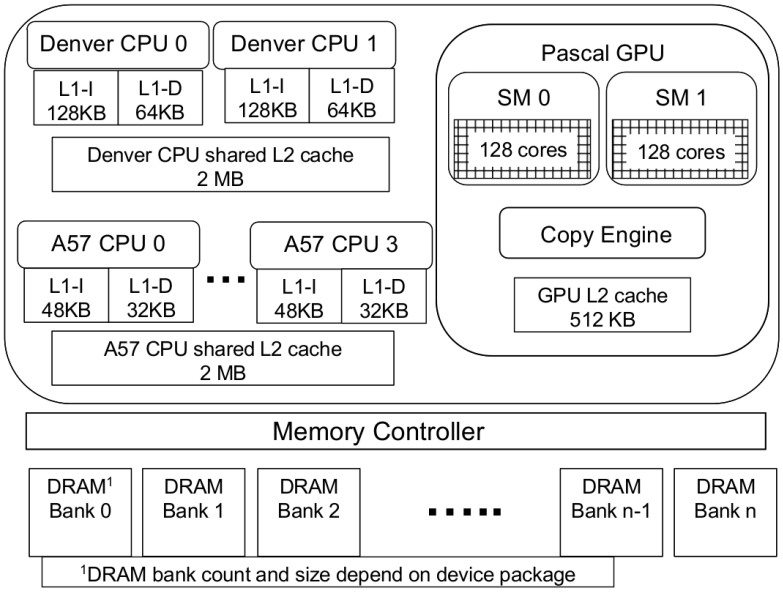


Figure 2.12: TX2 block diagram[10]

copying of A can be avoided while different kernels can work on the same memory.

Zero-copy memory

As we can be seen in Figure 2.12, the Jetson TX2 platform shares memory between the CPUs and GPU. This enables to use host-pinned memory (not pageable) and unified managed memory in zerocopy mode. This means that the host CPU and GPU have pointers to the same memory location to work on data without the copy operations from host to device or vice-versa. Host pinned Zerocopy-memory retrieves a device pointer to a memory segment which is pinned to the CPU (not pageable). Using this device pointer the GPU can access the memory location. To ensure coherence of this memory location caches are disabled which lead to a significant slowdown in performance. The unified managed memory model offers an abstraction layer to the memory access. Memory is allocated in the unified memory space and therefore in host and device memory. The Unified memory model ensures the consistency between those two memory location behind the scenes using page on demand technology. For the programmer point of few its seems to work with only one pointer on the GPU and CPU without explicit copy operations to the device and back. Since the Jetson TX2 platform offers shared memory, the unified memory model allocates the memory segment only once and uses therefore also Zerocopy-memory but this time with caches enabled. This has the advantage of a speed up compared to the host pinned Zerocopy-memory but needs cache flushes to ensure synchronization between CPU and GPU caches and can induce some unpredictable overhead on page misses, since some CPU interaction is need to resolve the page miss. Listing 2.2 shows the calls to allocate host-pinned and managed unified memory segments.

Listing 2.2: Host pinned zero copy memory

```
double * A;
double * A_target;

// Allocate memory on host
cudaHostAlloc(&A,
              NX*sizeof(double),
              cudaHostAllocMapped);

// Get corresponding device pointer
cudaHostGetDevicePointer((void **) &A_target, A, 0);

// Allocate memory using managed unified memory
cudaMallocManaged(&A,
                  NX*sizeof(double),
                  CudaMemAttachGlobal);

            ...
```

2.4.3 Coalescent access

In a kernel launched on the GPU, multiple threads might work in parallel and access memory in parallel. Therefore it would be desirable that the access to memory of the single threads are not evicting already cached memory of other threads. This can be achieved by using a as coalescent memory access as possible.Figure 2.13, shows an access pattern where each block processes 1024 consecutive elements in memory. This leads to a fast and efficient access since the thread in one block have less competition for memory. If a coalescent access to the data in global memory can not be provided a solution can be to copy a memory segment into the shared memory of the block to provide a coalescent during the further computations.

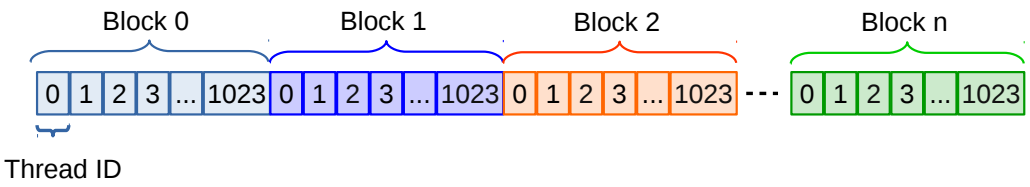


Figure 2.13: Coalescent memory access

2.5 OpenMP offloading

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. OpenMP uses a portable and scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer. It is a multi threading implementation which instructs a master thread to fork the code segment to a specified number of slave threads (See 2.14).

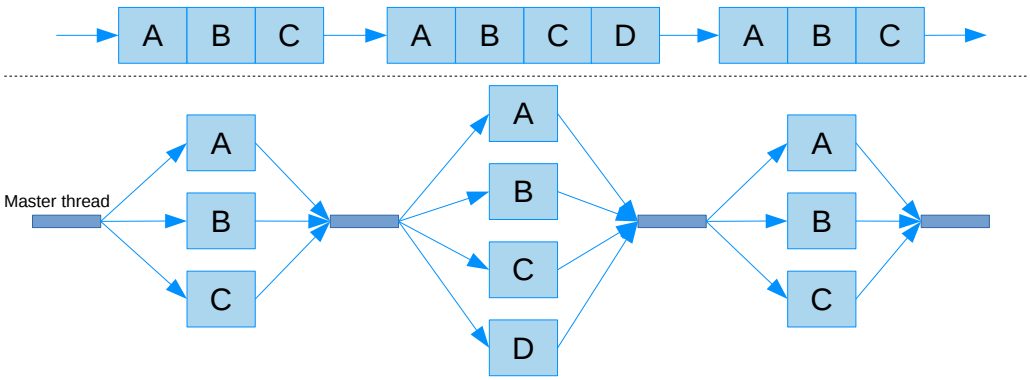


Figure 2.14: OpenMP model

To parallelize a section of code, for example a for loop, it is marked with appropriate compiler directives to specify how to distribute it along the slave threads. Each thread has its unique id starting at 1, whereas the master thread has the id 0. By default each slave thread executes the code by its own. Using work sharing constructs allows to distribute the workload along the available slave threads so each thread performs its particular piece of work. An easy example how the parallelization work, is shown in listing 2.3. Here a parallel region is executed once by each launched thread. Each thread prints out its assigned id. For the master thread it is id 0 and for the slave threads the incremented number starting from 1. The threads can use the OpenMP runtime library to retrieve their id during the execution. In this example the call to `omp_get_thread_num()` was used. Listing 2.4 shows an example how a for loop is parallelized using openMP. The workload is distributed by OpenMP along the optimal number of threads, which is usually the number of available cores in the system. In more detail, the `omp parallel` clause marks the for loop as a parallel region and the `omp for` clause instructs the compiler to distribute the work along the available threads. The `omp schedule` directive defines to use a static schedule with chunk size 1. Therefore each thread processes one element per iteration. If the chuck size would be bigger, for example 10, each thread would perform 10 iterations until it is free for the next 10 elements. Since the `omp nowait` clause was not used, the threads wait at the end of the parallel region until every thread has finished. Therefore a implicit barrier is generated.

Listing 2.3: OpenMP Hello World!

```
int main(int argc, char* argv[]) {
#pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf("Hello World from core: %d\n", ID);
    }
    return 0;
}
```

Listing 2.4: OpenMP example on CPU

```
void foo(int n, float *a, float *b) {
#pragma omp parallel for \
  schedule(static, 1)
    for (int i=1; i<n; i++){
        b[i] = (a[i] + a[i-1]) / 2.0;
    }
}
```

2.5.1 Offloading kernels

To instruct OpenMP to offload a kernel the `omp target` clause is needed. This clause generates a target region which copies data to the target environment executes the kernel and copies the results back. This section describes the offloading process on the generalized matrix multiplication of Polybench-C. Listing 2.5 shows the offloaded kernel.

Listing 2.5: GEMM - sequential c

```
for (int i = 0; i < NX; i++){
    for (int j = 0; j < NY; j++){
        C[i][j] *= beta;
        for (int k = 0; k < NZ; ++k)
            C[i][j] += alpha * A[i][k] * B[k][j];
    }
}
```

Simple offloading

With the `omp target` clause the offloading to the target is triggered as shown in listing 2.6. Each target region is also a data region which means that data used

in the kernel is mapped to the device. If not specified with the `omp target map` clause, data is mapped using the *tofrom* classifier. This means, data is copied to the device before the kernel execution and copied back after the kernel execution. This does not necessarily apply to constant variables since they can be used as absolute values. Since no `omp target teams` clause was used, the compiler generates one team (CUDA-block) whose master thread executes the parallel region. The `omp parallel for` (`thread_limit`) launches the slave threads on the device and distributes the work along them. If the number of threads is not defined the maximum number allowed per CUDA-block is used.

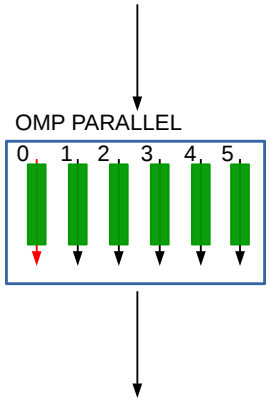
Listing 2.6: GEMM - simple offloading

```
#pragma omp target map(from: C[0:NX][0:NY]) \
                    map(to: A[0:NX][0:NZ], \
                        B[0:NZ][0:NY])
#pragma omp parallel for
for (int i = 0; i < NX; i++){
    for (int j = 0; j < NY; j++){
        C[i][j] *= beta;
        for (int k = 0; k < NZ; ++k)
            C[i][j] += alpha * A[i][k] * B[k][j];
    }
}
```

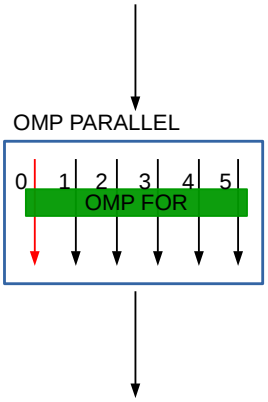
Teams and Threads

This subsection describes the parallelization and work sharing constructs `omp parallel`, `omp teams`, `omp for` and `omp distribute` working with teams and threads. These OpenMP clauses can be combined to instruct how the workload should be distributed and to achieve as much parallelism as possible. In OpenMP, the threads are launched in teams. An OpenMP team corresponds to a CUDA block. The `omp parallel` clause launches a team of slave threads, where each slave thread executes the same parallel region (See Figure 2.15a). To distribute the code of a parallel region along the slave threads the clause `omp for` clause can be used. This clause instructs the compiler to split the parallel region, in this case a for loop, into multiple chunks and assigns those chunks to the slave threads (See Figure 2.15b). The chunk size and the schedule which are assigned to the threads can be specified using the clause `omp schedule(type,chunk size)`.

Since GPUs can hide latencies if multiple thread blocks are executing concurrently it is desirable to distribute the parallel work also across multiple teams. This is done in CUDA using the concept of grid and blocks. This is solved similarly in OpenMP using the `omp teams` directive which launches multiple teams of threads (CUDA-blocks). As shown in Figure refpic:openmpteams multiple teams are launched but each master thread of the teams executes the same region of code. Therefore the `omp teams distribute` clause is needed to distribute the

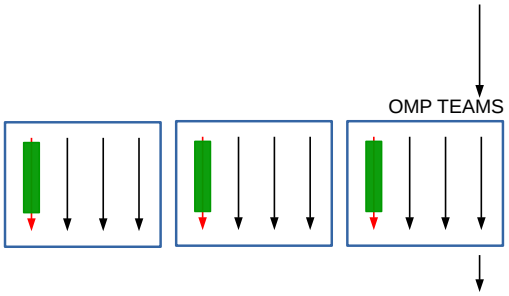


(a) : `omp parallel` launches team of threads where each thread executes the same parallel region.

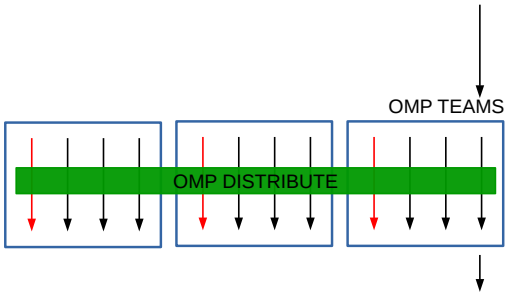


(b) : `omp parallel for` distributes the work load across the threads in a team

Figure 2.15: OpenMP parallel launch



(a) : `omp teams` launches a league of teams, where the master threads of the teams execute the same parallel region



(b) : `omp teams distribute` launches a league of teams and distributes the workload along the master threads. The distribution within the teams is not specified yet and needs further work sharing constructs as `omp parallel for`

Figure 2.16: OpenMP teams launch

region along the teams (Figure 2.16b). But there is still the problem, that only the master threads of the teams are executing the assigned region. Therefore the `omp teams distribute` needs to be combined with `omp parallel for` to achieve also full parallelism within the teams. It is important to note, that there is no synchronization between the teams.

2.6 GPU scheduling on the TX2

As described previously, the CUDA kernels are executed in multiple blocks consisting of several threads. Those blocks are assigned to the SMs if enough resources are available and the warp scheduler, which is implemented in hardware, swaps warps immediately if stalls (example waiting for memory) in a running warp occur. This way memory latencies can be hidden and the SM is fully utilized. This scheduling is done in hardware and can not directly be influenced by the programmer. CUDA offers streams to order GPU operations. Operations in a stream are executed in FIFO order, but the order along different streams is determined by the GPU scheduling policy. It is possible to assign priorities to the streams to prioritize the execution of certain streams. Therefore kernels from multiple streams may execute congruently or out of launch time order. The rules the GPU uses to schedule blocks originating from different streams is not well documented, but Tanya Amert et al. have revealed some restrictions with different experiments [10]. Their research considers multiple running instances of synthetic benchmarks with different configuration of block resource requirements, kernel durations and copy operations. Further they assume several queues where kernel and copy operations are enqueued:

- Execution engine queue (EEQ): Queue which holds blocks to be assigned to a SM (FIFO)
- Copy engine queue (CEQ): Queue which holds copy operations to be assigned to copy engine (FIFO)
- One FIFO queue per CUDA stream.

Figure 2.17 shows the assumed queue architecture. Generally said, the head of all stream queues is placed in FIFO order in the EEQ. Only the blocks of the kernel placed at the head of the EEQ can be assigned to the SMs if enough recourses are available. If all blocks of a kernel are assigned, this kernel is removed from the EEQ. After all blocks of a kernel have completely finished their execution on the SMs, the kernel is also removed from its stream queue. Then the next kernel from the stream queue can be placed in the EEQ. This rules, discovered by Tanya Amert et al.[10], can help a programmer to implement and tune a CUDA application to be scheduled the way as expected.

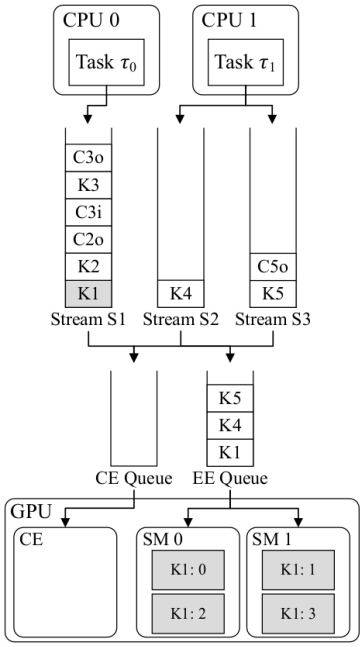


Figure 2.17: TX2 queue architecture [10]

2.7 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 [14]. Other researchers [15, 16] 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 a 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 [17]. The cost of this approach may be an underutilization of powerful GPUs. Dividing the workload into smaller preemptable chunks could reduce this problem [18, 19]. Others evaluated techniques to manage accesses to memory [20] 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 [18][19], 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 [10]. They ran different experiments to understand how the GPU schedules 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 [21].

Capodieci et al. [11] 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.

2.7.1 GPU-GUARD

To adapt the GPU execution to the Predictable execution model, the kernels are also premized similar to the CPU premization mentioned in the previous chapter 2.2 by use of the Hercules compiler [22]. Therefore kernels are separated into the *Memory* and *Compute* phases. Really short kernels which are not usefully premizable are hold as is and treated as *Memory* phases. The other kernels are tiled into loop segments only accessing data segments of the size of 48kBytes. This size comes from the limitation that each thread block can use up to 48kBytes of shared memory called scratch pad memory. In the following compute phases the running threads can access their data in the scratchpad memory (shared memory) without further memory misses. To synchronize the PREM phases with PREM phases running on the host, the tool *GPU-Guard* was introduced by Björn Forsberg et al. [20]. Figure 2.18 shows the initial architecture of GPU-Guard. It consists of the CUDA/GPUguard program and the GPUguard Loadable Kernel Module (LKM). Since the only way to communicate between the GPU and CPU on the Jetson TX2 platform is through main memory, the communication between the LKM and the kernel on the GPU is based on polling on a host pinned memory location. Each thread block in the CUDA code writes to its shared memory location to indicate that it wants to enter a new PREM phase. Then it is busy waiting on this location until the request is acknowledged by the GPU-guard LKM. In the beginning of a CUDA-kernel execution each thread block performs a checks in request to GPUguard. This way GPUguard knows how many CUDA blocks belong to a single kernel. The same happens at the end of a CUDA-kernel where each thread block requests a checks out. After all CUDA blocks performed a successful checkout, GPUguard know s that the kernel has finished and is ready to synchronize the next kernel. After all blocks have checked in, the single blocks acquire a memory phase and busy wait on their shared memory locations until it is acknowledged. GPUguard on the other side only acknowledges the *Enter memory phase* request after all running blocks have requested it. The same happens on the request *Enter compute phase*. This way it is ensured that all blocks running on the SM congruently are entering a memory phase the same time. Therefore the whole GPU (all running block on the GPU) is either in *memory phase* or *compute phase*. This process is illustrated in Figure 2.19.

The GPUguard LKM consist of a high resolution timer callback function and throttle threads. After the GPU was allowed to enter a memory phase, the throttle threads are launched on all cores to throttle down the CPU execution. This way it is ensured, that the CPU is not accessing the communication bus or the memory.

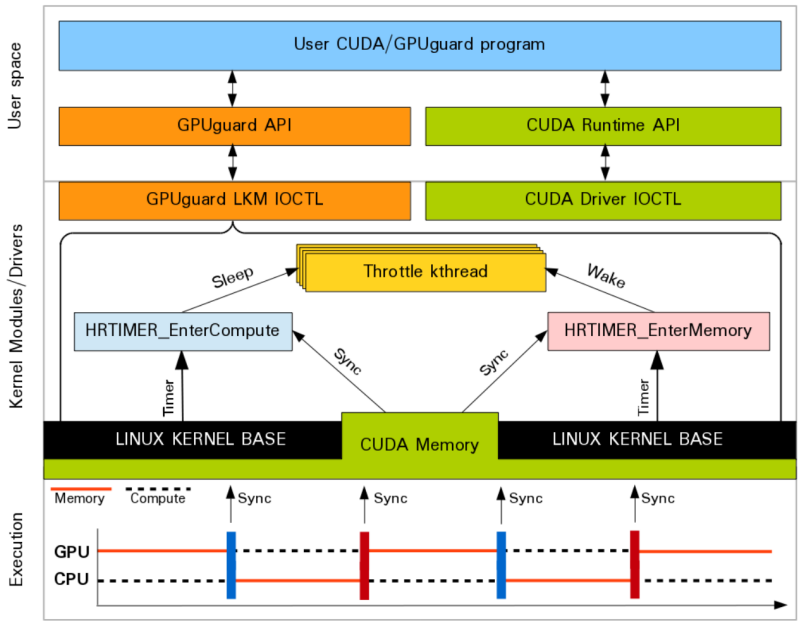


Figure 2.18: An architectural overview of the implemented *GPUguard*.[20]

To enable a real synchronization with the PREM applications on the host, those throttling threads were replaced with call to Memguard to acquire a memory phase. This call is blocking until a new Memory phase slot is available. In this case GPUguard acknowledges the GPU *Enter memory phase* and the GPU enters the memory phase. On a *Enter compute phase* request on the other hand the lock in Memguard is returned, so other PREM tasks can enter their memory phases. This is also done after all blocks have checked out, to ensure that the GPU gives back the memory lock also for shorter not premized kernels (treated as memory phase). To enable the integration of the MEMguard call, some modification to the reloading of the high resolution timer had to be made. Since the call to the hypervisor is blocking, this time needs to be added to the newly configured WCET interval.

The high resolution timer is used to ensure that the GPU kernel do not overrun their given memory and compute phase times. After acknowledging the *Enter memory phase* request of the GPU the high resolution timer is configured to overrun after the estimated memory WCET time. If the running block did not request a *Enter compute phase* after the high resolution timer has fired, a memory phase overrun is detected. If a request was detected, the timer is updated with the estimated compute WCET. Again, if no phase change request was detected after the timer has fired, a compute overrun is detected. The detected overruns are returned at the end of the kernel execution and can be used to tune the WCET values for the kernel. Those values are not determined per phase, but per kernel, since constant memory phase and compute phase times are expected. After no more overruns are detected, the estimated values can be used for a correct synchronization between the host and GPU PREM tasks.

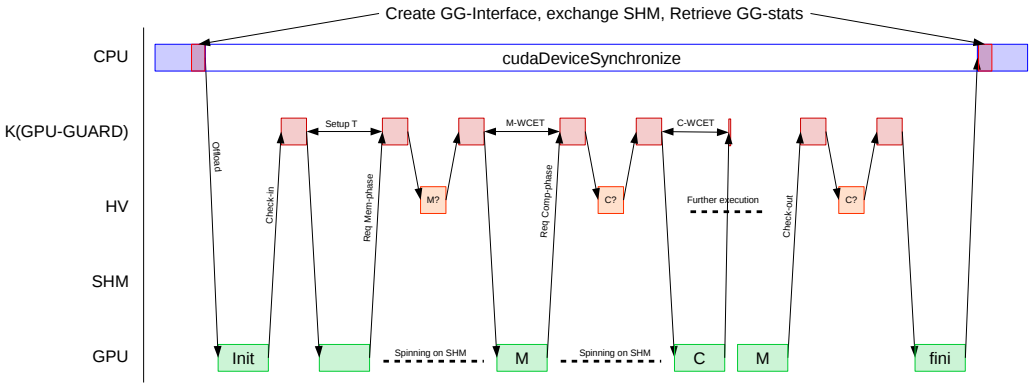


Figure 2.19: GPU-guard synchronization with HV

This way of PREM synchronization between the host and GPU introduces two main causes of overhead: First the shared memory communication is rather slow, since caches are disabled on host-pinned shared memory. But communication cannot be performed in other ways, this over head is not reducible. Secondly the WCET times are estimated over multiple runs (up to several 1000 if starting of WCET=100ns and adding 100ns on each detected overrun) of the kernel. This leads to a pessimistically estimated WCET and kernels that perform faster than the configured WCET time in the high resolution timer have to spin until this interval has expired. This could be reduced by using a higher sampling rate in the LKM to detect earlier finishes in the phases and acknowledging faster phases change requests.

2.7.2 PREM-Synchronization with GPUguard

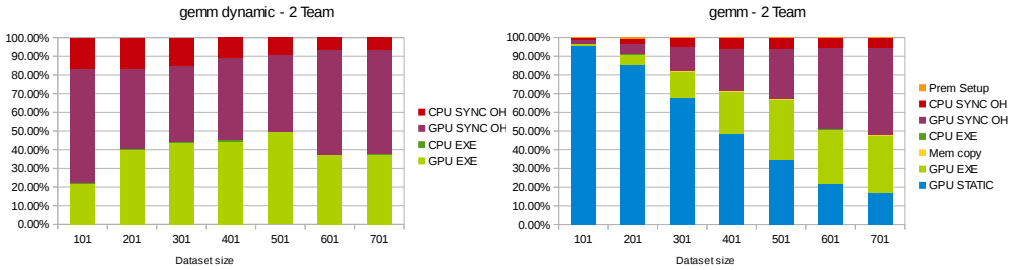
During the individual project we made some measurements to evaluate the synchronization overhead induced by the premization with the Hercules compiler and the synchronization with GPUguard. To measure this overhead we instrumented the premized executable to retrieve the times the program spends in the phases presented in table 2.1.

- CPU-time: Total execution time - kernel (including memcopy) time
- GPU-time: Using nvprof - kernel execution time
- Premized time: Time collected using binary which is premized by Hercules
- Notpremized time: Time collected using binary which was not premized by Hercules

Figures 2.20a and 2.20b show how much time an executable spends in the single phases compared to the total execution time. It can be seen, that the dynamic synchronization overhead is always around 50% which degrades the performance significantly. This delay is mostly induced due to the slow communication through the shared main memory.

Name	Desribtion	Measurement method
PREM setup	Time spend to setup PREM environment	In libpremnotify
CPU Sync overhead	PREM synchronization overhead introduced on CPU	Premized CPU time minus CPU time without premization
GPU sync overhead	PREM synchornization overhead introduced on GPU	Premized GPU time minus CPU time without premization
CPU execution	CPU execution without premization	Notpremized execution totaltime minus kernel time
Memcopy	Memcopy operations	Using nvprof
GPU execution	GPU kernel execution without premization (run target team)	Notpremized execution nvprof(run target team)
GPU static overhead	Static GPU overhead (init device and load binary)	nvprof

Table 2.1: Measured execution phases



(a) : GEMM: Dynamic overhead - 2 team **(b) :** GEMM: Static overhead - 2 team

2.7.3 Mem-guard

Memguard was introduced by Heechul Yun et al. [23]. It is used to provide isolation between PREM tasks during their memory phases. PREM tasks request Memguard to enter a memory phase. With this requests Memguard ensures that only one task on the system can enter a memory phase at a time. After the request is granted, Memguard monitors the memory usage of the running task using performance counters. If a core outruns its assigned memory budget the corresponding jobs is preempted and put to sleep. After a preset time instance the memory budget is renewed and the task is allowed to run further. Thanks to the use of performance counters, only cache misses are detected. Therefore a task is allowed to run further if it keeps hitting into is local cache. Memguard consists of two main parts, the per core regulator and the reclaim manager. The

monitoring of the memory usage and throttling of the running tasks is done in the per core regulator and the reclaim manager is responsible to maintain the global shared reservation (receiving and re-distribution) of the memory budget for all regulators in the system[23].

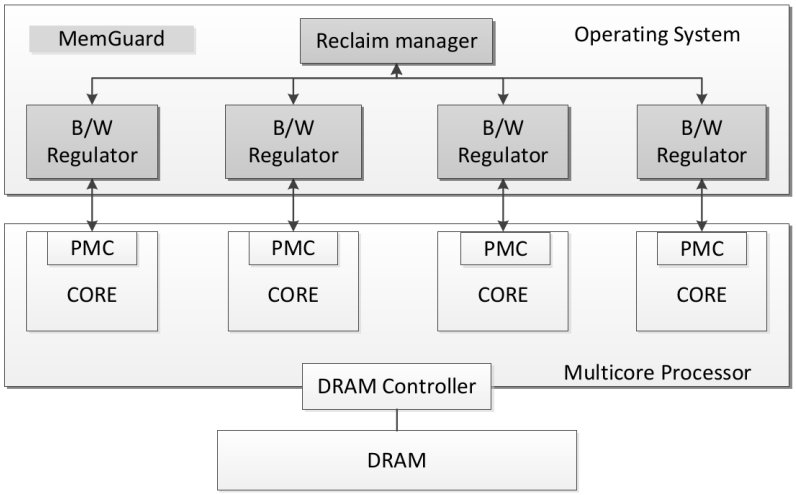


Figure 2.21: MemGuard system architecture. [23]

Chapter 3

Methodology

In this section, we present the used synchronization mechanisms and benchmarks. We ran all experiments on the Jetson TX2 in NV Power Mode MAXN and with all frequencies configured to the maximum values by running *jetsonclock.sh*[1] in order to get reproducible results. The TX2 was flashed with the Jetpack versions 3.1 (CUDA-8) and 3.3 (CUDA-9). First, we performed different artificial experiments based on sequential and random walks to analyze the caches on the GPU and their behaviour. We ran the walks in different kernel and thread configurations to characterize the interference on memory accesses. Later, we evaluated the possible synchronization mechanisms to schedule the PREM phases on the GPU, and then we measured the kernel to kernel interference based on a 2D Convolution example. To reduce the interference and therefore the induced execution jitter, we tiled the 2D Convolution into the PREM phases and scheduled the single tiles with the globaltimer available on both SMs. Table 3.1 gives a short overview of selected parameters describing the target platform.

Parameter	Value
OS	Ubuntu 16.04
Jetpack	3.1 and 3.3
Main memory	8 GB
CPU	NVIDIA Denver2 (dual-core)
	ARM Cortex-A57 (quad-core)
GPU	Pascal (2 SMs with 128 cores)

Table 3.1: Technical specification of the TX2

3.1 Platform characterization

In this section, we describe the used experimental setup to evaluate the cache sizes and the contention between the running threads on the TX2. All the experiments are based on random and sequential walks performing read-only accesses on a

[1]Script provided by NVIDIA to configure board clocks

shared array. During the walks, each thread records the average element access time. For these experiments, we evaluated the change of the average access time in function of the used dataset size, the number of congruently running threads and kernels, and induced interference from the CPUs.

3.1.1 SASS

Independent of the used time source to measure the average element access time, it is crucial to know how the instrumented code segment executes and that it includes the desired instructions. To allow the extensive parallelize on the GPU, NVIDIA had to relax the rules for data coherence and dependencies to allow a more radical reordering of instructions. This can lead to the problem that code segments we would like to measure can be moved out of the instrumented code region. To ensure that the measurement is performed correctly, one has to inspect the assembly output of the NVCC.

To compile a CUDA program with NVCC, the programmer has to specify the real and virtual target architecture. The virtual architecture (ex. `compute_62`) instructs NVCC to generate the Parallel Thread Execution (PTX) Assembly for this architecture. The PTX assembly can be seen as an intermediate, assembly-like representation of the CUDA kernels and is not heavily reordered yet. The compiler optimizes and assembles the PTX code for the target architecture if the programmer passes the real architecture parameter (ex. `sm_62`). In this step, heavy reordering and loop unrolling take place to optimize the executable for the target architecture. These optimizations are performed in different ways and with different results from CUDA version to CUDA version. Therefore it is important to inspect the resulting binary. NVIDIA provides the useful program `cuobjdump -sass prog` which shows the passed binary content in the form of a Streaming ASSembly (SASS) output. SASS is a human-readable output which allows the programmer to understand the order of the issued instructions.

To visualize to reordering problem we have a look at the simple kernel code shown in listing 3.1. At the beginning of the kernel, the current cycle counter value is read and stored into the variable temp1. Then a for loop is executed to fill in the `data_clock` array. After this for loop, the cycle counter is reread and stored in variable temp2. In the end, the two retrieved values are subtracted to get the total amount of cycles consumed by the for loop and is stored in global memory.

We compile this kernel now for the architecture of the TX2. Once by use of CUDA 8 (Listing 3.2) and once with CUDA 9 (Listing 3.3). On first sight, the shown SASS output seems to be similar between the two versions. For sure we have the same amount of instructions, but if we have a closer look, we see that the reordering took place differently. For the instructions filling the array in the for loop, this is not a problem, since we still have the same result. However, the problem lies with the retrieval of the cycle counts performed with the `CS2R Rx, SR_CLOCKLO` calls. In the example compiled with CUDA-8 we can see that the compiler reordered the code heavily and placed the two `SR_CLOCKLO` calls to the beginning of the kernel. This leads to the problem that the consumed cycles by the for loop are not recorded at all. With CUDA-9 on the other hand, the `SR_CLOCKLO` have not been reordered and the measurement takes place correctly.

Listing 3.1: Simple kernel for SASS generation

```
#define DSIZE 4
__global__ void kernel(unsigned int*data_clock){
    unsigned int temp1, temp2;
    temp1 = clock();
    for ( int i = 0; i< DSIZE; i++){
        data_clock[i] = i;
    }
    temp2 = clock();
    data_clock[0] = temp2 - temp1;
}
```

This example shows that it is essential to inspect the instrumented kernels to ensure that we measure the desired pieces of code. To have the same result with both CUDA versions in this example. It is sufficient to pass `DSIZE` as a kernel parameter so the for loop cannot be unrolled that extremely by the compiler.

Listing 3.2: CUDA-8 SASS generation

```
/*0008*/     MOV R1, c[0x0][0x20];
/*0010*/     MOV R0, c[0x0][0x140];
/*0018*/     CS2R R5, SR_CLOCKLO;

/*0028*/     IADD32I R2.CC, R0, 0x4;
/*0030*/     CS2R R0, SR_CLOCKLO;
/*0038*/     MOV32I R4, 0x1;

/*0048*/     MOV32I R7, 0x2;
/*0050*/     IADD.X R3, RZ, \
             c[0x0][0x144];
/*0058*/   { IADD R0, -R0, R5;
/*0068*/     STG.E [R2], R4; }

/*0070*/   { MOV32I R5, 0x3;
/*0078*/     STG.E [R2+-0x4], RZ; }

/*0088*/     STG.E [R2+0x8], R5;
/*0090*/     STG.E [R2+0x4], R7;
/*0098*/     STG.E [R2+-0x4], R0;

/*00a8*/     EXIT;
/*00b0*/     BRA 0xb0;
/*00b8*/     NOP;
```

Listing 3.3: CUDA-9 SASS generation

```
/*0008*/     MOV R1, c[0x0][0x20];
/*0010*/     CS2R R0, SR_CLOCKLO;
/*0018*/     MOV R2, c[0x0][0x140];

/*0028*/     IADD32I R2.CC, R2, 0x4;
/*0030*/     MOV32I R4, 0x1;
/*0038*/     MOV32I R5, 0x3;

/*0048*/     MOV32I R7, 0x2;
/*0050*/     IADD.X R3, RZ, \
             c[0x0][0x144];
/*0058*/     STG.E [R2], R4;

/*0068*/     STG.E [R2+0x8], R5;
/*0070*/     STG.E [R2+0x4], R7;
/*0078*/     STG.E [R2+-0x4], RZ;

/*0088*/     CS2R R5, SR_CLOCKLO;
/*0090*/     IADD R0, -R0, R5;
/*0098*/     STG.E [R2+-0x4], R0;

/*00a8*/     NOP;
/*00b0*/     EXIT;
/*00b8*/     BRA 0xb8;
```

3.1.2 Measurement overhead

To retrieve the time a program spends in a kernel CUDA offers three time sources: `clock()`, `clock64()` and the `globaltimer`. To see how much overhead is induced by the single measurement methods, we run a simple experiment that calls each time source two times and measures the time between the two subsequent calls. Depending on the used measurement method in the following experiments, we subtract this measurement overhead from the results. Table 3.2 shows the measured overhead. For the two clock sources, we did not measure the time of the call with the globaltimer. Therefore no information regarding the call duration in nanoseconds is available. The difference in nanoseconds between two subsequent calls to read the globaltimer ranges between 0 and 128 ns. This is due to the resolution of the globaltimer and exhibits the limitations that the globaltimer can only be used for bigger code segments.

Time source	Overhead in Cycles	Overhead [ns]
clock()	6	-
clock64()	55	-
globaltimer	-	0-128

Table 3.2: Measurement overhead

3.1.3 Cache analysis

To analyze the available caches and their size on the TX2, we launched a kernel to perform a random, respectively a sequential, walk on differently sized datasets. We launched the kernels with one CUDA-block consisting of one thread to ensure that no competition on memory occurs during the walk. Listing 3.4 shows how the kernel performs the walk. We used the `clock64()` function to retrieve the cycle count to avoid overflows of the 32bit cycle counter in the case of bigger datasets. For the two walks, we allocate an integer (4bytes) array of the desired length and fill each element with its index of the next element to create a closed loop. For the random walk, this linked list is shuffled to create one closed random loop.

By default, NVCC compiles the executable to cache global loads only in the L2 cache. This is useful since the L1 cache is used as fast storage for local kernel data if register spilling happens. If global loads would be additionally cached in the L1 cache, this could lead to more evictions for local data and slow down the execution. If desired, the programmer can configure NVCC to use L1 cache for global loads via passing the `-dlcm=ca` parameter to the compilation step.

In the Pascal GPU architecture shared memory and L1 cache do not share the same physical memory. To ensure that this is true, we run parallel (launched in different streams) to the kernel performing the walks dummy kernels spinning on the globaltimer for the duration of the walk and occupying the whole 64kBytes of shared memory of the two SMs. If shared memory and the L1 cache shared the physical memory, this would influence the element access times.

We perform this experiment for dataset sizes starting at 1kByte up to 1.5Mbytes in three configurations: i) Default settings during the compilations, ii) Enable L1 cache for global loads and iii) Enable L1 cache for global loads and occupy shared memory by kernels running in parallel. We repeated the measurement for each dataset size 100 times and then took the average element access times from those 100 runs.

Listing 3.4: Walk kernel

```
time_start = clock64();
for(int j = 0; j < params.buffer_length; j++){
    current = params.targetBuffer[current];
    sum += current;
}
time_end = clock64();
time_acc = (clock_t)(time_end - time_start);
// prevent optimization
*params.target_realSum = sum;

// Write element access time with measurement overhead
params.target_times[i] = (time_acc/params.buffer_length)-oh;
```

3.1.4 Interference between threads

To investigate the contention between multiple threads in a CUDA-block, we launched kernels to perform the random and sequential walks on a fix sized dataset. The changing parameter between the walks was the number of threads performing the walk in parallel. Starting from on thread and going up to 1024 (max. number of threads in a CUDA-block) threads performing the walk in parallel. We repeated this experiment for different dataset sizes: 1, 2, 3, 12, 16, 64, 128, and 256KBytes. Since the executable is compiled with the default NVCC settings, all datasets should easily fit into the GPU L2 cache.

The linking of the walks took place in the same way as described in section 3.1.3 Cache analysis. All threads performing the walk share the same array to iterate over the elements. However, each thread starts on a different element specified with its thread id (See Figure 3.1). If more threads then elements available participate in the experiment some threads start at the same index — this way each thread has to load its own element for each access.

Since the datasets used in this experiment are smaller than for the single thread walks, it is sufficient to use the `clock()` function without experiencing any cycle counter overflows. This provides the advantage that the measurement adds less overhead compared to the method using `clock64()`.

We repeated each measurement for ten times, and then we collected the average, minimum, and maximum element access times from all access times measured by the single threads.

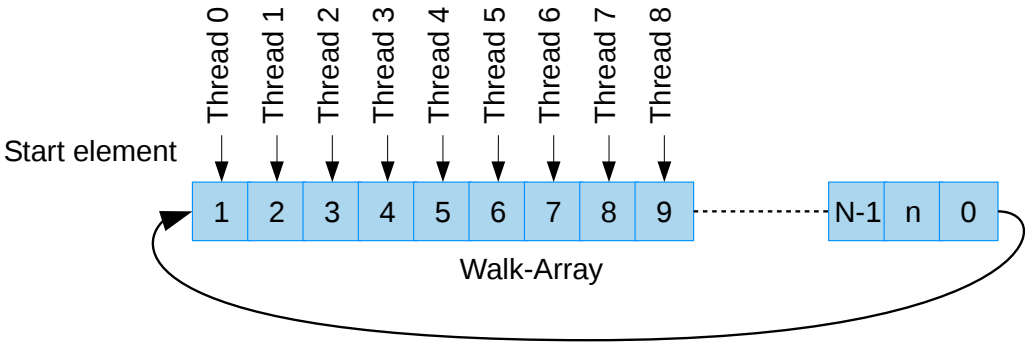


Figure 3.1: Starting elements of the participating threads to perform the walks.

3.1.5 Interference between kernels

Contention on the GPU happens not only inside a running kernel but also if multiple kernels are launched in parallel from different streams. It is desirable for the programmers to launch different kernels at the same time to increase the utilization of the GPU since this allows to use as much of the available resources as possible. However, since the running kernels have to access main memory, share the L2 GPU cache and other resources, this can lead to heavy contention.

We tried to characterize this competition on shared resources based on our random and sequential walk experiments. We fix the experiment parameters dataset size and number of threads per kernel for each experiment and express the average, minimum and maximum element access times of the threads collected in all running kernels in the function of the number of kernels executing the walks in parallel.

In this setup, each running kernel holds its copy of the dataset, and the threads inside running kernels always access the same element. That way, we can measure the overhead introduced by the interference between kernels and can reduce the interference between the threads running inside a kernel.

We repeated the experiment for the datasets sizes of 128kBytes, 256kBytes, and 512 kBytes to fill the L2 GPU cache with 1, 2, and four kernels running in parallel. For each dataset size, the kernels have been launched in configurations using 1, 32 or 128 threads.

As in the previous experiments, each participating thread was recording its average element access time for each repetition (we performed ten repetitions per setup). We took then the average, minimum, and maximum average element access time from the collected times to print them in the function of the number of participating kernels.

3.1.6 Interference CPU to GPU

Since the CPU clusters and the GPU share the main memory on the TX2, contention in memory accesses may happen. To see how this contention takes place on the GPU, we repeated the experiments described in the chapters 3.1.3 Cache analysis and 3.1.5 Interference between kernels with additional memory interference originating from the CPU.

The interference induced by the CPUs is created by the code segment shown in Listing 3.5. The CPU walks through an array and read and writes the elements. Before the experiment starts, this array is linked either sequentially or randomly, generating a closed loop. We have chosen the element size to be the size of a cache line (64 Bytes) - for each element, a new cache line needs to be fetched - and the dataset sizes to be 8Mbytes (CPU L2 cache is 2MB) to induce as much load to the memory controller as possible. We ran this interference on all CPU cores during the experiments running on the GPU.

In a preliminary experiment, we evaluated the impact of the CPU interference shortly by using the `tegrastats` tool provided by NVIDIA. We found that the sequential interference loaded the memory controller up to 70% and the random interference only up to 17%. Therefore it can be expected that the sequential interference has a more significant impact on the GPU execution.

Listing 3.5: CPU interference

```
while(*finishInference == 0){
     for (unsigned int i = 0; i < ARRAY_SIZE; i++) {
          sum += dst[current].index;
          current = src[current].index;
          dst[current].index += src[current].index;
     }
 }
```

3.2 Time-triggered GPU

In this section, we describe an approach to apply PREM to GPU kernels by the use of tiling. First of all, we evaluate different synchronization mechanisms which could be used to synchronize the PREM-phases. Then we evaluate the Polybench kernels based on their sensitivity to memory interference, to find the most appropriate kernel for the experiment. Eventually, we split the chosen 2D Convolution kernel into tiles to allow the application of the PREM phases for the tile processing.

3.2.1 Synchronization mechanism

A precondition for applying PREM to GPU workloads is the availability of fast synchronization. In our previous work, locks in shared memory were used to synchronize PREM phases on the CPU [1]. 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, a similar approach would be to use host-pinned mapped memory, which ensures non-cached zero-copy operation between CPU-GPU, to arbitrate main memory accesses [20].

Additionally to the synchronization through memory, the GPU on the TX2 offers a global nanoseconds timer called `globaltimer`. This timer runs synchronously on both SMs and is accessible through a 64bit register read operation. If the synchronization is based on this method, it will offer another advantage to synchronize with the time sources available on the CPU.

To evaluate an appropriate synchronization mechanism, we assessed the synchronization overhead between multiple concurrently running kernels using host-pinned zero-copy memory. Further, we measured the timer granularity of the globaltimer and checked that the globaltimer provides the same timestamps if read from different blocks and kernels.

3.2.2 Convolution benchmark

Polybench-ACC [2] is a collection of computational kernels such as matrix multiplication, 2D or 3D convolution, or linear equation solver, used to show the performance of software solutions such as compilers. Mentioned algorithms are the core of many high-performance applications such as neural networks or image processing. For our experimental part, we evaluated the sensitivity of all polybench kernels to memory interference from CPU and selected 2D convolution as a good candidate of a memory interference sensitive task. The compute complexity is $\mathcal{O}(n^2m^2)$ and the memory complexity is $\mathcal{O}(n^2 + m^2)$ where n is the dimension of square input and output arrays and m is the dimension of the square convolution mask.

3.2.3 Reduction of intra-GPU interference

To find and address the causes of unpredictable execution on the GPU, we performed several measurements with different kernel configurations. First, we evaluated the kernel-to-kernel interference on the original 2D convolution kernel from polybench-ACC. Then the implementation of the 2D Convolution was changed to a tiled version. 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 tiles in shared memory, which are first loaded from the global memory and at the end written back. This technique is commonly used to coalescence memory accesses in global memory to speed up the GPU execution [24, 25]. We take an advantage of this tiled implementation since it naturally splits into three PREM-phases: *prefetch*, *compute* and *writeback*. If *prefetch* and *writeback* phases are scheduled to have exclusive access to the main memory the *freedom from interference* paradigm is satisfied as long shared memory bank conflicts are avoided inside a CUDA-block.

3.2.4 Tiling

Figure 3.2 shows an example of how the 2D Convolution could be tiled to be launched by a kernel using two CUDA-blocks. We split the given dataset into multiple tiles fitting into the shared memory of the block. In the figure, we can see the first two tiles marked with the red and the green squares. Block0 will

process the red tile and Block1 the green one. We can further see that the two tiles overlap in the middle. This is because the convolution mask has the dimension of 3 to 3 elements, and therefore, the border elements need to be considered. For each tile, the block shifts the convolution mask (orange for Block0 and bright green for Block1) first along the X-axis and performs the convolution until it reaches the right end of the tile. Then the mask is applied on the third line of the tile. After the block has processed the whole tile, the next tile, starting two rows below, is loaded into the blocks shared memory to be processed. This procedure is repeated by the running blocks until all tiles have been processed. As we can see, the method of tiling the dataset naturally splits into the three phases prefetch (load data into shared memory), compute (process the data in shared memory) and writeback (write back the results from shared memory to the output array) for each tile.

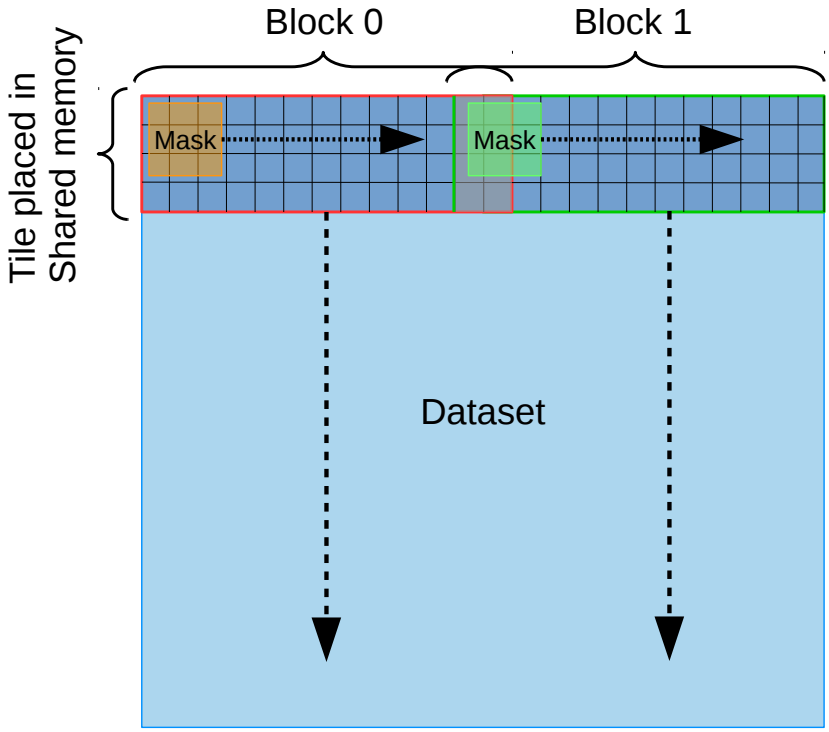


Figure 3.2: Tiling of the 2D Convolution

3.2.5 Tile scheduling

We performed several experiments to evaluate the interference between parallel running 2D Convolution Kernels. To evaluate how the interference and the resulting execution time jitter can be reduced, we implemented a simple scheduling mechanism, based on the globaltimer, to schedule either the whole tile processing or of the blocks or the individual phases of a tile. Figures 3.3, 3.4, 3.5 and 3.6 visualize how the scheduling takes place for one tile for the example of two kernels, consisting of two CUDA blocks, running in parallel. This step is repeated for the following tiles. We implemented four ways to schedule the tiles:

i) Whole tile scheduling kernel-wise – Figure 3.3 shows the concept of scheduling the whole tiles (Prefetch, Compute, and Writeback phases are rigidly connected). The term kernel-wise scheduling indicates that the blocks inside one CUDA kernel start their tile processing at the same time instance. For each tile, the scheduling start point is marked with the point SP_x. The scheduling offset PF_{off} is fixed by the programmer and specifies how much the tile processing of the two kernels is shifted. In case more then one kernel is used, the offset is calculated with the formula $PFoff_{kernelx} = Kernel_{id} * PF_{off}$. If a block has finished its current tile, it waits until the hyper period T_{hyper} is expired and then adds the offset $PFoff_{kernelx}$ to the next starting point SP_{x+1}. We define the hyper period with $T_{hyper} = nof_{kernels} * PF_{off}$ with the minimum length of one tile period. Finally, we can define the start time of the tile processing with:

$Tile_{start_{kernelx}} = Start_{kernelx} + WU + (n-1) * T_{hyper} + PF_{off_{kernelx}}$, where n is the current tile id, and WU is the warm-up period of the first tile to fill the instruction cache.

ii) Whole tile scheduling block-wise – The block-wise scheduling takes place the same way as the kernel-wise scheduling, with the difference, that the tile-processing of all participating blocks is shifted. Therefore we redefine $PFoff_{kernelx} = ((Kernel_{id} * nof_{blocks_{kernelx}}) + bockid) * PF_{off}$. Figure 3.4 shows the block-wise shifting. We did not show the warm-up phase due to simplification reasons. This scheduling approach offers the possibility to additionally reduce the interference between the blocks running inside a single kernel.

iii) Phase scheduling kernel-wise – To be able to evaluate the interference between the PREM phases, we implemented another mechanism to shift the individual phases. In this setup, the prefetch and compute phases are scheduled together, and the writeback phase can be scheduled alone. Figure 3.5 shows the example of kernel-wise scheduling. Again, for the sake of simplification, we removed the warm-up period in this figure. We can see, that the scheduling reference point for the prefetch and compute phases is called SP_x as in the previous example. Additionally, we can see the schedule start point for the writeback phases called $WBSx$. The two shifting offsets for the prefetch, and writeback phases are called PF_{off} and WB_{off}. The two offsets are specified by the programmer and WB_{off} can be at shortest $T_{WB}/(nof_{kernels} - 1)$. The constants T_{PF}, T_C and T_{WB} represent the measured duration of the PREM phases. T_{WBoff} is defined as $T_{WBoff} = (nof_{kernels} - 1) * PF_{off} + T_{PF}$. Therefore we define $T_{hyper} = T_{WBoff} + (nof_{kernels} - 1) * WB_{off} + T_{WB}$. With this definition, we allow the writeback phase to be scheduled during another compute phase. Now we can define the Prefetch phase start point of the current tile as:

$PF_{start_{kerx}} = Start_{kerx} + WU + (n-1) * T_{hyper} + ker_{id} * PF_{off}$

and the start point of the writeback phase as:

$WB_{start_{kerx}} = Start_{kerx} + WU + (n-1) * T_{hyper} + T_{WBoff} + ker_{id} * WB_{off}$, where n is the current tile id, and WU is the warm-up period of the first tile to fill the instruction cache.

iv) Phase scheduling block-wise – The block-wise phase scheduling is performed the same way as the kernel-wise scheduling, but the phase offsets are calculated per block in the system. This leads to the start points:

$PF_{start_{kerx}} = Start_{kerx} + WU + (n-1) * T_{hyper} + ((Ker_{id} * nof_{blocks_{kerx}}) + bockid) * PF_{off}$
and the start point of the writeback phase as:
$WB_{start_{kerx}} = Start_{kerx} + WU + (n-1) * T_{hyper} + T_{WBoff} + ((Ker_{id} * nof_{blocks_{kerx}}) + bockid) * WB_{off}$, where n is the current tile id, and WU is the warm-up period of the first tile to fill the instruction cache.

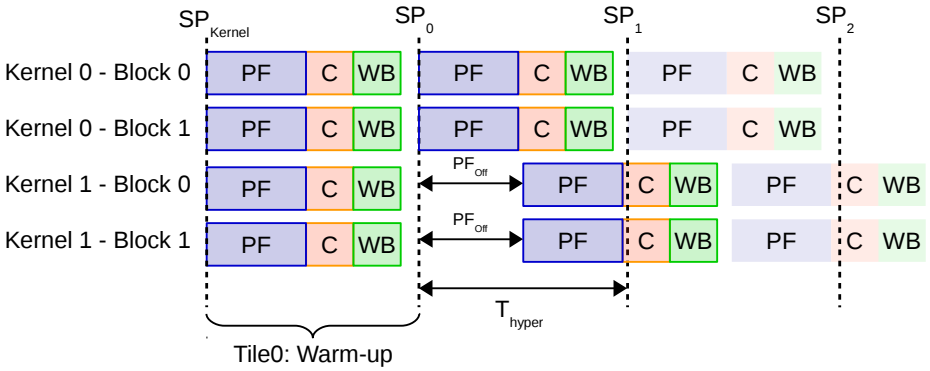


Figure 3.3: Tile scheduling – Kernel-wise

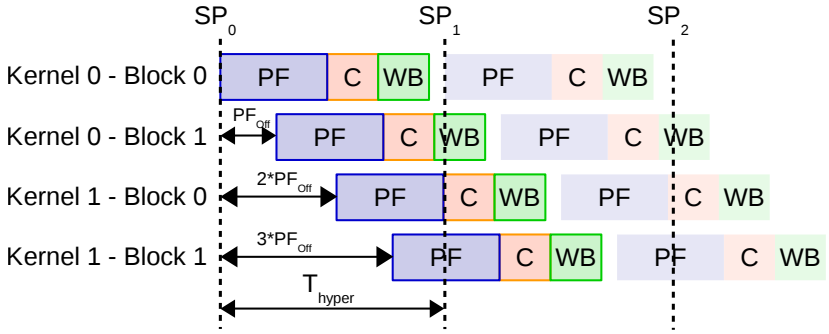


Figure 3.4: Tile scheduling – Block-wise

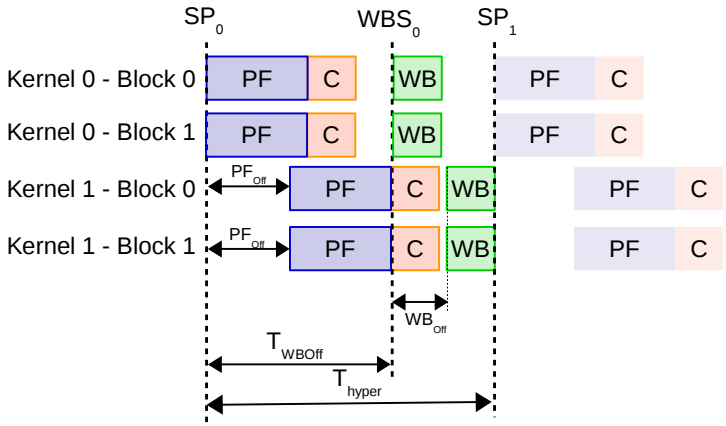


Figure 3.5: Phase scheduling – Kernel-wise

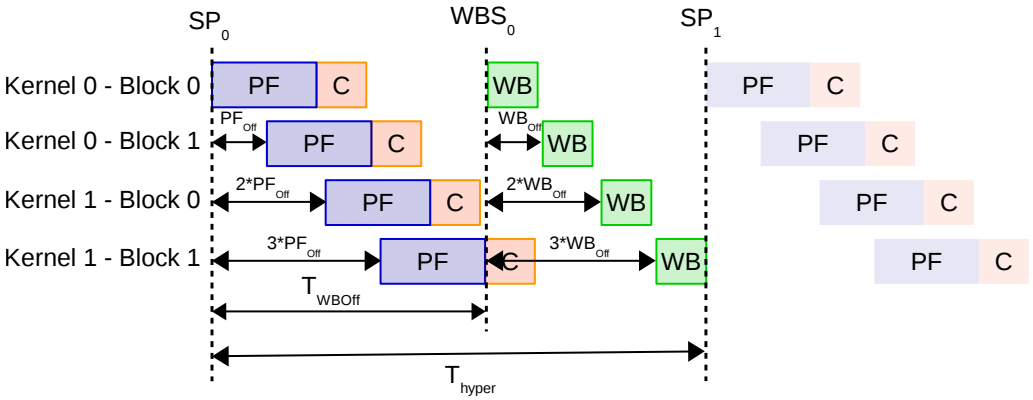


Figure 3.6: Phase scheduling – Block-wise

3.3 KCF-Tracker interference analysis

To apply the Premization to a less artificial example than the 2D Convolution benchmark, we performed some initial measurements on the KCF tracker[1] implemented by Vít Karafiát and Michal Sojka [4] to see how the launched kernels, computing the complex matrix operations, interfere with each other. To measure this interference, we run the kernels in standalone and with one to three interfering kernels in parallel. To ensure that a single kernel does not consume the whole GPU thread resources (2048 threads per SM) we wanted to launch each kernel with two CUDA-blocks à 512 threads. With this setup, we ensure that four kernels run in parallel: $512 * 2 * 4 = 4096$ Which equals the number of threads that can be assigned to both SMs at a time instance. Therefore, we had to change the KCF-tracker kernels to use grid-stride loops, which allows launching the kernels with fewer threads than elements.

We test the interference between the sqrt-norm, sqrt-magnitude, conjugate, sum-channels, matrix multiplication, matrix division, matrix addition, matrix multiplication with a constant, matrix addition with a constant and matrix element multiplication kernels. Each of the listed kernels is run in parallel of 0 to 3 instances of the other kernels. The tested kernel is instrumented to record its block start and end times to retrieve the total kernel execution time: $T_{kernel} = Block_{maxEndTime} - Block_{minStartTime}$. Each measurement is repeated 1000 times, and the average execution time, the minimum and the maximum execution times are collected and stored. With the collected data we created a heatmap to show the influence between the kernels.

We did not extend the execution time of interfering kernels if the tested kernel has a longer execution time to be as close as possible to the execution times inside the tracker.

[1] https://github.com/CTU-IIG/kcf

Chapter 4

Experimental evaluation

In this section, we find the results and their discussion from the described experiments. All source code we used for the experiments can be found in the git repository: `https://github.com/CTU-IIG/tt-gpu`

4.1 Kernel interference

We evaluated the cache structure and the interference between multiple GPU kernels running in parallel based on different runs of random and sequential walks. In the first experiments, we measured the average element access times of the kernels launched with one CUDA-Block and one thread performing the walks on differently sized datasets.

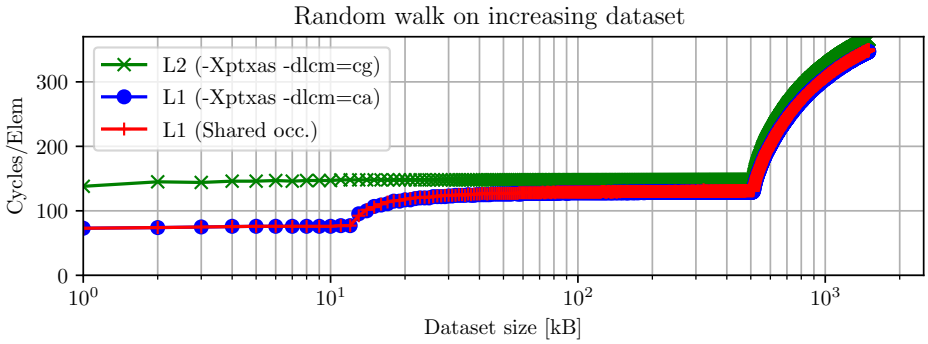
In the later experiments, we evaluated how the average access time is influenced if multiple threads and multiple kernels perform these walks at the same time. In the end, we assessed the impact of memory interference induced from the CPU clusters on the walks.

4.1.1 Cache analysis

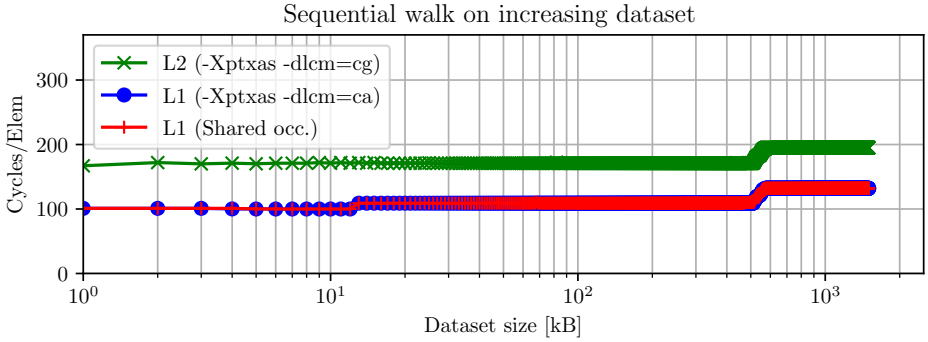
As we described this experiment in section 3.1.3, we measure the average element access time based on random and sequential walks on differently sized datasets. Figures 4.1a and 4.1b show the results for the different cache configurations if a random and sequential walk is performed. For the experiment enabling the L1 cache for global loads, the cache exhibits a size of 12kBytes. Moreover, for the experiment occupying the shared memory, we can see, that this does not influence the average element access time for the L1 cache. This indicates that the shared memory and the L1 cache indeed do not share the physical memory location.

Further, the L2 cache exhibits a size of 512kBytes which is also confirmed by the technical specifications. We can see this by the increase of the element access times for the datasets bigger than 512kBytes.

Even though the measured average element access times are relatively big, this appears to be reasonable, since in the GPU not the single memory accesses are optimized to have short latency, but if a latency appears, it is hidden by other threads performing work in parallel.



(a) : Random walk



(b) : Sequential walk

Figure 4.1: Revealing Cache hierarchy based on random and sequential walks performed with one thread each.

4.1.2 Walks - Multiple threads

This experiment evaluates the contention between multiple threads within a CUDA-block. An integer (4 bytes) array is linked that each element contains the index of the next element to access for the sequential walks. For the random walks, we performed the randomization that only one big closed loop is created to iterate through the whole range. Each thread inside a kernel traverses this array during one run and collects its average element access time. We repeated this experiment for different dataset sizes and a changing number of threads. Each thread in the block starts at a different element of the array and performs the walk through the whole range. If more threads then elements are available, multiple threads may start at the same element. From the average access times of all participating threads, we took the average, minimum, and maximum number of cycles to access one element. Figure 4.2b shows the result of the different runs of the random walks on different dataset sizes. The X-axis represents the number of threads performing one walk in parallel, whereas the Y-axis visualizes the average element access time. One can see, how the access time increases for all datasets linearly up to the point where 32 threads are used. Between 32 and 64 threads the average access time is reduced again. We expect this to be due to the warp-scheduling in the GPU since one fully occupied warp with 32 threads and an only partly occupied warp with one to 31 threads are running in parallel. The threads in the partly occupied warp can finish their walk significantly earlier, which leads to a

decrease in the average access time of one element. Between 64 and 256 threads the slow-down remains mostly stable. After this point, the average access time starts to increase linearly again.

We expect this slow-down only to be partly related to the contention on caches since all the used dataset sizes should fit into L2 cache. If the L2 GPU cache uses a random replacement policy, the threads might evict cache lines from each other even if the used dataset should fit into L2 cache. Further, a regular pattern appears for the minimum and maximum access times per thread if more than 256 threads are performing the walk in parallel. This pattern exhibits a more significant jitter for all odd 32 number of threads whereas the jitter is almost zero for even 32 number of threads. Therefore this overhead and jitter might additionally be influenced by the warp-scheduling. This measurement indicates that it is desirable to use a multiple of 64 threads in a block to reduce execution jitter between the single threads and therefore reduce the maximum execution time of the running block.

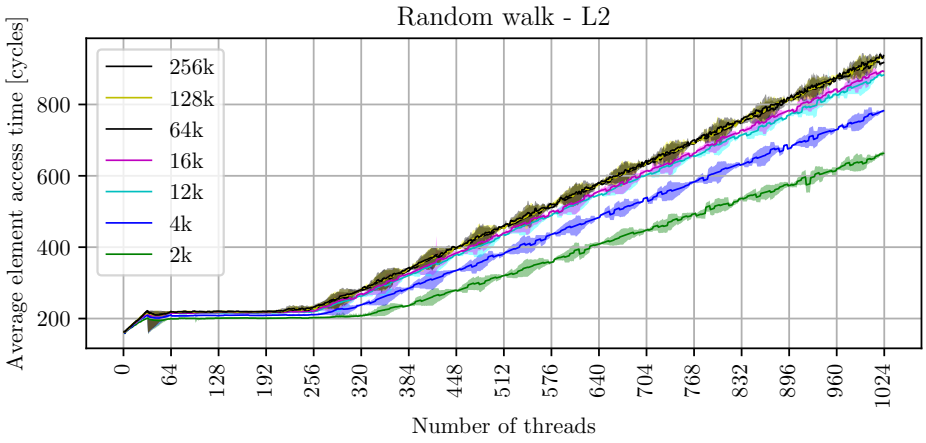
Figure 4.2a on the other hand, does not show such a pattern on the average element access time for the sequential walks. We can see, that the contention between the running threads remains relatively stable for all dataset sizes. This is expected since the GPU architecture is designed to support such streaming memory accesses by the participating threads.

4.1.3 Walks - Multiple kernel

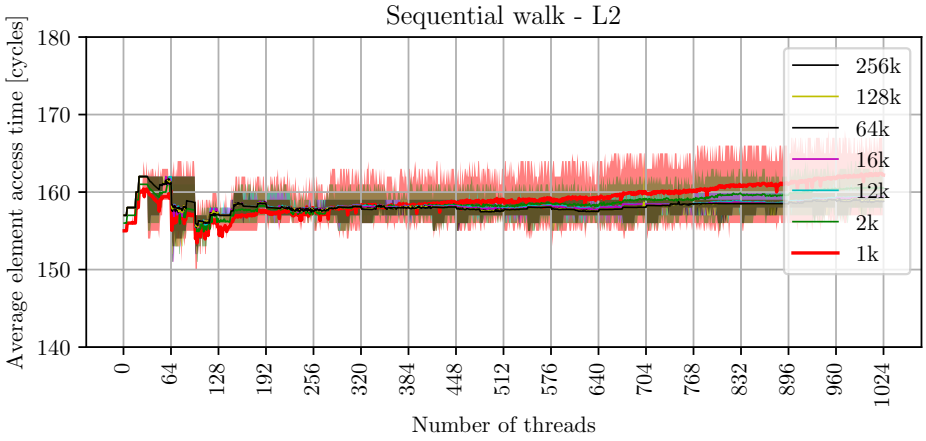
Concurrent execution of multiple kernels is an important feature to utilize the available resources efficiently. Inevitable property of simultaneously executed kernels is, that contention in memory accesses can occur. To evaluate the effect of this contention, we ran multiple kernels performing random and sequential walks in parallel.

The maximum number of kernels launched in parallel is 16 since we observed with nvprof that if more kernels from different streams are launched, they are not executed in parallel. Each kernel consisted of one block with 1, 32, and 128 threads, and the walks were performed on datasets with the size of 128kBytes, 256kByte, and 512kByte and holds its copy of the linked dataset. All threads start their walks on the same element to reduce contention inside a kernel. The walk is performed in the same way as described in experiment 3.1.5 *Interference between kernels*. In the case of the sequential walk, the threads iterate through the array element by element. For each thread in the kernels, the average number of cycles to access an element is recorded. From all the measured access times, the average, minimum, and maximum access time is calculated and presented in Figures 4.3a and 4.3b.

In the case of the random walk, the execution on the 512 kB dataset experiences a slowdown already with the launch of the second kernel. Since the GPU L2 cache on the TX2 has a size of 512 kBytes, it is not possible to fit the datasets of both kernels into this cache. Therefore the kernels evict cache lines from each other. The same happens for the experiments using the 256 kByte dataset for 3 or more kernels and the 128 kByte dataset for 5 or more kernels running in parallel.



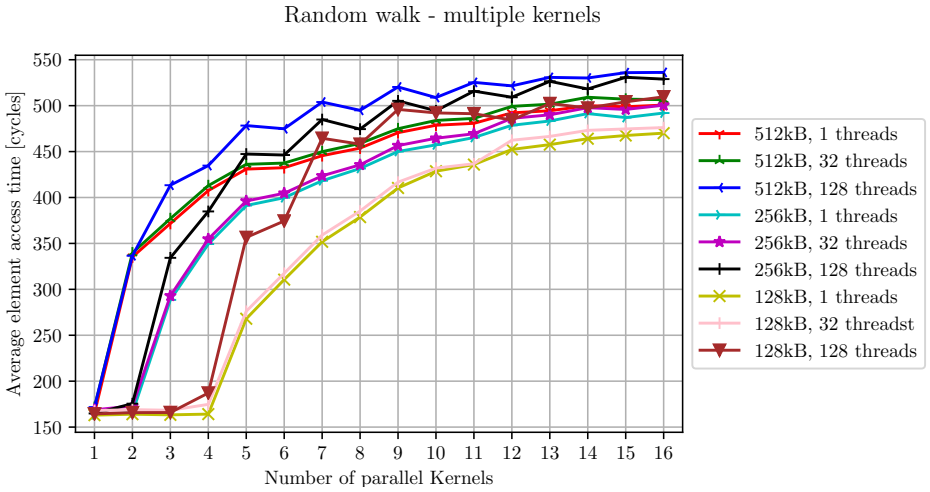
(a) : Sequential walk



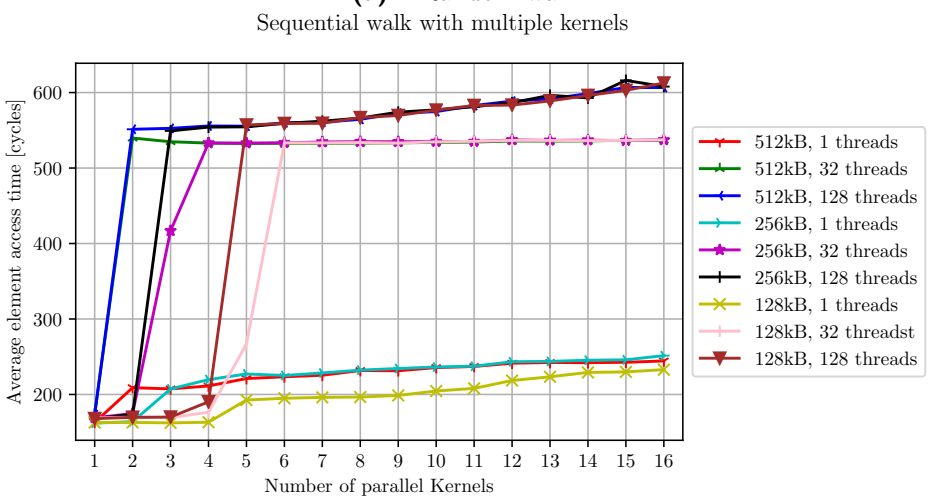
(b) : Random walk

Figure 4.2: Walks performed by multiple threads in one block on datasets with different sizes. Each thread accesses a different element on its start and records its average element access time. The transparent background shows the minimum and maximum times a thread took to finish its walk.

The sequential walk is shown in Figure 4.3b and experiences the slow-down at the same number of kernels running in parallel. Compared to the random walk experiment the average element access times are much higher (ca. 580 cycles compared to 350 cycles for the random walk) as soon as the datasets of the kernels do not fit into L2 cache anymore. This is an indicator that sequential memory accesses exhibit a higher competition on caches between multiple kernels. It the kernels are launched with only one thread each performing the sequential walk, this contention remains quite low.



(a) : Random walk



(b) : Sequential walk

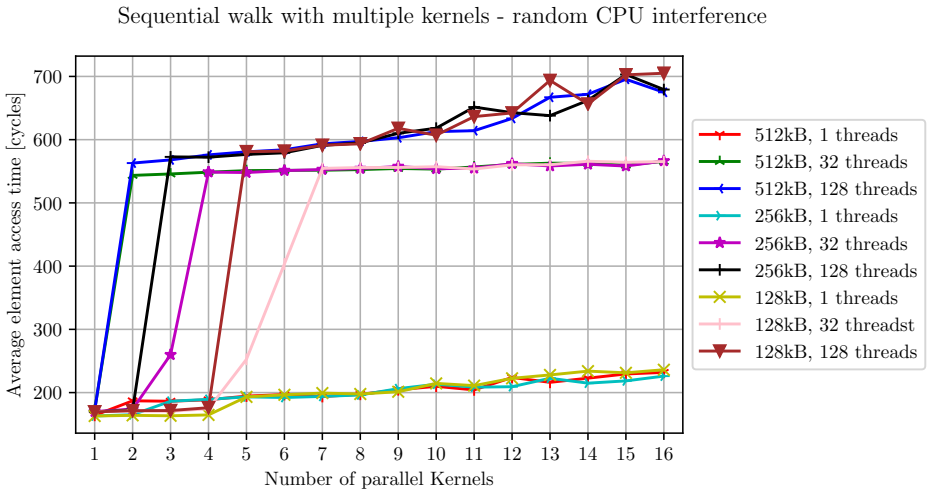
Figure 4.3: Walks performed by up to 16 kernels in parallel. The kernels consist of 1, 32 and 128 threads and perform the walks on the datasets with the sizes 128, 256 and 512 kBytes. The transparent background shows the minimum and maximum times a thread took to finish its walk..

4.1.4 CPU interference

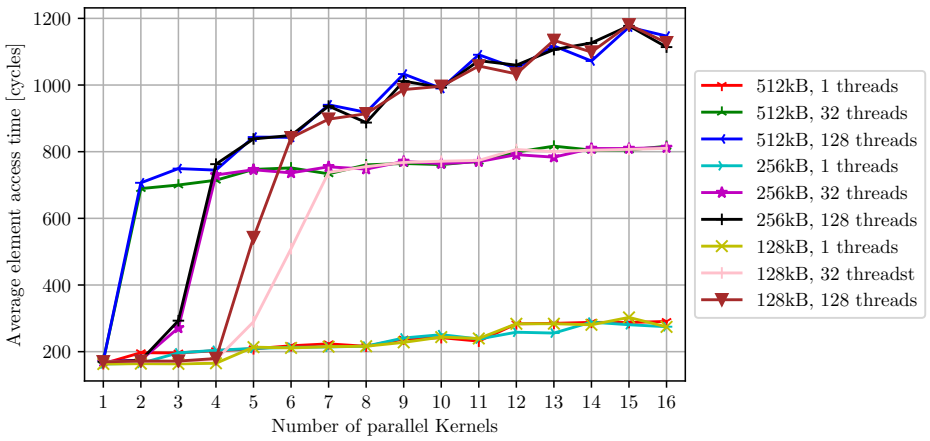
Walks multiple kernels

We repeated the experiments running the sequential walks from different kernels with additional competition coming from the CPU. Since the GPU and CPU on the TX2 do not share a cache but main memory, this competition happens in the memory controller. We performed two experiments with all six cores on the CPU performing either random or sequential memory read and writes during a sequential walk. Figures 4.4a and 4.4b present the results of these two experiments. If we compare the two figures to Figure 4.3b, we can see, that the random CPU interference adds a much smaller overhead compared to the sequential CPU

interference. This is also highlighted if *tegrastat* [1] is run during the experiments which showed that the random interference utilized the memory-controller to around 17% whereas the sequential interference reached a utilization up to 70%. These experiments provide an insight into the competition on main memory and indicate that sequential memory accesses produce a bigger competition.



(a) : Random interference



(b) : Sequential interference

Figure 4.4: The walks are performed by multiple kernels in parallel. During their execution, the CPU introduced sequential and random memory interference to add additional competition to the main memory accesses.

Walks single thread

To investigate in more detail the influence of the CPU interference to the average element access time, we repeated the Cache analysis experiments shown in section 4.1.1 with additional interference induced from the CPU. To extend this experiment,

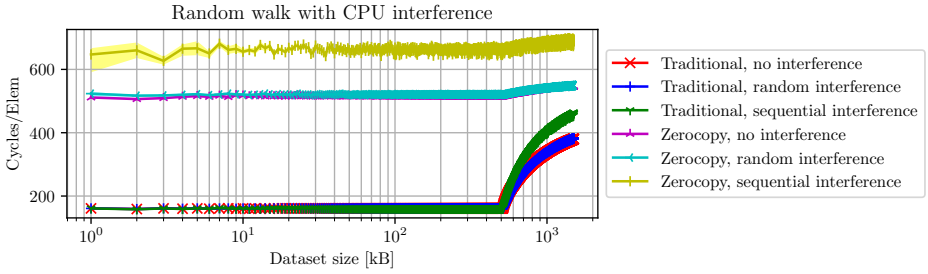
[1] System monitoring tool provided by NVIDIA.

we performed the walks also on host pinned Zerocopy memory. Further, we compiled the executable with default settings, which means that global loads are only served through the L2 cache.

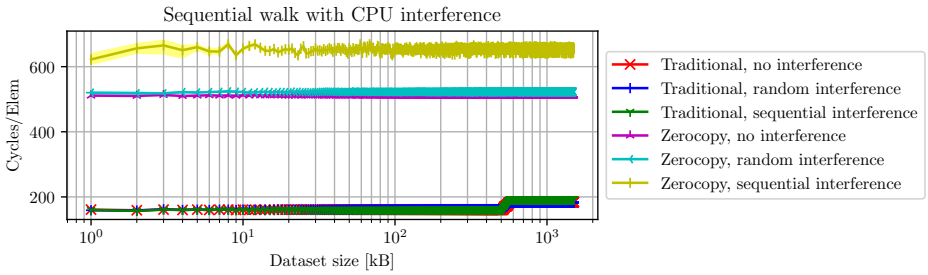
We can see in Figures 4.5a and 4.5b that the CPU interference only influences the average access times if the datasets do not fit into the L2 cache of the GPU. This makes sense since the CPU and the GPU only share the main memory, and therefore the contention happens on the memory bus. This effect is especially more dominant for the random walks since the SM needs to fetch a new cache line for each random element access and can profit in the case of the sequential walk from elements already staying in the cache.

We can see that the Zerocopy memory version exhibits much slower element access times even if no interference is present. We can see in the case of the random walk slightly increasing access times for datasets bigger than 512kBytes. But not as much as in the case of the traditional memory model. This indicates that the Zerocopy memory is not or only party cached on the GPU. This is underlined with the observation in the case of the sequential walk, where the access times remain constant for all dataset sizes.

If we look at the influence of the CPU interference in case of Zerocopy memory, we can see that the interference induced from the PCU influences the average element access times for all dataset sizes which supports the assumption, that Zerocopy memory is not cached at all.



(a) : Random walk



(b) : Sequential walk

Figure 4.5: Walks performed with one thread each. L2 cache only for global load, traditional and Zerocopy memory is evaluated. We repeated those experiments with additional memory interference from the CPU side.

4.2 Experiments for time-triggered GPU execution

In the following experiments, we evaluate the contention on the GPU based on the 2D Convolution kernel of Polybench, evaluated an appropriate synchronization mechanism for the PREM phases and performed some first scheduling experiments with the tiled version of the 2D Convolution. This part of the experiments was part of the accepted paper *Experiments for Predictable Execution of GPU Kernels* for the OSPERT19 conference.

4.2.1 Zero-copy memory synchronization evaluation

We evaluated synchronization based on locks in zero-copy 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). As mentioned in 3.2.1, these times are sufficient for synchronizing PREM phases on the CPU, but since PREM phases on the GPU are expected to be in the range of 1 to 4 µs, the overhead of this synchronization mechanism is deemed too high.

4.2.2 GPU timer granularity

We evaluated the globaltimer as a synchronization mechanism between GPU tasks. According to the documentation [26], the globaltimer should have a resolution in the nanoseconds 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 streaming multiprocessors. To evaluate these properties, we ran a kernel from Listing 4.1 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 4.6 shows a zoom into the first few iterations of collected timestamps collected by the first block of the launched kernel. Although not shown in the figure, we observed that the other blocks running during the kernel execution on both SMs retrieved the same timestamps synchronously. Running the experiment in the default settings gives disappointing results. The measured resolution was only 1 µs. The “Default” points on the left side show the timestamps collected by block 0. The right side of the figure shows the histogram of the differences between two subsequent timestamps. For the “Default” setup, it is clearly seen

Listing 4.1: 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]));
```

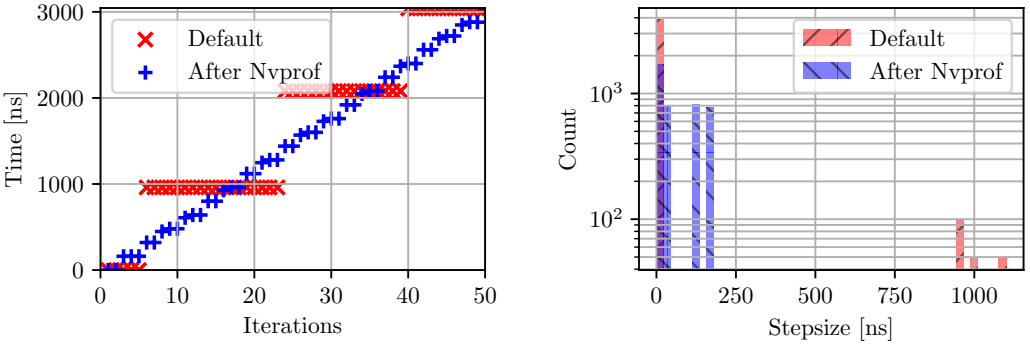


Figure 4.6: Timestamps and step sizes of the globaltimer after reboot and after one run of nvprof for one block. The retrieved timestamps of the other blocks exhibited the same resolution.

that the resolution of the globaltimer is around 1 µs. 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. 4.6. 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.

4.2.3 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 taken. 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

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

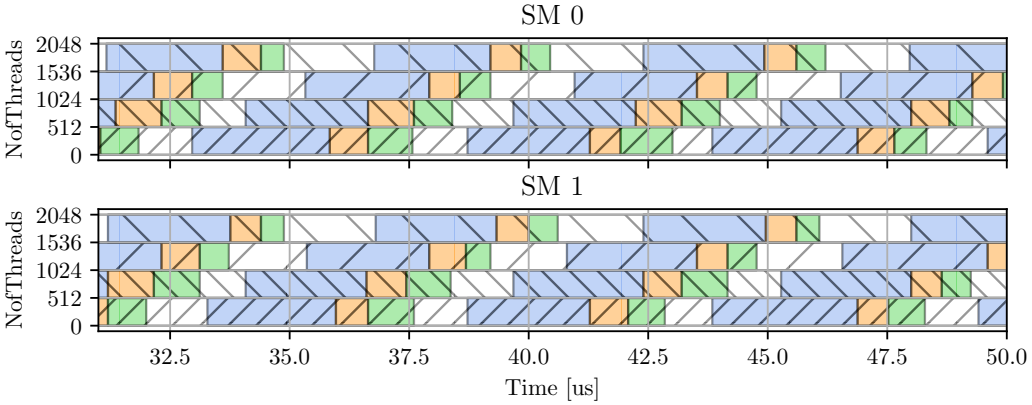


Figure 4.7: This figure shows a zoom into the execution of the tiled kernel. The total execution time is 2.8ms. Whole tiles scheduled against each other with an 1.4 µs offset on both streaming multiprocessors. The blocks within a kernel are scheduled at the same time instance. Blue, orange and green 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 end, the resulting data is written back to global memory. This procedure is also shown in listing 4.2. 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 4.7 shows how 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.

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 different offsets (as in Fig. 4.7).

Figure 4.8 and Table 4.1 show the average execution time and execution jitter of the scenarios. For the sake of simplicity, we show only the tile scheduling with the offsets of 1.3 µs and 1.4 µs since at this point, the jitter starts to be reasonably reduced. 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 taken as the time interval where all kernels finished their work, called scenario execution time. 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

Listing 4.2: Tile scheduling pseude code

```
static __global__ void kernelTiled(data)
  {
  // Allocate shared memory
  __shared__ float A_SHM[PREM_SHM_SIZE];
  __shared__ float B_SHM[PREM_SHM_SIZE];

  // Spin until PREM schedule start time
  spinUntil(start_time);
  __syncthreads();

  for tile_id=block_index; \
      tile_id < number_of_tiles; \
      tile_id + number_of_blocks:
    syncPrefetch(start_time,\
                 tile_id,\
                 kernel_id,\
                 TILE_OFFSET);
    __syncthreads();

    /* ---------------------------------- */
    // Prefetch data
    // from global memory into A_SHM
    /* ---------------------------------- */
    __syncthreads();

    /* ---------------------------------- */
    // Compute on SHM
    /* ---------------------------------- */
    __syncthreads();

    /* ---------------------------------- */
    // Write back data
    // from B_SHM to global memory
    /* ---------------------------------- */
}
```

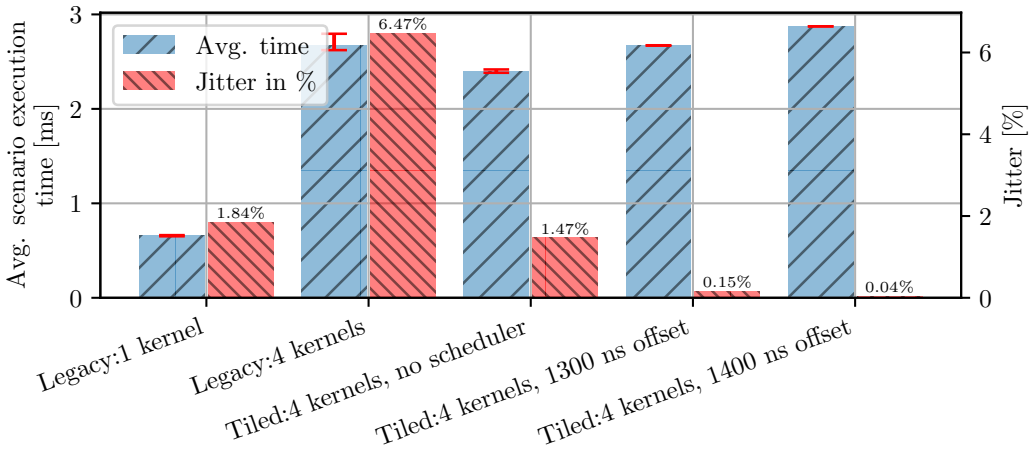


Figure 4.8: Comparison of scenario execution time. Tiles are scheduled against each other.

configuration. The Worst observed execution time (WOET) is still slightly shorter than the WOET of the single kernel version executed four times in a row, but the execution jitter is around 6.47% of the average execution time.

We can see that the tiled implementation with four 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.

To have a more elaborate overview of the influence of the tile scheduling offset to the observed execution jitter, we refer to Figure 4.9. We can see with the blue dots the average scenario execution time and with the red dots the corresponding execution jitter. The dotted black line represents the average scenario execution time of the baseline (for 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 this observation, we classify the tile offsets of 1.3 µs and 1.4 µs as usable to reduce the execution jitter while still having a usable scenario execution time.

4.2.4 Phase evaluation

As shown previously, execution jitter can be reduced if the start times of the tile processing are shifted against each other. As a next step, we evaluate how the

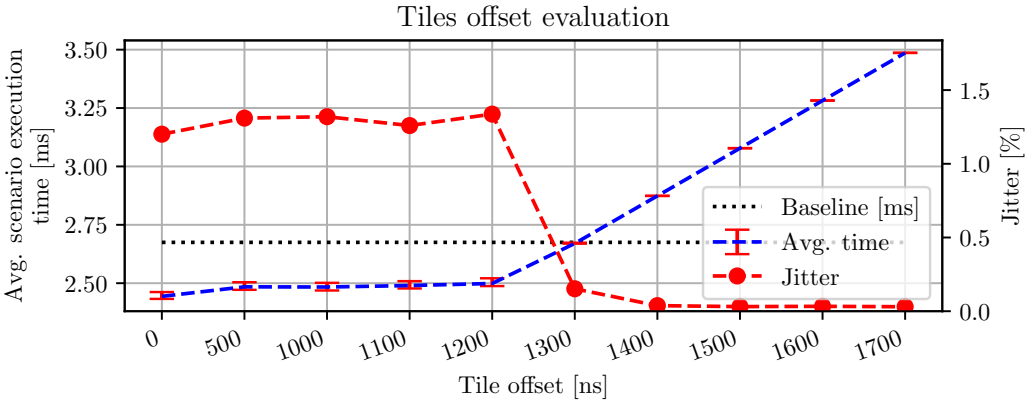


Figure 4.9: Evaluation to find an appropriate tile offset for the 2D Convolution. The black line represents the baseline average execution time the four legacy kernels took to complete.

Scenario	Average [ms]	Jitter [%]
Legacy, 1 kernel	0.657	1.84
Legacy, 4 kernels	2.675	6.47
Tiled, 4 kernels, no scheduler	2.397	1.47
Tiled, 4 kernels, 1300 ns offset	2.67	0.15
Tiled, 4 kernel, 1400 ns offset	2.874	0.04

Table 4.1: Average execution times and jitter corresponding to Figure 4.8

PREM phases influence each other if no schedule is used. Figure 4.10 shows on the left the CDF plot of the measured single phase times of the tiled 2D convolution benchmark launched in one kernel with one block. We chose this configuration to ensure that no contention between the blocks inside the kernel occurs. The phase durations are still the same since we did not change the tile size. All phases show an outlier in the first iteration since the processing of the first tile suffers from instruction cache misses. If this warm-up iteration is removed, the average phase times are 1.781 µs, 0.506 µs and 0.421 µs for the *prefetch*, *compute* and *writeback* phases.

In contrast to this isolated measurement, Figure 4.10 shows on the right the CDF plot when the tiled version is launched with four kernels running in parallel. Even if the warm-up iteration, in the beginning, is removed, the WOET times are 3 to 6 times longer compared to the phase times without any interference. If one takes the WOET times of the single phases (PF: 7.68 µs, C: 3.2 µs and WB: 2.88 µs), sums them up to a whole WOET tile and multiplies this WOET tile with the total number of tiles (512), one would end up with WCET of around $(WOET_{PF} + WOET_{C} + WOET_{WB}) * N_{tiles} = 7.045$ms. This number represents a very pessimistically estimated WCET, but we have to consider it since we have no detailed model of the GPU.

To evaluate in more detail how the phases interfere, further experiments were

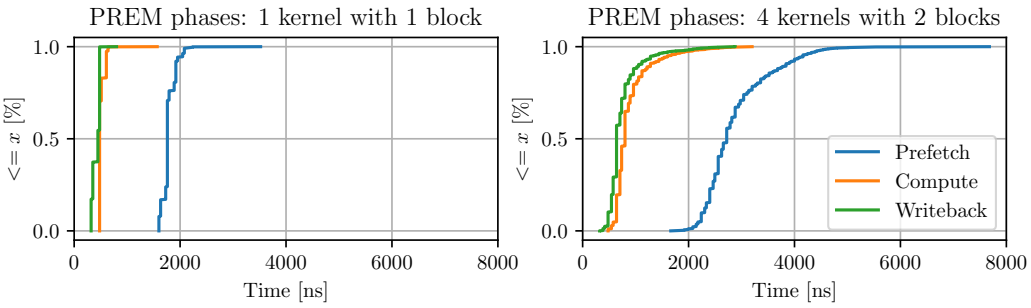


Figure 4.10: CDF of phase execution times without a scheduler

performed. In these experiments, we measured the average phases times, which is not directly linked to the total scenario execution time since multiple tiles, each tile consists of the PREM phases, are processed during one kernel execution – even if the measured jitter of the single phases is high, this can sum up to a lower scenario execution jitter as shown in the previous experiments.

Figure 4.11 shows the result when the *prefetch* and *compute* phases are shifted against each other. The *writeback* phase was scheduled later to not run concurrently with the first two phases. Again, the blocks in one kernel start at the same time instance to process their current tile. In Figure 4.11, the average compute time bars are stacked on top of the average prefetch time bars. The bars on the right represent the execution jitter of the two phases compared to the total average execution time (PF + C). One can see that the average 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 4.12 shows the results of executing only the *writeback* phases shifted against each other. Similarly to the *prefetch* phases, the less the *writeback* phases overlap, the more the execution time is reduced. The execution jitter is reduced after a shift of 400 ns.

In the experiments shown in Figures 4.11 and 4.12, the two blocks of each kernel had the same start times to start processing their current tile. This lead to possible contention between the two blocks. Therefore, we repeated the experiments with all blocks having different start times to processes their tiles. This reduced the contention even more with the result of a similar average phase time but reduced phase execution jitter as shown in Figures 4.13 and 4.14. In the case of 3 µs shift, the prefetch jitter was reduced by $\approx 17\%$ and the compute phase jitter by $\approx 48\%$. Similarly, the *writeback* phase execution jitter was reduced by $\approx 25\%$ for the phase-shift offset of 1 µs. Even if the phase execution jitter is reduced, the additional shift between two blocks in the same kernel introduces an extension of the average total kernel execution time.

When we compare the above-described results with our previous application of PREM on the ARM CPUs of the Jetson TX1 [1], the *prefetch* and *writeback*

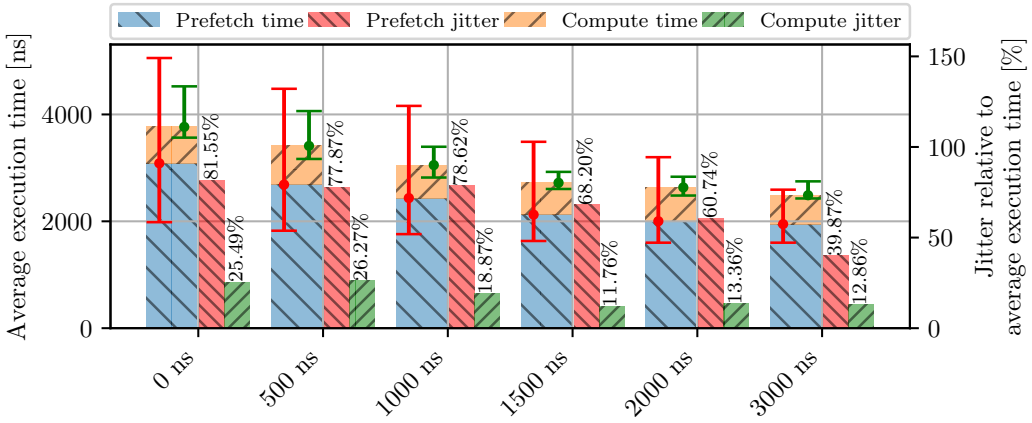


Figure 4.11: Only *prefetch* and *compute* phases are scheduled against each other (X-axis shows shift offset). *Writeback* phases are moved away by the schedule and do not influence the previous two phases. In this experiment the two blocks running in a kernel are scheduled at the same time instance.

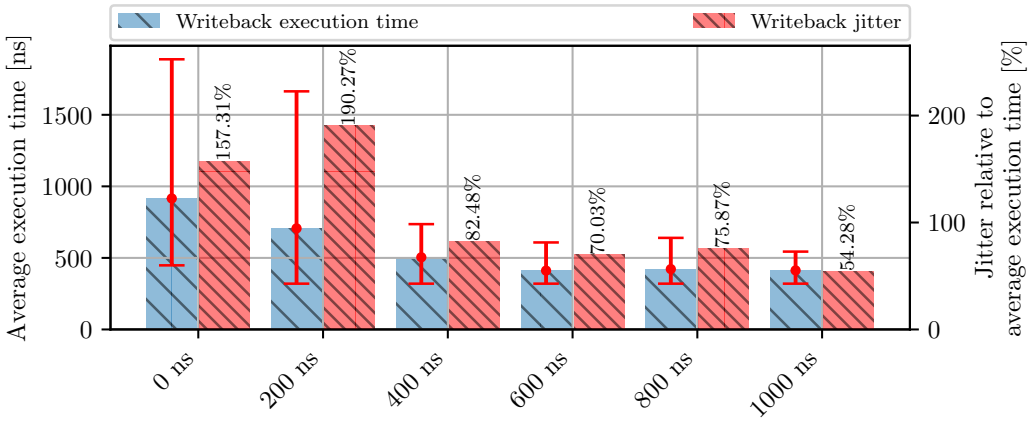


Figure 4.12: Execution time and jitter of *writeback* phases scheduled against each other (X-axis shows shift offset). *prefetch* and *compute* phases are scheduled away to isolate the *writeback* phases

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 three phases have much shorter and differently distributed phase execution times. 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. 4.8), 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)

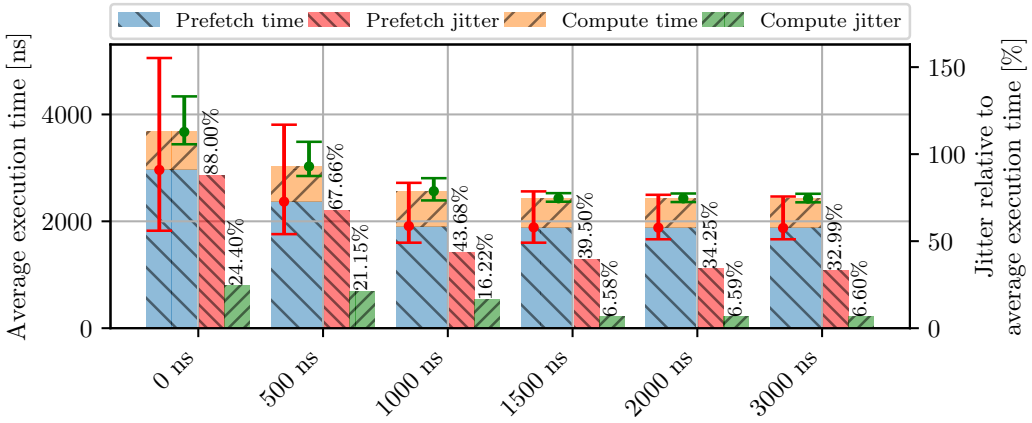


Figure 4.13: Only *prefetch* and *compute* phases are scheduled against each other (X-axis shows shift offset). *Writeback* phases are moved away by the schedule and do not influence the previous two phases. In this experiment the two blocks running in a kernel are scheduled at the same time instance.

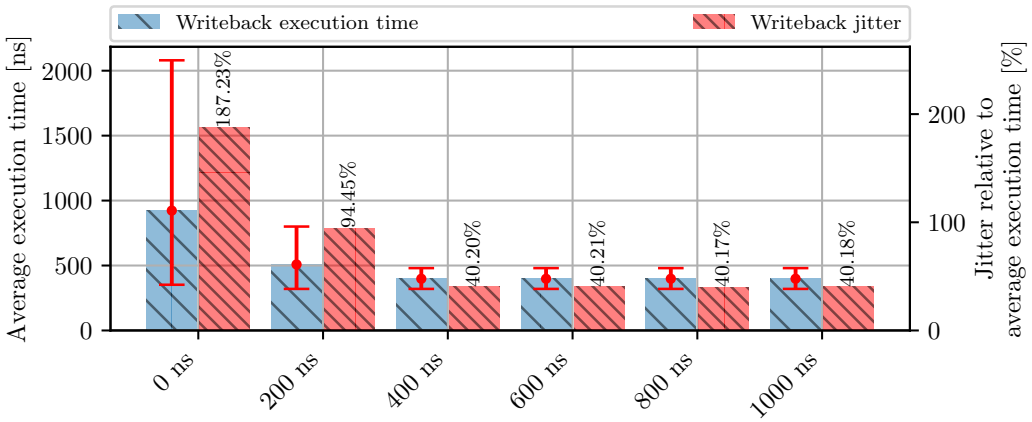


Figure 4.14: Execution time and jitter of *writeback* phases scheduled against each other (X-axis shows shift offset). *prefetch* and *compute* phases are scheduled away to isolate the *writeback* phases

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*.

4.3 KCF-Tracker kernel interference

In Figures 4.15a and 4.15b we can find an overview of interference between the kernels running in the KCF tracker. If we have a look at the first column of Figure 4.15a, we see the average execution times of the single kernels running in isolation. As we can see, not all kernels have the same duration to finish their jobs.

This means, if we test the interference between the kernels, some of them interfere only in the beginning of the execution of the kernel to be measured, namely the sum_c, conjugate, sqrt_magnitude, add_constant, multiply_constant and the matrix addition kernels introduce only a short interference period in the

first part of the execution of the matrix multiplication, matrix division, matrix element-wise multiplication, and sqrt_normalize. If those short kernels are more memory intensive, this leads to a more significant execution jitter as we can see in Figure 4.15b. Especially the conjugate, add_constant, sqrt_norm and multiply_constant kernels introduce a high jitter in the other kernels. Further, we can see that the element-wise multiplication and sqrt_norm kernels are the most sensitive kernels regarding the execution jitter.

If we look at the average execution time, we see that the matrix multiplication, matrix division, element-wise multiplication, and the sqrt_magnitude kernels introduce the biggest extension in the execution time to the other kernels.

Those two heat maps give us a good first overview which kernels should be avoided to run in parallel and where PREM might lead to a benefit.

4.4 Summary

Based on the performed experiments, we found that the TX2 offers an L2 cache of size 512kBytes and an L1 cache with the size of 12kBytes. By default, the L2 cache is used for global loads, and the L1 cache is used to buffer spilled registers of the kernels. However, NVCC offers an opt-in argument to enable even the L1 cache for global loads. Further, we found that the GPUs shared memory does not share the physical memory with the L1 cache.

The experiments performing the random and sequential walks with different thread and kernel configurations exhibited the interference between the threads and showed that it is beneficial to access element inside a kernel in a sequential manner, to reduce contention between threads. We also found that it is advantageous to launch the kernels with a multiple of 64 threads to reduce execution jitter between the threads. If multiple kernels are run in parallel, we see, that it is good to ensure, that all data fits into the L2 cache, to reduce the contention between the kernels running in parallel.

Further, we found a fast synchronization mechanism based on the globaltimer, after it was reconfigured by one run of nvprof. This fast synchronization allowed us to run the tiled 2D Convolution benchmark in time triggered manner, to reduce the interference between the running kernels. This lead to the reduction of the execution jitter from 6.47% for the four kernel legacy implementation to 0.15% for the tiled 2D Convolution with a tile offset of 1.3µs. To the end of the time-triggered experiments, we presented the interference between the single PREM phases and suggested that it is advantageous to loosen the PREM rules slightly on the GPU since phases cannot be stacked the same way as in CPU PREM. In the end, it is important to mention, that this evaluation was only performed on the 2D Convolution benchmark yet and might change if further benchmarks are taken into account. We performed the first step into this direction with the evaluation of the kernels present in the KCF tracker.

(a) : Heatmap KCF execution time

(b) : Heatmap KCF execution jitter

Figure 4.15: KCF heat maps

Chapter 5
Conclusion

In this thesis, we evaluated mechanisms for the low-overhead application of predictable execution model (PREM) to GPU kernels. First, we assessed the contention between CUDA-kernels running in parallel based on parallel and sequential walks. Then 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 of their blocks 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, it looks to be already sufficient to allow partly isolated execution on the tiles to provide a predictable execution jitter. Therefore, a possibility might be to loosen the PREM rules between GPU kernels slightly to provide only partly isolated memory access.

5.1 Future work

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 additionally on other kernels. Especially we would like to evaluate the behaviour of the PREM phases of more compute intensive kernels. Based on this, 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.

Appendix A

Abbreviated terms

COTS	Commercial Off-The-Shelve
CPU	Central Processing Unit
DP	Double-Precision
FFT	Fast Fourier Transform
FPGAs	Field-programmable gate arrays
FUs	Functional Units
GEMM	GEneralized Matrix Multiplication
GPU	Graphics processing units
HW	Hardware
KCF	Kernelized Correlation Filter
LD/ST	Load/Store
LKM	Loadable Kernel Module
MPSoC	Multi-Processor Systems-on-Chip
NVCC	NVIDIA CUDA Compiler
PREM	PRedictable Execution Model
PTX	Parallel Thread Execution
SASS	Streaming ASSembly
SFUs	Special Function Units
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessor
WCET	Worst case execution time
WOET	Worst observed execution time

Appendix B

Bibliography

[1] 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.

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

[3] 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*, pp. 269–279, April 2011.

[4] V. Karafiàt and M. Sojka, "Aplication of prem model on parallel implementation of KCF tracker algorithm," May 2018.

[5] J. Bin, S. Girbal, D. Gracia Perez, A. Grasset, and A. Merigot, "Studying co-running avionic real-time applications on multi-core cots architectures," 02 2014.

[6] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," pp. 741–746, 03 2010.

[7] D. Luebke and G. Humphreys, "How gpus work," pp. 96–100, 02 2007.

[8] M. Galloy, "Cpu vs gpu performance." `https://michaelgalloy.com/2013/06/11/cpu-vs-gpu-performance.html`, 2013. [Online; accessed 20-Jan-2019].

[9] NVIDIA, "Whitepaper, NVIDIA Tesla P100." `https://www.nvidia.com/object/pascal-architecture-whitepaper.html`, 2016.

[10] 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)*, pp. 104–115, Dec 2017.

[11] 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)*, pp. 119–130, Dec 2018.

[12] M. Harris, "NVIDIA, An Easy Introduction to CUDA C and C++." https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/, 2012. [Online; accessed 15-May-2019].

[13] D. Kanter, "Nvidia's gt200: Inside a parallel processor." https://www.realworldtech.com/gt200/2/, 2008. [Online; accessed 20-Jan-2019].

[14] 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)*, pp. 1–10, Sep. 2017.

[15] X. Mei and X. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, pp. 72–86, Jan 2017.

[16] 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)*, pp. 235–246, March 2010.

[17] 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," pp. 1–13, 06 2016.

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

[19] 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, pp. 1522–1532, June 2014.

[20] 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*, pp. 318–321, March 2017.

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

[22] "Git repository, hercules-public." https://iis-git.ee.ethz.ch/H2020-Compiler/HerculesCompiler-public, 2018. [Online; accessed 31-Oct-2018].

[23] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," pp. 55–64, 04 2013.

[24] 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*, pp. 1–11, Nov 2011.

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

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