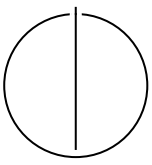
SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

GPU Coroutines in Autonomous Driving

Jaden Rotter



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

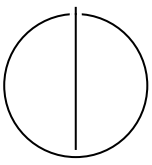
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

GPU Coroutines in Autonomous Driving

GPU Coroutines in Autonomes Fahren

Author:	Jaden Rotter
Examiner:	Supervisor
Supervisor:	Jianfeng Gu
Submission Date:	Submission date



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, Submission date Jaden Rotter

Acknowledgments

Abstract

Contents

1 Introduction

Autonomous driving presents a formidable challenge in computing, requiring complex high-performance workloads that heavily rely on GPUs, yet traditional scheduling techniques fail to deliver the necessary predictability and reliability demanded by real-time systems. GPUs hardware is optimized for maximizing throughput, a focus that compromises low-latency performance critical for real-time applications. Similarily, traditional Graphics Processing Unit (GPU) scheduling methods do not consider the strict deadlines essential for real time systems, particularily under conditions of GPU contention, where multiple tasks compete for limited computational resources. This thesis will implemented an approach to GPU scheduling using coroutines on persistant threads to reduce latency variability and address contention issues, ultimately delivering the predictability and reliability necessary for real-time autonomous driving applications.

Test

1.1 Background and Motivation

In autonomous driving and other real time system tasks, GPUs are essential for data processing and machine learning; however, existing GPU scheduling techniques fail to meet the strict timing requirements of real-time applications. GPU tasks are traditional queued and scheduled based on availabilty to optimize for high throughput applications like graphics rendering and offline machine learning training. These tasks run to completion before switching tasks, meaning multiple tasks, such as the different modules in autonomous driving cannot efficiently share resources. Thus, depending on the current workload for the GPU, execution times and latency can vary. The proposed solution uses asynchronous programming concepts to halt execution and switch tasks, enhancing responsiveness and performance. These limitations make existing GPU scheduling mechanisms unsuitable for real-time applications like autonomous driving, where deadlines need to be met to ensure the safety of the system and passengers.

1.1.1 Real Time Systems

Real time systems are designed with strict timing constraints in mind, to ensure predictable and deterministic behavior. They use a concept of deadlines, which are subdivided into soft and hard-deadlines. Hard deadlines are critical and a failure leads to the systems failure or unsafe conditions. For example, in autonomous driving, collision avoidance with another vehicle and brake activation are hard deadlines. If these deadlines are not met, the safety of the passengers and the systems is at risk. On the other hand, soft deadlines are not critical and missing these deadlines degrades performance, but does not cause system failure. In autonomous driving, this would show in route planning and navigation updates, where a delay would lead to suboptimal paths, but safety is not comprimised. Real time systems need to be capable of effectively and efficiently switching from lower priority tasks, soft deadline tasks, to high priority, hard deadline tasks to ensure the system safety and reliability.

1.1.2 Autonomous Driving Systems

Autonomous driving systems, such as self-driving cars, are subdivided into several modules, perception, localization, mapping, planning and control, which help the system to understand its enviornment, plan its trajectory and drive without any human help. Perception gathers and interprets vast amounts of data from numerous sensors, such as Light Detection and Ranging (LiDAR) sensors, cameras, radar, and ultrasonic sensors, to identify and classify surrounding objects. After perception, localization and mapping use a Simultaneous Localization and Mapping (SLAM) algorithm to build maps and determine the vehicle's precise location within its enviornment and the generated map. Using the foundation set by localization and mapping, path planning determines the vehicles trajectory from its current location to a target destination, while respecting traffic laws and avoiding obstacles. Lastly, control actively follows the trajectory and sets the speed, steering and braking required for the planned path. Autonomous driving sytems are constantly computing each of these modules based on updates in the surroundings, which requires significant computational ressources to maintain a low latency.

These autonomous driving modules use vast amounts of data for their computation tasks, which are based in deep learning, a machine learning technique that demands vast amounts of computational resources. Deep learning, inspired by the structure of the human brain, utilizes neural networks, with many layers of neural nodes, to automatically extract patterns, enabling decision-making tasks such as classification and object detection. In perception, deep learning models, particularily CNNs identify objects, lane markings, pedestrians, vehicles, and traffic lights from the raw sensor

data. Beyond perception, deep learning enhances localization by refining sensor fusion techniques, improving accuracy in determining the vehicles position. Furthermore, deep learning also supports planning and control, through reinforcement learning and predictive models to optimize trajectories. The real-time nature of these components are needed to reduce the latency, which requires these models to continously be updating their predictions as new sensor input arrives and the vehicle moves. This places a high demand on the hardware, requiring high-performance accelerators to process vast amounts of data efficiently. To meet these requirements, these computations are scheduled on to a GPU, which provides the high degree of parallelism required by the vast amount of data and computations within autonomous driving.

1.1.3 GPUs

GPUs have revolutionized data processing with their ability to handle massive amounts of data and execute highly-parallelized computations, capabilities fundamental to artificial intelligence (AI) and machine learning tasks. Traditional CPUs, by contrast, rely on a multiple instruction multiple data (MIMD) architecture that excels at handling complex control logic at high frequencies by executing different instructions on seperate data streams concurrently. However, for large-scale data processing tasks that require the repeated execution of singular instructions, CPUs are burdened by the overhead of complex control logic, where a simpler, more parallel design would be more effective. GPUs, created to overcome this limitation, use a single instruction multiple thread (SIMT) architecture with thousands of simpler, slower threads executing simultaneously. In this model, each data element is processed by its own thread, allowing the same instruction to be executed simultaneously across numerous data streams. This approach is particularily well suited to deep learning, which uses neural networks composed of layers of numerous interconnected, identical neurons. Neural networks models rely on repetitive mathematical operations, such as matrix multiplications, convolutions, and activation functions, that can be executed concurrently across datasets. The parallelism in GPUs makes them the ideal platform over CPUs for deep learning and data processing tasks, required by autonomous driving systems.

1.1.4 NVIDIA Tesla V100 Architecture

For the purpose of this thesis, the NVIDIA Tesla V100 accelerator, which uses the Volta GV100 GPU architecture, was selected due to its availability and high-performance computing capabilities. At the core of its execution model is the warp, a group of 32 threads executing in SIMT fashion. Each warp is scheduled and dispatched within one of the 4 processing partitions of a Streaming Multiprocessor (SM). The warp

scheduler and warp dispatcher each handle 32 threads per clock cycle. Within each SM partition, there are Tensor Cores for deep learning, 64 bit-floating point (FP) cores, Load/Store (LD/ST) units, a register file, and SFUs for mathematical functions such as sine and square root. Additionally, each partition contains 16 Cuda cores, which can execute both FP 32 bit and integer (INT) 32 bit instructions, but not simultaneously. Each partition has its own L0 instruction cache, while all partitions share a L1 instruction and data cache. A SM can support up to 64 concurrent warps, allowing a maximum of $64 * 32 = 2048$ threads per SM. At a higher level, two SMs are grouped to form a Texture Processing Cluster (TPC). The Tesla V100 GPU consists of six GPCs, each containing seven TPCs, resulting in a total of 84 SMs across the chip. This hierarchical organization, from warps to SMs, TPCs, and GPCs, enables the Tesla V100 to efficiently handle massively parallel workloads, making it well-suited for high-performance computing and deel learning applications.

1.2 Motivation

In autonomous driving and other real time system tasks, GPUs are essential for data processing and machine learning; however, existing GPU scheduling techniques fail to meet the strict timing requirements of real-time applications. GPU tasks are traditional queued and scheduled based on availabilty to optimize for high throughput applications like graphics rendering and offline machine learning training. These tasks run to completion before switching tasks, meaning multiple tasks, such as the different modules in autonomous driving cannot efficiently share resources. Depending on the current workload for the GPU, execution times and latency can vary. These limitations make existing GPU scheduling mechanisms unsuitable for real-time applications like autonomous driving, where deadlines need to be met to ensure the safety of the system and passengers.

2 Objectives

The objective for this thesis is to develop, implement and evaluate a persistant thread with coroutine based scheduling approach for GPU threads to improve real-time performance by ensuring a predictable scheduling behavior. [Zhe+22] This requires analyzing the limitations of existing GPU scheduling techniques, which, due to their batch processing design, lack the ability to perform task switching. To adress this, a new scheduling mechanism will be designed that allows GPU execution to be halted by yielding, temporarily giving the processing resources to a different process. Resuming execution will involve saving and restoring the executing state for the previous process. Enabling GPUs to yield and switch between processes dynamically will improve responsiveness and reduce latency. The end completed solution will be a library that can be used with an API to automatically use coroutines on persistant threads.

2.0.1 Measurement and Evalutation

To measure the success of a coroutine based solution, the latency of new time sensitive task scheduling and deadline success will be evaluated. Unfortunately, this extra overhead, required to halt and restart cuda kernels, will result in overall weaker performance. Furthermore, the success will be measured on the implementability and ease of use to programm with and combine into Apollo, an open source autonomous driving platform.

3 Literature Review

3.1 GPU Partitioning

3.2 a

4 Proposed Approach

4.1 Section

The proposed solution will use a library with an API to implement coroutines on persistent threads. When a new task, from the CPU, is scheduled, the task must actively notify the GPU that a new task exists and the GPU needs to halt. This signaling will happen, by using GPU flags that can be set and evaluated or potentially through a maximum latency approach. The maximum latency approach, will maintain a maximum latency between when a task is scheduled onto the GPU and when the task needs to start, by periodically checking for new tasks. Next a mechanism to saving the thread state and resetting the thread state will be implemented. The thread state consists of the Program Counter (PC), thread register values, and memory state, which consists of thread private memory, warp and block level memory, and the persistant state. These values must be stored in the global memory and then resused to restore execution. Furthermore, to prevent deadlocks a synchronisation mechanism needs to be developed, to disallow deadlocks. If a GPU kernel uses a syncThreads() call and a coroutine attempts to yield it at that point, a deadlock will occur.

Abbreviations

GPU Graphics Processing Unit

AI artificial intelligence

MIMD multiple instruction multiple data

SIMT single instruction multiple thread

GPC Graphics Processsing Cluster

TPC Texture Processing Cluster

SM Streaming Multiprocessor

FP floating point

INT integer

LD/ST Load/Store

SFU special function units

LiDAR Light Detection and Ranging

CNN Convolutional Neural Network

PC Program Counter

SLAM Simultaneous Localization and Mapping

List of Figures

List of Tables

Bibliography

[Zhe+22] S. Zheng, Z. Zhou, X. Chen, D. Yan, C. Zhang, Y. Geng, Y. Gu, and K. Xu. "LuisaRender: A High-Performance Rendering Framework with Layered and Unified Interfaces on Stream Architectures." In: *ACM Trans. Graph.* 41.6 (Nov. 2022). ISSN: 0730-0301. DOI: 10.1145/3550454.3555463.