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

Acknowledgments

Abstract

Contents

1 Introduction

1.1 Motivation

Autonomous driving systems place stringent demands on computational performance, predictability and safety. 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. While this thesis does not implement the proposed approach directly within an autonomous driving stack, the challenges of meeting real time requirements in autonomous systems, particularly in GPU workloads, serve as the primary motivation for this research.

1.1.1 Evolution of Autonomous Driving Architectures

Historically, early autonomous driving systems addressed the real time requirements using a distributed architecture. In this approach, major functional modules, such as perception, localization, planning, and control, were mapped to seperate compute units, which together formed a processing pipeline **6809196**. 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 at the expense of high hardware cost and increased system complexity.

The rise of increasingly powerful GPUs has enabled a shift toward centralized computing in order to simplify the hardware and complexity while reducing costs. In this centralized architecture, all core driving modules share a common compute node consisting of a CPU-GPU system. The compute node integrates a CPU for task scheduling and system control, complemented by a GPU optimized for compute intensive workloads. 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 simultaneously responsible for all processing tasks, too many simultaneous scheduled tasks can lead to delays in execution time. Typical real time system solutions based on a CPU architecture, ensure system safety under contention by preempting non critical tasks. Tasks requiring high responsiveness can thus be directly executed after preemption of the resident threads and processes. This preemption is supported both natively at the OS and user space levels on the CPU, but not the GPU.

Modern GPUs, are unable to natively support real time programming models and instead rely on a proprietary hardware scheduler that restricts the programmer's scheduling control. The scheduler in particular is optimized for throughput rather than deterministic execution, which is required by real time systems. 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 to enforce strict task priorities makes a native utilization of GPUs difficult for safety critical, real time workloads. Particularaly in the context of autonomous driving, the repercussions of latencies pose a safety concern.

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. Persistent threads are specialized kernels that are launched at the start of the application and remain active throughout the lifetime of the system. 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 support the implementation of GPU coroutines for predictable, low latency scheduling on real time systems?

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

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

- **GPU Device Memory Management:** Efficient allocation, deallocation, and usage of GPU memory to ensure low latency access and avoid contention between tasks.
- **GPU Task Enqueuing and Dequeuing:** Mechanisms for submitting tasks to the GPU and retrieving results asynchronously.
- **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 feasability 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 originally designed for graphics rendering, relied on a complex compiler based architecture with little documentation, and required a separate build process. Combined with my 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 measuring overheads was found from which parts were taken to build a fully functional system. This new scheduler serves as a proof of concept foundation which provides long running GPU kernels to receive and execute tasks efficiently on which coroutines can later be implemented for real time systems. Furthermore it provides the framework on which GPU coroutines can yield and enforce 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 a number of persistent thread components to enable the efficient execution of GPU code within persistent threads.

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.

Memory Management System

The execution context for each task includes its associated memory allocations. To reduce the overhead of repeated allocations and deallocations, memory is mapped into a running epoch buffer of preallocated memory assigned before the launch of the persistent threads. Tasks then operate within this preassigned memory region, eliminating the latency costs of frequent device memory management operations. This system further provides a logical partition of input and output buffers in order to reduce data interdependencies.

Concurrent Memory and Execution models

To support concurrent memory transfers between the host and device, as well as simultaneous kernel execution, the system leverages CUDA stream manipulation. This reduces interdependencies between data transfers and execution tasks, enabling higher overlap and better utilization of GPU resources.

GPU Task Coordination and Synchronization

The scheduler enables multiple thread blocks to execute distinct tasks concurrently, ensuring that no two blocks perform the same task simultaneously Furthermore, this system allows the host to efficiently schedule and queue tasks to any available thread block, maintaining high utilization of the GPU. By enforcing exclusive task execution, the mechanism prevents race conditions and ensures correctness across all kernels.

2 Background

This background chapter begins by examining the role of GPUs in real time sytems, focusing on the architectural and programming constraints that influence scheduling and performance. The section then devlops into an analysis of real time programming models for GPUs with consideration of persistent threads and coroutines as implemented into the autonomous driving system Apollo. These coroutines are further evaluated through LuisaCompute's automatic GPU coroutine generator. The goal is to provide 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 behaviour. **10155700** These constraints are expressed in terms of the systems ability to meet task deadlines, categorized as either soft or hard. Hard deadlines are critical to the safety of the system. Missing these deadlines results in potential system failure or unsafe conditions, such as collision avoidance or brake activation. To ensure the safety of the system, these deadlines need to be prioritized over the less critical soft deadlines. Soft deadlines, for example route planning or navigation updates, are deadlines that should be ensured, but do not lead to system failure if not met. Delays here may lead to suboptimal paths or low display responsiveness, but do not risk safety. Real time systems ensure system safety by preempting soft deadline tasks to prioritize hard deadline, 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, used 6 Pentium processors among which tasks were divided. The CPU centric design along with resource partitioning, allowed the system to meet real time requirements using existing strategies, similar to how Apollo manages the runtime CPU environment today.

These systems used execution tasks such as sensor fusion, localization, and path planning, which were still CPU friendly. However, as CNNs were increasingly used due to their high accuracy in perception tasks, CPUs struggled to continue performantly processing data in real time. As CNNs quickly became the dominant approach towards perception based tasks and the complexity of these models continued growing, GPUs began to be increasingly integrated into these systems. One of the earliest projects using GPUs for autonomous driving, was NVIDIA in 2015, which used a GPU to train a CNN that could steer a car end to end from raw camera input. 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. 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 the vast increase in throughput over CPUs, 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. The architecture thus adapted to accomodate an increasing number of threads, enabling support for larger graphics matrices. These tasks required very little control overhead, which allowed the GPU threads to be very simple.

Althought the CPU offers some parallelism, it is primarily optimized for sequential tasks, relying heavily on individual thread performance for tasks that cannot be parallelized. In order to achieve greater performance on these workloads, CPUs dedicate a "significant portion of transistors to non computational tasks", which reduces overall throughput. These sequential tasks are greatly improved through complex control logic, such as prefetching, branch prediction, speculative execution and out of order execution, which all allow CPU greater non parallel performance gains. As a result, CPUs generally have a higher clock rate than GPUs, which forgo the control overhead in favor of increasing arithmetic intensity **Owens2007-kp**.

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

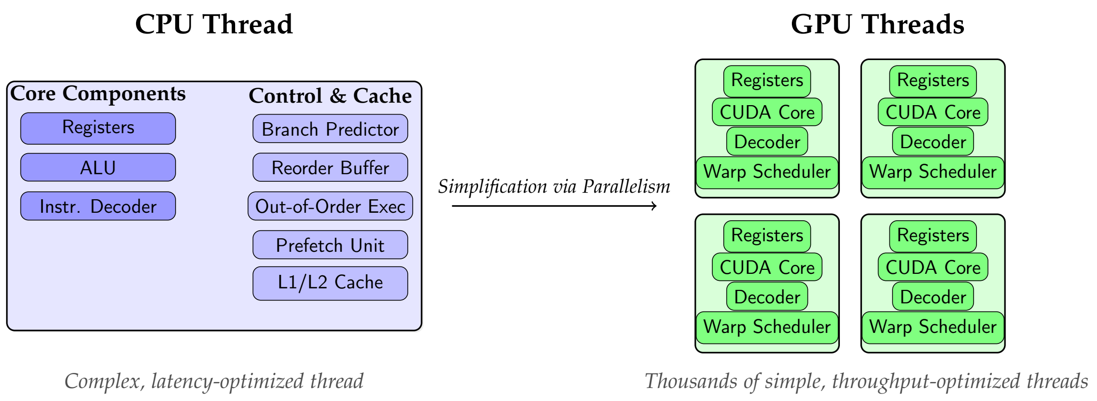

Figure 2.1: CPU vs GPU Thread Architecture

Although core components are named differently, both CPU and GPU threads work similarly with an instruction decoder, registers and an arithmetic unit. The differences arise when trying to maximize a single control flow. In contrast to the CPU, the GPU threads execute instructions 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, GPU 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 outperforms the single threaded GPU applications, as seen by the following comparison of single threaded matrix multiplication in Figure 2.2 and Figure 2.3. These figures show the comparison of executing a matrix multiplication using only one GPU thread versus one CPU thread on a matrix multiplication task.

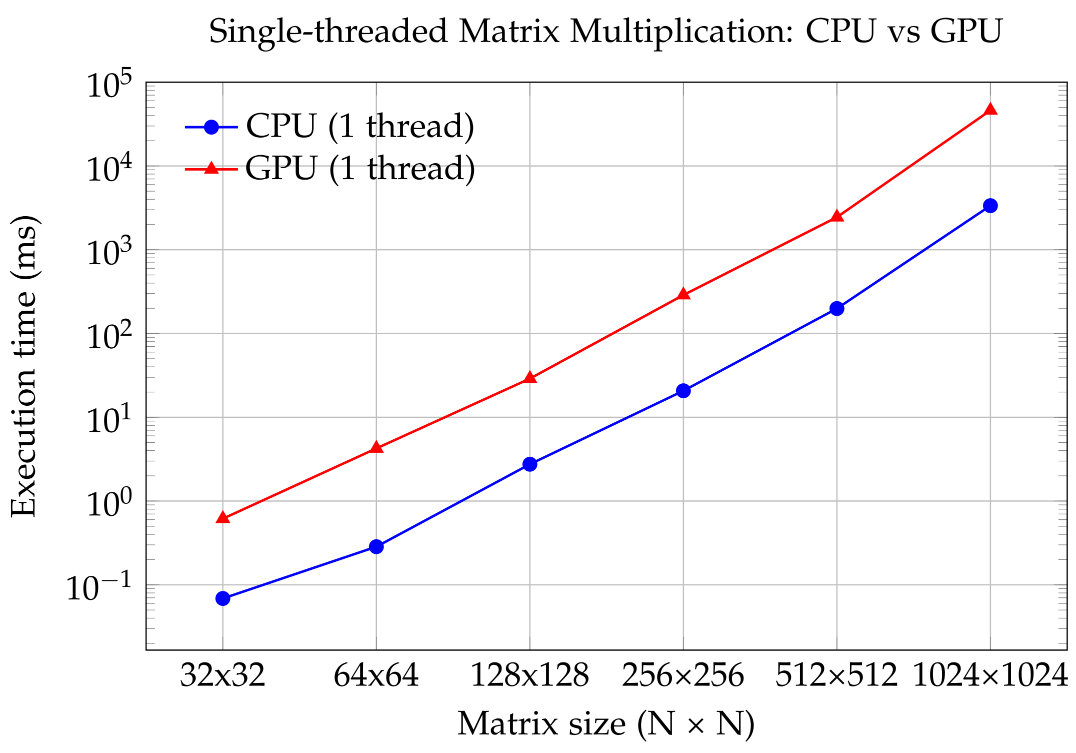

Figure 2.2: Single threaded Matrix Multiplication Execution between CPUs and GPUs averaged over 10 executions

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, GPUs struggle in applications that fail to utilize the architectural parallelism. For each of the matrices tested, the CPU is on average around 13 times faster than the GPU. 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 is significantly delayed.

For the context of this thesis, any scheduler running on the GPU employing complex control logic will incur significant latency overhead. In general, scheduling decision, task management, and resource assignment, must follow a sequential control flow, as these processes lack inherent parallelism. Consequently, efficient GPU task management often relies on a CPU side scheduler to handle task queuing and resource allocation, with the GPU side component primarily focused on dispatching tasks to available compute units.

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. At a high level, the GPU features 5,120 CUDA cores, 640 Tensor Cores, delivering up to 7 TFLOPS of double precision performance. The system has 16 GB of VRAM and provides 900GB/s of bandwidth between on-chip memory and device memory. The GPU interfaces with the host system over PCIe, allowing a theoretical maximum of 16GB/s for transfers between CPU memory and VRAM.

The performance of GPU is often limited by the disparity between compute throughput and memory transfer rates. While the GPU can execute arithmetic operations at extremely high speed, fetching and storing data from VRAM is comparatively slower, which can stall execution if memory accesses is not optimized. Even slower are the memory transfers between host and VRAM and must be minimized to avoid further bottlenecks or stalls. Understanding this imbalance is fundamental to design of efficient scheduling and memory access patterns, which are essential to fully exploit the device capabilities. 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, blocks of threads executing the same instruction code, which 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 these different threads sequentially with respect to one another. This serialization stalls part of the warp using masks, which disable the write back of the compute units. In effect, thread divergence, reduces the effective parallelism and degrades performance.

SM Architecture

The Tesla V100 GPU SM architecture, contains 4 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 the 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.

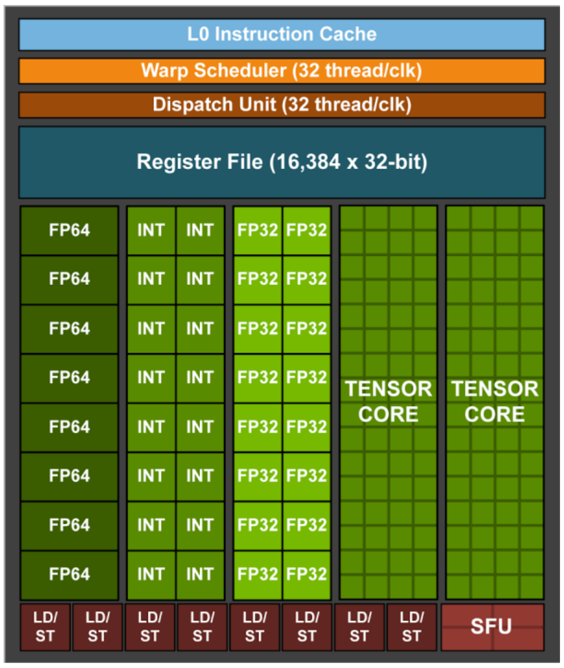
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
TENSOR CORE
LD/ ST
SFU

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 appropiate 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. Beyond, the warp scheduler and dispatcher, each processing partition contains a number of execution units. In particular, 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.

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, while 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 allow 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). SM threads all share the same on-chip L1 data and shared memory cache and the 256KiB 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 will fail. Furthermore, if multiple thread blocks with high shared memory demands get scheduled to the same SM, they will saturate the L1 data and shared memory cache and force frequent evictions and write backs to global memory. These hardware limits must be carefully considered when mapping tasks to SMs.

The built-in hardware scheduler normally either handles these constraints or fails the kernel launch. Implementing a custom scheduler on top of the hardware scheduler, requires consideration of these GPU limitations. Forcing new scheduling behavior may conflict with or be constrained by the hardware scheduler's own mechanisms.

Scheduling and Task Mapping

The specific mapping of tasks to SM, TPC, and GPC 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 a multitude of factors: hardware resources, parallelism, priorities, and dependencies. Similarly, 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 without interruption.

The CTAs is entirely managed by the SM on which it is currently executing. As shown previously 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 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 accumulate answers in global memory, skipping the high memory latency accesses of coordinating synchronous work.

2.4 GPU Programming using the CUDA API

NVIDIA GPUs are intended to be used as computational accelerators for a host system, which manages and schedules tasks using CUDA. In this model, the GPU acts as a standalone processor with its own independent memory and execution pipelines. For tasks to be scheduled on the GPU, the host process must first launch the process using the CUDA API. The CUDA API is an extension of C++, which allows the CPU to interact with the GPU. Using CUDA, the host invokes device functions, called kernels, by specifying the number of threads and the memory parameters. These kernel calls are asynchronous, meaning that once the host issues the command, the GPU executes it independently, while the CPU can continue with other work or synchronize 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 organization into dimensions. 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 Memory and Bandwidth Considerations

Both the GPU and CPU maintain distinct memory regions for different hardware architectural and performance reasons. CPU memory is optimized for low latency access to support serial and branch heavy workloads, while GPU memory is optimized for extremely high bandwidth to sustain thousands of parallel threads, accepting higher latency in exchange. These memories are connected by an interconnect such as PCIe or NVLink, whose bandwidth is far lower than that of the GPU's local memory, making constant access to host memory impractical without significant performance loss. By keeping data local, each processor can operate at full efficiency, and the GPU can execute kernels independently without frequent synchronization with the CPU. This separation is thus a deliberate design choice to match each processor's optimization goals, avoid bottlenecks from the limited interconnect speed, and enable independent execution.

As the memory regions are distinct, for the GPU to support the CPU with tasks, the CPU needs to use CUDA to allocate and send data to the GPU memory. When transferring 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 automatically, host arrays allocated directly in pinned memory using `cudaMallocHost()` or `cudaHostAlloc()` skip the added step of pinning memory and enable faster transfers. Particularity 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.

2.4.3 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 in the grid. 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.

```
__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 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.

2.4.5 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.

2.5 GPU Programming Models for Real Time Systems

2.5.1 Persistent Threads

Persistent GPU threads provide the programmer enhanced control over hardware scheduling and reduces scheduling overhead. In particular, this increased control enables the introduction of advanced GPU scheduling strategies, such as coroutines or mega kernels. By running persistent threads, kernel configurations can be loaded ahead of runtime, minimizing runtime overhead during execution.

To maximize hardware utilization, interactions between the host and GPU should be minimized. For each new scheduled task, the GPU must typically allocate memory, copy data into buffers, allocate SM resources, load the kernel, set up kernel parameters such as shared memory and thread configurations, execute the kernel, and finally copy results back and free resources. This sequence introduces significant overhead for every API call. In systems with recurring or periodic tasks, such as autonomous driving, this overhead becomes particularly costly, as the same configurations are repeatedly allocated and freed.

Persistent kernels address this problem by reducing redundant operations. Instead of repeatedly allocating memory and system resources, long running thread remain preconfigured on the GPU. 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.

2.5.2 Megakernels

Megakernels are very similar to persistent kernels, but instead of preallocating GPU resources, they employ kernel fusion. Individual kernels are fused together to form a megakernel, which contains either the entire device program or multiple smaller kernels. Practically, this method is similar to a persistent kernel, by optimizing GPU scheduling by reducing API overhead. The megakernel is then essentially a persistent kernel, which is launched at the start of the program, executes all of the system tasks, before returning.

2.5.3 Coroutines

Coroutines are a form of asynchronous programming that enable cooperative multitasking between functions. Unlike thread- or process-level context switches, which involve greater overhead, coroutines maintain only a function-level context, allowing fast and lightweight task switching. 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 **Zheng2022LuisaRender**. 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

Apollo the autonomous driving system, already employs the coroutines for the runtime CPU environment. These coroutines are implemented using the native x86 calling conventions to save continuations using the stack. To switch a function, the CPU merely needs to save its state and then jump to the new function instruction address.

In x86 calling convention, when a function is called, the CPU pushes the address of the next instruction after the function onto the stack and jumps to the instruction referenced by the call instruction. The called function accesses its variables via the stack and registers. Here, registers are categorized into two groups: volatile, caller saved and non-volatile, callee saved. Volatile registers may be freely modified by the callee without restoration, whereas callee saved registers must be preserved and restored before the function returns.

When the function executes a return instruction, the CPU pops the return address off the stack and continues execution from that point. To yield a function, the minimum context that must be saved and restored includes the callee saved registers, since these are guaranteed to be preserved across function calls. Additional state, such as local variables or other registers, can be manually saved to the stack to be restored later when the coroutine resumes.

Consider the following CPU coroutine code taken from the Apollo project.

```
1  ctx_swap:
2      pushq %rdi
3      pushq %r12
4      pushq %r13
5      pushq %r14
6      pushq %r15
7      pushq %rbx
8      pushq %rbp
9      movq %rsp, (%rdi)
10
11     movq (%rsi), %rsp
12     popq %rbp
13     popq %rbx
14     popq %r15
15     popq %r14
16     popq %r13
17     popq %r12
18     popq %rdi
19     ret
```

Listing 2.2: CPU Coroutine

The CPU coroutine implementation uses the context switching function, ctx_swap, which swaps the continuations by saving the current execution context and loading a new one. As previously stated, standard calling conventions uses registers to pass the function parameters to the function. In this case, this function accepts two parameters, %rdi and %rsp, each representing an addresses used to store and retrieve the coroutine continuations. %rdi is used by the first half of the function to store the current execution state, while %rsi is used in the second half to restore and load the coroutine continuation.

The first section of ctx_swap, up to line 10, is responsible for saving the current continuation. This is achieved by pushing all callee saved registers onto the current stack, preserving the processor state. To resume the continuation later, the memory location of the continuation is stored in %rdi, by copying the current stack pointer into the register. This value allows the function to restore the original context during subsequent switches.

The second section loads the new coroutine context, by reversing the first sections actions to saving the state. The stack pointer is updated with the new continuation, such that all the saved registers can be retrieved and execution can resume. Once within the correct stack frame, all callee saved registers are restored in the reverse order of their saving, effectively loading the processor registers with the new coroutine's state. The callee saved register states are thus preserved across coroutine function calls, with further saved variables easily accessible as local variables on the same stack frame.

In comparison to traditional context switches involving processes or threads, coroutine context switching via ctx_swap operates with significantly lower overhead, resulting in a faster execution. Furthermore, these continuations benefit from the user explicitely defining the suspension points. User defined suspension points typically involve much smaller contexts than preemption points, as they align more closely with convenient locations within the executing program. Consequently, coroutine contexts can often be kept minimal by leveraging only these carefully chosen points that require smaller saved states.

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 by the system of programmer throughout the duration of its lifetime. The kernel function will saturate 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. Given these restrictions, the kernel must manually yeild execution using a user space sheduler.

Inlining Attempting to implement the CPU style coroutines using the ctx_swap function does not work, given that the GPU handles function calls differently than the GPU. CPU function calls store the call stack within the stack, by pushing instruction pointers that can be popped with ret. The GPU does not use these, but instead saves stack pressure by aggressively inlining function calls. Inlining function calls allows the GPU to reduce overhead when beginning a new function as that function code is already readily available and removes the need for a cpu style stack frame.

In particular, GPUs strongly optimize away from call stacks and provide only minimal support for them. Originally, CUDA did not allow recursive functions and only with around CUDA 5.0 did they support them. This support has to be manually implemented using the nvcc flag, lstinline[language=cuda]'-rdc=true'. This recursion comes with limitations of restricted stack size and added performance overhead. Attempting to implement deep call stacks leads to exponentially large instruction Unfortunately for tasks dependent on deep call stacks such as recursion, this leads to exponentially large instruction memory.

Persistent Threads to support Coroutines

Given that call stacks are not effectively or efficiently supported in CUDA, the GPU needs to provide a user level scheduling mechanism to control the execution decisions. Due to the nature of kernels executing until completion, the GPU coroutine scheduler needs to be based on a persistent kernel, which can schedule coroutines. 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 computationally expensive, so the coroutine contexts need to be managed locally and saved in global memory to free up limited SM resources for new tasks.

3 Related Work

The increasing use of GPUs in real time systems has led to the development of custom programming models both reducing kernel launch overhead and improving predictable GPU scheduling. Applications in domains such as robotics, autonomous systems, and scientific computing increasingly adopt these models to minimize execution latency, improve resource utilization, and ensure timing determinism. These applications can be generally divided into three categories: compiler driven frameworks, runtime scheduling frameworks, and manual implementations.

3.1 Compiler Driven Frameworks

Compiler driven frameworks optimize GPU workloads through automated code generation. The application first provides the user with a simple DSL, which abstracts away the low level device details. At runtime, the framework's compiler parses the DSL into an AST, which is then converted into an IR. From this intermediate representation, the compiler transforms and optimizes the code, before being compiled JIT into GPU executable device code.

One such project, *Mirage* implements these ideas to improve large language model inference, by fusing kernels into a singular megakernel. Similarily, *Halide* provides a framework to automatically make scheduling decisions for users, by decoupling the algorithm from the execution schedule. The execution schedule is then determined through compiler optimizations using autoschedulers that require minimal manual tuning. Lastly, *Luisa* is structured similar to the other projects, but offers performance benefits specifically for graphics and simulation workloads like ray tracing and rasterization. Luisa has support for acceleration strucutures, ray traversal APIs, and shader abstraction, providing developers with high level rendering code, while retaining low level performance.

Built on top of Luisa's execution model, *LuisaCompute-Coroutines* extends the framework to support coroutines built on persistent threads. Coroutines are expressed simply within the DSL using suspension points. These suspention points allow the device to preserve context and yield, allowing for fine grained scheduling. In particular this approach is efficient and leverages itself for use in real time systems due to the asynchronous coroutine programming approach.

3.2 Runtime Scheduling Frameworks

Beyond compiler based transformations, runtime based frameworks provide 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. Using a reservation based model, RT-GPU enables fine grained control over scheduling to ensure the task deadlines. Additionally, *ROSGM* is a further GPU management framework designed specifically for ROS 2 robotics systems. The ROSGM system interposes a layer to intercept the CUDA API calls to insert metadata for each GPU task. The API interception allows for custom deadlines and priorities that manage how GPU tasks are queued and issued to the device.

3.3 Manual Persistent Kernels

In addition to the other models, manual implementations support fine grained scheduling control specifically tuned to a single application. For example, in scientific computing, simulators like *HOOMD-blue* manually fuse multiple computation steps into a single persistent kernel. Furthermore, Jetson implemented GPU persistent threads to support their real time RedHawk Linux system. Unfortunately, this GPU persistent thread implementation is not open source. These implementations offer low-level control and high efficiency, but demand significant expertise in GPU programming.

3.4 Platform Integration: GPU Scheduling in Apollo

This thesis aims to integrate GPU real time scheduling into the open source autonomous driving platform Apollo. Apollo, developed by Baidu, relies on CyberRT, a real time platform to coordinate CPU task execution. CyberRT manages the different driving modules within the system and coordinates cooperative asynchronous scheduling using coroutines. The coroutine capability; however, does not extend to GPUs, which does not ensure their predictability.

Starting this thesis, I initially explored integrating LuisaCompute-Coroutines into Apollo, to provide the system with GPU coroutines to pair with the existing CyberRT CPU coroutines. Similar to the CPU coroutines, these GPU coroutines were intended to deliver the system predictable execution latencies, with the added benefits GPU persistent thread offer. However, due to several integration barriers, including incompatible build systems, sparse documentation, and time constraints, it became clear that the integrating this system into Apollo would not be feasible within the scope of this thesis.

Consequently, this work's focus shifted to a manual implementation of a GPU persistent thread scheduler using CUDA. This change allowed me the opportunity to study the low level aspects of GPU scheduling from both an architectural and programming perspective. While lacking the automation and abstraction of a compiler driven implementation, this manual approach allows for finer application specific implementations.

4 Design and Implementation Requirements

At a high level, there are four important components: the task queue, memory buffers, stream design, and gpu block synchronization. The task queue is managed, controlled, and allocated by the host and provides the arguments and tasks for the GPU to execute. The implementation of the task queue gives the programmer fine-tuned implementation oppertunities to manually design and schedule workloads. The memory buffers provide an epoch based staging area for input arguments and output results as well as persistent memory for the further extension of coroutines.

4.1 Task Management System

For persistent threads to execute kernels at runtime, a staging mechanism is required to enqueue tasks and maintain the task context necessary for execution. As discussed in Chapter 2, any task management system that relies on device side scheduling logic is inherently ineffecient due to characteristics of GPU architecture. Therefore, 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 enables better control over task odering, reduces idle time, and improves overall predictability, essential for real time workloads. Furthermore, this system should be compatible with coroutine based execution and priority scheduling in the future.

4.1.1 Task Queue Design

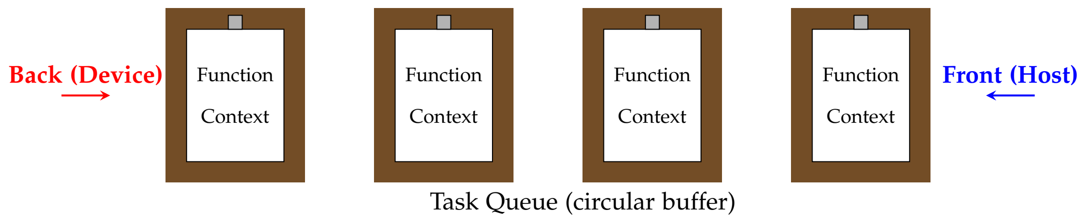

Figure 4.1: GPU Task Queue for Task Management

The architecture implemented into the persistent thread scheduler uses a loop through buffer, as a FIFO queue, in a producer-consumer architecture. As shown in Figure 4.1, each GPU task is represented as a clipboard, which contains the function context. The context contains both the information needed to execute the device function, as well as metadata for the control of the task within the queue. The host acts as a producer, enqueuing tasks into the task queue at the front, while the device consumes tasks from the back.

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

To ensure that tasks are not overwritten, the host tracks the total number of tasks currently in the system in relation to the total length of the task queue. Every time a new task is added to the queue, the tracker is incremented, unless the queue is full, which results in a failure response, similar to the native CUDA system. For the host to decrement the length tracker, the device must execute the task, notify the host, upon which the host can copy the results back. After the results are copied back, a new task can then be enqueued to the same postion in the queue.

4.1.2 Extensibility for Coroutines and Priority Scheduling

The function contexts stored in the task queue buffer were specifically designed with the future implementations of coroutines and priority scheduling in mind. By introducing additional control variables into the function context, the host system can assign tasks different priorities. Furthermore, the queue already provides space to store the continuation of the coroutine, 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. However with slight modifications, 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 accidently overwritten.

In conjuction 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 it would be necessary to define specific set points within the program and save a variable in the function context that indicates the next instruction to be executed within the kernel.

4.2 Function Context

Each task entry defines its execution context through an explicit function identifier and its associated function parameters. This entry acts exactly like a coroutine continuation, which stores the necessary state to resume execution within the function. The function id is used in conjuction with a lookup table to execute the GPU device code. The memory location for the parameters is allocated by the host as part of the enqueuing and memory management systems.

4.2.1 GPU Function Pointers

To execute new functions from the persistent threads, the task queue needs to be able to reference the specific function. Generally referencing functions on a CPU requires only the function pointer to execute the code defined at that memory location. When GPU functions are compiled, the device code lives in the GPU address space and is not accessible from the CPU. The CPU only has access to functions denoted by the `__global__` keyword, which allows the execution of GPU kernels, not enqueuing of GPU functions. In order to be able to access and run the functions specified by the CPU, the task queue supports a lookup table to map integers to specific functions. The lookup table allows the host to memcpy in function ids to the task queue when enqueuing new tasks.

4.2.2 Function Parameters

When the CPU assigns tasks to the GPU, it passes either allocated GPU memory pointers or explicit parameters. These explicit parameters then get propogated to all the individual threads executing the kernel code, resulting in greater api memory overhead. When enqueuing new tasks to the task queue, the memory has to be transfered at runtime before the device function calls.

The GPU task in the queue originally had a pointer to the allocated memory and upon recieving 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 consumes variable memory requiring further pointers to GPU memory. In order to consolidate the memory pointers, the task queue was simplified to contain only allocated memory pointers in order to automatically load kernel memory.

In this method, enqueing the GPU tasks forces the programmer to streamify the data and automatically load the memory 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 then the loading of it into the device function. Should the input memory be oversubscribed, the task then has a preallocated buffer to store any updated context for the continuation of the coroutine.

4.3 Memory Management System

The memory management system supports the task queue, while eliminating unnecessary memory allocation and deallocation overheads. By managing memory centrally, the task queue's function contexts can remain simple and flexible. This memory system is managed for the lifetime of the persistent kernel, removing the need for costly dynamic memory operations at runtime. While maintaining a memory buffer is simple, designing a runtime strategy to partition memory among tasks and reclaim it efficiently presents a more complex challenge. To visualize this, Figure 4.2 shows the memory buffer logically partitioned into epochs, with each epoch containing input argument and output result buffers for the corresponding tasks.

4.3.1 Memory Buffer Design

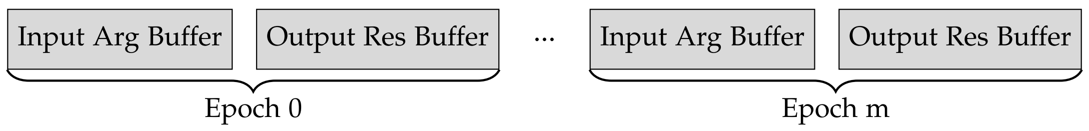

Figure 4.2: Continuous Memory Buffer Logically Partitioned

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 and output result section. The entire overhead of managing the position of task memory within the buffers is entirely managed by the host and shared with the device through the function context within the task queue.

As tasks are allocated within the epochs, eventually one of the corresponding memory buffer will overflow. When the host detects that allocating memory for a task either results in the input or output buffer overflowing, the host begins allocation in the next epoch. After fully allocating the memory for an individual buffer, that buffer remains untouched by the scheduler until the scheduler loops around the entire buffer queue and reaches the same epoch again. An epoch is free to schedule again when all results of tasks allocated within that epoch have been collected and marked as free.

The host manages these epochs, by tracking the number of tasks within each epoch and the current offset within the current epoch. To enqueue tasks input arguments are immediately loaded at the current offset within the current epoch. Tasks are only considered as free after the memory has been copied back to the host and the host marks the epoch memory as freed. When the buffer queue loops and returns, if the specific buffer parameters and memory size has been correctly set, potentially through profiling, the buffer will be free and can be reused.

4.3.2 CUDA Stream Optimization

As discussed in Chapter 2, modern GPUs feature 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, enabling overlap between data movement and computation if 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 concerns between input staging, output collection, and kernel execution.

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 proceed in parallel, minimizing idle time and maximizing throughput.

4.4 GPU Block Synchronization

In order to utilize hardware efficiently, the persistent kernel launches multiple independent GPU blocks across the available SMs. Each block concurrenlty dequeues and executes task 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:

```
__device__ int dequeue(volatile mailbox_elem_t * from_device){

  int old_d_tail = d_tail;
  unsigned int next = (old_d_tail + 1) % WORK_QUEUE_LENGTH;
  int terminate = 0;

  if(threadIdx.x == 0 && threadIdx.y == 0) {

        int prev_state = atomicCAS(&d_task_queue[old_d_tail].executing, 2, 1);

    if (prev_state != 2){
      terminate = 1;
    }
    else {
      int updated_idx = atomicCAS(&d_tail, old_d_tail, next);
    }

  }
  __syncthreads();
  if(terminate) {
    return terminate;
  }

  __syncthreads();
  bool execution = execute(d_task_queue[old_d_tail]);

  d_task_queue[old_d_tail].executing = 0;

    DeviceWriteMyMailboxFrom(THREAD_FINISHED);
  return 1;
}
```

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 next 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.

4.5 Architecture

The individual contributions explictely discussed are joined together in the following design architecture below. The architectural aspects discuss previously are the main aspects of the work of this thesis, with additional tools and synchronization methods such as the mailboxes being used from the LightKernel project.

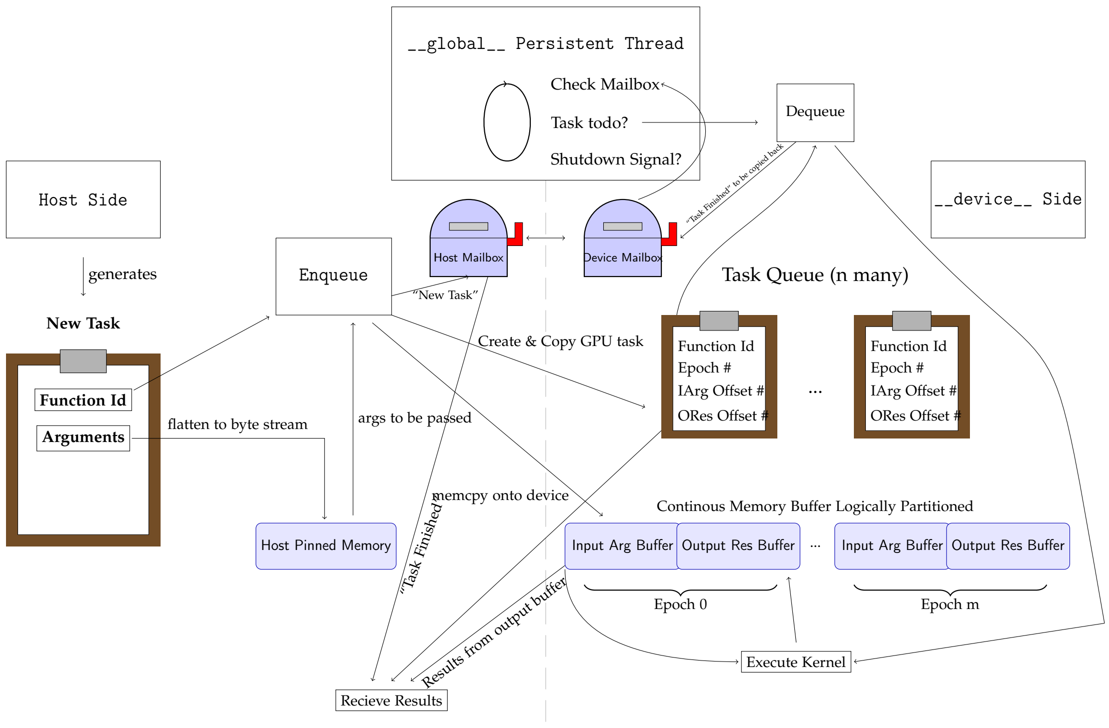

Figure 4.3: Persistent Thread Architecture implemented into LightKer

The Figure 4.3 depicts the individual components in a logical program architectural overview. The left side of the graphic shows the host side code and methods, which allow the enqueuing of tasks and launching of the persistent thread kernel. On the right hand side of the graphic, the actual device side code allocation and kernel execution is depicted which allows the processing of memory and write back of results to the memory buffers. The mailboxes are constantly enqueuing and dequeuing new tasks throughout the execution of the kernels.

4.6 Further Implementation Considerations

4.6.1 Serialization of Data for Memory Copies

Task memory 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 so they can 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 same 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.

4.6.2 Variable Launch Configurations

One of the weaknesses of persistent threads is the inabilty to change the launch configurations. Standard GPU kernel launches specify the thread configuration and grid layout of GPU threads executing the code and their physical placement in 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 different persistent kernels can be started with various kernel launch configurations, each with different task queues, or through code refactoring the kernels can be adapted to the existing GPU thread block organization.

5 Experiments and Evaluation

This chapter evaluates the proposed GPU persistent thread model with respect to the objectives of this work:

1. Reducing scheduling latency by minimizing kernel overhead.
2. Establishing a baseline platform for future research in real time GPU scheduling, serving as a foundation for a coroutine based approach.
3. Enhancing understanding of real time GPU scheduling, CUDA architecture, and GPU execution, programming, and scheduling models.

The implementation introduces a GPU persistent thread model designed as both a building block for future scheduling research and as a means to improve the efficiency of executing multiple tasks on the GPU by reducing kernel launch overhead. This persistent GPU model allows for application specific fine tuning with particular focus on hardware resource management. In this model, the persistent kernel executes throughout the lifetime of the application and dynamically processes incoming tasks streamed to the GPU, avoiding repeated kernel launches.

The evaluation focuses on measuring the model's effectiveness in meeting the stated objectives. In particular, it examines scheduling latency, kernel overhead, and suitability as a baseline for real time programming models such as coroutines. The results highlight both the benefits of the approach and its limitations, especially regarding task variability and scaling across the GPU.

5.1 Experimental Setup

In order to evaluate the effectiveness of persistent threads in reducing the scheduling latency, the solution was tested on a matrix multiplication benchmark against a baseline implementation. To test the persistent threaded implementation, a simple matrix multiplication task was repeatedly scheduled on the persistent thread implementation and compared with a simple kernel invocation code. The persistent kernel was then tested with a number of varying parameters, including memory overhead, GPU work, and number of persistent kernel blocks.

5.1.1 Matrix Multiplications for Performance Testing

The current execution model requires manual task wrappers for scheduling and executing GPU tasks, making it more complex to provide a varying range of different tasks. The manual task wrappers, designed to reduce the overhead of allocating GPU memory, are required to serialize and deserialize the allocated 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 selected matrix multiplications were of a predetermined size of 16x16, due to the constraints of the GPU persistent kernel. Normally, the programmer decides at kernel launch the configurations of threads within the kernel; however, this feature is not available for persistent threads. Persistent threads configurations are determined at launch and can not be reallocated throughout the duration of the task. For application based workloads, these values would need to be determined through profiling the workloads and manually implementing the values. In this case, matrix multiplications are significantly faster when the task has at least as many threads as the solution matrix values, which led this implementation to using a 16x16 persistent kernel launch configuration.

5.1.2 Testing Environment

Throughout the matrix multiplication performance testing, as the benchmark and the persistent kernel implementation both vary in application goals, a simulated work environment was designed. For the tests, the persistent kernel task schedules a number of tasks to the GPU and then immediately starts waiting to receive the results. As the tasks are being scheduled, the GPU already begins to execute enqueued tasks. After finishing individual tasks, the host is signaled with the message that the tasks are finished. In comparison, the matrix multiplication benchmark just executes tasks serially in an alloc, memcpy, kernel launch, memcpy loop.

In a real time persistent kernel implementation, the CPU needs to independently run tasks within the system and cannot synchronously wait on results to be streamed back. Rather, the CPU needs the results to be waiting when the it goes to check if the results have returned or not. Unfortunately, this model, when compared to any synchronous execution, will have waiting overhead. This is implemented within the test kernel as a busy waiting scheme; however, this increases the GPU traffic and slightly weakens the GPU implementation as it contests the same streams for data transfers.

5.1.3 Small Kernel and Small Memory Transfers

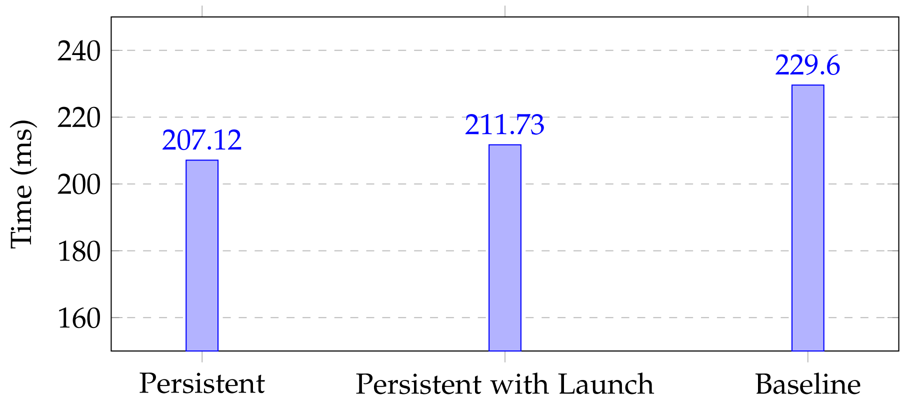

Figure 5.1: Small Kernel and Memory Transfer Tasks

Figure 5.1 shows the first experiment that was run comparing the task scheduling from the GPU persistent kernel in comparison to the baseline model. Here each implementation executes 32 tasks 16x16 matrix multiplication tasks scheduled to the GPU. The matrixes for testing are generated at runtime and compared to the correct CPU implementation using the same functios.

On average over 10,000 runs, the persistent kernel achieved an execution time of 189.165ms, while the baseline model required 229.604ms, demonstrating a clear performance benefit for this small kernel, small memory transfer case. However, the measurements exhibited high variability, with execution times differing by as much as ±50ms between runs, even averaged over 10000 different executions. Despite this noise, the persistent kernel consistently outperformed the baseline in the majority of trials.

5.1.4 Small Kernel and Larger Memory Transfers

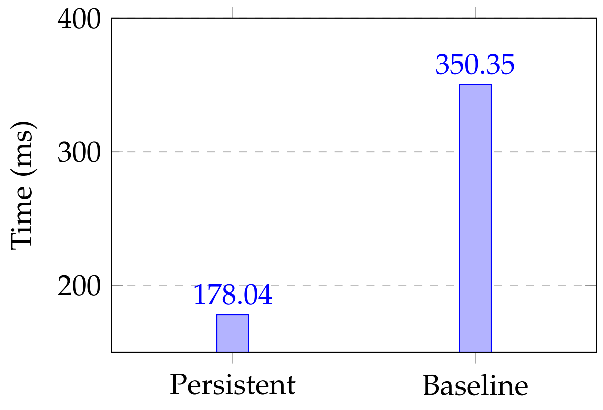

Figure 5.2: Small Kernel and Larger Memory Transfer Tasks

In this secondary evaluation, the GPU executes the same task as before but now transfers an additional 1 MiB of memory to the task and returns it. Interestingly, the persistent thread implementation runs slightly faster than in the previous test, while the baseline control becomes slower. This unexpected result prompted further testing with a larger input size of 10 MiB, which completed in 623.915ms.

The observed increase in speed is likely due to the underlying memory management strategy. To conserve valuable GPU resources, the memory buffers are logically partitioned using offsets. When tasks use very small memory frames, their data ends up placed very close together in memory. During execution, tasks that read from memory locations adjacent to others simultaneously writing to nearby regions can cause contention and reduce performance. By increasing the memory size transferred per task, these memory regions are spaced farther apart, reducing contention and allowing the kernel to run more efficiently.

This separation applies to both input and output buffers. When transferring larger amounts of data (e.g., 1 MiB or more), the serialized GPU memory transfers no longer interfere with kernel execution, leading to the improved performance observed.

5.1.5 Multiple Kernels

5.2 Profiling and Analysis with `nsys`

To better understand the execution characteristics of the implementation, `nsys` profiling was performed for both the baseline (non-persistent) and persistent-threaded approaches. In the baseline version, the profiler output is easy to interpret: each kernel launch is explicitly visible on the timeline, interleaved with host-device memory transfers. This makes it straightforward to determine where computation occurs, identify launch overheads, and correlate them with memory transfer costs.

In contrast, profiling the persistent-threaded implementation presents significant challenges. Since the persistent kernel is launched only once and remains active for the duration of execution, there are no distinct kernel launch entries on the timeline for each individual task. Instead, the majority of the visualized activity in `nsys` consists of frequent host-to-device and device-to-host memory transfers corresponding to the queuing and completion of tasks. The actual execution of the tasks within the persistent kernel is effectively "hidden" from the profiler's high-level view, as it takes place inside the single long-running kernel.

This difference means that the persistent-threaded execution does not lend itself well to the same form of visual, task-by-task inspection available in the baseline case. While memory transfer patterns can still be analyzed, the lack of discrete kernel events makes it difficult to directly measure per-task execution time or to distinguish between overlapping computation and data movement. To obtain deeper insights, low-level instrumentation or custom device-side logging would be required, as standard profiling tools are optimized for discrete kernel launches rather than continuous, event-driven GPU execution.

Consider the evaluation of the current execution.

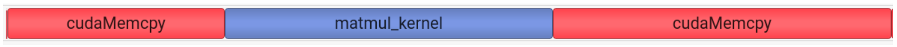

Figure 5.3: Simple `nsys` timelines for the baseline implementations.

5.3 Coroutine Implementation

While the primary objective of this work was the implementation of GPU persistent threads, it became evident that the same architectural foundations could be extended to support coroutine functionality with relatively modest additional effort. The existing design already incorporates a task queue, which the GPU processes in sequence, and a dedicated memory address space for function contexts, making it possible to store and retrieve the continuation state of executing tasks. These features naturally lend themselves to coroutine-style execution, where tasks can yield control and later resume from the same execution point.

5.3.1 Yielding and Changing Tasks

In essence, a coroutine yields its execution by voluntarily releasing its currently allocated compute resources, allowing other tasks to make progress. Within the present implementation, this can be modeled directly using the existing task queue mechanism. The task queue is implemented as a circular buffer, processing tasks in order from the head to the tail. When a task yields, it is effectively repositioned within this queue, relinquishing its slot so that another, potentially higher-priority, task can execute in its place. The most urgent pending task can then take over the vacated slot, ensuring minimal idle time for the GPU.

A more advanced extension would involve replacing the simple circular buffer with a *priority heap*. In such a structure, tasks are automatically sorted and selected for execution based on their priority levels, allowing the scheduler to make more informed decisions about task ordering. While the current system executes tasks to completion once they are selected, this model could still allow tasks to voluntarily halt and later resume execution, improving responsiveness for workloads with mixed task lengths.

5.3.2 Continuation and Resumption

Implementing coroutine behavior also requires explicit mechanisms for saving and restoring task state. When a task yields, the GPU must store sufficient continuation data so that it can later resume execution from precisely the point where it was interrupted. The current memory structure already provides a convenient, per-task memory region where such continuation data can be stored. This includes both the computational state and the control information necessary for resuming execution at the correct instruction address within the device function.

To enable this, each task must be preallocated with enough device memory during scheduling to hold both its input parameters and any continuation data that may be required. This is achieved by copying the task's input stream into the designated memory region and advancing the memory offset to reserve additional space for storing the continuation state. When a task yields, the current execution context—such as register values, loop counters, and program counters—is written into this reserved space. Upon resumption, the scheduler retrieves this data and restores the task's execution context, allowing it to continue seamlessly from its previous stopping point.

By combining these mechanisms with the persistent thread framework, the GPU could effectively execute multiple long-lived, cooperative tasks, enabling finer-grained scheduling and more efficient utilization of GPU resources, especially in irregular or latency-sensitive workloads.

5.4 Limitations and Future Work

The current system does not support priority based scheduling decisions or utilizing coroutines for software preemption like execution. Future work implementing the strategies outline by this thesis towards utilizing the persistent kernel implemenation as a baseline for these tasks would enable further predictable real time scheduling capabilities. In this case, the memory management system might need to progress from an epoch based scheduler to a more complex management system. Low priority tasks would interrupt the freeing of epoch memory. Furthermore, the current manual streaming of input and output parameters restricts 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.

Further improvements could also include:

- Implementing task batching, where each worker processes multiple tasks before polling again, to reduce synchronization overhead.
- Optimizing the polling mechanism with adaptive backoff strategies to minimize resource consumption.
- Leveraging asynchronous memory copies with double buffering and pinned memory to better overlap data transfers and computation.
- Extending the evaluation to more complex kernels and real-world workloads beyond matrix multiplication to better characterize the benefits and limitations of persistent threads.

Overall, while the current implementation demonstrates the feasibility of persistent GPU threads, there remains substantial opportunity for optimization and scaling to fully leverage the advantages of this approach.

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

CNN Convolutional Neural Network

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
JIT	just in time
IR	intermediate representation
AST	abstract syntax tree

List of Figures

List of Tables