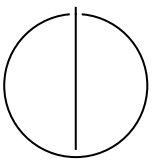
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

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

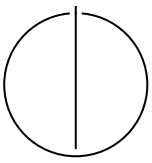
TECHNISCHE UNIVERSITÄT MÜNCHEN

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 GPU coroutines and 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 decisions, and execute corresponding actions. In this architectural design, each module is mapped to an individual compute unit, resolving any resource contention issues between independent modules. This system design ensures that the load on the compute units is consistent and responsive to the individual modules. For example, the planning node only ever computes the next, most time critical, planning tasks. In this manner, the distributed architecture allows for fine tuning the timing between modules to achieve low latency responses from the hardware.

Although the system safety was ensured by distributing numerous compute resources, this approach is wasteful and expensive, especially as recent hardware advances in GPUs allow massive cost reduction by centralizing modules onto one processing node. In addition to the cost and design savings, the intermodule latency is reduced as the results and inputs of different modules are colocated. This new singular compute node, responsible for all tasks simultaneously, needs to ensure that the execution order respects real time deadlines for the safety of system, passenger and enviornment.

Despite the centralized architecture being more performant, using a singular compute node risks hardware contention, which can lead to execution latencies. Variable execution latencies depending on the execution queue comprimise the system, if GPU tasks can not be preempted to ensure the immediate execution of time critical tasks.

Unlike GPUs, CPUs already natively support numerous real time systems at both the OS and programming model levels; however, they currently lack the computational performance needed to meet the strict latency requirements of autonomous driving systems, for which GPUs are explicitly required. 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. To meet these constraints, autonomous vehicles uses a heterogenous computing architecture with a main CPU, which offloads machine learning and processing work to a specialized processor, such as the NVIDIA GPUs. 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 under task contention. NVIDIA 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 Real Time Systems

Real time systems, such as autonomous driving, are designed with strict timing constraints in mind, to ensure predictable and deterministic behavior [3]. 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.

2.2 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 [4]. 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 [5].

GPUs are particularily 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.

2.3 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 [6]. Consider the following graphic Figure 2.1, which highlights the difference in thread complexity.

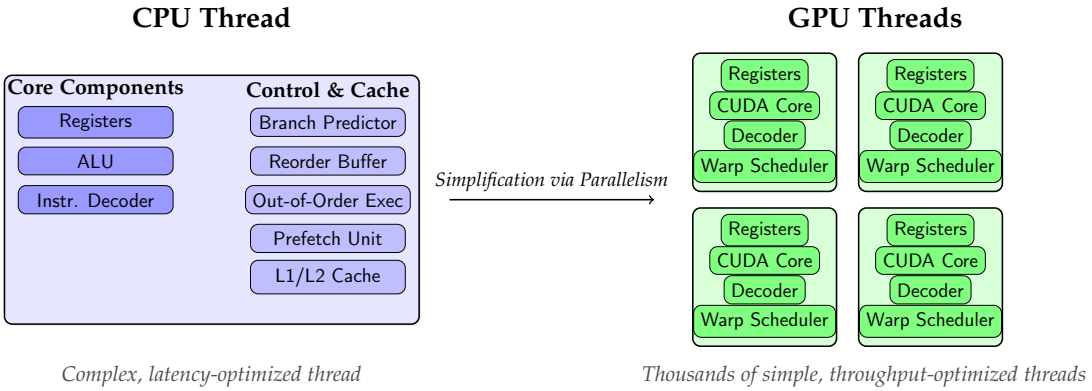


Figure 2.1: CPU vs GPU Thread Architecture

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.

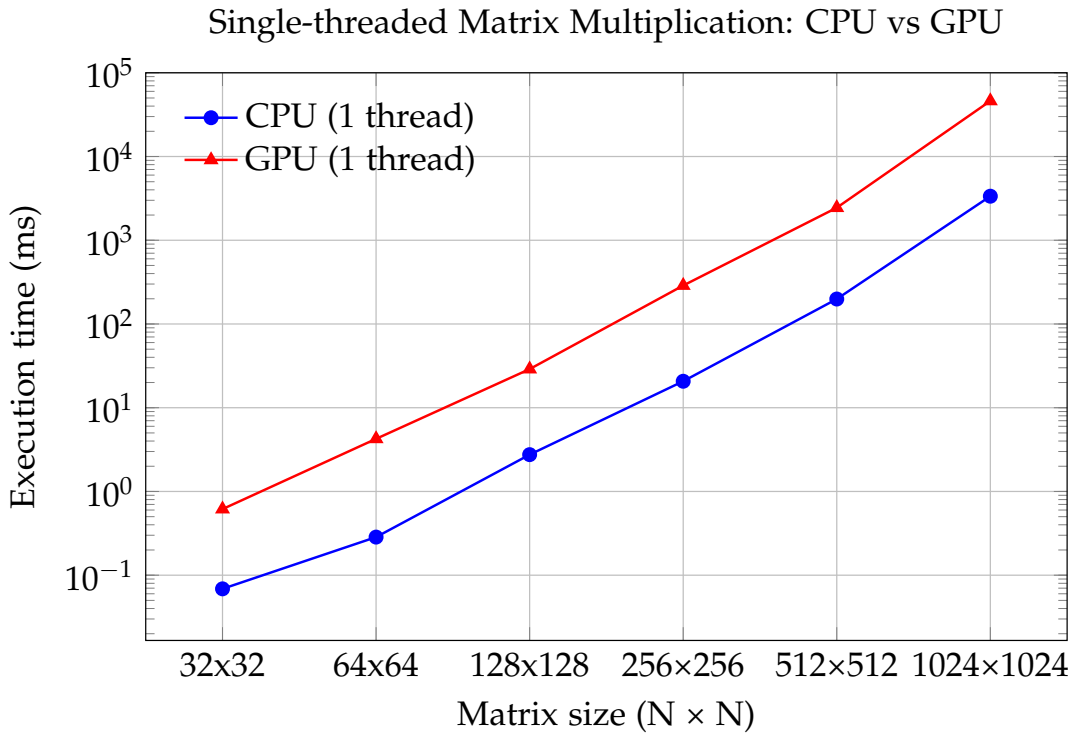


Figure 2.2: Single threaded Matrix Multiplication Execution between CPUs and GPUs averaged over 10 executions

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.

2.3.1 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

[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.

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 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.

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.

2.4 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 acts as a standalone processor with its own memory, which is not shared with the host system or CPU's main 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. 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.

Consider the following example program, which allocates device memory and launches a kernel consisting of one block and 32 threads.

```
__global__ void increment(float *x) {
    x[threadIdx.x] += 1.0f;
}

int main() {
    const int N = 1024;
    float h_x[N];
    for (int i = 0; i < N; ++i)
        h_x[i] = i * 1.0f;

    float *d_x;
    cudaMalloc((void**)&d_x, N * sizeof(float));
    cudaMemcpy(d_x, h_x, N * sizeof(float), cudaMemcpyHostToDevice);

    increment<<<1, 32>>>(d_x);

    cudaMemcpy(h_x, d_x, N * sizeof(float), cudaMemcpyDeviceToHost);
    cudaFree(d_x);
    return 0;
}
```

Listing 2.1: Simple CUDA Kernel

The codeblock above depicts the launching and execution of a simple GPU kernel that demonstrates the memory allocation scheme used for executing kernel code. The kernel itself is the execution of the GPU device program denoted by `__global__` function, while the `<<<_, _>>>` syntax enables the programmer to specifically partition their execution tasks across waiting threads. The first value in the `<<<1,32>>>` determines the block dimensions, which are either given as an array or a 3 dimensional tensor. Similarily, the second value determines the thread dimensions in the same format as the block dimension. In particular, this code allocates a singular block with an array of 32 threads, which completely saturates a singular warp. These dimensional vectors allow the different threads to maintain lockstep execution, while processing different values of the same array, as seen by using the dimensional properties assigned to the individual threads by the runtime system.

The main function, executed by the CPU or host, initializes the parameters, executes the kernel and then copies the memory back. The host array, `h_x` is allocated to the stack, which exists only in CPU memory, which needs to be passed to the GPU. Passing the array by value, something common in C++ code, seems at first the most simple; however, poses two seperate issues. Firstly, when passing arrays as parameters, they decay to pointers, which CUDA forbids, as the pointer passed to the device does not

have any meaning. Secondly, if the array were wrapped in a struct and passed to the function to circumvent the first issue, the array would be allocated to every single thread independently. In the example above, the array would be allocated 32 times, each independent from one another, taking up further memory bandwidth and both on chip and global device memory. In this case, each individual thread, would get the array passed by value, leading to a total 32 threads * 1024 floats * 4 bytes per float or 128KiB. Instead, them memory is allocated in the device memory, transferred once and each thread recieves only the device pointer `d_x`, which can be used to copy the results back to the host.

2.4.1 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.

2.4.2 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.

2.4.3 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 on the same cuda stream. Both A and B, allocate and copy memory onto the device, schedule their kernels and then copy the results back. Regardless of the ordering of API calls, both kernels can not run simultaneously or use both H2D and D2H copy engines simultaneously.

To best utilize the hardware and ensure correct results, a different stream need to be used for each synchronous kernel launch. Each kernel must remain in the same stream as the memory copys related to that stream in order to ensure the arguments used by the kernel are accurate and that the results are not prematurely returned. By placing each kernel in its own stream, the CUDA API, depending on the time and specific situation run multiple kernels, run kernels and memory transfers, or perform both D2H and H2D API calls simultaneously.

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

4 Luisa Coroutines

The first proposed approach to implementing a GPU scheduler for autonomous driving focuses on integrating the LuisaCompute coroutine platform into Apollo. The goal is to enable GPU coroutines within Apollo, allowing the autonomous driving framework to suspend and later resume GPU kernel execution. This capability would allow the scheduler to better enforce bounded response latencies by directly scheduling the highest priority tasks at the appropriate time, rather than waiting for current GPU workloads to complete.

4.1 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 enable cooperative multitasking between routines, where instead of a thread or process level context between different tasks, they maintain only a function level context switch that allows fast task switching. 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 [7]. 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.

4.2 CPU Coroutines

Unlike complicated threads or processes, the handling of coroutines allows for simple context switching between different functions. On x86 architecture, the CPU when calling a new function will push the next instruction address to the stack and jump to the new program, with the variables being passed across registers. x86 convention divides the number of available registers into volatile, caller saved regsiters and non volatile, callee saved registers. At the end of the function call, the callee saved registers must remain the same. Therefore the minimum context of a coroutine consists of all the callee saved registers as these are the registers that must be reproduced. Consider the following CPU coroutine code from Apollo.

```
ctx_swap:
    pushq %rdi
    pushq %r12
    pushq %r13
    pushq %r14
    pushq %r15
    pushq %rbx
    pushq %rbp
    movq %rsp, (%rdi)

    movq (%rsi), %rsp
    popq %rbp
    popq %rbx
    popq %r15
    popq %r14
    popq %r13
    popq %r12
    popq %rdi
    ret
```

Listing 4.1: CPU Coroutine

This function ctx_swap receives two paramaters, %rdi and %rsp, both being addresses to store and retrieve the coroutine continuations. The first section of ctx_swap saves the current continuation, by pushing all the callee saved registers onto the current stack and saving the stack pointer to the memory location pointed to by the address in %rdi. Then the stack pointer is updated with the new coroutine's stack pointer and all teh callee saved registers are loaded in the reverse order as they were saved of the new program. In comparison to the context switching involved by process or thread switching, this executes very quickly and allows for switching the thread context to multiple different applications.

4.3 GPU Coroutines

In contrast to CPU coroutines, the GPU coroutines are more complex due to the large number of concurrent threads and concurrent warps that are executing and the specifics of GPU programming, especially in regards to the stack management. GPUs launch kernels that can not be interrupted or yield their resources through the kernel's lifetime. The kernel function will staturate the number of warps and threads that were allocated to it until termination. After each kernel terminates the state afterwards is not preserved for the next incoming kernel. Furthermore, unlike CPU function calls where the instruction pointer is pushed onto the stack and the program then jumps to the new function, GPUs save stack pressure by aggressively inlining function calls. The advantage of inlining function calls is that there is no overhead when beginning a new function as that function code is already readily available. Unfortunately for tasks dependent on deep call stacks such as recursion, this leads to an exponentially large instruction memory. The underlying PTX code allows for recursion using the `nvcc -rdc=true` flag.

Due to the management of call stacks and batch execution of kernels on GPUs, the GPU requires a persistent kernel implementation with a manual implementation of coroutine state in order to save and resume their continuation. Due to the nature of kernels executing until completion, the GPU coroutine scheduler needs to be based on persistent threads, which schedule coroutines onto their threads. These coroutines themselves need to have suspension points to manually give control back to the scheduler in order to execute the next task. Saving every register value for every thread across every coroutine is to computationaly expensive, so the coroutine contexts need to be managed locally and saved in global memory to free up limited SM resources for new tasks.

The implementation developed in LuisaCompute-Coroutine enables GPU coroutines by providing a coroutine based API that acts as a **JIT!** compiler for generating GPU kernels at runtime. LuisaCompute offers a DSL embedded in C++, allowing programmers to explicitly define coroutine suspension points using standard C++20 coroutine syntax, such as co_await. These coroutine constructs are not executed immediately but are instead interpreted symbolically into an **AST!**.

At runtime, LuisaCompute builds a symbolic representation of the kernel's control and data flow in an abstract syntax tree (AST), through operator overloading and expression tracking. This symbolic trace is then lowered into an intermediate representation (IR), which encodes the coroutine as a state machine, capturing both the control flow and the coroutine's execution context. The resulting IR is compiled into GPU code, such as PTX for CUDA, using LuisaCompute's JIT backend. Once compiled, these coroutine-based kernels are dispatched and executed on persistent GPU threads, which

maintain their state across kernel invocations and facilitate efficient asynchronous execution and task switching on the GPU.

As part of this work, I initially explored the possibility of integrating LuisaCompute coroutines into the Apollo autonomous driving platform. However, due to the lack of documentation and my limited understanding of both Apollo and LuisaCompute in both the implementation of tasks into Apollo and the underlying abstract syntax tree (AST) and intermediate representations (IRs) used in LuisaCompute's JIT compilation system, I struggled with dependency issues and was ultimately unable to complete the integration. Rather than continuing down this path, I decided to simplify the problem and shift focus toward developing a custom implementation of persistent GPU threads, which still reduce the overhead involved with launching GPU kernels.

5 Persistent Threads

Persistent GPU threads allow the hardware resources to be partitioned and reduce kernel execution and scheduling overhead. While the persistent GPU threads do not allow for asynchronous programming such as with coroutines, still help ensure runtime guarantees and support execution preemption. Furthermore, with the entire GPU partitioned across persistent threads, the hardware resources can be manually assigned according to the SM number.

To implement these persistent GPU threads, I sought to find an open source implementation; however, I only found the LightKer implementation, used to profile persistent threads. The LightKer implementation implemented Mailboxes to schedule

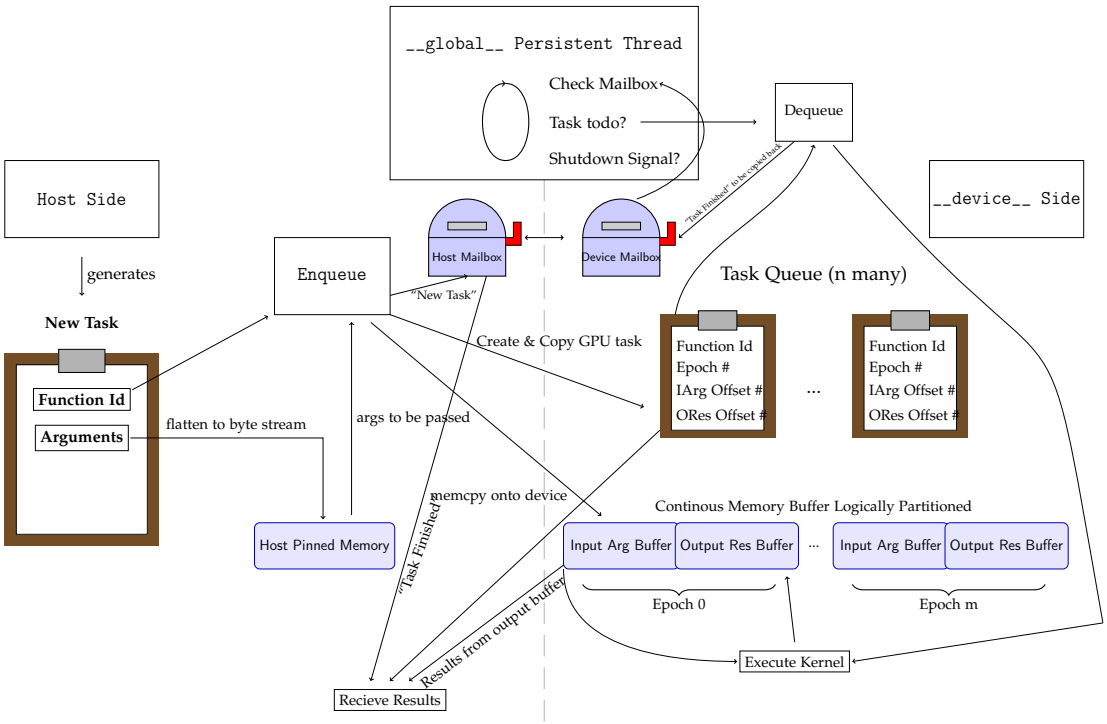


Figure 5.1: Persistent Thread Architecture

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] 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.

[4] 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.

[5] 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].

[6] 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.

[7] 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.