AMALTHEA-based GPU Response Time Analysis for NVidia's Jetson TX2

Daniel Leoncio Paredes Zevallos

A thesis presented for the degree of
Master of Engineering

Supervised by:
Professor Dr. Carsten Wolff
Herr Robert Hoettger

Fachhochschule Dortmund, Germany
October 2019

Abstract

The purpose of the thesis is to develop a response time algorithm for AMALTHEA-based models to analyze timing behaviors of CUDA kernels on NVIDIA Jetson TX2's GPU. This work was developed on the context of the Bosch WATERS Challenge 2019 [6]. The challenge focuses on timing-analysis for heterogeneous software and hardware systems based on centralized end-to-end architectures. The embedded platform NVIDIA Jetson TX2 was the selected platform for testing. Furthermore, key concepts related to NVIDIA's GPU architecture are presented, as well as a detailed explanation of rules behind platform's GPU scheduler. This work is based on these rules and experimental results show the accuracy of our approach to estimate completion times for kernels executed on Jetson TX2 platform. Experiments use real timing data from NVIDIA's platform. Moreover, the algorithm is implemented in Eclipse APP4MC, which allows an AMALTHEA-based response time analysis for NVIDIA's Jetson TX2.

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. I am aware that the thesis in digital form can be examined for the use of unauthorized aid and in order to determine whether the thesis as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources I agree that it shall be entered in a data base where it shall also remain after examination, to enable comparison with future theses submitted. Further rights of reproduction and usage, however,are not granted here. This thesis was not previously presented to another examination board and has not been published.

Place and Date Signature

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt und indirekt übernommenen Gedanken sind als solche kenntlich gemacht.Ich weiß, dass die Arbeit in digitalisierter Form daraufhin überprüft werden kann, ob unerlaubte Hilfsmittel verwendet wurden und ob es sich - insgesamt oder in Teilen - um ein Plagiat handelt. Zum Vergleich einer Arbeit mit existierenden Quellen darf sie in eine Datenbanke ingestellt werden und nach der Überprüfung zum Vergleich mit künftige ingehenden Arbeiten dort verbleiben. Weitere Vervielfältigungs- und Verwertungsrechte werden dadurch nicht eingeräumt. Die Arbeit wurde weder einer anderen Prüfungsbehörde vorgelegt noch veröffentlicht.

Ort, Datum Unterschrifft

Table of Contents

List of Figures

List of Tables

Abbreviations and Symbols

Abbreviations

API	**A**pplication **P**rogramming **I**nterface
CE	**C**opy **E**ngine
CPU	**C**entral **P**rocessing **U**nit
EE	**E**xecution **E**ngine
FIFO	**F**irst **I**n **F**irst **O**ut
GPU	**G**raphics **P**rocessing **U**nit
HMP	**H**eterogeneous **M**ulti-**P**rocessor
SM	**S**treaming **M**ultiprocessor
OS	**O**perating **S**ystem

Symbols

τ	Set of tasks or kernels
τ_i	Task or kernel i
T_i	Period of τ_i
g_i	Grid size of τ_i
b_i	Number of thread per block within g_i
C_i	Execution time workload of a thread within b_i
u_i	Utilization of τ_i
r_i	Release time of τ_i

f_i	Completion time of τ_i
R_i	Response time of τ_i
U_t	Total utilization of τ
b_{max}	Maximum amount of threads in a SM
g_{max}	Maximum amount of blocks that can be allocated in a SM
g_f	Available blocks at some point in time
t_a	Point in time at which a block will be allocated

Chapter 1

Introduction

1.1 Motivation

Car manufactures want to reduce cost in terms of money and time required to develop, test and validate a new piece of software and hardware due to a change of supplier. For that reason, centralized end-to-end architectures are the solution they are aiming to, because, for car companies such as BWW and Audi, the car of future will be similar to a "data center on wheels" [1].

Centralized end-to-end architectures would be the first step stone towards decoupling software and hardware [2]. This type of architectures will not only take advantage of internet connectivity, cloud computing and powerful heterogeneous processing units, but also allow scalable, hierarchical and highly integrated system. In other words, car manufactures prefer low-latency, hierarchical and cost effectiveness of centralized end-to-end architectures, because of today's requirements of computational power, bandwidth, integration, safety and real-time [3].

However, car manufactures do not forget that in centralized end-to-end architectures different types of software would run on top of an heterogeneous hardware supplied by companies such as NVIDIA, Mobileye or Qualcomm.

Thus, it is important to analyze and understand how software behave under those conditions, in order to ensure a predictable and efficient system.

1.2 Industrial challenge WATERS 2019

Predictability is a key property for safety-critical and hard real-time systems [4]. Analyzing time related characteristics is an important step to design predictable embedded systems. However, in multi-core or heterogeneous systems based on centralized end-to-end architectures is harder to satisfy timing constrains due to scheduling, caches, pipelines and out-of-order executions [5]. Thus, development of timing-analysis methods for these types of architectures has become, nowadays, one of the main focus of research in both industry and academic environment.

Every year *the WATERS Challenge* is announced. The purpose of the WATERS industrial challenge is to share ideas, experiences and solutions to concrete timing verification problems issued from real industrial case studies [6].

This year, 2019, the challenge focuses on timing-analysis for heterogeneous software-hardware systems based on centralized end-to-end architectures. The platform chosen for this purpose is the NVIDIA® Jetson™ TX2 platform, which has a heterogeneous architecture equipped with a Quad ARM A57 processor, a dual Denver processor, 8GB of LPDDR4 memory and 256 CUDA cores of NVIDIA's Pascal Architecture. An AMALTHEA model based on this platform is available [6]. Developers can design solutions and test them later on real hardware.

1.3 NVIDIA Jetson TX2: Architecture Overview

NVIDIA Jetson TX2 is an embedded system-on-module (SOM). It is ideal for deploying advanced AI to remote field locations with poor or expensive internet connectivity, robotics, gaming devices, Virtual Reality (VR), Augmented Reality (AR) and portable medical devices. In addition, it offers near-real-time

responsiveness and minimal latency which is key for intelligent machines that need mission-critical autonomy like mining[7].

The main components of the Jetson TX2 are a dual-core ARMv8 based NVIDIA Denver2, quad-core ARMv8 Cortex-A57, 8GB 128-bit LPDDR4 and integrated 256-core Pascal NVIDIA GPU. The quad-core Cortex-A57 and the dual-core NVIDIA Denver2 can be seen as a cluster of heterogeneous multiprocessors (HMP) [8]. Both HMP and GPU share a 8GB SRAM memory as shown in Figure 1.1. Hereafter, whenever the term **host** will refer to HMP, similarly **device** will refer to GPU.

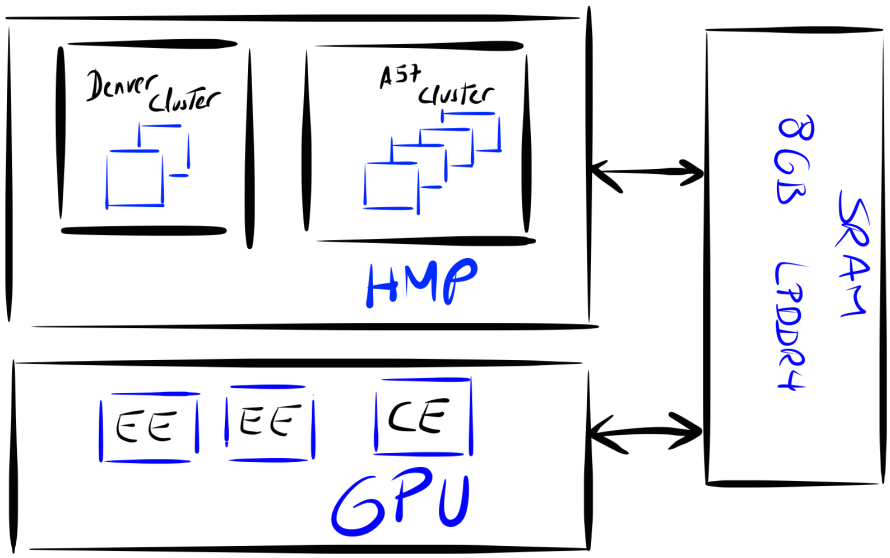

Figure 1.1: Jetson TX2 Architecture Overview

Any NVIDIA GPU has two types of engines, **Copy Engines** (CE) and **Execution Engines** (EE). The Jetson TX2 has only one CE and two EE also known as **Streaming multiprocessors**. The CE is in charge of data transfers from host to device and viceversa. There is, moreover, the possibility that EE and CE run concurrently.

The GPU uses **streams** to run applications. The number of streams depends on the GPU resources. An application can run in one or multiple streams, the GPU

scheduler, by default, manages how the application are allocated on streams in order to maximize throughput. In Chapter 2, it is discussed how the TX2 GPU scheduler behaves in case of multiple applications in more detail.

1.4 Jetson TX2 Amalthea Model

APP4MC is a platform for engineering multi- and many-core embedded systems. This platform enables the creation and management of complex tool chains including simulation and validation based on AMALTHEA models[9]. In the context of the WATERS Challenge 2019, Bosch offers an AMALTHEA model of the Jetson TX2. In this model, a CPU runnable reads data from memory, executes some computation (Ticks) and writes back data into memory as shown in Figure 1.2.

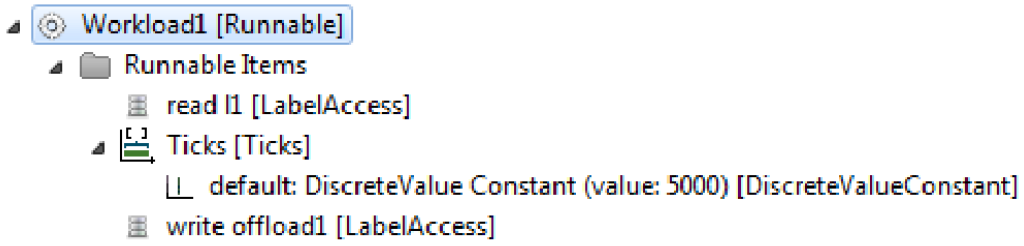

Figure 1.2: Runnable example for a CPU [6]

In the case of GPU modeling, the runnable follows the same pattern as in the CPU case: read, execution, write back. However, the reading operation is actually to copy memory from host to device, thus it is modeled as *memory reading from host* and then as *memory writing to device*. On the other hand, the writing back operation requires to copy memory from device to host therefore it is modeled as *memory reading from device* and then as *memory writing to host* as shown in Figure 1.3.

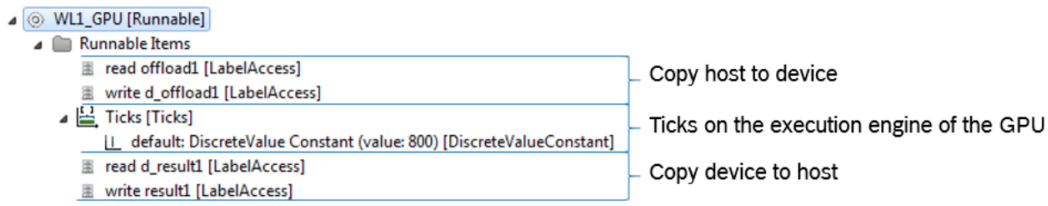

Figure 1.3: Runnable example for a GPU [6]

Chapter 2

CUDA and Jetson TX2

In this chapter an overview of the theorical background of the NVIDIA GPU software and hardware model is given. There is also an introduction to the concepts of threads, blocks, kernels and streaming multriprocesor, and how they apply to this work's study case. Jetson TX2's memory hierarchy and scheduler are presented in addition to the rules behind the Jetson TX2's hardware scheduler. At the end there is a comprehensive example.

2.1 NVIDIA GPU Software Model

Nowadays applications run on heterogeneous hardware and GPUs are important in order to achieve high performance computing. Since 2006, a running software on NVIDIA GPUs are known as a *CUDA application* [10]. A CUDA application runs concurrently multiple instances of special functions called **kernels**. Each instance runs on a **thread**. Moreover, these threads are arranged in **blocks** and blocks compose **grids** as shown in Figure 2.1.

There is also a hierarchical memory structure. Threads, blocks and grids have access to different memory spaces as ilustrated in Figure 2.2. The types of memory are summarized in Table 2.1.

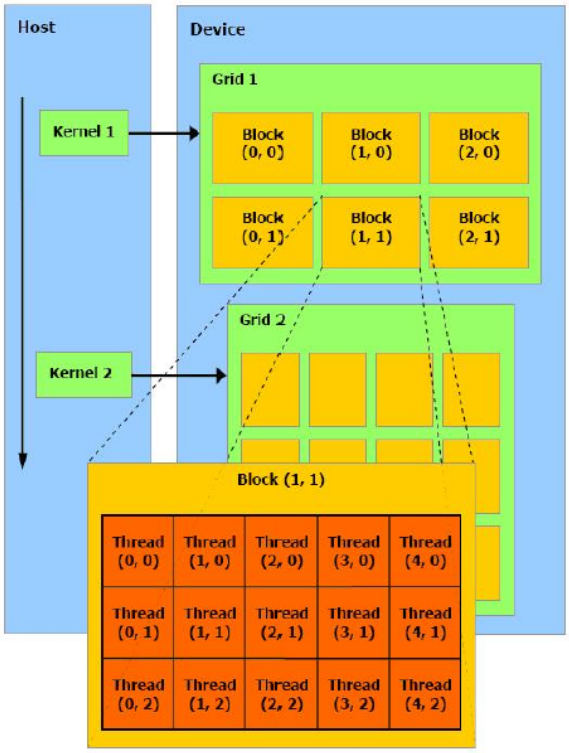

Figure 2.1: Organisation of grids, blocks, threads, and kernels [11].

Table 2.1: Types of memories in a GPU

Memory	Main Characteristics	Scope	Lifetime
Global	R/W, Slow and big	Grid	Application
Texture	ROM, Fast, Optimized for 2D/3D access	Grid	Application
Constant	ROM, Fast, Constants and kernel parameters	Grid	Application
Shared	R/W, Fast, it's on-chip	Block	Block
Local	R/W, Slow as global, when registers are full	Thread	Thread
Registers	R/W, Fast	Thread	Thread

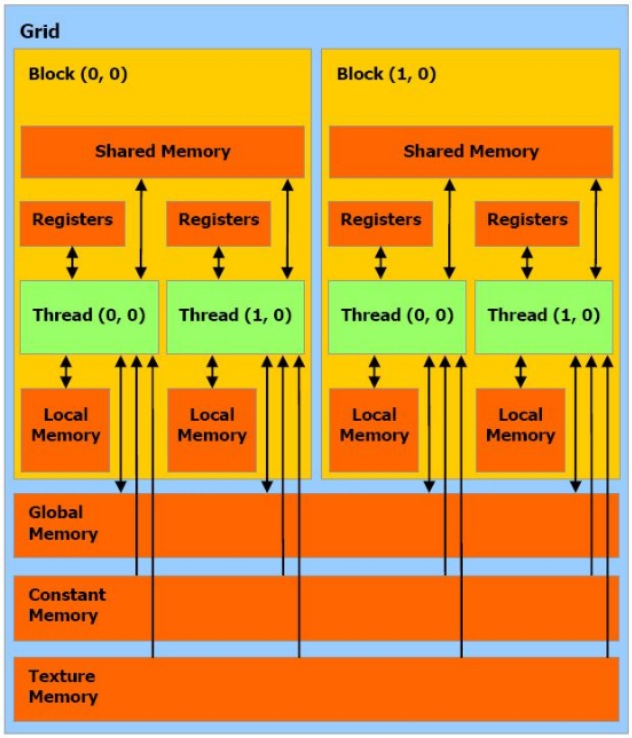

Figure 2.2: Memory hierarchy [11].

In summary, CUDA applications solve problems that were modeled based on *divide and conquer* principle. Each thread executes a kernel on a small subset

of data. Thus, CUDA software model not only allows users to achieve high computational performance, but also high scalable CUDA applications.

2.2 NVIDIA GPU Hardware Model

The CUDA architecture is based on **Streaming Multiprocessors** (SM) which perform the actual computation. Each SM has its own control units, registers, execution pipelines and local memories, but they also have access to global memory as ilustrated in Figure 2.3. A **stream** is a queue of CUDA operations, memory copies and kernel launches. Streams are presented in more detail in following sections.

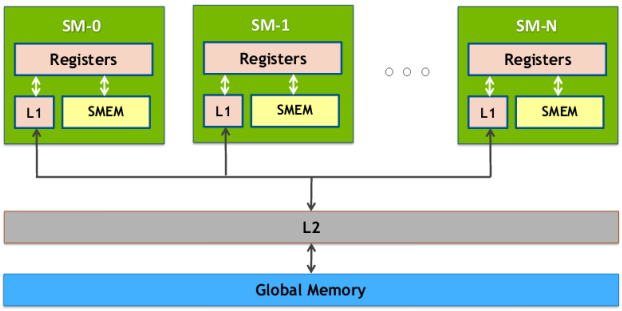

Figure 2.3: Memory hierarchy

When a kernel grid is launched, blocks are enumerated and assigend to the SMs. Once the blocks are assigned, threads are managed in **wraps** by the **wrap scheduler**. A wraps is a group of 32 threads that run in parallel. Thus, it is highly recommendable to use block sizes of size $32N, N \in \mathbb{N}$, otherwise there would be *inactive* threads. An example is shown in Figure 2.4 where there is a block of 140 threads but since the wrap scheduler works with wraps, 20 threads are wasted because no other block can make use of them.

The amount of threads and blocks that can run concurrently per SM depends on the number of 32-bit registers and shared memory within SM as well as the CUDA computing capability of the GPU. Information related to the maximum amount of blocks or threads as well as the computing capability of the GPU

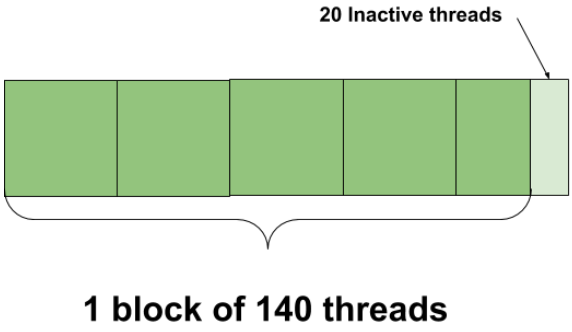

Figure 2.4: Inactive threads

can be displayed executing a device query tool, which is installed by default on the Jetson TX2. Some information about Jetson TX2 is presented below:

```
CUDA Device Query (Runtime API) version (CUDART static linking)
Detected 1 CUDA Capable device(s)
Device 0: "NVIDIA Tegra X2"
  CUDA Driver Version / Runtime Version     9.0 / 9.0
  Total amount of global memory:            7850 MBytes
  ( 2) SM, (128) CUDA Cores/SM:             256 CUDA Cores
  L2 Cache Size:                            524288 bytes
  Total amount of shared memory per block:  49152 bytes
  Total number of registers per block:      32768
  Max. number of threads per SM:            2048
  Max. number of threads per block:         1024
  Max dim. size of a thread block (x,y,z):  (1024, 1024, 64)
  Max dim. size of a grid size    (x,y,z):  (2^31-1, 65535, 65535)
```

2.3 NVIDIA Jetson TX2's GPU Scheduler

It is common to use several kernels in an application. In order to reduce computation time and maximaze GPU utilization, it is desired to run multiple kernels in parallel. CUDA uses streams to achieve this goal. As mentioned

before, a stream is a queue of CUDA operations, memory copies and kernel launches. Thus, it is possible either to launch multiple kernels within one streams or multiple kernels on multiple streams. Operations within the same stream are managed in FIFO (First In First Out) fashion, thus, the term **stream queue** in this work is used to refer FIFO queues within a stream. The Jeston TX2's GPU assigns resources to streams using its internal scheduler.

Predictability is an important characteristic of safety-critical systems. It requires both functional and timing correctness. However, a detailed information about the Jetson TX2's GPU scheduler behaviour is not publicly available. Without such details, it is imposible to analyze timing constrains. Nevertheless, there are some efforts [12], [13] and [14] aimed at revealing these details through black-box experimentation.

NVIDIA GPU scheduling policies depend on whether the GPU workloads are launched by a CPU executing OS threads or OS processes. This work focuses on the first case, because GPU computations launched by OS processes have more unpredictable behaviours, as stated in [12] and [13]. In this section, GPU scheduling policies devired by [12] are presented and an example clarifies their use.

Some terms should be defined first. When one block of a kernel has been scheduled for execution on a SM it is said that the block was **assigned**. Moreover, it's said a kernel was **dispatched** as soon as one of its blocks were assigned, and **fully dispatched** once all its blocks were assigned. The same applies to copy operations and CE.
There are, in addition, FIFO CE queues used to schedule copy operations and FIFO EE queues used to schedule kernel launches. Stream queues feed CE and EE queues. Bellow the rules that determine scheduler and queues' behaviours are presented.

- **General Scheduling Rules**:
  - **G1** A copy operation or kernel is enqueued on the stream queue for its stream when the associated CUDA API function (memory

transfer or kernel launch) is invoked.
  - **G2** A kernel is enqueued on the EE queue when it reaches the head of its stream queue.
  - **G3** A kernel at the head of the EE queue is dequeued from that queue once it becomes fully dispatched.
  - **G4** A kernel is dequeued from its stream queue once all of its blocks complete execution.
- **Non-preemptive execution**:
  - **X1** Only blocks of the kernel at the head of the EE queue are eligible to be assigned.
- **Rules governing thread resources**:
  - **R1** A block of the kernel at the head of the EE queue is eligible to be assigned only if its resource constraints are met.
  - **R2** A block of the kernel at the head of the EE queue is eligible to be assigned only if there are sufficient thread resources available on some SM.
- **Rules governing shared-memory resources**:
  - **R3** A block of the kernel at the head of the EE queue is eligible to be assigned only if there are sufficient shared-memory resources available on some SM.
- **Copy operations**:
  - **C1** A copy operation is enqueued on the CE queue when it reaches the head of its stream queue.
  - **C2** A copy operation at the head of the CE queue is eligible to be assigned to the CE.
  - **C3** A copy operation at the head of the CE queue is dequeued from the CE queue once the copy is assigned to the CE on the GPU.
  - **C4** A copy operation is dequeued from its stream queue once the CE has completed the copy.
- **Streams with priorities**:
  - **A1** A kernel can only be enqueued on the EE queue matching the priority of its stream.

– **A2** A block of a kernel at the head of any EE queue is eligible to be assigned only if all higher-priority EE queues (priority-high over priority-low) are empty.

Authors in [12] mentioned that rules related to **registry resources** are expected to have exactly the same impact as threads and shared-memory rules.

An example of block allocation for different kernels on the Jetson's GPU is shown in Figure 2.5. Each rectangle represents a block: the *j-th* block of kernel k is labeled K*k:j*. End and start time of each block are represented by its right and left boundaries. The height of each rectangle is the number of threads used by that block. Dashed lines correspond to time points that are interesting for analysis. In Figure 2.6 is presented the state of queries for each time represented by dashed lines and in Tables 2.2, 2.3 and 2.4 a description of how each scheduling rule influence queries' behaviour.

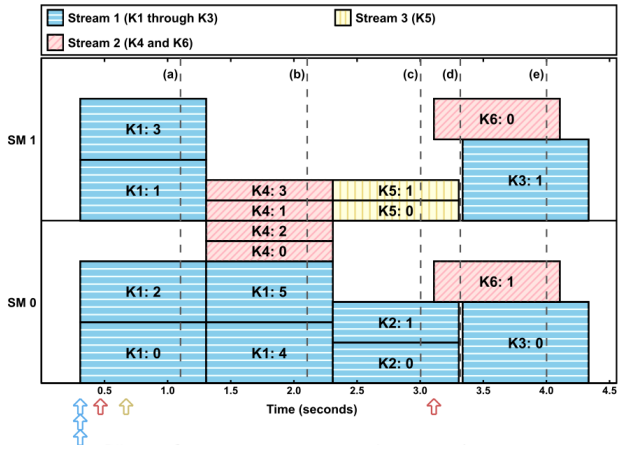

Figure 2.5: Basic GPU scheduling experiment [12]

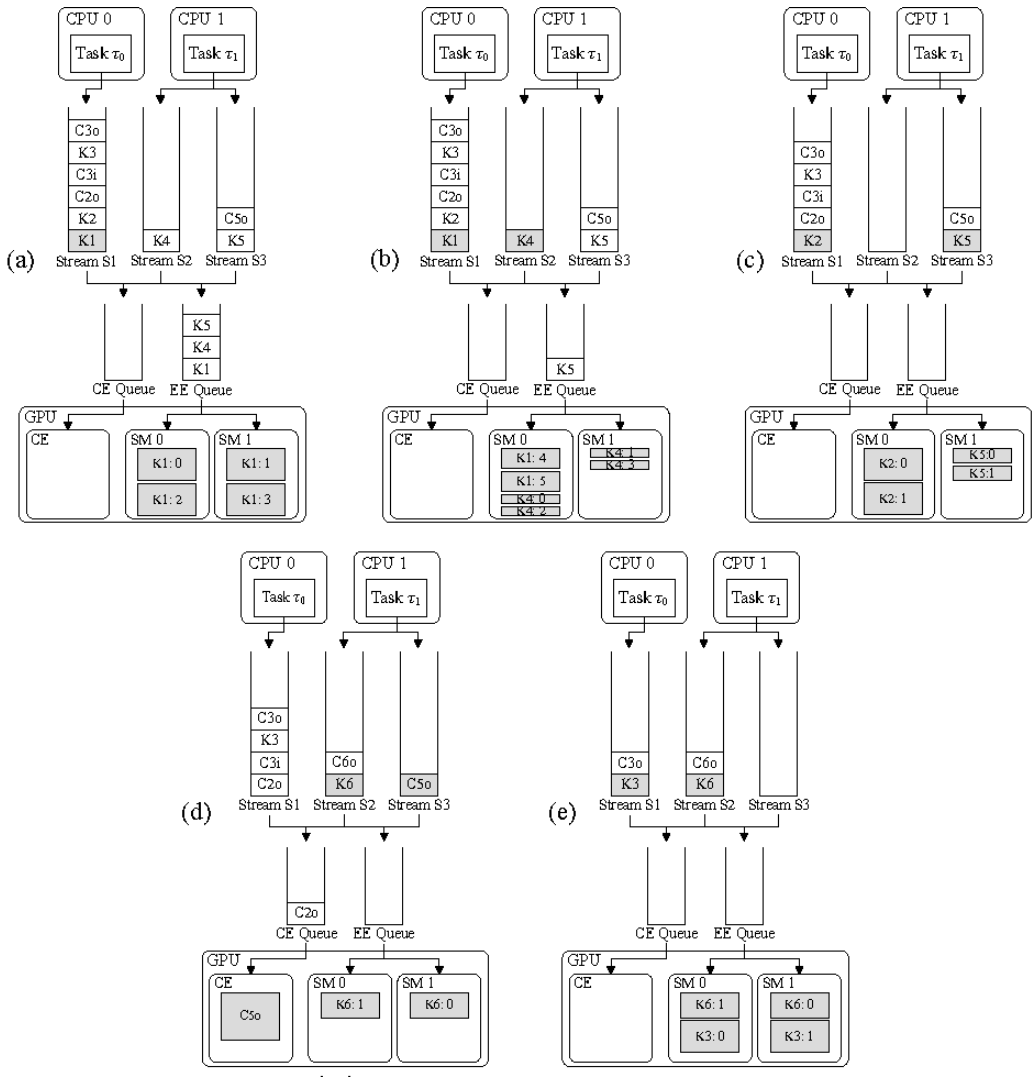

Figure 2.6: Detailed state information at various time points in Fig. 2.5 [12] .

Rules	(a) t=1.0s	(b) t=2.1s	(c) t=3.0s	(d) t=3.4s
G1	All Kernels except for K6 were enqueued on their streams. K6 is launched at t = 3.2s	K6 operations are not yet enqueued on S2. Same reason as in (a).	Same situation as in (b).	K6 operations were enqueued at t=3.2s on S2.
G2	K1, K4, K5 were at the head of their streams. They were enqueued on EE queue.	There are not new kernel at the head of stream queues.	K2 was enqueued on EE queue.	K6 kernel was enqueued on EE, because it was at the head of S2.
G3	No kernels fullfill this rule.	K1, K4 have dispatched all their blocks. K5 is the only one on the EE queue.	K5, K2 were dequeued from EE queue, because all their blocks were dispatched.	K6 was fully dispatched, thus was dequeued from EE queue.
G4	No kernels fullfill this rule. K1 still has running blocks.	K1, K4 still have running blocks. Thus they cannot be dequeued from their stream queues.	K1, K4 were dequeued from their stream queues, because all their blocks finished execution. K2, K5 still have running blocks, they cannot be dequeued from stream queues.	K6 still have running blocks. Thus cannot be yet dequeued from S2.

Table 2.2: Detailed state information at various time points in Fig. 2.6

Rules	(a) t=1.0s	(b) t=2.1s	(c) t=3.0s	(d) t=3.4s
X1	K4 cannot be launched because of this rule, even when there are enough resources (512 threads)	K4 was the next kernel on the EE queue. It was launch because K1 already dispatched it's remaining blocks.	K5 blocks became eligible then dispatched. After that K2 blocks became eligible and then dispatched.	K6 blocks became eligible, because K6 was at the head of EE queue.
R1	Applies only to K1.	K5 is eligible, but check R3	K5 became eligible. K2 became eligible after K5.	There were enough resources for K6.
R2	Applies only to K1.	K5 is eligible, but check R3	There were enough thread resources for K2 and K5 (1024 threads in SM0, and 1536 threads in SM1).	There were enough thread resources in each SM for K6 (free 512 threads per SM , each K6 block needed 512 threads).
R3	Applies only to K1.	There is not enough shared memory to launch K5. Each K5 block requires 32KB (64KB in total), but K4 blocks are consuming the whole shared memory available per SM (64KB).	There were enough shared memory for K2 and K5 (64KB in each SM).	K6 blocks required no memory shared.

Table 2.3: Detailed state information at various time points in Fig. 2.6

Rules	(a) t=1.0s	(b) t=2.1s	(c) t=3.0s	(d) t=3.4s
C1	No copy operations at the head of streams.	No copy operations at the head of streams.	No copy operations at the head of streams.	C5o, C2o were enqueued on CE queue.
C2	No available copy operations.	No available copy operations.	No available copy operations.	C5o was assigned to CE.
C3	No available copy operations.	No available copy operations.	No available copy operations.	C5o was dequeued from CE.
C4	No copy operations at the head of streams.	No copy operations at the head of streams.	No copy operations at the head of streams.	C5o is still copying. Thus it cannot be dequeued from S3.

Table 2.4: Detailed state information at various time points in Fig. 2.6

Chapter 3

Jetson TX2's GPU scheduler response time analysis

In this chapter, the main contribution of this work is presented, the response time analysis for Jetson TX2's GPU scheduler based on the set of scheduling rules explained in the last chapter. A task model is defined, works assumptions are declared, and a brief introduction to GPU response time analysis is given. In addition, the last sections of this chapter present examples, description of a special case of this work, computational complexity discussion.

3.1 Task model

There is a set of tasks or kernels τ of n independent kernels $\{\tau_1, \tau_2, ..., \tau_n\}$ on a single GPU. Each kernel has a period T_i defined as the separation between two consecutives releases of τ_i, thread execution time workload C_i and a grid of blocks g_i. Each block contains b_i threads.

$$\tau = \{\tau_i\}; \quad i \geq n \wedge n \in \mathbb{N} \tag{3.1}$$

$$\tau_i = \{T_i, C_i, g_i, b_i\} \tag{3.2}$$

Thus each kernel τ_i has a total of $g_i \cdot b_i$ threads, and the total execution time workload of τ_i is $C_i \cdot g_i \cdot b_i$. The utilization of each kernel is defined as the total execution time workload divided by the period, as stated in [15].

$$u_i = \frac{C_i g_i b_i}{T_i} \tag{3.3}$$

In addition, the total utilization of the set of tasks τ is defined as:

$$U_t = \sum_{\tau_i \in \tau} u_i \tag{3.4}$$

For a kernel τ_i, r_i denotes its release time, f_i its completion time as f_i and $R_i = f_i - r_i$ its response time. This work assumes that a kernel τ_i has a deadline equal to its period T_i.

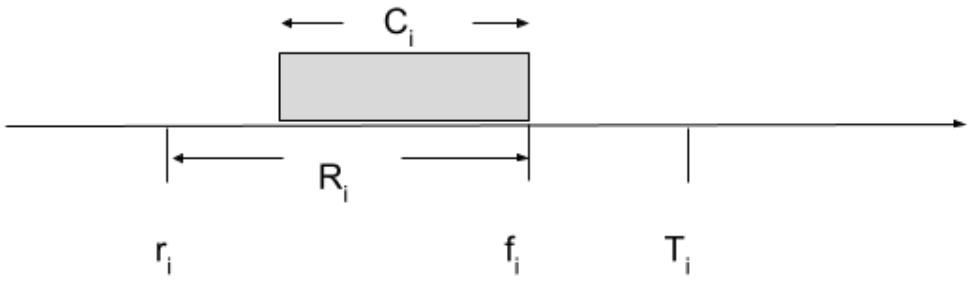

Figure 3.1: Time chart

3.2 Assumptions

For the calculation of response times there are two assumption:

3.2.1 All blocks have the same amount of threads

The election of the optimal number of threads for a specific kernel is a hard task. For that reason there have been some efforts towards that direction [16], [17], [18], [19]. However, NVIDIA developers recommend, for practical purposes, on their offical guides [20] and [21] to use block sizes equals to either 128, 256, 512 or 1024, because it has been documented that these values are more likely to take full advantage of the GPU resources. This work assumes that all the blocks, regardless the kernel, are the same size because for this work is the first step to more complex analysis.

$$b_i = b, \quad \forall \tau_i \in \tau \tag{3.5}$$

3.2.2 One big streaming multiprocessor

This assumption is derived from the previous one. Each streaming multiprocessor in the Jetson TX2 has 2048 available threads and since b_i can be either 128, 256, 512 or 1024 ($2048/b_i = k, k \in \mathbb{N}$), we can think of the two streaming multiprocessors as a big one of 4096 threads. It means that it could be allocated $2048/b_i$ blocks per SM or $4096/b_i$ blocks in the big SM. Hereafter, this work asummes that Jetson TX2's GPU has only one SM. Thus, g_{max} defines the maximum number of blocks that can be allocated in the SM at some point in time.

$$g_{max} = \frac{b_{max}}{b}, \quad g_{max} \in \mathbb{N} \tag{3.6}$$

Where b_{max} is the maximum amount of threads in the GPU in the case of Jetson TX2 is 4096.

3.3 Introduction to GPU Response Time Analysis

In addition to the variables defined in the assumptions section, g_f defines the number of blocks that are available at some point in time t, and t_a the point in time in which a block $b_i \in g_i$ can be allocated.

For example in Figure 3.2a is shown that for a $t = t_1$ the amount of free blocks g_f is lower than g_{max} while in Figure 3.2b for a $t = t_2$, $g_f = g_{max}$.

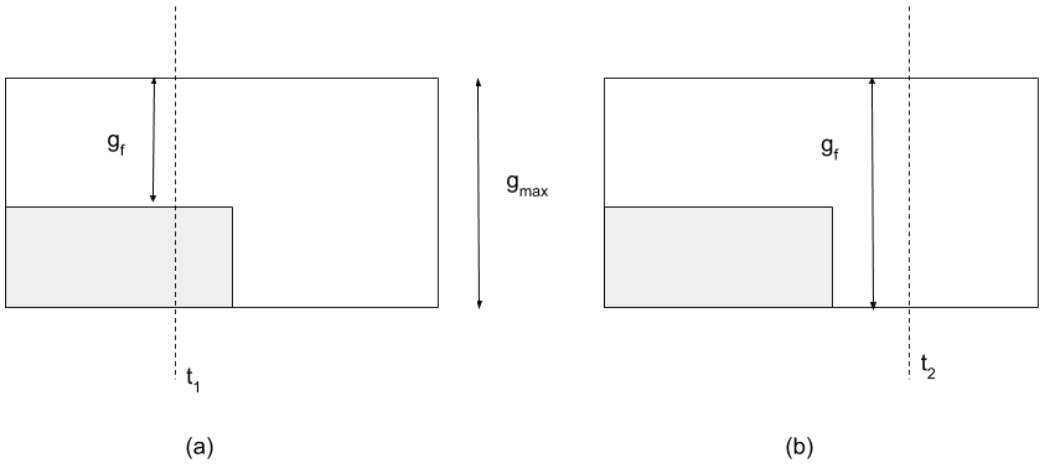

Figure 3.2: Free blocks (a) at $t = t_1$, $g_f < g_{max}$ (b) at $t = t_2$, $g_f = g_{max}$

In Figure 3.3 we present two cases. Let us assume there is a new kernel K4 which wants to allocate a block $b_i \in g_i$. In Figure 3.3a the release time r_4 of the kernel 4 is lower than t_1, which means that $t_a = t_1$ because $r_4 \leq t_1$ and kernel 3 (K3) was already dequeued. In Figure 3.3b r_4 lies between t_2 and t_3, in that case $t_a = r_4$, because all previous kernels were already dequeued and there are enough resources.

Assuming t_a is known, we would need to calculate how many blocks can be allocated at that point of time. In other words, we need to know the value of g_f at t_a. In Figure 3.4a a new kernel K3 with 6 blocks $g_3 = 6$ is going to be allocated on the Jetson's GPU. Each block have 512 threads, which means that $g_{max} = 8$. The GPU is not executing any kernel at $t = t_a$ as shown in Figure 3.4b therefore $g_f = g_{max} = 8$ at $t = t_a$. Given that $g_3 < g_f$ all the blocks of K3

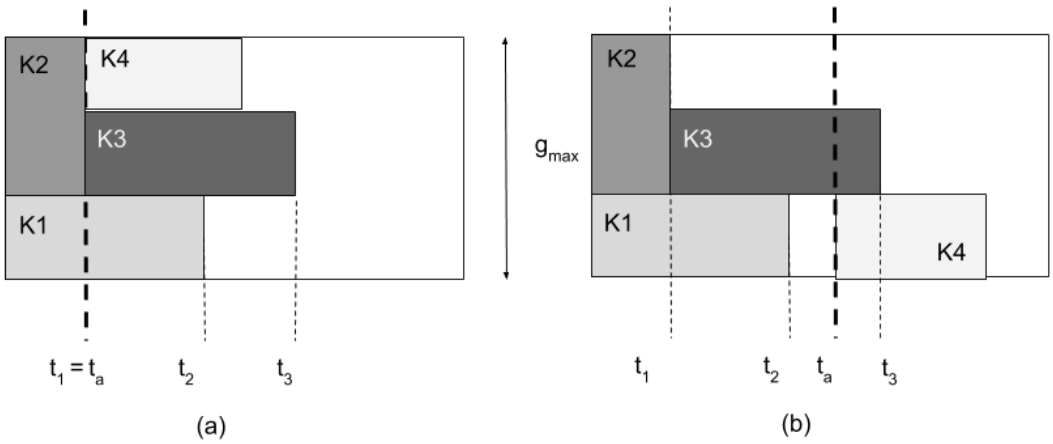

Figure 3.3: (a) $t_a = t_1 \quad \forall r_4$ s.t $r_4 \leq t_1$ (b) $t_a = r_4 \quad \forall r_4$ s.t $t_2 \leq r_4 \leq t_3$

will be allocated at the same time as shown in Figure 3.4c. The completion time f_3 of kernel K3 is t_a plus the thread execution time given by C_3, $f_3 = C_3 + t_a$. If we assume that the release time r_3 is the same as t_a then the completion time for K3 is the same as the response time R_3, otherwise $R_3 \geq f_3$.

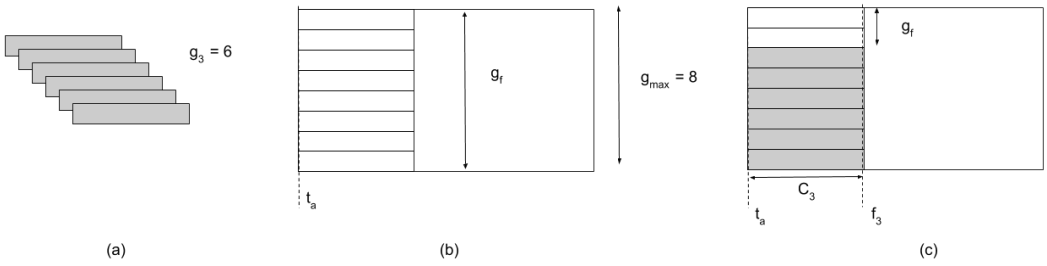

Figure 3.4: (a)New kernel $K3$ with 6 blocks to allocate $g_3 = 6$. (b) State prior to $K3$ of the GPU (c) state after K3 allocation

Once f_3 and R_3 are calculated, it's important to update the values of t_a and g_f, because these values will be used by the following kernel. Let us start with g_f, it is easy to notice that after K3 allocation there are two free blocks $g_f - g_3 = 2$ as a result the new value of $g_f = 2$. On the other hand, by definition t_a is the point in time in which a block $b_i \in g_i$ can be allocated, therefore t_a will not change because $g_f > 0$.

In Figure 3.5 we analyze another highly probable scenario. We use the same kernel K3 as in the last example ($g_3 = 6$). However, as shown in Figure 3.5b, there were two kernels allocated previously to K3. Kernel K1 with 5 allocated blocks $g_1 = 5$ and K2 with 3 allocated blocks $g_2 = 3$. Note that these kernels have different completion time $f_2 > f_1$. Nevertheless, what matters is not either K1 or K2 completion time but the value of t_a and g_f. In this example, t_a is the same as K1 completion time and g_f has the same value as g_1, $g_f = 5$. Thus, 5 blocks from K3 will be allocated first as shown in Figure 3.5c.

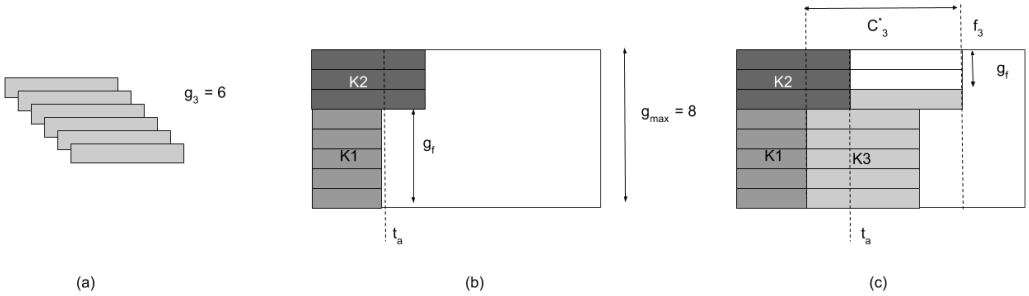

Figure 3.5: (a)New kernel $K3$ with 6 blocks to allocate $g_3 = 6$. (b) State prior to $K3$ of the GPU. Kernels K1 and K2 were previously allocated (c) state after K3 allocation

The next logical question is where the last block of K3 should be allocated. The answer again is given by the updated values of t_a and g_f. Looking at Figure 3.5c is easy to get these new values. The new value of t_a is f_2 since $f_2 < (t_a + C_3)$, and for $t = t_a$ the corresponding g_f is g_2. Thus, the last K3 block is allocated at $t = t_a = f_2$ and that give us the completion time for K3 that is $f_3 = f_2 + C_3$ or $f_3 = t_a + C_3$. In Figure 3.5c, we defined a new variable C_3^* as the total amount of time in which K3 was using GPU resources.

After K3 allocation, t_a and g_f should be updated again. In this example, the new g_f is the old value of g_f minus the last allocated K3 blocks $g_f = g_f - 1 = 2$, while t_a remains the same $t_a = f_2$ because the conditions are the same as in the later example where there was only one kernel.

3.4 Response Time Analysis Algorithm

Our algorithm is focused on the calculation of t_a and g_f for each block regardless of from which kernel τ_i comes. In addition, it is important to notice that t_a and g_f depend on how previous blocks were allocated and on the GPU state at some point in time, as it was described above and illustrated in the Figure 3.4 and Figure 3.5.
The output of our algorithm is a set of release times $f_1, f_2, \cdots, f_n$ where n is the length of τ which values f_i depend on t_a and C_i.

$$f_i = f(t_a, C_i) \tag{3.7}$$

A basic version of our algorithm is described in Algorithm 1. This version is derived directly from the examples illustrated in Figure 3.4 and Figure 3.5; in other words, this basic algorithm is a summary of the section above. We have omitted details such as how t_a and g_f are updated in the case that $g_f \geq g_i$, however we still keep the big picture of what is necessary at each step.

Input : τ
Output: $f_1, \cdots, f_n$
Initialization: $t_a = 0$, $g_f = g_{max}$, $i = 1$
while $i \leq n$ **do**
  if $g_f \geq g_i$ **then**
    $f_i = t_a + C_i$;
    Update g_f and t_a;
    i++ ; // Next kernel
  else
    $g_i = g_i - g_f$;
    Update g_f and t_a;
  end
end

Algorithm 1: Basic real time analysis algorithm

In order to analyze a new kernel τ_i and update t_a and g_f we need to track old values of $g_f \quad \forall t \leq t_a$. Fortunately, it is only necessary to track g_f at specific

points of time. Some relevant points of time , as it was shown in the previous example described by Figure 3.5, are given by completion times of previous kernels, in other words we must track g_{i-k} and f_{i-k} where $k \in 1, 2, ..., i-1$, because updated values of g_f and t_a depend on these as well.

Let us define a set h of pair of values (t_k, g_k) where g_k are the number of free blocks at $t = t_k$ such that $t_k \geq t_a$. In a further example we will show step by step how this array h is filled and updated in order to have a better understanding.

A complete version of our algorithm is presented in Algorithm 2.

Input : τ
Output: $f_1, \cdots, f_n$
Initialization: $t_a = 0$, $g_f = g_{max}$, $i = 1$, $h = \{\}$
while $i \leq n$ **do**
  if $g_f \geq g_i$ **then**
    $f_i = t_a + C_i$;
    $h = \{h; (f_i, g_i)\}$;
    $t_a = t_a$;
    $g_f = g_f - g_i$;
    i++ ; // Next kernel
  else
    $g_i = g_i - g_f$;
    $h = \{h; (t_a + C_i, g_f)\}$;
    $[t_a, \text{index}] = \min(h[:, 1])$;
    $g_f = h[\text{index}, 2]$;
  end
end

Algorithm 2: Real time analysis algorithm

Our algorithm is based on three main updates: h, t_a and g_f. The set h can be seen as an array of size Nx2, where N is the number of tracked pairs. For this reason, when $g_f > g_i$ we used MATLAB notation of `min` function `[value, index] = min(A)`, where `index` is the position of the pair or row $(t_k, g_k) \in h$ that has the minimun of all time values saved in h. Once we know which pair has the minimun time, we just assign $t_a = t_k$ and $g_f = g_k$. It is important to mention again that by definition of h, all the tracked times should be greater

or equal than the current t_a, meaning that pairs that have tracked times lower than t_a must be removed.

3.5 Example

Let us say there are four kernels we want to allocated, all with the same period $T = 15$ and block size of 512 threads, $b = 512$, which means $g_{max} = 8$. The four tasks are defined as $\tau = \{\tau_1 = \{15, 4, 2, 512\}, \tau_2 = \{15, 6, 7, 512\}, \tau_3 = \{15, 6, 2, 512\}, \tau_4 = \{15, 5, 5, 512\}\}$.

At the beginning $t_a = 0$, $i = 1$, $h = \{\}$ and $g_f = g_{max} = 8$. Let us start with τ_1. Kernel τ_1 and inital state of GPU are shown in Figure 3.6(a) and Figure 3.6(b) respectively.

- $g_f \geq g_1$? yes, because $g_1 = 2$
- $f_1 = t_a + C_1 = 0 + 4 = 4$
- $h = \{h, (f_1, g_1)\} = \{(4, 2)\}$
- $t_a = 0$
- $g_f = g_f - g_1 = 8 - 2 = 6$
- $i = 2$

After τ_1 allocation the GPU state is as shown in Figure 3.6(c), as it is observed, t_a remains the same but g_f now is 6. Furthermore, that is the initial GPU state when τ_2 arrives.

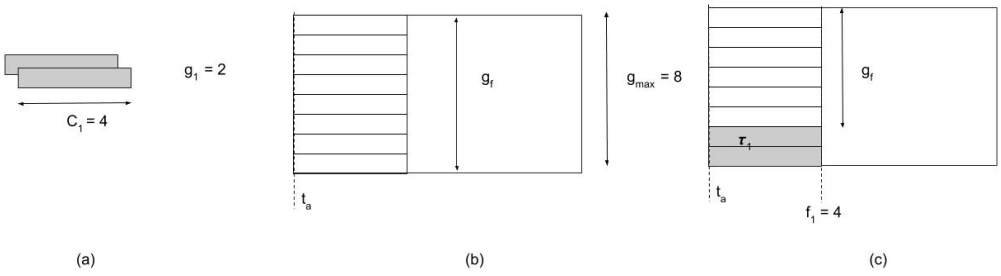

Figure 3.6: (a) Kernel τ_1 (b)GPU state prior to τ_1 allocation (c) GPU state after τ_1 allocation

Since $i = 2$, it's time to analyze τ_2. Figure 3.7(a) shows the number of blocks that should be allocated for τ_2. It this case $t_a = 0$ and $g_f = 6$ as shown in Figure 3.7(b).

- $g_f \geq g_2$? no, because $g_i = 7$
- $g_2 = g_2 - g_f = 7 - 6 = 1$
- $h = \{h, (t_a + C_2, g_f)\} = \{(4, 2), (6, 6)\}$
- $[t_a, \text{index}] = \min(h[:, 1]) = \min([4, 6])$
- $[t_a, \text{index}] = [4, 1]$
- $g_f = h[\text{index}, 2] = h[1, 2] = 2$
- $h = h - \{(4, 2)\} = \{(4, 2), (6, 6)\} - \{(4, 2)\}$
- $h = \{(6, 6)\}$

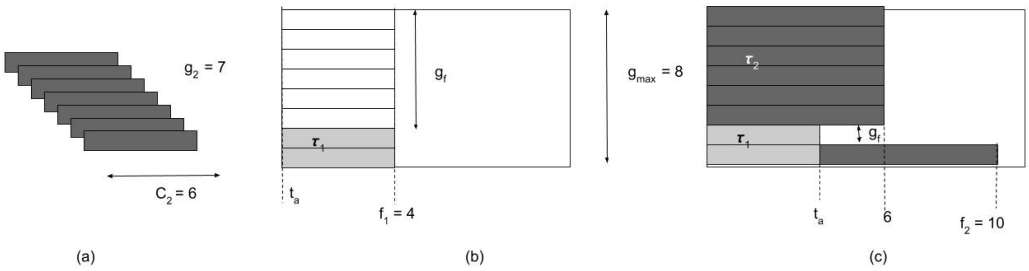

Figure 3.7: (a) Kernel τ_2 (b)GPU state prior to τ_2 allocation (c) GPU state after τ_2 allocation

Values for t_a, g_f, g_2 and h were updated. Notice that current value of t_a is the completion time of τ_1 and g_f is g_1, that is why, as mention before, it is important to track f_1 and g_1. However, completion time for τ_2 is not known yet. Let us continue with the analysis.

- $g_f \geq g_2$? yes, because $g_2 = 1$
- $f_2 = t_a + C_2 = 4 + 6 = 10$
- $h = \{h, (f_2, g_2)\} = \{(6, 6), (10, 1)\}$
- $t_a = 4$
- $g_f = g_f - g_2 = 2 - 1 = 1$
- $i = 3$

In Figure 3.7(c) the GPU state is shown, t_a and g_f values after τ_2 allocation. This setup is the starting point for the analysis of τ_3 as observed in Figure 3.8(b). Since we already described step by step the analysis for τ_1 and τ_2, we skip some details in τ_3 analysis, however we will point out something important about h.

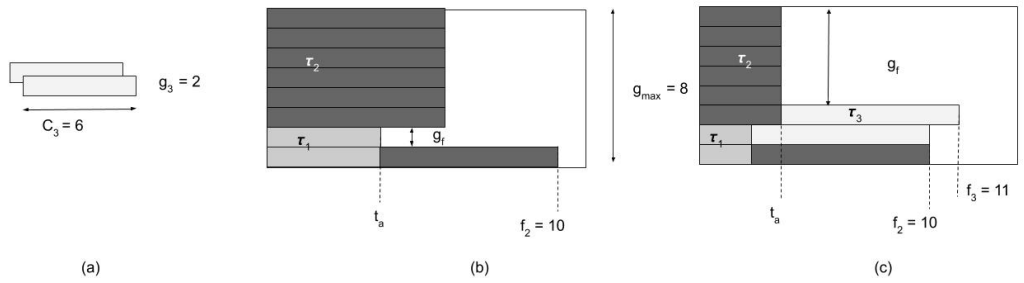

Figure 3.8: (a) Kernel τ_3 (b)GPU state prior to τ_3 allocation (c) GPU state after τ_3 allocation

The initial setup for τ_3 is $t_a = 4$, $g_f = 1$ and $h = \{(6, 6), (10, 1)\}$. The number of blocks and thread execution time of τ_3 is illustrated in Figure 3.8.

- $g_f \geq g_3$? no
- $g_3 = g_2 - g_f = 1$
- $h = \{(6, 6), (10, 1), (10, 1)\}$
- $h = \{(6, 6), (10, 2)\}$
- $[t_a, \text{index}] = [6, 1]$
- $g_f = h[\text{index}, 2] = h[1, 2] = 6$
- $h = \{(10, 2)\}$

As mention before, we performed an *extra* step with h in which h went from having three pairs to having just two. The reason behind it lies on the definition of h. The set h of pair of values (t_k, g_k) where g_k are the number of free blocks at $t = t_k$; notice that at $t = 10$ there are two *free* tracked blocks, one that comes from τ_2 and other from τ_3 as shown is Figure 3.8(c), as well as the results of the following τ_3 analysis.

- $g_f \geq g_3$? yes

- $f_3 = t_a + C_3 = 12$
- $h = \{(10, 2), (12, 1)\}$
- $t_a = 6$
- $g_f = g_f - g_3 = 5$
- $i = 4$

The analysis for τ_4 is straightforward. The blocks and thread execution time for τ_4, the GPU prior to τ_4 allocation and GPU state after τ_4 allocation are shown in Figure 3.9. It is easy to find out that the completion time f_4 is 11.

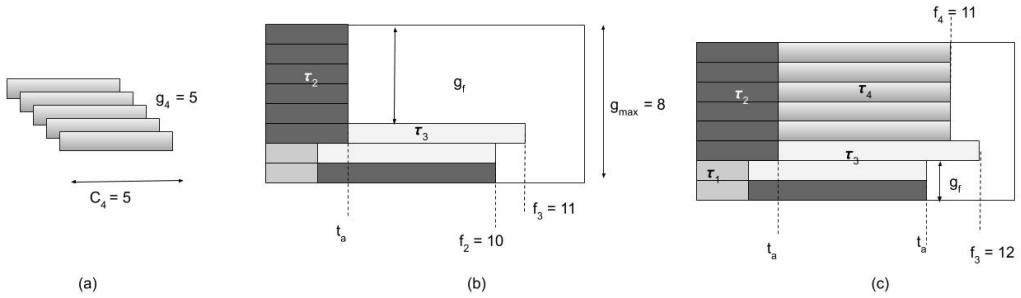

Figure 3.9: (a) Kernel τ_4 (b)GPU state prior to τ_4 allocation (c) GPU state after τ_4 allocation

We have calculated the completion times $f_i \quad \forall i \in \tau$, $f = \{4, 10, 12, 11\}$. Given the fact that all kernels were scheduled at the same time, the release time for all kernels is 0. Thus, the response time for each kernel is the same as their completion times. Furthermore, we can conclude that all the kernels can be scheduled because $R_i \leq T \quad \forall i \in \tau$.

3.6 A Special case

In this section we demostrate that if all kernels are released at the same time and also have the same thread execution time, $C_i = C \quad \forall i \in \tau$, our response time analysis has not algorithmic behavior, instead it's a set of three equations.

In this special case, there is no need of h, because t_a and g_f can be calculated directly with two equations. The goal is, as always, to find t_a, g_f and f_i, however, we will exploit the fact that C_i is the same for all kernels. The case in which

$g_i \leq g_f$ is trivial to analyze, therefore we will analyze the other case. If Figure 3.10 the block distribution when $g_i > g_f$ is shown, thus in order to find g_f^* and t_a^*, that are the updated values, we must calculate K.

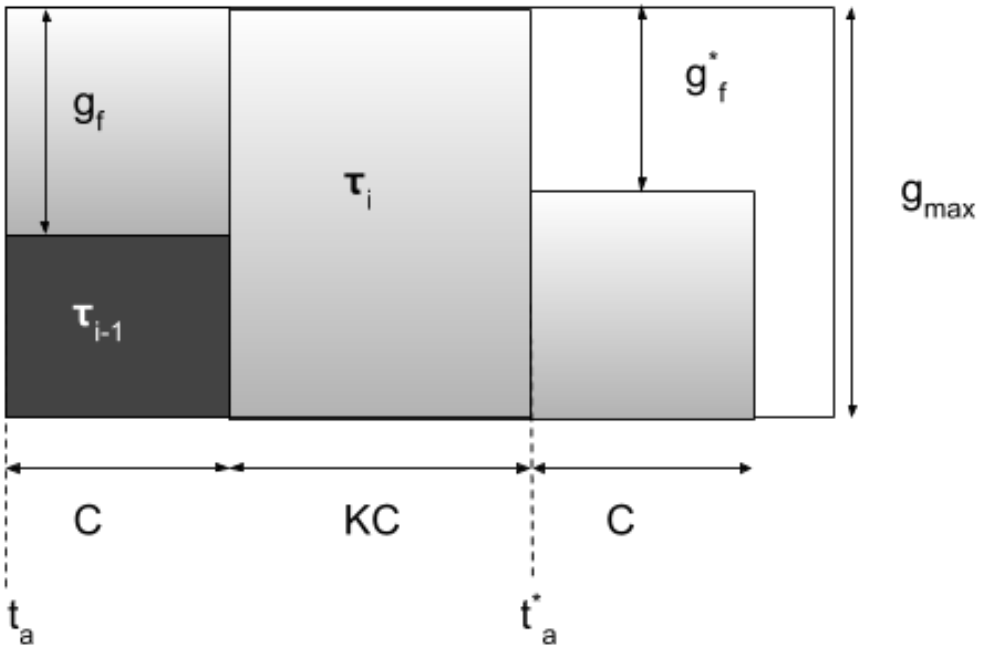

Figure 3.10: New kernel allocation

It is not hard to see that K is given by equation 3.8. The interpretation is that K is the maximum amount of times that g_{max} blocks can be allocated.

$$K = \lfloor \frac{g_i - g_f}{g_{max}} \rfloor \tag{3.8}$$

From K , the value of g_f^* is calculated using equation 3.9. If we observed the blocks in Figure 3.10, we notice that g_f^* can be calculated using geometry. The area of the first block is g_f, of the second one Kg_{max} and the last one is $g_i - g_f - Kg_{max}$.

$$g_f^* = g_{max} - (g_i - g_f - Kg_{max}) \tag{3.9}$$

The value of t_a^* is calculated in a similar fashion and is given by equation 3.10. On the other hand, the calculation of f_i remains the same.

$$t_a^* = t_a + C + KC \tag{3.10}$$

We are at a point at which we are able to calculate, with formulas, the updated values of t_a and g_f, nonetheless, we still have the algoritmic behavior given by the `if` condition. Therefore to no longer use that dependency, we will make use of the absolute value and the signum function. The signum function of a real number x is defined as follows:

$$\text{sgn}(x) = \begin{cases} -1 & \text{if x<0,} \\ 0 & \text{if x=0,} \\ 1 & \text{if x>0.} \end{cases} \tag{3.11}$$

We redefine K with the absolute value using equation 3.12, we use absolute value in 3.12 because we want K to be zero when $g_i \leq g_f$.

$$K = \lfloor \frac{\|g_i - g_f\|}{g_{max}} \rfloor \tag{3.12}$$

In addition, we define α which value is given by equation 3.13. The value of α is zero when $g_i \leq g_f$ and one otherwise.

$$\alpha = \frac{\text{sgn}(g_i - g_f) + 1}{2} \tag{3.13}$$

With 3.12 and 3.13 we eliminate the `if` condition and at the same time the algoritmic behavior. Therefore, the updated values of t_a and g_f are calculated using 3.14 and 3.15. To emphasize again, when $g_i \leq g_f$ 3.14 and 3.15 are the same steps described in Algorithm 2.

$$t_a = t_a + \alpha C + KC \tag{3.14}$$

$$g_f = \alpha g_{max} - (g_i - g_f - K g_{max}) \tag{3.15}$$

3.7 Computational Complexity

Our algorithm described in Algorithm 2 has two branches, inside the `while` loop, given by an `if` conditional. In Table 3.1 is shown the computational complexity of each step of the real time analysis algorithm.

Table 3.1: Computational Complexity

Step	Type of operation	Average Cost
$f_i = t_a + C_i$	Sum	O(1)
$h = \{h; (f_i, g_i)\}$	Append	O(1)
$t_a = t_a$	Sum	O(1)
$g_f = g_f - g_i$	Sum	O(1)
i++	Sum	O(1)
$g_i = g_i - g_f$	Sum	O(1)
$h = \{h; (t_a + C_i, g_f)\}$	Append	O(1)
$[t_a, \text{index}] = \min(h[:, 1])$	Min	O(n)
$g_f = h[\text{index}, 2]$	Index	O(1)

The first branch is when $g_f \geq g_i$. The computational complexity, given by big O notation, of that branch is O(1), because all the operations in this branch are O(1).

In the case of the second branch the computational complexity is O(n), because `min` function is the most costly operation. In the worst case scenario `n` is the number of kernels we want to allocate, it is important to highlight that only on

the first branch the lenght of h increases. Thus, the computational complexity of the `if` statement is O(n).

Let us analyze the outer `while` loop. The number of iterations depends on number of kernels, their grid sizes g_i and how many blocks can be allocated in total in the GPU or g_{max}. An estimation can be given by $\frac{g}{g_{max}}$, where g is $\sum g_i$, g contains the information about number of kernels and their grid sizes. Thus, computational complexity of our algorithm is O($\frac{ng}{g_{max}}$).

It can be observed in Figure 3.11 the relationship between computation time and number of kernels and total of allocated blocks. The right side graph shows the polinomial behavior between time and number of kernels. On the other hand, the lelf side graph shows a more linear relation between computation time and number of allocated blocks.

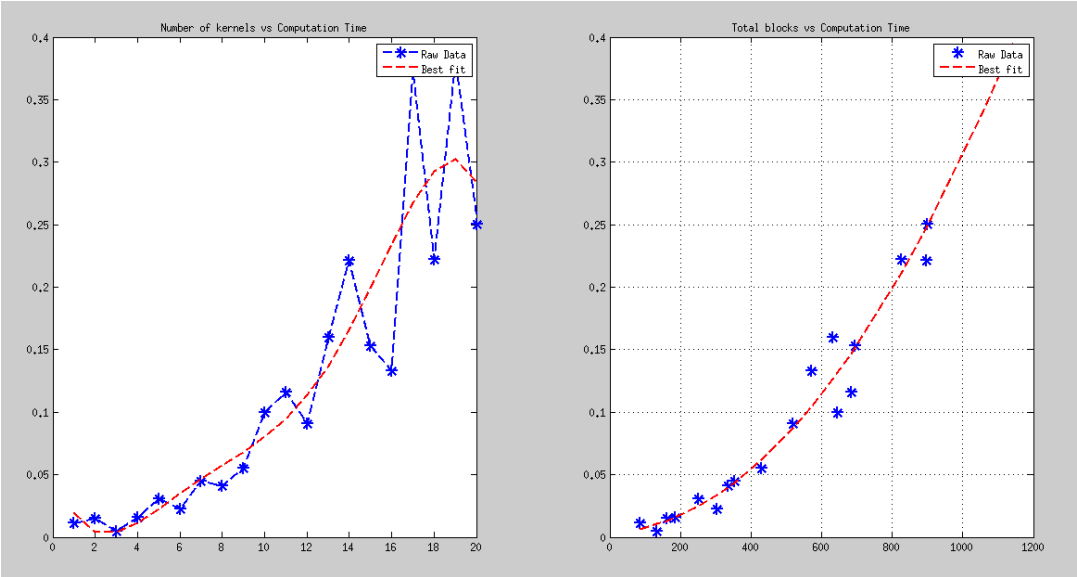

Figure 3.11: Best fit for computational complexity

We analyze in more detail the influence of g_{max} and maximum number of allocated blocks on computation time. Let us assume that the number of blocks per kernel increase in linear fashion to the number of kernels. In other words, the more kernels we want to launch, the more blocks each kernel will have. As shown in Figure 3.12b. Results are shown in Figure 3.12. The value of g_{max} has

a strong influence on computation times. When $g_{max} = 128$ computation times are an order of magnitude lower than when $g_{max} = 8$. The inverse relation is consistent with computational complexity. The value of g_{max} tells us how many blocks can be allocated on the GPU at the same time which implies fewer iterations of our algorithm.

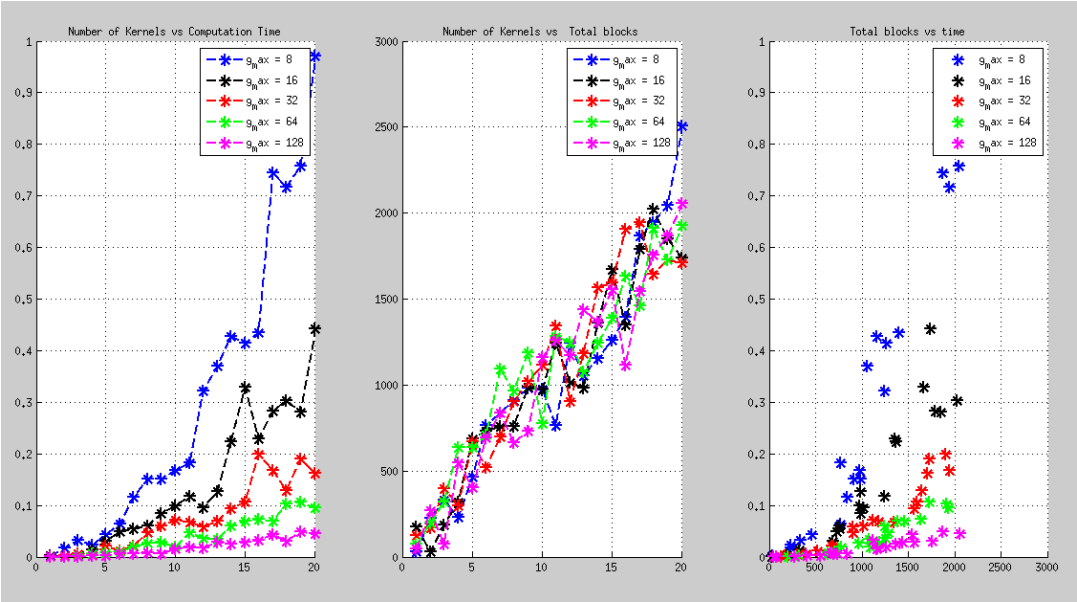

Figure 3.12: Analysis of computational complexity when g_{max} is variable

Let us assume now the value of g_{max} is constant, but the number of of allocated blocks increase as shown in Figure 3.13b. Results are shown in Figure 3.13. In Figure 3.13c, we can see that for the same amount of blocks the algorithm runs faster because there are fewer kernels.

The graphs above tell us that our algorithm perform better when the number of threads per block is low. In addition, the grid size has a minor influence on the algorithm performance as shown in Figure 3.13.

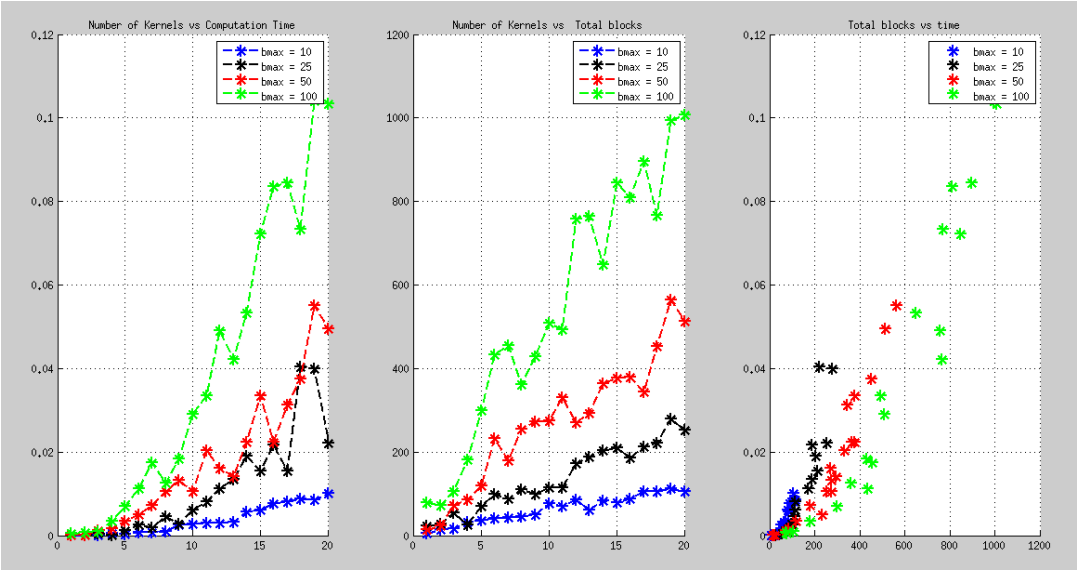

Figure 3.13: Analysis of computational complexity when b_{max} is variable

Chapter 4

Experimental Results

In this chapter we presents our experimental results. A complete example using AMALTHEA models can be found in the Appendix 1. We compare results from Jetson TX2 platfrom againts our APP4MC implementation. The former are used as ground truth to verify our implementation and assumptions.

4.1 Ground truth generation

Amert et. al [12] published their code in github. They developed a CUDA Scheduling Viewer, which is a tool for examining block-level scheduling behavior and co-scheduling performance on CUDA devices. The input are configuration files on the JSON format, and the output can be displayed as figure using a Python script, which is provided as well. An example output is shown in Figure 4.1

Our test scenario was similar to the example presented in the last chapter. We had four kernels we wanted to allocate. The parameters were: block size = 512 threads, and $g_{max} = 8$. The four kernels were defined as $\tau = \{\tau_1 = \{15, 4, 2, 512\}, \tau_2 = \{15, 6, 7, 512\}, \tau_3 = \{15, 6, 2, 512\}, \tau_4 = \{15, 5, 5, 512\}\}$.

An example of a kernel description in the configuration file was as follows:

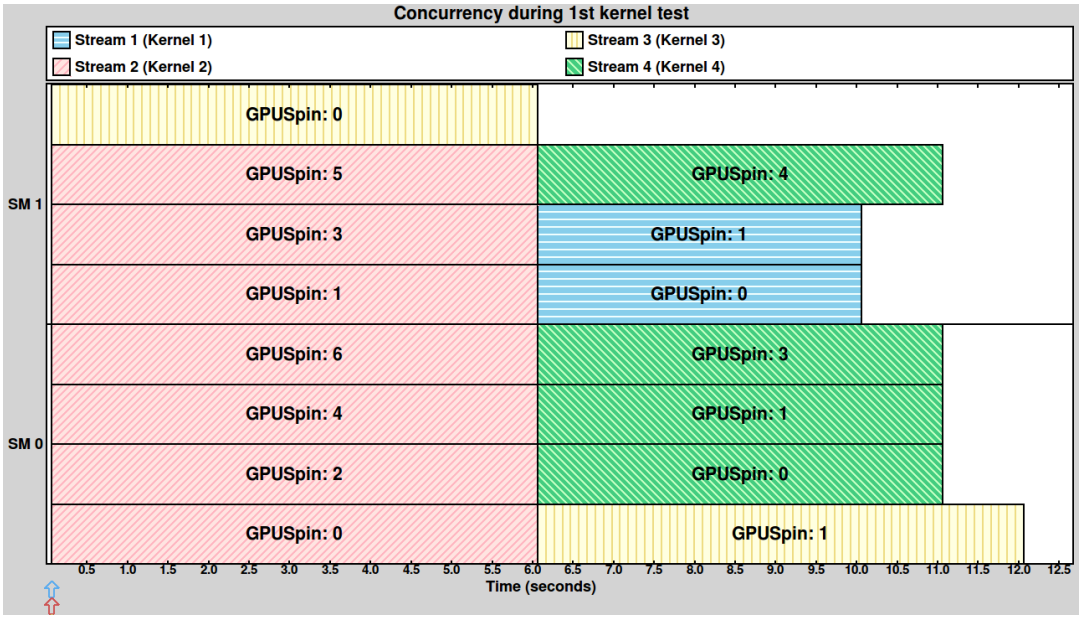

Figure 4.1: Output of the CUDA Scheduling Viewer

```
"filename": "./bin/timer_spin.so",
"log_name": "k3.json",
"label": "Kernel 3",
"thread_count": 512,
"block_count": 2,
"additional_info": 6000000000
```

The `filename` is the benchmark binary file we used as a kernel. For all the kernels was `timer_spin.so`. This file defines a bare-bones CUDA benchmark which spins waiting for a user-specified amount of time to complete. The execution time in nanoseconds or C_i was set as `additional_info`. The `log_name` was the JSON file that contained metadata and results related to a specfied kernel (`label`). In addition, `thread_count` and `block_count` were the values of b_i and g_i respectively.

4.2 Implementation results

We implemented our algorithm in Eclipse APP4MC. The goal was not to test how many kernel the Jetson could manage, instead we focused on verifying our assumptions and therefore our algorithm.

We set up three test scenarios. The four previously described kernels were launched in different order. The first scenario was the one presented in Figure 4.1. The kernels were launched on the following order: K2, K3, K4, K1. As showed in Figure 4.1 the completion times were $f = \{6, 12, 11, 10\}$. The results from APP4MC are shown in Figure 4.2.

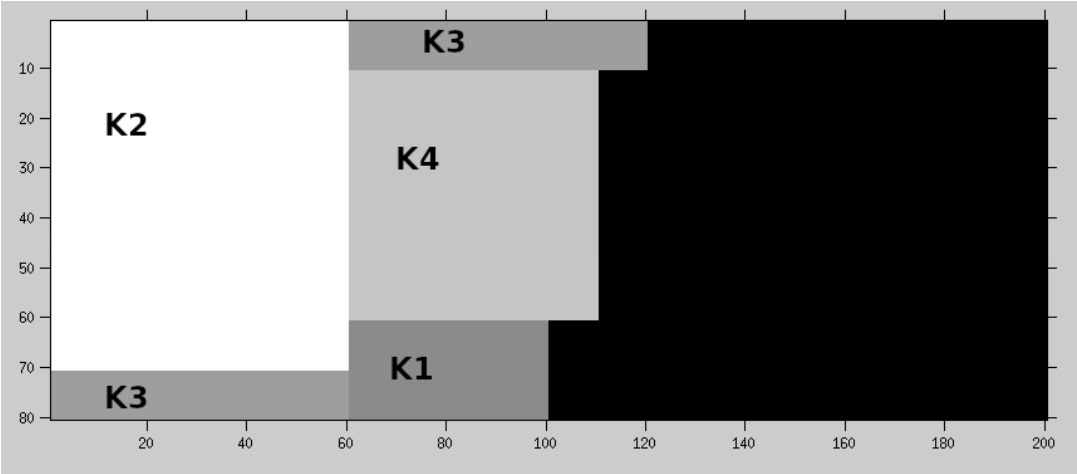

Figure 4.2: APP4MC: Scenario 1 - K2,K3,K4,K1

In the second scenario kernels were launched on the following order: K2, K4, K1, K3. As observed in Figure 4.3 the completion times were $f = \{6, 11, 10, 12\}$. Notice that *GPUSping: 5* for kernel 4 should be shown, but there is a bug in the code from [12] in which sometimes the log file doesn't contain all the data. On the other hand, results from APP4MC are shown in Figure 4.4. The block allocation differ from Jetson's allocation because our code follows our assumption described in section 3.3.2.

In the third scenario kernels were launched on the following order: K2, K1, K3, K4. As observed in Figure 4.5 the completion times were $f = \{6, 8, 12, 11\}$.

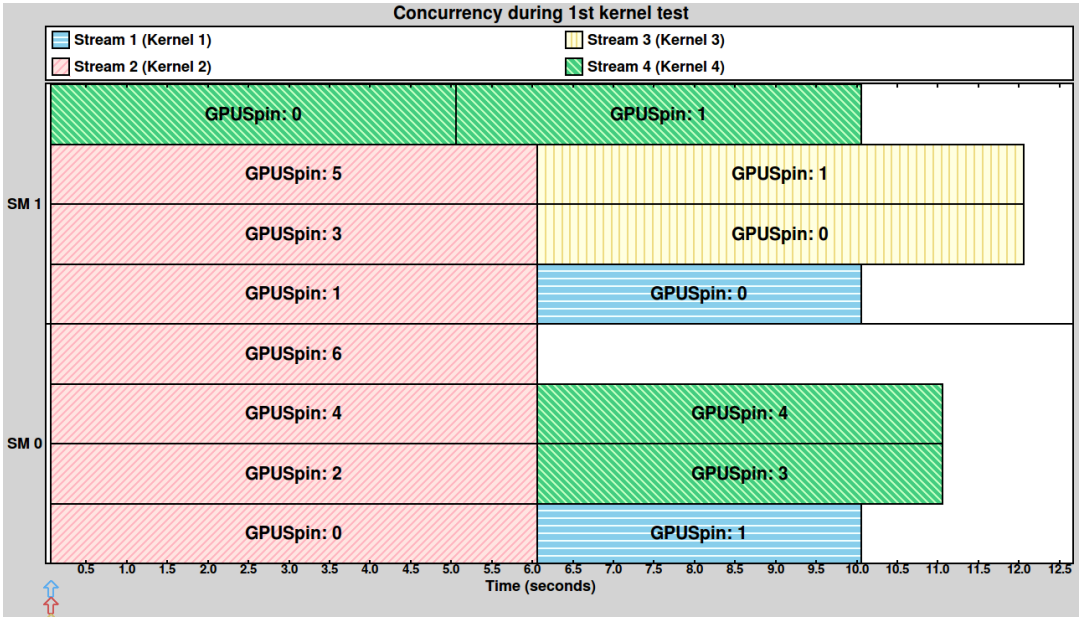

Figure 4.3: JetsonTX2: Scenario 2 - K2,K4,K1,K3

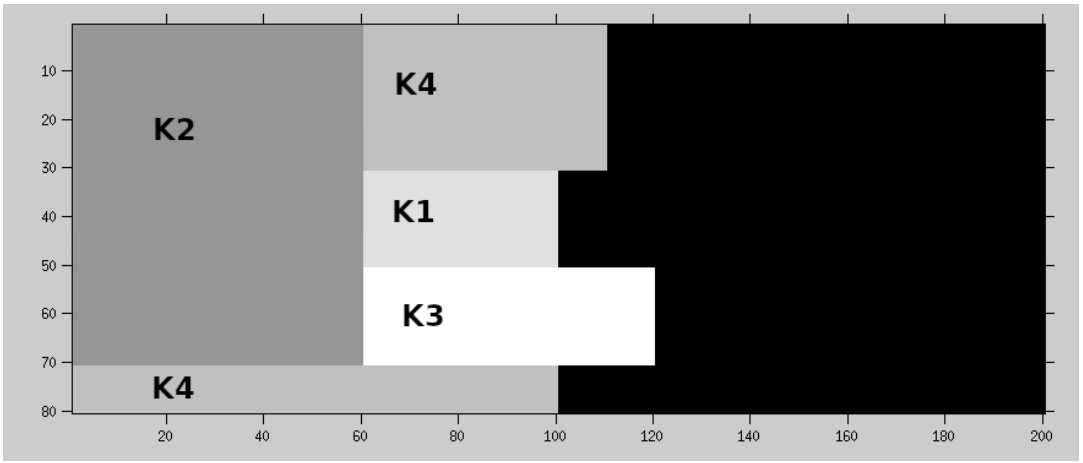

Figure 4.4: APP4MC: Scenario 2 - K2,K4,K1,K3

Notice in this case that *GPUSping:4* from kernel 4 and *GPUSping:1* from kernel 5 overlap in the figure. This is, again, an error on how the log file was created. We tested [12] C implementation using `printf`, and the values were correct. Nevertheless, results from APP4MC shown in Figure 4.4 remain congruent with the results of its counterpart.

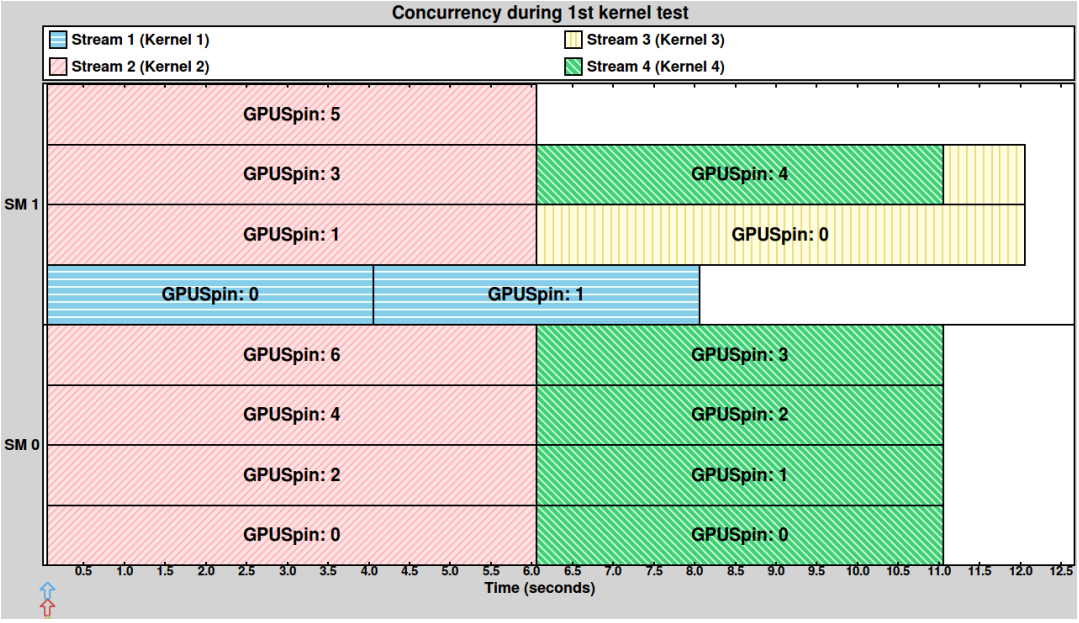

Figure 4.5: JetsonTX2: Scenario 3 - K2,K1,K3,K4

4.3 More results

In this section we present more results. We focused in this section on the interaction between kernels with several blocks, and kernels with long execution time.

In Figure 4.7 can be observed the result after executing five kernels. The kernels were defined as follows:

- $\tau_1 = \{15, 4, 2, 512\}$: small block count, short execution time.
- $\tau_2 = \{15, 7, 1, 512\}$: big block count, short execution time.
- $\tau_3 = \{15, 10, 4, 512\}$: big block count, long execution time.

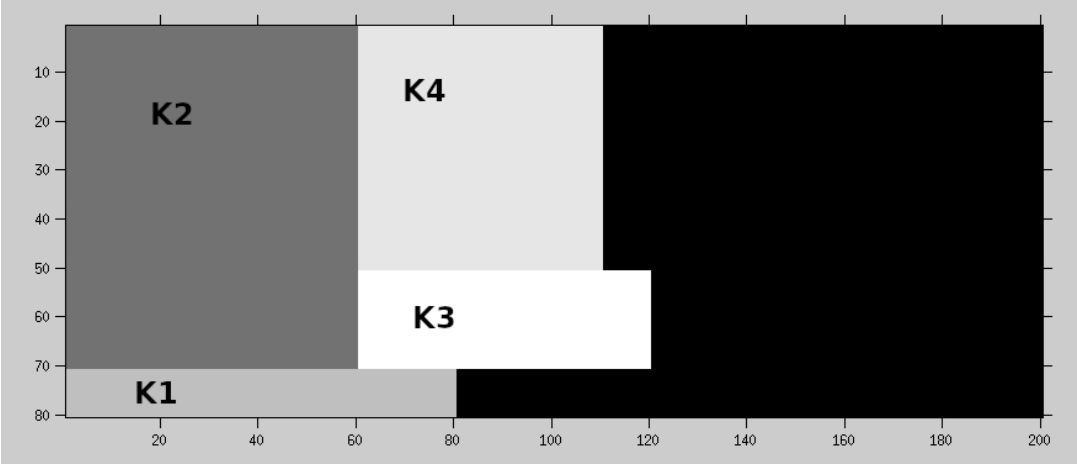

Figure 4.6: APP4MC: Scenario 3 - K2,K1,K3,K4

- $\tau_4 = \{15, 1, 11, 512\}$: small block count, very long execution time.
- $\tau_5 = \{15, 3, 6, 512\}$: big block count, medium execution time.

Our result shown in Figure 4.8 is still consistent with the ground truth. We obtained the same completion time for each kernel.

In this experiment the setup was as follows:

- $\tau_1 = \{15, 5, 1.5, 512\}$: small block count, small execution time.
- $\tau_2 = \{15, 4, 1.5, 512\}$: small block count, small execution time.
- $\tau_3 = \{15, 7, 1.5, 512\}$: big block count, small execution time.
- $\tau_4 = \{15, 1, 4, 512\}$: very small block count, big execution time.
- $\tau_5 = \{15, 8, 2.5, 512\}$: big block count, large execution time.

As expected completion times shown in Figure 4.9 and Figure 4.10 are the same for each kernel.

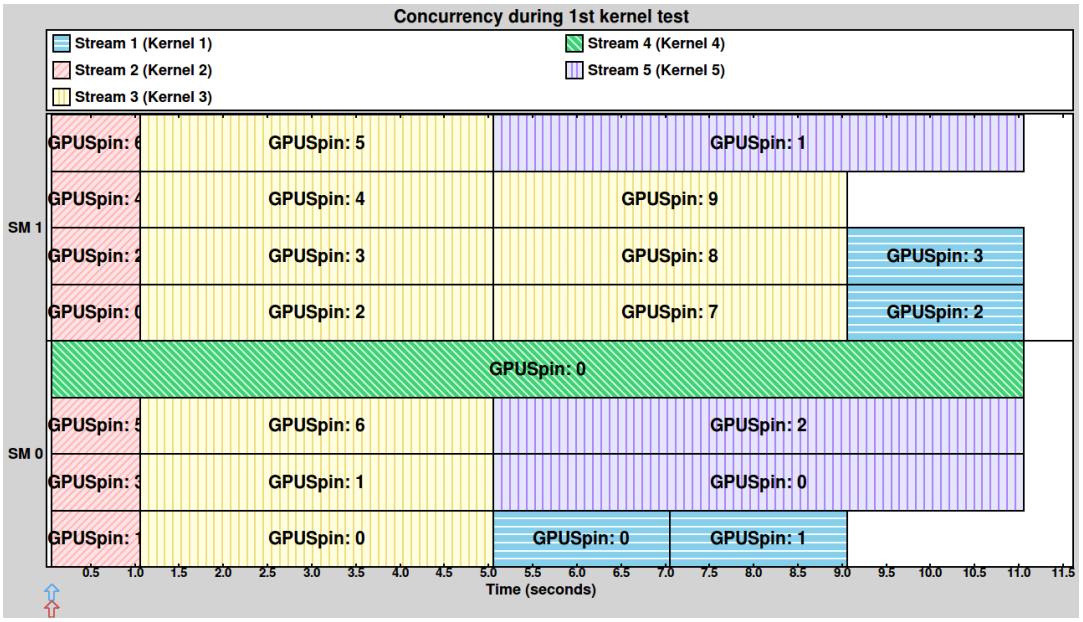

Figure 4.7: JetsonTX2

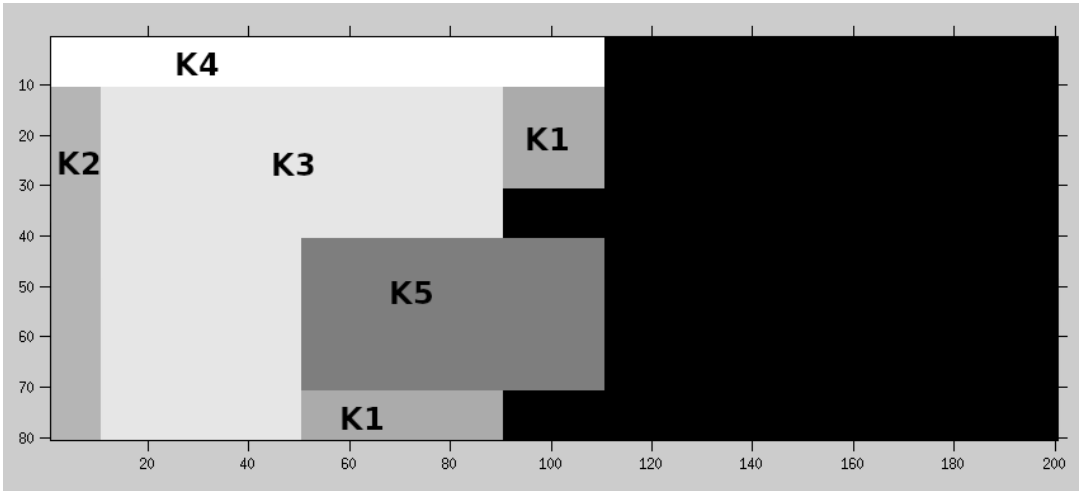

Figure 4.8: APP4MC

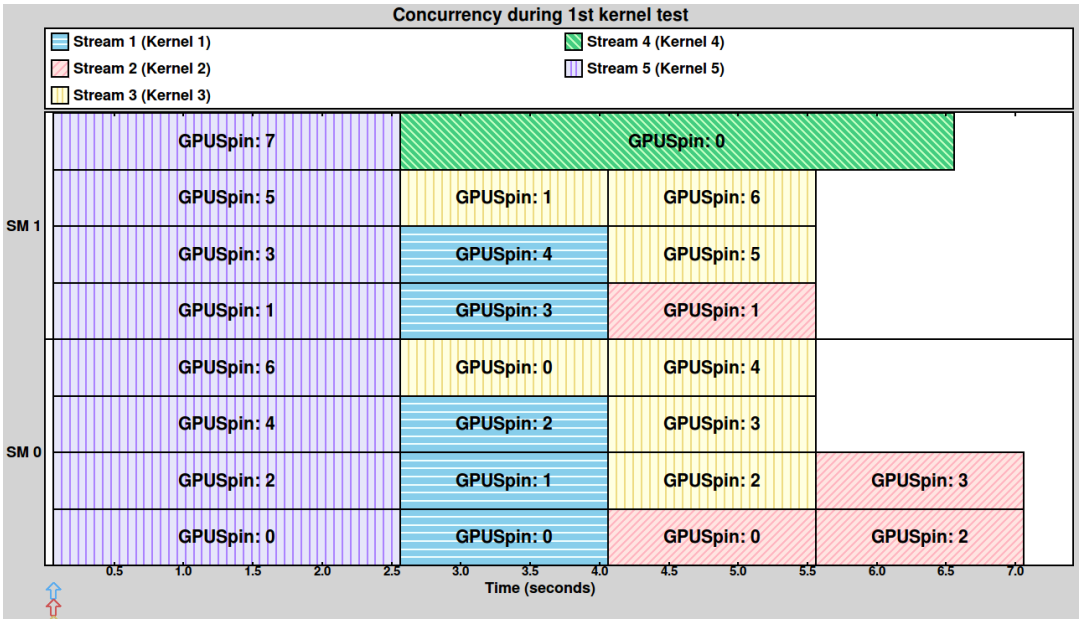

Figure 4.9: JetsonTX2

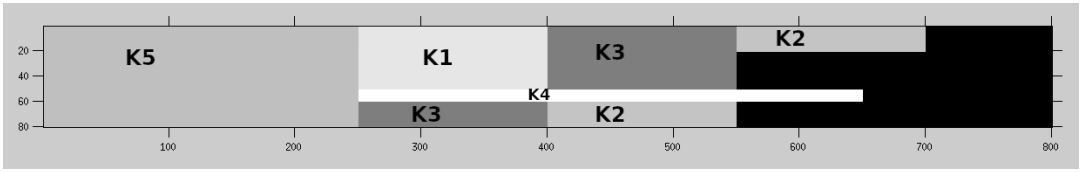

Figure 4.10: APP4MC

Chapter 5

Conclusion

5.1 Thesis summary

We gave an overview, in Chapter 1, of the motivation of this work, the Bosch WATERS Challenge 2019, and explained the architecture of NVIDIA Jetson TX2 platform and its AMALTHEA model. Furthermore, in Chapter 2, we introduced key concepts related to NVIDIA's GPU such as kernel definition, block and threads, as well as its memory hierarchy. We described in detail Jetson TX2 hardware architecture, and explained the rules behind hardware scheduler. In Chapter 3 we introduced our real time analysis algorithm for Jetson TX2's scheduler. In addition we gave some examples to explain in detail each step. Finally in Chapter 4 we showed our experimental results.

5.2 Conclusions

It has been proved that for purposes of mathematical calculation, the main assumption, due to all blocks have the same thread count we considered all GPU streaming multiprocessors as one big streaming multiprocessor, was correct. Our experimental results show the accuracy of our algorithm to estimate completion times for kernels executed on Jetson TX2 platform. Moreover, we implemented

our algorithm on Eclipse APP4MC, which allows AMALTHEA based response time analysis for NVIDIA's Jetson TX2.

5.3 Future work

There are several potential directions for extending this thesis. First, the possibility to develop a complete model-based development for NVDIA platform using AMALTHEA models, which may include automatic CUDA C code generation for deployment and testing. Second, there is still no full understanding of how Jetson TX2's scheduler decides which kernel should run first. Thus, developers should either consider all possible cases when analyzing completion times or reverse engineering Jetson TX2's scheduler. Finally, we didn't consider memory transaction and other constrains such as amount of shared memory, influence of the *null* stream and priorities within scheduler. Therefore, there is still room to go deeper into this topic.

Appendix 1: Example APP4MC

Here is a complete example of how to use our response time analysis algorithm in APP4MC to analyze AMALTHEA based Jetson TX2's models.

Listing 5.1: Complete example

```
1  /**
2   ************************************************************
3   * Copyright (c) 2018 Robert Bosch GmbH.
4   *
5   * This program and the accompanying materials are made
6   * available under the terms of the Eclipse Public License 2.0
7   * which is available at https://www.eclipse.org/legal/epl-2.0/
8   *
9   * SPDX-License-Identifier: EPL-2.0
10  *
11  * Contributors:
12  *     Robert Bosch GmbH - initial API and implementation
13  ************************************************************
14  */
15
16 package app4mc.example.tool.java;
17
18 import java.util.ArrayList;
19 import java.util.HashMap;
20 import java.util.Map;
21 import java.util.Random;
22
23 import org.eclipse.app4mc.amalthea.model.AmaltheaFactory;
24 import org.eclipse.app4mc.amalthea.model.DiscreteValueConstant;
25 import org.eclipse.app4mc.amalthea.model.SWModel;
26 import org.eclipse.app4mc.amalthea.model.Ticks;
27 import org.eclipse.app4mc.amalthea.model.Runnable;
28 import org.eclipse.app4mc.amalthea.model.util.*;
29 import org.eclipse.emf.common.util.EList;
30
31 public class rta{
32
33     public static Long findIndexOfMinValue(Map<Long, Integer> hashMap){
34         ArrayList<Long> al = new ArrayList<Long>();
```

```
35          for (Long m: hashMap.keySet()) {
36                  al.add(m);
37          }
38          Long minVal  = Long.MAX_VALUE;
39          for (int i=0; i<hashMap.size();i++) {
40              if (al.get(i)<minVal) {
41                  minVal = al.get(i);
42              }
43          }
44
45          return minVal;
46  }
47
48      public static Map<Long, Integer> updateH ( Map<Long, Integer> h, Long ticks, Integer
          blocks){
49          if ( h.containsKey(ticks) ){
50              h.put( ticks , h.get( ticks ) + blocks);
51          }
52          else h.put( ticks, blocks );
53          return h;
54      }
55
56      // Input: swmodel
57      // Output: Completion times
58      public static ArrayList<Long> rtaAlgorithm(SWModel swmodel){
59          EList<Runnable> rList = swmodel.getRunnables();
60
61          ArrayList<Long> c_i = new ArrayList<Long>();
62          ArrayList<Integer> g_i = new ArrayList<Integer>();
63          ArrayList<Long> f = new ArrayList<Long>();
64
65          // Set values c_i, g_i
66          for (int i = 0; i < rList.size(); i++) {
67              Runnable rr = rList.get(i);
68              c_i.add( ((DiscreteValueConstant) SoftwareUtil.getTicks(rr, null).get(0).getDefault()).
          getValue() );
69              g_i.add( CustomPropertyUtil.customGetInteger(rr, "GridSize") );
70      }
71
72          // Initialization algorithm
73          Long t_a = (long) 0;
74          Integer g_max = 8;
75          Integer g_f = g_max;
76          Map< Long, Integer> h = new HashMap<Long, Integer>();
77          int current_kernel = 0;
78          Long minimumRegisteredTicks;
79
80          // Main loop
81          while ( current_kernel < rList.size() ) {
82              if (g_f >= g_i.get(current_kernel) ){
83                  f.add(current_kernel, t_a + c_i.get(current_kernel) ) ;
84
85                  h = updateH(h, f.get(current_kernel), g_i.get(current_kernel) );
86
87                  g_f = g_f − g_i.get(current_kernel);
```

```
88              current_kernel++;
89          }
90          else {
91              g_i.set(current_kernel, g_i.get(current_kernel) − g_f);
92
93              h = updateH(h, t_a + c_i.get(current_kernel), g_f );
94              minimumRegisteredTicks = findIndexOfMinValue(h);
95
96              g_f = h.get(minimumRegisteredTicks);
97              t_a = minimumRegisteredTicks;
98
99              h.remove(minimumRegisteredTicks);
100         }
101     }
102
103     return f;
104 }
105
106 public static void main(String[] args) {
107   // Creating a SWModel
108     SWModel swmodel = AmaltheaFactory.eINSTANCE.createSWModel();
109
110     Random rand = new Random();
111     int tick;
112     int NumberOfRunnables = 5;
113     int minTicks = 10;
114     int maxTicks = 20;
115     int minGridSize = 2;
116     int maxGridSize = 20;
117     int gridSize;
118
119     for(int i=0; i<NumberOfRunnables; i++) {
120       Runnable r = AmaltheaFactory.eINSTANCE.createRunnable();
121       Ticks ticks = AmaltheaFactory.eINSTANCE.createTicks();
122       DiscreteValueConstant dvc = AmaltheaFactory.eINSTANCE.
    createDiscreteValueConstant();
123         tick = rand.nextInt((maxTicks − minTicks) +1) + minTicks;
124       dvc.setValue(tick);
125       ticks.setDefault(dvc);
126       r.getRunnableItems().add(ticks);
127       gridSize = rand.nextInt((maxGridSize − minGridSize) +1) + minGridSize;
128         CustomPropertyUtil.customPut(r, "GridSize", gridSize);
129       swmodel.getRunnables().add(r);
130     }
131
132     ArrayList<Long> f = new ArrayList<Long>();
133
134     f = rtaAlgorithm(swmodel);
135
136     for( int i=0; i<f.size(); i++){
137         System.out.println("\t Computation time:" + f.get(i));
138     }
139 }
140 }
```

References

[1] "BMW and audi want to separate vehicle hardware from software." https://www.electronicdesign.com/automotive/bmw-and-audi-want-separate-vehicle-hardware-software.

[2] "End to end architecture." https://www.future-mobility-tech.com/en/technology/end-to-end-architecture.

[3] S. Kanajan, H. Zeng, C. Pinello, and A. Sangiovanni-Vincentelli, "Exploring trade-off's between centralized versus decentralized automotive architectures using a virtual integration environment," in *Proceedings of the conference on design, automation and test in europe: Proceedings*, 2006, pp. 548–553.

[4] T. A Henzinger, "Two challenges in embedded systems design: Predictability and robustness," *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, vol. 366, pp. 3727–36, Nov. 2008.

[5] C. Cullmann *et al.*, "Predictability considerations in the design of multi-core embedded systems," *Ingénieurs de l'Automobile*, vol. 807, pp. 36–42, Sep. 2010.

[6] "WATERS 2019 – industrial challenge." https://www.ecrts.org/waters/waters-industrial-challenge/.

[7] D. Franklin, "NVIDIA jetson tx2 delivers twice the intelligence to the edge." https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/.

[8] *NVIDIA jetson tx2 system-on-module*. NVIDIA Corporation, 2014.

[9] E. APP4MC, "Project profile: Eclipse app4mc." http://www.amalthea-project.org/.

[10] NVIDIA, "Homepage - cuda zone." https://developer.nvidia.com/cuda-zone.

[11] *NVIDIA cuda c: Programming guide*. NVIDIA Corporation, 2010.

[12] 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)*, 2017, pp. 104–115.

[13] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, and F. D. Smith, "Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems," in *30th euromicro conference on real-time systems (ecrts 2018)*, 2018, vol. 106, pp. 20:1–20:21.

[14] J. Bakita, N. Otterness, J. H. Anderson, and F. D. Smith, "Scaling up: The validation of empirically derived scheduling rules on nvidia gpus," *OSPERT 2018*, p. 49, 2018.

[15] M. Yang and J. Anderson, "Response-time bounds for concurrent gpu," *Proceedings of 29th Euromicro Conference on Real-Time Systems Work in Progress Session*, pp. 13–15, 2019.

[16] D. Mukunoki, T. Imamura, and D. Takahashi, "Automatic thread-block size adjustment for memory-bound blas kernels on gpus," in *2016 ieee 10th international symposium on embedded multicore/many-core systems-on-chip (mcsoc)*, 2016, pp. 377–384.

[17] R. Lim, B. Norris, and A. Malony, "Autotuning gpu kernels via static and predictive analysis," in *2017 46th international conference on parallel processing (icpp)*, 2017, pp. 523–532.

[18] Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "Understanding the impact of cuda tuning techniques for fermi," in *2011 international conference on high performance computing simulation*, 2011, pp. 631–639.

[19] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning gemm kernels for the fermi gpu," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 11, pp. 2045–2057, Nov. 2012.

[20] *NVIDIA cuda c: Best practices*. NVIDIA Corporation, 2019.

[21] *NVIDIA cuda c: Programming guide*. NVIDIA Corporation, 2019.