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]. The distributed architecture processes the individual driving tasks into the modules perception, localization, planning, and control, which together form a processing pipeline. The stages of the pipeline enable the vehicle to interpret its surroundings, determine its position within them, make decision, and execute corresponding actions. In this architectural design, each module is mapped to an individual compute unit, where every unit only runs 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.

The original distributed system design was necessary in part due to the vast amount of computations and processing required by the autonomous driving system. In particular, the core modules of an autonomous driving system each require complex neural nets in order to allow the vehicle to function autonomously. Currently, CPUs alone are not performant enough to react to the real time constraints imposed by autonomous vehicles. These constraints are necessary in order to ensure the safety of the passengers, system and enviornment. To meet these constraints, autonomous vehicles uses a heterogenous computing architecture with specialized chips, such as the NVIDIA GPUs, to support the heavy processing tasks required by the system. The GPUs themselves are not designed to be used in real time systems and using standard real time system algorithms for these chips is infeasible due to the different programming models.

Unfortunately, for the same architectural reason as why GPUs excell on compute heavy workloads, they can not guarantee execution latencies. The GPUs are implemented on a batch system algorithm, where throughput is prioritized above all else. Unlike real time systems typically implemented on CPUs, GPU kernels do not natively support interruption to allow higher priority tasks to execute [2]. The GPU kernels are queued and scheduled based on availability and executed until completion without interruption. By prioritizing throughput over responsiveness and latency, the GPUs may become contented between various different tasks, which leads to variable execution and latency times, dependent on the current and queued loads. Variable latencies are unacceptable in real time systems where strict execution deadlines must be preserved in order to react to the changing enviornment in time. This paper proposes a method to allow the GPU to be partitioned, reducing kernel launch latency, and enable the integration of GPUs into real time systems.

2 Background

2.1 Integration of GPUs in Autonomous Driving Systems

Autonomous driving system require GPUs for the computational acceleration they provide to the many parallel machine learning models that interpret sensor data, make decisions and ensure safe navigation in real time. From perception to planning and control, each stage of the autonomous driving pipeline relies on models that must operate within tight latency constraints. These models include convolutional neural networks (CNNs) for image and object recognition, recurrent or transformer-based architectures for temporal sensor fusion and prediction, and reinforcement learning agents or decision trees for behavioral planning [3]. The sheer diversity and complexity of these tasks require a hardware platform that can execute thousands of operations in parallel in order to achieve the throughput necessary [4].

GPUs are particularly well suited to these workloads because of their massively parallel architecture and high memory bandwidth, which perfectly meet the demands of deep learning tasks. Unlike CPUs, which optimize for sequential instruction execution and low latency branching, GPUs are designed to handle large batches of matrix and tensor operations simultaneously. This makes them ideal for real time inference of deep neural networks. Furthermore, modern GPU architectures provide specialized cores, such as tensor cores in NVIDIA GPUs, that are explicitly optimized for mixed precision matrix multiplication, which is a core operation in most machine learning models. By offloading intensive compute tasks to the GPU, autonomous systems can maintain low latency and high accuracy, both of which are crucial for safety and performance in dynamic driving environments.

2.2 GPU versus CPU Architecture

GPUs are capable of delivering this vast increase in throughput over CPUs as measured by GFLOPS, despite the lower clock rate, by simplifying the thread context in order to afford greater parallelism. They were originally developed to accelerate graphics rendering, a task heavy in parallizable computations, which require only a very simple control overhead and as such have adapted the architecture to support as many possible different threads. Typical workloads designed for CPU are based on sequential workloads, such as human input or complex logic, which requires complex thread overhead to speed up branches and I/O, through prefetching, branch prediction, and out of order execution. These CPUs achieve higher single threaded performance by 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]. Consider the following graphic Figure 2.1, which highlights the difference in thread complexity.

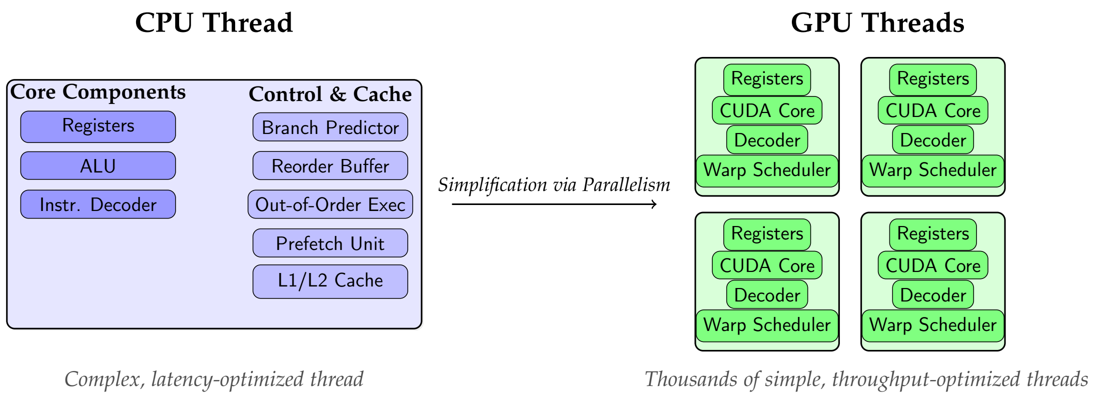

Figure 2.1: CPU vs GPU Thread Architecture

Although core components are 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 the CPU's predictor. Furthermore, the GPU amortisizes the cost of managing an instruction stream accross multiple threads, which execute the same instructions at the same time versus the CPU execution model which The additional complex logic involved allows single threaded CPU applications to far outperform single threaded GPU applications, as seen by the following comparison of single threaded matrix multiplication in Figure 2.2 and Figure 2.3.

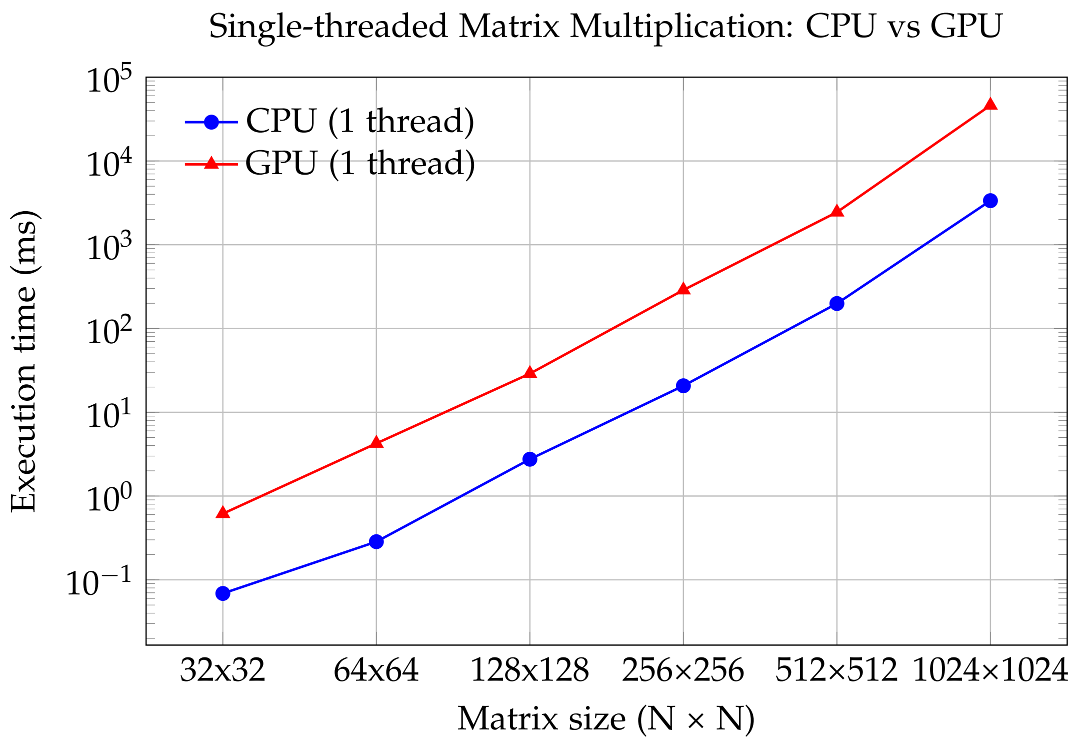

Figure 2.2: Single threaded Matrix Multiplication Execution between CPUs and GPUs averaged over 10 executions

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 2.3: Data Matrix from Figure 2.2

As seen in Figure 2.2 and Figure 2.3 applications that fail to utilize the concurrancy of GPUs, either due to programmatical errors or a lack of parallelism in the task, will struggle to achieve high performance. For each of the matrices tested, through prefetching, a higher clock rate, and branch prediction, the CPU is on average around 13 times faster than the GPU running the exact same algorithm. Following these results, the GPU should only be used in place of the CPU, when the application is well tuned to the programming model and a scheduler must respect this difference. If a GPU scheduler is written without regard for these differences, it will may not be able to achieve the desired performance benefit. This fundamental architectural and programming style for GPUs must be understood in order to maximize the throughput.

2.2.1 GPU Hardware Architecture based on the Tesla V100 GPU

For the purposes of programming and scheduling tasks onto the Tesla V100 GPU[1], the GPU appears as an array of independent highly parallelized processors, called SMs. The SMs are grouped into specific TPCs, which are then further grouped into GPCs, but the specific mapping of tasks to SM, TPC, and GPC is determined by the proprietary hardware scheduler, the GigaThread Engine. The exact workings and scheduling methods of the GigaThread Engine are not publicly available, but this module maps CTAs, groups of threads executing the same instruction code, to the individual SMs based a multitude of factors: hardware resources, parallelism, priorities, and dependencies. Similarily, the global memory and L2 cache utilization are determined by the hardware and transparent to the programmer. After the CTA gets mapped to the specific SM, the device code then executes till completion. Each SM manages its scheduled CTAs through its own execution pipelines, register files, shared memory and scheduling units that function independently from one another. For SMs to communicate with one another, they must use either the global on-chip device HBM2 or through the global L2 cache which is shared and coherent across all SMs. Although these memory accesses allows individual SMs to communicate with each other, accesses require hundreds of cycles, which introduce further latencies when compared to local SM L1 memory caches. Ideally, the SMs execute independently of one another and cumulate answers in global memory, skipping the high memory latency accesses of coordinating synchronous work.

Applications are scheduled to the SM by the GigaThread Engine consisting of a CTA, or block of threads executing the same instruction code, which then get subdivided into warps to be executed across the SM's execution units. On the GPU, the smallest unit of execution is the Warp, a group of 32 threads that executes instructions in lockstep. Warps always consist of 32 threads, even if the CTA is not divisible by 32 and cannot be fully partitioned across the warps. The lockstep execution pattern of warps, enforce that each thread within the warp executes the same instruction, even if several threads are inactive. As a consequence, control logic that forces divergent threads significantly

[1]For the purpose of this thesis, the NVIDIA Tesla V100 GPU chip, which uses the Volta GV100 architecture, was selected due to its availability and high performance computing capabilities.

slows execution and overutilizes CUDA cores, as the individual threads are forced to execute sequentially.

The Tesla V100 GPU SM architecture, self contains an entire execution pipeline within each of its 4 processing partitions, which collectively share an L1 instruction cache as well as an L1 data and shared memory cache. As CTAs are distributed across multiple Warps, these collective L1 caches allow the instruction memory and shared memory to be stored across different Warps within the same CTA. The main components of each processing partition are a L0 instruction cache, warp scheduler, dispatch unit, execution units. Every clock cycle, the Warp scheduler schedules a singular Warp of 32 threads, which get passed to the Dispatch unit. The dispatch unit then dispatches a new instruction to the Warp every clock cycle. As for any given instruction there are not enough execution units of the same type, the instructions get queued onto the execution units. Depending on the current queue and any delays, such as global memory accesses or dependencies, the Warp scheduler will interweave different Warps together onto the Dispatch Unit to hide latencies. Within each Streaming Multiprocessor (SM) processing partition's execution units, 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. An in depth view of the processing partition's architecture is provided in Figure 2.4.

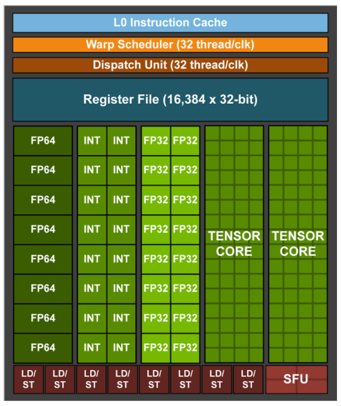
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU

Figure 2.4: Architecture from the whitepages: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

The programmer is limited by the hardware constraints of each SM and the total number of SM. On the Tesla V100 architecture, there are 80 SM, each supporting up to 64 concurrent warps, allowing a maximum of $64 * 32 = 2048$ threads per SM. In total, that leaves a maximum of $80 * 2048 = 163,840$ total threads. For comparison, the current CPU I have, an Intel Tiger Lake (i5-1135G7), has 4 cores, with 2 hardware threads per core supporting a maximum of 8 hardware threads. Even server chips such as the Intel Xeon Gold (6148) only supports 20 hardware threads. Although the GPU oversubscribes the number of Warps and threads, versus the total number of execution units, at a minimum it can still execute $4 * 32 * 80 = 10840$ threads at a time. These hardware enforced limits must observed when programming the GPU.

2.3 GPU Programming using the CUDA API

NVIDIA has provided an underlying CUDA API to allow programmers to run tasks on the GPU in a heterogenous computing architecture. The GPU, in this context device, memory is standalone from the CPU, in this context host, memory. However, the GPU execution is dependent on tasks received from the CPU. Given that the device memory is seperate from the host memory, typical GPU workloads work by first allocating memory on the device, copying the memory over, executing the program, copying the memory back, and then freeing the device memory. Typically, the CPU uses the GPU, by copying the memory over onto the chip, launching the kernel application, and then copying the result from the GPU back to the CPU. Although the CUDA API manages the actual underlying steps, the API allows the programmer to specifically program and optimize the GPU for their specific task.

2.3.1 Memory and Bandwidth Considerations

When transfering data from the host, the CUDA API does not have access to the CPU's disk memory and requires the memory to be pinned to the CPU's RAM. While the CUDA runtime can perform this pinning autonomatically, host arrays allocated directly in pinned memory using `cudaMallocHost()` or `cudaHostAlloc()` skip the added step of pinning memory and enable faster transfers. Particularily in applications that are bandwidth bottlenecked, this added transfer latency is significant. If the pinned memory is too large; however, it restricts the memory availability for the programs currently running on the CPU, which may degrade performance by forcing the swapping of memory to disk storage.

In CUDA, understanding how memory is transferred and managed across the host and the device is crucial for optimizing performance. For the programmer, the device memory is partitioned into main memory, the VRAM, with a size of 16 GB on the Tesla V100 architecture, as well as on chip memory. The programmer can decide between three different models of memory management `__constant__`, `__device__`, and `__shared__` memory. Both constant and device memory are allocated to the global memory, with constant memory only being writeable by the CPU and allowing faster access times due to the reduced coherency required. The shared memory is shared among all warps and threads of a given SM in the L1 data and shared memory cache. This memory is far more performant than device memory, but limited in size, due to its location on chip.

2.3.2 Kernel Launches

The task of launching and running device code begins from a kernel launch, which passes the function, its parameters, pointers, and the grid and block dimensions to the GPU. Each kernel launch specifies how work is diveded among thread blocks and individual threads. Every block is mapped to a SM injectively and is constrained by that SM's hardware resources including the number of threads, registers, available warps, and shared memory. If there are no available SM's to meet these requirements the kernel launch will fail. Should the CTA or thread block not fully saturate the hardware resources, additional blocks may be scheduled to the same SM. Each thread and block is assigned unique identifiers, threadIdx and blockIdx, that allow them to determine their position in the execution grid. These identifiers are crucial for structuring parallel computations and can be used to optimize memory access patterns. For instance, when threads in a warp access consecutive memory locations, the memory accesses can be coalesced into a single transaction, significantly improving memory throughput

The exact syntax is seen below,

2.3.3 CUDA Streams

CUDA API calls are queued to the GPU using cuda streams, which enforce the execution order of tasks. `cudaStream_t` defines a command queue for the GPU, which is similar to a Linux file pointer in that it returns an index referring to the specific allocated stream. Each stream allows the queuing of operations such as kernel launches, memory copies, and memory set operations. Commands submitted to the same stream are executed sequentially in the order they were issued, ensuring deterministic behavior within that stream. Multiple streams, however, can run concurrently, enabling overlapping execution of kernels and memory operations to maximize GPU utilization and improve overall performance. By carefully managing streams, developers can optimize task parallelism and resource usage on the GPU.

The Tesla V100 GPU has two seperate hardware copy engines for copying data from the host to the device and back. The copy engines support the transfer in both directions, with one engine specifically being allocated for the unidirectional D2H transfer and the other for H2D. Using only one stream for multiple kernels fails to maximize the device memory bandwidth. For example, consider the launch of two independent kernels, kernel A and kernel B, each

Executing both kernls in the same stream would force kernel B to wait for kernel A to finish before being able to copy memory to the device using the H2D

Using different streams, the streams non deterministically execute with respect to one another, except for the default stream.

2.3.4 Synchronization between Streams

Using multiple streams enables the programmer greater controll over the execution and perform optimization suited to the hardware.

By default everything goes into stream 0, but operations defined in different streams may execute concurrently.

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 [6]. 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 [7]. 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.

2.3.5 Necessesity of GPUs in Autonomous Driving

2.3.6 Autonomous Driving as a Real Time System

2.4 Background

2.4.1 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.

2.4.2 NVIDIA Tesla V100 Architecture

Real time systems, such as autonomous driving, are designed with strict timing constraints in mind, to ensure predictable and deterministic behavior [7]. 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.

3 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.

3.0.1 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`

4 Literature Review

4.1 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 Graphics Processing Unit (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.

5 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

GPC Graphics Processsing Cluster

TPC Texture Processing Cluster

SM Streaming Multiprocessor

FP floating point

LD/ST Load/Store

SFU special function units

CTA Cooperative Thread Array

HBM2 High Bandwidth Memory

VRAM Video Random Access Memory

RAM Random Access Memory

D2H Device to Host

H2D Host to Device

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] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron, "Enabling task parallelism in the cuda scheduler," in *Workshop on Programming Models for Emerging Architectures*, 2009, pp. 69–76.

[3] 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.

[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] 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`.

[7] 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`.

[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`.