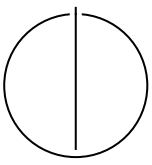
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Exploring GPU Programming Models for Autonomous Driving: From Coroutine Integration to Persistent Thread Optimization

Jaden Rotter



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

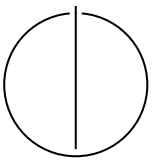
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Exploring GPU Programming Models for Autonomous Driving: From Coroutine Integration to Persistent Thread Optimization

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 systems place stringent demands on computational performance and predictability, yet GPUs, needed to process sensor data and run machine learning tasks, can not natively support time critical demands. The current CUDA GPU hardware scheduler, functioning as a black box, maximimizes throughput rather than deterministic execution, leading to unpredictable latency from resource contention, which poses a serious safety hazard. The proprietary nature of the hardware scheduler makes it difficult to enforce timing guarantees, prioritize critical tasks, and suspend resident kernels. In the absence of preemption or fine grained resource control, high priority workloads can be delayed by longer running, lower priority kernels. This thesis investigates GPU scheduling strategies tailored to real time systems, focusing on persistent threads as a mechanism to improve responsiveness, reduce latency variability, and ensure the timely execution of safety critical tasks in autonomous driving.

Early autonomous driving systems used a distributed architecture to ensure the timing guarantees of individual modules [1]. These systems break the driving task into the modules perception, localization, planning, and control, which together form a processing pipeline. This pipeline enables the vehicle to interpret its surroundings, determine its position within them, make decision, and execute corresponding actions. These modules were mapped to individual compute units, where every subsystem only ran tasks related to their modules. Among the benefits of this system design is that the load on the compute units is consistent. For example, the planning node only ever computes planning related tasks. In this manner, the distributed architecture allows for fine tuning the timing between modules to achieve a low latency responses from the hardware.

Today, as the hardware has become more performant, autonomous driving has become centralized, in order to lower costs and latency between modules. By centralizing the compute resources, the system uses a singular compute node, which manages all of the tasks simultaneously. This structural choice benefits the system design as well as the latency cost as all the tasks are colocated. The drawback to the centralized computing node is the contention between tasks, where the execution time is variable depending on the current execution queue.

To support this centralized architecture, modern compute nodes integrate both a CPU and a GPU, with the GPU playing a key role in meeting the performance and latency

requirements imposed by the data heavy processing and machine learning tasks in autonomous driving systems. Each of the individual modules involved rely on Firstly, perception gathers and interprets vast amounts of data from numerous sensors, using complex CNN networks to recognize pattern within the images [2]. Commercial use of CNN have been around for over 30 years [3], but were only widespread adopted due in part to advances in hardware GPU performance [4]. 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. The SLAM algorithm, path planning and control all require different deep networks, which are computationaly inefficient for CPUs. In order to capture real time updates in the surroundings, the autonomous driving system has to constantly compute each of these modules, which requires significant computational ressources that a CPU can not deliver with the necessary latency.

Due to their highly parallel architecture, GPUs excell over CPUs on the highly parallelisable computational workloads required by machine learning and processing tasks. GPUs were originally developed to accelerate graphics rendering, a task heavy in computations for each of the numerous pixel values consumer products have. Graphics rendering tasks differ from normal CPU workloads in the control overhead required. Typical interactive systems, for which CPU were designed, use workloads that include numerous branches, I/O, and sequential workflows. Achieving higher single-threaded performance means dedicating a "significant portion of transistors to non-computational tasks like branch prediction and caching", which GPUs can forgo in favor of increasing arithmetic intensity [5]. Due to the fundamental architecture differences, compute heavy tasks that do not require complex control logic are far faster on GPUs as opposed to CPUs.

Unfortunately, for the same architectural reason as why GPUs excell on compute heavy workloads, they can not guarantee execution latencies. CPU techniques used to achieve real time latency guarantees and meet deadlines can not be transferred to GPUs due to the architectural and programming differences. GPU kernels are queued and scheduled based on availability and executed until completion without interruption. If all the compute resources are saturated, incoming tasks become delayed, as there is no native thread context switching, something typical interactive or real time systems employ to achieve high responsiveness. The batch system used by GPUs leads to variable execution and latency times, dependent on the current and queued loads.

Real time systems, such as autonomous driving, are designed with strict timing

constraints in mind, to ensure predictable and deterministic behavior [6]. 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 system 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 safety both for the passengers and nearby individuals.

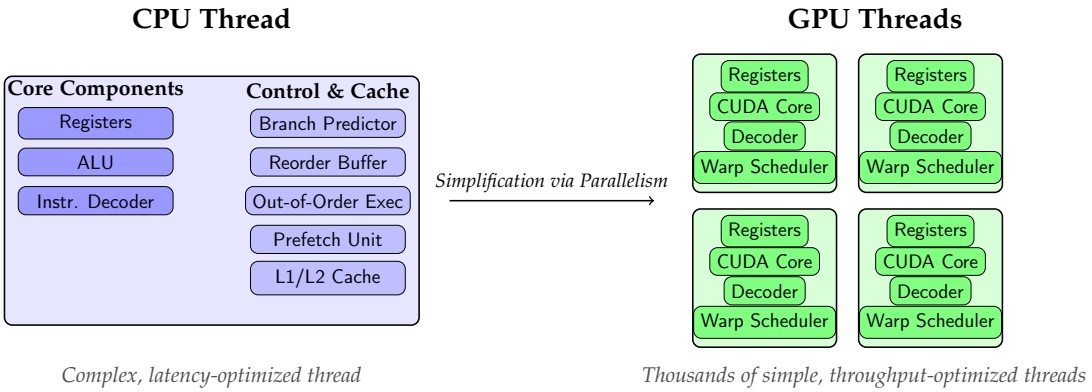


Figure 1.1: CPU vs GPU Thread Architecture

Although core components may be named differently, both CPU and GPU threads work fundamentally similarily with an instruction decoder, registers and an arithmetic unit. The differences arise when trying to maximize a single control flow. The CPU will prefetch instructions, reorder them to most effectively use the functional units and speculatively compute instructions based on a branch predictor. On the other hand, GPU threads can not execute instructions out-of-order, use only manual prefetching, and have a simple branch predictor that is far more conservative than in CPU's.

GPUs have revolutionized data processing and machine learning training and inference with their ability to handle massive amounts of data and execute simple, but highly-parallelized computations. Data processing systems, based on traditional CPUs, 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 tasks that require the repeated

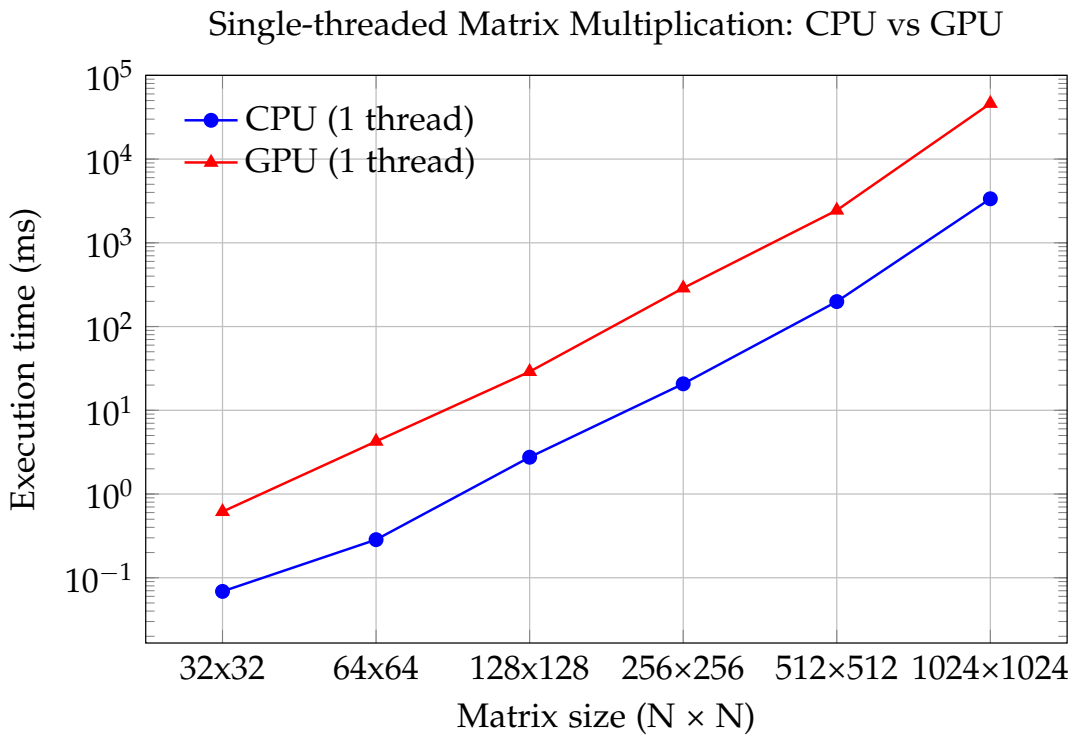


Figure 1.2: Single threaded Matrix Multiplication Execution

Matrix Size (n × n)	CPU Time (ms)	GPU Time (ms)	Speedup (GPU/CPU)
32 × 32	0.068705	0.615584	11.970438
64 × 64	0.285104	4.249685	15.554119
128 × 128	2.751817	28.923530	11.419879
256 × 256	20.706290	287.933700	13.906730
512 × 512	198.716200	2446.466000	12.323010
1024 × 1024	3356.762000	46097.410000	13.745850

Figure 1.3: Data Matrix

execution of singular instructions such as large-scale data processing workloads, CPUs are burdened by the overhead of the 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, but slower threads executing simultaneously. For example, the current architecture generally supports as many as 2048 threads in a single block, which can run on a single Streaming Multiprocessor (SM), scaling the number of concurrent threads by the number of SMs, by the number of SMs, 80 on the newer models. The parallelism provided by the SIMT model allows a far higher throughput that outperforms CPUs. Here, each data element is processed by its own thread, allowing the same instruction to be executed simultaneously across numerous data streams.

Real time systems, which rely extensively on machine learning tasks and data processing tasks, Higher data processing performance is vital to autonomous systems, which rely extensively on machine learning tasks and sensor data processing. In autonomous systems a multitude of modules require deep learning, a machine learning technique, to automatically extract patterns, such as computer vision, and make decisions [7]. Deep learning, inspired by the structure of the human brain, is composed of layers of numerous interconnected, identical nodes called neural networks. Neural network models rely on mathematical operations between different neighboring layers to perform inference, essentially the prediction making or recognition process. The layers are represented as matrices, which then get multiplied and convoluted with one another and are used by specialized functions, called activation functions, to introduce non-linearity. With very large neural networks with thousands or millions of nodes, the inference computation is repetitive and slow. The capability to execute these repeptitive workloads concurrently across different dataset makes GPUs fundamental to performance in tasks using complex neural networks. Furthermore, GPUs are necessary to process the data rate produced by high-bandwidth, high-frequency sensors used in autonomous driving. The parallelism in GPUs makes them the ideal platform over CPUs for deep learning and data processing tasks, tasks fundamental for autonomous driving systems.

Real time systems, such as autonomous driving, are designed with strict timing constraints in mind, to ensure predictable and deterministic behavior [6]. 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 system 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 safety both for the passengers and nearby individuals.

1.0.1 Necessesity of GPUs in Autonomous Driving

1.0.2 Autonomous Driving as a Real Time System

1.1 Background

1.1.1 Asynchronous Programming and Coroutines

Asynchronous programming is a method of programming a system to handle tasks concurrently instead of sequentially. Typically used in conjunction with tasks that delay or have high wait-times, such as I/O heavy jobs, asynchronous programming reduces overall execution time by more efficiently using processing ressources. For example, while waiting for I/O heavy input like sensor I/O, asynchronous code lets other tasks execute in the meantime, before returning when the data arrives. For real-time systems, asynchronous programming additionally uses the intermittant execution model to enforce determinism. By allowing the GPUs to switch between concurrent tasks, hard deadlines can be immediately enforced without delay.

Coroutines, an implementation of asynchronous programming, uses suspendable functions to halt execution. Suspendable functions are implemented by capturing the current context, know as the continuation, of the currently running thread and save the data to be run later [8]. After being saved, a new process can take over execution, without interrupting or overwritting the state of the previous process. Once the intermittant process or higher priority process has finished execution, the original task can continue executing by restoring the process context, which was previously saved. Capturing the continuation of a function allows the resumption of the program to be strategically deferred.

1.1.2 NVIDIA Tesla V100 Architecture

For the purpose of this thesis, the NVIDIA Tesla V100 accelerator, which uses the Volta GV100 Graphics Processing Unit (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 [9]. Each warp is scheduled and dispatched within one of the 4 processing partitions of a 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.

2 Objectives

The objective for this thesis is to develop, implement and evaluate a persistant thread with coroutine based scheduling approach for GPU threads in Apollo[1], an open source autonomous driving platform, to improve real-time safety and determinism by ensuring a predictable scheduling behavior. 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, the new coroutine based scheduling mechanism LuisaCompute-coroutine[2] will be employed. The Luisa coroutine scheduling was designed for graphics rendering tasks, around which their domain specific language (DSL) is written. Therefore, the scheduler and its DSL will need to be adapted to fit the requirements of the autonomous driving domain, which prioritizes responsiveness, fault tolerance, and strict timing guarantees. By integrating this feature into Apollo, the GPU will be able to use coroutines for more flexible and efficient task management, ultimately enhancing the reliability and safety of the real-time autonomous driving platform.

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 in comparison to the old system. Furthermore, the implementability and effectiveness of this application will be analyzed through practical experiments and benchmarks, focusing on how well the system maintains deterministic behavior under varying workloads and task complexity.

[1] Apollo is an open-source project developed by Baidu. For more information, visit the GitHub repository: `https://github.com/ApolloAuto/apollo`

[2] [8] and the accompanying source code `https://github.com/LuisaGroup/LuisaCompute-coroutine/tree/next`

3 Literature Review

3.1 GPU Coroutines for Flexible Splitting and Scheduling of Rendering Tasks

Previous work, on which this thesis is based, offers a new method in rendering task management to demonstrate a flexible, fine-grained scheduling mechanism on GPUs [8]. Previously discussed in the background and introduction, this paper focuses on splitting tasks into discrete segments that can be suspended and resumed as needed on GPUs. Specifically, large, monolithic kernels could be rewritten into smaller tasks using coroutines. This capability, important for tasks with downtimes, reduced the latency associated with waiting for an entire batch of rendering computation to complete and maximized the utilization of the GPU's parallel processing capabilities.

In this paper, a flexible DSL was presented, which allowed the writing in a mega-kernel fashion with $suspend statements to define suspention marks, needed for the coroutines. These points, allow the creation of different discrete segments that can be suspended and resumed as needed. The DSL has support for a multitude of languages, specifically C# and CUDA. With minimal changes to the original code, the new DSL can be included, which becomes dynamicaly recorded and translated to an intermediate representation. From the intermediate representation, after a set of compiler transformations and analysis the state frame can be extracted, where a scheduler can then define the execution and manage the state frames. Important here, is that the scheduler can be chosen or rewritten, which will be used for Apollo. Ultimately, through the design of an easy-to-use and easily accessible library, this work can be implemented into autonomous driving systems.

4 Proposed Approach

The proposed solution leverages LuisaCompute-coroutines to introduce a coroutine-based scheduling mechanism into Apollo. The approach begins by integrating the LuisaCompute-coroutine framework to manage GPU threads more flexibly, allowing rendering and real-time computation tasks to be suspended, resumed, and interleaved in a deterministic manner. This integration aims to adapt the coroutine scheduler—originally designed for graphics rendering tasks—to meet the real-time and safety-critical demands of autonomous driving. By doing so, the system can address the limitations of Apollo's existing batch-based scheduling, facilitating more dynamic task management and offering significant improvements in responsiveness and predictability.

The implementation will start with a test integration into a single, self-contained module within Apollo. This initial step serves as a trial, focusing on adapting the LuisaCompute-coroutine scheduling to manage GPU threads effectively for one critical subsystem. Once the viability and benefits of this approach are validated—through detailed performance benchmarks and safety evaluations—the solution will be iteratively expanded to encompass additional modules across the platform. This staged rollout not only minimizes disruption to Apollo's existing architecture but also provides a controlled environment to fine-tune the integration, setting the stage for broader adoption of coroutine-based scheduling across the entire system.

Abbreviations

GPU Graphics Processing Unit

CPU central processing unit

DSL domain specific language

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

CNN Convolutional Neural Network

SLAM Simultaneous Localization and Mapping

List of Figures

List of Tables

Bibliography

[1] K. Jo, J. Kim, D. Kim, C. Jang, and M. Sunwoo, "Development of autonomous car—part i: Distributed system architecture and development process," *IEEE Transactions on Industrial Electronics*, vol. 61, no. 12, pp. 7131–7140, 2014. DOI: 10.1109/TIE.2014.2321342.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *ImageNet classification with deep convolutional neural networks*, https://proceedings.neurips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf, Accessed: 2025-7-17.

[3] L. D. Jackel, D. Sharman, C. E. Stenard, B. I. Strom, and D. Zuckert, "Optical character recognition for self-service banking," *AT&T Technical Journal*, 1995.

[4] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," Apr. 2016. arXiv: 1604.07316 [cs.CV].

[5] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," en, *Comput. Graph. Forum*, vol. 26, no. 1, pp. 80–113, Mar. 2007.

[6] J. Sun, K. Duan, X. Li, N. Guan, Z. Guo, Q. Deng, and G. Tan, "Real-time scheduling of autonomous driving system with guaranteed timing correctness," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2023, pp. 185–197. DOI: 10.1109/RTAS58335.2023.00022.

[7] W. Jeon, G. Ko, J. Lee, H. Lee, D. Ha, and W. W. Ro, "Chapter six - deep learning with gpus," in *Hardware Accelerator Systems for Artificial Intelligence and Machine Learning*, ser. Advances in Computers, S. Kim and G. C. Deka, Eds., vol. 122, Elsevier, 2021, pp. 167–215. DOI: https://doi.org/10.1016/bs.adcom.2020.11.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0065245820300905.

[8] 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," *ACM Trans. Graph.*, vol. 41, no. 6, Nov. 2022,

ISSN: 0730-0301. DOI: 10.1145/3550454.3555463. [Online]. Available: https://doi.org/10.1145/3550454.3555463.

[9] N. Corporation, "Nvidia tesla v100 gpu architecture," NVIDIA Corporation, Tech. Rep., 2017, Accessed: 2025-04-09. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.