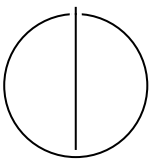
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

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

Jaden Rotter



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

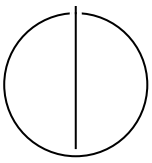
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

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

GPU Coroutines in Autonomes Fahren

Author:	Jaden Rotter
Examiner:	Supervisor
Supervisor:	Jianfeng Gu
Submission Date:	Submission date



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

Munich, Submission date Jaden Rotter

Acknowledgments

Abstract

Contents

1 Introduction

Autonomous driving systems place stringent demands on computational performance, 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.

Modern, CUDA capable GPUs, rely on a proprietary hardware scheduler that functions as a black box to the programmer. This scheduler is optimized for throughput rather than deterministic execution, which introduces unpredictable latency from resource contention between tasks. Critically, tasks with strict timing requirements may suffer delays if long running, lower priority kernels are already resident on the device. The inability to preempt GPU kernels or to enforce strict task priorities makes the utilization of GPUs difficult for safety critical, real time workloads. Particularily in the context of autonomous driving, the repercussions of latencies pose a serious safety concern.

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 [1]. 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. Moving to a singular compute node allows savings in cost, design, and intermodule latency. This compute node uses the GPU for compute intensive workloads, while leaving the CPU free to orchestrate the scheduling and control of the system.

Despite the benefits afforded by a centralized architecture and the advances in hardware, the system still 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, 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 at both the OS level as well as the programming level, which this thesis aims to abstract and implement for GPUs in a real time autonomous driving system.

To enable the GPU to support these real time features, the device scheduling has to be handled by the programmer instead of natively. Rather than relying on native kernel launches, this thesis explores GPU coroutines and persistent threads as mechanisms for enabling predictable, low latency execution. Coroutines provide cooperative multitasking between GPU threads, allowing the system to make scheduling decisions otherwise restricted by the proprietary driver. This additional scheduling ability allows the system to implement custom schedulers that prioritize critical work. Persistent threads, long lived threads that remain active throughout the lifetime of the system, are fundamental to the management of these coroutines. Furthermore, persistent threads have the added benefit of reducing kernel launch overhead.

As part of this research, an attempt was made to directly integrate a coroutine based GPU scheduling system into an autonomous driving system. However, the lack of documentation and time as well as the complexity of the existing framework proved to be a limitation. As a result, this thesis pivoted to implementing a custom persistent thread scheduler on which coroutines can be implemented for real time systems. This approach enables long running GPU threads to receive new tasks, yield between them, and implement task prioritization in software. The software yielding emulates the desired preemption to improve responsiveness under load.

In summary, this thesis investigates real time GPU scheduling techniques for autonomous driving. By building a persistent thread architecture to support coroutine based task control, the goal is to reduce latency variability and enable timely execution of safety critical GPU workloads. This thesis aims to bridge the gap between the throughput oriented design of modern GPUs and the strict timing guarantees demanded by real time autonomous systems.

2 Objectives

The primary objective of this thesis is to explore GPU scheduling techniques that enable predictable, low latency execution for real time autonomous systems. In particular, the aim is to address the limitions of current GPU execution models, which prioritize throughput at the expense of timing guarantees, by investigating alternative scheduling strategies suitable for safety critical environments. Furthermore, this thesis aims to explore the GPU hardware architecture and CUDA programming model in order to design an efficient optimized scheduling strategy.

The initial objective of this work was to integrate an existing coroutine based GPU scheduler into a autonomous driving framework. This integration was intended to evaluate the viability of fine grained GPU scheduling within a complex, real time system. Additionally, this work further sought to evaluate the scheduling latencies in a coroutine based implementation, from which specific timing guarantees may be derived. Ultimately, given the steep learning curve of both the GPU coroutine framework and the autonomous driving platform, and my limited prior experience with GPU programming and compiler theory, the integration proved more difficult than expected and required a narrowing of scope. As a result, this original objective was reconsidered.

The thesis therefore shifted focus to a more foundational and controlled apporach. Instead of embedding GPU scheduling into an existing system, this work focuses on designing and implementing a custom persistent thread scheduler using native CUDA. This scheduler serves as a minimal proof of concept foundation for enabling persistent GPU threads to receive tasks, yield between them, and emulate prioritization. By building this system, it becomes possible to explore how real time behaviors can be emulated in software, to study the limitations of the CUDA execution model, and to understand teh design trade-offs involved in managing GPU concurrency manually.

This revised objective emphasizes both the practical and conceptual aspects of real time GPU scheduling. Practically, it provides a working framework for testing scheduling behavior under load. Conceptually, it offers insights into the GPU architecture and CUDA programming model in maximizing the hardware capabilities and utilization. The findings of this work aim to inform future efforts to integrate persistent thread or coroutine based scheduling into real world autonomous driving systems.

3 Related Work

Real time GPU programming models are a result of more autonomous systems and the increased hardware performance of GPUs in those applications. Programmers want to ensure these systems are predictabe. These applications, ranging from robotics to scientific computing, use GPU programming models to reduce kernel overhead while improving resource utilization and predictability. Although GPUs offer high throughput and timely execution important for these domains, they require non native programming models to ensure predictable execution models. Researchers have proposed various solutions to adapt GPU execution models to these real time constraints. The solutions fall into three categories: compiler driven frameworks, runtime scheduling frameworks, and manual implementations.

3.1 Compiler Driven Frameworks

Compiler driven frameworks focus on optimizing GPU workloads through automated code generation. These systems are based on persistent threads, which reduce kernel launch overhead, maximize on chip memory reuse. The compilers generate code using a specific DSL, which abstracts the low level device code. The compiler then parses the DSL into an abstract syntax tree. The abstract syntax tree allows the compiler to transform and optimize the code into an intermediate representation, which is transformed into device code JIT.

One such project, *Mirage* implements these ideas to improve large language model inference, by merging kernels into a singular megakernel. The singular megakernel is essentially a persistent kernel that eliminates extra memory copies between kernels, lowers kernel launch overhead, and retains memory on chip. 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 through autoschedulers that requires 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, allowing developers to 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 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 specificially for ROS 2 robotics systems. The ROSGM system interposes a layer to intercept the CUDA API calls to insert metadata to 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 Background

4.1 Real Time Systems

Real time systems, such as autonomous driving, are designed with strict timing constraints in mind, to ensure predictable and deterministic behavior [2]. deadlines, which are subdivided into soft and hard-deadlines. Hard deadlines are critical and a failure leads to the systems failure or unsafe conditions. For example, in autonomous driving, collision avoidance with another vehicle and brake activation are hard deadlines. If these deadlines are not met, the safety of the passengers and the system is at risk. On the other hand, soft deadlines are not critical and missing these deadlines degrades performance, but does not cause system failure. In autonomous driving, this would show in route planning and navigation updates, where a delay would lead to suboptimal paths, but safety is not comprimised. Real time systems need to be capable of effectively and efficiently switching from lower priority tasks, soft deadline tasks, to high priority, hard deadline tasks, to ensure the safety both for the passengers and nearby individuals.

4.2 Integration of GPUs in Autonomous Driving Systems

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

GPUs are particularily well suited to these workloads because of their massively parallel architecture and high memory bandwidth, which perfectly meet the demands of deep learning tasks. Unlike CPUs, which optimize for sequential instruction execution

and low latency branching, GPUs are designed to handle large batches of matrix and tensor operations simultaneously. This makes them ideal for real time inference of deep neural networks. Furthermore, modern GPU architectures provide specialized cores, such as tensor cores in NVIDIA GPUs, that are explicitly optimized for mixed precision matrix multiplication, which is a core operation in most machine learning models. By offloading intensive compute tasks to the GPU, autonomous systems can maintain low latency and high accuracy, both of which are crucial for safety and performance in dynamic driving environments.

4.3 GPU versus CPU Architecture

GPUs are capable of delivering this vast increase in throughput over CPUs as measured by GFLOPS, despite the lower clock rate, by simplifying the thread context in order to afford greater parallelism. They were originally developed to accelerate graphics rendering, a task heavy in parallizable computations, which require only a very simple control overhead and as such have adapted the architecture to support as many possible different threads. Typical workloads designed for CPU are based on sequential workloads, such as human input or complex logic, which requires complex thread overhead to speed up branches and I/O, through prefetching, branch prediction, and out of order execution. These CPUs achieve higher single threaded performance by dedicating a "significant portion of transistors to non computational tasks like branch prediction and caching", which GPUs can forgo in favor of increasing arithmetic intensity [5]. Consider the following graphic Figure 4.1, which highlights the difference in thread complexity.

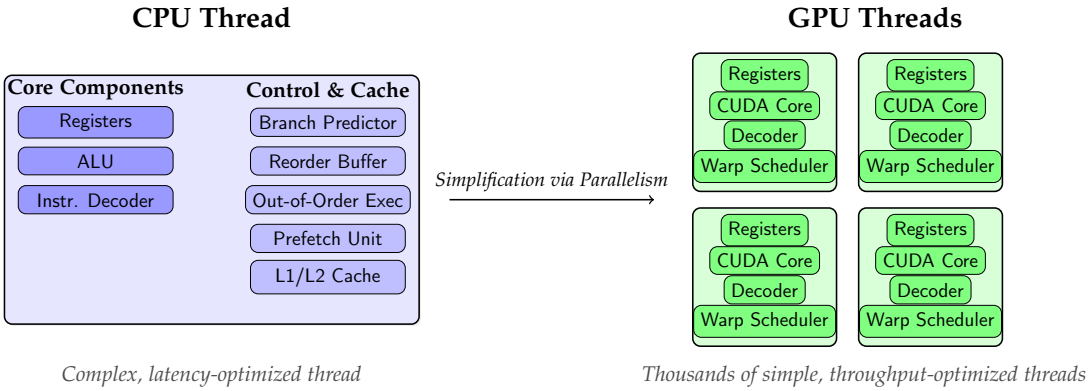


Figure 4.1: CPU vs GPU Thread Architecture

Although core components are named differently, both CPU and GPU threads work fundamentally similarily with an instruction decoder, registers and an arithmetic unit.

The differences arise when trying to maximize a single control flow. The CPU will prefetch instructions, reorder them to most effectively use the functional units and speculatively compute instructions based on a branch predictor. On the other hand, GPU threads can not execute instructions out-of-order, use only manual prefetching, and have a simple branch predictor that is far more conservative than the CPU's predictor. Furthermore, the GPU amortisizes the cost of managing an instruction stream accross multiple threads, which execute the same instructions at the same time versus the CPU execution model which The additional complex logic involved allows single threaded CPU applications to far outperform single threaded GPU applications, as seen by the following comparison of single threaded matrix multiplication in Figure 4.2 and Figure 4.3.

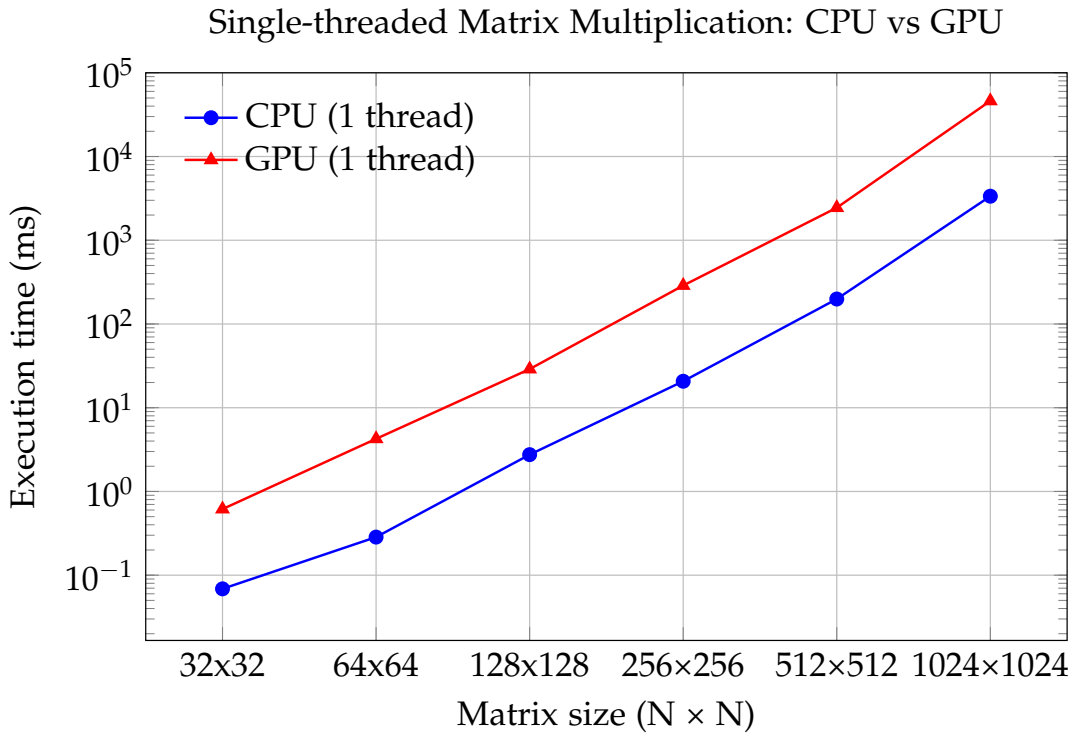


Figure 4.2: Single threaded Matrix Multiplication Execution between CPUs and GPUs averaged over 10 executions

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 4.3: Data Matrix from Figure 4.2

As seen in Figure 4.2 and Figure 4.3 applications that fail to utilize the concurrancy of GPUs, either due to programmatical errors or a lack of parallelism in the task, will struggle to achieve high performance. For each of the matrices tested, through prefetching, a higher clock rate, and branch prediction, the CPU is on average around 13 times faster than the GPU running the exact same algorithm. Following these results, the GPU should only be used in place of the CPU, when the application is well tuned to the programming model and a scheduler must respect this difference. If a GPU scheduler is written without regard for these differences, it will may not be able to achieve the desired performance benefit. This fundamental architectural and programming style for GPUs must be understood in order to maximize the throughput.

4.3.1 GPU Hardware Architecture based on the Tesla V100 GPU

For the purposes of programming and scheduling tasks onto the Tesla V100 GPU[1], the GPU appears as an array of independent highly parallelized processors, called SMs. The SMs are grouped into specific TPCs, which are then further grouped into GPCs, but the specific mapping of tasks to SM, TPC, and GPC is determined by the proprietary hardware scheduler, the GigaThread Engine. The exact workings and scheduling methods of the GigaThread Engine are not publicly available, but this module maps CTAs, groups of threads executing the same instruction code, to the individual SMs based a multitude of factors: hardware resources, parallelism, priorities, and dependencies. Similarily, the global memory and L2 cache utilization are determined by the hardware and transparent to the programmer. After the CTA gets mapped to the specific SM, the device code then executes till completion. Each SM manages its scheduled CTAs through its own execution pipelines, register files, shared memory and scheduling units that function independently from one another. For SMs to communicate with one another, they must use either the global on-chip

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

device HBM2 or through the global L2 cache which is shared and coherent across all SMs. Although these memory accesses allows individual SMs to communicate with each other, accesses require hundreds of cycles, which introduce further latencies when compared to local SM L1 memory caches. Ideally, the SMs execute independently of one another and cumulate answers in global memory, skipping the high memory latency accesses of coordinating synchronous work.

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

The Tesla V100 GPU SM architecture, self contains an entire execution pipeline within each of its 4 processing partitions, which collectively share an L1 instruction cache as well as an L1 data and shared memory cache. As CTAs are distributed across multiple Warps, these collective L1 caches allow the instruction memory and shared memory to be stored across different Warps within the same CTA. The main components of each processing partition are a L0 instruction cache, warp scheduler, dispatch unit, execution units. Every clock cycle, the Warp scheduler schedules a singular Warp of 32 threads, which get passed to the Dispatch unit. The dispatch unit then dispatches a new instruction to the Warp every clock cycle. As for any given instruction there are not enough execution units of the same type, the instructions get queued onto the execution units. Depending on the current queue and any delays, such as global memory accesses or dependencies, the Warp scheduler will interweave different Warps together onto the Dispatch Unit to hide latencies. Within each Streaming Multiprocessor (SM) processing partition's execution units, there are Tensor Cores for deep learning, 64 bit-floating point (FP) cores, Load/Store (LD/ST) units, a register file, and SFUs for mathematical functions such as sine and square root. An in depth view of the processing partition's architecture is provided in Figure 4.4.

Figure 4.4: Architecture from the whitepages: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

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

4.4 GPU Programming using the CUDA API

NVIDIA has provided an underlying CUDA API to allow programmers to run tasks on the GPU in a heterogenous computing architecture. The GPU acts as a standalone processor with its own memory, which is not shared with the host system or CPU's main memory. However, the GPU execution is dependent on tasks received from the CPU. Given that the device memory is seperate from the host memory, typical GPU workloads work by first allocating memory on the device, copying the memory over, executing the program, copying the memory back, and then freeing the device memory. Although the CUDA API manages the actual underlying steps, the API allows the programmer to specifically program and optimize the GPU for their specific task.

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

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

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

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

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

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

Listing 4.1: Simple CUDA Kernel

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

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

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

4.4.1 Memory and Bandwidth Considerations

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

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

4.4.2 Kernel Launches

The task of launching and running device code begins from a kernel launch, which passes the function, its parameters, pointers, and the grid and block dimensions to the GPU. Each kernel launch specifies how work is diveded among thread blocks and individual threads. Every block is mapped to a SM injectively and is constrained by that

SM's hardware resources including the number of threads, registers, available warps, and shared memory. If there are no available SM's to meet these requirements the kernel launch will fail. Should the CTA or thread block not fully saturate the hardware resources, additional blocks may be scheduled to the same SM. Each thread and block is assigned unique identifiers, threadIdx and blockIdx, that allow them to determine their position in the execution grid. These identifiers are crucial for structuring parallel computations and can be used to optimize memory access patterns. For instance, when threads in a warp access consecutive memory locations, the memory accesses can be coalesced into a single transaction, significantly improving memory throughput.

4.4.3 CUDA Streams

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

The Tesla V100 GPU has two seperate hardware copy engines for copying data from the host to the device and back. The copy engines support the transfer in both directions, with one engine specifically being allocated for the unidirectional D2H transfer and the other for H2D. Using only one stream for multiple kernels fails to maximize the device memory bandwidth. For example, consider the launch of two independent kernels, kernel A and kernel B, each on the same cuda stream. Both A and B, allocate and copy memory onto the device, schedule their kernels and then copy the results back. Regardless of the ordering of API calls, both kernels can not run simultaneously or use both H2D and D2H copy engines simultaneously.

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

5 Luisa Coroutines

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

5.1 Coroutines

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

Coroutines enable cooperative multitasking between routines, where instead of a thread or process level context between different tasks, they maintain only a function level context switch that allows fast task switching. Coroutines, an implementation of asynchronous programming, uses suspendable functions to halt execution. Suspendable functions are implemented by capturing the current context, know as the continuation, of the currently running thread and save the data to be run later [6]. After being saved, a new process can take over execution, without interrupting or overwritting the state of the previous process. Once the intermittant process or higher priority process has finished execution, the original task can continue executing by restoring the process context, which was previously saved. Capturing the continuation of a function allows the resumption of the program to be strategically deferred.

5.2 CPU Coroutines

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

```
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 5.1: CPU Coroutine

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

5.3 GPU Coroutines

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

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

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

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

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

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

6 Persistent Threads

Persistent GPU threads allow the hardware resources to be partitioned and reduce kernel execution and scheduling overhead and are fundamental for other applications seeking to enhance GPU scheduling. Many other applications such as coroutines or mega kernels are based on persistent threads as a means of reducing kernel launch overhead and allow further control in application specific scheduling decisions, otherwise black boxed by the cuda api. Running persistent threads essentially allows the memory and system configurations to be loaded in ahead of runtime and reduces overhead during execution.

Failing to implement LuisaCompute's coroutines into Apollo, the goal shifted to finding a manual GPU scheduling functionality to implement and use. In restricting the scope from GPU coroutines to simply implementing persistent threads into Apollo, this thesis attempted to find an open source implementation which would allow the fine grained scheduling controll specific to Apollo. Unfortunately, given that most persistent thread implementations are highly specific they are not open source, which led to selecting an implementation that required more work to make it feasible. The only open source persistent thread implementation that was readily available was LightKer, a research project, which measured the hypothetical speedup of using persistent threads over sequential kernel launches. Seeking to implement LightKer into Apollo first required a complete restructuring of the code base to support a real time system.

The LightKer implementation itself intended to measure the performance difference between sequential kernel executions versus a persistent kernel implementation. This implementation constructed simple trivial kernels and tested the overhead difference between calling them explicitely from the host in kernel launches versus implicitely in the persistent kernel. Unfortunately, this application does not support variable tasks or memory transfers at runtime, necessary to pass arguments and results back and forth. As such, the implementation only succeeds in measuring latency differences between scheduling tasks using a device side while loop versus explicit kernel launches. However, the LightKer implementation does provide a simple framework for using a GPU-Host mailbox for scheduling kernel tasks as well as a helpful persistent kernel launch structure.

6.1 Architecture

From a high level, the architecture implemented into this project appears as follows, with the GPU-Host Mailbox, being used from the Lightkernel project as well as the internal structure.

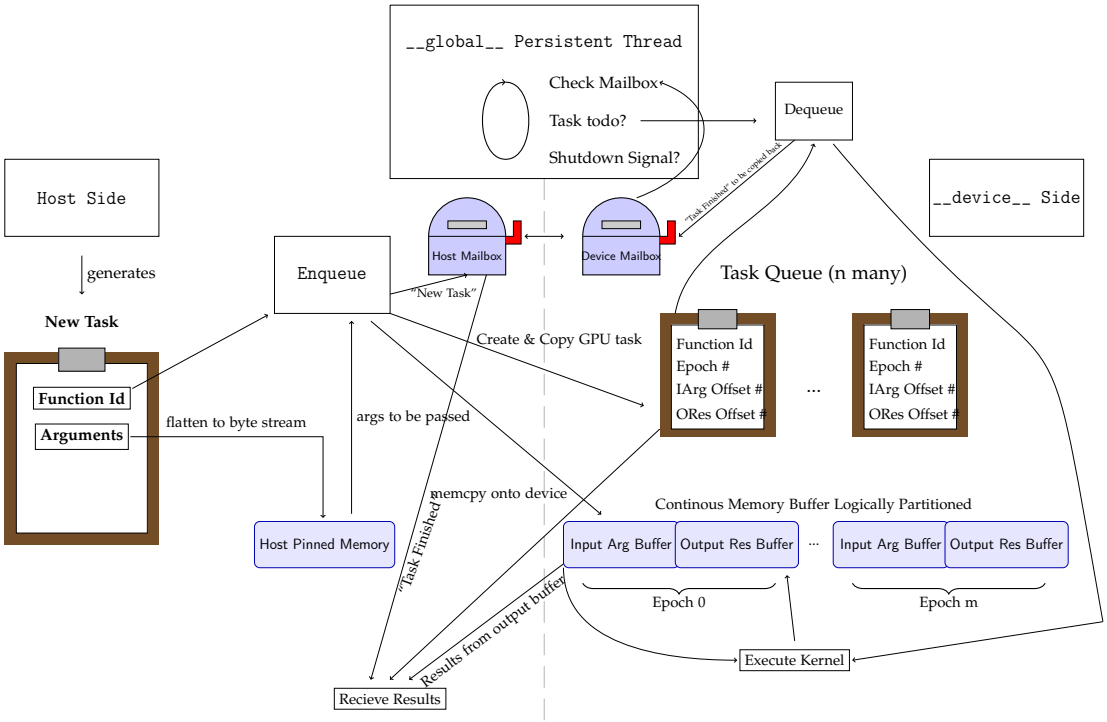


Figure 6.1: Persistent Thread Architecture implemented into LightKer

At a high level, there are three important components: the task queue, memory buffers, and the persistent threads themselves. 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. Lastly, the persistent threads are the fundamental execution units executing the GPU code.

The Figure 6.1 depicts these individual components in a logical program architecture 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.

6.2 Implementation

GPU kernels are essentially device side functions launched by the host process, which execute with parameters and a logical grid of threads to be distributed across the SMs. In order to replicate both the parameterization and execution model without explicitely launching new kernels, the persistent threads must maintain a mechanism for storing function pointers and their associated arguments to assign work to idle compute resources dynamically. Similar to other persistent thread implementations, this project implements a task queue, that captures the full execution context required to execute the GPU code. As shown in the Figure 6.1, the task queue is represented as an array of clipboards each encapsulating a function context. Beneath the task queue, Figure 6.1 also depicts the staging area used for memory management. The memory management is implemented using an epoch based allocation strategy. These memory buffers serve as a real-time memory management structure for GPU kernels, reducing the need for direct runtime memory allocation while enabling in-place memory reuse for tasks.

Based on the results in Figure 4.2, any sequential scheduling mechanism executing directly on the device would incur significantly performance penalties when compared to a CPU based approach. As a result, both the task queue and the associated memory buffers are managed from the host side. This includes deteriming the active epoch and assigning task indices within the corresponding memory or queue structures. The host is responsible for copying all required parameters into device memory and delegates only the dequeuing and execution of tasks to the GPU.

Furthermore, 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.

6.2.1 Task Queue

The task queue functions as a cache like buffer between the host and the executing GPU code, enabling the asynchronous enqueuing of tasks. Rather than relying on the cuda driver to dynamically partition the GPU and assign tasks to threads, task parameters are instead written into a pre-allocated memory region. This memory acts as a staging area and remains allocated throughout the duration of the persistent thread kernel, with periodic cleanup.

Each task entry defines its execution context through an explicit function identifier and its associated parameters. This entry acts exactly like a coroutine continuation, which stores the necessary state to resume execution within the function. The GPU kernel, launched with persistent threads, enters a loop in which each block repeatedly dequeues and executes tasks.

Within this loop, each GPU block retrieves the next task from the queue, processes it, and stores its results. The task queue maintains both input and output buffer offsets for each tasks, allowing blocks to fetch parameters and write results. 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.

Upon completion of the task, the GPU block writes the results to the specified output buffer offset. This scheme works because the CPU knows the exact size of the input and output arguments, knowledge it already must have for issuing cudaMemcpy operations.

Once a the block finishes executing a task, it signals the host that the task is complete and then continues processing the next available task in the queue.

6.2.2 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.

6.2.3 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. Manually extending this method allows the user to manually allocate more memory than is needed and use that memory to yield and run coroutines.

6.2.4 Memory Model

In order to provide the incoming scheduled kernels a staging area for allocated memory, the implementation contains a running epoch memory model. The memory model contains n epochs, with an input and output staging area for each respective epoch. As input are copied into epochs, eventually the corresponding input memory buffer will overflow. The host enqueuing functionality manages the current epoch and the current offset within that epoch, which allows the host to detect when the input buffer will lead to an overflow. Once the buffer is saturated, the host proceeds to allocate further memory to the next epoch. After the device finishes executing the task and copies results to the corresponding epoch's result buffer, the host is notified and that memory is marked as free. 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.

The motivation for this epoch based strategy both asynchronously manages the execution of tasks as well as overhead reduction of continuously freeing and allocating new memory. As these persistent buffers remain allocated throughout the entirety of the kernel, there is no longer any overhead involved in gpu side memory allocation

or freeing, as the memory is tied to the lifetime of the kernel and only internaly considered allocated or freed. When the buffer queue loops and returning back, if the specific buffer parameters and memory size has been correctly set, potentially through profiling, the buffer will be free and can be reused. The buffer is only considered free if all tasks within the buffer are considered free, which removes any complicated GPU side memory allocation schemes.

6.2.5 Cuda Streams Optimization

To fully leverage the GPU's memory transfer capabilities, input and output transfers are performed in separate, overlapping CUDA streams. As previously noted, the GPU features distinct engines for H2D and D2H transfers, which can operate concurrently. By assigning input transfers to a dedicated H2D stream and output transfers to a separate D2H stream, this implementation avoids intra-stream dependencies and enables parallel data movement in both directions.

7 Evaluation

Abbreviations

GPU Graphics Processing Unit

CPU central processing unit

DSL domain specific language

GPC Graphics Processsing Cluster

TPC Texture Processing Cluster

SM Streaming Multiprocessor

FP floating point

LD/ST Load/Store

SFU special function units

CTA Cooperative Thread Array

HBM2 High Bandwidth Memory

VRAM Video Random Access Memory

RAM Random Access Memory

D2H Device to Host

H2D Host to Device

JIT just in time

List of Figures

List of Tables

Bibliography

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

[2] 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.

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

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

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

[6] 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.