TUM

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Persistent GPU Threads for Real Time System Scheduling

Jaden Rotter

TUM

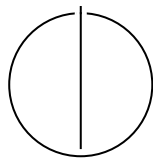## SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Persistent GPU Threads for Real Time System Scheduling

Persistente GPU Threads für die Echtzeit Systemplanung

Author:	Jaden Rotter
Examiner:	Michael Gerndt
Supervisor:	Jianfeng Gu
Submission Date:	August 18th, 2025

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

Munich, August 18th, 2025   Jaden Rotter

Acknowledgments

Abstract

This thesis implements a real-time persistent GPU scheduler that can serve as a foundation for a future coroutine and priority-based scheduling implementation. By keeping threads alive across kernel executions, this approach reduces scheduling overhead and allows tasks to be managed more efficiently and at a finer granularity on the GPU. Results indicate that these persistent threads are faster and more flexible than traditional kernel launches, making them a promising building block for future coroutine-based programming in areas such as autonomous driving. These findings highlight the significant impact that low-level GPU scheduling strategies can have on the performance and responsiveness of real-time workloads.

Contents

1 Introduction

1.1 Motivation

Autonomous driving systems place stringent demands on computational performance, predictability, and safety [1]. These demands arise from the need to process vast amounts of sensor data and run complex perception and decision-making algorithms in real time, while ensuring timely and deterministic responses to a dynamic environment. To meet the computational requirements of such systems, GPUs have become essential due to their performance on machine learning workloads. However, the current GPU programming and execution model is poorly suited to real-time constraints [2]. The autonomous driving platform, Apollo, which serves as the primary motivation for this thesis, illustrates this challenge [3] [4]. While Apollo integrates GPU-accelerated workloads, it does not natively provide real-time GPU support. Although this thesis does not implement the proposed approach directly within Apollo, the challenges of meeting real-time requirements in autonomous systems, particularly in GPU workloads, serve as the central motivation for this research.

1.1.1 Evolution of Autonomous Driving Architectures

Historically, early autonomous driving systems addressed real-time requirements using a distributed architecture. In this approach, major functional modules, such as perception, localization, planning, and control, were mapped to separate compute units, which together formed a processing pipeline [5]. Each module could thus operate with predictable timing characteristics, avoiding contention with other modules. In this manner, the distributed architecture allowed for fine-tuning the timing between modules to achieve low-latency responses from the hardware. This modular architecture ensured responsiveness and real-time guarantees, albeit at the expense of high hardware costs and increased system complexity.

The rise of increasingly powerful GPUs has enabled a shift toward centralized computing, simplifying hardware and reducing complexity while lowering costs [6]. In this centralized architecture, all core driving modules share a common compute node consisting of a heterogeneous CPU-GPU system. The compute node integrates a CPU for task scheduling and system control, which assigns computationally-intensive

workloads to the GPU. Moving to a singular compute node allows savings in cost, design, and intermodule latency.

1.1.2 Limitations of Current GPU Execution Models

Despite the benefits afforded by a centralized architecture and the advances in hardware, the system risks GPU oversubscription. As the GPU is responsible for all processing tasks, too many simultaneously scheduled tasks can lead to delays in execution time. Typical real-time solutions based on an CPU architecture ensure system safety under contention by preempting non-critical tasks. Tasks requiring high responsiveness can then be executed directly after the preemption of resident processes. Although native support for this preemption already exists on CPUs within both the kernel and user space, GPUs currently lack such capabilities.

Modern GPUs do not natively support real-time programming models in the same way CPUs do [7]. GPU execution is managed by a hardware scheduler that handles thread blocks, warps, and memory accesses according to fixed, internal policies. This internal GPU hardware scheduler is optimized for throughput rather than deterministic execution, which is required by real-time systems. In effect, programmers have limited control over the exact scheduling of threads or tasks when using standard scheduling techniques. Critically, tasks with strict timing requirements may suffer delays if long-running, lower-priority kernels are already resident on the device. The inability to natively preempt GPU kernels or enforce strict task priorities renders standard GPU kernel scheduling methods unsuitable for safety-critical, real-time workloads. Particularly in the context of autonomous driving, the repercussions of latencies pose a safety hazard.

1.2 Problem Statement

Rather than relying on native kernel launches, this thesis investigates the use of persistent threads to enable low-latency execution and develops the foundational framework needed to integrate coroutines into the system for further real-time determinism. Persistent kernels are specialized functions that are launched at the start of the application and remain active throughout the system's lifetime. They function as a user-level scheduler, continuously polling for work, executing tasks, and executing scheduling decisions.

Building on this framework, the central research question addressed in this thesis is:

How can a persistent GPU thread model be designed to enable the implementation of GPU coroutines for predictable, low latency scheduling on real time systems?

As part of this research, the following aspects of the persistent threaded implementation will be considered and evaluated:

- **GPU Task Management:** Mechanism for submitting tasks to the GPU and retrieving results asynchronously.

- **GPU Device Memory Management:** Efficient allocation, deallocation, and usage of GPU memory to ensure low-latency access and avoid contention between tasks.

- **GPU Stream Management:** Organizing and scheduling multiple GPU streams to enable concurrent task execution with memory transfers, while maintaining predictable execution orders.

- **Framework for Coroutines:** Designing a foundation for GPU coroutines that allows cooperative multitasking and the implementation of custom scheduling strategies on top of persistent threads.

1.3 Objectives and Contributions

1.3.1 Original Objective

The original objective of this thesis was to integrate an existing coroutine-based GPU scheduling framework into an autonomous driving system. This integration aimed to evaluate the feasibility of fine-grained GPU scheduling within a complex, real-time environment. Furthermore, by measuring scheduling latencies in the system, it would be possible to derive and refine strict timing guarantees.

1.3.2 Challenges and Scope Adjustment

Direct implementation of the coroutine framework proved infeasible within the available time frame. The system was initially designed for graphics rendering, relying on a complex compiler-based architecture with limited documentation, and required a separate build process. Combined with limited prior experience in GPU programming and compiler theory, these factors made integration within the timeline challenging.

To simplify the problem, the focus shifted from coroutines to the underlying execution model, persistent threads. No suitable open source implementation was found that could be directly integrated into an autonomous driving system. Instead, a minimal, open-source, custom persistent thread scheduler, LightKernel, was found and extended to be suitable for real-time scheduling [8]. The implementation initially measured only scheduling overheads from kernel launches, but provided a foundation for implementing fully functional persistent threads. This new scheduler serves as a proof of

concept foundation that implements long-running GPU kernels to receive and execute tasks efficiently. Furthermore, it provides the framework on which GPU coroutines can be implemented to enable suspension and prioritization between tasks, allowing exploration of how real-time behaviors can be approximated within the constraints of the CUDA execution model.

1.3.3 Contributions

This thesis designs and implements several persistent thread components to enable the efficient execution of GPU code.

1. **Task Management System:** An execution management system was developed to allow tasks to be queued and executed independently of resident executing kernels. This system captures the full execution context of functions, enabling persistent threads to retrieve, schedule, and execute tasks without requiring new kernel launches.

2. **Memory Management System:** To eliminate memory allocations throughout the runtime of the persistent kernel, a memory management system was designed. Tasks scheduled to the persistent kernel utilize the memory management system for their associated memory allocations. In this system, memory is mapped into a running epoch buffer of preallocated memory assigned before the launch of the persistent kernel. This system also provides a logical partitioning of input and output buffers to reduce data interdependencies.

3. **Concurrent Memory and Execution models:** To support concurrent memory transfers between the host and device, as well as simultaneous kernel execution, the system leverages multiple independent CUDA streams to resolve data dependencies. This system enables higher overlap between memory and compute operations, resulting in better utilization of GPU resources.

4. **GPU Task Coordination and Synchronization:** The persistent kernel is spread across multiple thread blocks to execute distinct tasks concurrently. This system is synchronized within the device code to ensure that no two blocks execute the same task simultaneously. By enforcing exclusive task execution, the mechanism prevents race conditions and ensures correctness across all kernels. Furthermore, this system enables the host to schedule efficiently and queue tasks to any available thread block, thereby maintaining high GPU utilization.

2 Background

This background chapter begins by examining the role of GPUs in real-time systems, focusing on the architectural and programming constraints that influence scheduling and performance. The section then develops into an analysis of real-time programming models for GPUs, considering persistent threads and coroutines as implemented in the autonomous driving system Apollo. Futhermore, this section provides the necessary technical background to understand how GPU design influences system behavior and scheduling under real-time constraints.

2.1 Real Time Systems

Real-time systems are designed with strict timing constraints to ensure predictable and safe behavior. [9] These constraints are expressed in terms of the system's ability to meet task deadlines, classified as either soft or hard deadlines. Hard deadlines are critical to the system's safety, and missing these deadlines can result in potential system failure or unsafe conditions. For example, missing the deadline on tasks such as collision avoidance or brake activation severely impacts the system's safety. Soft deadlines, in contrast, are less critical and can be missed without compromising system integrity. To ensure the system's safety, tasks are prioritized based on both their deadline urgency and the criticality of their impact. Prioritization in real-time systems often requires preempting soft deadlines or non-urgent tasks to prioritize hard deadlines and critical tasks.

2.2 Integration of GPUs in Autonomous Driving Systems

Early autonomous systems relied solely on CPU based compute engines for system control and task execution. For example, the Stanley autonomous car, which won the 2005 DARPA Grand Challenge, utilized six Pentium processors, among which tasks were divided [10]. The CPU-centric design, combined with resource partitioning, enabled the system to satisfy real-time requirements using established approaches.

As deep learning techniques advanced, CNNs began to outperform traditional methods in perception tasks such as object detection, lane recognition, and environment

understanding. This shift necessitated the integration of GPUs to handle the increased computational demands. Early autonomous driving systems, which relied solely on CPUs, struggled to continue performing well in processing data with CNNs in real-time. One of the earliest projects using GPUs for autonomous driving was NVIDIA's 2015 project, which utilized a GPU to train a CNN that could steer a car end-to-end from raw camera input [11]. Today, nearly all modern autonomous platforms rely on GPUs for sensor processing, perception, and decision-making tasks.

The rapid adoption of GPU into these systems depends heavily on their massive performance gain on highly parallel workloads, such as neural network inferencing and training. Deep neural networks, inspired by the structure of the human brain, learn patterns through layers of weighted nodes. These nodes perform large numbers of matrix operations, tasks that are highly parallelizable and therefore perfectly suited for GPU architectures [12] [13]. As a result, GPUs can exploit the inherent task parallelism to significantly outperform CPUs in both training and inference.

2.3 GPU versus CPU Architecture

GPUs deliver a significant increase in throughput over CPUs, by simplifying the thread context to afford greater parallelism. Initially developed to accelerate graphics rendering, a task that is heavily reliant on parallelizable computations, GPU architectures were designed to support increasing numbers of threads. Because these tasks required very little control overhead, GPU threads were intentionally kept simple, avoiding the complex control logic found in CPU thread management.

Although the CPU offers some parallelism, it is primarily optimized for sequential tasks, relying heavily on the performance of individual threads. To achieve greater performance on these tasks, CPUs dedicate a substantial "portion of their transistors to non-computational tasks", specifically control and memory management logic [14]. Features such as prefetching, branch prediction, speculative execution, and out-of-order execution enable CPUs to handle irregular, sequential control flows efficiently. However, this complexity comes at the cost of reducing the fraction of hardware dedicated purely to computation. In contrast, GPUs minimize control overhead, instead allocating resources to maximize arithmetic throughput across many lightweight threads. Consequently, CPUs operate at higher clock rates, while GPUs achieve higher throughput by increasing arithmetic intensity.

Consider the following graphic, Figure 2.1, which highlights the difference in thread complexity.

CPU Thread   **GPU Threads**

Simplification via Parallelism →

Complex, latency-optimized thread   *Thousands of simple, throughput-optimized threads*
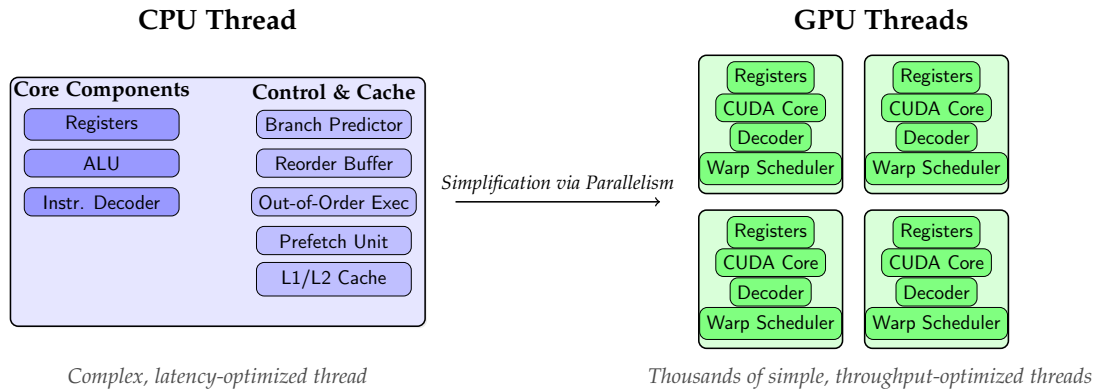
Figure 2.1: CPU vs GPU Thread Architecture

Although core components are named differently, both CPU and GPU threads operate similarly, utilizing an instruction decoder, registers, and an arithmetic unit. The differences arise when trying to maximize a single control flow. In contrast to CPU threads, GPU threads execute instructions entirely in order, rely on manual prefetching, and use a simpler, conservative branch predictor, which limits their ability to optimize single-threaded performance. Instead of optimizing single-threaded performance, GPUs achieve high throughput by executing thousands of lightweight threads in parallel, amortizing latency across them. This design favors workloads with high data parallelism, enabling the GPU to hide memory and execution latencies through massive concurrency rather than complex control logic.

The additional complex logic that CPUs use to improve single-threaded applications outperform the single-threaded GPU applications, as seen by the following comparison of single-threaded matrix multiplications in Figure 2.2 and Figure 2.3. These figures compare the execution of a matrix multiplication using a single GPU thread versus a single CPU thread on a matrix multiplication task.

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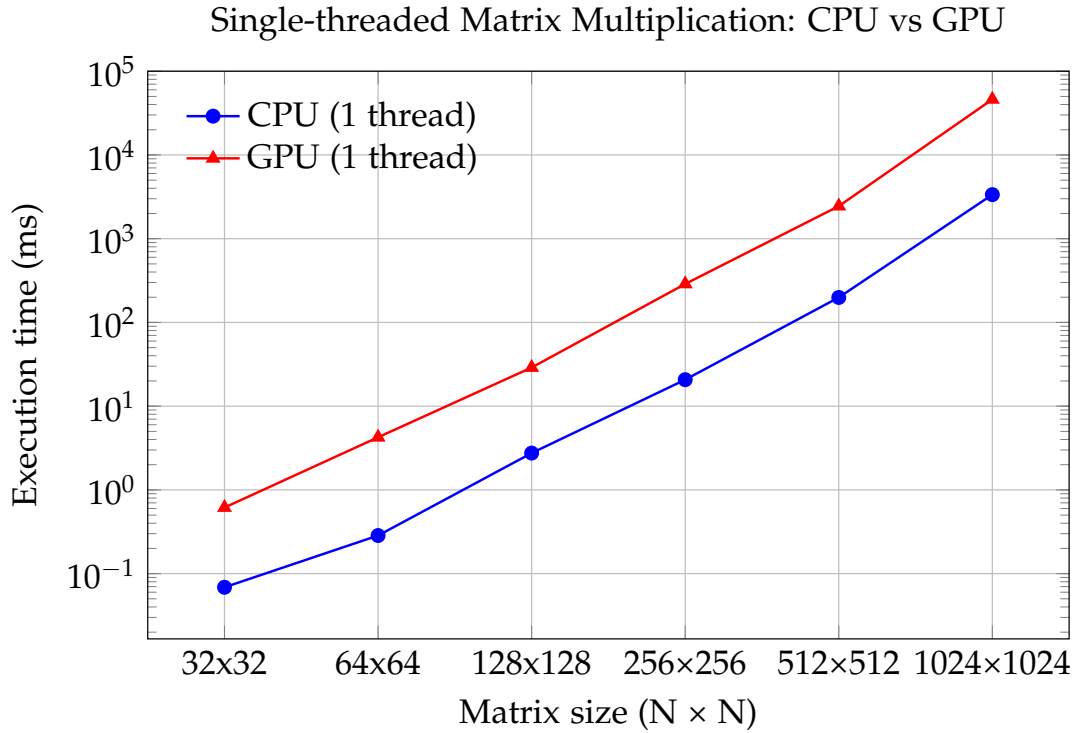
Figure 2.2: Single threaded Matrix Multiplication Execution between CPUs and GPUs averaged over 10 executions

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 Figures 2.2 and 2.3, GPUs struggle in applications that fail to utilize their architectural parallelism. For each of the matrices tested, the CPU is on average around 13 times faster than the GPU. Consequently, the GPU should only be used in place of the CPU when the application is parallelizable and lacks complex control flow logic; otherwise, the application will significantly underperform.

For this thesis, any GPU resident scheduler that employs complex control logic will introduce significant latency overhead. Scheduling decisions, task management, and resource assignment are inherently challenging to parallelize, making them inefficient on GPU hardware. As a result, the scheduling logic is delegated to the CPU, where complex control mechanisms can be executed efficiently. Consequently, the GPU's role is reduced to dispatching and executing lightweight tasks, while the CPU handles task queuing and resource allocation.

2.3.1 GPU Architecture

The NVIDIA Tesla V100 GPU, based on the Volta architecture, was selected for this project due to its availability and suitability for high-performance computing workloads [15]. In particular, this GPU is used for all related tests, including Figure 2.2, Figure 2.3, and the results in Chapter 5. The following analysis of the V100's hardware architecture will provide both the reasoning and implementation fundamentals for designing an efficient GPU scheduling strategy.

Thread Hierarchy and Execution Model

From the programmer's perspective, when assigning tasks, the GPU appears as an array of independent, highly parallelized processors, called SMs. Each SM receives work in the form of CTAs, which are blocks of threads executing the same instruction code. These define the organization and grouping of threads for execution. The executing CTA is subdivided into warps, the smallest execution unit on the GPU, each consisting of 32 GPU threads executing instructions in lockstep. The lockstep execution ensures all threads within a warp execute the same instruction simultaneously. By enforcing lockstep execution, the GPU can schedule and dispatch threads to compute units with a simplified design, allowing further parallelism.

While lockstep execution enables efficient SIMD-style throughput, it also introduces a potential performance hazard. If threads within a warp follow different control flow paths, these threads diverge, and the GPU is forced to execute them sequentially with respect to one another. During this serialization, inactive threads in the warp are masked out, preventing them from writing results back to memory. The result is a reduction in effective parallelism, leading to degraded overall performance.

SM Architecture

The Tesla V100 GPU SM architecture contains four processing partitions, each with its own complete execution pipeline. These partitions share an L1 instruction cache as well as a combined L1 data and shared memory cache, enabling threads within different

warps of the same CTA to efficiently access shared instructions and data. Within each processing partition, there is an L0 instruction cache, a warp scheduler, a dispatch unit, and multiple execution units. An in-depth view of each processing partition's architecture is provided in Figure 2.4, taken from the NVIDIA Volta Whitepages.
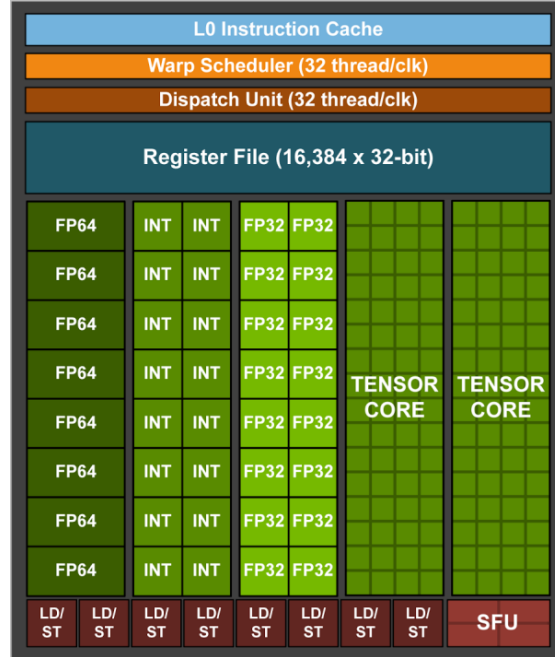
Figure 2.4: SM Processing Partition Architecture taken from the Volta Whitepages

Every clock cycle, the warp scheduler selects a warp of 32 threads to issue to the dispatch unit, which dispatches decoded instructions to the appropriate functional units. If there are not enough execution units of the required type for a given instruction, the instruction is queued. Depending on the current queue and delays, such as global memory accesses or dependencies, the warp scheduler will interweave different instructions from other ready warps, ensuring that the execution units remain busy. This thread interweaving allows GPUs to hide latencies and resource contention through thread oversubscription.

In GPU scheduling, oversubscription can improve hardware utilization, albeit at the expense of introducing overhead. By scheduling more warps than can actively execute, the GPU can potentially interleave tasks and keeps resources busy during stalls. Whether this is beneficial depends on the workload goals: oversubscription increases throughput, but it may also add latency to individual tasks. Real-time workloads may benefit from oversubscription, but the level should be tuned to the system's

performance goals to ensure minimum timing guarantees.

SM Scaling and Thread Concurrency

Compared to CPU hardware threads, GPUs scale far more aggressively, supporting a much larger number of threads. On the Tesla V100, there are 80 individual SMs, each capable of supporting up to 64 resident warps. With 32 threads per warp, this yields $64 \times 32 = 2048$ threads per SM. Across all 80 SMs, the theoretical maximum concurrency is $80 \times 2048 = 163{,}840$ resident threads. For comparison, a typical Intel Tiger Lake i5-1135G7 CPU has 4 cores with 2 hardware threads each, for a maximum of 8 concurrent hardware threads. High-end server CPUs, such as the Intel Xeon Gold 6148, only support 20 hardware threads. Although the GPU oversubscribes the number of warps and threads to hide latencies, the total number of dispatch and scheduling units allows for a maximum of $4 * 32 * 80 = 10840$ instructions issued per cycle when the hardware is fully utilized.

In practice, this theoretical maximum is rarely achieved due to resource bottlenecks. All threads within an SM share the same on-chip L1 data and shared memory cache, as well as the 256 KiB on-chip register file ($16{,}384 \times 32$ bit registers per processing partition, with four partitions per SM). Depending on the kernel's resource usage, high register pressure will cause the kernel launch to fail. Furthermore, suppose multiple thread blocks with high shared memory demands are scheduled to the same SM. In that case, they will saturate the L1 data and shared memory cache, forcing frequent evictions and writebacks to the L2 cache or global memory. These hardware limits must be carefully considered when mapping tasks to SMs.

Native GPU code handles these constraints automatically, without requiring any input from the programmer. When launching kernels, the host, driver, and device each analyze the programming configurations to see if the hardware can support the task. Should the hardware fail, the launch silently fails with a `cudaError_t` return type.

However, building a GPU scheduler on top of this system introduces additional challenges. A custom scheduler must account for the same hardware constraints, thread limits, register pressure, shared memory, and occupancy when mapping tasks to the GPU. At the same time, the scheduler operates within the bounds of the proprietary hardware scheduler, which already enforces its own resource allocation and dispatch policies. As a result, attempts to impose new scheduling behavior may conflict with or be restricted by the GPU's built-in mechanisms, limiting the degree of control available to the programmer.

Scheduling and Task Mapping

Within the CUDA programming model, the specific mapping of tasks to SMs, TPCs, and GPCs is determined natively by the proprietary hardware scheduler, the GigaThread Engine. While the exact documentation is not public, this module maps CTAs to the individual SMs based on a multitude of factors: hardware resources, parallelism, priorities, and dependencies. Similarly, the global memory and L2 cache utilization are determined by the hardware and are transparent to the programmer. After the CTA gets mapped to the specific SM, the device code then executes till completion without interruption.

The CTAs is entirely managed by the SM on which it is currently executing. As shown previously in Figure 2.4, the SM has its own execution pipelines, register files, shared memory, and scheduling units on which the CTA executes. For SMs to communicate with one another, they must use either the global on-chip device HBM2 or the global L2 cache, which is shared and coherent across all SMs. Although the global memory 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 accumulate answers in global memory, thereby skipping the high memory latency associated with coordinating synchronous work.

2.4 GPU Programming using the CUDA API

NVIDIA GPUs are designed as computational accelerators for a host system, which manages and schedules tasks using the CUDA programming model. In this model, the GPU acts as an independent processor with its own memory and execution pipelines. For tasks to be scheduled on the GPU, the host process must first launch it through the CUDA API, an extension of the C++ programming language that enables the CPU to communicate with and control the GPU.

Using CUDA, the host invokes device functions, called kernels, while specifying execution parameters such as the number of threads and memory configuration. These kernel launches are asynchronous, meaning that once the host issues the call, the GPU executes it independently, allowing the CPU to continue other work or synchronize with the GPU later as needed.

2.4.1 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. The launch configuration specifies the number of CTAs and their logical dimensional organization. Every block is then mapped to a single SM and is constrained by that SM's hardware resources, including the maximum number of threads, registers, resident warps, and available shared memory. If no available SM can meet these requirements, the kernel launch will fail. Conversely, if a CTA does not fully saturate the hardware resources of an SM, additional further blocks may be scheduled concurrently on the same SM.

2.4.2 Host to GPU Memory Transfers and Bandwidth Considerations

Both the GPU and CPU maintain distinct memory regions to match their respective architectural and performance requirements. CPU memory is optimized for low-latency access to efficiently handle serial and branch-intensive workloads. GPU memory, on the other hand, is designed for high throughput, allowing thousands of threads to execute in parallel while tolerating higher latency per access.

Since the memory regions are distinct, the CPU must explicitly allocate and transfer data to the GPU using the CUDA API. Transfers require the CPU memory to be pinned in RAM, as CUDA cannot access the disk directly. While the CUDA runtime can automatically pin memory, host arrays allocated directly in pinned memory using `cudaMallocHost()` or `cudaHostAlloc()` eliminates this extra step and enables faster transfers. However, allocating excessive pinned memory reduces the RAM available to other CPU processes, potentially causing paging and degrading system performance. Careful management of pinned memory is therefore essential for maximizing data transfer efficiency without compromising the host system's performance.

In CUDA, understanding how memory is transferred and managed across the host and the device is crucial for optimizing performance. The device memory consists of a global 16 GB block on the Tesla V100 architecture, as well as on-chip processor memory. To improve memory performance, the programmer can leverage 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.

2.4.3 Memory Coalescing

For the GPU to coalesce memory operations and enable SIMD-style execution across multiple data points, each thread maintains registers that store its execution context, including its position within the kernel and executing CTA. In PTX, these special

registers provide unique identifiers such as threadIdx and blockIdx, which indicate a thread's coordinates within its block and a block's coordinates within the grid. These identifiers are essential for structuring parallel computations and optimizing memory access patterns. For example, when threads within a warp access consecutive memory locations, the hardware can coalesce those accesses into a single memory transaction, significantly improving throughput.

2.4.4 Example Kernel Launch

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

```cuda
__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 code block above illustrates the launch and execution of a simple GPU kernel, as well as the memory allocation scheme used for executing kernel code. The kernel itself is the execution of the GPU device program denoted by the `__global__` function. In contrast, the <<<_, _>>> syntax enables the programmer to partition their execution tasks across waiting threads specifically. The values in <<<1,32>>> specify the grid and block dimensions, determining how many blocks are launched and how many threads execute per block. These dimensions can be given as integers or as a dim3 struct for 1D, 2D, or 3D layouts. 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 sections of the same array. This is achieved by utilizing the dimensional properties assigned to each thread 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 CPU stack, which is explicitly copied to the GPU. Trying to pass the array by value, something common in C++ code, seems at first the simplest; however, it poses two separate issues. Firstly, when passing arrays as parameters, they decay to pointers, which CUDA forbids, as the pointer address passed to the device has no meaning on the GPU. Secondly, if the array were wrapped in a struct and passed to the function to circumvent the first issue, the array would be allocated independently for every single thread. In the example above, the array would be allocated 32 times, each independent of the others, 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 of 32 threads * 1024 floats * 4 bytes per float or 128KiB. Instead, the memory is allocated in the device memory, transferred once, and each thread receives only the device pointer `d_x`, which can be used to copy the results back to the host.

2.4.5 CUDA Streams

CUDA API calls are queued to the GPU using CUDA streams, which enforce the execution order of tasks. A stream, represented by the type `cudaStream_t` acts as a handle to a specific command queue, similar to how a file pointer in Linux refers to a particular file descriptor. Within a stream, operations such as kernel launches, memory copies, and memory set operations are enqueued and executed strictly in the order they are issued, ensuring deterministic behavior. Commands submitted to the same stream are executed sequentially in the order they were issued, ensuring deterministic behavior within that stream. When using multiple streams, the GPU can execute operations concurrently, allowing kernels and memory operations to overlap in time. By carefully managing streams, developers can expose task parallelism, reduce idle time, and utilize GPU resources more effectively.

The Tesla V100 GPU features two separate hardware copy engines for copying data from the host to the device and vice versa. 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 utilize the device's memory bandwidth fully. 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 the API calls, the two kernels cannot execute concurrently or share the hardware copy engines, since their execution is serialized by being placed in the same CUDA stream. To maximize hardware utilization and ensure correctness, each kernel and its associated memory transfers should be placed in the same dedicated stream, allowing the CUDA runtime to safely overlap kernel execution, memory copies, and bidirectional transfers when possible.

2.5 GPU Programming Models for Real Time Systems

To meet urgent deadlines, systems must prioritize and ensure the timely execution of critical tasks. Prioritizing execution requires that high-priority tasks are scheduled first when resources are available, and that their deadlines are still met even when resources are fully occupied. Achieving this responsiveness requires preemption or context switching between resident tasks and scheduled critical tasks. In Apollo, this responsiveness is implemented through coroutines, which cooperatively yield execution to enable timely task switching. Coroutines are particularly well-suited to GPU scheduling, since GPUs lack integrated hardware preemption and therefore depend on cooperative mechanisms to maintain responsiveness.

2.5.1 Coroutines

Coroutines are a form of asynchronous programming that enables cooperative multitasking between functions. Unlike traditional thread or process switches, which can occur at any time and require a larger context, coroutine switches happen only at programmer-defined suspension points. This makes them particularly useful for enabling runtime kernel task switching on the GPU. Here, execution of a kernel can be suspended to allow another kernel to run, and then later resumed without blocking other work.

A coroutine suspends execution by capturing its current context, known as the continuation, which contains the execution state needed for later resumption [16]. Once suspended, another task can execute without overwriting or disrupting the saved kernel state. When the interim task completes, the coroutine can continue exactly where it left off by restoring its continuation. This ability to strategically pause and resume execution makes coroutines well-suited for real-time workloads, where rapid switching between concurrent tasks can help meet hard deadlines without delay.

CPU Coroutines in Apollo

Coroutines on the CPU are implemented using the native x86 calling conventions and the program stack to save and restore continuations dynamically. Under x86 convention, when a function is invoked, the CPU pushes the return address, the location of the next instruction after the call, onto the stack before jumping to the target function. The called function accesses its local variables through the stack and registers, which are divided into two categories: volatile, caller saved, and non-volatile, callee saved. The callee may freely modify volatile registers, while callee saved registers must be preserved and restored before the function returns.

When execution reaches a return instruction, the CPU pops the return address off the stack and continues execution from that point. To implement coroutine behaviour, however, the function must be able to yield and later resume. This requires saving the callee-saved registers, since their preservation is guaranteed, as well as any additional state such as local variables or registers, that is necessary for continued execution. These elements can be stored on the stack, allowing a coroutine to pause and continue seamlessly later.

The Apollo project demonstrates this mechanism by implementing CPU coroutines directly on top of the x86 calling convention, as illustrated in the following code block.

```
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 2.2: CPU Coroutine

The CPU coroutine implementation uses the context switching function, ctx_swap,

which captures the current continuation and restores the state of another coroutine. This function accepts two parameters, %rdi and %rsi, which point to the memory locations used to store and retrieve coroutine continuations. Execution happens in two stages. In the first stage, the continuation is stored and its address is saved to the register %rdi. The second stage then moves the stack frame to the location of its continuation, using the register %rsi, before restoring the new coroutine context.

Each stage loads or saves its registers in a specific, defined order. In particular, to ensure the memory is reread into the correct registers, the order of pushing elements onto the stack is the reverse order of popping elements. As the stack works on a last-in-first-out principle, this ensures that all the registers have the correct values upon resumption. In comparison to traditional context switches involving processes or threads, coroutine context switching via ctx_swap operates with significantly lower overhead, resulting in faster execution.

2.5.2 GPU Coroutines

Unlike CPU coroutines, which can implement context switching by saving and restoring stack frames, GPUs cannot use the exact mechanism, as GPU threads do not maintain conventional stack frames. Instead, their execution state is managed differently, requiring specialized approaches to preserve and resume coroutine execution.

Attempting to implement CPU-style coroutines using the ctx_swap function does not work because GPUs handle function calls differently. CPUs stores a call stack of function stack frames, which can be easily accessed and manipulated by pushing and popping values or addresses. GPUs, in contrast, avoid traditional stack frames by aggressively inlining function calls, reducing overhead since function code is readily available and no stack frame is needed.

Implementing and manipulating deep call stacks on GPUs is primarily handled by the compiler and hardware, leaving very little control to the programmer, which makes such approaches impractical. For example, CUDA initially did not support recursive functions, which were only introduced later via the `-rdc=true` flag. Even then, recursion comes with strict limitations on stack size and adds significant performance overhead. Attempting to rely on deep call stacks can quickly lead to excessive instruction and register usage, illustrating why GPU architectures and compilers discourage CPU-style stack manipulation.

Persistent Threads to support Coroutines

Given that call stacks are not effectively or efficiently supported in CUDA, the GPU must support a user-level runtime scheduling mechanism to manage contexts dynamically.

Due to the nature of kernels executing until completion, the GPU coroutine scheduler must be based on a persistent kernel, which can schedule coroutines throughout the application's lifetime. The coroutines running on the persistent threads will then define suspension points to manually relinquish control to the scheduler, allowing it to switch tasks. Because storing every register for all threads across coroutines is too expensive and inaccessible, coroutine contexts must be explicitly synthesized and saved in global memory, freeing the limited SM resources for new tasks.

2.5.3 Persistent Kernels

Persistent GPU threads provide the programmer with enhanced control over hardware scheduling by running on top of the hardware scheduler, enabling manual execution decisions that would otherwise be unavailable. In addition to this increased flexibility, persistent kernels reduce runtime scheduling overhead. As hardware resources are preallocated, thread and block configurations are loaded ahead of execution. The runtime avoids repeatedly configuring these settings, allowing kernels to run more efficiently. In systems with recurring or periodic tasks, such as autonomous driving, this overhead becomes particularly costly. With the implementation of persistent threads, only input and output buffers need to be updated for each new task, minimizing execution overhead and allowing kernels to operate with only the essential arguments required for computation.

3 Related Work

The increasing use of GPUs in real time systems has led to the development of custom programming models designed to reduce kernel launch overhead and improve predictable GPU scheduling. Such models are particularly useful in systems where minimizing execution latency, improving resource utilization, and mainting timing determinism is important. These approaches can be broadly categorized into compiler driven frameworks, runtime scheduling frameworks, and manual implementations, the latter of which is the focus of this work.

3.1 Compiler Driven Frameworks

Compiler driven frameworks optimize GPU workloads through automated code generation. The user writes codes in a high level DSL that abstracts low level device details. At runtime, the framework parses the DSL into an AST, which is then converted into an IR. From this intermediate representation, the compiler applies transformations and optimizations before compiling the code JIT into GPU executable device code.

Several projects demonstrate how these frameworks are applied in practice. *Mirage*, for example, improves large language inference by fusing kernels into a single megakernel [17]. This achieves the same effect as persistent threads in removing extra kernel task scheduling overhead. Similarly, *Halide* seperates algorithm specification from execution schedule, allowing compiler autoschedulers to optimize performance with minimal manual tuning [18]. Finally, *Luisa* targets graphics and simulation workloads, offering acceleration structures, ray traversal APIs and shader abstraction to provide high level rendering code while retaining low level performance [19].

Luisa, in particular, offers a model that can be adapted for real time applications. Built on top of Luisa's execution model, *LuisaCompute-Coroutines* extends the framework to support coroutines running on persistent threads [20]. Coroutines are expressed simply within the DSL as explicit suspension points, which allow the device to preserve context and yield, enabling fine grained scheduling. This approach is especially efficient and well suited for real time systems, as it leverages asynchronous coroutine execution to improve responsiveness and resource utilization.

3.2 Runtime Scheduling Frameworks

Beyond compiler based transformation and code generation, runtime based frameworks offer an alternative approach by enabling real time GPU scheduling. For example, *RT-GPU*, a runtime system, provides deadline aware scheduling of GPU workloads by partitioning GPU resources [21]. Using a reservation based model, RT-GPU enables fine grained control over scheduling to ensure the task deadlines are met. Additionally, *ROSGM* is a GPU management framework designed specifically for ROS 2 robotics systems [22]. ROSGM interposes a layer to intercept the CUDA API calls and attach metadata to each GPU task, allowing custom deadlines and priorities to manage how GPU tasks are queued and issued to the device.

Overall, these frameworks demonstrate different approaches to achieving predictable GPU execution. Each of these frameworks relies on a form of high level user abstraction to manage scheduling and low level execution. While compiler and runtime systems simplify development through abstraction and automation, they do not allow for highly application specific fine tuning. Manual implementations, in contrast, enable this level of control and form the basis for the fine grained GPU scheduling explored in this thesis.

4 System Design and Implementation

This thesis initially aimed to introduce GPU coroutine support to the open-source autonomous driving platform Apollo by implementing the LuisaCompute-Coroutines framework discussed in Chapter 3. The goal was to extend Apollo's existing CPU coroutine infrastructure to GPUs, providing fine-grained, predictable scheduling for real-time tasks. However, due to integration challenges and the project timeline constraints, directly implementing LuisaCompute-Coroutines proved infeasible. Consequently, the scope shifted toward a manual implementation: a persistent thread scheduler for GPUs that lays the groundwork for coroutine support while still enabling fine-grained control over GPU execution. This section presents the design and implementation of that scheduler and the foundation it provides for future coroutine integration.

4.1 Platform Integration: GPU Scheduling in Apollo

This thesis initially explored the integration of LuisaCompute-Coroutines into Apollo, aiming to provide GPU coroutines that complement the existing CyberRT CPU coroutines. Similar to the CPU coroutines, these GPU coroutines were intended to deliver predictable execution latencies, with the added latency improvements that GPU persistent threads offer. However, due to several integration barriers, including foreign build systems, sparse documentation, time constraints, and a limited familiarity with both projects, GPUs, and compiler theory, it became clear that integrating this system into Apollo would not be feasible within the scope of this thesis.

With the LuisaCompute's coroutines approach impractical, the focus shifted to implementing manual GPU scheduling functionality. Finding no suitable open source implementation for coroutines, this thesis instead turned to persistent threads. Most existing persistent thread implementations are highly application-specific and not readily available, which means that adapting any implementation requires substantial work. The only suitable implementation found was LightKer, a research project designed to measure the speedup of using persistent threads compared to sequential kernel launches [8]. To implement LightKer into a real-time system, the project required a complete restructuring of the codebase. Although full integration into Apollo and support for coroutines remain unfinished, this work extends the real-time capabilities of LightKer and provides a foundation for future GPU coroutine integration.

The LightKer implementation was primarily designed to measure the performance difference between sequential kernel executions and a persistent kernel implementation. It constructed simple, trivial kernels and compared the overhead of explicit host-launched kernels versus implicit execution within the persistent kernel. However, the framework does not support the runtime features necessary for a fully functional system. Nonetheless, it provides host device synchronization through a mailbox system and establishes a fundamental foundation for launching persistent kernels, which proved helpful in enabling real-time support for the system.

4.2 System Design and Architectural Overview

At a high level, the system incorporates four key components to enable real-time support: the task queue, memory buffers, stream design, and GPU block synchronization. The task queue is managed, controlled, and allocated by the host, providing the arguments and tasks for the GPU to execute. In particular, this implementation of the task queue provides the programmer with fine-grained opportunities to design and schedule workloads manually. The memory buffers serve as an epoch-based staging area for input arguments and output results, as well as persistent memory for the further extension of coroutines. Stream design and GPU block synchronization ensure that the hardware is utilized efficiently, minimizing idle time and improving performance.

The individual contributions described above are combined in the system architecture presented below. The architectural elements discussed form the core of this thesis, with additional tools and synchronization mechanisms, such as the mailbox system, adopted from the LightKernel project.
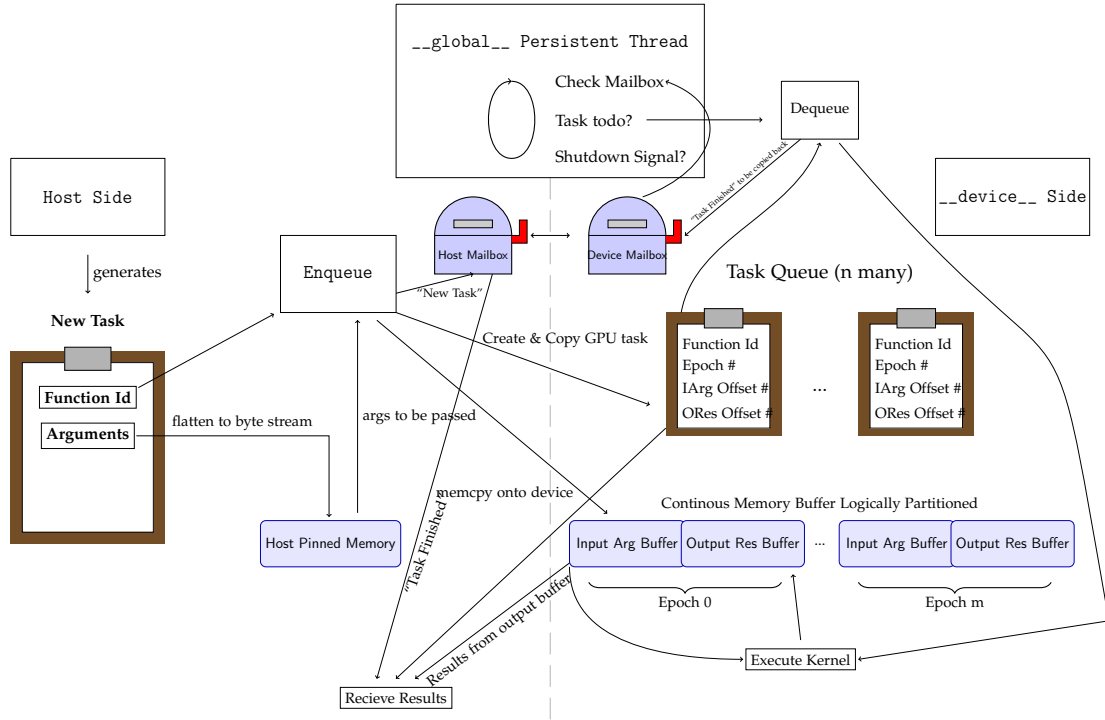
Figure 4.1: Persistent Thread Architecture implemented into LightKer

The Figure 4.1 depicts the individual components in a logical program architectural overview. The left side of the figure shows the host-side code and methods, which handle the enqueuing of tasks and launching of the persistent thread kernel. On the right-hand side, the device-side code is shown, illustrating memory allocation and kernel execution, which processes tasks and writes results back to the memory buffers. The mailboxes continuously notify each other of newly enqueuing and dequeuing tasks throughout kernel execution.

The system is initialized by launching the persistent kernel, as shown in the uppermost box in Figure 4.1, which runs for the entire duration of the program. The persistent kernel receives tasks enqueued from the host system, by the transfer of two different objects. The first object is the task itself, represented by a clipboard, which is copied into the task queue. The second object is the task memory, serialized by the host and copied into the continuous memory buffer. The device is then notified through the mailbox synchronization mechanism and begins executing tasks. Once a kernel finishes execution, the device notifies the host, which then copies the memory back to the host.

4.3 Task Management System

For persistent threads to execute kernels at runtime, a staging mechanism is required to enqueue tasks and preserve the task context needed for execution. As discussed in Chapter 2, any task management system that relies on device-side scheduling logic is inherently inefficient due to characteristics of GPU architecture. To address this, the task management system in this design adopts a host-driven scheduling model, where tasks are prepared and dispatched from the CPU, while persistent threads on the GPU handle execution. This approach improves control over task ordering, reduces idle time, and improves overall predictability, factors essential for real-time workloads. Additionally, this system is designed to remain compatible with future extensions, including coroutine-based execution and priority scheduling.
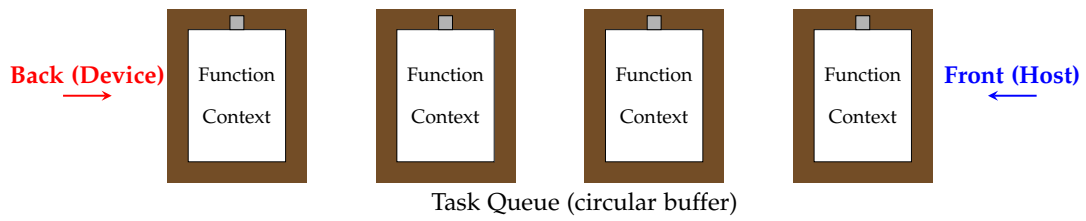
4.3.1 Task Queue Design

Figure 4.2: GPU Task Queue for Task Management

The architecture implemented in the persistent thread scheduler utilizes a loop-through buffer as a FIFO queue, following a producer-consumer model. As shown in Figure 4.2, each GPU task is represented as a clipboard, which encapsulates the function context. This context contains both the information required to execute the device function and metadata to manage the task within the queue. The host acts as a producer, enqueuing tasks into the front of the queue, while the device consumes functions from the back.

To improve scheduling efficiency, all task enqueuing and dequeueing logic resides on the host side of the system. Enqueuing a task involves copying its context into the task queue and initializing the associated control variables in the function context. Before execution, the device simply checks the control variables to ensure that only valid tasks are executed. Once a task is executed by a CTA, the device block will advance the back pointer to the next available task in the queue.

To prevent being overwritten, the host tracks the total number of active tasks relative to the total length of the task queue. Every time a new task is added, the tracker is incremented unless the queue is full, in which case a failure response is returned,

similar to the native CUDA system. The tracker is decremented only after the device executes the task, notifies the host, and the host copies the results back. Once the results are retrieved, a new task can then be enqueued in the same position in the queue.

4.3.2 Extensibility for Coroutines and Priority Scheduling

The function contexts stored in the task queue buffer were designed with future extensions for coroutines and priority scheduling in mind. By introducing additional control variables into the function context, the host system can assign tasks different priorities. In addition, the queue already provides space to store coroutine continuations, allowing a task to yield and record the current instruction for later resumption.

Currently, the system executes tasks in a loop using a synchronized busy waiting scheme. With slight modifications, however, it can be extended to support priorities. If every task additionally contained a priority or deadline in its function context, the persistent threads polling active tasks could select and execute the most critical tasks first. In this case, the host would need to actively track executing tasks to ensure they are not accidentally overwritten.

In conjunction with this system, the function context can also store the coroutine continuation. While precomputed values could be easily stored in the function context, capturing the resumption address is more challenging. On the GPU, this would require defining explicit set points within the program and saving a variable in the function context to indicate the next instruction to execute within the kernel.

4.4 Function Context

Each task entry defines its execution context through an explicit function identifier and the corresponding function parameters. This entry effectively serves as a coroutine continuation, storing the necessary state to resume execution within the function. The function ID is used in conjunction with a lookup table to dispatch and execute the appropriate GPU device code. Meanwhile, the memory location for the parameters is allocated by the host as part of the enqueuing and memory management systems.

4.4.1 GPU Function Pointers

To execute new functions from the persistent kernel, the task queue must be able to reference the specific function . On a CPU, this is typically achieved using a function pointer, which directly executes the code at a given memory location. On a GPU, however, compiled device functions reside in the GPU address space and are not directly accessible from the CPU. The CPU can only invoke functions marked with

the `__global__` keyword, which launch GPU kernels, but not standard device GPU functions. As the system should execute asynchronously without any direct input from the GPU persistent thread, this thesis used a lookup table to match tasks to device functions. When enqueuing new tasks, the host copies the corresponding function ID into the task queue, enabling the persistent kernel to dispatch the correct function.

4.4.2 Function Parameters

When the CPU assigns tasks to the GPU, it passes either allocated GPU memory pointers or explicit parameters to the GPU. These explicit parameters are then propagated to all the individual threads executing the kernel code, resulting in significant API memory overhead. In particular this approach results in variable memory utilizations and does not provide any standardization mechanism to encapsulate function contexts.

The GPU task in the queue originally had a pointer to the allocated memory, and upon receiving compute resources, would schedule the task with the memory to the individual persistent thread. Unfortunately, this method is dependent on the specific task and parameters, and it consumes variable memory, requiring further pointers to GPU memory. To consolidate memory pointers, the task queue was simplified to contain only allocated memory pointers, allowing for automatic loading of kernel memory.

In this method, enqueuing GPU tasks forces the programmer to streamify the data and automatically load it into preallocated memory partitions. The task queue then only consists of the actual memory partition pointers, both start and end. Executing a task then requires the interpretation of the memory and loading it into the device's function. Should the input memory be oversubscribed, the task then has a preallocated buffer to store any further context for the continuation of the coroutine.

4.5 Memory Management System

The memory management system supports the task queue and task parameters while eliminating unnecessary memory allocation and deallocation overhead. By handling memory centrally, the task queue's function contexts can remain flexible and straightforward. This memory system is allocated for the lifetime of the persistent kernel, removing the need for costly dynamic memory operations at runtime. Although maintaining a contiguous buffer is simple, designing a runtime strategy to partition memory among tasks and reclaim it efficiently presents a more complex challenge. As illustrated in Figure 4.3, the buffer is logically partitioned into epochs, enabling a cyclical allocation and deallocation strategy. Rather than freeing individual memory regions, an entire epoch is released once it becomes unused, after which it can be reused for new tasks.

4.5.1 Memory Buffer Design

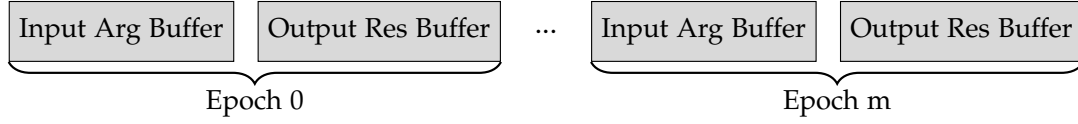| Input Arg Buffer | Output Res Buffer | ... | Input Arg Buffer | Output Res Buffer |

Epoch 0   Epoch m

Figure 4.3: Continuous Memory Buffer Logically Partitioned

The memory management system consists of a single, logically partitioned memory buffer, designed to reduce the complexity of allocation algorithms. Each epoch within this buffer contains both an input argument section and an output result section. The responsibility of managing the task memory positions within these buffers lies entirely with the host, which shares the information with the device through the function context in the task queue.

As tasks are allocated within the epochs, one of the corresponding memory buffers will eventually overflow. When the host detects that allocating memory for a task would cause either the input or output buffer to overflow, allocation continues in the next epoch. Once memory within an epoch has been fully allocated, it remains untouched by the scheduler until the buffer queue cycles back to that epoch. An epoch becomes available for reuse once all results from the tasks it contains have been copied back and the memory is marked free.

The host manages epochs by tracking the number of tasks within each epoch and the current offset of allocations. When enqueuing tasks, input arguments are written at the current offset of the active epoch. Tasks are only considered complete after their results have been copied back to the host, and the host marks the corresponding epoch memory as freed. By the time the buffer queue loops back, provided the buffer parameters and memory sizes have been appropriately chosen, potentially through profiling, the epoch will be available for reuse.

4.5.2 CUDA Stream Optimization

As discussed in Chapter 2, the Tesla V100 GPU features two independent memory transfer engines to support bidirectional data movement between host and device. One engine is dedicated to H2D transfers, while the other handles D2H transfers. These engines can operate concurrently, allowing for overlap between data movement and computation when used correctly.

To fully exploit this hardware capability, input and output transfers must be carefully organized. If tasks are enqueued into a single CUDA stream, transfers and kernel executions are serialized, causing the GPU to wait unnecessarily and leaving one of

the transfer engines underutilized. To avoid this, the implementation employs multiple CUDA streams, separating the input staging dependencies from output collection.

Specifically, input arguments are transferred from the host to the device using a dedicated H2D stream, while task results are copied back to the host through a separate D2H stream. This separation prevents intra-stream dependencies between input and output operations, ensuring that transfers in opposite directions do not block one another. This design enables continuous execution of persistent threads. While one batch of tasks is being executed on the GPU, the next batch can be staged in device memory, and previously completed results can be copied back to the host.

By combining the persistent task queue with a dual-stream transfer strategy, the scheduler achieves efficient utilization of both memory transfer engines and GPU compute resources. The result is a pipeline where host-to-device transfers, device computation, and device-to-host transfers execute in parallel, minimizing idle time and maximizing throughput.

4.6 GPU Block Synchronization

To utilize hardware efficiently, the persistent kernel launches multiple independent GPU blocks across the available SMs. Each block concurrently dequeues and executes tasks from the shared task queue. However, when the kernel consists of multiple blocks, concurrent access to the shared queue introduces the risk of interference between blocks. Without coordination, multiple CTAs may race for the same task, resulting in lost work, duplicated execution, or even corrupted input or output buffers.

The most critical source of contention is the global device task queue tail pointer `d_tail`, which identifies the next task to execute. Since all blocks of the persistent kernel on the device update this shared variable, race conditions may cause two blocks to claim the same task, while others may skip tasks entirely. To guarantee correctness, the dequeue operation must therefore be synchronized.

This synchronization is achieved using atomic device instructions, which enforce mutual exclusion when updating shared memory locations. The device function `dequeue` in the following block demonstrates the mechanism:

```
1  __device__ int dequeue(volatile mailbox_elem_t * from_device){
2
3    int old_d_tail = d_tail;
4    unsigned int next = (old_d_tail + 1) % WORK_QUEUE_LENGTH;
5    int terminate = 0;
6
7    if(threadIdx.x == 0 && threadIdx.y == 0) {
8
```

```
 9        int prev_state = atomicCAS(&d_task_queue[old_d_tail].executing, 2, 1);
10
11    if (prev_state != 2){
12      terminate = 1;
13    }
14    else {
15      int updated_idx = atomicCAS(&d_tail, old_d_tail, next);
16    }
17
18  }
19  __syncthreads();
20  if(terminate) {
21    return terminate;
22  }
23
24  bool execution = execute(d_task_queue[old_d_tail]);
25
26  d_task_queue[old_d_tail].executing = 0;
27
28  DeviceWriteMyMailboxFrom(THREAD_FINISHED);
29
30  return 1;
31 }
```

Listing 4.1: Synchronized Block Execution of Tasks

The device function `dequeue` is responsible for ensuring the correct execution of tasks by individual CTAs within the shared task queue. Its primary role is to guarantee that each task is executed exactly once, and that no two thread blocks attempt to process the same task concurrently.

At the core of this mechanism lies the global queue pointer `d_tail`, which identifies the next task to be executed. Since `d_tail` is shared across all CTAs, it is constantly updated as blocks dequeue and complete tasks. To prevent inconsistencies caused by concurrent updates, the current value of `d_tail` is first copied into a block local variable `old_d_tail`. This snapshot ensures that the block works with a stable reference to the task index, even if other blocks advance the global pointer in parallel.

Once the local index has been secured, a designated thread within the block attempts to claim ownership of the task using an atomic compare and swap, `atomicCAS`. If the operation succeeds, the block has exclusive rights to execute the task, and the global pointer `d_tail` is atomically advanced to the following task index. If the claim fails, it means another block has already taken the task, and the current block terminates early.

By following this procedure, the `dequeue` function ensures that:

- Each task is mapped to a single thread block only.

- No task is executed more than once.

- Updates to the shared queue pointer remain consistent across all CTAs.

Through the combined use of atomic operations and local snapshots of global state, the system maintains correctness even under highly concurrent execution across multiple streaming multiprocessors.

4.7 Further Implementation Considerations

4.7.1 Serialization of Data for Memory Copies

Task memory, whether written or read from memory, first needs to be either serialized or deserialized, which requires manual GPU kernel wrappers. When a task is selected for execution, its parameters are deserialized from the input buffer, converted into an internal representation, and used to invoke the corresponding device function. After execution, the results of the computation are serialized back into the output buffer, allowing them to be transferred to the host once the task is complete.

This serialization/deserialization process acts as a bridge between the host-managed task queue and the device-executed kernels. By keeping inputs and outputs in a raw, buffer-based format, the system avoids the overhead of allocating separate memory for each task and instead reuses the preallocated memory partitions. At the same time, serialization ensures that heterogeneous tasks with different argument types can be uniformly stored and scheduled through the exact queue mechanism.

In practice, each task type requires a lightweight wrapper kernel responsible for unpacking its arguments, invoking the correct device function, and packing the results back into the buffer. While this introduces some additional development effort, it makes the overall system highly extensible: new task types can be supported simply by defining the corresponding wrapper without modifying the scheduler or memory manager.

4.7.2 Variable Launch Configurations

One of the weaknesses of persistent threads is the inability to change the launch configurations. Standard GPU kernels specify the thread configuration and grid layout of GPU threads executing the code, as well as their physical placement within the architecture. The GPU automatically decides the execution placement of the kernels from the Gigathread Engine during the launch of GPU code from the host. As the

persistent threads are already launched at program start, the configuration remains the same throughout the lifetime of the persistent thread. Therefore, these persistent threads can not support variable launch configurations at runtime without terminating the kernel and restarting a new kernel with different launch configurations. However, multiple persistent kernels can be started with various kernel launch configurations, each with a varying queue of tasks. Through code refactoring, the kernels can also be adapted to the existing GPU thread block organization. Unfortunately, for the same reason, tasks extending across multiple GPU thread blocks need to be divided into multiple tasks.

5 Experiments and Evaluation

The evaluation focuses on measuring the model's effectiveness in meeting the stated objectives as defined in Chapter 1. In particular, it examines the reduced scheduling latency and kernel overhead when invoking kernel functions. The results highlight both the benefits of the approach and its limitations.

5.1 Limitation in Memory Transfers during Evaluation

During testing, the evaluation revealed nondeterministic behavior in the execution results. In some runs, the kernels scheduled did not use the correct input matrices but instead used uninitialized memory, resulting in output matrices containing only zeros. Under further testing, this issue never occurred when debugging print statements were inserted, suggesting that the nondeterministic issue was timing-related. In particular, the input arguments were being processed before they were available. While this issue was not addressed in the current implementation, documenting it is important to understand the limitations and potential challenges associated with using persistent GPU threads for real-time workloads.

Given that the print statements ensured correct execution and the values inside the kernel functions were observed to be incorrect, the issue lies with the enqueuing of tasks to the GPU. The task enqueuing process first copies memory, then copies the task into the respective buffers within the same stream, and only afterward notifies the GPU of the new functions. Therefore, the memory should be consistent and correct before the task is even in the buffer to be executed. However, the GPU periodically polls for new tasks after a deadline to avoid missing updates if the CPU enqueues a task while the message flag is reset. When this polling occurs as the task is being copied into the memory buffer, it is possible that a memory copy is completed only partially, resulting in correct execution values but an incorrect offset. This can lead to incomplete or invalid matrices being processed.

This hypothesis remains unproven; however, in cases with incorrect values, kernel execution times are significantly reduced, which can lead to incomplete analysis of the results. Therefore, the evaluation in the following graphs contains only the execution runs that correctly computed all values within the task queue.

5.2 Experimental Setup

To assess the effectiveness of persistent threads in reducing scheduling latency, the solution was tested against a baseline implementation using a matrix multiplication benchmark. For this analysis, a simple random matrix multiplication task was repeatedly generated and scheduled 32 times using the persistent thread implementation. The measured latency was then compared to a straightforward kernel invocation. All experiments were conducted on an NVIDIA Tesla V100 GPU running CUDA version 11.8.

5.2.1 Matrix Multiplications for Performance Testing

The current execution model requires manual task wrappers for scheduling and executing GPU tasks, which adds complexity when supporting a variety of functions. These wrappers reduce the overhead of GPU memory allocation, but require serialization and deserialization of the data during execution. To keep testing manageable, this thesis focuses on matrix multiplications as a representative example to demonstrate the solution's capabilities.

In particular, the benchmark uses matrix multiplications of a fixed size, 16 x 16, chosen due to the inflexibility of changing configurations during the kernel's lifetime. Typically, the programmer specifies kernel thread configurations at launch; however, this is not possible with persistent threads. Their configuration is fixed at launch and cannot be changed during execution. In practice, such values need to be determined by profiling the workload and implementing them manually. Since performance improves when the number of threads matches or exceeds the number of elements in the solution matrix, this implementation uses a matrix with a 16×16 grid dimensionality.

5.2.2 Simulated Workload for Testing

Because the benchmark and persistent kernel implementations differ in their application goals, a simulated workload was designed for testing purposes. In these tests, the persistent kernel task schedules 32 random matrix multiplication tasks to the GPU and then immediately begins waiting for results. As the tasks are being scheduled, the GPU concurrently executes already enqueued tasks. After completing individual tasks, the host is notified via the mailbox message. By contrast, the matrix multiplication benchmark executes tasks serially in an allocation, memory copy, kernel launch, and memory copy back loop.
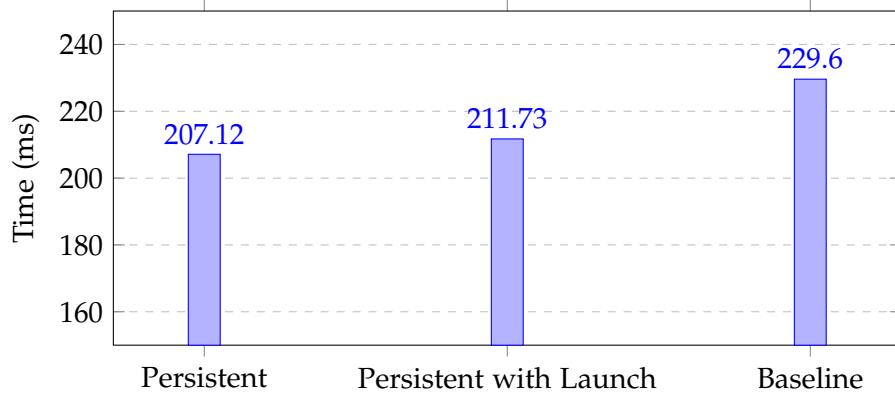
5.2.3 Latency Measurements

Figure 5.1: Small Kernel and Memory Transfer Tasks

Figure 5.1 presents the first experiment, comparing task scheduling using the GPU persistent kernel against the baseline model. Here, each implementation executes 32 16x16 matrix multiplication tasks scheduled to the GPU. The matrices for testing are generated at runtime and compared to the correct CPU implementation using the same functions.

Across 10,000 runs, the persistent kernel averaged achieved an execution time of 207.12 ms, compared to 229.60 ms for the baseline model. This demonstrates a clear performance advantage for the persistent kernel in cases where small kernels are used with small memory transfers. However, the measurements exhibited high variability, with execution times differing by as much as ± 50 ms between runs, even when averaged over 10,000 executions. Despite this variability, the persistent kernel consistently outperformed the baseline in trials.

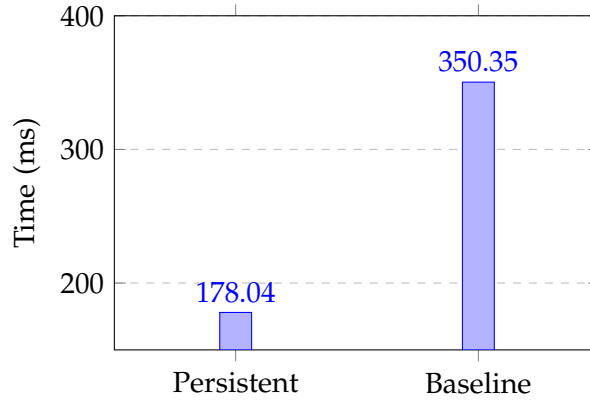### 5.2.4 Latency Measurements with Large Memory Transfers

Figure 5.2: Small Kernel and Larger Memory Transfer Tasks

In this secondary evaluation, the GPU executes the same task as before but with an additional 1 MiB of memory transferred to and from each task. Interestingly, the persistent thread implementation ran slightly faster than in the previous test, while the baseline control became significantly slower. For reference, the persistent kernel completed in 178 ms, compared to 350 ms for the non-persistent version. This unexpected outcome prompted further testing with a larger input size of 10 MiB for the persistent kernel implementation, which completed in 623.915ms.

The performance improvement is likely attributable to the underlying memory management strategy. To conserve GPU resources, memory buffers are partitioned logically using offsets. When tasks use very small memory frames, their data is placed closely together in memory. During execution, this proximity can lead to contention: tasks from memory locations adjacent to those being written to by other tasks may experience reduced performance as memory operations are serialized. By increasing the memory size transferred per task, the allocated regions are spaced farther apart, reducing contention and allowing the kernel to execute more efficiently.

5.3 Multiple GPU Blocks

Unfortunately, particularly in the case of running multiple GPU blocks, the memory copying errors were observed with a much higher frequency. As the GPU blocks were polling more frequently and able to consume tasks more quickly, they had faulty, uninitialized parameters. The results from this test failed too often to support any significant data.

6 Conclusion and Future Work

This work presents a GPU persistent thread model, designed both as a foundation for future research in scheduling and as a way to improve the efficiency of executing multiple tasks on the GPU by reducing kernel launch overhead. The model supports application-specific fine-tuning, with particular focus on managing hardware resources. In this approach, the persistent kernel runs throughout the application's lifetime, dynamically processing incoming tasks streamed to the GPU and avoiding repeated kernel launches.

The implementation meets the objectives established for developing a real-time GPU scheduler. In particular, it demonstrates that persistent threads reduce scheduling overhead and improve latency. Furthermore, Chapter 4 outlines how priorities and coroutines could be integrated into this persistent thread model. Overall, this work enhances understanding of real-time GPU scheduling, CUDA architecture, and GPU execution and programming models, providing a foundation for future improvements and research in high-performance real-time GPU computing.

6.1 Limitations

6.1.1 Input Parameter Memory Limitation

The primary limitation observed is the non-deterministic results discussed in Chapter 5. This behavior arises from the way tasks are enqueued. While the GPU should manage tasks to provide support for fine-grained scheduling, it must also ensure data consistency. In this implementation, the GPU needs to decouple memory transfers and context information from the actual execution logic. Once the transfer and context information has been transferred, then the action can be executed.

This could be addressed in several ways to synchronize the GPU with the CPU further. Currently, the synchronization mechanism of GPU mailboxes only supports a single mailbox. Instead, the implementation could support a mailbox per task element or one per SM, allowing GPU CTA blocks to constantly synchronize on these tasks, rather than on the tail of the queue. Furthermore, this approach enables the GPU to assign tasks to specific partitions, thereby reducing interference. Another approach is to introduce a separate execution buffer that only allows task execution after all

necessary data has been copied and made stream-coherent. Lastly, the GPU could track the number of tasks waiting for execution using a synchronization method, which is only updated after tasks are coherent.

6.1.2 Persistent Kernel Limitations

The persistent kernel scheduler is intended to work in harmony with the internal hardware scheduler. Conflicts between the different schedulers can negatively influence the performance or result in a failed kernel launch. For example, in this thesis, the initial evaluation was performed on larger matrices, which worked well for benchmarking the implementation but failed to execute on the GPU persistent thread implementation due to register pressure. Furthermore, thread blocks can be inoptimally managed and scheduled by the user level scheduler to resource contented SMs. In such cases, theinternal handler would have been better. Consider the case of scheduling two tasks that are heavy in shared memory to the same SM, which would lead to contention and reduced performance. Attempting to schedule tasks manually requires a detailed understanding of both hardware profiling and resource usage, making the approach substantially more complex.

When optimizing the persistent kernel implementation using NVIDIA Nsight Systems (nsys), the profiling process also became more challenging. Nsys provides detailed kernel execution latency breakdowns of kernels. However, because the persistent kernel itself executes the tasks, it becomes challenging to identify which task is currently running and to measure its latency. As a result, task latencies must be measured explicitly, either through CUDA events or with custom tracking mechanisms.

Lastly, in this implementation, the custom wrappers written for serializing and deserializing parameters introduce further complications. Supporting a wide range of tasks required significant implementation effort, and these wrappers added extra overhead. This not only slowed down the system but also increased the complexity for the user, making the approach more labor-intensive compared to an automatic solution.

6.2 Future Work

The current system does not support priority-based scheduling decisions or the use of coroutines for software preemption. Future work could build on the strategies outlined in this thesis to extend the persistent kernel implementation with these features, enabling more predictable real-time scheduling. In this case, the memory management system may need to evolve from an epoch-based scheduler to a more complex management system. Additionally, the current manual streaming of input and output parameters limits task variability and automation. Developing a generalized parameter

serialization and deserialization layer that supports arbitrary kernel arguments would increase flexibility and reduce overhead in task preparation.

Future work may also explore the use of persistent kernels for GPU partitioning and automatic task generation. By fully allocating GPU resources, individual SMs can be assigned dedicated task queues, keeping different modules independent from one another. This form of GPU partitioning reduces interference between tasks and improves isolation. Furthermore, implementing a CUDA API interception layer to capture and schedule tasks automatically would significantly enhance both the use of persistent threads in other projects and the ease of adapting these tasks to new workloads.

6.3 Conclusion

Overall, while the current implementation demonstrates the feasibility of persistent GPU threads, it remains a relatively more straightforward exploration of the concept. There is substantial opportunity for further optimization, refinement, and scaling to unlock the full potential of this approach and incorporate additional real-time scheduling strategies into the framework. Future work could build on this foundation to develop more sophisticated scheduling strategies, enhance automation, and provide better support for complex real-time workloads.

Abbreviations

GPU Graphics Processing Unit

CPU central processing unit

DSL domain specific language

GPC Graphics Processsing Cluster

TPC Texture Processing Cluster

SM Streaming Multiprocessor

CNN Convolutional Neural Network

CTA Cooperative Thread Array

HBM2 High Bandwidth Memory

D2H Device to Host

H2D Host to Device

JIT just in time

IR intermediate representation

AST abstract syntax tree

List of Figures

Bibliography

[1] https://images.nvidia.com/aem-dam/en-zz/Solutions/auto-self-driving-safety-report.pdf, Accessed: 2025-8-17.

[2] G. A. Elliott and J. H. Anderson, "Real-world constraints of gpus in real-time systems," in *Proceedings of the 2011 Workshop on Cyber-Physical Systems for the Automotive Industry (CPS-NA)*, 2011. [Online]. Available: https://www.cs.unc.edu/~anderson/papers/cpsna11.pdf.

[3] F. Zhu, L. Ma, X. Xu, D. Guo, X. Cui, and Q. Kong, "Baidu apollo auto-calibration system: An industry-level data-driven and learning-based vehicle longitude dynamic calibrating algorithm," *arXiv:1808.10134*, 2018. [Online]. Available: https://arxiv.org/abs/1808.10134.

[4] Baidu Apollo Team, *Apollo: Open source autonomous driving platform*, https://github.com/ApolloAuto/apollo, Accessed: 2025-04-09, 2017.

[5] 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.

[6] H. Tabani, F. Mazzocchetti, P. Benedicte, J. Abella, and F. J. Cazorla, "Performance analysis and optimization opportunities for NVIDIA automotive GPUs," Apr. 2021. arXiv: 2104.07735 [cs.AR].

[7] Y. Wang, C. Liu, D. Wong, and H. Kim, *Unleashing the power of preemptive priority-based scheduling for real-time gpu tasks*, 2024. arXiv: 2401.16529 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2401.16529.

[8] *LightKer*, en.

[9] 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.

[10] G. M. Hoffmann, C. J. Tomlin, M. Montemerlo, and S. Thrun, *Autonomous automobile trajectory tracking for off-road driving: Controller design, experimental validation and racing*, `https://ai.stanford.edu/~gabeh/papers/hoffmann_stanley_control07.pdf`, Accessed: 2025-8-17.

[11] M. Bojarski, D. D. 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*, 2016. arXiv: `1604.07316 [cs.CV]`. [Online]. Available: `https://arxiv.org/abs/1604.07316`.

[12] 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.

[13] 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]`.

[14] 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.

[15] `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, Accessed: 2025-8-17.

[16] 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`.

[17] M. Wu, X. Cheng, S. Liu, C. Shi, J. Ji, K. Ao, P. Velliengiri, X. Miao, O. Padon, and Z. Jia, *Mirage: A multi-level superoptimizer for tensor programs*, 2025. arXiv: `2405.05751 [cs.LG]`. [Online]. Available: `https://arxiv.org/abs/2405.05751`.

[18] *Halide: A language for fast, portable data-parallel computation*, en.

[19] *LuisaCompute: High-Performance rendering framework on stream architectures*, en.

[20] *LuisaCompute-coroutine: LuisaCompute with coroutine support (experimental)*, en.

[21] A. Gheibi-Fetrat, A. Ahmadi-Tonekaboni, F. Koohi-Ronaghi, P. Hajipour, S. Babayan-Vanestan, F. Fotouhi, E. Mortazavian-Farsani, P. Khajehpour-Dezfouli, S. Safari, S. Hessabi, and H. Sarbazi-Azad, *Rtgpu: Real-time computing with graphics processing units*, 2025. arXiv: `2507.06069 [cs.AR]`. [Online]. Available: `https://arxiv.org/abs/2507.06069`.

[22] R. Li, T. Hu, X. Jiang, L. Li, W. Xing, Q. Deng, and N. Guan, "Rosgm: A real-time gpu management framework with plug-in policies for ros 2," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2023, pp. 93–105. DOI: 10.1109/RTAS58335.2023.00015.