SHARING GPUS FOR REAL-TIME AUTONOMOUS-DRIVING SYSTEMS

Ming Yang

A dissertation submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2020

ABSTRACT

Ming Yang: Sharing GPUs for Real-Time Autonomous-Driving Systems
(Under the direction of James H. Anderson)

Autonomous vehicles at mass-market scales are on the horizon. Cameras are the least expensive among common sensor types and can preserve features such as color and texture that other sensors cannot. Therefore, realizing full autonomy in vehicles at a reasonable cost is expected to entail computer-vision techniques. These computer-vision applications require massive parallelism provided by the underlying shared accelerators, such as graphics processing units, or GPUs, to function "in real time." However, when computer-vision researchers and GPU vendors refer to "real time," they usually mean "real fast"; in contrast, certifiable automotive systems must be "real time" in the sense of being *predictable*.

This dissertation addresses the challenging problem of how GPUs can be shared predictably and efficiently for real-time autonomous-driving systems. We tackle this challenge in four steps.

First, we investigate NVIDIA GPUs with respect to scheduling, synchronization, and execution. We conduct an extensive set of experiments to infer NVIDIA GPU scheduling rules, which are unfortunately undisclosed by NVIDIA and are beyond access owing to their closed-source software stack. We also expose a list of pitfalls pertaining to CPU-GPU synchronization that can result in unbounded response times of GPU-using applications. Lastly, we examine a fundamental trade-off for designing real-time tasks under different execution options. Overall, our investigation provides an essential understanding of NVIDIA GPUs, allowing us to further model and analyze GPU tasks.

Second, we develop a new model and conduct schedulability analysis for GPU tasks. We extend the well-studied sporadic task model with additional parameters that characterize the parallel execution of GPU tasks. We show that NVIDIA scheduling rules are subject to fundamental capacity loss, which implies a necessary total utilization bound. We derive response-time bounds for GPU task systems that satisfy our schedulability conditions.

Third, we address an industrial challenge of supplying the throughput performance of computer-vision frameworks to support adequate coverage and redundancy offered by an array of cameras. We re-think the design of convolution neural network (CNN) software to better utilize hardware resources and achieve increased throughput (number of simultaneous camera streams) without any appreciable increase in per-frame latency (camera to CNN output) or reduction of per-stream accuracy.

Fourth, we apply our analysis to a finer-grained graph scheduling of a computer-vision standard, OpenVX, which explicitly targets embedded and real-time systems. We evaluate both the analytical and empirical real-time performance of our approach.

To my sister, Yuan Yang.

ACKNOWLEDGEMENTS

It was serendipity that led me to UNC five years ago. As I stand at the end point of this journey now, with this dissertation being completed, I realize that anything I accomplished would not have been possible had I not received all the guidance, aid, and support from people I met. I am indebted and thankful to them.

First and foremost, I would like to express my deepest appreciation to my advisor, Jim Anderson, for his unwavering support, for his patience whenever I was slowly progressing, for his motivation and encouragement every time I was frustrated, and for his guidance that led me forward. I would also like to thank my dissertation committee: Parasara Sridhar Duggirala, Jan-Michael Frahm, Shahriar Nirjon, F. Donelson Smith, and Shige Wang, for their valuable advice.

I would also like to extend my sincere thanks to many colleagues I worked with. In particular, I am especially thankful to people who helped with the work of this dissertation: Tanya Amert, Joshua Bakita, Nathan Otterness, Thanh Vu, and Kecheng Yang. This dissertation would not have been possible without their contributions.

I am also grateful for the internships that General Motors and Aurora Innovation offered so I had the opportunities to enjoy sitting in the autonomous vehicles and experiencing how they work. I very much appreciate Shige Wang for his advice regarding and beyond my research and career. I am also thankful to Glenn Elliott for many inspiring discussions. I thank Joseph D'Ambrosio and Ken Conley for their support.

I am also thankful to my other co-authors: Alex Berg, Pontus Ekberg, Vance Miller, Saujas Nandi, Catherine Nemitz, Eunbyung Park, Sarah Rust; and other people I worked with in UNC: Shareef Ahmed, Akash Bapat, Lee Barnett, Micaiah Chisholm, Calvin Deutschbein, Shiwei Fang, Cheng-Yang Fu, Zhishan Guo, Clara Hobbs, Bashima Islam, Tamzeed Islam, Namhoon Kim, Seulki Lee, Yubo Luo, Mac Mollison, Sims Osborne, Abhishek Singh, Stephen Tang, Peter Tong, Sergey Voronov, and Bryan Ward.

I gratefully acknowledge the assistance I received from the staff of the UNC Computer Science Department. Special thanks to Denise Kenney, Beth Mayo, Jodie Gregoritsch, Adia Ware, and Mellisa Wood for keeping the graduate school paperwork in order. Many thanks to Murray Anderegg, Bil Hays, David Musick, Mike Stone, and other staff for their help with various hardware and system issues.

I cannot leave Chapel Hill without mentioning my friends. I owe special thanks to Weiwei Li, for preventing me from insanely giving up my PhD study. I was also very lucky to meet Shiwei Fang by chance in the roommate lottery. Thanks to them and my other friends for enriching my PhD life experience.

As I believe everyone needs a supporting system to survive the PhD process, mine is my dearest sister—in my darkest time or whenever, she was always there for me, persuasive and supportive, and finally, I share this victorious ending of this journey with her. Lastly, I thank my parents for their unwavering support, and my wife for her patience and love.

The research in this dissertation was funded by NSF grants CPS 1446631 and CPS 1837337, ARO grant W911NF-17-1-0294, and support from General Motors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACC	Adaptive Cruise Control
ADAS	Advanced Driver Assist System
AI	Artificial Intelligence
AP	Average Precision
ASICs	Application-Specific Integrated Circuits
BAND	Bandwidth-Aware Non-preemptive Device
CAN	Controller Area Network
CDF	Cumulative Distribution Function
CE	Copy Engine
C-FL	Clustered Fair-Lateness
CKE	Concurrent Kernel Execution
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DARPA	Defense Advanced Research Projects Agency
DNN	Deep Neural Network
DPM	Deformable Part-based Model
DRAM	Dynamic Random-Access Memory
DSPLIB	DSP LIBrary
DSP	Digital Signal Processor
EDF	Earliest-Deadline-First
EE	Execution Engine
FCW	Forward Collision Warning
FFT	Fast Fourier Transform
FIFO	First-In-First-Out
FP	Fixed-Priority
FPGA	Field Programmable Gate Arrays
FPPI	False Positive Per Image

FPPW	False Positive Per Window
FPS	Frames Per Second
G-EDF	Global Earliest-Deadline-First
GEL	G-EDF-like
G-FIFO	Global First-In-First-Out
G-FL	Global Fair-Lateness
GPGPU	General-Purpose Graphics Processing Unit
GPS	Global Positioning System
GPU	Graphics Processing Unit
GVA	Guest Virtual Address
HDL	Hardware Description Language
HOG	Histogram of Oriented Gradients
HPC	High-Performance Computing
HRT	Hard Real-Time
HSA	Heterogeneous System Architecture
ILP	Integer Linear Program
IMU	Inertial Measurement Unit
IPC	Inter-Process Communication
ISA	Instruction Set Architecture
JLFP	Job-Level Fixed-Priority
KDE	Kernel Density Estimation
LDW	Lane Departure Warning
LITMUS^{RT}	LInux Testbed for MUltiprocessor Scheduling in Real-Time systems
LKAS	Lane-Keeping Assistance Systems
LKM	Loadable Kernel Module
LRU	Least Recently Used
MB	Mega Bytes
MMIO	Memory-Mapped Input/Output
MPA	Machine Physical Address

MPH	Miles Per Hour
MPS	Multi-Process Service
NHTSA	National Highway Traffic Safety Administration
NPP	NVIDIA Performance Primitives
OS	Operating System
PCA	Principal Component Analysis
PCI	Peripheral Component Interconnect
PGM^{RT}	Processing Graph Methods for Real-Time
PKM	Preemptive Kernel Model
PREM	PRedictable Execution Model
PR	Precision-Recall
PTX	Parallel Thread Execution
PT	Persistent Threads
RL	Reinforcement Learning
RNN	Recurrent Neural Network
RPC	Remote Procedure Call
RTA	Response-Time Analysis
RTOS	Real-Time Operating System
SAE	Society of Automotive Engineers
SIFT	Scale-Invariant Feature Transform
SIMD	Single-Instruction Multi-Data
SIMT	Single-Instruction Multi-Thread
SLAM	Simultaneous Location And Mapping
SM	Streaming Multiprocessor
SPT	Shadow Page Table
SRT	Soft Real-Time
SSD	Single-Shot multibox Detector
SVM	Support Vector Machine
SWaP	Size, Weight, and Power

TLB	Translation Lookaside Buffer
TPN	Timed Petri Nets
VOC	Visual Object Classes
WCET	Worst-Case Execution-Time

CHAPTER 1: INTRODUCTION

After being envisioned for decades, autonomous vehicles seem to be just over the horizon. We expect this innovation to reduce traffic accidents, save lives, mobilize those with disabilities for driving, protect the environment, liberate workers from commuting, and ultimately transform how our cities function. More importantly, we expect these enhancements to be achieved *safely* because the fundamental goal of saving lives is rooted in the capabilities of technology. The safety requirements for autonomous vehicles focus around executing appropriate driving decisions in real time, imposing the requirements of both *logical correctness* and *temporal correctness*. Satisfying such safety requirements is challenging because of the complex software components and hardware platforms that underpin autonomous-driving systems.

One primary software component in an autonomous-driving system is the perception system that semantically senses and understands the surrounding environment. A common approach used in the perception system is vision-based sensing because compared to lidars and radars, cameras are less expensive and uniquely capable of perceiving colors and textures. With a proliferation of computer-vision research in the literature, many state-of-the-art features have been applied in today's autonomous vehicle prototypes.

However, existing computer-vision frameworks often fail to provide enough inference performance, not to mention that such performance is unpredictable, so the requirement for temporal correctness remains to be established. Despite the goal of improving performance and ensuring predictability, any induced accuracy loss must be tolerable. We investigate solutions for providing sufficient and predictable performance in the use of the underlying hardware accelerator with a focus on graphics processing units (GPUs).

GPUs are commonly used for accelerating computer-vision applications, as they provide the massive parallelism needed by these types of applications, which process millions of pixels per image streaming from multiple video sources to provide full coverage of a vehicle's surroundings. With applications providing disparate functionalities and advancements of GPUs offering more processing capacity, the incentives to share GPUs among multiple applications by enabling predictable simultaneous execution have grown. Also, any unnecessary idleness incurred by CPU-GPU interactions in existing computer-vision frameworks should be minimized for better performance.

Motivated by these challenges, our research considers how GPUs can be shared across multiple applications to improve throughput with bounded response times and high per-stream accuracy. In the following section, we provide the details of our application focus on computer vision to enable safe autonomous driving.

1.1 Computer Vision in Autonomous Driving

With the development in autonomous driving ongoing for decades, the latest research that spurred industrial production of autonomous vehicles came from the series of Grand and Urban challenges sponsored by the Defense Advanced Research Projects Agency (DARPA) (Buehler *et al.*, 2007, 2009). Today, semi-autonomous advanced driver assist systems (ADASs) are commonly available in high-end mass-market vehicles. Although fully autonomous driving is yet to be realized, promising progress has been seen in many industrial prototypes, such as Waymo's fleet of cars, which have completed over 20 million test miles since 2009, achieving an average of 11,017 miles between disengagements[1] in 2018 (Waymo, 2020; Crowe, 2019b).

Existing work on autonomous-driving systems can be classified as either a learning-based end-to-end system, considered a black-box that receives sensory inputs and generates the final control commands, or a modular system that is comprised of sub-components, which is the choice applied in many systems participating in the DARPA challenges and industrial prototypes. As the class focused on here, these systems are modular autonomous-driving platforms consisting of sensors, perception systems, and planning and control systems. These components capture the environment, distill semantic information, and plan routes while controlling the vehicle, respectively.

Every component is positioned along a critical path between sensing to control, and is subject to time constraints that are just as important as producing accurate results. Vehicles currently operating on the road have been assessed by safety certification authorities, *e.g.*, the National Highway Traffic Safety Administration (NHTSA), that follow functional safety standards, such as ISO 26262 (ISO, 2011). Although these safety standards are yet to be adopted for autonomous vehicles, the process for safety certification will be crucial to making these become trusted modes of transportation for daily use. The urgency in achieving certified autonomous driving is exemplified by recent interest in "real-time" computer vision.

[1]While this metric is limited, it remains the only one used for measuring technological progress in autonomous driving. Disengagement occurs when the human driver must retake control of the vehicle during situations that the autonomous system is unable to handle.

Among common sensor types—cameras, lidars, and radars—cameras are the least expensive, and they preserve features such as color and texture that other sensors cannot. Therefore, realizing full autonomy in vehicles at a reasonable cost is expected to entail deploying computer-vision algorithms. As a result, there has been intense recent interest in real-time computer vision. Unfortunately, *in interpreting the term "real time," a significant disconnect exists* between computer-vision researchers and automotive system engineers. For the former, "real time" suggests *real fast* processing on a dedicated high-end hardware platform, and for the latter, it means *predictable*, notably on more impoverished embedded hardware that must be shared by many disparate computations. The "predictable" interpretation of real time is based on automobiles being safety-critical systems. When certifying such a system, showing that certain computations *predictably occur within specified time bounds* is more important than "real fast" performance under ideal conditions.

This disconnect between real-fast computer vision and real-time safety has led to unfortunate consequences. Several highly publicized accidents involving semi-autonomous and fully autonomous vehicles operating on open roads resulted in fatalities. Post-crash analyses of these incidents revealed a critical trade-off between time and accuracy at the nexus of computer vision and real time. As one analyst concluded, "public authorities should … ensure that self-driving cars have the capacity to process all the needed information to adopt decisions *in real time*" (emphasis added) (Renda, 2018).

This observation suggests the need for *real-time certification*, which may result in the real-fast vs. real-time computer vision disconnect becoming more problematic. Today, computer-vision-based features are largely provided in settings where the driver is assumed to be able to retake full control of the vehicle. As such, these features are not yet subject to strict certification requirements. Moving forward, however, increased autonomy will come to mass-market vehicles, and the line between semi-autonomy and full autonomy will be crossed eventually, making strict certification critical. A challenge will exist for certifying computer-vision applications that are merely "real fast," as these will be difficult to accomplish, if not impossible.

On the other hand, certifiable computer-vision implementations should not diminish throughput performance, although sometimes a trade-off must be made. The throughput requirements of an autonomous system are determined by the necessity of ensuring adequate coverage and redundancy (for safety) that can be offered by a large quantity and diversity of cameras. A typical configuration in experimental automated-driving vehicles includes ten or more cameras that cover different fields-of-view, orientations, and near and far ranges. Each camera generates a stream of images at rates of 10 to 40 frames per second (FPS) depending on its function (*e.g.*, lower rates for side-facing cameras, higher rates for forward-facing ones). All these image streams must be processed simultaneously by the onboard computer-vision applications, making sufficient throughput performance essential to support the processing of multiple data streams that fully cover a vehicle's surroundings.

To address these challenging problems of enabling real-time computer vision at the application level, achieving real-time performance and high utilization of the underlying hardware platform is important. Next, we discuss a key enabler for accelerating computer vision, the graphics processing unit (GPU).

1.2 Graphics Processing Units

GPUs evolved from a fixed-function, special-purpose hardware pipeline for graphics rendering into a programmable general-purpose hardware platform, as the term general-purpose GPU (GPGPU) was coined (Owens *et al.*, 2008). GPUs became a key technological enabler that advanced the techniques of deep learning, which have existed for decades but have started growing in use recently. The massive parallelism enabled by the architecture and programming model of GPUs reduces the training processing time for deep neural networks (DNNs) and makes the inference process nearly instantaneous for real-world applications.

Taking NVIDIA GPUs as an example, rendering hardware pipelines are replaced with programmable units called CUDA (Compute Unified Device Architecture) cores that are organized into streaming multiprocessors (SMs). Each SM supports a large number of GPU threads on the fly, while a subset of these threads can be executed simultaneously. These numbers can vary across GPU generations, so are generation-specific parameters. In contrast, the programming model remains consistent over generations.

Without discussing the programming model in more detail here, which we do in Section 2.3.3, we briefly describe the basic process of a GPU program. A GPU program is launched from a CPU (or host) process as follows: **(i)** allocate memory for the GPU to use, **(ii)** copy the input data from CPU memory to GPU memory, **(iii)** execute a GPU program, called a *kernel*,[2] to process data, **(iv)** copy the results from GPU memory back to CPU memory, and **(v)** free unneeded memory. A GPU kernel is comprised of these GPU threads, which are organized into a grid of blocks, and also accesses different types of memory, including registers, shared memory, and global memory. To manage concurrent kernel execution and algorithmic serialization, kernels are submitted to CUDA streams, and those from the same stream are serialized by submission order, whereas kernels from different streams can execute concurrently.

[2]This terminology is not to be confused with an operating system (OS) kernel.

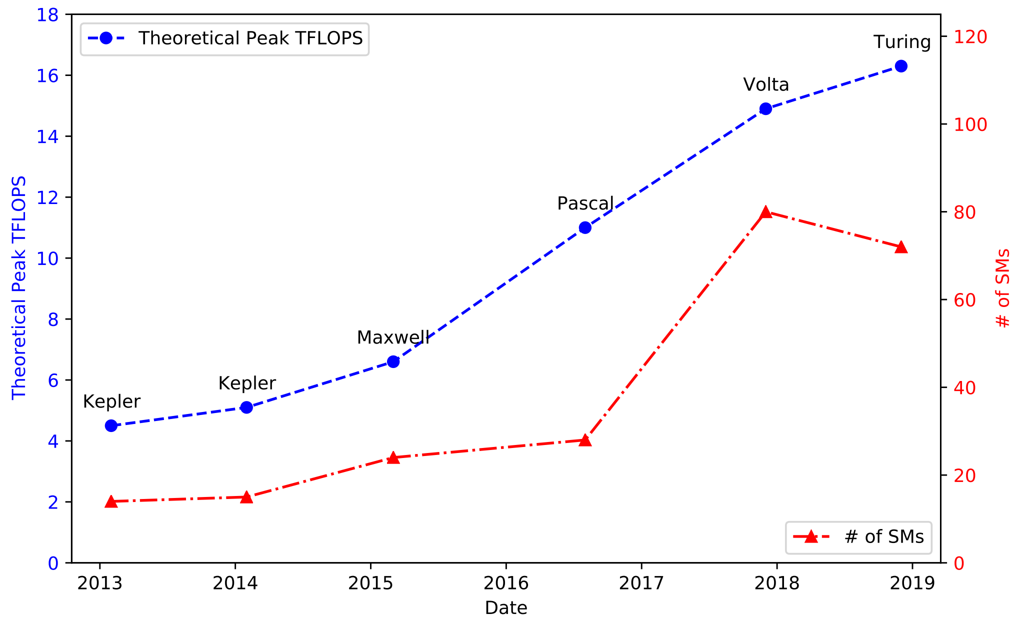


Figure 1.1: Trends in NVIDIA GPU performance and the number of SMs over generations. Each pair of a circle and a triangle data point that share the same date represent one NVIDIA TITAN GPU, annotated with its architecture code name; the circle corresponds to the performance metric on the left, and the triangle corresponds to the number of SMs on the right.

The concurrency between kernel executions becomes more important as GPUs become increasingly powerful, and the trend of increasing performance in TFLOPS (tera floating point operations per second) is illustrated in Figure 1.1. In this figure, each data point represents an NVIDIA flagship TITAN GPU product,[3] with its launch date, theoretical peak performance on the left, and the number of SMs on the right. These upward trends indicate the rapidly increasing performance of the GPU and its capacity to support additional on-the-fly GPU threads. Note that each SM supports $2,048$ threads across all GPU architecture generations. Although from the Volta to Turing architectures, the number of SMs decreased, the performance continued to climb because the amount of global memory doubled, leading to more opportunities for concurrency among memory-intensive GPU applications.

Similar trends are followed by product lines of embedded GPUs, which are comparatively impoverished. Nonetheless, it further emphasizes the necessity of sharing GPUs because processing cycles should not be wasted, as their use in embedded environments is more sensitive to the size, weight, power, and cost issues. Even though a GPU kernel utilizes all resources most of the time during execution, it may leave the GPU partially idle once its threads start exiting.

[3]Because the model names are similar, we annotate the data points with the architecture code name, and the points with the same date represent the same products.

While sharing GPUs is important, the process incurs challenges for real-time systems. The design principle of GPUs is throughput-oriented, given their origin of accelerating graphics rendering, only later being extended to the application of general-purpose computation, such as neural-network training. The former use case pursues "real-fast" performance, such as generating 30 frames per second for animation rendering, while the latter bears the throughput of finishing the training process with a large number of images over hours or even days. Guided by this performance requirement, the design of GPUs often sacrifices real-time predictability in favor of throughput. For example, GPU kernels from multiple processes are executed in a multi-programmed manner, resulting in unpredictable performance that depends on other co-scheduled workloads (Otterness *et al.*, 2017b; Capodieci *et al.*, 2018). Other forms of execution apply different scheduling rules, such as executing kernels from multiple threads in the same address space simultaneously when possible. However, these scheduling rules are undisclosed, including the condition of concurrency and execution order.

Closed-source software stacks and unknown hardware details have hindered research using NVIDIA GPUs in many aspects. Given their need for high efficiency in utilizing the hardware platform, embedded systems often require internal and low-level software and hardware information, which must be obtained through reverse engineering efforts for NVIDIA GPUs. The findings obtained in this way, such as scheduling rules, are rarely perfectly accurate due to the possible existence of edge cases. Even though they can be reaffirmed through large-scale experiments, they remain vulnerable to changes from product updates, so they are limited to short-term applicability. While we encourage openness that eliminates these issues, we perform reverse engineering to move forward, as this information is essential for sharing GPUs in real-time systems, as discussed below.

1.3 Real-Time Systems and GPU Scheduling

We mentioned the required temporal correctness and predictable performance of real-time autonomous-driving systems. To achieve them, we require a solid real-time analysis foundation upon which we can derive real-time guarantees. Although this has been well-studied for CPUs, little work is available for GPU scheduling, especially with shared GPUs. In this section, we first review several basic real-time concepts, which are introduced in greater detail in the Chapter 2. Then, we motivate the necessity of a novel task model and analysis for shared GPUs.

Most real-time applications are embodied in recurrent *tasks*, such as the repetitive capture of new images. A classic abstraction for recurrent tasks is the *sporadic task model* (Mok, 1983), which is applied in this dissertation. A *sporadic task* repeatedly releases *jobs* with a minimum time separation called a *period*, and these jobs are subjected to finishing before their *deadlines*. Each task has a *utilization* that represents the fraction of processor capacity sufficient for meeting its timing constraints. With the timing parameters of the period, deadline, and utilization, a task model can express the timing requirements of a *task system* (or *task set*) that is comprised of a set of tasks.

The sporadic task model is analyzed to validate temporal correctness, *i.e.*, to determine if a task system is *schedulable*—if it always satisfies all timing constraints when scheduled by a particular scheduler. These timing constraints are categorized as hard real time (HRT) or soft real time (SRT), where the former requires all deadlines to be met, and the latter tolerates deadline misses. Various definitions of SRT exist, and we focus on one that requires a bounded response time, which is the time difference between the finish time and release time of a job. The concept of *schedulability* is a core aspect of real-time analysis, as an important goal of real-time analysis is to provide *schedulability tests* that determine if a task system is schedulable. These types of tests are specific to the scheduling algorithm applied.

For a task system to be schedulable, its *total utilization* cannot exceed the entire capacity of a platform. However, a task system with a total utilization less than the capacity of the platform may still not be schedulable due to *capacity loss* incurred by non-optimal scheduling algorithms or real-workd overheads.

Algorithmic non-optimality-induced capacity loss, or algorithmic capacity loss, is incurred by all previous work on schedulability analysis for real-time GPUs. This issue is partly due to the GPU programming model and the native low-level scheduler available on GPU hardware. We demonstrate this inevitable capacity loss for the current NVIDIA GPUs in Section 4.2.3, and it is more severe when GPUs are managed as resources (Elliott *et al.*, 2013) or limited to exclusive access (Capodieci *et al.*, 2018). Specifically, with resource-based management (Elliott *et al.*, 2013), executing GPU code introduces *task suspensions*, which are typically handled with *suspension-oblivious analysis* (Brandenburg and Anderson, 2010) that analytically views the suspension time as CPU computation time, resulting in significant processing-capacity loss.

Additionally, these works have to intervene with respect to GPU kernel launches at runtime, at the driver or hypervisor levels, to enforce customized scheduling policies. In contrast, we tackle the problem of deriving schedulability analysis differently by aiming for the native NVIDIA GPU scheduling rules, which must first be revealed through reverse engineering because we can only demonstrate the predictability of a *known* scheduler, *i.e.*, one that manages GPU kernels that share GPUs. Such predictability then must be proved through real-time schedulability analysis that is conducted on models, *e.g.*, the sporadic task model, just as with CPUs. However, the difference from the architecture and programming model of CPUs necessitates a new task model for GPU tasks. From these goals, we present our thesis statement in the following.

1.4 Thesis Statement

Computer-vision applications in autonomous-driving systems are subjected to real-time requirements and demand high-throughput performance. Existing computer-vision frameworks that lack real-time concepts and underutilize the underlying hardware platforms must be reconsidered. We exploit pipelining and parallelism in GPU-accelerated computer vision for improved throughput with bounded response times and good accuracy.

The real-time community has investigated the benefits of GPUs. However, prior schedulability-analysis research was subject to algorithmic capacity loss in the scheduling controls through middleware/OS-level management. We remove this management layer and target the native GPU scheduler. Furthermore, we enable GPU sharing to improve resource utilization and derive real-time schedulability analysis based upon the revealed GPU scheduling rules to guarantee predictable performance for tasks that share GPUs.

This usage of shared GPUs is applied to our improved computer-vision framework that supports the thesis of this dissertation as follows:

> *Capacity loss in GPU-using real-time task systems can be lessened through sharing GPUs. Such sharing can be guaranteed with bounded response times. Furthermore, throughput can be improved without compromising such guarantees by exploiting pipelining and parallelism in real-time tasks. These real-time guarantees of tasks on shared GPUs are enabled with a new model for GPU execution and corresponding schedulability analysis.*

1.5 Contributions

The contributions of this dissertation are briefly summarized in the following subsetions, which outline our efforts towards producing a study of NVIDIA GPUs, a model of GPU execution, a response-time bound analysis, a computer-vision framework, and a case study.

1.5.1 A Study of NVIDIA GPUs beyond Official Documentation

To share GPUs and derive a schedulability analysis for GPU tasks managed by the native NVIDIA GPU scheduler, the first step is to determine the details of the scheduler. In addition to undisclosed scheduling details, interactions between CPUs and GPUs, such as kernel launches, suffer from problematic synchronization issues that behave differently with the various ways to execute GPU tasks. We examine these GPU execution approaches to determine which is appropriate for executing real-time tasks.

As discussed above, the public documents from NVIDIA are incomplete in describing GPU scheduling details, and the software stack is closed-source. For the goal of sharing GPUs and enabling simultaneous execution, the GPU scheduling rules must be identified. Specific details to be resolved include the scheduling order of GPU kernels from multiple CUDA streams, how resources in GPUs are allocated to these GPU kernels that impact GPU scheduling, and how other special programming features influence the scheduling. Taking a team effort,[4] these questions are answered comprehensively through reverse engineering and are included in our study results in Chapter 3.

In addition to the scheduling rules, we expose pitfalls pertaining to the synchronization between CPUs and GPUs. For any task consisting of a combination of CPU and GPU computations, synchronization points exist (*e.g.*, a CPU program must wait until a GPU produces a result). Synchronization inherently leads to blocking terms in scheduling analysis, and we realized that synchronization blocking occurring in the GPU task is not straightforward. Further, some forms of synchronization lead to a significant capacity loss for both CPUs and GPUs. We constructed experiments that expose these synchronization effects and describe them in Chapter 3 along with pitfalls the unaware programmer may encounter.

We realized a fundamental trade-off exists for designing real-time tasks that use a GPU. A conventional choice is to write and execute the task program as an operating system (OS) process in its own non-shared address space that provides cross-task memory isolation. If this choice is used, however, the NVIDIA GPU programming environment does not permit concurrent computations on the GPU, even if sufficient resources are available. Depending on how GPU programs are organized and written, this can lead to capacity loss on the GPU. An alternate choice is to write and execute a task as a schedulable thread that shares a process address space with other task threads. While cross-task memory isolation is lost, the GPU programming

[4]Tanya Amert, Nathan Otterness, and I collaboratively published a series papers investigating NVIDIA GPUs. Our individual contributions will be stated in Chapter 3.

environment provides mechanisms that allow concurrent computations on the GPU. NVIDIA provides a third option that leverages a middleware environment and is claimed to provide the best approach offering memory isolation and enabled concurrency. We performed a case study using exemplar computer-vision tasks in autonomous vehicles to evaluate these trade-off options, with our results and guidelines presented in Chapter 3.

1.5.2 A Model of GPU Execution

The above contributions provide a foundation for understanding GPUs, allowing us to further establish an abstract model of GPU execution. As with CPUs, a task model is created by considering the hardware architecture and programming model that reflect important aspects of GPUs. We extend the existing classic sporadic task model with additional parameters to describe GPU tasks, as presented in Chapter 4. These parameters are combined to express information about the parallel execution of tasks on GPUs.

As previously mentioned, NVIDIA GPUs employ scheduling rules that inherently sometimes cause unnecessary idleness of its processing units, implying a fundamental capacity loss when trying to ensure response-time bounds for GPU tasks. We express such a loss by providing a *total utilization bound* that is tight with a counterexample, a task system that has total utilization matching this bound but unbounded response times. We also demonstrate that capacity loss can be extreme if intra-task parallelism is forbidden. These two constraints of the utilization bound and intra-task parallelism establish a schedulability condition, and any task system satisfying this condition has a bounded response time.

1.5.3 A Response-Time Bound Analysis for Applications Sharing GPUs

Based on the revealed GPU scheduling rules and newly proposed model of GPU execution, we present the first response-time bounds for tasks that share GPUs. We apply similar analysis techniques proposed previously (Liu and Anderson, 2010; Devi and Anderson, 2005, 2008; Erickson *et al.*, 2014; Yang and Anderson, 2014a), including the directed-acyclic-graphs (DAGs) based analysis for heterogeneous platforms to derive end-to-end response-time bounds for our applications. These applications are decomposed into nodes of either CPU or GPU execution. For CPU nodes and tasks, we compute response-time bounds using existing analytical results for two real-time schedulers, the global earliest-deadline-first scheduler (G-EDF) and the global fair-lateness scheduler (G-FL), and for GPU nodes and tasks, we apply our analysis approach from Chapter 4. The end-to-end response time is then calculated by accumulating the response-time bounds of the nodes along the critical path of the DAG.

1.5.4 A Computer-Vision Framework Providing Improved Throughput

As stated above, a fundamental challenge is to design a computer-vision framework that supports independent streams of images from multiple cameras. Because the hardware for embedded systems is constrained, enabling the effective utilization of every part of the system by computer-vision applications is essential.

We focus on the demanding computer-vision application needed by autonomous cars of the timely detection and recognition of objects in the surrounding environment using convolutional neural networks (CNNs). Many CNN implementations similarly seem to have limited throughput and poor GPU utilization. As explained further in Chapter 5, the root of the problem lies in viewing processing as a single run-through of all the processing layers comprising a CNN. However, in processing an image sequence from a single camera, each layer's processing of image i is independent of its processing of image $i+1$. If the intermediate per-image data created at each layer is buffered and sequenced, there is no reason why successive images from the same camera cannot be processed in different layers concurrently.

This motivates us to consider the classic *pipelining* solution to sharing resources for increased throughput.[5] While pipelining offers the potential for greater parallelism, concurrent executions of the same layer on different images remain precluded. Allowing this may be of limited utility with one camera, but the potential exists for further increasing throughput substantially when processing streams of images from multiple independent cameras. This motivates us to extend pipelining by enabling each layer to execute in *parallel* on multiple images.

We altered the design of an existing CNN framework, called Darknet (Redmon, 2016), to enable pipelined execution, with parallel layer execution as an option, in the CNN. We also implemented a technique whereby different camera image streams are combined into a single image stream by combining separate per-camera images into a single composite image. This compositing technique can greatly increase throughput at the expense of some accuracy loss. We also enabled the balancing of GPU work across the platform's two GPUs,[6]

[5]For example, instruction pipelining in processors can fully utilize data-path elements and increase throughput.

[6]We use the NVIDIA DRIVE PX Parker AutoChauffeur, also called 'DRIVE PX2' or just 'PX2'.

which have differing processing capabilities. The CNN variants arising from these options and various other implementation details are discussed and evaluated in Chapter 5.

In summary, we provide an effective solution for a challenging problem faced by the autonomous-vehicle industry by designing CNN systems that provide the throughput, timeliness, and accuracy for computer-vision applications used in automated driving. By rethinking how the CNN is executed, we show that its throughput can improve by more than twofold with a minor increase in latency and no change in accuracy. We also show that throughput can be further improved with proper GPU load balancing. Finally, we show that the compositing technique can enable high throughputs of up to 250 FPS on the PX2. While this technique does incur some accuracy loss, we show that such loss can be mitigated by redoing the "training" CNNs require. We claim these results are sufficiently fundamental to guide CNN construction for similar automotive computer-vision applications on other frameworks or hardware.

1.5.5 A Case Study Evaluating Analytical and Empirical Real-Time Performance

We conduct a case study to evaluate analytical and empirical real-time performance with real-world applications by applying our analysis described previously. In particular, we address the limitation of coarse-grained scheduling from prior works on a computer-vision standard, OpenVX, which specifically targets computer vision in real-time embedded systems.

In prior work, our research group partially addressed the issues of enabling real-time OpenVX by proposing a new OpenVX variant in which individual graph nodes are treated as schedulable entities (Elliott *et al.*, 2015; Yang *et al.*, 2015). This variant allows greater parallelism and enables the computation of end-to-end graph response-time bounds. However, graph nodes remain high-level computer-vision functions, which is problematic for two reasons. First, these high-level nodes still execute sequentially, so some parallelism remains inhibited. Second, such a node will typically involve executing on both a CPU and a GPU. When a node accesses a GPU, it suspends from its assigned CPU. Suspensions are notoriously difficult to handle in schedulability analysis without inducing significant capacity loss, as discussed above. We show that these problems can be addressed through more fine-grained scheduling of OpenVX graphs.

We show how to transform the *coarse-grained* OpenVX graphs proposed in our group's prior work (Elliott *et al.*, 2015; Yang *et al.*, 2015) to *fine-grained* variants in which each node accesses either a CPU or a GPU, but not both. Such transformations eliminate the suspension-related analysis difficulties at the expense of minor overheads caused by the management of data sharing. Additionally, our transformation process exposes opportunities for new potential parallelism at many levels. For example, because we decompose a coarse-grained OpenVX node into finer-grained schedulable entities, portions of such a node can now execute in parallel. Also, we allow not only successive invocations of the same graph to execute in parallel but even successive invocations of the same *node*.

For our case-study experiments conducted to assess the efficacy of our fine-grained graph-scheduling approach, we considered six instances of an OpenVX-implemented computer-vision application called HOG (histogram of oriented gradients), used in pedestrian detection, as scheduled on a multicore and GPU platform. These instances reflect a scenario where multiple camera feeds must be supported. We compared both analytical response-time bounds and observed response times for HOG under coarse-grained vs. fine-grained graph scheduling. We found that bounded response times could be guaranteed for all six camera feeds only under fine-grained scheduling based on our analysis. In fact, under coarse-grained scheduling, just one camera could (barely) be supported. We also found that the observed response times were substantially lower under fine-grained scheduling, and the overhead introduced by converting from coarse-grained to fine-grained had a modest impact. These results demonstrate the importance of enabling fine-grained scheduling in OpenVX if real time is *really* a first-class concern.

1.6 Organization

For the remainder of this dissertation, we first provide a comprehensive background survey in Chapter 2 of autonomous driving research, applications of computer vision, GPU hardware platforms, and real-time systems research with a focus on real-time GPUs and DAG scheduling. Following this background of relevant work, we present our investigation of NVIDIA GPUs in Chapter 3 and a new model of GPU execution with schedulability analysis for real-time tasks that share GPUs in Chapter 4. We next describe our computer-vision framework design in Chapter 5, along with its experimental evaluation. Finally, we present a case study that evaluates the analytical and empirical real-time performance using our proposed analysis for GPU applications in Chapter 6. We conclude and discuss future work in Chapter 7.

CHAPTER 2: BACKGROUND

In this chapter, we survey related prior works and provide background information for this dissertation. We first review the development of autonomous-driving systems with a focus on computer-vision applications as our research context. We then describe the usage of GPUs versus other accelerators for computer-vision applications and the necessary fundamentals of GPUs, including the hardware architecture and GPU programming model. Lastly, we provide the fundamentals of real-time systems, and review the state-of-the-art literature on modeling and analyzing GPUs in real-time systems.

2.1 Autonomous Driving

In this section, we first review state-of-the-art progress in autonomous-driving systems in both industry and academia. We then summarize typical system architectures of autonomous driving and explain each primary component. To motivate our research, we focus on explaining the safety- and timing-critical certification issues that relate to our work.

2.1.1 State of the Art

Autonomous driving is expected to reduce traffic accidents and roadway fatalities, of which 94% are caused by human errors, as reported by the National Highway Traffic Safety Administration (NHTSA) in a recent technical report (National Highway Traffic Safety Administration, 2015). In addition to reducing vehicle crashes, autonomous driving can improve the mobility of people who have disabilities that relate to driving, benefit the environment by improving fuel efficiency and reducing emissions during congestion, give time back to commuters, and even reshape the layouts of our cities. With these benefits, it is projected that the nationwide adoption of autonomous driving could lead to nearly $800 billion USD in annual social and economic benefits in America by 2050 (Montgomery *et al.*, 2018).

Although no industry players are even close to achieving fully-autonomous driving (Ackerman, 2017), semi-autonomous *advanced driver assist systems (ADASs)* are now commonly available in high-end mass-market vehicles; examples include Tesla Autopilot, Volvo Pilot Assist, Mercedes-Benz Drive Pilot, and

Cadillac Super Cruise (Hughes, 2017). These ADASs include functionalities such as adaptive cruise control (ACC), forward collision warnings (FCW), lane departure warnings (LDW), lane-keeping assistant systems (LKAS), and intelligent speed assistance (ISA). Each of these functionalities alone only qualifies as Level 1 automation, driver assistance, as defined in the taxonomy by the Society of Automotive Engineers (SAE) (SAE, 2018). When combined, these ADASs may enable Level 2, partial automation, which allows self-driving functionality in certain conditions such as on the highway, with the driver monitoring the environment and being ready to take over at any time as a fallback. With Level 0 denoting no automation, Levels 2 and below require the driver to monitor the environment at all times.

In contrast, Levels 3 and above encompass the classes of autonomous-vehicle (AV) systems. Level 3, conditional automation, allows the driver to focus on tasks other than driving, but requires the driver to be ready to quickly take over *upon request*. For instance, Audi's AI Traffic Jam Pilot achieves Level 3, at speeds below 37 miles per hour (MPH) (Ross, 2017).

Levels 4 and 5 require operating autonomously without relying on the driver as a fallback. Level 4, high automation, is limited to certain operational design domains such as highways, whereas Level 5, full automation, must be operational anywhere in any condition. Level 4 and Level 5 autonomous vehicles are still unavailable in the market as of today, and are being developed by many companies. Notably, one of the most press-worthy advancements is the fleet of Waymo cars, which have been drive-tested for over 20 million miles since 2009 (Waymo, 2020); in 2018, this fleet achieved an average of 11,017 miles traveled between disengagements, where emergency overrides by a human operator were enforced (Crowe, 2019a,b). However, this metric for distance between disengagements has been criticized by autonomous-driving companies, including Waymo themselves (Fernandez, 2020), because it is misleading when used for measuring progress as miles tested in different settings are hardly comparable, *e.g.*, empty highway vs. busy urban areas.

The commercialization and production development of autonomous driving were ignited by a series of the Defense Research Advance Projects Agency (DARPA) sponsored challenges: the Grand Challenges in 2004 and 2005, and the Urban Challenge in 2007 (Buehler *et al.*, 2007, 2009). Autonomous vehicles participating in the Grand Challenges were expected to drive through unrehearsed off-road terrain without manual intervention. In its first event in 2004, none of the 15 racing teams finished more than 5% of a 142-mile course on the desert within the ten-hour time limit (Thrun *et al.*, 2006). In the second event in 2005, five out of 23 racing teams finished, and "Stanley" (from Stanford University), finishing in under seven hours, won first place (Thrun *et al.*, 2006), six minutes before the second place of "Sandstorm" and

"H1ghlander" (Urmson *et al.*, 2006). These Grand Challenges spurred innovation in the areas of perception, collision avoidance, vehicle control, and others.

If driving on a desert as required by the DARPA Grand Challenges is not considered relevant enough to our daily life, the third event—the DARPA Urban Challenge—should be. The Urban Challenge set the goal to traverse through traffic in urban settings without violating California traffic rules (Buehler *et al.*, 2009; Urmson *et al.*, 2008). Eleven of the 53 teams passed the National Qualification Event, and competed in the Urban Challenge Final Event. In the end, six teams finished the challenge, with "Boss" of Carnegie Mellon University winning first place.

In all of these production and prototype cars, cameras and computer-vision technologies were used alongside processing techniques for other sensors such as lidars and radars.

The DARPA Challenges revitalized interest in autonomous driving, for which the initial interest can be traced back to the late 1980s, when Carnegie Mellon University's Navigation Laboratory (Navlab) presented a series of Navlab vehicles with basic autonomy (Thorpe *et al.*, 1988, 1991a,b). The basic system architecture developed by Thorpe *et al.* (1991b) is common in today's approaches. In 1988, Thorpe *et al.* presented a van equipped with one TV camera and one laser rangefinder to enable road-following and collision detection and avoidance (Thorpe *et al.*, 1988). Later, Pomerleau (1989, 1992) presented the neural network ALVINN, which generates navigation direction outputs from camera and laser rangefinder image inputs. At the end of the Navlab project, an important milestone was achieved as Jochem and Pomerleau (1995) finished "No hands across America," a trip from Pittsburgh, PA to San Diego, CA, with the steering direction controlled based on the image inputs and the throttle and brake handled by human.

During the same period, the Eureka Project PROgraMme for a European Traffic of Highest Efficiency and Unprecedented Safety (PROMETHEUS), the largest R&D autonomous-driving project at the time, was carried out and spread broad interest in Europe in active safety systems (Williams, 1988; Ibaez-Guzmn *et al.*, 2012). Dickmanns and Zapp (1987) first demonstrated the early results of vision-based guidance for high-speed autonomous driving on a well-structured highway with one-way traffic. In the PROMETHEUS project's final demonstration, two vehicles, "VITA II" of Daimler-Benz (Ulmer, 1994) and "VaMoRs-P" of Bundeswehr University Munich (Dickmanns *et al.*, 1994), demonstrated autonomous driving at a speed up to 130 km per hour in public traffic on highways (Dickmanns *et al.*, 1997).

The development of autonomous driving has been ongoing for some time, and has provided a valuable reference for autonomous-driving system architectures, which we review next.

2.1.2 Autonomous-Driving System Architecture

In this section, we review autonomous-driving system architectures that have been used in state-of-the-art proof-of-concept research projects, and then focus on modular systems, for which we provide a detailed review of each sub-component, and the connections and interfaces between them. Of these components, the perception component is highlighted in our review, given our focus on computer-vision applications.

2.1.2.1 Architecture

Many of the aforementioned autonomous systems (Section 2.1.1) use a *modular system* architecture, in which components that provide different functionalities are encapsulated separately and are related as the nodes in a connected graph, from the sensor input to the vehicle control (Behere and Törngren, 2016). Another approach is learning-based *end-to-end driving systems*, which were demonstrated very early on (Pomerleau, 1989) and emerged again recently. These end-to-end driving systems are usually neural networks that receive sensor inputs through the network to directly generate control commands.[1] We briefly review the history and state of the art of end-to-end driving systems, and then discuss why we instead focus on modular systems.

End-to-end driving systems. The earliest end-to-end driving system was ALVINN, presented by Pomerleau (1989). ALVINN is a neural network that generates directions for lane-following with inputs from a camera and a laser rangefinder. It was trained using supervised learning with simulated road images, and it demonstrates lane-following on real roads under certain conditions. Muller *et al.* (2006) extended its autonomous capability to off-road obstacle avoidance using stereo cameras and a convolutional neural network (CNN). The results were significantly improved in a similar approach by NVIDIA, in which a larger convolutional network and training dataset were used (Bojarski *et al.*, 2016), as demonstrated on highways and parking lots. These approaches are categorized as *imitation learning*, as they accept human input as part of the training data, and the goal of the system is to drive like a human. However, they fail in situations where turning guidance is needed at intersections, thus Codevilla *et al.* (2018) proposed *conditional imitation learning* to add training signals about a driver's intention to solve the ambiguity.

An alternative approach is *reinforcement learning* (RL), the goal of which is to maximize future rewards under predefined reward policies (Mnih *et al.*, 2015). Sallab *et al.* (2017) demonstrated deep RL in a simulator

[1] Often called *pixel-to-torque* systems.

using recurrent neural networks (RNNs) that integrated information and handled partially observable scenarios. Kendall *et al.* (2019) then demonstrated real-road lane-following trained with a few episodes[2] using RL for the first time.

Although these learning-based end-to-end driving systems have shown promising progress, without the necessary understanding of the functions learned inside these neural networks, safety certification for neural networks is difficult if not impossible. Certification for safety-critical systems (*e.g.*, autonomous-driving systems) is rigorous, and follows principles to guarantee correct specification and that the implementation satisfies the specification (Cheng *et al.*, 2018a). Adapting this certification process to neural networks presents new challenges. The specification for a neural network is implicit in its training data, which results in difficulties for design-time analysis and reviews, and product acceptance tests. Neural networks are generally treated as no more than a black box due to their lack of interpretability, thus making it difficult to trace implementation from requirements to coding. With the common usage of non-linear, piecewise activation functions for neurons, criteria such as those for decision coverage are intractable, as the conditional branching options are exponential to the number of neurons (Cheng *et al.*, 2018a). These challenges have been investigated in prior works (Bedford *et al.*, 1996; Rodvold, 1999; Burton *et al.*, 2017; Doshi-Velez and Kim, 2017; Salay *et al.*, 2017; Olah *et al.*, 2018; Cheng *et al.*, 2018b; Aravantinos and Diehl, 2018), but have not been fully addressed yet. Therefore, we focus on modular systems that are less affected by these certification issues in our work.

Modular systems. Modular systems are structured as a graph of connected modules/components in a system (Behere and Törngren, 2016; Yurtsever *et al.*, 2019). Such a decomposition of functionalities, compared with end-to-end systems, provides finer-grained implementation control and better understandability of the whole system. It provides advantages such as the convenience of transferrable knowledge about each sub-component, independent problem-solving for developing each sub-component, multiple end-to-end paths for tasks of different levels of criticality (*e.g.*, a path for emergency handling), and other advantages of a decoupled modular design. However, these advantages have a price: the disadvantages of modular systems include communication overhead between components, scheduling and synchronization issues, and the nature of error propagation in complex systems (McAllister *et al.*, 2017).

[2]Episodes are independent training processes that maximize the reward, *e.g.*, one driving trip for an autonomous vehicle.

We use a functional architecture presented by Behere and Törngren (2016) as a reference to discuss typical modular autonomous driving systems, as the core functions are common to other systems as well (Thrun *et al.*, 2006; Urmson *et al.*, 2008; Montemerlo *et al.*, 2008; Guizzo, 2011; Wei *et al.*, 2013; Broggi *et al.*, 2013; Ziegler *et al.*, 2014; Akai *et al.*, 2017; Maddern *et al.*, 2017; Yurtsever *et al.*, 2019). These include perception functions such as sensor fusion, semantic understanding, and world modeling; planning and control functions such as route planning, reactive control, fault management; and vehicle platform operation (steering and throttling) (Behere and Törngren, 2016). We review *sensor hardware*, *perception*, and *planning and control* in the following.

2.1.2.2 Sensor Hardware

On autonomous vehicles, multiple sensors provide the necessary understanding and coverage of the surrounding environment, and safety redundancy on autonomous vehicles. Typical sensors include cameras, and radar and lidar sensors. Here we summarize the working mechanisms, advantages, and disadvantages of each.

Cameras. Cameras are widely used in autonomous-driving systems and ADASs. A typical configuration in a contemporary experimental autonomous vehicle includes ten or more cameras that cover different fields of view, orientations, and near/far ranges. Each camera generates a stream of images at rates ranging from 10 to 40 frames per second (FPS), depending on its function (*e.g.*, lower rates for side-facing cameras, higher rates for forward-facing ones). All streams must be processed simultaneously by different computer-vision applications, such as object detection (cars, people, bicycles, signs), scene segmentation (finding roads, sidewalks, and buildings), object tracking, *etc*.

In addition to their advantage of a proliferation of computer-vision algorithms, cameras are more economical than radars and lidars. 2D image sensing lacks depth information, but this issue is mitigated by using stereo cameras or vision-based depth algorithms (Saxena *et al.*, 2006; Eigen *et al.*, 2014; Laina *et al.*, 2016; Cheng *et al.*, 2018c). However, the accuracy of vision-based perception is drastically limited by weather conditions (*e.g.*, rain or snow) or illumination conditions (*e.g.*, glaring sunlight or night illumination). The image quality under severe conditions is currently a pressing problem.

Radars. Radar (radio detection and ranging) is used to estimate the distances and relative velocities of objects. It consists of a transmitter that emits pulsed electromagnetic waves at the speed of light, and a receiver that receives waves reflected by the object. The distance and velocity of objects can be calculated by measuring the time between the transmission and reflection. Compared with cameras or lidar, radar is lightweight and less affected by illumination or weather conditions. However, emissions generated by radars on different vehicles may interfere with each other (Brooker, 2007). Moreover, a large number of radars is required to adequately cover a range of distances and fields of view: 0.15–30 m, $\pm 80°$ for short-range, 1–100 m, $\pm 40°$ for medium-range, and 10–250 m, $\pm 15°$ for long-range radars (Patole *et al.*, 2017). Although radar generates less data, implying fewer processing resources required, its detection results are subject to lower angular accuracy, *i.e.*, it is less capable of resolving two objects at the same distance but at a different azimuth (Rasshofer and Gresser, 2005; Patole *et al.*, 2017; Kocić *et al.*, 2018).

Lidars. Lidar (light detection and ranging) operates as an optical analog to radar (Warren, 2019). It also measures the reflection delay of time-of-flight transmission from objects and produces a point cloud of 3D data points of its surroundings. It is not affected by illumination conditions, as it is operational under both day and night conditions. Moreover, it achieves higher accuracy (± 0.1 m) than radar within a distance of 200 m (Yurtsever *et al.*, 2019) by using infrared light waves (mostly in the 900 nm wavelength range (Kocić *et al.*, 2018)) instead of radio waves (in the 4–12 mm wavelength range (Shaffer, 2017)). However, this results in a sensitivity to environmental conditions such as fog, rain, snow, and dust (Rasshofer and Gresser, 2005).

GPS receiver and IMU. Other sensors also provide essential vehicle status information, such as Global Positioning System (GPS) receivers for obtaining positioning information and inertial measurement units (IMUs) for monitoring vehicle movement and velocity. These are useful for localization, navigation, and ego-motion (vehicle's movement relative to the scene) estimation tasks (Sukkarieh *et al.*, 1999).

2.1.2.3 Perception

Perception refers to the geometric and semantic understanding of the vehicle's surroundings from the sensory inputs, which are processed or fused to build a world model that feeds into the rest of the system. This world model consists of layers of information from the static to the dynamic (Papp *et al.*, 2008): (i) static objects such as roads and traffic signs, (ii) static objects that contain dynamic information such as traffic lights, (iii) temporary objects such as construction zones, and (iv) dynamic objects such as vehicles, pedestrians, bicyclists, *etc.*, which must be tracked for behavior analysis and trajectory prediction (Behere and Törngren, 2016). In addition to building a world model, the perception system is also responsible for estimating vehicle state and ego-motion and localizing the vehicle relative to a reference map. Localization can be considered a separate component, but given its reliance on sensory inputs and its goal of positioning the vehicle in the world model, we conveniently view it as part of the perception system. We briefly describe these perception tasks.

Localization. Localizing the vehicle is a prerequisite for navigation, and it helps in other perception tasks such as locating traffic lights. The three standard methods for localization are GPS-IMU fusion, simultaneous localization and mapping (SLAM), and map-based localization (Yurtsever *et al.*, 2019).

Inertial measurement using IMUs is used for dead reckoning, for which the error accumulates, thus it must be corrected by absolute positioning information provided by GPS (Zhang *et al.*, 2012). This GPS-IMU fusion approach is efficient in open areas, but suffers from low accuracy where GPS signals are weak.

SLAM generates a map of an unknown environment through online depth sensing while localizing the vehicle within the map simultaneously. It does not require a pre-built map, but can be less efficient compared to GPS-IMU fusion due to the costly process for building the map, especially in outdoor environments (Ranganathan *et al.*, 2013).

Map-based localization is essentially landmark-based matching using sensor data, including point cloud data (Levinson *et al.*, 2007; Levinson and Thrun, 2010) and images (McManus *et al.*, 2013; Wolcott and Eustice, 2014). Coarser-grained landmark information is most efficient for matching, for example, road-marker matching using lidars (Hata and Wolf, 2014) or cameras (Suhr *et al.*, 2016), and traffic-sign matching using cameras (Qu *et al.*, 2015).

Object detection. Object detection aims to determine the locations and classes of objects in camera images of the vehicle's surroundings. It is crucial to be aware of other traffic participants to avoid accidents, and of signals from traffic lights, signs, and road marks to obey traffic rules. A robust approach to object detection uses sensor fusion that combines different types of sensor data so that various sensors complement each other (Enzweiler and Gavrila, 2011; Cho *et al.*, 2014; González *et al.*, 2016; Chen *et al.*, 2017b). Lidar and radar inherently provide scale information about objects and enable 3D object detection (Zhou and Tuzel, 2018), which, however, remains challenging because of the change in an object's appearance relative to range and the sparsity of data points beyond a certain distance (Yurtsever *et al.*, 2019). However, vision-based object detection has multiple well-established approaches ranging from classical pipelines to end-to-end neural networks. Given our focus on vision-based object detection in this dissertation, we provide a detailed review in Section 2.2.

Object tracking. In a dynamic environment, an autonomous-driving system must predict the future trajectory of the surrounding objects in order to avoid collision and to make appropriate driving decisions such as maneuvering. A common object-tracking approach is to associate object detection or extracted features to object trajectories, called data association (Shi *et al.*, 1994; Wu and Nevatia, 2007; Huang *et al.*, 2008; Breitenstein *et al.*, 2009). To improve the accuracy of data association, sensor fusion is commonly used (Mobus and Kolbe, 2004; Cho *et al.*, 2014). Following data association, traditional filtering methods, such as Kalman filters (Giebel *et al.*, 2004), are applied to perform physical-model-based state estimation and actual-measurement-based state correction. A recently trending research direction for object tracking is to apply neural networks to solve the data-association step, or to apply end-to-end learning for object tracking (Leal-Taixé *et al.*, 2016; Tang *et al.*, 2016; Schulter *et al.*, 2017; Milan *et al.*, 2017). A survey on object tracking is provided in (Janai *et al.*, 2019).

Semantic segmentation. Semantic segmentation refers to pixel-level labeling to separate regions that are not well-defined by bounding boxes, and is considered particularly useful for high-level scene understanding. Semantic segmentation accuracy and efficiency have been significantly improved by using neural networks that are able to learn appropriate features, which are instead hand-tuned in traditional methods (Garcia-Garcia *et al.*, 2018). The current state-of-the-art technique is the fully convolutional network (Long *et al.*, 2015), which modifies classification models by replacing fully connected layers with convolutional ones to obtain spatial heatmaps, which are passed through a deconvolutional layer for upsampling and pixel-level labeling (Garcia-Garcia *et al.*, 2018). Garcia-Garcia *et al.* (2018) surveyed deep learning techniques for semantic segmentation. For image streams, there are methods that utilize temporal cues in frame sequences to improve the accuracy and frame-processing throughput (Shelhamer *et al.*, 2016; Tran *et al.*, 2015). Such methods introduce interesting real-time scheduling problems in the trade-off between accuracy, throughput, and latency.

Other perception tasks. Other than these core tasks, the perception system also includes other tasks such as depth estimation, road-shape estimation, vehicle state estimation, ego-motion compensation, *etc*., for which detailed descriptions and reviews are provided in (Luettel *et al.*, 2012; Hussain and Zeadally, 2018).

2.1.2.4 Planning and Control

With the produced vehicle state estimation and a world model of the environment, the planning and control system navigates and controls the vehicle accordingly. A planning and control system comprises several layers: (i) a global planner that computes highest-level routes to reach destinations, (ii) a behavior planner that follows the route and makes local driving decisions (*e.g.*, lane-following, turning, and stopping), (iii) a motion planner that continuously finds and selects motion paths to achieve local driving goals, and (iv) a low-level vehicle controller that actuates the vehicle to follow the selected motion path. We briefly review these components in this section.

Global planner. The global planner operates at the highest level in the planning system to search for routes to the destination in the route network. This route network is presented as a directed graph with nodes corresponding to waypoints, edges corresponding to road segments, and edge weights corresponding to traveling costs determined by factors such as distance and traffic conditions. With this graph representation, route planning can be reduced to a minimum-cost path search problem. However, solving this search problem in a large graph using classical approaches like Dijkstra's (Dijkstra *et al.*, 1959) or the A* (Hart *et al.*, 1968) algorithm may be impractical. A survey of other efficient solutions can be found in (Bast *et al.*, 2016).

Behavior planner. A global route cannot be followed precisely, because local conditions must be further considered. These local conditions include the behaviors or driving conventions of other vehicles, traffic signals such as stop signs, construction zones, and traffic lights. Typical scenario examples include the negotiation of unprotected left-turns and intersections. According to the perceived surrounding environment and the predicted behavior of other dynamic objects, the behavior planner makes tactical driving task decisions or selects motion goals to follow the global route in general, and may introduce dynamic workloads into the system. Surveys of state-of-the-art methods can be found in (Paden *et al.*, 2016; Badue *et al.*, 2019).

Motion planner. After the behavior planner issues the motion goal, the motion planner is responsible for generating motion paths or trajectories to achieve the goal. The generated motion paths need to be physically possible considering the vehicle's kinematic and dynamic constraints while avoiding collisions. The corresponding motion-planning problem has been well-studied in the robotics field using graph-search-based, sampling-based, interpolating-curve-based, and numerical-optimization-based approaches. A survey on motion-planning methods can be found in (González *et al.*, 2015; Paden *et al.*, 2016; Yurtsever *et al.*, 2019).

Vehicle controller. In the end, the motion path is executed by the vehicle controller, which sends commands to the steering wheel, throttle, and brakes of the vehicle to achieve lateral acceleration, longitudinal acceleration, and deceleration, respectively. The controller is also responsible for maintaining the stability of the vehicle, rejecting infeasible motion-path requests, or correcting motion errors. A survey of vehicle control techniques can be found in (Yurtsever *et al.*, 2019).

2.1.3 Summary

In summary, we have described the primary components that compose a basic functional architecture of an autonomous-driving system. Other components such as vehicle-to-vehicle and vehicle-to-infrastructure communication, diagnosis, and fault management, and human-interface components are beyond the scope of our discussion. Interested readers can refer to (Behere and Törngren, 2016; Yurtsever *et al.*, 2019).

We conclude this section with the remark that safety certification for autonomous-driving systems presents the challenge of guaranteeing the system's timing correctness, which can be complicated with sophisticated hardware platforms such as GPUs, and complex software such as computer-vision applications, which we review next.

2.2 Computer Vision for Autonomous Driving

Computer vision plays a significant role in understanding a vehicle's surroundings using inexpensive sensor devices and preserved color and texture information. We reviewed the state-of-the-art methods for various core perception tasks in Section 2.1.2.3, from which we noted a general transition from using classic computer-vision pipelines to applying neural networks. Both methods are further reviewed herein, with a focus on object detection and its accuracy metrics. We focus on object detection because it is a well-established problem in computer vision for autonomous driving and forms the basis for many other perception tasks, such as object tracking and instance segmentation. We then review commonly used computer-vision frameworks and their pros and cons for real-time embedded systems. Lastly, we focus on a computer-vision application standard, OpenVX (Khronos Group, 2019b), that specifically targets real-time embedded systems, and discuss its challenges in supporting real-time analysis, as we will address these challenges in our case-study chapter (Chapter 6).

2.2.1 Methods

Object detection consists of *localization* and *classification* of multiple objects in an image. The localization operation predicts the position and dimensions (in pixel units) of a rectangle ("bounding box") in the image that contains an object of interest. The classification operation predicts with a probability what specific class (car, bicycle, sign, *etc.*) the object belongs to. Collectively, localization and classification are here referred to as *object detection*.

The development of object detection has achieved several milestones across the stages of various classic detection pipelines and the current state-of-the-art deep-learning neural networks (Zou *et al.*, 2019). We describe these milestones and their representative algorithms.

2.2.1.1 Classic Pipelines

A classic object-detection pipeline consists of three stages: image preprocessing, region of interest extraction, and object classification (Janai *et al.*, 2019). Images are preprocessed for color normalization, scaling, color space transformation (*e.g.*, RGB to grayscale), *etc.* A region of interest is usually determined and extracted using a sliding window that searches over an image using all possible positions and scales. An object is classified by examining if a window contains any target objects.

Milestones. *Supervised learning* is a machine-learning approach that infers a function that maps an input to an output through training processes with labeled example data (Mohri *et al.*, 2018). Papageorgiou and Poggio (2000) presented one of the first sliding-window-based detectors that applied a supervised-learning model, *support vector machine* (SVM), for classification and regression analysis. Viola and Jones (2001) later extended the detector for fast face detection with speed-up techniques: image integrals for faster computation, AdaBoost for feature selection, and detection cascades to avoid computational overheads on background regions. In these early works, the *Haar wavelet*[3] feature was commonly used for its low computing resource requirement. Later, gradient-based features were widely adopted after *histogram of oriented gradients* (HOG) (Dalal and Triggs, 2005a)—a landmark detector inspired by *scale-invariant feature*

[3]Haar wavelets compute the differences between pixels of two neighboring rectangle areas.

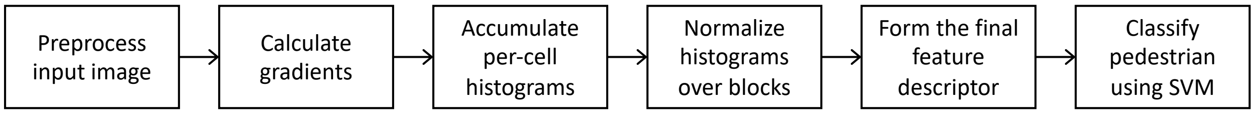

Preprocess input image
Calculate gradients
Accumulate per-cell histograms
Normalize histograms over blocks
Form the final feature descriptor
Classify pedestrian using SVM

Figure 2.1: HOG pedestrian detection pipeline.

transform (SIFT) (Lowe, 2004)—improved detection results by more than one order of magnitude compared with the Haar wavelet-based detector (Mohan *et al.*, 2001). Therefore, we use HOG as our detection pipeline exemplar and describe it in detail shortly. HOG was further extended as a *deformable part-based model* (DPM) (Felzenszwalb *et al.*, 2009), which breaks a target object into parts, *e.g.*, cars into wheels, windows, and body (Zou *et al.*, 2019), and achieved peak performance among the classic object detection pipelines.

HOG. HOG was first used for detecting pedestrians in images (Dalal and Triggs, 2005a), and it can be trained to detect other classes of objects as well. The basic principle of HOG is that the local intensity gradient distribution can characterize the appearance and shape of objects. As shown in Figure 2.1, it consists of multiple pipeline stages: it preprocesses the input image, calculates a directional gradient for each pixel, sorts gradients into histograms, normalizes lighting and contrast, and then performs classification.

The images are first divided into small regions called "cells," and each cell accumulates a histogram of orientations. Each pixel contributes a weighted vote, as a function of the gradient magnitude, to the orientation of the pixel. These histograms of cells are then normalized over a relatively larger region, called "blocks," in order to smooth the gradient magnitude differences caused by local variations in illumination and contrast. In the end, the combined histograms of normalized cells in a sliding detection window form the final feature descriptor, which an SVM classifier takes as input to determine if a pedestrian exists in a window.

The entire computation is performed at multiple image-scale levels, using successively smaller versions of the original image, so as to detect objects of different sizes. The computation workload is deterministic and immune to the number of objects in the image.

2.2.1.2 Deep Learning Methods

At present, most perception tasks use neural-network-based deep-learning methods for better accuracy and efficiency, rather than classic computer-vision pipelines. In contrast with the hand-crafted features used in classic pipelines, a deep-learning neural network can "learn" to localize and classify objects in images via a process called *supervised training* in which a large number of images labeled with "ground-truth" bounding boxes and class identifications are input to the CNN running in *training mode*. Training typically results in computing a large set of parameters (or "weights") defining how the CNN will respond when given images not in the training set (*e.g.*, images from cameras mounted on a car). A "trained" neural network can then be deployed and processes images in *inference mode*. The weights of the neural network are usually not updated during inference.[4]

In addition to the revival of deep learning with AlexNet (Krizhevsky *et al.*, 2012), which is a deep CNN that dominated the ImageNet challenge (a competition for object classification), we summarize the milestones of CNN-based object detection.

Milestones. The development of object-detection CNNs has branched into two family trees: (i) the multi-stage-region-proposal-based CNNs that first generate candidate region proposals where objects potentially exist, then extract features, and finally classify and localize objects, and (ii) the one-stage regression/classification-based CNNs that localize and classify objects in one consolidated step (Zhao *et al.*, 2019; Zou *et al.*, 2019).

(i) Multi-stage region proposal-based. As mentioned above, the CNN AlexNet marked the success of deep learning for a classification task. Following this success, the regions-with-CNN-features (R-CNN) approach was introduced (Girshick *et al.*, 2014), which naturally employs AlexNet to extract features from regions proposed by selective search (Uijlings *et al.*, 2013) for object detection. These features are then classified using *linear SVM*[5] to predict the presence of objects within each region. For positive regions, a bounding-box regressor finalizes the bounding box for the objects. R-CNN showed better accuracy than previous detectors, however, it suffered from a speed bottleneck due to the overhead of computing features repetitively for overlapping regions. To mitigate this issue, SPPNet enables one-time feature computation for the whole image (He *et al.*, 2015). Both R-CNN and SPPNet use multi-stage pipelines. With this, the process of consolidating stages began, intended for reducing storage expense and improving processing efficiency. A novel CNN architecture, Fast R-CNN, enabled simultaneous classification and bounding-box regression training by introducing multi-task loss (Girshick, 2015). Faster R-CNN then integrated the region proposal stage with the region proposal network (He *et al.*, 2015), becoming the first end-to-end CNN.

(ii) Single-stage regression/classification-based. Object detection was framed as a regression/classification problem for the first time when Redmon *et al.* (2016) introduced YOLO, with which "you only look once"

[4]Online learning requires such updates, but are beyond our discussion.

[5]SVM that performs linear classification.

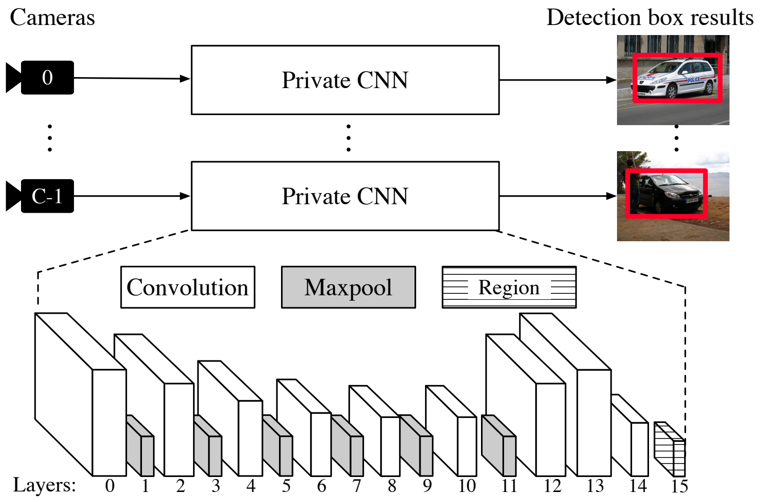


Figure 2.2: Traditional camera processing setup and the Tiny YOLOv2 network architecture. Larger boxes indicate longer execution times. Frame data flows left to right.

at an image to both localize and classify objects. We take YOLO as a representative CNN in our work, and we provide a detailed description below. Another single-stage CNN, the single-shot multibox detector (SSD) (Liu *et al.*, 2016), further improved detection accuracy and processing speed by handling objects of different sizes using multiple feature maps of various resolutions from the layers, and by discretizing the bounding box output space using a set of default bounding boxes with different aspect ratios.

You only look once: YOLO. Redmon and Farhadi continued improving YOLO in a series of studies for better accuracy and speed. We take a relatively simple one, Tiny YOLOv2 (simply "YOLO" in the remainder of this dissertation), as an example to describe the computation workload in each layer.

Tiny YOLOv2 implementation. Tiny YOLOv2 is a lower-accuracy, higher-speed derivative of YOLOv2 by the YOLOv2 authors (Redmon and Farhadi, 2017; Redmon, 2017). This version of YOLO runs roughly five times faster than the standard version of YOLO ($\geq$ 200 FPS vs. $\geq$ 40 FPS on a high-end GPU) (Redmon, 2017). The network architecture shown in Figure 2.2 is configured as it would be by default for C cameras (denoted 0 through $C-1$). In this default system, each camera uses a separate "private" instantiation of the network. Each instantiation contains nine `convolution` or convolutional layers interleaved with six `maxpool` layers and a `region` layer at the end. All YOLO networks run on the *Darknet* (Redmon, 2016) CNN framework. A *convolutional layer* convolves over the input by computing the dot product between input and filters to produce an activation map of features. A *maxpool layer* applies a non-linear transform to the input by selecting the maximum values from non-overlapping regions. YOLO's *region layer* uses the

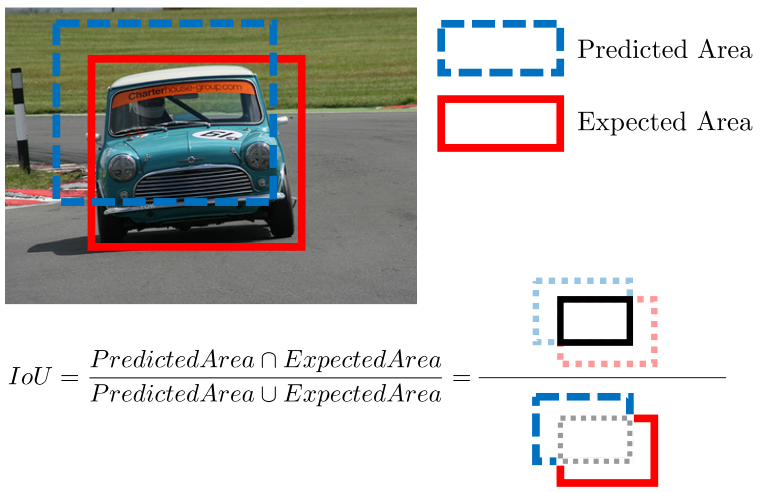

Predicted Area
Expected Area
IoU = PredictedArea ∩ ExpectedArea / PredictedArea ∪ ExpectedArea =

Figure 2.3: IoU illustrated with an image example from PASCAL VOC dataset (Everingham *et al.*, 2010).

activation maps of the last convolutional layer alongside precalculated “anchor” values to predict the output locations and bounding boxes of the objects of interest. These anchors characterize the common shapes and sizes of bounding boxes in the training data and help improve the prediction accuracy. We summarize the accuracy metrics in the following section.

2.2.2 Object-Detection Accuracy Metrics

For classic pipelines, the metrics of *false positive per window* (FPPW) and *per image* (FPPI) were used, *e.g.*, HOG was evaluated using FPPW and the dataset INRIA (Dalal and Triggs, 2005a). Recently, *mean average precision* (mAP) became a new standard. Here we provide an intuitive understanding of mAP. For a more detailed introduction, we refer the reader to a tutorial by Hui (2018), and for further details on the precision-recall curve and mAP to Section 4.2 in Everingham *et al.* (2010).

2.2.2.1 IoU, Precision, and Recall

In order to determine the precision of object detection, we first need to measure the quality of object detection, which is done using the *Intersection over Union* (IoU) measure illustrated in Figure 2.3. This is defined as the proportion of overlapping area between the predicted and expected areas of an object over their union:

$$IoU = \frac{PredictedArea \cap ExpectedArea}{PredictedArea \cup ExpectedArea}\,.$$

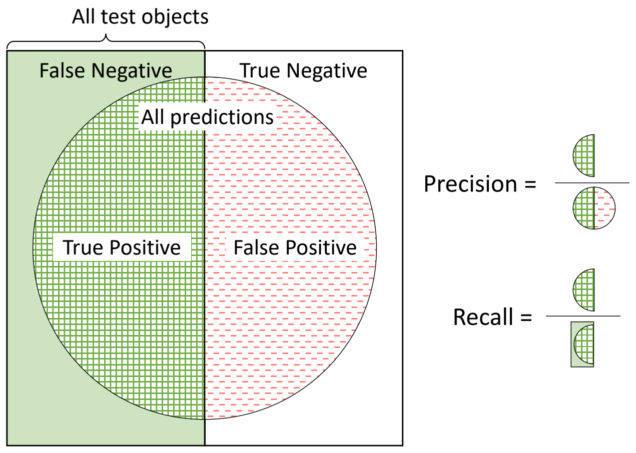


Figure 2.4: Precision and recall.

If the IoU of a prediction is greater than a chosen threshold (*e.g.*, 0.5, as specified in the PASCAL Visual Object Classes (VOC) dataset (Everingham *et al.*, 2010)), we consider it to be a "correct" detection, or a true positive.

Being able to measure correct detections for a given class (*e.g.*, car), enables us to measure the precision and recall for the class, which are defined as follows.

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

We illustrate the metrics of precision and recall in Figure 2.4. Precision measures the proportion of correctly detected objects over all predictions, while recall measures the proportion of correctly detected objects of all test objects. A good object detector should be able to detect most of the objects in the scene (high recall/completeness) and to do so accurately (high precision/correctness). Please note, there are typical trade-offs to be made, *e.g.*, in order to improve precision, usually the recall suffers and vice versa.

2.2.2.2 Precision-Recall Curve and Mean Average Precision

A precision-recall (PR) curve is used to evaluate these trade-offs between precision and recall or false positives and false negatives. Figure 2.5 plots the PR curves of two algorithms with recall on the x axis and precision on the y axis. In general, PR curves with higher precision represent more accurate algorithms, but as illustrated in this example, the two curves cannot be straightforwardly compared. Therefore, a numerical

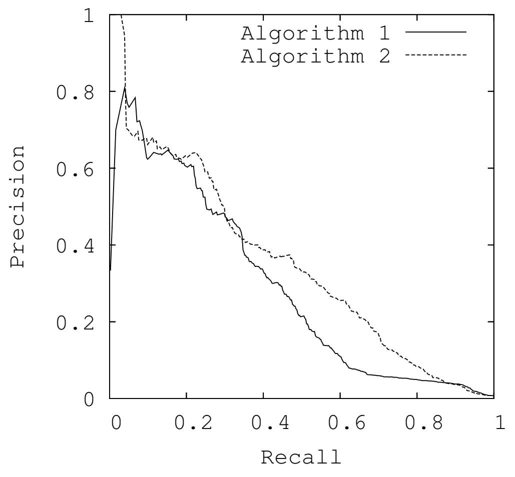


Figure 2.5: Example Precision-Recall curves (Davis and Goadrich, 2006).

value that summarizes the precision-recall curve was proposed, called *Average Precision* (AP). Specifically, it is calculated by taking the average precision of 11 equally spaced recall levels $\{0, 0.1, \ldots, 1\}$ (Everingham *et al.*, 2010). Using $p(r)$ to denote the precision with respect to the recall r,

$$AP = \frac{1}{11} \sum_{r \in 0, 0.1, \ldots, 1} p_{interp}(r)$$

$$\text{with } p_{interp}(r) = \max_{\tilde{r} > r} p(\tilde{r}).$$

An algorithm with high AP has relatively high precision at all levels of recall, and thus can perform more accurate detection. AP is a per-object-class value, and mAP is the mean AP over all classes (*e.g.*, car, people, dog, *etc.*).

2.2.3 Frameworks and Libraries

While accuracy is determined by the model design and training process, inference speed is determined by the framework and libraries, which are needed to provide real-time performance for safety-critical autonomous-driving systems. We review the commonly used inference frameworks and libraries: OpenCV, Caffe, TensorFlow, and PyTorch. Given our focus on their usage in onboard systems, we describe their design as inference engines; their design for training is beyond the scope of our discussion.

OpenCV. *OpenCV* is an open-source computer-vision library launched in 1999 (Kaehler and Bradski, 2016) that targeted real-fast applications with a design for computational efficiency. In addition to its initial support for classic computer-vision pipelines, it evolved to include an inference engine for neural networks. Recently, a new *OpenCV Graph API* (G-API) has been released for dataflow-graph-based programming, which models applications as graphs of image operations and executes the graph (OpenCV, 2020). It manages the data dependencies and the data flowing between operations. The execution of a graph is triggered by a single call in contrast to multiple calls of image functions with a conventional programming paradigm. This G-API includes basic functions such as graph construction, validation, and execution. The benefits of graph-based execution are yet to be fully explored, as the current implementation only supports pipelined execution between different types of computing devices.

Caffe. Caffe is the first widely used open-source deep learning framework. It was developed and is maintained by the UC Berkeley team (Jia *et al.*, 2014). Its clean separation between model representation and program implementation enables easy deployment between platforms and convenient model modifications for research experiments. Such separation is deemed to be *declarative programming*, which requires users to construct and express the structure of a program using pre-defined components—*what* is to be computed. For example, the neural network model is specified in configuration files in Caffe. In addition, Caffe's modular design eases the extension of network layers, loss functions, and data formats.

Apache MXNet. *Apache MXNet* (or simply MXNet) is maintained as an Apache incubation project (Chen *et al.*, 2015; The Apache Software Foundation (ASF), 2020). It combines declarative symbolic expressions and imperative tensor computations to maximize computation and memory efficiency and flexibility. In contrast to declarative programming, *imperative programming* describes the control flow—*how* something is to be computed. MXNet uses declarative programming for specifying the neural network model to apply graph optimization such as graph pruning (dead code elimination), kernel fusion, and memory-space reuse, but for tensor computation, it allows imperative programming to leverage the convenience of using the host language and to debug the program interactively.

TensorFlow. *TensorFlow* is an open-source machine-learning system developed by Google (Abadi *et al.*, 2016). It adopts declarative programming, as does MXNet, using a dataflow-graph-based programming model to scale on distributed clusters and operate in heterogeneous systems. With this uniform dataflow-graph-based programming abstraction, users define the program as a symbolic dataflow graph in the first phase. In the second phase, system-level optimization is applied before the program is executed. Such optimization can be informed by the structure of the graph, and system information such as the specifications of available computational devices. TensorFlow applies standard optimization techniques such as graph pruning, common subexpression elimination, constant folding, kernel fusion, *etc*. In addition to the standard kernel functions provided, such as matrix multiplication, users can customize their own. To ease the debugging and prototyping process, TensorFlow later added *Eager Execution*, which enables imperative programming to evaluate operations immediately without declaring a whole computational graph (Agrawal *et al.*, 2019).

PyTorch. In contrast, *PyTorch* (Paszke *et al.*, 2019) only applies imperative programming to provide users full control and better usability, which is promoted by making careful trade-offs between usability and performance. Its developers prioritized researchers' needs for a familiar programming environment in the Python ecosystem, simple and consistent interfaces, and convenient debugging and plotting tools. Paszke *et al.* also argued that for the fast-paced field of artificial intelligence, a simple implementation design that is more adaptive to changes is better. These principles make PyTorch the best for training, and for deployment on systems with abundant resources. In their estimation, the performance was within 17% of that of the fastest framework. Therefore, PyTorch is not ideal for safety-critical real-time embedded systems, where any waste of system resources should be avoided.

Before summarizing the challenges of implementing a real-time certification-amenable framework, we review a newly developed computer-vision standard, OpenVX.

2.2.4 OpenVX Standard

OpenVX is an open standard C API published by the Khronos Group for real-time and embedded computer-vision applications (Khronos Group, 2019b). By specifying a set of computer-vision function interfaces, which a conformant implementation should support, OpenVX aims for cross-platform acceleration for computer-vision applications. Although the interfaces are defined in C, the underlying backend can be implemented in any language, such as OpenCL or CUDA.

OpenVX uses dataflow-graph-based programming, which is widely used in the other computer-vision frameworks mentioned above (OpenCV, MXNet, and TensorFlow). In an OpenVX graph, each node represents an image operation, for which the formats of the inputs and outputs are specified. The nodes of operations are explicitly constructed into graphs. Edges are implicitly defined by sharing data objects between nodes. Similar to TensorFlow, users are allowed to customize node functions in addition to the ones the OpenVX implementation provides, as we see in the following OpenVX graph example.

Function name	Description
vxColorConverNode	Converts color spaces of the image (into gray in this case).
myComputeScaleLevelsNode	Computes scale levels and resizes the image. (User)
vxHOGCellsNode	Computes the gradient orientation histograms of the cells (Section 2.2.1.1).
vxHOGFeatureNode	Normalizes gradient magnitude of the cells and produces HOG feature vectors for sliding windows.
myClassifyPedestrianNode	Classifies if sliding windows contain pedestrian. (User)
myCollectDetectionLocationsNode	Collects locations (bounding boxes) where pedestrian is detected. (User)

Table 2.1: Description of functions used in HOG pedestrian detection example. Customized node functions are prefixed with `my` and indicated with *(User)*.

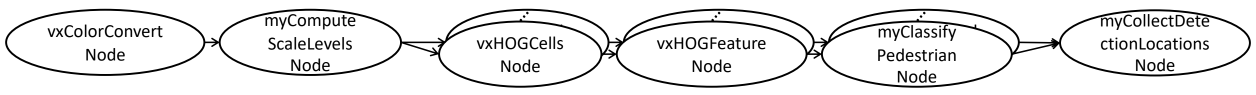

vxColorConvert Node
myCompute ScaleLevels Node
vxHOGCells Node
vxHOGFeature Node
myClassify Pedestrian Node
myCollectDetectionLocations Node

Figure 2.6: OpenVX graph example: pedestrian detection using HOG.

We illustrate an OpenVX graph for a HOG-based pedestrian-detection application in Figure 2.6, with the description of nodes in Table 2.1. As we reviewed above, HOG processes multiple scales of the image to handle objects of different sizes, so the graph branches out after `myComputeScaleLevelsNode`, with each branch processing a scale. Each edge has a data object, which is omitted in our illustration for simplicity.

Graph optimization is encouraged, *e.g.*, aggregate function replacement (kernel fusion), inter-process communication (IPC) aggregation, tiled processing, pipelining, *etc*. (Rainey *et al.*, 2014). Exercising these optimization techniques and implementing efficient kernels require an understanding of the deployment targets in regard to the hardware architecture and the processing and memory efficiency. Therefore, it encourages hardware vendors, the expected adopters of OpenVX, to provide the best-optimized implementation for their platforms, such as CPUs, GPUs, and FPGAs. Several major vendors (AMD, Intel, NVIDIA, *etc*.) have adopted OpenVX; however, many have discontinued their conformance testing for new standard versions. Although the wide adoption of OpenVX has yet to happen, it is the state-of-the-art standard where real time meets computer vision.

For application in a real-time embedded system, the OpenVX standard and aforementioned computer-vision application present challenges for deriving real-time guarantees, which we summarize next.

2.2.5 Challenges of Real-Time Certification Amenable Framework

Certifying that timing requirements are met in autonomous-driving systems will be important as we progress to seeing mass-market vehicles with greater autonomy, requiring real-time computer-vision frameworks that are amenable to certification. We next discuss four major challenges of achieving such frameworks.

Lack of real-time concepts. The major research challenge facing all computer-vision frameworks is that they lack control over timing and scheduling. The lack of timing concepts, such as deadlines and periods, makes it impossible to express any timing requirements, and no tools for measuring worst-case execution time are available. Moreover, the priorities of an application cannot be specified, leaving ambiguity for scheduling ordering and analysis. A full-fledged framework with these real-time concepts is needed.

Coarse-grained scheduling. Frameworks such as OpenVX and Caffe execute neural networks in coarse-grained *schedulable entities*. For example, network graphs are executed end-to-end, and details about inter- or intra-graph parallelization or pipelining are unspecified. A real-time graph scheduler on a heterogeneous platform requires a finer-grained view, which raises many questions. For example, what should the schedulable entities be? Do we schedule individual nodes, or even lower-level components? How do we handle multiple graphs that execute in parallel and share accelerators such as GPUs? To definitively answer these questions, we need to implement and compare multiple scheduling policies. Although we noticed in TensorFlow and MXNet that more parallelism is explored at a fine-grained level, the scheduling of applications running atop these frameworks is handled by best-effort schedulers in the operating system, rather than real-time schedulers.

Lack of system-level awareness in applications. The performance evaluation of these frameworks is conducted by running applications in isolation. However, this evaluation setup misrepresents real systems, resulting in misleading conclusions. In a deployment environment, many applications coexist and share resources—memory capacity, input-output (I/O) devices, and processors. However, none of these frameworks provides an awareness of other applications or a communication mechanism between them, leaving critical system-level optimization elusive if not inaccessible. To achieve system-level optimization, a framework may extend its control of applications to a middleware-level manager, or expose their timing and scheduling information to the OS.

Gap between computer-vision and real-time researchers. Autonomous-driving development hits a stone wall when the notion of "real time" is different for computer-vision and real-time researchers. Nonetheless, instead of burdening computer-vision researchers with acquiring expertise on real-time systems, we need to bridge this gap through a framework that is amenable to real-time certification, and that maintains usability for computer-vision developers. This requires declarative programming interfaces that allow computer-vision application developers to define a program without worrying about running the workload with real-time guarantees. It also requires configuration and evaluation tools that allow system engineers to protect the computer-vision system from interference and obtain real-time analysis results, such as response-time bounds, without worrying about conducting theoretical analysis.

These challenges are further complicated by the underlying hardware accelerators we review next.

2.3 GPUs and Accelerators

Autonomous driving has become more promising with advances in computer vision, which dramatically took off as deep-learning methods became more efficient with the computational power that accelerators like GPUs provide. Our research focuses on GPUs for reasons that we explain shortly. Nonetheless, we deem our research methodology to be applicable to other accelerators as well. We will briefly describe these accelerators and compare them. Then we focus on NVIDIA GPUs and provide a detailed review of their programming model, software architecture, and hardware architecture. After presenting these GPU fundamentals, we review prior work on GPU acceleration and scheduling. Lastly, we discuss the challenges of achieving real-time processing on GPUs.

2.3.1 Accelerators

Cars are manufactured by integrating components (including computing hardware and software) from a supply chain of various vendors ("Tier-1" and "Tier-2" suppliers). To meet the computational demands of computer-vision applications, supplier offerings are usually based on various application-specific integrated circuits (ASICs), such as field-programmable gate arrays (FPGAs), or on digital signal processors (DSPs) or GPUs. FPGAs and DSPs have advantages over GPUs in power and density, but require specialized knowledge to use the hardware and its tool-chain environment. Moreover, integrating computer-vision applications supplied by different sources using proprietary black-box solutions built on FPGAs or DSPs can increase product cost. Because of these complicating factors, we consider GPUs to be the primary solution for computer-vision applications. GPUs can be used with familiar development tools and a large number of open-source software choices, TensorFlow (Abadi *et al.*, 2016) being a notable example.

We describe these accelerators in detail and compare them in terms of processing speed, power consumption, development toolchain and libraries, and development time.

GPUs. A GPU is no longer just a graphics engine as it was originally intended to be, but also a highly parallel programmable processor for general-purpose (GP) computation—a GPGPU. Among the GPUs produced by major vendors—NVIDIA, AMD, and Intel—NVIDIA dominates the market, and its GPUs are widely used in research. NVIDIA GPU families range from server and desktop products to embedded-class ones. We focus on the embedded platforms, as they fit autonomous vehicles the best, and we survey the speed performance of NVIDIA embedded GPUs first.

The performance of "integrated" and "discrete" NVIDIA embedded GPUs is significantly different. *Integrated GPUs* (iGPUs) and CPUs are integrated at the silicon level on the same chip and share memory, whereas *discrete GPUs* (dGPUs) and CPUs are connected through a *peripheral component interconnect* (PCI) bus and have separate memory. In the latest NVIDIA Drive AGX series, which has been advertised for production-level autonomous vehicles, the Xavier platform includes NVIDIA Volta-architecture iGPUs, which provide a processing speed of 20 tera one-byte-integer operations per second (TOPS (INT8)) or 1.3 tera single-precision-floating-point operations per second (TFLOPS (FP32)). In comparison, the Pegasus platform includes additional NVIDIA Turing-architecture dGPUs, which provide 130 TOPS (INT8) or 8.1 TFLOPS (FP32)) (NVIDIA, 2020b). The performance difference between the two platforms is more than six-fold; however, the higher performance of dGPUs comes with the price of power consumption.

Unfortunately, the official power consumption specifications of dGPUs and iGPUs in the NVIDIA Drive AGX series are unavailable, although the total power consumptions of the Xavier and Pegasus platforms have been published (Mujtaba, 2018). Xavier is equipped with eight-core "Carmel" CPUs (of ARM v8 instruction set architecture (ISA)) and other accelerators for deep learning, video, and image processing, in addition to one 512-core Volta iGPU. In total, Xavier consumes 30 W of power. Pegasus consists of two Xavier boards and two 3072-core Turing dGPUs. In total, Pegasus consumes 500 W (Mujtaba, 2018). We can estimate the power consumption of one dGPU to be 220 W, which is more than seven-fold that of one iGPU (less than 30

W). These numbers are imprecise, and the actual power consumption varies with the running workload, but they do give a sense of the power cost for higher performance.

There are various tools and libraries for NVIDIA GPUs. All computer-vision frameworks discussed in Section 2.2.3 support NVIDIA GPUs, and these accelerators are the only ones supported by some frameworks (*e.g.*, Caffe). Moreover, NVIDIA provides a collection of acceleration libraries: math libraries such as *cuBLAS* and *cuFFT*, deep-learning libraries such as *cuDNN* and *TensorRT*, and image and video processing libraries like *NVIDIA Performance Primitives* (NPP). All these frameworks and libraries were built with CUDA, which includes a full-fledged development toolchain. These tools and libraries ease programming effort and shorten development time. The advantage of abundant tools and libraries alone could justify GPUs to be chosen over the other accelerators we discuss next.

FPGAs. *Field-programmable gate arrays* (FPGAs) are configurable and reprogrammable after manufacturing, as the name suggests. Optimizing an algorithm-specific configuration requires an understanding of hardware design. The configurations of FPGAs are described in *hardware description languages* (HDLs), which are complicated and different from high-level languages like C, hence increasing development time. Moreover, development time for FPGAs is longer than for GPUs owing to the poor availability of tools and libraries. There are limited ones provided by major FPGA vendors such as Xilinx, *e.g.*, Vivado, a design suite for configuring Xilinx FPGAs. However, they are neither free nor open-source.

A configurable FPGA's power consumption largely depends on the workload it is designed to execute. An application-specific comparison shows that FPGAs are relatively more power efficient than other accelerators (Nurvitadhi *et al.*, 2016).

DSPs. *Digital signal processors* (DSPs) are specialized and optimized for processing signals captured by devices such as cameras and radars, and are considered a low-cost, low-latency, and power-efficient solution for signal-processing algorithms. These algorithms can be executed on CPUs or other accelerators as well, but with lower efficiency. As with programs running on CPUs, DSP programs use a high-level programming language such as C/C++ or assembly language. Using a high-level language shortens development time; moreover, free libraries are available to ease development effort. For instance, a DSP Library (DSPLIB) includes processing algorithms such as convolution and fast Fourier transform (FFT) (Texas Instrument, 2018). However, similar to CPUs, DSPs are more suitable for serial tasks than for massively parallel ones.

	GPUs	FPGAs	DSPs
Performance	++	++	+
Tools and libraries	++	−	+
Development time	++	−	+
Power efficiency	+	++	++

Table 2.2: Comparison between accelerators. The more "+, the better; the more "−, the worse.

We summarize the comparison between GPUs, FPGAs, and DSPs in Table 2.2. Note that no up-to-date comparison among state-of-the-art accelerators is available in the literature; several dated comparisons are summarized in (HajiRassouliha *et al.*, 2018). Nonetheless, this rough comparison should be sufficient to justify the wide usage of GPUs by autonomous-driving companies and our focus on GPUs in this dissertation.

2.3.2 GPU Hardware Architecture

The work presented in this dissertation refers to the Kepler, Maxwell, Pascal, and Volta GPU architectures by NVIDIA. NVIDIA introduced these four different generations of GPU architectures, in that order, within a time span of approximately five years (2012–2017)—a pace of change more rapid than normally seen in CPU generations. Fortunately, across the NVIDIA GPU generations, the core architecture designs remained the same. We first describe the GPU hardware architecture elements that are common across generations. Later, we present the concrete architectures of the CUDA-enabled devices we use in detail.

Figure 2.7 shows the NVIDIA GPU architecture, while omitting subtle changes across the aforementioned generations. It is comprised of *copy engines* (CEs) for transferring data between GPUs and the host system, and an *execution engine*[6] (EE) for computing results. Copy engines share the system memory bus with the GPU global memory, which consists of several banks of dynamic random-access memory (DRAM). We describe the components of the EE and their interactions in the following.

The GPU workload (the GPU programs to be executed) is scheduled in a hierarchical manner. As shown at the top of Figure 2.7, a GigaThread engine dispatches the workload to the processing units, called *streaming multiprocessors* (SMs). The number of SMs varies between generations, with the newer architectures having a greater number, as mentioned in the Section 1.2. The hardware is designed to contain more transistors, allowing more SMs, which enables greater parallelism. Each SM is further organized into multiple partitions called *processing blocks*, the number of which varies across generations. As we will discuss in the CUDA

[6]It is also called the computing engine, but we call it the execution engine to avoid confusion with the copy engine when abbreviated.

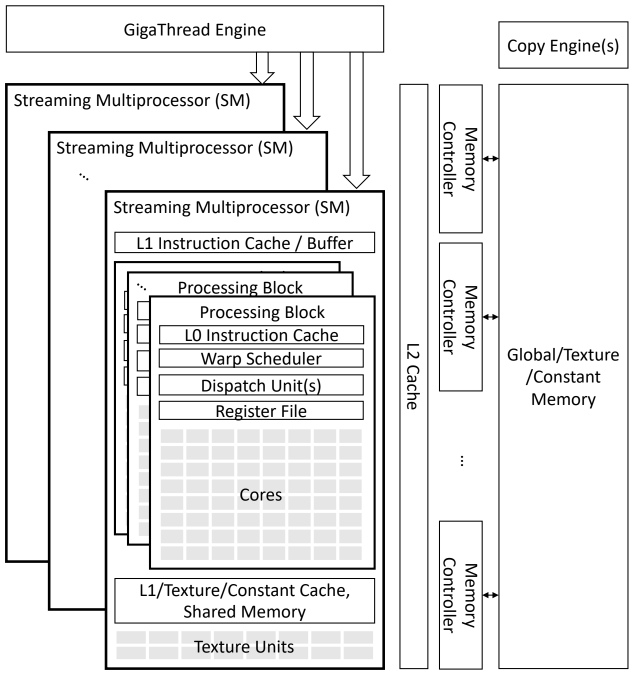


Figure 2.7: NVIDIA GPU architecture

programming model section below (Section 2.3.3), the GPU program is executed by threads, which are grouped into thread arrays called *warps*. A warp is the smallest unit that is scheduled for execution in the processing block by the *warp scheduler*. In each cycle, the warp scheduler selects a warp to run, for which each *dispatch unit* dispatches the next instruction, and these are finally executed by the various cores.

There are different types of cores: execution units for single-precision floating-point operations (FP32), double-precision float-point operations (FP64), integer operations (INT32), and, in the latest Volta architecture, new Tensor cores for deep-learning operations (NVIDIA, 2017a). The FP32 cores (the most frequently used ones) are also referred to as *CUDA cores*. The number of CUDA cores per SM has decreased with the newer NVIDIA GPU generations, but the level of resources has been retained, thus increasing the resource quota for each core and hence each thread. There are also texture units that are dedicated to image sampling and filtering.

These cores access data from the hierarchical memory. The memory that is nearest and fastest is the register file. The register file is reserved when the GigaThread engine dispatches a workload to the SMs. Therefore, the thread register usage limits the number of running threads per SM. In each processing block, an L0 instruction cache is introduced with the Volta architecture, and it provides higher efficiency than the instruction buffer that existed in prior architectures. The processing blocks in an SM share the L1 instruction

Architecture Generations	Kepler	Maxwell	Pascal	Volta
Processing Blocks / SM	1	4	2	4
Warp Schedulers / Processing Block	4	1	2	2
Dispatching Units / Processing Block	8	2	2	1
FP32 Cores / SM	192	128	64 or 128	64
FP64 Cores / SM	64	4	32	32
Tensor Cores / SM	N/A	N/A	N/A	8
Shared Memory Size / SM	Configurable options: 16, 32, or 48 KB	96 KB	64 KB	Configurable up to 96 KB

Table 2.3: Comparison between NVIDIA GPU architecture generations

cache, L1 data cache, texture cache, constant cache, and shared memory. Most importantly, the shared memory allows data sharing and synchronization between threads running in the SM within a block. Because the shared memory also needs to be reserved for access, its usage of a GPU program is another limiting factor for the number of active threads in the SM. All SMs share an L2 cache and have access to the global memory and the read-only texture and constant memory.

We noted changes in the number of several components. A summary of these changes is provided in Table 2.3. For each generation, products can be configured with different numbers of SMs, and different L2 cache sizes and global memory sizes. Therefore, these configurations are omitted here.

2.3.2.1 CUDA-Enabled Devices

We consider both dGPUs and iGPUs. An iGPU, such as the NVIDIA Jetson TX2 shown on the left in Figure 2.8, is part of a system-on-chip (SoC) implementation that is combined with conventional multicore CPUs. The SoC is packaged along with DRAM and external connectors as a small (approximately 7 inches square) single-board computer. The iGPU shares hardware resources, such as the memory controller and DRAM, with the CPU cores. The TX2 runs the Linux OS, with additional support from closed-source binary drivers provided by NVIDIA. The TX2's low size, weight, and power (SWaP) requirements and low price make it a good exemplar of a GPU-enabled platform designed for embedding in autonomous systems.

Figure 2.8 (left) shows the high-level architecture of the TX2. The TX2 contains a six-core heterogeneous ARMv8 CPUs, 8 GB of DRAM, and an integrated Pascal GPU. The TX2's GPU consists of two SMs, each comprised of 128 GPU cores. The SMs together can be logically viewed as an EE. Additionally, there is

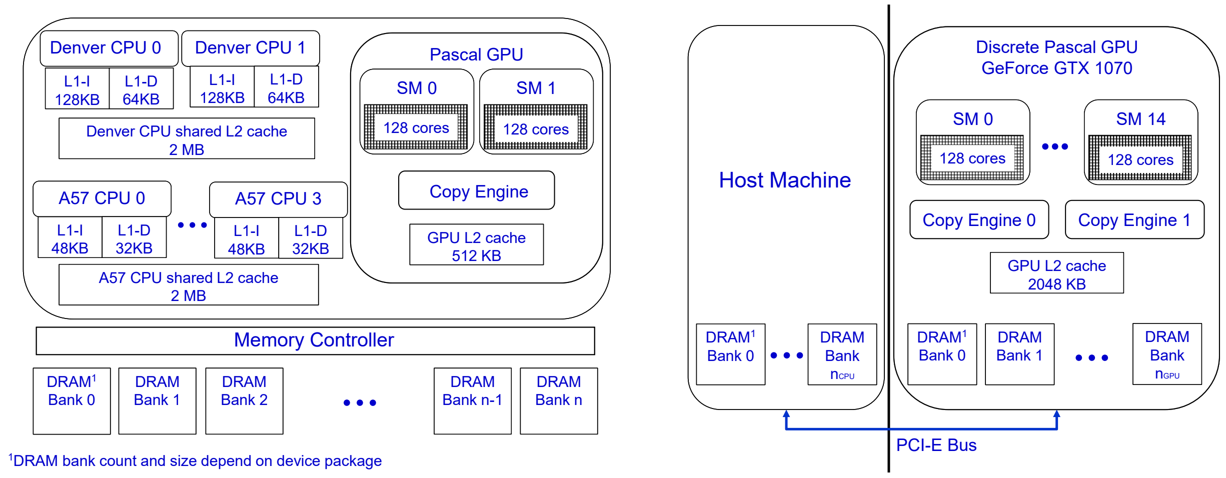


Figure 2.8: Jetson TX2 Architecture (left) and GeForce GTX 1070 Architecture (right)

a hardware CE that can copy data between memory regions allocated for CPU use and regions allocated for GPU use. The iGPU has fewer GPU cores than is typically found in high-end GPUs used for graphics, gaming, and high-performance computing (HPC) applications. We are interested in exploiting any potential for sharing the TX2's GPU for multiple tasks so that its computing capacity is not unnecessarily wasted.

Shown on the right in Figure 2.8 is the architecture of the GTX 1070, an example of a dGPU. Discrete GPUs consist only of the SMs and local device memory, typically packaged on an adapter card for mounting in a PCIe expansion slot on a computer motherboard. Like all dGPUs, the GTX 1070 does not share memory with the host CPU, instead using the PCIe bus to copy data to and from the host memory. This GPU features many more SMs than the TX2, increasing the potential benefit attainable if it is shared among multiple tasks. It also has two CEs, and a larger cache.

2.3.3 GPU Programming Model

Today's GPUs evolved from a fixed-function special-purpose hardware pipeline for graphics rendering. Initially, the hardware pipeline was configurable but not programmable (Owens *et al.*, 2008). Each stage in the pipeline then became more flexible for running user programs when general-purpose programs were adapted and mapped to the graphics pipelines. However, such hardware pipelines are limited by their performance bottleneck—the slowest stage of processing. The solution for mitigating this issue was unified processing units, as in the modern GPU hardware already discussed in Section 2.3.2, so that the mapping process for graphics APIs was no longer necessary. This change eventually made possible the GPGPU programming model we describe in this section.

Algorithm 1 Vector Addition Pseudocode.

```
 1: kernel VECADD(A ptr to int, B: ptr to int, C: ptr to int)
    ▷ Calculate index based on built-in thread and block information
 2:     i := blockDim.x * blockIdx.x + threadIdx.x
 3:     C[i] := A[i] + B[i]
 4: end kernel

 5: procedure MAIN
    ▷ (i) Allocate GPU memory for arrays A, B, and C
 6:     cudaMalloc(d_A)
 7:     ...
    ▷ (ii) Copy data from CPU to GPU memory for arrays A and B
 8:     cudaMemcpy(d_A, h_A)
 9:     ...
    ▷ (iii) Launch the kernel
10:     vecAdd<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C)
    ▷ (iv) Copy results from GPU to CPU array C
11:     cudaMemcpy(h_C, d_C)
    ▷ (v) Free GPU memory for arrays A, B, and C
12:     cudaFree(d_A)
13:     ...
14: end procedure
```

The GPGPU programming model follows a single-instruction multi-thread (SIMT) programming paradigm. SIMT on GPUs is equivalent to a single-instruction multi-data (SIMD) programming paradigm, as each thread usually processes a different data element. Following the SIMT programming paradigm, GPU threads process data elements on multiple processing units in the GPU, providing data parallelism in GPU computing space. To enable SIMT, NVIDIA provides CUDA programming interfaces, which we illustrate with a VECADD program as follows.

With GPUs as co-processors to CPUs, a *CUDA program* runs as a task (process or thread) on a CPU and relies on a GPU for some part of its computational requirements.[7] The general structure of a CUDA program when it needs to interact with a GPU is as follows: **(i)** allocate memory for the GPU to use; **(ii)** copy input data from CPU memory to GPU memory; **(iii)** launch execution of a GPU program called a *kernel* to process the data; **(iv)** copy the results from GPU memory back to CPU memory; **(v)** free any unneeded memory.

CUDA extends the C/C++ language to allow the definition of GPU functions as *kernels*. CUDA kernels are written from the perspective of a single *GPU thread*. Consider the CUDA program expressed in pseudocode in Algorithm 1 as an example: it uses the kernel VECADD, which defines that each GPU thread adds a single pair of elements and stores the sum in a corresponding location in an output array. Each

[7]Note that both CPU and GPU computations are specified in the same CUDA program.

GPU thread is assigned an index, which can be retrieved via special global system-defined variables, as demonstrated in Line 2. In this example, thread indices are used to locate the array elements.

A kernel runs on a GPU through a set of *thread blocks* that can be executed in any order. Each thread block, or simply *block*, comprises a number of threads. As seen in Line 10 of Algorithm 1, the number of blocks and threads per block are programmer-specified and can be set at runtime when a kernel is launched. The CUDA runtime uses these values to launch and organize GPU threads (or simply *threads*[8]).

These threads are scheduled on GPUs with block granularity: resources are reserved and released per block, rather than per thread or kernel. In other words, *blocks are the schedulable entities on NVIDIA GPUs.* Threads in a block are launched as a whole when an SM becomes ready with enough resources for the entire block. All threads in a block are executed on the same SM non-preemptively until their completion. A block is assigned to the first SM that is ready. However, when multiple SMs are available, it is undocumented which SM the block will be assigned to—the block-to-SM assignment rules have been partially dissected in a recent work by (Saudo *et al.*, 2020). Multiple blocks from the same kernel can execute in parallel. A kernel starts when at least one thread of one block starts; a kernel completes when all threads in all blocks have exited.

However, not all threads in a block execute in parallel, nor is this an efficient way to explore parallelism. Threads in a block are further grouped into *warps*. It is within a warp that the same instruction is executed by all threads; different warps from the same thread block can execute different instructions. In this way, stalls due to memory accesses and waiting for special units can be hidden by scheduling another ready warp. Moreover, when the program diverges into different branches within a warp, threads will run in lock-step. Threads of only one branch are running at a time, and the remaining threads are left inactive, wasting resources. Such penalties depend on the size of the warp—a smaller warp causes less waste, but increases the hardware complexity for warp scheduling and management. Throughout all NVIDIA GPU architectures, the warp size has remainsed 32.

Warp scheduling determines the low-level execution, but unfortunately, its scheduling policy is implemented in the hardware and unknown, and subject to change between architecture generations. For example, prior to the Volta architecture, the execution of branches was serialized, whereas since the Volta architecture, the execution of branches can be interleaved. Fortunately, at block granularity there exist higher-level scheduling and resource management, for which we conduct our real-time analysis.

[8]We henceforth use the term *thread* to mean a hardware thread on the GPU, and refer to threads that execute on CPUs explicitly as CPU threads.

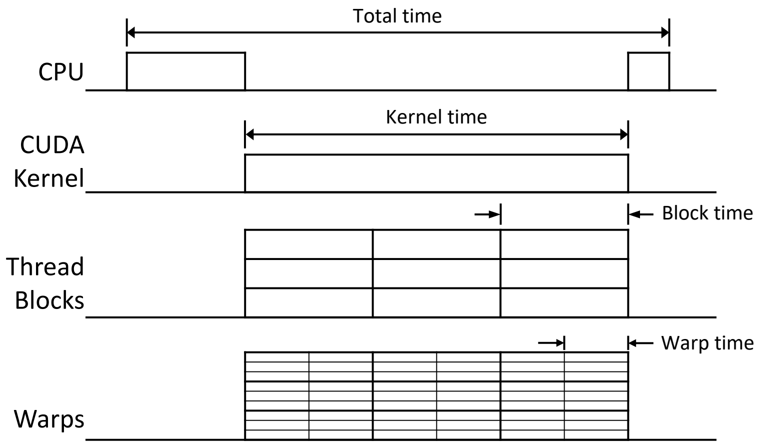


Figure 2.9: Disgram illustrating the relation between CUDA programs, kernels, thread blocks, and warps.

We illustrate the execution of these entities described so far in Figure 2.9. In this figure, we show the execution timeline of one kernel launch. As mentioned, parallelism is explored between warps and between blocks. The figure distinguishes time at different granularities: *warp time*, the time taken to execute one warp, *block time*, the time taken to execute a single block (from the start time of its first warp to the end time of its last warp), *kernel time*, the time taken to execute a kernel (from the start time of its first block to the end time to its last block), and *total time*, the time taken to execute the entire CUDA program (both CPU and GPU portions).

We refer to kernels and memory-copy operations collectively as *GPU operations*, which are submitted to a GPU in *CUDA streams*. Operations within a stream are executed in FIFO order. By default, the *NULL stream* is used, but users can submit operations to multiple user-defined streams.[9] Kernels from different streams can run concurrently if sufficient internal resources are available. Concurrent kernel execution (CKE) was first introduced in the Fermi architecture and then enhanced with the Hyper-Q features introduced in the Kepler architecture (NVIDIA, 2009, 2012). Copy operations are handled by the GPU's CE and can be in parallel with kernel executions on the EE.

True parallelism is limited to kernels in the same *CUDA context*, which is usually one per OS process. It has been shown that kernels from different CUDA contexts are not truly concurrent but execute in a multi-programmed manner (Otterness *et al.*, 2017b), *i.e.*, blocks from different kernels never overlap. Fortunately, NVIDIA provides the Multi-Process Service (MPS), which funnels kernels from different OS processes into a shared CUDA context so they can execute in parallel (NVIDIA, 2019e). MPS was further improved

[9]CUDA documentation guarantees that operations within a stream are executed in FIFO order, but does not describe how operations from different streams are ordered.

to include quality-of-service features to manage execution resource provisioning on the Volta architecture. Moreover, since the introduction of Volta, processes using MPS can retain individual CUDA contexts.

As mentioned above, a CUDA program makes CUDA API calls that can be synchronous or asynchronous; for many calls, a variant of both is available. For example, `cudaMemcpy` and `cudaMemcpyAsync` both copy data between regions of CPU memory and GPU memory, or between two regions of GPU memory, but when memory is pinned (*i.e.*, page-locked), `cudaMemcpyAsync` can return control to the calling CPU task before the copy is completed, whereas `cudaMemcpy` blocks the CPU task until the memory copy completes.

Kernel launches are always supposed to be asynchronous. The CUDA documentation[10] (NVIDIA, 2019c), however, uses a narrow definition of "asynchronous" that can be misleading. According to the documentation, "concurrent host execution is facilitated through asynchronous library functions that return control to the host thread before the device completes the requested task." Notably, this definition does not imply that asynchronous API calls are *nonblocking* to the CPU. As noted in Section 3.3, we have found situations in which kernel launches still cause CPU blocking even if the API call returns before the requested kernel completes.

In summary, we have reviewed different accelerators and NVIDIA GPU hardware architecture and programming models thus far, which provide GPU fundamentals for the review of prior work on non-real-time GPUs.

2.3.4 Prior Work on Non-Real-Time GPUs

As GPUs support general-purpose computation beyond embedded and real-time systems, we dedicate this section to reviewing prior works on non-real-time GPUs. We first review reverse-engineering works that reveal details beyond what is included in the official documentation. This revealed information is important for resource management, which we review next. We then review GPU sharing techniques and GPU virtualization. Lastly, we review non-real-time GPU scheduling frameworks.

As we will see in our real-time GPU review, many of the techniques for non-real-time systems inspired a variant usage in real-time ones.

[10]Section 3.2.5.1 of the Programming Guide for CUDA version 10.2.89.

2.3.4.1 GPU Reverse Engineering

GPU program optimization, modeling and analysis, and resource management require a deep understanding of GPU runtime and hardware details: GPU microarchitecture, memory hierarchy, scheduling policies, *etc*. However, owing to the limited documentation and closed-source software hardware stack of NVIDIA GPUs, these details are often either unavailable or confusing.

This has led to a series of microbenchmark-based reverse-engineering studies. An early GPU architecture, Tesla, was benchmarked by Volkov and Demmel (2008) to achieve fast fine-tuned linear algebra using measurement of the memory-system structure and performance, kernel launch overheads, and processing-unit latency. Many related works have continued such reverse-engineering effort for every NVIDIA GPU architecture generation (Wong *et al.*, 2010; Meltzer *et al.*, 2013; Mei and Chu, 2016; Jia *et al.*, 2018; Jain *et al.*, 2019). They noted findings that were inconsistent with the official documents, *e.g.*, existence of translation lookaside buffer (TLB) (Volkov and Demmel, 2008) and specifics regarding texture and constant caches (Wong *et al.*, 2010). Such inconsistency still persists in other respects, such as asynchronous execution, which we will discuss. These inconsistencies are problematic as they could result in a burden of debugging and even damage in safety-critical settings.

These reverse-engineering efforts focused on discovering the structures of memory hierarchies and measuring the performance of low-level hardware units. Their results were shown to be useful for application performance optimization (Volkov and Demmel, 2008) and interference reduction via hardware isolation (Jain *et al.*, 2019).

To enable OS-level management of NVIDIA GPUs, the interactions between black-box library, driver, and hardware must be understood. By tracing the events between these entities, Menychtas *et al.* (2013) demonstrated that an interaction state machine can be automatically inferred. This state machine was then leveraged to build an interception-based OS-level GPU scheduling management framework (Menychtas *et al.*, 2014). The open-source NVIDIA driver Nouveau also relies on tracing memory mapped I/O access to reverse engineer NVIDIA GPU drivers (X.org Foundation, 2014); however, Nouveau does not support CUDA.

These reverse-engineering efforts reveal interactions within the black-box software hardware stack, but not GPU scheduling policies, which are fundamental for real-time analysis. We will discuss our effort to understand GPU scheduling rules in this dissertation. After we published these scheduling rules, Saudo *et al.* (2020) further investigated the block-to-SM assignment policy and noted the impacts of memory usage (shared memory and registers) on block-to-SM assignments, but as they noted, the complete policy has yet to be revealed. Nonetheless, because of the principle of balancing blocks to SMs as they also noted, any benefit to our real-time analysis from considering the block-to-SM assignment would be subtle.

2.3.4.2 GPU Resource Management

We now review GPU resource-management techniques for cache, memory, and computing resources.

GPU cache management. The GPU cache is an efficient but limited resource in reducing memory access latency. However, massive threads and a bursty memory-access pattern of GPU workloads can cause cache thrashing and interference. Prior works presented techniques to mitigate cache interference through cache bypassing and cache isolation. Special cache operators for cache bypassing control are available in Parallel Thread Execution (PTX) Instruction Set Architecture (ISA) provided by NVIDIA (NVIDIA, 2020). Some prior works have experimented with different cache-bypassing policies (Li *et al.*, 2015a,b,c; Ausavarungnirun *et al.*, 2015), but they relied on profiling mechanisms they implemented on a GPU simulator, GPGPU-sim (Bakhoda *et al.*, 2009), leaving their work inapplicable on real hardware unless adopted by vendors.

Jain *et al.* (2019) reverse-engineered the hash function that maps physical memory addresses and cache sets. They first identified all pairs of addresses that are mapped into the same cache set and thus can evict each other. Then they find the only valid hash function by brute force, under the assumptions that **(i)** physical addresses are the input of such hash function, **(ii)** only bitwise operations are used in the hash function for fast hardware implementation, and **(iii)** caches apply Least Recently Used (LRU) as the eviction policy. They had to modify (the open-source portion of) NVIDIA driver to obtain physical addresses. With their modification and the reverse-engineered hash function, they implemented page coloring to partition L2 cache allocation.

GPU memory management. In addition to cache isolation, Jain *et al.* (2019) implemented DRAM partitioning. Using the same approach, they reverse-engineered the hash function that maps physical memory addresses to DRAM banks. Based on their observation, GPU DRAM and L2 cache sets can be organized into memory modules. They showed that memory interferences can be mitigated by isolating applications to different memory modules through page coloring. Other prior work experimented with different memory management on the simulator GPGPU-sim (Jog *et al.*, 2014; Mao *et al.*, 2016; Jog *et al.*, 2016; Ausavarungnirun *et al.*, 2017, 2018).

GPU computing resource management. Managing computing resources on GPUs via partitioning allows for spatial multitasking, which has been investigated on GPU simulators (Adriaens *et al.*, 2012; Ubal *et al.*, 2012; Aguilera *et al.*, 2014a,b; Ukidave *et al.*, 2014; Tanasic *et al.*, 2014).

For real hardware, Wu *et al.* (2015) presented a software approach that transforms programs to execute the workloads on the assigned SMs. A transformed program determines the work of each thread based on the SM ID instead of thread ID used in the traditional CUDA programming model. With such a SM-centric transformation, a kernel can be partitioned to a subset of SMs, allowing spatial scheduling of concurrent kernels. This approach was applied by Janzén *et al.* (2016) to improve the throughput of poorly scaling HPC tasks, and applied by Jain *et al.* (2019) along with memory reservations.

We have reviewed prior works on GPU resource management in this section. Some of these techniques are also useful for GPU sharing, which we review next.

2.3.4.3 GPU Sharing

As GPUs evolve to increase their capacity, it is difficult for applications to scale as well, or for a single application to continue fully utilizing resources on new GPUs. Moreover, any application may underutilize GPUs during its exiting stage, where in extreme cases only one SM may be used. Therefore, GPU sharing is necessary. In the previous section, we reviewed resource-management techniques, many of which impact GPU sharing efficiency. For example, partitioning computing and memory resources allows spatial multitasking sharing while reducing resource contention. In this section, we review other techniques for mitigating interference and other GPU-sharing techniques, including kernel fusion, kernel slicing, and kernel preemption.

GPU interference and concurrency. A prior body of work examined how to characterize kernels and predict the interference between their concurrent execution. Many prior works regarding interference analysis or mitigation use GPU simulators (Kerr *et al.*, 2009; Goswami *et al.*, 2010; Sethia and Mahlke, 2014; Jog *et al.*, 2015; Wang *et al.*, 2016; Kayıran *et al.*, 2013; Hu *et al.*, 2016; Xu *et al.*, 2016a; Dai *et al.*, 2018). However, these methods rely on hardware changes or metrics that are unavailable with real hardware. For example, Jog *et al.* (2015) focused on memory interference using a misses-per-kilo-instruction metric, which is unavailable with real hardware-tracing tools (NVIDIA, 2019d). Therefore, we focus on research work using real hardware.

Phull *et al.* (2012) proposed a performance degradation-analysis model based on the profiling of an application's repetitive GPU access and execution pattern—specifically, time gaps between GPU calls and average GPU execution time. They proposed two interference prediction methods: simulation-based one and timed Petri nets (TPN) (Ramchandani, 1973), which is a formal model for analyzing execution in concurrent systems. The predicted interference results are used to inform kernel-to-device assignment in a cluster environment. Because such prediction relies on a repetitive execution pattern, when it is unavailable, an online monitoring and response mechanism takes charge to detect excessive interference and reassign interfering kernels.

Before concurrent kernel execution was introduced in NVIDIA Kepler architecture, kernel fusion is a common technique that virtually enables concurrency on GPUs by merging multiple kernels into one (Wang *et al.*, 2010; Ravi *et al.*, 2011; Gregg *et al.*, 2012). Ravi *et al.* (2011) proposed grading the potential improvement of consolidating two kernels using the *affinity score*, which is determined by considering kernels' resource requirements: compute resources (number of threads) on SMs and shared-memory usage. Higher contention due to conflicting resource requirements leads to a lower affinity score, indicating fewer concurrency benefits. In addition, they proposed a couple of *molding* policies to change the resource requirements by varying the number of threads in the kernel, enabling more concurrency opportunities. However, their experiments were limited to the kernels that could accept any number of threads. Such limitations were removed in a later work that supported the transformation of any kernel to *elastic kernels* (Pai *et al.*, 2013).

Such a transformation is also applied to divide a kernel into smaller slices that comprise fewer threads and require fewer resources that can create more opportunities for concurrency. To achieve this dynamically, Zhong and He (2013) proposed the *Kenelet* runtime, which transforms intermediate-level or assembly-level code and thus does not require source-code modification. For memory-copy operations, Gdev slices memory transactions into chunks in order to overlap bidirectional memory transfers (Kato *et al.*, 2012).

Other similar work that estimates the interference or explores the benefits of concurrent kernel execution include (Ravi *et al.*, 2011; Pai *et al.*, 2013; Ukidave *et al.*, 2016; Chen *et al.*, 2017a; Carvalho *et al.*, 2017; Wen *et al.*, 2018; Carvalho *et al.*, 2020).

GPU preemption. GPU preemption helps prevent starvation and improves the fairness and responsiveness of GPU kernels. Compute preemption at the instruction level was introduced with the NVIDIA Pascal architecture (NVIDIA, 2016b). Before that, to enable GPU preemption, Tanasic *et al.* (2014) proposed SM draining. During *SM draining*, each SM is preempted by waiting for the existing thread blocks to complete. As we will reveal in Section 3.2.2, kernels from streams with higher priorities preempt ones with lower priorities in a similar way at the boundary of thread-block execution. This preemption mechanism relies on the GPU execution model: each thread block is executed independently and has its own states. Because the running thread blocks exit normally and ones that have not commenced yet have no contexts, it is unnecessary to save the context of the preempted kernel, which prevents context-switching overheads. However, the preemption latency depends on the thread blocks' remaining execution time, which can be long or forever with a persistent kernel.

To reduce preemption latency, Park *et al.* (2015) experimented on GPGPU-sim with another preemption technique, *SM flushing*, which stops thread blocks immediately if they are *idempotent* (De Kruijf and Sankaralingam, 2011). A thread block is idempotent if it produces the same results regardless of being rerun multiple times. Because not all kernels conform to this idempotent condition, SM flushing has limited applicability. In addition, although SM flushing decreases preemption latency, it forces the rerunning of a thread block, resulting in overheads that may decrease the overall throughput. To mitigate such overheads, preemption techniques—SM draining, SM flushing, or the baseline context switching—are dynamically selected considering the execution progress of the thread block (Park *et al.*, 2015).

Wang *et al.* (2016) and Lin *et al.* (2016) also experimented on GPGPU-sim to produce a finer-grained preemption mechanism.

We have discussed many GPU-sharing techniques from interference mitigation to efficient preemption. Next, we review an important GPU sharing scenario in cloud environment using GPU virtualization.

2.3.4.4 GPU Virtualization

As a power-efficient and cost-effective accelerator, GPUs are widely used in cloud systems (Microsoft Azure, 2020; Amazon AWS, 2020; Google Cloud, 2020) where resources are shared among clients, each of which is often assigned a virtual machine that provides the functionality of a physical computer. The virtualization of GPUs as a GPU-sharing technique, however, is a new and challenging area for which we review the implementation techniques: API intercepting and remoting, driver-level virtualization, and hardware-supported virtualization. We focus on the virtualization solutions for GPGPU computing and omit those for graphics.

API intercepting and remoting. User-space API intercepting and remoting is a more common GPU virtualization technique, because it does not require interaction with the closed-source driver or hardware. A common approach to API intercepting and remoting is similar to that of remote procedure calls (RPCs). It takes three steps: first, the GPU calls in the guest OS are *intercepted* by wrapping the runtime API library; second, the intercepted calls are sent to and processed *remotely* by the host OS; and third, the computation results are returned back to the guest OS. Adapting to library updates is affordable through automatically generated wrapper library. With the wrapper library dynamically linked, this also avoids any modification and recompilation to the client applications. Prior works have proposed various performance improvement methods for this approach (Gupta *et al.*, 2009, 2011; Duato *et al.*, 2010; Giunta *et al.*, 2010; Shi *et al.*, 2011).

Driver-level virtualization. This is possible with open-source GPU device drivers made by vendors like AMD (AMD, 2016) or third parties involved in reverse-engineering efforts (X.org Foundation, 2020).

Using an open-source Nouveau driver on NVIDIA GPUs, GPUvm creates *shadow copies* of GPUs to isolate GPU usage from multiple guest OSs, each of which owns *shadow page tables (SPTs)* and GPU command channels (Suzuki *et al.*, 2014). In addition, GPUvm provides fair time-sharing of GPUs using a *bandwidth-aware non-preemptive device (BAND)* scheduler, and prioritizes guest OSs considering their utilization and their periodically replenished budgets (credits). The BAND scheduler is based on the Credit scheduler in Xen (Bensaou *et al.*, 2001; Barham *et al.*, 2003) with a modification that compensates for credit (budget) accounting errors due to non-preemptive GPU execution.

Huang *et al.* (2016) implemented a virtualization solution for the *Heterogeneous System Architecture (HSA)* (Kyriazis, 2012). The goal of HSA is to improve the communication efficiency and programmability of heterogeneous platforms. It reduces memory copies by sharing memory space between the CPU and GPU, and supports user-mode queuing for dispatching kernels that would otherwise be passed through the device driver, and uses memory-based instead of interruption-based kernel completion signaling to reduce the communication latency. Therefore, virtualizing HSA-based GPUs must be memory-based as well. As page tables are shared between the CPU and GPU, the memory translation from *guest virtual address (GVA)* to *machine physical address (MPA)* can be accomplished by sharing the SPT between CPU and GPU as well. Given that HSA manages GPU commands in user-space queues, the hypervisor only needs to initialize

GVA for these queues to the GPU. This memory-based management in HSA makes virtualization simpler compared with other GPU system architectures. From the GPU's perspective, queues from guest OSs are managed and scheduled similarly to ones from processes in the host OS.

Hardware-supported virtualization. Hardware-supported virtualization for GPUs depends on the features provided by the vendors. The Intel VT-d (Abramson *et al.*, 2006) and AMD-Vi (AMD, 2016) support a single guest OS per GPU. The NVIDIA GRID (Herrera, 2014) supports sharing the GPU with multiple guest OSs by separating the physical address space of the GPU and enabling independent address translation. Such hardware isolation is important for real-time systems. However, the NVIDIA GRID is limited to graphics.

2.3.4.5 GPU Scheduling

GPU scheduling is managed in the runtime, driver, and hardware stack. We review work devoted to controlling GPU scheduling through code modification, runtime replacement, or system-level control.

GPU persistent threads. In contrast to traditional GPU programming where thread blocks are completely scheduled by the hardware, a different programming style that uses *persistent threads (PT)* enables software scheduling. It takes fewer threads that are persistent through the kernel's lifetime to fetch and process all the workload from the work queues. In PT-style programming, workloads are scheduled into the work queues bypassing hardware scheduling. Most of the advantageous features summarized by Gupta *et al.* (2012) have been natively supported in later CUDA generations, as we discuss next.

Tzeng *et al.* (2010) applied PT for dynamically generated irregular-parallel workloads, which were poorly supported in GPUs at the time. In traditional non-PT GPU programming, to handle the generated workloads, the host must copy the results and then launch new kernels accordingly. In contrast, using PT, the generated workloads can be appended to the work queues for further processing. Later, dynamically generated workloads were natively supported by a feature named *dynamic parallelism* that supports kernel launches from the GPU (NVIDIA, 2019c).

Early on, synchronization between GPU threads was only supported within each thread block. Global synchronization across thread blocks required dividing kernels at synchronization points, and was accomplished through extra host-device communication to relaunch kernels for the portion after the synchronization points, similar to the way generated workloads are handled. Using PT, Stuart and Owens (2009) handled global synchronization in their implementation for message-passing APIs on GPUs by allowing all thread blocks to be resident. Later in 2017, NVIDIA supported global synchronization with a feature named *cooperative groups* (NVIDIA, 2019c).

GPU kernel reordering. Before Hyper-Q was introduced with the Kepler architecture, kernels that were launched in multiple streams could suffer from false serialization, which happened when kernels from the same stream were submitted to the single hardware queue and thus had to be serialized. Wende *et al.* (2012) proposed a producer-consumer solution, in which a consumer thread pulls kernels from the work-item queues of multiple producer threads in round-robin fashion, effectively avoiding having kernels from the same stream in the hardware queue. Kernel ordering is also explored in (Lázaro-Muñoz *et al.*, 2017; Zhou *et al.*, 2018).

GPU as a first-class device. In current system architectures, GPUs are managed as I/O devices by drivers. However, as GPUs become capable of processing more general-purpose computation, it is expected that managing GPUs as first-class devices in the OS will benefit from system-level optimization. Meanwhile, it also enables the usage of the GPU for OS tasks, as demonstrated by the file system encryption tasks supported in PTask (Rossbach *et al.*, 2011), Gdev (Kato *et al.*, 2012), GPUstore (Sun *et al.*, 2012).

PTask manages GPUs as first-class devices with a set of OS abstractions (Rossbach *et al.*, 2011). These abstractions provide a dataflow programming model that represents applications as graphs, in which vertices represent CPU or GPU work. The work and graph are declared by application programmers and then scheduled by the OS. In this way, the OS can exploit graph scheduling optimization and eliminate unnecessary data movements with system-wide visibility. For example, PTask scheduling in the *data-aware* mode prioritizes scheduling a task to the device where its input data resides, and incorporates OS priority in the *priority* mode to improve scheduling fairness.

In a PTask graph, each vertex represents the work of one type of device. The OS tracks GPU vertices and schedules them like normal CPU processes. However, because proprietary GPU drivers do not provide kernel-mode interfaces, a GPU vertex still requires user-mode CPU portions, if not a user-mode CPU process in the PTask implementation.

This requirement is lifted with an open-source driver in another framework Gdev, which also manages GPUs as first-class devices (Kato *et al.*, 2012). In Gdev, a runtime library support is integrated in a Linux kernel module alongside a custom GPU device driver, allowing both OS and user-space programs to use the same API. Gdev manages these API calls directly, rather than using API interception (like the GPU virtualization work above) or additional programming abstraction (like PTask). By managing GPUs as first-class devices, Gdev implements GPU resource management, such as using shared memory for IPC and using data swapping to support a virtual address space that exceeds the physical size limit.

Menychtas *et al.* (2014) also argued for the OS's responsibility for GPU resource management. As mentioned above, they built an OS-level scheduling framework that intercepted hardware-software interfaces revealed in their prior reverse-engineering work (Menychtas *et al.*, 2013). CPUs and GPUs communicate to send GPU operation requests, and receive completion signals through memory-mapped I/O (MMIO). To intercept kernel launches, the device control registers are unmapped to enforce page-fault handling, where GPU scheduling management can be inserted. Meanwhile, GPU kernel completion signals are constantly polled to monitor execution status. This approach enforces scheduling policies for any GPU application, whereas API-driven approaches (*e.g.*, PTask and Gdev) require applications to use the modified or wrapper libraries.

Using this approach, Menychtas *et al.* (2014) experimented with a *disengaged timeslice* scheduler using token-based timeslice control to ensure fairness between GPU applications and a *disengaged fair queueing* scheduler that allows CKE to improve efficient GPU utilization. However, it had the drawback of being unable to differentiate between kernel and copy operations, and thus the GPU had to be managed as a whole, preventing any exploration of the parallelism between CE and EE.

2.3.5 Challenges for Real-Time GPUs

As a commonly used accelerator for computer-vision applications in safety-critical autonomous-driving systems, the GPU must provide real-time guarantees. After reviewing GPU hardware architecture, programming models, and prior work on non-real-time GPUs, we discuss the challenges of enabling real-time GPUs.

Understanding GPU scheduling. Existing reverse-engineering work focuses on GPU microarchitecture and memory hierarchies and structures, leaving the GPU scheduling rules to be revealed. A complete set of GPU scheduling policies are fundamental for real-time analysis. However, only limited information pertaining to the possible CKE between streams and the serialization within streams is included in the official documentation. The exact condition for concurrent execution and the scheduling order of kernels from multiple streams remains unknown.

Synchronization pitfalls. In reviewing prior reverse-engineering works, we noticed an inconsistency between experimental observations and the documents about the cache on NVIDIA GPUs. Unfortunately, a similar inconsistency persists in CPU-GPU synchronization issues. Asynchronous APIs can cause unexpected synchronization behavior, resulting in problematic timing issues.

Performance implication of execution models. GPU tasks execute differently in *process-based* and *thread-based* contexts. In a process-based context, each task is executed as an OS process with its own address space and memory isolation between tasks. In addition, the aforementioned synchronization pitfalls do not exist across processes. However, process-based execution does not permit concurrency on the GPU. In a thread-based context, all tasks running as threads in a shared address space can have kernels running concurrently on the GPU, but memory isolation is lost. Furthermore, using the MPS tool expands the options of the execution models, raising the question of how real-time GPU tasks should be executed (process vs thread vs MPS).

GPU task modeling and real-time analysis. Real-time analysis is conducted using task models. While there have been extensive studies about CPU task models, prior work on GPU task models is limited. We have discussed how the programming model and execution paradigm of GPUs are different from those of CPUs, and therefore a new task model for GPUs with the revealed GPU scheduling rules is required to lay the groundwork for real-time schedulability analysis. The challenge is not only to analyze GPU tasks themselves, but to analyze tasks running the CPU-GPU heterogeneous platforms. We provide the fundamentals of real-time systems and review prior work on real-time GPUs next.

2.4 Real-Time Systems

The central subject of this dissertation is real-time systems using GPUs that are to be shared by computer-vision applications in autonomous-driving systems. In this section, we provide essential background on real-time systems. We first describe a commonly used real-time task model and several scheduling algorithms. In discussing real-time correctness under these algorithms, we describe two key concepts—schedulability and feasibility—followed by some prior correctness results including some schedulability tests and response-time-bound analysis. In particular, we focus on reviewing real-time scheduling for directed acyclic graphs (DAGs), upon which we build an analysis of the heterogeneous CPU-GPU platforms we use. Then we transition from

Notation	Definition	
τ	A task system (task set)	$\tau = \{\tau_1, \tau_2, \tau_3, \ldots, \tau_n\}$
τ_i	The ith task of τ	
T_i	The period of τ_i	
C_i	The worst-case execution requirement of τ_i	
D_i	The relative deadline of τ_i	
u_i	The utilization of τ_i	$u_i = C_i/T_i$
U_{sum}	The total utilization of τ	$U_{sum} = \sum u_i$
R_i	The response-time bound of τ_i	$R_i = max_j(R_{i,j})$
$\tau_{i,j}$	The jth job of τ_i	
$r_{i,j}$	The release time of $\tau_{i,j}$	
$f_{i,j}$	The finish time of $\tau_{i,j}$	
$d_{i,j}$	The (absolute) deadline of $\tau_{i,j}$	$d_{i,j} = r_{i,j} + D_i$
$R_{i,j}$	The response time of $\tau_{i,j}$	$R_{i,j} = f_{i,j} - r_{i,j}$
$x_{i,j}$	The tardiness of $\tau_{i,j}$	$x_{i,j} = max(0, f_{i,j} - d_{i,j})$

Table 2.4: Real-time task model notation.

theory to practice, and review several real-time framework implementations that are used in this dissertation. Lastly, after reviewing all the fundamentals, we thoroughly review prior work on real-time GPUs.

2.4.1 Real-Time Task Model

Real-time autonomous-driving systems encompass many applications that are under temporal constraints due to their position on the critical path from capturing the environment to executing vehicle-control commands. Some are event-driven, such as computer-vision applications processing images that arrive continuously, and some are time-driven, such as the logging and monitoring components that check system status periodically. These real-time applications in autonomous-driving and other systems share a characteristic: they issue new units of work recurrently. Such recurrence is described formally for reasoning about these applications and validating their temporal correctness. A commonly used formalization, the one we also use, is the *sporadic task model* (Mok, 1983), which we describe next with terms and notation that has been widely adopted by real-time systems community (IEEE TCRTS, 2020), as summarized in Table 2.4.

In the sporadic task model, a real-time task system (or task set) τ consists of n sequential tasks $\tau = \{\tau_1, \tau_2, \tau_3, \ldots, \tau_n\}$. Each task τ_i releases jobs recurrently with a minimum inter-release time separation defined by its *period* T_i. A task is *periodic* if its jobs are separated by *exactly* its period, or *sporadic* if its jobs are separated by *at least* its period, allowing inter-release delays. In the remainder of this dissertation, we focus on sporadic task systems unless explicitly stated otherwise.

Another parameter of relevance to a task τ_i is its worst-case execution requirement C_i, which defines an upper bound of the actual execution requirement per job. For a CPU task, C_i quantifies the worst-case execution time of τ_i on a CPU core, whereas we will define it differently for GPU tasks.

Each task has a *relative deadline* D_i, which defines its timing constraint: each of its jobs needs to be completed within D_i time units after the job's release time (defined below). A task system is categorized to have *implicit deadlines* if all the tasks' relative deadlines are equal to their periods ($D_i = T_i$ for all $\tau_i \in \tau$), *constrained deadlines* if all the tasks' relative deadlines have upper bounds defined by their periods ($D_i \leq T_i$ for all $\tau_i \in \tau$), or *arbitrary deadlines* otherwise.

Abstractly, a sporadic task τ_i can be characterized by a three-tuple (T_i, C_i, D_i). These parameters are known *a priori*. Informally and simply put, they define *how often* the jobs are released, *how much* processing capacity each job might require, and *how soon* each job needs to be finished, respectively. The period and execution requirement together define a task's need for processing capacity, *i.e.*, its *utilization*, $u_i = C_i/T_i$, and the *total utilization* of a task system is $U_{sum} = \sum u_i$.

Task τ_i's j^{th} job, denoted by $\tau_{i,j}$, has a *release time* $r_{i,j}$, a *finish time* $f_{i,j}$, and an *(absolute) deadline* $d_{i,j}$, where $d_{i,j} = r_{i,j} + D_i$. On many occasions, a more natural timing requirement relates to a job's *response time* $R_{i,j} = f_{i,j} - r_{i,j}$ or a task's *respons time* $R_i = max_j(R_{i,j})$. A related concept, *tardiness*, given by $x_{i,j} = max(0, f_{i,j} - d_{i,j})$, expresses the lateness of a job's completion.

2.4.2 Scheduling

Scheduling algorithms assign and schedule jobs on processors to produce a *schedule*, which is a particular assignment of all jobs in the task system.

Feasibility and schedulability. Feasibility and schedulability are used to describe the temporal correctness of real-time scheduling. A schedule is *feasible* if it meets all timing constraints. From a different perspective, a task set is *feasible* if a feasible schedule can be produced for it on a particular platform. Moreover, we say a task set is *schedulable* by a particular scheduling algorithm if the scheduling algorithm always produces a feasible schedule for the task set. We say a scheduling algorithm is *optimal* if all feasible task sets are schedulable by this scheduling algorithm.

To validate the temporal correctness of real-time systems, the goal of real-time analysis is to derive the procedures that examine if a task set is feasible and if it is schedulable by a given scheduling algorithm; these procedures are called feasibility tests and schedulability tests, respectively. Specifically, a *feasibility test* determines if there exists a scheduling algorithm under which a given task set is schedulable; a *schedulability test* determines if a given task set is schedulable under a given scheduling algorithm.

Scheduling algorithms. The modules that implement the scheduling algorithms to generate schedules are called *schedulers*, which can be either static or dynamic. *Static schedulers* produce task-set-specific schedules offline, and thus are easy to implement and convenient to certify, but are inflexible for handling workload changes. *Dynamic schedulers* generate schedules online, and thus can manage dynamic systems and consider task status changes while making scheduling decisions. In this dissertation, we focus on dynamic schedulers that are priority-driven.

Priority-driven schedulers assign priorities to jobs, and schedule the jobs with the highest priorities on the available processors. *Fixed-priority* (FP) schedulers determine jobs' priorities at a task level, *i.e.*, they assign the same priority to all the jobs of a task. For example, the fixed-priority *rate-monotonic* (RM) scheduler gives to the jobs of tasks with shorter periods higher priority (Liu and Layland, 1973). Similarly, the fixed-priority *deadline-monotonic* (DM) scheduler gives the jobs of tasks with shorter relative deadlines higher priority (Audsley *et al.*, 1991). As a generalization, *job-level fixed-priority* (JLFP) schedulers assign per-job priorities, *i.e.*, the jobs of a task can have different priorities. The simple *first-in-first-out* (FIFO) scheduler is a JLFP scheduler that gives jobs with an earlier release time higher priority. The *earliest-deadline-first* (EDF) scheduler is a well-studied JLFP scheduler that gives jobs with earlier absolute deadlines higher priority (Liu and Layland, 1973). We focus on JLFP schedulers in this dissertation. Schedulers that can change jobs' priorities at runtime are beyond the scope of our discussion.

We are specifically interested in the spectrum of JLFP schedulers that prioritize a job by a time instant within a bounded time window containing the job's release time—such a priority is *window-constrained* (Leontyev and Anderson, 2010). Both FIFO and EDF use window-constrained prioritizations. We discuss these JLFP schedulers and explain why we are interested in them in the Section 2.4.3.

Another categorization of schedulers is based on how they organize processors. A scheduler that can schedule any job on any one of m processors is called a *global scheduler*. Otherwise, a scheduler that assigns each task to one of a collection of disjoint sets of processors is called either *clustered or partitioned scheduler*, depending on the size of each set. The size of each set is equal to one under a *partitioned scheduler*. The size of each set may be larger than one under a *clustered scheduler*. Combined with a specific processor organization, the aforementioned schedulers have variants such as *global EDF* (or G-EDF), *clustered EDF* (or C-EDF), *partitioned RM* (or P-RM), *etc*.

2.4.3 Response-Time Bounds

The timing guarantees we derive for real-time tasks sharing GPUs are response-time bounds in soft real-time (SRT) systems, which have less rigorous validation requirements than hard real-time (HRT) systems. We define HRT and SRT in the following.

Hard and soft real time. HRT and SRT express different timing requirements. For application-specific deadlines, HRT imposes the strict requirement that all the deadlines must be met. In contrast, SRT tolerates deadline misses. Such tolerance of SRT can be defined from various perspectives. We refer readers who are interested in a comprehensive review on SRT definitions to prior works (Liu, 2000; Brandenburg, 2011). In the context of this dissertation, SRT requires all tasks' response times to be bounded. For such response-time bounds, of course, the shorter the better. Whether the derived response-time bounds are acceptable is determined by system requirements. For example, for autonomous driving systems, we say the end-to-end reponse-time bound, the bound of latency from image capture to vehicle-control-command ready time, should be shorter than an alert driver's reaction time, which is approximately 700 ms (Green, 2000).

We focus on SRT for autonomous-driving systems for the three reasons. First, most components of autonomous-driving systems such as deep-learning frameworks only run on general-purpose OSs like Linux, which is not a hard real-time OS (RTOS) even with patches like PREEMPT_RT or an EDF-like scheduler SCHED_DEADLINE. Guaranteeing HRT correctness on a general-purpose OS is difficult to do without introducing inefficiencies, and it is beyond the scope of this work. Although a small part of an autonomous-driving system, such as the low-level vehicle execution control, can be contained within an RTOS like QNX (BlackBerry, 2020) for HRT, this is not the case for the computer-vision components of interest. Second, both FP and JLFP global schedulers are subject to a utilization bound of $(m+1)/2$ in the HRT case (Andersson *et al.*, 2001), whereas JLFP global schedulers that use window-constrained prioritization are free of capacity loss in SRT cases (Leontyev and Anderson, 2010). In other words, these JLFP schedulers provide bounded response times for task sets that fully utilize multiprocessors (Leontyev and Anderson, 2010). Third, current computer-vision algorithms produce non-deterministic results with imperfect accuracy, which is compensated for by multiple sensory systems (lidar, radar, and camera). In such cases, enforcing strict HRT correctness is less beneficial than preventing its capacity loss and instead guaranteeing that results are ready within a bounded time as in SRT.

Prior work. We review a series of studies, in which the analysis techniques used for deriving response-time bounds are used in this dissertation as well. Devi and Anderson (2005) derived response-time bounds under preemptive and non-preemptive G-EDF without capacity loss. Leontyev and Anderson (2007) later established response-time bounds for G-FIFO, and followed up with a generalized analysis of JLFP schedulers that use window-constrained prioritization, which ensures the important characteristic that each job's priority will not be exceeded by that of any future job after a bounded time from the job's release (Leontyev and Anderson, 2010). They showed that this characteristic is sufficient for a global scheduler to guarantee response-time bounds. Erickson and Anderson (2012) then introduced *global fair-lateness scheduling* (G-FL), which uses linear programming to generate scheduling parameters (relative deadlines) that minimize the maximum response-time bounds under window-constrained schedulers.

2.4.4 Real-Time DAG Scheduling

As we reviewed in Section 2.2.3, many frameworks for computer-vision applications apply graph-based execution, as these applications—either classic pipelines or neural networks—are essentially graphs. In addition, task systems with precedent constraints between tasks on heterogeneous platforms can be well-modeled using DAGs. Thus, we review the prior work on real-time DAG scheduling herein, and we provide a formal DAG task model definition in Section 4.1.

Speedup bounds. The problem of HRT schedulability analysis for DAG tasks was shown to be computationally intractable by Baruah *et al.* (2012), so they used the notion of a *speedup bound* as a metric of the approximation quality. An algorithm A is said to have a speedup bound b if any task system that is feasible on m unit-speed processors is schedulable by A on m processors of speed b. A series of works were produced to tighten the speedup bound (Lakshmanan *et al.*, 2010; Andersson and de Niz, 2012; Nelissen *et al.*, 2012; Bonifaci *et al.*, 2013; Li *et al.*, 2013; Qamhieh *et al.*, 2013; Saifullah *et al.*, 2013; Baruah, 2014; Saifullah *et al.*, 2014; Qamhieh *et al.*, 2014; Jiang *et al.*, 2016).

Both Bonifaci *et al.* (2013) and Li *et al.* (2013) derived the speedup bound of $2 - 1/m$ for DAG tasks under G-EDF. This result was then generalized and improved by Baruah (2014). Meanwhile, Li *et al.* (2013) argued that because an optimal scheduler is hypothetical, a speedup bound is insufficient for schedulability tests, for which they proposed the notion of a *capacity augmentation bound*. An algorithm A is said to have a capacity augmentation bound of b if it produces feasible schedules on m processors of speed b for any task system that has a total utilization of at most m with each task's critical path length shorter than its relative deadline on unit-speed processors. Note that the critical path length of a DAG task is its WCET on infinite number of processors.

A decomposition approach was used in some studies to transform DAG tasks into independent sequential tasks by reassigning their scheduling parameters (relative deadlines) (Nelissen *et al.*, 2012; Saifullah *et al.*, 2013, 2014; Qamhieh *et al.*, 2014; Jiang *et al.*, 2016). Some researchers considered synchronous parallel tasks—a special form of DAG—that were phased into serial and parallel segments with synchronization points in between (Saifullah *et al.*, 2013; Andersson and de Niz, 2012; Nelissen *et al.*, 2012). Lakshmanan *et al.* (2010) studied speedup bounds under P-DM for a fork-join model—a special case of the synchronous parallel task model—where a *join* task is added after each parallel segment. Kim *et al.* (2013) later extended the fork-join model to support an arbitrary number of threads under G-DM.

Response-time analysis for HRT. Response-time analysis (RTA) explicitly computes a response-time bound for each task, and determines HRT schedulability by checking if such bounds are within tasks' relative deadlines. This is not to be confused with SRT in which the bounds are allowed to exceed the relative deadlines. RTA has been applied to derive schedulability tests for the fork-join task model (Axer *et al.*, 2013), synchronous parallel tasks (Chwa *et al.*, 2013; Maia *et al.*, 2014), and DAG tasks (Qamhieh *et al.*, 2013, 2014; Serrano *et al.*, 2016, 2017; Nasri *et al.*, 2019).

Response-time bound analysis for SRT. Similar to the decomposition method, another transformation approach was proposed for periodic and sporadic DAG tasks in SRT systems (Liu and Anderson, 2010). These DAG tasks were transformed to ordinary sporadic tasks by guaranteeing precedent constraints with reassigned release times, which we further illustrate with examples in Section 4.1. Response-time bounds could then be derived for the resultant sporadic tasks under G-EDF. An early-release technique was used to retain work-conserving execution by allowing jobs to execute before their release times. They later applied this approach in a study of the communication cost for distributed systems (Liu and Anderson, 2011). This transformation was later applied by Yang *et al.* (2016b) for DAG tasks on heterogeneous platforms, who reduced response-time bounds by combining DAGs and optimizing deadline settings using linear programming. Later, Dong *et al.* (2017) further reduced response-time bounds by replacing the transformation approach with a direct analysis using techniques introduced by Devi and Anderson (2005).

Some works examined other characteristics of DAG tasks. A federated scheduling was studied that differs from partitioned scheduling in that a task is either partitioned to a processor that might be shared with other tasks, or is assigned to a set of processors with exclusive access (Li *et al.*, 2014; Baruah, 2015a,b,c; Jiang *et al.*, 2017; Li *et al.*, 2017; Baruah, 2018). A DAG task may have conditional branches that need to be considered (Baruah, 2015c; Baruah *et al.*, 2015; Fonseca *et al.*, 2015; Melani *et al.*, 2015, 2016; Baruah, 2018). To balance the trade-off between preemptive and non-preemptive execution, a limited preemptive scheme was considered (Serrano *et al.*, 2016, 2017; Nasri *et al.*, 2019).

2.4.5 Real-Time Frameworks

We describe the real-time frameworks that we use for implementing and evaluating real-world applications and our real-time analysis.

LITMUS$^{\text{RT}}$. **LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems (LITMUS$^{\text{RT}}$), an open-source real-time extension on Linux, was used in our experiments to provide G-EDF and G-FL schedulers for CPU tasks. LITMUS$^{\text{RT}}$ was initiated at UNC by Calandrino *et al.* (2006) and developed and currently maintained by Brandenburg (2011) as the LITMUS$^{\text{RT}}$ Project (2018). It supports the sporadic task model and scheduler plugins for clustered, partitioned, and global scheduling. Along with the kernel extension, Feather-Trace, a tracing tool for obtaining execution-time measurements (Brandenburg and Anderson, 2007), and `liblitmus`, a user-space library for implementing real-time tasks, are available. A Linux-based OS is necessary for our evaluation of computer-vision workloads and GPU execution, as the computer-vision frameworks and GPU drivers are well supported on Linux.

PGM$^{\text{RT}}$. We use PGM$^{\text{RT}}$ (Elliott *et al.*, 2014), a real-time middleware that manages processing graphs, which is a general form of DAG. Our application uses PGM$^{\text{RT}}$ APIs to represent sporadic tasks created with `liblitmus` as nodes and enforce precedent constraints with edges. In PGM$^{\text{RT}}$, the readiness of the data passed on edges is managed in a producer-consumer scheme using *tokens*, and in our DAG task model, each job produces one token, but may consume multiple tokens from multiple incoming edges.

2.4.6 Real-Time GPUs

We now review prior work related to managing GPUs in real-time systems.

Exclusive GPUs. Limited by the unknown details of GPU scheduling rules, many works have treated GPUs as non-shared resources. We review these works herein.

TimeGraph is a device-driver level GPU management framework that prioritizes and isolates GPU tasks in an SRT setting (Kato *et al.*, 2011b). It is implemented on an open-source GPU driver, Nouveau (X.org Foundation, 2020), to support OpenGL, and has been ported to the PSCNV[11] GPU driver to support CUDA; the latter, however, has been discontinued. TimeGraph provides two scheduling policies: one that prioritizes tasks to prevent performance interference, and another that reduces scheduling intervention to favor throughput-oriented tasks at the cost of longer response times and priority inversions. It also provides budget-control-based performance-isolation mechanisms: one that permits GPU kernel submission as long as the remaining budget is positive, and imposes punishment for any resulting overrun (into negative budget) at the next replenishment, and another that profiles and predicts the execution time of GPU tasks to strictly prevent possible overruns. The execution-time profiling is achieved with a history table that maps methods and data sizes to execution times, provided that these methods are invoked repeatedly.

RGEM, another work by Kato *et al.* (2011a), is instead a user-space CUDA runtime library, which prioritizes GPU requests before sending them to the device driver. Like TimeGraph, RGEM prioritizes GPU kernels. Rather than treating GPUs as a whole as does TimeGraph, RGEM manages copy and kernel operations separately, allowing parallelism between CEs and EEs. Additionally, RGEM splits memory-copy operations into chunks to reduce blocking time.

GPUSync, a GPU management system with rich features, explicitly addresses GPU management as a synchronization problem (Elliott *et al.*, 2013). Specifically, on the multi-GPU systems that GPUSync targets, GPU allocation (task-to-GPU assignment) is managed by a *k*-exclusion locking protocol that arbitrates access to multiple GPUs. Similar to RGEM, CEs and EEs are managed separately. Task migration between GPUs is supported with consideration for memory-based affinity and migration costs such as bus interference. Moreover, to prevent the misbehavior of overutilization, GPUSync applies several budget-enforcement policies. In contrast to the profiling-based overrun control in TimeGraph, GPUSync recovers from overrun with three strategies: graceful exit upon overrun signals, early release of overrunning tasks' future jobs,

[11]PathScales fork of the Nouveau driver.

and budget inheritance. Furthermore, GPUSync handles interrupts with proper priorities by mapping GPU requests to GPUs; the handlers (worker threads) are prioritized with the highest-priority tasks assigned to the GPU that raises the interrupt.

Kernel slicing. We mentioned that RGEM splits memory transactions into smaller chunks. Such slicing techniques, which we reviewed for non-real-time GPUs in Section 2.3.4.3, are commonly used to simulate preemptive execution for real-time systems as well. In addition to memory copy slicing, a *preemptive kernel model* (PKM), implemented by Basaran and Kang (2012), splits kernels into subkernels, each of which executes a subset of thread blocks. Both RGEM and PKM require source-code modification for kernel slicing, whereas GPES (Zhou *et al.*, 2015), a framework implemented atop Gdev (Kato *et al.*, 2012), further enables transparent and automatic kernel-slicing techniques: compile-time source-code transformation and just-in-time code rewriting. However, kernel slicing is not yet applicable for all applications, such as ones that require global synchronization between all GPU thread blocks.

Timing analysis. Timing analysis for GPUs measures or analyzes a task's worst-case execution-time (WCET), one of the timing parameters used in schedulability analysis. Such timing analysis for GPU tasks is challenging because of the complex architecture of GPUs, which is different from that of CPUs. In particular, GPU execution is based on the SIMT model with massively parallel threads managed by undisclosed low-level scheduling control.

Berezovskyi *et al.* (2012) first formed an integer linear program (ILP) to compute the WCET of a group of threads on a single SM, and later improved the speed of this method with heuristics to produce an estimate (Berezovskyi *et al.*, 2013). Their follow-up work switched to a measurement-based statistical approach that extended to multiple SMs (Berezovskyi *et al.*, 2014, 2016). Horga *et al.* (2018) also proposed a measurement-based approach that combines symbolic execution and a genetic algorithm. Several works provided WCET analysis with information provided by a GPU simulator GPGPU-sim (Betts and Donaldson, 2013; Huangfu and Zhang, 2017b,a, 2018). None of these works considered WCET with CKE.

Resource management. Many GPU resource-management studies have been reviewed in Section 2.3.4.2. We now review ones that specifically target real-time GPUs. The aforementioned real-time GPU management systems require OS kernel or device-driver modification, which requires the maintenance effort of adapting the kernel-level implementation to the up-to-date codebase. To mitigate such effort, Suzuki *et al.* (2016)

implemented a loadable kernel module (LKM)-based real-time GPU resource management atop Gdev. Although it avoids kernel patches, it requires CUDA applications to use the API LKM provides.

In contrast to managing the GPU from the CPU side, a server-based approach for GPU resource management has been demonstrated: Kim *et al.* (2017, 2018) proposed creating GPU server tasks to handle GPU requests, enabling GPU task suspension and providing schedulability analysis.

Compared with GPU management that allows exclusive access only, fewer studies have investigated GPU sharing. Jain *et al.* (2019) enabled both computational and memory resource reservation for CKE, with SM-centric task assignment and page-coloring techniques based on knowledge of memory structures obtained via reverse engineering. Saha *et al.* (2019) also applied SM allocation and provided schedulability analysis with response-time bounds. Another SM partitioning work (Xu *et al.*, 2016b) considered sharing between real-time and best-effort tasks. Baek *et al.* (2020) proposed managing CKE through a middleware that improved the processing frame rate with resource monitoring and scheduling by considering application characteristics.

Other GPU scheduling. Zhou *et al.* (2018) studied the problem of real-time scheduling of DNN workloads with system-level performance considerations. They proposed S^{3}DNN, which applied batching to aggregate requests via data fusion, deadline-aware scheduling, and reordering kernels with the DNN structure known *a priori* to improve GPU utilization. While S^{3}DNN focuses on a single DNN, Xiang and Kim (2019) proposed a pipeline-based approach to improve the WCET of multiple DNN applications.

Capodieci *et al.* (2018) explored hardware-enabled preemption support for implementing an EDF scheduler with a *Constant Bandwidth Server* (CBS) (Spuri and Buttazzo, 1994) that provides bandwidth reservation and isolation between tasks. Although their implementation included techniques that were not revealed due to NDA restrictions, in-depth information about GPU scheduling for work submitted from multiple CPU processes was provided. Specifically, they revealed the data structures that manage GPU commands for multiple processes, and how these commands were organized and scheduled. Their implementation managed the movement of these data structures at the software level in a virtualized environment provided by NVIDIA, and changed the NVIDIA-implemented polling mechanism to an event-driven scheduler. They tested EDF and CBS schedulers that, however, only allowed requests from one application to be submitted at a time.

Cavicchioli *et al.* (2019) investigated the applicability of the new GPU programming standard Vulkan, CUDA graphs, and CUDA dynamic parallelism techniques for real-time systems. Specifically, they applied these techniques to reduce CPU-to-GPU interaction overheads, which were shown to be performance bottlenecks, especially for frequent GPU command submissions.

CPU-GPU memory interference. Interactions between CPU and GPU tasks have been investigated in other works as well. Forsberg *et al.* (2017) presented GPUGuard to isolate memory access from CPU and GPU tasks into phases, inspired by the PRedictable Execution Model (PREM). PREM was also extended to an SoC device, the NVIDIA Jetson TX1, by modifying MemGuard (Houdek *et al.*, 2017). Similarly, Ali and Yun (2018) leveraged MemGuard to throttle best-effort CPU tasks to control memory interference. The memory interference between CPU and GPU tasks was characterized by Cavicchioli *et al.* (2017), and later addressed in a server based approach (Capodieci *et al.*, 2017).

2.5 Chapter Summary

This chapter provided our research context for autonomous driving, the computer-vision applications of our focus, and reviewed the fundamentals and relevant research for GPUs and real-time systems that will be used in subsequent chapters.

In reviewing the state-of-the-art progress pertaining to autonomous driving, we learned that full autonomy has yet to be realized, with real-time guarantees being one of the challenges. Our review of the primary system components provides a context for explaining how the computer-vision applications that we focus on function in an autonomous-driving system.

We reviewed the milestones of computer vision across the stages of classic pipelines and deep learning, using the example of object detection. We also described metrics for evaluating object-detection accuracy, computer-vision frameworks and libraries, and the computer-vision standard OpenVX, which targets embedded and real-time computer vision. We summarized the challenges of supporting computer-vision workloads in a real-time certification-amenable framework.

We presented fundamental background about GPU hardware architecture and programming models, and a brief comparison between GPUs and other accelerators (FPGAs and DSPs). We provided a comprehensive review of studies of GPUs in both real-time and non-real-time settings.

We described the basic concepts of real-time systems and scheduling, including task models, timing criteria, classic schedulers, and our focus on SRT. We also reviewed prior work on real-time DAG scheduling, the frameworks we use, and real-time GPUs.

CHAPTER 3: SCHEDULING, SYNCHRONIZATION, AND EXECUTION[1]

In the preceding chapters, we have presented the challenge of sharing GPUs with guaranteed bounded response times. To address this challenge, the first step we undertake is to investigate NVIDIA GPUs with respect to its scheduling, synchronization, and execution.

We first present an in-depth study of GPU scheduling on an exemplar of current GPUs targeted towards autonomous systems. This study was conducted using only black-box experimentation and publicly available documentation. This is part of our effort to develop a model for describing GPU workloads and how they are scheduled. Our eventual goal is to develop a model for which real-time schedulability-analysis results can be derived, as has been done for various CPU workload and scheduling models that have been studied for years.

The exemplar chosen for this study is the NVIDIA TX2 we reviewed in Section 2.3.2.1. The TX2 is part of the Jetson family of embedded computers, which is explicitly marketed for "autonomous everything" (NVIDIA, 2020a). Moreover, it shares a common GPU architecture with the higher-end Drive PX2, which is currently available only to automotive companies and suppliers. The TX2 has two important attributes for embedded use cases: it provides significant computing capacity, and meets reasonable limits on monetary cost as well as size, weight, and power (SWaP). The TX2 is a very complicated device, so discerning exactly how it functions is not easy.

We have noted that NVIDIA GPU scheduling differs based on whether work is submitted to a GPU from a CPU executing **(i)** OS threads that share an address space or without inter-thread memory protections, or **(ii)** OS processes that have different address spaces that provide memory protection. However, to the best of

[1]Contents of this chapter previously appeared in the following papers:

Otterness, N., Yang, M., Amert, T., Anderson, J., and Smith, F. D. (2017a). Inferring the scheduling policies of an embedded cuda gpu. In *Proceedings of the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 47–52.

Amert, T., Otterness, N., Yang, M., Anderson, J. H., and Smith, F. D. (2017). GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *Proceedings of the 38th IEEE International Real-Time Systems Symposium*, pages 104–115.

Yang, M., Otterness, N., Amert, T., Bakita, J., Anderson, J. H., and Smith, F. D. (2018b). Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems*, pages 20:1–20:21.

our knowledge, the exact manner in which GPU scheduling is done in either case has never been publicly disclosed.

For Case (i), we present a set of rules (in Section 3.1), based on experimental evidence involving benchmarking programs, that fully define how the TX2's GPU schedules work submitted to it, assuming that certain optional GPU features that introduce additional complexities (see below) are not employed that introduce additional complexities. These rules take into account these aspects: **(i)** the ordering within and among *streams* of requests for GPU operations, **(ii)** the means by which various internal queues of requests are handled, **(iii)** the ability of the GPU to co-schedule operations so they execute concurrently, **(iv)** the selection mechanism used to determine the order in which requests are handled, and **(v)** the resource limits that constrain the GPU's ability to handle new requests. Our rules indicate that the TX2's GPU employs a variant of hierarchical FIFO scheduling that is amenable to real-time schedulability analysis, although scheduling is not fully work-conserving, as GPU computations are subject to various blocking delays.

The optional GPU features that we initially ignore include two features that cause certain request streams to be treated specially, which further complicates GPU scheduling. These features include the usage of a special stream called the *NULL stream* and the usage of *stream priorities*. The available documentation regarding both of these features often lacks details necessary for predicting specific runtime behavior. Through further experiments, we show how each of these features affects our derived scheduling rules (in Section 3.2).

GPU programs are most commonly developed assuming that different GPU computations are requested by OS processes with different address spaces, so Case (ii) above applies. We investigated GPU scheduling in this case as well (Section 3.2) and, unfortunately, found it to be less deterministic than in Case (i). In particular, concurrent GPU computations in this case are multiprogrammed on the GPU using time slicing in a way that can add significant overhead and execution-time variation. This result calls into question the common practice today of following a process-oriented approach in developing GPU applications, as reflected in open-source frameworks such as PyTorch[2] and Caffe.[3] We also investigated the usage of stream priorities under Case (ii) and found that they have no effect in this case (Section 3.2).

In addition to revealing NVIDIA GPU scheduling rules, we hope to provide guidance, recommendations, and warnings about numerous pitfalls to both research and implementation practitioners. We have found that writing programs for real-time tasks that combine CPU and GPU computations is harder than we first

[2]`http://pytorch.org`

[3]`http://caffe.berkeleyvision.org`

thought. Based on several years of study, experimentation, and experience with GPU programming, we are presenting here a compendium of specific issues that are essential background for developing task systems where real-time design meets GPUs.

NVIDIA GPUs serve as an exemplar of the push for throughput over predictability in GPUs. Recent developments in the NVIDIA GPU ecosystem are focused on improving ML applications, especially those for autonomous driving. Most of these improvements center around increasing throughput or reducing execution latency, but little, if any, attention has been paid to the requirements of the real-time tasks used in autonomous systems. This lack of attention is evident in the sparse efforts by NVIDIA to improve or document GPU scheduling behavior or improve the predictability of GPU execution times. For example, for a task consisting of a combination of CPU and GPU computations, their synchronization is necessary but is not well-documented, resulting in difficulties in determining why and when synchronization blocking occurs (Section 3.3), which impacts scheduling analysis.

Besides the aforementioned cases regarding how work is submitted to a GPU from a CPU, a third option that leverages a middleware environment—CUDA Multi-Process Service (MPS) (NVIDIA, 2019e)—is claimed to provide the best approach offering both memory isolation and concurrent execution on GPUs. We performed a case study using exemplar computer-vision tasks in autonomous vehicles to evaluate these execution options, with our results and guidelines presented in Section 3.4.

Lastly, we present a list of perils one can encounter in programming for NVIDIA GPUs, distilled from our research experience that has necessarily involved contructing many thousands of lines of GPU programming for performing experiments. The perils span a spectrum of pain ranging from simple documentation errors to functions that default in strange ways, to programming "gotchas." We present a list of perils with descriptions and examples of the ones most likely to cause problems in Section 3.5.

We believe that our investigation and findings will help bridge the gap between research and implementation in autonomous systems. For example, real-time researchers may not be familiar with GPU programming for applications of ML and other forms of AI used in real-time tasks. Likewise, programmers responsible for implementations are given little guidance about creating GPU-using task systems amenable to real-time analysis. We provide the necessary understanding required to apply GPUs in real-time tasks while avoiding numerous hidden pitfalls. We also expose GPU-related issues that must be mitigated for real-time guarantees to be possible in autonomous systems. We further believe that the fundamental issues presented herein are relevant to any real-time application using computational accelerators, and likely hold for other manufacturers' GPUs, DSPs, or FPGAs.

3.1 Basic GPU Scheduling Rules

In this section, we present GPU scheduling rules for the TX2 assuming the GPU is accessed only by CPU tasks[4] that share an address space, using only user-specified streams. (We consider the NULL stream later. Unless stated otherwise, "stream" should henceforth be taken to mean a user-specified stream.) We inferred the scheduling rules given here by conducting extensive experiments using CUDA 8, 9, and 10. We begin by discussing one of these experiments in Section 3.1.1. We use this experiment as a continuing example to explain various nuances of the rules, which are covered in full in Section 3.1.2. Before continuing, we note that all code, scripts, and visualization tools developed for this chapter are available online.[5]

3.1.1 Motivating Experiment

The experiment considered here involved running instances of a synthetic benchmark that launches several kernels. Our synthetic workload allows flexibility in configuring block resource requirements, kernel durations, and copy operations. We have also experimented with a variety of real-world GPU workloads involving image-processing functions common in autonomous-driving use cases, and to our knowledge, the scheduling rules presented in this paper are valid for such workloads as well.

In the experiment considered here, we configured each block of each benchmark kernel to spin for one second. As detailed in Table 3.1, three kernels, K1, K2, and K3, were launched by a single task to a single stream, and three additional kernels, K4, K5, and K6, were launched by a second task to two separate streams. The kernels are numbered by launch time. Copy operations occurred after K2 and K5, before and after K3 and K6, in their respective streams.

Figure 3.1 depicts the GPU timeline produced by this experiment. Each rectangle represents a block: the j^{th} block of kernel Kk is labeled "Kk:j." The left and right boundaries of each rectangle correspond to that block's start and end times, as measured on the GPU using the `globaltimer` register. The height of each rectangle is the number of threads used by the block. The y-position of each rectangle indicates the SM upon

[4]We use the term *task* to refer to OS threads that share an address space, and we use the term *thread* to mean a hardware thread on GPU as defined ealier in Section 2.3.3.

[5]`https://github.com/yalue/cuda_scheduling_examiner_mirror`.

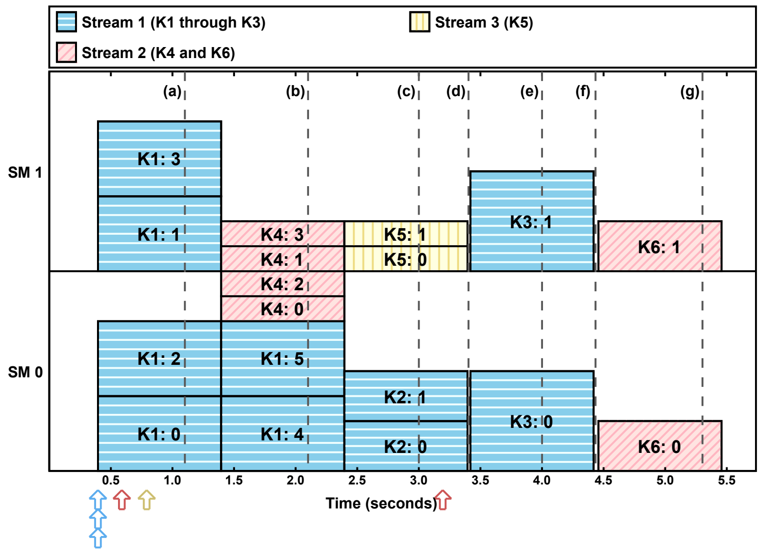


Figure 3.1: Basic GPU scheduling experiment.

which it executed. Arrows below the *x*-axis indicate kernel launch times. Dashed lines correspond to time points used in the continuing example covered in Section 3.1.2.

3.1.2 Scheduling Rules

With respect to kernels, blocks are the schedulable entities: the basic job of the GPU scheduler is to determine which thread blocks can be scheduled at any given time. These scheduling decisions are impacted by the availability of limited GPU *resources* that blocks utilize such as GPU shared memory, registers, and threads. Required resources are determined for the entire kernel when it is *launched*. All blocks in a given kernel have the same resource requirements.

We say that a block is *assigned* to an SM when that block has been scheduled for execution on the SM. A kernel is *dispatched* when at least one of its blocks is assigned, and is *fully dispatched* once all of its blocks have been assigned. Similarly, we say that a copy operation is *assigned* to a CE once it has been selected to be performed by the CE.

Example. Figure 3.2 provides additional details regarding the kernel launches in Figure 3.1. In Figure 3.2, we use additional notation to depict copies: C*k*i denotes an input copy operation of kernel K*k*, and C*k*o denotes an output copy operation of kernel K*k*. Each inset in Figure 3.2 corresponds to the time point in Figure 3.1 with the same designation (*e.g.*, inset (a) corresponds to the time point labeled "(a)"). We will

Kernel	Launch Info	Start Time (s)	# Blocks	# Thread per Block	Shared Memory per Block	Copy In	Copy Out
K1	CPU 0, Stream S1	0.0	6	768	0	-	-
K2	CPU 0, Stream S1	0.0	2	512	0	-	256MB
K3	CPU 0, Stream S1	0.0	2	1,024	0	256MB	256MB
K4	CPU 1, Stream S2	0.2	4	256	32KB	-	-
K5	CPU 2, Stream S3	0.4	2	256	32KB	-	256MB
K6	CPU 1, Stream S2	2.8	2	512	0	256MB	256MB

Table 3.1: Details of kernels used in the experiment in Figure 3.1. (Note that "start times" are defined relative to the end of benchmark initialization.)

repeatedly revisit Figure 3.2 to illustrate individual scheduling rules as they are stated, and then consider the entire example in full once all rules have been stated.

General scheduling rules. To our knowledge, the actual data structures used by the TX2 to schedule copy operations and kernels are undocumented. From our experiments, we hypothesize that several queues are used: one FIFO *EE queue* per address space, one FIFO *CE waiting queue* and one FIFO *CE queue* that are used to order copy operations for assignment to the GPU's CE, and one FIFO queue per CUDA stream (including the NULL stream, which we consider later). We refer to the latter as *stream queues*. We begin by listing general rules that specify how copy operations and kernels are moved between queues:

G1 A copy operation or kernel is enqueued on the stream queue for its stream when the associated CUDA API function (memory transfer or kernel launch) is invoked.

G2 A kernel is enqueued on the EE queue when it reaches the head of its stream queue.

G3 A kernel at the head of the EE queue is dequeued from that queue once it becomes fully dispatched.

G4 A kernel is dequeued from its stream queue once all of its blocks complete execution.

Example (cont'd). In Figure 3.2, there are two CUDA programs executing as CPU tasks τ_0 and τ_1 on CPUs 0 and 1, respectively, that share an address space. τ_0 uses a single stream, S1, and τ_1 uses two streams, S2 and S3. The various queues, two SMs, and single CE are depicted in each inset of Figure 3.2. The start times in Table 3.1 give the time the kernel, or its input copy operation if one exists, was issued. Output copy operations, when present, immediately followed kernel completions.

In inset (a), τ_0 has issued kernels K1, K2, and K3 and copy operations C2o, C3i, and C3o to stream S1. τ_1 has issued kernels K4 and K5 to streams S2 and S3, respectively, and copy operation C5o to stream S3. The operations in streams S1 and S3 were enqueued in the order the CUDA commands were executed

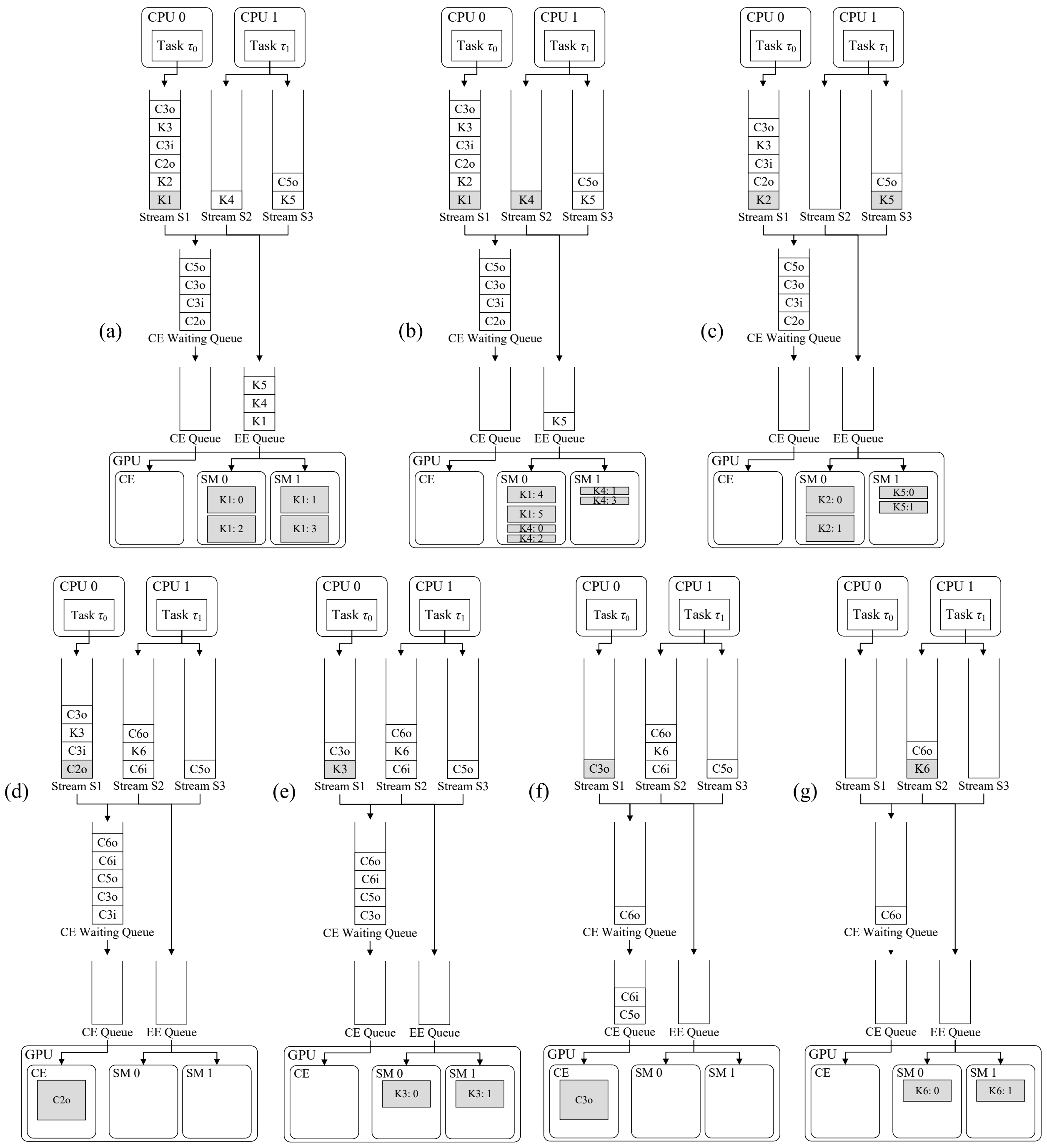


Figure 3.2: Detailed state information at various time points in Figure 3.1.

(Rule **G1**). Kernels K1, K4, and K5 are at the heads of their respective stream queues, and have been placed in the EE queue (Rule **G2**). K1 is dispatched to the GPU, so it is shaded in its stream queue. Each SM has two blocks of K1 assigned to it. In inset (b), the remaining blocks of K1 and K4 have been assigned to the GPU, so both K1 and K4 have been removed from the EE queue (Rule **G3**), but remain in their respective stream queues (Rule **G4**).

Non-preemptive execution. Ignoring complications considered later in Section 3.2, the kernel at the head of the EE queue will non-preemptively block later-queued kernels:

X1 Only blocks of the kernel at the head of the EE queue are eligible to be assigned.

Example (cont'd). In Figure 3.2 (a), two blocks of K1 have been assigned to each SM. As explained in detail below, on each SM, there are enough resources available that two blocks from K4 could execute concurrently with the two blocks from K1. However, K4 is not at the head of the EE queue, so its blocks are not eligible to be assigned (Rule **X1**).

Rules governing thread resources. On the TX2, the total thread count of all blocks assigned to an SM is limited to 2,048 per SM,[6] and the total number of threads each block can use is limited to 1,024. These resource limits can delay the kernel at the head of the EE queue:

R1 A block of the kernel at the head of the EE queue is eligible to be assigned only if its resource constraints are met.

R2 A block of the kernel at the head of the EE queue is eligible to be assigned only if there are sufficient thread resources available on some SM.

Example (cont'd). In Figure 3.2 (a), K1 is at the head of the EE queue, so its blocks are eligible to be assigned (Rule **R1**). However, only four blocks of K1 have been assigned. This is because, as seen in Table 3.1, each block of K1 requires 768 threads, and on the TX2, each SM is limited to allocate at most 2,048 threads at once. Thus, only four blocks of K1's six can be scheduled together, so the remaining two must wait (Rule **R2**). In inset (b), these remaining two blocks have been scheduled. K4, next in the EE queue, requires four blocks of only 256 threads each, so all of its blocks fit on the GPU simultaneously with the remaining two blocks of K1, and all four blocks of K4 are assigned.

[6]Recall from Section 2.3.2.1 that on the TX2, each SM has 128 hardware cores available. The (up to) 2,048 threads assigned to blocks currently executing on that SM are multiprogrammed on those cores.

Rules governing shared-memory resources. Another constrained resource is the amount of GPU memory shared by threads in a block. On the TX2, shared-memory usage is limited to 64KB per SM and 48KB per block. Similar to threads, a block being considered for assignment will not be assigned until all of the shared memory it requires is available.

R3 A block of the kernel at the head of the EE queue is eligible to be assigned only if there are sufficient shared-memory resources available on some SM.

Example (cont'd). Each block of K4 or K5 requires 32KB of shared memory, so at most two of these blocks can be assigned concurrently on an SM. In Figure 3.2 (b), two blocks of K4 are assigned to each SM. Even though there are available threads for K5 to be assigned and it is the head of the EE queue at that time, no block of K5 is assigned until the blocks of K4 complete (Rule **R3**), as shown in inset (c).

Register resources. Another resource constraint is the size of the register file. On the TX2, each thread can use up to 255 registers, and a block can use up to 32,768 registers (regardless of its thread count). Additionally, there is a limit of 65,536 registers in total on each SM. Unfortunately, using synthetic kernels makes it difficult to demonstrate limits on registers because the NVIDIA compiler optimizes register usage. However, based upon available documentation (NVIDIA, 2019b), we expect limits on register usage to have exactly the same impact as the limits on thread and shared-memory resources demonstrated above. Note also that the number of registers can be limited at compile-time using the `maxregcount` compiler option. Decreasing the number of registers used by a kernel makes its blocks easier to schedule at the expense of potentially greater execution time.

Copy operations. As noted earlier, the TX2 has one CE to process both host-to-device and device-to-host copies. Such copies are governed by rules similar to those above:

C1 A copy operation is enqueued on the CE waiting queue when the associated CUDA API function is invoked.

C2 A copy operation is enqueued on the CE queue when it reaches the head of the CE waiting queue and its stream queue.

C3 A copy operation at the head of the CE queue is eligible to be assigned to the CE.

C4 A copy operation at the head of the CE queue is dequeued from the CE queue once the copy is assigned to the CE on the GPU.

C5 A copy operation at the head of the CE waiting queue is dequeued from the CE waiting queue once it is enqueued on the CE queue.

C6 A copy operation is dequeued from its stream queue once the CE has completed the copy.

Example (cont'd). As shown in Figure 3.2 (a), once C2o, C3i, C3o, and C5o are issued, they are enqueued on the CE waiting queue (Rule **C1**). Upon being issued, C6i and C6o are enqueued on the CE waiting queue (Rule **C1**). In inset (d), once K2 and K5 complete execution and are dequeued from S1 and S3, the copies C2o and C5o become the heads of S1 and S3, repectively. However, only C2o reaches the head of the CE waiting queue. Thus, C2o is the only copy operation enqueued on the CE queue (Rule **C2**) and dequeued from the CE waiting queue (Rule **C5**). C2o is immediately assigned to the CE (Rule **C3**), so it is dequeued from the CE queue (Rule **C4**).

Example (cont'd). As shown in Figure 3.2 (d), once K2 and K5 complete execution and are dequeued from S1 and S3, the copies C2o and C5o become the heads of S1 and S3, respectively. C5o and C2o are thus enqueued on the CE queue (Rule **C2**). C5o is immediately assigned to the CE (Rule **C3**), so it is dequeued from the CE queue (Rule **C4**). The CE can perform only one copy operation at a time, so C2o remains at the head of the CE queue until C5o completes.

Full example. Inset (a) of Figure 3.2 corresponds to time $t = 1.1s$ in Figure 3.1. The first five kernels have been launched, and the kernels at the heads of each stream, K1, K4, and K5, have also been added to the EE queue, in issue order. K1 remains in the EE queue as it is not yet fully dispatched, so no blocks of K4 are eligible to be assigned. The copy operations C2o, C3i, C3o, and C5o are launched and enqueued on the CE waiting queue, in issue order.

Inset (b) corresponds to time $t = 2.1s$. The first four blocks of K1 have finished executing. Both K1 and K4 are now fully dispatched, so they have been removed from the EE queue, but remain at the heads of their stream queues. No blocks of K5 are able to be dispatched because their required shared-memory resources are not available.

Inset (c) corresponds to time $t = 3.0s$. Upon completion of K1 and K4, both of K5's blocks are assigned, and K2 is added to the EE queue and immediately becomes fully dispatched. C2o remains blocked by K2 in its stream queue due to FIFO stream ordering. C5o is similarly blocked.

Inset (d) corresponds to time $t = 3.4s$. K2 and K5 both complete execution, enabling C2o to be enqueued on the CE queue. C2o is assigned to the CE and removed from the CE queue. Due to FIFO stream ordering, both C3i and K3 are delayed behind C2o. Upon being issued, the copy operations C6i and C6o are enqueued on the CE waiting queue. Due to FIFO stream ordering, K6 is delayed behind C6i.

Inset (e) corresponds to time $t = 4.0s$. K3 is fully dispatched. C3o is blocked in its stream queue until K3 completes. Due to the FIFO CE waiting queue ordering, C5o and C6i are unnecessarily delayed befind C3o. As a result, the CE and EE queues are empty.

Inset (f) corresponds to time $t = 4.43s$. K3 completes execution, enabling C3o to be enqueued on the CE queue. C5o and C6i are both enqueued on the CE queue. C3o is assigned to the CE and removed from CE qeueue. The CE can perform only one copy operation at a time, so C5o remains at the head of the CE queue until C3o completes.

Inset (g) corresponds to time $t = 5.3s$. K6 is fully dispatched. C6o is blocked in its stream queue until K6 completes.

Summary. The basic scheduling rules above define a variant of hierarchical FIFO scheduling where work may sometimes be subject to blocking delays. As FIFO CPU scheduling has analyzable response-time bounds on multiprocessors (Leontyev and Anderson, 2007), the scheduler defined by these rules is also amenable to real-time schedulability analysis, as we will show in Chapter 4.

3.2 Additional Complexities

Unfortunately, the basic rules in Section 3.1 are not the end of the story. In this section, we consider additional features available to CUDA programmers that can impact scheduling. These include usage of the default (NULL) stream, stream priorities, and streams in independent process address spaces. We also comment on other less commonly used features that can influence scheduling that we do not examine in detail.

3.2.1 The NULL Stream

Available documentation makes clear that two kernels cannot run concurrently if, between their issuances, any operations are submitted to the NULL stream (NVIDIA, 2019b). However, this documentation does not explain how kernel execution order is affected. In an attempt to elucidate this behavior, we conducted experiments in which interactions between (user-specified) streams and the NULL stream were observed.

Kernel	Launch Info	Start Time (s)	Duration (s)	# Blocks	# Thread per Block
K1	Stream S1	0.0	1.0	6	768
K2	NULL Stream	0.2	1.0	1	1,024
K3	Stream S2	0.2	1.0	4	256
K4	Stream S2	0.4	1.0	4	256
K5	NULL Stream	0.6	1.0	1	1,024
K6	Stream S3	0.8	1.0	2	256

Table 3.2: Details of kernels used in the NULL-stream scheduling experiment in Figure 3.3.

We found that these interactions are governed by the following rules, which reduce to rule **G2** if the NULL stream is not used:

N1 A kernel K*k* at the head of the NULL stream queue is enqueued on the EE queue when, for each other stream queue, either that queue is empty or the kernel at its head was launched after K*k*.

N2 A kernel K*k* at the head of a non-NULL stream queue cannot be enqueued on the EE queue unless the NULL stream queue is either empty or the kernel at its head was launched after K*k*.

The result of an experiment demonstrating these rules is given in Figure 3.3. The kernels launched in this experiment are fully specified in Table 3.2. K2 and K5 were submitted to the NULL stream. K2 did not move to the EE queue until K1 completed (Rule **N1**); likewise, K5 did not move to the EE queue until both K3 and K4 had completed. No other kernel could move to the EE queue while K2 was at the head of the NULL stream queue, as all but K1 were launched after K2 (Rule **N2**). Because of the NULL-stream kernels, K6 was unnecessarily blocked and could not execute concurrently with K1, K3, or K4, so much capacity was lost. This result demonstrates that usage of the NULL stream is problematic if real-time predictability and efficient platform utilization are desired.

3.2.2 Stream Priorities

We now consider how the usage of prioritized streams impacts the rules defined in Section 3.1.2. CUDA programmers can prioritize some streams over others and can determine the allowable priority settings by the API call `cudaDeviceGetStreamPriorityRange`. On the TX2, this call returns only two priority values: -1 (*priority-high*) and 0 (*priority-low*). As we show below, a stream with no priority specified (*priority-none*) is treated as priority-low.

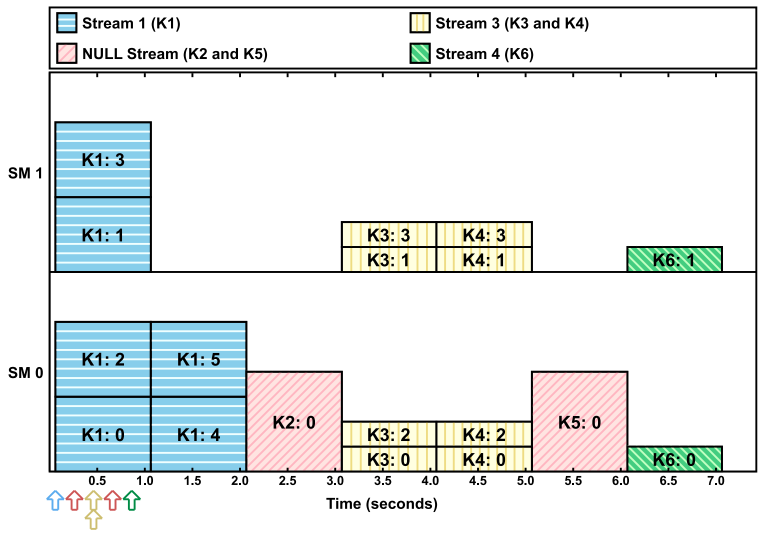


Figure 3.3: NULL stream scheduling experiment.

Experiments. In the rest of this subsection, we present the results of several experiments we conducted to evaluate the effects of using stream priorities. These experiments use the same synthetic benchmarking techniques applied in Section 3.1.2. Relevant kernel properties are given in per-experiment tables. After discussing these experiments, we postulate new scheduling rules that specify how stream priorities affect scheduling.

Scheduling of priority-low streams vs. priority-high streams. Given that blocks are schedulable entities, the handling of prioritized streams as described in the available CUDA documentation (NVIDIA, 2019b) is exactly as one might expect. In particular, stream priorities are considered each time a block finishes execution and a new block can be assigned, with blocks from priority-high streams always being favored for assignment if their resource requirements are met. Note that this assignment behavior can potentially lead to the starvation of priority-low streams.

We conducted an experiment to illustrate this behavior using the kernels defined in Table 3.3. Figure 3.4 shows the GPU timeline that resulted from this experiment. As seen, K1 was launched first in a priority-low stream and four of its eight blocks had already been assigned when K2 and K3 were later launched in two priority-high streams. When the four initially assigned blocks of K1 completed execution, freeing all SM threads, K2 (launched second) effectively preempted K1, preventing K1's remaining four blocks from being assigned. K3 was later dispatched when K2 completed, continuing the "starvation" of K1 until K3 completed.

Kernel	Launch Info	Start Time (s)	Duration (s)	# Blocks	# Thread per Block
K1	Stream S1 (low priority)	0.0	0.5	8	1,024
K2	Stream S2 (high priority)	0.2	0.5	16	1,024
K3	Stream S3 (high priority)	0.5	0.5	16	1,024

Table 3.3: Details of kernels used in the priority-stream scheduling experiment in Figure 3.4.

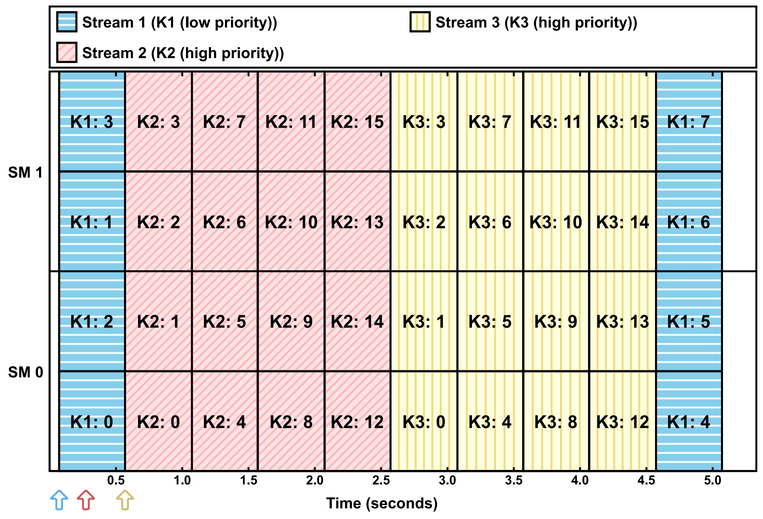


Figure 3.4: Experiment showing starvation of a priority-low stream.

Scheduling of priority-none streams vs. prioritized streams. The available CUDA documentation lacks clarity with respect to how scheduling is done when both priority-none and prioritized streams are used. We experimentally investigated this issue and found that priority-none streams have the same priority as priority-low streams on the TX2.

Evidence of this can be seen in an experiment we conducted using the kernels defined in Table 3.4. Figure 3.5 shows the resulting GPU timeline. In this experiment, K2 was launched in a priority-none stream, K1 and K4 were launched in priority-low streams, and K3 was launched in a priority-high stream. K1 was launched first and four of its eight blocks were immediately assigned to the GPU. Once these four blocks completed execution, K3 effectively preempted K1 because K3 was at the head of a priority-high stream. After K3 executed to completion, K1 resumed execution because K2 and K4 had equal priority and could not preempt it. When K1 completed, K2 and K4 were dispatched in launch-time order. If priority-none were higher than priority-low, then K2 would have preempted K1; if priority-none were lower than priority-low, then K4 would have run before K2, despite being released later.

Kernel	Launch Info	Start Time (s)	Duration (s)	# Blocks	# Thread per Block
K1	Stream S1 (low priority)	0.0	0.5	8	1,024
K2	Stream S2 (unspecified priority)	0.2	0.5	8	1,024
K3	Stream S3 (high priority)	0.3	0.5	8	1,024
K4	Stream S4 (low priority)	1.2	0.5	8	1,024

Table 3.4: Details of kernels used in the priority-stream scheduling experiment in Figure 3.5.

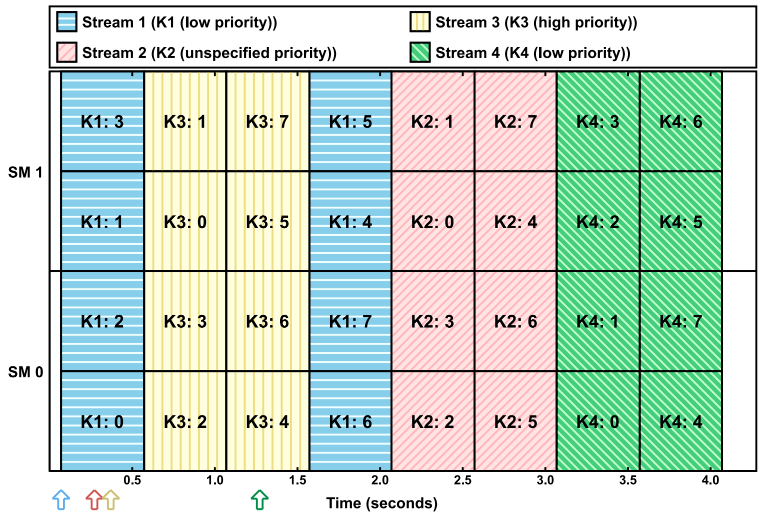


Figure 3.5: Experiment demonstrating two actual priority levels.

The scheduling of prioritized streams when resource blocking can occur. We wondered whether it was possible for kernels in priority-high streams experiencing resource blocking to be indefinitely delayed by kernels in priority-low streams that "cut ahead" and consume available resources. Our experimental results suggest this is not possible.

Evidence of this claim can be seen in an experiment we conducted using the kernels defined in Table 3.5, which yielded the resulting GPU timeline in Figure 3.6.[7] K1–K7 were launched in quick succession to seven distinct priority-low streams. Once all had been dispatched, 512 threads were available on one SM. Then K8, with one block of 1,024 threads, was launched to a priority-high stream, followed by K9, with one block of 512 threads, launched to another priority-low stream. At this time, none of the initial seven kernels had completed execution, so the priority-high K8 could not be dispatched due to a lack of available threads. When K1 ultimately completed, 512 threads were then available on each SM, but K8 still could not be dispatched

[7]This experiment generates different results with CUDA 9 and 10 on TX2. K5–K9 are serialized due to unknown reasons. However, this issue does not occur with discrete GPUs such as TITAN V.

Kernel	Launch Info	Start Time (s)	Duration (s)	# Blocks	# Thread per Block
K1	Stream S1 (low priority)	0.0	1.0	1	512
K2	Stream S2 (low priority)	0.1	1.0	1	512
K3	Stream S3 (low priority)	0.2	1.0	1	512
K4	Stream S4 (low priority)	0.3	1.0	1	512
K5	Stream S5 (low priority)	0.4	1.0	1	512
K6	Stream S6 (low priority)	0.5	1.0	1	512
K7	Stream S7 (low priority)	0.6	1.0	1	512
K8	Stream S8 (high priority)	0.65	0.5	1	1,024
K9	Stream S9 (low priority)	0.7	1.0	1	512

Table 3.5: Details of kernels used in the priority-stream scheduling experiment in Figure 3.6.

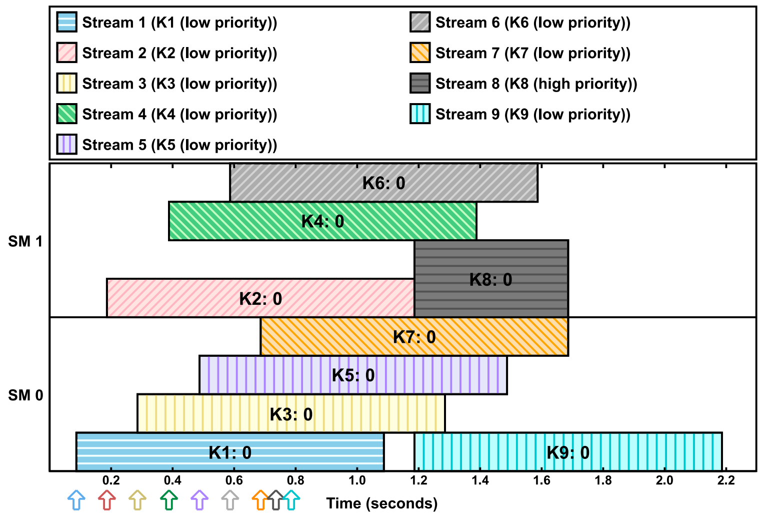


Figure 3.6: Experiment with both priorities and resource blocking.

because a block can run on only one SM. Note that even though K9 required only 512 threads, it could not be dispatched because K8 was of higher priority. Finally, when K2 completed, K8 was dispatched because 1,024 threads became available on one SM, and K9 was dispatched because 512 other threads were available and the priority-high K8 no longer blocked it.

Additional scheduling rules. Based on these experiments, we hypothesize that the TX2's GPU scheduler includes one additional EE queue, for priority-high kernels. The original EE queue described in Section 3.1.2 is for priority-low (and thus priority-none). These queues are subject to these rules:

A1 A kernel can only be enqueued on the EE queue matching the priority of its stream.

A2 A block of a kernel at the head of any EE queue is eligible to be assigned only if all higher-priority EE queues (priority-high over priority-low) are empty.

3.2.3 Multiple Processes and Other Complications

In this section, we describe a few complications that arise when CUDA programs are invoked by CPU processes (instead of tasks) that have distinct addresses spaces.

Multiprogramming on the TX2. In previous work involving the NVIDIA TX1, a less-capable predecessor of the TX2 with a similar basic layout, we found that kernels issued by multiple GPU-using processes were scheduled on the GPU using multiprogramming (Otterness *et al.*, 2017b). On the TX1, this meant that the GPU scheduler would switch between processes at the granularity of thread blocks, with kernels from different processes never actually executing at the same time. While studying the TX2, we observed that kernels issued by different processes are still multiprogrammed, but multiprogramming is implemented using GPU preemption features introduced in NVIDIA's Pascal GPU architecture.

This observation is supported by Figure 3.7, which gives the results of two experiments we ran in which two instances of a compute-heavy Mandelbrot-Set kernel were run together. In the first experiment, the two kernel instances were issued by two separate processes, while in the second, they were issued by two separate tasks. In these experiments, we recorded for each kernel the start and end times of each thread block on the GPU and then used these times to deduce the total number of threads that appeared to have been assigned to the kernel through its execution. These thread counts are plotted on the vertical axes of the four timelines in Figure 3.7. The top two timelines are from the first experiment, where processes were considered, and the bottom two timelines are from the second experiment, where tasks were considered.

The bottom two timelines indicate that, in the experiment involving tasks, scheduling was in accordance with the rules discussed in Section 3.1: all blocks from one kernel were fully dispatched before the second kernel began execution. In contrast, the top two timelines indicate that, in the experiment involving processes, the GPU scheduler's behavior is very different: 4,096 threads from each kernel appear to be fairly consistently running together at the same time. On the surface, this should appear to be impossible because the TX2 is only capable of running 4,096 threads across both of its SMs at any point in time (recall Footnote 6).

This behavior, however, is easily explained if we assume that CUDA thread blocks can be preempted. Recall that we used block-time measurements, which are recorded on the GPU itself, to calculate the number of running threads. Our GPU kernel code cannot detect being preempted itself, so the start and end times for each block can actually encompass intervals in which the block was preempted.

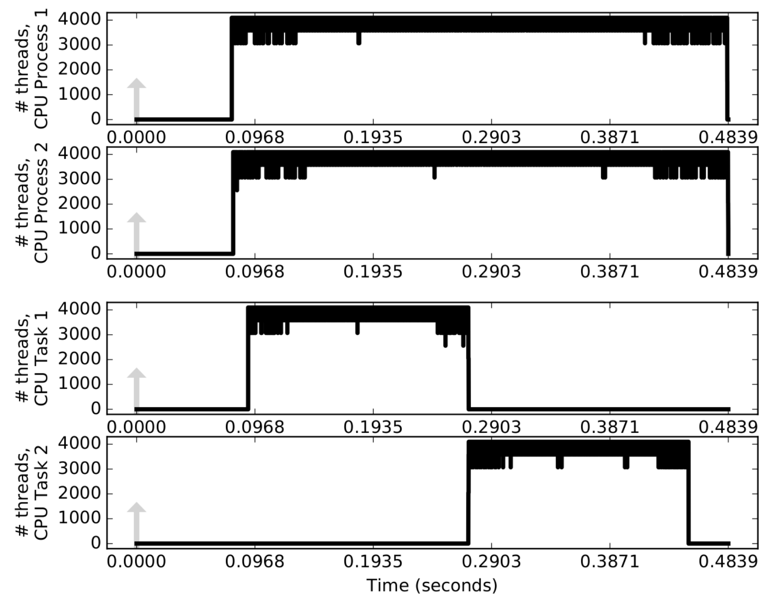


Figure 3.7: Timelines contrasting kernels issued from processes (different address spaces) vs. tasks (shared address space).

We therefore conclude that benchmarks from separate processes were being multiprogrammed on the TX2 using preemptions, as this explains the apparent over-provisioning of GPU resources. Additionally, we provide Figure 3.8 to further support the conclusion that concurrent kernels from different address spaces preempt one another.

In Figure 3.8, the curves labeled "Process 1" and "Process 2" show the cumulative distribution function (CDF) of block times for Mandelbrot-Set kernels issued concurrently from separate processes. Likewise, the "Task 1" and "Task 2" curves show the same data for kernels issued from separate tasks. Consistent with Figure 3.7, the worst-case block times for the "Process" curves is over twice that of the "Task" curves. This is in direct contrast to our prior work using the previous-generation TX1, where we found that block times remained nearly identical regardless of the number of concurrent processes (Otterness *et al.*, 2017b).

Stream priorities in multiple processes. We also experimentally examined the effects of using stream priorities when processes submit GPU operations instead of tasks. We found that assigning priorities to the streams of different processes had *no effect* on how the operations in these streams were scheduled. Thus, the usage of stream priorities has impact only in the context of task-based computing, not process-based computing.

We conclude Section 3.2 by noting several less commonly used features that may potentially impact scheduling. These include: **(i)** the nvcc compilation option, which enables per-task NULL streams (Harris,

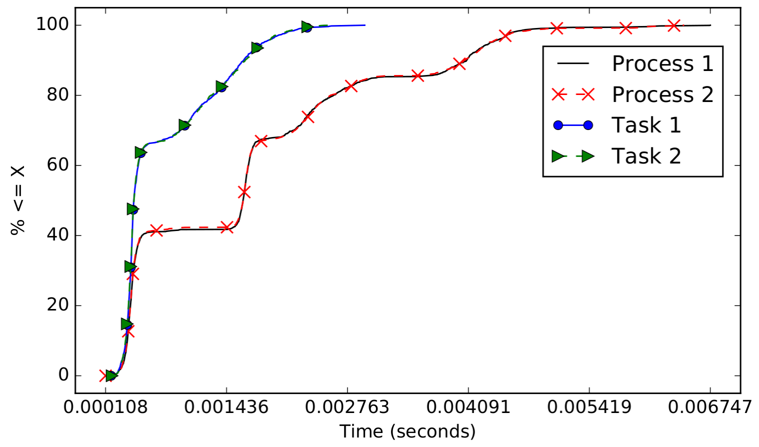


Figure 3.8: CDFs contrasting blocks times from processes vs. tasks.

2015); **(ii)** the CUDA API that enables kernels from user-specified streams to execute concurrently with the NULL stream (Harris, 2015); **(iii)** a mechanism introduced with the Pascal GPU architecture that allows an executing kernel to be preempted at the instruction level (NVIDIA, 2016b); and **(iv)** a feature called *dynamic parallelism* that allows kernels to dynamically submit extra work by calling other kernel functions inside the GPU (Adinets, 2014). We conjecture that these features are detrimental to use if real-time predictability is a requirement.

3.3 Synchronization and Blocking

CPU scheduling has been studied and well-understood for decades; in particular, real-time scheduling analysis of task systems is based on predictable scheduler and task behaviors. Incorporating GPUs into real-time analysis (as with all coprocessors), requires different models with new sets of issues to be considered. In this section, we discuss one set of issues that lead to a surprising number of pitfalls when NVIDIA GPUs are used: *synchronization*.

In Sections 3.1 and 3.2, we investigated the scheduling rules for kernels and copy operations in CUDA programs. However, this investigation focused on a limited context where few CUDA operations beyond kernel launches and memory copies were used. In most real-world CUDA software, programmers will likely encounter (both intentionally and unintentionally) the need for synchronization between CPU and GPU operations. The added complexity of synchronization can result in capacity loss, potentially leading to unbounded response times in task sets with high utilization. In this section, we explore various forms of CPU-GPU synchronization and the resulting implications for real-time systems. We limit attention for now to CPU tasks that share a single Linux address space and create user-defined streams. As covered in detail in Section 3.4, this setup allows potential concurrency among operations on the GPU.

3.3.1 Overview of GPU Synchronization

Most developers are familiar with the concepts of synchronization in a CPU-only context where two or more tasks must communicate or coordinate their actions. Synchronization becomes more complicated when a CPU task must coordinate with programs executed on the GPU. The common case is that the CPU task must determine when data in GPU memory is safe to access (*e.g.*, copy back to CPU memory). This is accomplished using *GPU synchronization*, where the GPU must complete outstanding work and reach a *synchronization point*: a point in time when data access can safely occur. There are also other, less common, cases when GPU synchronization is necessary.

In CUDA there are multiple ways to achieve GPU synchronization. They fall into two broad categories: *explicit synchronization*, which is always programmer-requested, and *implicit synchronization*, which can occur as a side effect of CUDA API functions intended for purposes other than synchronization. We have uncovered in our research some unfortunate pitfalls relating to actual GPU synchronization behavior, especially with respect to *blocking*. So, while these may not be pitfalls for non-safety-critical applications, ignoring the effects of certain specific mechanisms for achieving synchronization would be perilous in a safety-critical system where blocking must be anticipated and accounted for in analysis.

3.3.1.1 Explicit Synchronization

Explicit synchronization refers to synchronization points that the CUDA programmer explicitly requests using the CUDA API. Explicit synchronization is typically used after a program has launched one or more asynchronous CUDA kernels or memory-transfer operations and must wait for computations to complete. In contrast to implicit synchronization, the sole purpose of explicit-synchronization functions is to block the calling CPU task until the GPU reaches a synchronization point.

The CUDA documentation[8] states that explicit synchronization will block the calling task until "all preceding commands" have completed. For example, if the API function `cudaDeviceSynchronize` is invoked, "preceding commands" may encompass all commands issued to the device from all CPU tasks.

[8]Section 3.2.5.5.3 of the Programming Guide for CUDA version 10.2.89.

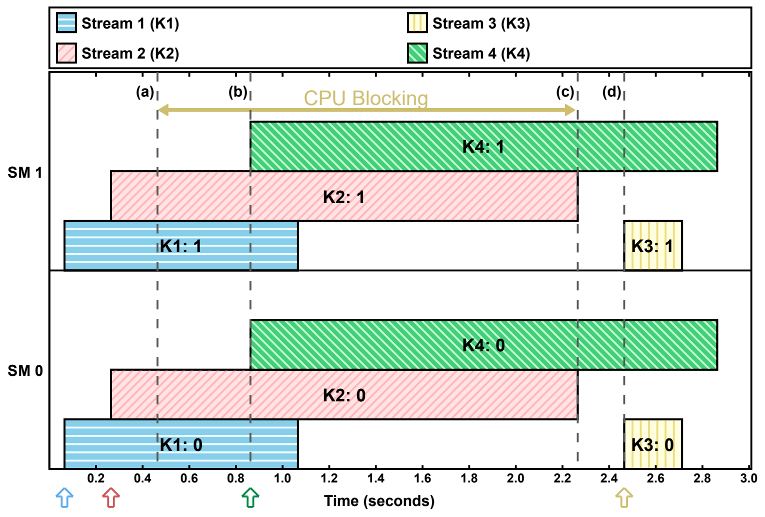


Figure 3.9: Explicit synchronization requested before K3, observed on the Jetson TX2.

Other explicit-synchronization options, including `cudaStreamSynchronize`, will only block until preceding commands from a specified stream have completed.

We carried out experiments using the same framework as in Section 3.1 to investigate the specific behaviors of GPU synchronization on the Jetson TX2. Figure 3.9 shows the behavior of explicit synchronization observed in one such experiment. The CUDA program executed to produce Figure 3.9 consists of four CPU tasks all sharing a single address space. Each CPU task launched one kernel in a separate user-defined stream. Kernel launches were separated by a small amount of time. Each kernel consisted of two blocks of 512 threads, and the figure shows that one block from each kernel was scheduled on each SM. Each thread performed a busy-loop for a set amount of time.

An explicit-synchronization command, `cudaDeviceSynchronize`, was issued at time **(a)** by the CPU task responsible for launching kernel K3. This caused K3's CPU task to be blocked until the prior commands, the execution of kernels K1 and K2, had both completed at time **(c)**. This behavior is exactly what one would expect, given the description of explicit synchronization from official documentation. However, our experiments also uncovered Pitfall 1 for the unwary:

Pitfall 1. *Explicit synchronization does not block future commands issued by other tasks.*

The fact that the launch of K4 by its CPU task was not blocked at time **(b)** is an example of this pitfall. Implicit synchronization, which we cover next, presents even more serious pitfalls.

3.3.1.2 Implicit Synchronization

Implicit synchronization occurs as a side effect of CUDA API calls that are otherwise unrelated to synchronization. For example, implicit GPU synchronization may occur due to freeing GPU memory or launching a kernel to the default stream. Presumably, this is because some modifications to GPU device state can only occur while no kernels are executing. The CUDA documentation about implicit synchronization[9] states that "two commands from different streams cannot run concurrently if any one of the following operations is issued in-between them by the host thread:

1. A page-locked host memory allocation
2. A device memory allocation
3. A device memory set
4. A memory copy between two addresses to the same device memory
5. Any CUDA command to the NULL stream."

Unlike the relatively straightforward documentation about explicit synchronization, our experiments revealed that this list includes several operations that do not necessarily cause implicit synchronization, and fails to include some functions that do. We consider this particularly problematic for real-time systems, where the ability to accurately model blocking is critical.

Pitfall 2. *Documented sources of implicit synchronization may not occur.*

Pitfall 2 became apparent to us when, in all of our experiments, we never observed implicit synchronization as a result of a device-memory operation (allocation, set, or copy) or a page-locked host memory allocation. Our experiments covered the two most recent CUDA versions, 8.0 and 9.0, and the three most recent NVIDIA GPU architectures, Maxwell, Pascal, and Volta. This, of course, does not prove that implicit synchronization can *never* happen under such circumstances, but it does indicate that the documentation's statement that "two commands cannot run concurrently" is not a reliable rule. The only case (from this list) in which we did observe implicit synchronization was launching GPU operations in the NULL stream.

Figure 3.10 shows a similar scenario to the one in Figure 3.9, with one key difference: the CPU task for K3 did not call `cudaDeviceSynchronize` before K3 was launched, but instead launched K3 in the

[9]Section 3.2.5.5.4 of the Programming Guide for CUDA version 10.2.89.

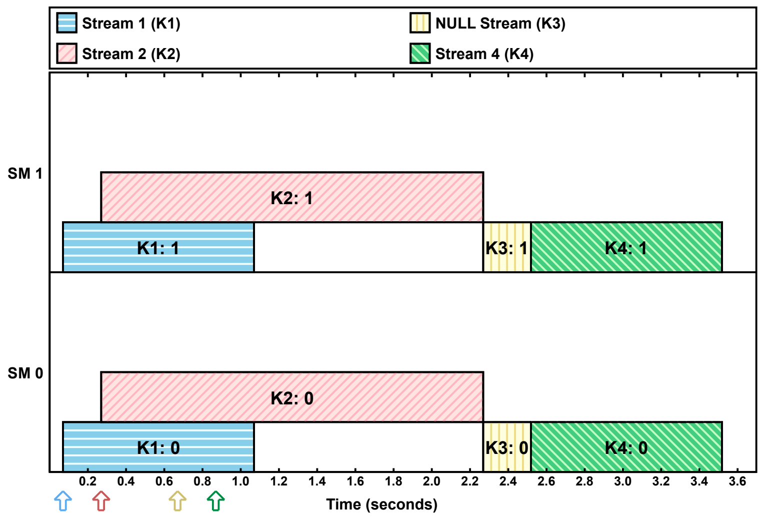


Figure 3.10: Implicit synchronization caused by launching kernel K3 in the NULL stream.

NULL stream. The implicit synchronization, and resulting loss of concurrency, is clearly visible in the figure. Execution of K3 must wait for the first two kernels to complete, and, in contrast to explicit synchronization, K4 is also prevented from running concurrently. Even though this loss of concurrency may be striking, it is notably explicitly documented, and can be used (or avoided) in a careful design for a real-time task.

We found, however, a different source of implicit synchronization that is a far more problematic pitfall, and is not even listed in the documentation on synchronization: *freeing device memory*.

Pitfall 3. *The CUDA documentation neglects to list some functions that cause implicit synchronization.*

Pitfall 4. *Some CUDA API functions will block future, unrelated, CUDA tasks **on the CPU**.*

Figure 3.11 shows the results of an experiment identical to the one in Figure 3.9, but this time the call to `cudaDeviceSynchronize` at time **(a)** was replaced with a call to `cudaFree`, which was used to de-allocate memory on the GPU. Pitfalls 3 and 4 can be observed in this plot. The fact that this blocked the calling CPU thread until all prior GPU work had completed at time **(c)** indicates that `cudaFree` created implicit synchronization. Similar to the NULL-stream behavior, implicit synchronization also prevented subsequent kernels from starting to execute until `cudaFree` completed at time **(c)**. We speculate that this behavior by `cudaFree` is necessary because alterations to memory-mapping state require a quiescent execution environment. However, the most surprising effect was not that K4 was blocked, but that K4's task was blocked *on the CPU* until time **(c)**, even though it issued an "asynchronous" kernel launch. This

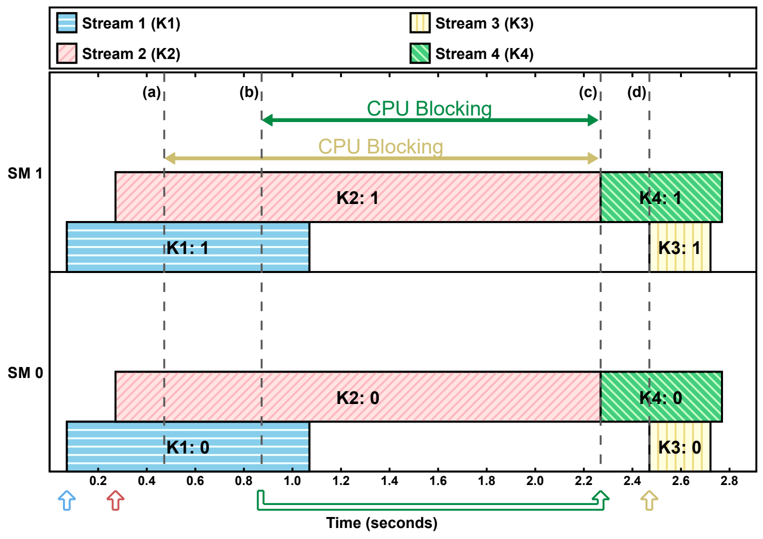


Figure 3.11: Implicit synchronization causing additional CPU blocking due to `cudaFree`.

reveals a pitfall that can harm real-time analysis that does not consider the fact that CPU tasks can experience blocking from GPU operations that are launched from unrelated tasks.

3.3.2 Overcoming Synchronization-Related Pitfalls

GPU synchronization has two problematic effects—introducing indeterminate amounts of blocking and reducing GPU concurrency. This means that programmers who develop real-time systems must understand the pitfalls inherent in explicit and implicit synchronization. This is especially true if the schedulability of a real-time task system relies on minimizing blocking or high GPU utilization. Avoiding pitfalls can be accomplished through careful construction of CUDA programs to, for example, avoid using the NULL stream or freeing memory outside of certain time intervals. A more robust method would be to adopt middleware that handles such problems transparently.

Our experiments indicate that GPU synchronization does not extend across GPU-using tasks that are isolated in separate address spaces. If synchronization is the dominant limiting factor on schedulability, it may be desirable to place each task in a separate address space (OS process). As explained in the next section, this organization means that that CUDA kernels from different tasks can no longer execute concurrently, but it may still be beneficial overall if synchronization-related blocking is a greater limiting factor.

It turns out that NVIDIA may be aware of this issue. Even though it is not currently available for embedded platforms such as the TX2, NVIDIA does provide useful middleware for discrete GPUs: the *CUDA Multi-Process Service* (MPS). MPS allows kernels from multiple processes to execute concurrently on a single GPU, while maintaining the desirable property that GPU synchronization from one process will not affect other processes. We explore the benefits of MPS further in Section 3.4.

3.4 Concurrency and Performance

In Section 3.1 and Section 3.2, we investigated different GPU scheduling behavior when running GPU-using real-time task systems in two contexts: **(i)** each task has its own distinct address space, *i.e.*, it runs as an OS process, and **(ii)** all tasks belong to the same address space, *i.e.*, each task runs as a schedulable thread within a process. We refer to these two contexts as *process-based* and *thread-based* tasks, respectively.

While process-based tasks have the advantage of memory protection, they do not actually execute on the GPU concurrently; instead, GPU operations are multiprogrammed in a way that makes predictable scheduling of GPU-related resources difficult if not impossible to achieve (Section 3.2). When operations are multiprogrammed on a GPU, their execution times depend on contention for shared GPU resources, making it hard to bound a task's overall execution time. Additionally, concurrency among GPU operations may be important in order to avoid wasting GPU processing cycles, especially when a single kernel cannot fully utilize the GPU's resources. Although this may be avoided by running tasks with user-defined streams in a shared address space, a shared address space may actually *reduce* concurrency in task systems where tasks regularly interfere with each other via implicit synchronization (Section 3.3). Fortunately, NVIDIA provides a third option: middleware called the *Multi-Process Service* (*MPS*) (NVIDIA, 2019e).

3.4.1 Multi-Process Service (MPS)

MPS enables concurrent execution of GPU operations launched by independent CPU address spaces. It has the potential to combine the advantages of both thread- and process-based tasks. Programs written using the CUDA API require no changes to use MPS—if MPS is running, CUDA programs transparently issue requests to MPS rather than directly to a GPU. Official documentation reports that MPS operates as a server process with its own CUDA context, and that CUDA API requests are redirected from client processes to the MPS server. Because the server's CUDA context is effectively shared, GPU operations launched by separate processes can execute concurrently on a shared GPU, providing the benefits of thread-based tasks. However, MPS also continues to preserve the advantage of process-based tasks: separate processes will not block each other with implicit or explicit synchronization.

It is not clear from available documentation how MPS actually schedules GPU operations and whether the GPU scheduling rules revealed in Section 3.1 are followed under MPS. For example, the documentation for MPS only mentions possible overlap between kernels and copy operations.[10] Given the documentation flaws discussed in Sections 3.3 and 3.5.2, one could be skeptical of the veracity of this claim, so we verified experimentally that those scheduling rules are also followed under MPS following the experimental methods in Section 3.1.

Maximizing the utilization of GPU resources using streams in thread-based tasks is suggested by NVIDIA's "Best Practices Guide" (NVIDIA, 2019a). However, it would be unwise to simply take this recommendation at face value when choosing between MPS or a process- or thread-based task organization in a safety-critical system. Additionally, ***MPS is not yet supported on embedded ARM platforms*** like the Jetson TX2, so the other management systems are still necessary on some systems. Therefore, we conducted a case study on computer-vision software, demonstrating the performance differences among the available configurations.

3.4.2 Case Study of Computer-Vision Tasks

Our motivation is primarily autonomous driving, so we chose to study algorithms for computer-vision tasks that provide functions commonly used for autonomous driving. In evaluating the results from this case study, we consider that the real-time tasks that use GPUs for autonomous driving may have multiple levels of criticality. Some may be safety-critical with hard deadlines and be provisioned for worst-case execution plus a margin for safety. Others may have only bounded tardiness requirements, or even be background work that can be provisioned for average-case execution.

We focus here on five programs from NVIDIA's provided sample code for VisionWorks:

- **Video Stabilization.** Smooths shaky video content. This is often a preprocessing step for a computer-vision pipeline.
- **Feature Tracking.** Tracks features between consecutive frames. This algorithm is used to track the positions of objects in a scene.

[10] "MPS allows kernel and memory copy operations from different processes to overlap on the GPU, achieving higher utilization and shorter running times" (NVIDIA, 2019e).

	Multiple Process-based Tasks	Multiple Thread-based Tasks
Without MPS	MP	MT
With MPS	MP(MPS)	MT(MPS)

Table 3.6: Abbreviations used for our four experimental scenarios.

- **Motion Estimation.** Estimates the direction of moving pixels, which is fundamental to calculating trajectories of moving objects, *e.g.*, pedestrians and other vehicles.
- **Hough Transform.** (*Hough*) A feature-extraction algorithm; the provided sample detects circles and lines in images.
- **Stereo Matching.** Uses input from two cameras to generate depth information by matching features in both frames.

Methodology. We adapted NVIDIA's VisionWorks samples to be compatible with our open-source experimental framework.[11] These samples generally only use a single CUDA stream. We ran four instances of the same sample program in each experiment. We configured each instance to process 1,000 frames from a video sequence while recording per-frame response times. Our framework allows running each program instance in a shared address space (multiple thread-based tasks, MT) or in independent address spaces (multiple process-based tasks, MP), both with and without the MPS server active. This produces experiments for each algorithm in four different scenarios as summarized in Table 3.6. Experimental results under MT(MPS) were always similar to MT with slight overheads caused by MPS, so we omit it in all of our results for clarity. We conducted these experiments on a Maxwell-architecture discrete GPU with CUDA 9.0. We briefly summarize results on other devices and different CUDA versions later.

Results. We show cumulative distribution function (CDF) and kernel density estimation (KDE)[12] plots of Hough and feature tracker as representatives in Figures 3.12–3.15. The KDE curve was produced using the `Python` package `scipy.stats.gaussian_kde`. In both the CDF and KDE plots, each curve represents the recorded response-time data in an experimental scenario. For example, the curves labeled "x4 MP" in Figures 3.12 and 3.13 represent the per-frame response time distributions where each of four Hough instances is run in a separate process. Result data for all five algorithms is summarized in Table 3.7, which lists the maximum, 99th-percentile, 90th-percentile, mean, and median frame times for each scenario and algorithm.

Observation 3.1.

[11] Again, `https://github.com/yalue/cuda_scheduling_examiner_mirror`.

[12] KDE is a statistical method for estimating a continuous probability density function (PDF) from a set of discrete sample values.

VisionWorks Samples	Scenarios	Max	$99^{th}\%$	$90^{th}\%$	Mean	Median
Video Stabilization	MP	17.55	12.88	5.43	3.31	2.69
	MP (MPS)	36.73	**11.12**	**5.37**	**2.81**	**2.06**
	MT	**17.0**	13.87	8.94	4.72	3.63
Feature Tracking	MP	**5.64**	**3.87**	**1.45**	**1.08**	**0.96**
	MP (MPS)	14.73	6.04	1.51	1.31	1.09
	MT	31.11	20.86	11.51	4.68	2.68
Motion Estimation	MP	**28.64**	**21.25**	17.33	16.75	17.24
	MP (MPS)	33.05	22.66	**15.75**	**14.3**	**14.89**
	MT	42.86	26.12	16.53	15.07	15.14
Hough Transform	MP	**13.56**	**11.61**	7.28	5.68	5.7
	MP (MPS)	18.35	11.66	**6.44**	**3.74**	**3.18**
	MT	58.65	22.64	15.82	9.12	8.94
Stereo Matching	MP	75.13	50.54	30.42	24.14	24.77
	MP (MPS)	**59.73**	**45.05**	**26.87**	22.59	24.41
	MT	125.96	58.82	34.36	**20.75**	**18.95**

Table 3.7: Per-frame response time data (in milliseconds) of VisionWorks samples. The fastest scenario for each time metric is indicated by bold text.

MP(MPS) exhibits good average-case performance.

Observation 3.1 is supported by the data in Table 3.7. 90^{th}-percentile, mean, and median performance under configuration MP(MPS) were consistently good with the top performance for three of the five algorithms. For Feature Tracking, MP was best in all metrics, and for Stereo Matching, MT had better mean and median performance. The results for average-case performance indicate that using MP(MPS) would likely be an attractive option for soft-real-time systems, *e.g.*, systems that can occasionally drop a video frame without compromising safety. We conjecture that the average-case performance advantage of MP(MPS) over MP in most cases is due to improved concurrency and lower GPU context-switching overheads.

Feature Tracking was the most notable exception to Observation 3.1. In this case, MP was only slightly better than MP(MPS) when comparing the 90^{th}-percentile, mean, and median performance. We conducted additional experiments using NVIDIA's CUDA-profiling tool, `nvprof`, to gain some insight into this behavior. We found that Feature Tracking's overall execution time is heavily influenced by a large number of memory transfers, rather than CUDA kernel executions. This likely means that MPS only provides limited GPU concurrency benefits to Feature Tracking, which failed to outweigh other MPS-related overheads.

Observation 3.2. Worst-case and 99^{th}-percentile runtimes were typically better under MP.

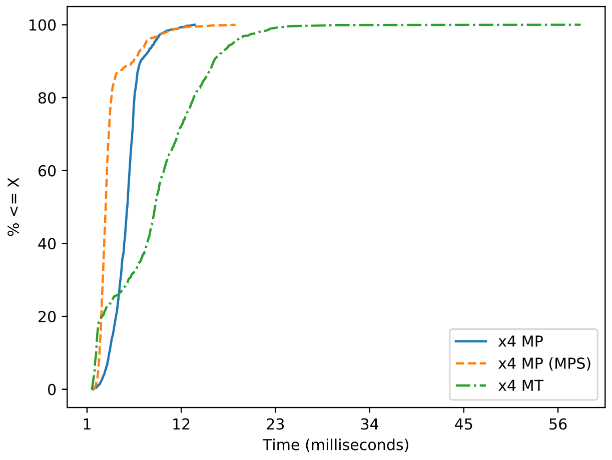


Figure 3.12: Per-frame response time CDFs for Hough.

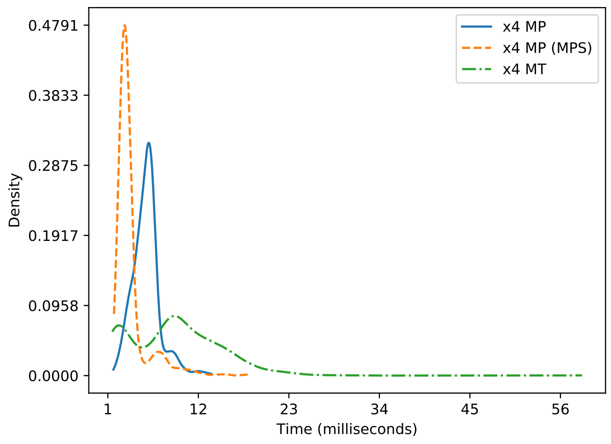


Figure 3.13: Per-frame response time KDEs for Hough.

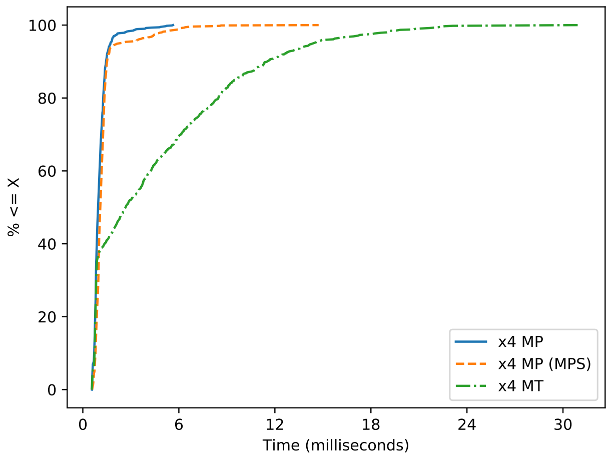


Figure 3.14: Per-frame response time CDFs for Feature Tracking.

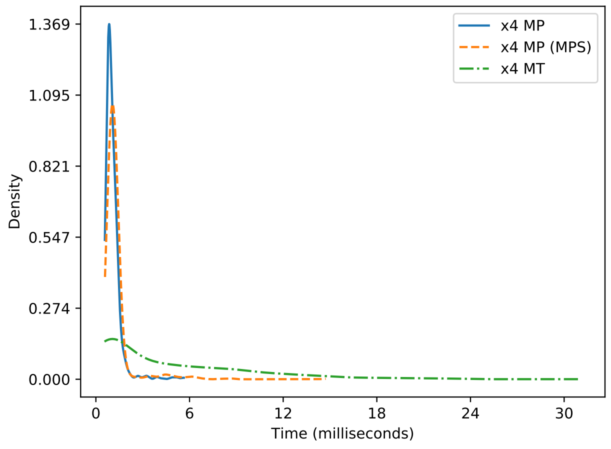


Figure 3.15: Per-frame response time KDEs for Feature Tracking.

While MP(MPS) largely resulted in average-case improvements, Table 3.7 shows three of our five applications (Feature Tracking, Motion Estimation, and Hough Transform) showed the smallest worst-case and 99^{th}-percentile execution times under MP. This indicates that MP may be a better option for certain task systems where worst-case performance is more important than average-case. Our results illustrate why the trade-offs between process-based and thread-based designs for tasks must be evaluated for individual algorithms.

Observation 3.3. MP and MP(MPS) exhibit more predictable execution times than MT.

Observation 3.3 is supported by Figures 3.13 and 3.15, where the KDE shows a tight unimodal distribution for MP and MP(MPS) but not MT. A unimodal distribution function with little dispersion indicates that the response times exhibit low variance. MT, in contrast, shows both bimodal (in Figure 3.13) and unimodal (in

Figure 3.15) distributions with significant dispersions (indicating high variance). Even if specific "spikes" are more difficult to observe in the corresponding CDF plots, the difference in response-time ranges are also apparent from the endpoints of the CDF curves in Figures 3.12 and 3.14.

Observation 3.4. The MT configuration generally performed poorly.

Observation 3.4 is supported by Table 3.7 and the plots. The only metrics where MT outperformed the other scenarios were the mean and median times for Stereo Matching, and worst-case response time for Video Stabilization (where MT was only slightly better than MP).

Other results. In addition to the results presented above, we also conducted this case study using CUDA 8.0 on a Maxwell discrete GPU (GTX 860M) and CUDA 9.0 on Pascal discrete GPUs (GTX 1050 and GTX 1070). Even though we chose to omit tables of results from the other GPUs and CUDA versions, we made similar observations excepting that the performance of all configurations was better on a Pascal GPU. Additionally, the experimental results with CUDA 8.0 on the same Maxwell GPU stayed nearly identical to those using CUDA 9.0.

Pitfall 5. *The suggestion from NVIDIA's documentation to exploit concurrency through user-defined streams may be of limited use for improving performance in thread-based tasks.*

We assumed that enabling concurrent GPU execution was of significant importance for limiting capacity loss in real-time workloads on embedded systems, and therefore fell victim to Pitfall 5. Instead, our results show that MT rarely outperforms tasks running as multiple processes, even without MPS. Additionally, any performance improvement via fine-tuned stream organization for MT can also be achieved with MP(MPS). That being said, even though enabling concurrency using MP(MPS) is generally beneficial, it unfortunately is not an option on ARM-based embedded platforms like the Jetson TX2. We would encourage NVIDIA to consider this shortcoming in hope that one day it may be addressed.

3.5 Perils of CUDA Programming for Real-Time Tasks

In the previous sections we presented several specific pitfalls in correctly designing and running CUDA programs for real-time tasks. Elements of both CUDA's design and documentation contribute to this ensemble of perils to avoid. In this section, we discuss some of the broader categories of pitfalls.

3.5.1 Synchronous Defaults

As hinted in Section 3.3, one of the primary pitfalls when designing a real-time task system that uses a GPU is that *all possible* blocking must be accounted for in analysis. Therefore, reducing the amount of blocking on both the CPU and GPU is essential. On the GPU, this requires issuing all CUDA operations to user-defined (non-NULL) streams, and carefully controlling the use of other API functions, like `cudaFree`, that cause blocking via implicit synchronization.

Even though it may seem like an easy task for a programmer to just specify a user-defined stream as opposed to the NULL stream, we note that simple mistakes in doing so may be easy to miss. This is particularly true when using the `Async` versions of CUDA API functions, such as `cudaMemsetAsync`. For example, consider the code snippets in Listings 3.1 and 3.2, which present a particular example of Pitfall 6 below.

Listing 3.1: Causes implicit synchronization.

```
if (!CheckCUDAError(cudaMemsetAsync(
  state->device_block_smids, 0,
  data_size))) {
  return 0;
}
```

Listing 3.2: Correctly asynchronous.

```
if (!CheckCUDAError(cudaMemsetAsync(
  state->device_block_smids, 0,
  data_size, state->stream))) {
  return 0;
}
```

Pitfall 6. *`Async` CUDA functions use the GPU-synchronous NULL stream by default.*

Listing 3.1's call to `cudaMemsetAsync` is missing a final argument specifying a user-defined stream, which causes the NULL stream to be used by default. As pointed out in Section 3.3.1.2, NULL-stream usage causes implicit synchronization and hence blocking. This mistake is corrected in Listing 3.2. This specific mistake actually led to *months* of mystifyingly inconsistent results in our own experiments—despite our relatively deep experience examining the subtleties of CUDA behavior (note that these code snippets are parts of much larger listings). *Would an ML application developer catch such a mistake or appreciate its impact?* Note that NVIDIA's CUDA compiler does not catch this mistake because the compiler is based on the C++ programming language, which allows default arguments to functions.

Even though the examples in Listings 3.1 and 3.2 only use `cudaMemsetAsync`, Pitfall 6 applies to other CUDA API functions as well, such as `cudaMemcpyAsync`. The fact that the CUDA documentation indicates that these functions cause implicit synchronization, as discussed in Section 3.3 and Section 3.5.2, makes potential programmer errors even harder to notice in cases where synchronization is due to NULL-stream usage rather than memory operations.

To summarize this discussion, CUDA provides a brittle programming environment: difficult-to-spot mistakes can have profound consequences for real-time tasks.

3.5.2 Flawed Documentation

Another substantial danger stems from the inaccurate official documentation provided by NVIDIA. While function signatures and data structures seem to receive accurate (but often sparse) official documentation, scheduling and synchronization remain under-discussed. We have demystified some scheduling rules (Section 3.1). In our work to demystify implicit synchronization (see definition in Section 3.3.1.2), however, we came across not only missing documentation, but incorrect documentation.

Pitfall 7. *Observed CUDA behavior often diverges from what the documentation states or implies.*

Consider Table 3.8. In all but one of the cases we investigated, the documentation claims implicit synchronization will occur when it does not. While this absence of synchronization may positively benefit performance, it also may cause incorrect timing analysis. Furthermore, program logic may be broken in the (albeit unlikely) case that the program relies on a function like `cudaMemsetAsync` to trigger GPU synchronization.

Unfortunately, the documentation also contains less-benign flaws. Take `cudaFree` and `cudaFreeHost` as an example. Our experiments in Section 3.3 found these functions to not only cause implicit synchronization, but block other CPU tasks from proceeding while `cudaFree` waits on the GPU. Much to our surprise, the documentation mentions neither of these side effects, leaving the reader to assume that these functions behave similarly to other CUDA functions and have no side effects.

Our experiments also revealed that `cudaMalloc` and `cudaMallocHost` may also cause cross-task CPU blocking in a similar manner to `cudaFree` in certain situations, even though these functions do not trigger implicit synchronization. As we have not yet determined the specific causes for this behavior, this property is indicated by an entry of '?' in certain cells in Table 3.8. In any case, we failed to find any mention of this variant of CPU blocking in the CUDA documentation, and investigating these functions remains an open topic that we plan to explore in future work.

An especially worrying pitfall is the following:

	Observed Behavior			Documented Behavior	
Source	Blocks Other CPU Tasks	Implicit Sync. (Section 3.3.1.2)	Caller Must Wait for GPU	Implicit Sync. (Section 3.3.1.2)	Caller Must Wait for GPU
`cudaDeviceSynchronize`	No	No	Yes	No	Yes
`cudaFree`	Yes	Yes	Yes	**No** (undoc.)	**No** (impl.)
`cudaFreeHost`	Yes	Yes	Yes	**No** (undoc.)	**No** (impl.)
`cudaMalloc`	?	No	No	**Yes**	No (impl.)
`cudaMallocHost`	?	No	No	**Yes**	No (impl.)
`cudaMemcpyAsync D-D`	No	No	No	**Yes**	No
`cudaMemcpyAsync D-H`	No	No	No	**Yes**[*]	No
`cudaMemcpyAsync H-D`	No	No	No	**Yes**[*]	No
`cudaMemset (sync.)`	No	Yes	No	Yes	No
`cudaMemsetAsync`	No	No	No	**Yes**	No
`cudaStreamSynchronize`	No	No	Yes	No	Yes

Table 3.8: Observed vs. documented synchronization sources in CUDA. For `cudaMemcpyAsync` we distinguish the direction of copy between device and host: (D-D) internal to GPU memory; (D-H) GPU memory to CPU memory; (H-D) CPU memory to GPU memory. [*]The documentation is contradictory for these instances, but the more detailed option indicates that these functions only cause synchronization if host memory is not page-locked. We were unable to observe this regardless of whether host memory was page-locked or not.

Pitfall 8. *CUDA documentation can be contradictory.*

In one case, namely `cudaMemcpyAsync`, we discovered that the CUDA documentation actively contradicts itself. Section 3.2.5.1 of the CUDA Programming Guide states "The following device operations are asynchronous with respect to the host: ... Memory copies performed by functions that are suffixed with `Async`," but Section 2 of the CUDA Runtime API documentation states "For transfers from device memory to pageable host memory, [`cudaMemcpyAsync`] will return only once the copy has completed." This raises further doubts about the correctness of other parts of the CUDA documentation.

We note that the CUDA API contains 146 non-deprecated or compatibility-related functions, and we have only tested a small fraction of these in depth. Therefore, it is likely that our findings with Pitfalls 7 and 8 apply to other portions of the documentation that we have yet to observe.

3.5.3 Unknown Future

All of the pitfalls discussed in this section, as well as the need to compare the alternatives considered in Section 3.4 empirically, can be attributed to a single overarching problem: the black-box nature of current GPU-enabled platforms means that *developers do not have a reliable model of GPU behavior*. Much of our group's prior work has focused on developing such a model. However, this highlights what is perhaps the most important pitfall:

Pitfall 9. *What we learn about current black-box GPUs may not apply in the future.*

Despite the fact that we validated our experimental results on several of the most recent CUDA versions and GPU architectures, there is no guarantee that our results will hold after future GPU-architecture or CUDA-version updates.

Even though other safety-critical hardware inevitably undergoes changes and updates, future-proof programs can still be developed against a stable specification. Likewise, the only way to truly mitigate Pitfall 9 is for GPU manufacturers to release stable, accurate documentation about their GPU platforms, along, preferably, with giving developers greater control over GPU scheduling and synchronization. Only then can we have a reliable GPU model upon which to base real-time analysis and certification. We hope that work such as ours signals to manufacturers like NVIDIA that greater openness is a desirable feature when marketing in safety-critical domains.

Unfortunately, there is little indication that NVIDIA plans to move towards open hardware or software in the immediate future. In the meantime, one of our continuing objectives is to produce tools, such as our experimental framework, that can be quickly adapted to new GPU hardware. So far, our tools have allowed us to quickly re-validate our prior results every time NVIDIA updates its black-box hardware or software.

3.6 Chapter Summary

We presented an in-depth study of GPU scheduling behavior on the NVIDIA TX2, an exemplar of platforms marketed today for supporting autonomous systems. We found that the GPU scheduler of the TX2 has predictable FIFO-oriented properties that are amenable to real-time schedulability analysis if all work is submitted by CPU tasks that share an address space or by CPU processes that are managed by MPS. On the other hand, GPU scheduling on the TX2 becomes more unpredictable and complex when GPU computations are launched by CPU processes that have distinct address spaces without using MPS. Unfortunately, this process-oriented approach is common in GPU program development.

We acknowledge that it is not possible to confirm with certainty the scheduling behavior of any GPU through only black-box experimentation. However, we counter this drawback by noting that plans are already underway to use these devices to realize safety-critical autonomous capabilities (NVIDIA, 2017b). If NVIDIA is unwilling to release details about internal GPU scheduling policies, then the only option available to us is to attempt to discern such policies through experimentation. Moreover, when mass-market vehicles eventually evolve to become fully autonomous, certification will become a crucial concern. It is simply not possible to certify with any degree of certainty a safety-critical system built using components that have unknown behaviors.

In the next chapter, we will derive response-time bounds for the GPU scheduler specified by the rules in Sections 3.1 and 3.2.

In addition to the revealed GPU scheduling rules, this chapter contributes a list of potential pitfalls when developing CUDA applications for real-time systems. Reasons for these pitfalls include GPU synchronization, application performance, and problems with documentation. We uncovered these pitfalls via microbenchmark experiments, examining the performance of real-world computer-vision applications, and a careful reading of official GPU documentation. While there is no guarantee of stability in our observations as NVIDIA's hardware and software continues to evolve, we hope that our open-source experimental system will at least make it apparent when changes do occur.

The experimental methodology employed in this chapter is of a general nature and can be applied to other NVIDIA GPUs.

Acknowledgement. The contents of this chapter previously appeared in a series of papers that Tanya Amert, Nathan Otterness, and I collaboratively produced. Nathan implemented the CUDA scheduling examiner framework for the experiments. Tanya implemented the visualization scripts that plot the scheduling timeline figures (*e.g.*, Figure 3.1). I ported the computer-vision sample code from NVIDIA CUDA samples to the aforementioned framework for the evaluation presented in Section 3.4.

For revealing GPU scheduling rules, I designed the initial set of experiments in (Otterness *et al.*, 2017a), which Tanya redesigned in (Amert *et al.*, 2017), writing the current version that I modified to correct the scheduling rules for copy operations, in Section 3.1. The experiments for additional complexities—NULL stream (Tanya), process-based GPU access (Nathan), and stream priorities (me), are included in Section 3.2. Nathan and I conducted the experiments in Section 3.3, and I conducted the experiments in Section 3.4. Joshua Bakita and Nathan summarized the list of perils in Section 3.5.

CHAPTER 4: TASK MODEL AND SCHEDULABILITY ANALYSIS FOR TASKS SHARING GPUS[1]

Previous chapters established a fundamental understanding of NVIDIA GPUs, allowing us to conduct abstract analysis and derive real-time guarantees for systems that share NVIDIA GPUs. In this chapter, we introduce a task model and a schedulability analysis for GPU tasks. We analyze GPU applications using a DAG model, in which each node of a DAG contains either CPU or GPU work. We will first review how prior work of DAG scheduling analysis can be applied in Section 4.1. Then, based on the CUDA programming model and GPU background provided in Section 2.3, we introduce a task model to describe the GPU tasks.

Using this task model, we abstractly analyze the NVIDIA GPU scheduler specified by the scheduling rules revealed in Section 3.1. We show that *intra-task parallelism* is necessary to prevent extreme capacity loss. Then we demonstrate that NVIDIA GPUs inherently suffer from a fundamental capacity loss (even with intra-task parallelism), which we express by providing a *total utilization bound*. We show that this bound is tight with a counterexample, a task system that has total utilization matching this bound but unbounded response times. This total utilization bound forms the schedulability condition, under which we present a response-time bound analysis for GPU tasks that are allowed to execute concurrently on NVIDIA GPUs.

4.1 DAG-based Task Model

In this section, we review prior relevant work on the real-time scheduling of DAGs. We specifically consider a system $G = \{G^1, G^2, \dots, G^N\}$ comprised of N DAGs. The DAG G^i consists of n^i nodes, which correspond to n^i sequential tasks $\tau^i_1, \tau^i_2, \dots, \tau^i_{n^i}$. Each task τ^i_v releases a (potentially infinite) sequence of *jobs* $\tau^i_{v,1}, \tau^i_{v,2}, \dots$. The edges in G^i reflect precedence relationships. A particular task τ^i_v's *predecessors* are those tasks with outgoing edges directed to τ^i_v, and its *successors* are those with incoming edges directed from τ^i_v.

[1]Contents of this chapter previously appeared in the following papers:

Yang, K., Yang, M., and Anderson, J. (2016a). Reducing response-time bounds for DAG-based task systems on heterogeneous multicore platforms. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pages 349–358.

Yang, M., Amert, T., Yang, K., Otterness, N., Anderson, J. H., Smith, F. D., and Wang, S. (2018a). Making OpenVX really "real time". In *Proceedings of the 39th IEEE International Real-Time Systems Symposium*, pages 80–93.

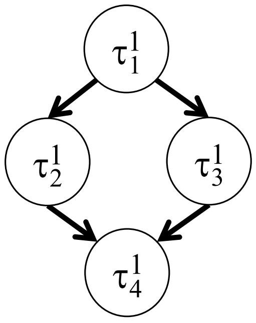


Figure 4.1: DAG G^1.

The jth job of task τ_v^i, $\tau_{v,j}^i$, cannot commence execution until the jth jobs of all of its predecessors finish. Such dependencies only exist for the *same invocation* of a DAG, not across invocations. That is, while jobs are sequential, *intra-task parallelism* (*i.e.*, parallel node invocation) is possible: *successive jobs of a task are allowed to execute in parallel.*

Example 4.1. Consider DAG G^1 in Figure 4.1. Task τ_4^1's predecessors are tasks τ_2^1 and τ_3^1, *i.e.*, for any j, job $\tau_{4,j}^1$ waits for jobs $\tau_{2,j}^1$ and $\tau_{3,j}^1$ to finish. If intra-task parallelism is allowed, then $\tau_{4,j}^1$ and $\tau_{4,j+1}^1$ could execute in parallel. $\Diamond$

For simplicity, we assume that each DAG G^i has exactly one *source task* τ_1^i, with only outgoing edges, and one *sink task* $\tau_{n^i}^i$, with only incoming edges. Multi-source/multi-sink DAGs can be supported with the addition of singular "virtual" sources and sinks that connect multiple sources and sinks, respectively. Virtual sources and sinks have a WCET of zero.

Source tasks are released sporadically, *i.e.*, for the DAG G^i, the job releases of τ_1^i have a minimum separation time, or *period*, denoted T^i. A non-source task τ_v^i $(v > 1)$ releases its jth job $\tau_{v,j}^i$ after the jth jobs of all its predecessors in G^i have completed. That is, letting $r_{v,j}^i$ and $f_{v,j}^i$ denote the release and finish times of $\tau_{v,j}^i$, respectively, $r_{v,j}^i \geq \max\{f_{w,j}^i \mid \tau_w^i \text{ is a predecessor of } \tau_v^i\}$. The *response time* of job $\tau_{v,j}^i$ is defined as $f_{v,j}^i - r_{v,j}^i$, and the *end-to-end response time* of the jth invocation of the DAG G^i as $f_{n^i,j}^i - r_{1,j}^i$.

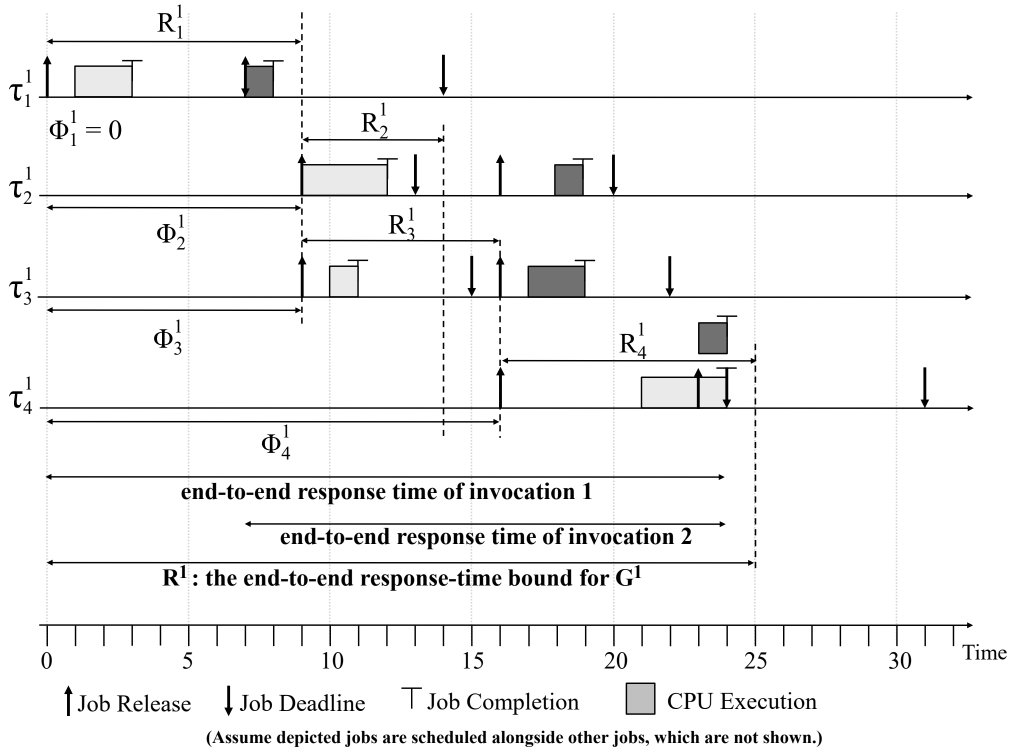
Deriving response-time bounds. An end-to-end response-time bound, the sum of response-time bounds of the nodes on the critical path of a DAG, can be computed inductively for a DAG by scheduling its nodes in a way that allows them to be viewed as sporadic tasks and by then leveraging response-time bounds applicable to such tasks (Liu and Anderson, 2010). In considering the scheduling of DAGs on a heterogeneous hardware platform that consists of different types of devices, response-time bounds for nodes running on each type of device can be derived separately. However, for the CPU-GPU platform we consider, while we have mentioned such analysis for CPU nodes, a new analysis for GPU nodes is needed.

When viewing nodes of a DAG G^i as sporadic tasks, precedence constraints must be respected. This can be ensured by assigning an *offset* Φ_v^i to each task τ_v^i based on the response-time bounds applicable to "up-stream" tasks in G^i, and by requiring the j^{th} job of τ_v^i to be released exactly Φ_v^i time units after the release time of the j^{th} job of the source task τ_1^i, *i.e.*, $r_{v,j}^i = r_{1,j}^i + \Phi_v^i$, where $\Phi_1^i = 0$. With offsets so defined, every task τ_v^i in G^i (not just the source) has a period of T_i. Also, letting C_v^i denote the WCET of τ_v^i, its *utilization* can be defined as $u_v^i = C_v^i / T^i$.

Example 4.1 (cont'd). Figure 4.2 depicts an example schedule for the DAG G^1 in Figure 4.1. The first (respectively, second) job of each task has a lighter (respectively, darker) shading to make them easier to distinguish. Assume that the tasks have deadlines as shown, and response-time bounds of $R_1^1 = 9$, $R_2^1 = 5$, $R_3^1 = 7$, and $R_4^1 = 9$, respectively. Based on these bounds, we define corresponding offsets $\Phi_1^1 = 0$, $\Phi_2^1 = 9$, $\Phi_3^1 = 9$, and $\Phi_4^1 = 16$, respectively. With these response-time bounds, the end-to-end response-time bound that can be guaranteed is determined by R_1^1, R_3^1, and R_4^1 and is given by $R^1 = 25$. The task response-time bounds used here depend on the scheduler employed. For example, if all tasks are scheduled via the G-EDF scheduler, then per-task response-time bounds can be determined from tardiness analysis for G-EDF (Devi and Anderson, 2008; Erickson and Anderson, 2011). In fact, this statement applies to any G-EDF-like (GEL) scheduler (Erickson *et al.*, 2014).[2] Such schedulers will be our focus. Recall that, according to the DAG-based task model introduced here, successive jobs of the same task might execute in parallel. We see this with jobs $\tau_{4,1}^1$ and $\tau_{4,2}^1$ in the interval $[23, 24)$. Such jobs could even finish out of release-time order due to execution-time variations. $\Diamond$

Early releasing. Using offsets may cause non-work-conserving behavior: a given job may be unable to execute even though all of its predecessor jobs have completed. Under any GEL scheduler, work-conserving behavior can be restored in such a case without altering response-time bounds (Devi and Anderson, 2008; Erickson and Anderson, 2011; Erickson *et al.*, 2014) via a technique called *early releasing* (Anderson and Srinivasan, 2000), which allows a job to execute "early," before its "actual" release time.

[2]Under such a scheduler, each job has a priority point within a constant distance of its release; an earliest-priority-point-first order is assumed.



R^1_1
τ^1_1
$\Phi^1_1 = 0$
R^1_2
τ^1_2
Φ^1_2
R^1_3
τ^1_3
Φ^1_3
R^1_4
τ^1_4
Φ^1_4
end-to-end response time of invocation 1
end-to-end response time of invocation 2
R^1: the end-to-end response-time bound for G^1
0
5
10
15
20
25
30
Time
Job Release
Job Deadline
Job Completion
CPU Execution
(Assume depicted jobs are scheduled alongside other jobs, which are not shown.)

Figure 4.2: Example schedule of the tasks in G^1 in Figure 4.1.

Schedulability. For DAGs as considered here, schedulability conditions for ensuring bounded response times hinge on conditions for ensuring bounded tardiness under GEL scheduling. Assuming a CPU-only platform with M processors, if intra-task parallelism is forbidden, then the required conditions are $u^i_v \leq 1$ for each v and $\sum u^i_v \leq M$ (Devi and Anderson, 2008; Erickson *et al.*, 2014). On the other hand, if arbitrary intra-task parallelism is allowed, then only $\sum u^i_v \leq M$ is required and *per-task utilizations can exceed* 1.0 (Erickson and Anderson, 2011; Erickson *et al.*, 2014). These conditions remain unaltered if arbitrary early releasing is allowed.

As noted earlier, response-time bounds for GPU nodes are needed in order to compute end-to-end response-time bounds. We derive such bounds for NVIDIA GPUs next.

4.2 GPU Response-Time Bound

In this section, we derive a response-time bound for tasks executing on NVIDIA GPUs. We provided a comprehensive background relating to NVIDIA GPUs in Section 2.3, of which we review the essentials to facilitate establishing the task model for GPU tasks.

4.2.1 NVIDIA GPU Details

The compute units of NVIDIA GPUs are SMs, typically comprised of 64 or 128 physical GPU cores. The SMs together can be logically viewed as an EE. Execution on these GPUs is constrained by the number of available GPU threads, which we call *G-threads* to distinguish from CPU threads; on current NVIDIA GPUs, there are 2,048 G-threads per SM.

CUDA programs submit work to a GPU as kernels. A kernel is run on the GPU as a series of thread blocks. These thread blocks, or simply *blocks*, are each comprised of a number of G-threads. The number of blocks and G-threads per block (*i.e.*, the *block size*) are set at runtime when a kernel is launched. The GPU scheduler uses these values to assign work to the GPU's SMs. *Blocks are the schedulable entities on the GPU.* All G-threads in a block are always scheduled on the same SM, and execute non-preemptively.

In Chapter 3, we documented scheduling rules used by NVIDIA GPUs when either all GPU work is submitted from the same address space or NVIDIA's multi-process service (MPS) (NVIDIA, 2019e) is used, which we assume. For simplicity, we restate here only the rules needed for our purposes. These rules govern how kernels are enqueued on and dequeued from a FIFO *EE queue*, as depicted in Figure 4.3. CUDA also provides a concept called a *CUDA stream* that adds an additional layer of queueing in the form of *stream queues* prior to the EE queue. However, as explained later, we assume streams are used in a way that effectively obviates these additional queues. (Our statement of Rule G2 has been simplified from the original to reflect this assumption.) The following terminology is used in the rules below. A block is *assigned* when it is scheduled for execution on an SM. A kernel is *dispatched* when one or more of its blocks are assigned. A kernel is *fully dispatched* when its last block is assigned.

G2 A kernel is enqueued on the EE queue when launched.

G3 A kernel at the head of the EE queue is dequeued from that queue once it becomes fully dispatched.

X1 Only blocks of the kernel at the head of the EE queue are eligible to be assigned.

R1 A block of the kernel at the head of the EE queue is eligible to be assigned if its resource constraints are met.

Constrained resources include shared memory, registers, and (of course) G-threads. We assume that an NVIDIA-provided CUDA compiler option limiting register usage is applied to obviate blocking for registers. We consider techniques for handling shared-memory-induced blocking later.

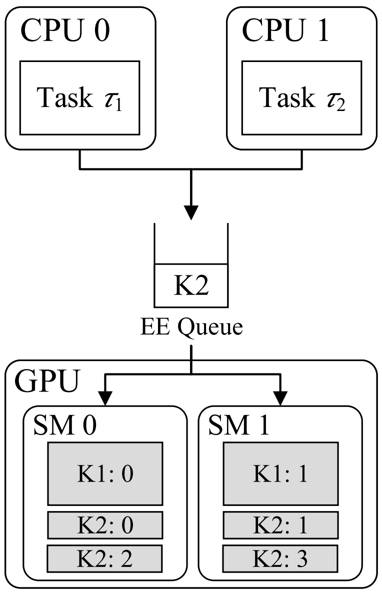


Figure 4.3: GPU scheduling; kernel Kk's bth block is Kk:b.

Example 4.2. In the simple case depicted in Figure 4.3, the GPU is comprised of two SMs. Two tasks submit one kernel each, and these are immediately enqueued on the EE queue upon launch (Rule G2). Kernel K1 is comprised of two blocks of 1,024 G-threads each; K2 is comprised of six blocks of 512 G-threads each. K1 is fully dispatched, so it has been dequeued from the EE queue (Rule G3). The remaining two blocks of $K2$ do not fit on either SM, and thus are not yet assigned (Rule R1); K2 is not fully dispatched, so it is still at the head of the EE queue. Any new kernel K3 would be behind K2 in the EE queue, so its blocks would be ineligible to be assigned until K2 is fully dispatched (Rule X1). $\Diamond$

4.2.2 System Model

One of the virtues of having each node restricted to a type of device in a DAG, as we mentioned in Section 4.1, is that concerns pertaining to CPU and GPU work can be considered separately. Therefore, GPU kernels are just sporadic tasks that can be analyzed independently from CPU tasks. In deriving a response-time bound, we therefore restrict our attention to a set τ of n independent sporadic GPU tasks $\{\tau_1, \tau_2, \cdots, \tau_n\}$, which are scheduled via the rules in Section 4.2.1 on a single GPU with multiple SMs. Each task τ_i has a period T_i, defined as before.

With our focus on GPU-using tasks, additional notation (to be illustrated shortly) is needed to express execution requirements. Each job of task τ_i consists of B_i *blocks*, each of which is executed in parallel by *exactly*[3] H_i G-threads. H_i is called the *block size*[4] of τ_i, and $H_{max} = \max_i\{H_i\}$ denotes the maximum block size in the system. We denote by C_i the per-block worst-case execution *workload* of a block of τ_i, where one unit of execution workload is defined by the work completed by one G-thread in one time unit. In summary, a GPU task τ_i is specified as $\tau_i = (C_i, T_i, B_i, H_i)$.

Note that C_i corresponds to an amount of execution *workload* instead of execution *time*. As each block of task τ_i requires H_i threads concurrently executing in parallel, the worst-case execution time of a block of τ_i is given by C_i/H_i.

Definition 4.1. (block length) For each task τ_i, its *maximum block length* is defined as $L_i = C_i/H_i$. The *maximum block length* of tasks in τ is defined as $L_{max} = \max_i\{L_i\}$.

The *utilization* of task τ_i is given by $u_i = C_i \cdot B_i/T_i$, and the *total system utilization* by $U_{sum} = \sum_{\tau_i \in \tau} u_i$.

Let $\tau_{i,j}$ denote the $j^{th}(j \geq 1)$ job of τ_i. The *release time*[5] of job $\tau_{i,j}$ is denoted by $r_{i,j}$, its *(absolute) deadline* by $d_{i,j} = r_{i,j} + T_i$, and its *completion time* by $f_{i,j}$; its *response time* is defined as $f_{i,j} - r_{i,j}$. A task's response time is the maximum response time of any its jobs.

SM constraints. We consider a single GPU platform consisting of g identical SMs, each of which consists of m G-threads (for NVIDIA GPUs, $m = 2048$). A single block of τ_i must execute on H_i G-threads that belong to the *same* SM. That is, as long as there are fewer than H_i available G-threads on each SM, a block of $\tau_{i,j}$ cannot commence execution even if the total number of available G-threads (from multiple SMs) in the GPU exceeds H_i. On the other hand, different blocks may be assigned to different SMs for execution even if these blocks are from the same job.

Similar to G-thread limitations, there are per-SM and per-block limits on shared-memory usage on NVIDIA GPUs. In experimental work involving computer-vision workloads on NVIDIA GPUs spanning several years, *we have never observed blocking due to shared-memory limits on any platform for any workload*. Thus, in deriving our response-time bound in Section 4.2.4, we assume that such blocking does not occur.

[3]Blocks are executed in units of 32 G-threads called *warps*. *Warp schedulers* switch between warps to hide memory latency. This can create interference effects that must be factored into the timing analysis applied to blocks, which we assume is done in a measurement-based way. With warp-related interference incorporated into timing analysis, the G-threads in a block can be treated as executing simultaneously.

[4]Current NVIDIA GPUs require block sizes to be multiples of 32, and the CUDA runtime rounds up accordingly. Additionally, the maximum possible block size is 1,024. A task's block size is determined offline.

[5]For the time being, we assume that jobs of GPU tasks are not early released, but we will revisit this issue at the end of Section 4.2.4.

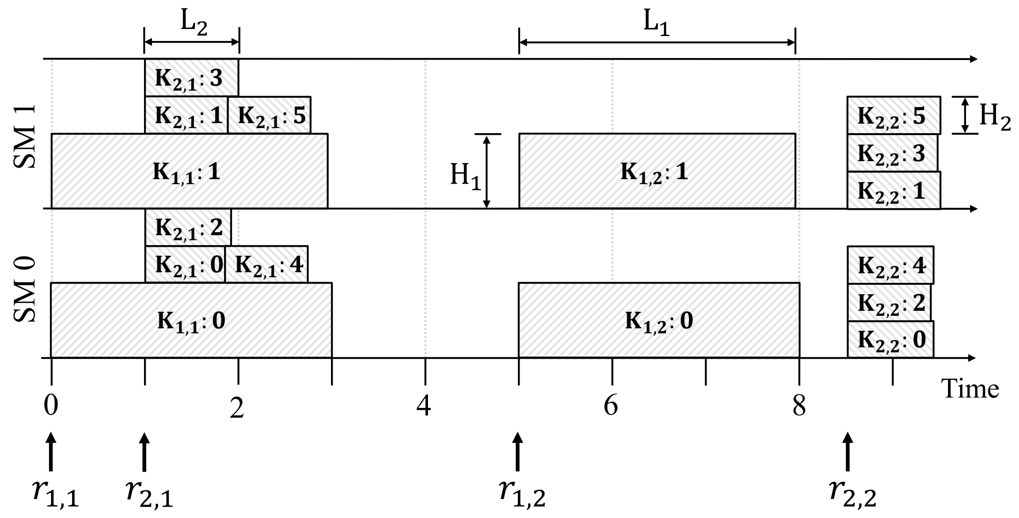


Figure 4.4: A possible schedule corresponding to Figure 4.3, where $m = 2048$, $\tau_1 = (3072, 5, 2, 1024)$, and $\tau_2 = (512, 8, 6, 512)$; rectangle $K_{i,j}$:b corresponds to the bth block of job $\tau_{i,j}$.

After deriving the bound, we discuss ways in which shared-memory blocking can be addressed if it becomes an issue.

Example 4.3. Our GPU task model is illustrated in Figure 4.4. There are two SMs, and $B_1 = 2$ and $B_2 = 6$. The height of a rectangle denoting a block is given by H_i and its length, which denotes its runtime duration, by L_i; the area is bounded by C_i. ♢

Intra-task parallelism. We assume that intra-task parallelism is allowed: consecutive jobs of the same task can execute in parallel if both are pending and sufficient G-threads are available. This is often the case in computer-vision processing pipelines where each video frame is processed independently. Additionally, Theorem 4.1 below shows that severe schedulability-related consequences exist if intra-task parallelism is forbidden. Practically speaking, intra-task parallelism can be enabled by assuming per-*job* streams. A stream is a FIFO queue of operations, so two kernels submitted to a single stream cannot execute concurrently. Thus, the alternative of using per-*task* streams would preclude intra-task parallelism. Note that, with each job issuing one kernel at a time, any actual stream queueing is obviated.

4.2.3 Total Utilization Constraint

According to the rules in Section 4.2.1, idle G-threads can exist while the kernel at the head of the EE queue has unassigned blocks. In particular, this can happen when the number of idle threads on any one SM is insufficient for scheduling such a block. Such scenarios imply that some *capacity loss* is fundamental when seeking to ensure response-time bounds for GPUs. We express such loss by providing a *total utilization bound* and proving that any system with U_{sum} at most that bound has bounded response times. The utilization bound we present relies on the following definition.

Definition 4.2. (unit block size) The *unit block size*, denoted by h, is defined by the greatest common divisor (gcd) of all tasks' block sizes and m, *i.e.*, $h = \gcd(\{H_i\}_{i=1}^{n} \cup \{m\})$.

The theorem below shows that capacity loss can be extreme if intra-task parallelism is forbidden.

Theorem 4.1. *With per-task streams, for any given g, m, H_{max}, and h, there exists a task system τ with U_{sum} greater than but arbitrarily close to h such that the response time of some task may increase without bound in the worst case.*

Proof. For any m and H_{max}, $m = Z \cdot h$ for some integer $Z \geq 1$ and $H_{max} = K \cdot h$ for some integer $K \geq 1$, because h is a common divisor of m and H_{max}. Recall that there are g SMs. Consider the following task system:

τ_i	C_i	T_i	B_i	H_i	L_i
τ_1	h	1	1	h	1
τ_2	$2\varepsilon \cdot H_{max}$	$1+\varepsilon$	1	H_{max}	2ε
τ_3	$2\varepsilon \cdot h$	$1+\varepsilon$	$g \cdot Z - K$	h	2ε

For this task system, $u_1 = h$, $u_2 = \frac{2H_{max}}{1+\varepsilon}\varepsilon$, and $u_3 = \frac{2h\cdot(g\cdot Z-K)}{1+\varepsilon}\varepsilon$, so $U_{sum} \to h^+$ as $\varepsilon \to 0^+$.

Now consider the following job execution pattern, which is illustrated in Figure 4.5 for $g = 2$: τ_1 releases its first job at time 0, τ_2 and τ_3 release their first jobs at time $1-\varepsilon$, all three tasks continue to release jobs as early as possible, and every block executes for its worst-case execution workload. Assume that, every time when τ_2 and τ_3 simultaneously release a job, the job of τ_2 is enqueued on the EE queue first. Note that in Figure 4.5, block boundaries for τ_3 are omitted when possible for clarity.

At time 0, $\tau_{1,1}$ is the only job in the EE queue and is therefore scheduled. Then, at time $1-\varepsilon$, $\tau_{2,1}$ and $(g \cdot Z - K - 1)$ blocks of $\tau_{3,1}$ are scheduled for the interval $[1-\varepsilon, 1+\varepsilon)$. As a result, all available G-threads are then occupied. Therefore, the remaining one block of $\tau_{3,1}$ must wait until time 1 when $\tau_{1,1}$ finishes. Note that, with per-task streams, a job cannot enter the EE queue until the prior job of the same task completes. $\tau_{1,2}$ enters the EE queue at time 1 after $\tau_{3,1}$, which entered at time $1-\varepsilon$. Thus, $\tau_{1,2}$ must wait to begin execution until after the last block of $\tau_{3,1}$ is assigned and once sufficient G-threads become available at time $1+\varepsilon$.

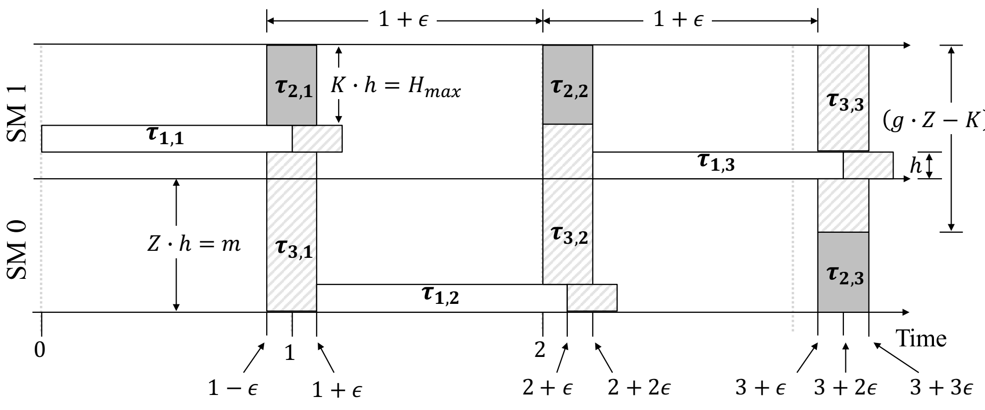


Figure 4.5: Unbounded response time using per-task streams.

This pattern repeats, with task τ_1 releasing a job every time unit but finishing a job every $1+\varepsilon$ time units. Thus, its response time increases without bound. □

For example, on the test platform considered in Section 6.2, $g = 80$, $m = 2048$, and h can be as small as 32. Thus, close to 99.98% of the hardware capacity may be wasted!

In contrast, as we show in Section 4.2.4, if intra-task parallelism is allowed, then any task set with $U_{sum} \leq g \cdot (m - H_{max} + h)$ has bounded response times. Furthermore, the following theorem shows that this utilization bound is tight (under our analysis assumptions).

Theorem 4.2. *With per-job streams, for any given g, m, H_{max}, and h, there exists a task system τ with U_{sum} greater than but arbitrarily close to $g \cdot (m - H_{max} + h)$ such that the response time of some task may increase without bound in the worst case.*

Proof. For any m and H_{max}, integers P and Q exist such that $m = P \cdot H_{max} + Q$, where $P \geq 1$ and $0 \leq Q < H_{max}$. Furthermore, by the definition of h, $H_{max} = K \cdot h$ for some integer $K \geq 1$, and $m = Z \cdot h$ for some integer $Z \geq 1$. Thus, $m = P \cdot H_{max} + Q = P \cdot K \cdot h + Q = Z \cdot h$. Consider the following task set (if $Q = 0$, then τ_2 need not exist):

τ_i	C_i	T_i	B_i	H_i	L_i
τ_1	$\varepsilon \cdot H_{max}$	1	$g \cdot P$	H_{max}	ε
τ_2	$\varepsilon \cdot Q$	1	g	Q	ε
τ_3	h	1	$g \cdot (Z - K + 1)$	h	1

For this task system, $u_1 = (H_{max} \cdot g \cdot P)\varepsilon$, $u_2 = (Q \cdot g)\varepsilon$, and $u_3 = h \cdot g \cdot (Z - K + 1) = g \cdot (m - H_{max} + h)$, so $U_{sum} \to g \cdot (m - H_{max} + h)^+$ as $\varepsilon \to 0^+$.

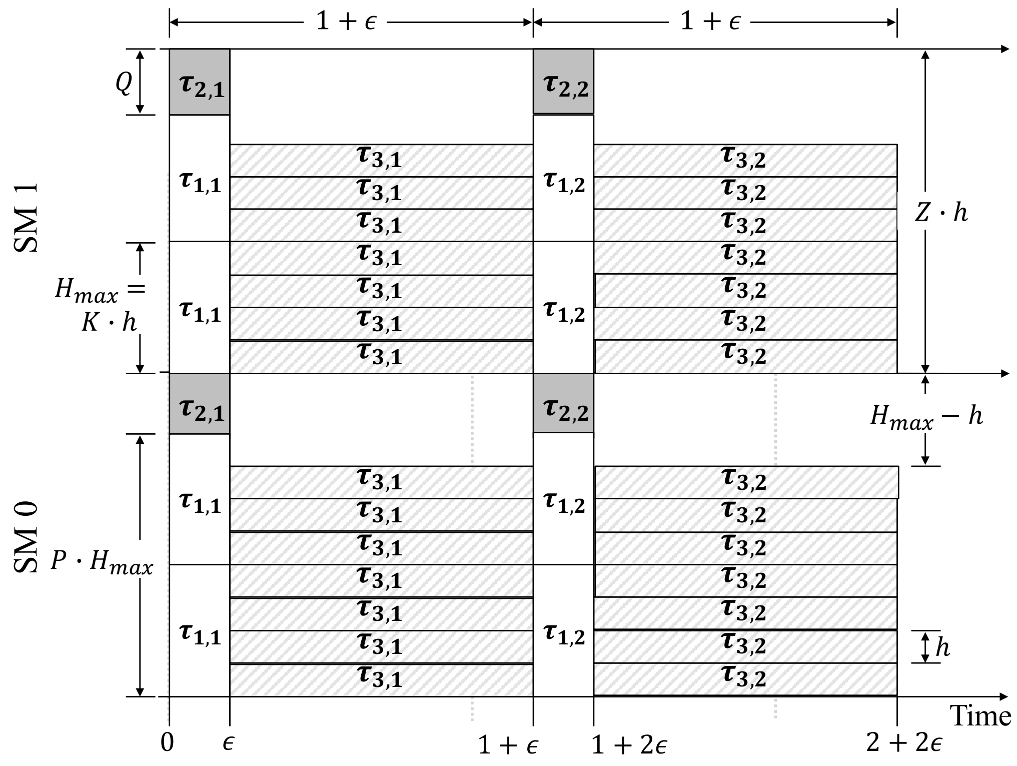


Figure 4.6: Unbounded response time using per-job streams.

Now consider the following job execution pattern, which is illustrated in Figure 4.6 for $g = 2$: all three tasks release jobs as soon as possible, *i.e.*, at time instants $0, 1, 2, \ldots$, the EE enqueue order is always τ_1, τ_2, and then τ_3, and every block executes for its worst-case execution workload.

At time 0, the $g \cdot P$ blocks of τ_1 are scheduled first, leaving Q available G-threads in each SM. Next, the g blocks of τ_2 are scheduled using the remaining Q G-threads on each SM. Thus, all G-threads are fully occupied in the time interval $[0, \varepsilon)$. As we often see in experiments, the $g \cdot (Z - K + 1)$ blocks of τ_3 are distributed to the g SMs evenly, and are scheduled for the time interval $[\varepsilon, 1 + \varepsilon)$. Note that, although we currently do not have sufficient evidence to guarantee this even distribution, it at least represents a potential *worst case*.

Notice that there are only $m - (Z - K + 1) \cdot h = (H_{max} - h)$ G-threads available on each of the g SMs during the interval $[\varepsilon, 1 + \varepsilon)$. Therefore, none of the blocks of $\tau_{1,2}$, which has a block size of H_{max}, will be scheduled before time $1 + \varepsilon$. As a result, no blocks of $\tau_{2,2}$ or $\tau_{3,2}$ will be scheduled before time $1 + \varepsilon$ either, because they are enqueued after $\tau_{1,2}$ on the FIFO EE queue.

This pattern repeats, with each of the three tasks releasing a job every time unit but finishing a job every $1 + \varepsilon$ time units, so the response time of each task increases without bound. $\square$

4.2.4 Response-Time Bound

In this section, we derive a response-time bound assuming per-job streams are used (*i.e.*, intra-task parallelism is allowed) and the following holds.

$$U_{sum} \le g \cdot (m - H_{max} + h) \tag{4.1}$$

Our derivation is based on the following key definition.

Definition 4.3. (busy and non-busy) A time instant is called *busy* if and only if at most $(H_{max} - h)$ G-threads are idle in *each* of the g SMs; a time instant is called *non-busy* if and only if at least H_{max} G-threads are idle in *some* of the g SMs. A time interval is called *busy* if and only if every time instant in that interval is busy.

By Definition 4.2, h is the minimum amount by which the number of idle G-threads can change, so "more than $(H_{max} - h)$ G-threads are idle" is equivalent to "at least H_{max} G-threads are idle." Thus, busy and non-busy time instants are well-defined, *i.e.*, a time instant is either busy or non-busy.

To derive response-time bounds for all tasks in the system, we bound the response time of an arbitrary job $\tau_{k,j}$. The following two lemmas bound the unfinished workload at certain time instants. In the first lemma, t_0 denotes the latest non-busy time instant at or before $\tau_{k,j}$'s release time $r_{k,j}$, *i.e.*, $t_0 = r_{k,j}$ or $(t_0, r_{k,j}]$ is a busy interval.

Lemma 4.1. *At time t_0, the total unfinished workload from jobs released at or before t_0, denoted by $W(t_0)$, satisfies $W(t_0) \le L_{max} \cdot (g \cdot m - H_{max})$.*

Proof. Suppose there are b blocks, $\beta_1, \beta_2, \ldots, \beta_b$, that have been released but are unfinished at time t_0. For each block β_i, let $H(\beta_i)$ denote its block size and let $C(\beta_i)$ denote its worst-case execution workload. By definition, t_0 is a non-busy time instant, so by Definition 4.3, at least H_{max} G-threads are idle in some SM at time t_0. Because this SM has enough available G-threads to schedule any of the b blocks, they all must be scheduled at time t_0. These facts imply

$$\sum_{i=1}^{b} H(\beta_i) \le g \cdot m - H_{max}. \tag{4.2}$$

$$
\begin{aligned}
\text{Therefore,}\quad W(t_o) &= \sum_{i=1}^{b} C(\beta_i) \\
&= \{\text{by the definition of } L_i \text{ in Definition 4.1}\} \\
&\quad \sum_{i=1}^{b} (L(\beta_i) \cdot H(\beta_i)) \\
&\leq \{\text{by the definition of } L_{max} \text{ in Definition 4.1}\} \\
&\quad \sum_{i=1}^{b} (L_{max} \cdot H(\beta_i)) \\
&= \{\text{rearranging}\} \\
&\quad L_{max} \cdot \sum_{i=1}^{b} H(\beta_i) \\
&\leq \{\text{by (4.2)}\} \\
&\quad L_{max} \cdot (g \cdot m - H_{max}).
\end{aligned}
$$

The lemma follows. □

Lemma 4.2. *At time $r_{k,j}$, the total unfinished workload from jobs released at or before $r_{k,j}$, denoted by $W(r_{k,j})$, satisfies $W(r_{k,j}) < L_{max} \cdot (g \cdot m - H_{max}) + \sum_{i=1}^{n}(B_i \cdot C_i)$.*

Proof. Let $\mathsf{new}(t_0, r_{k,j})$ denote the workload released during the time interval $(t_0, r_{k,j}]$, and let $\mathsf{done}(t_0, r_{k,j})$ denote the workload completed during the time interval $(t_0, r_{k,j}]$. Then,

$$
W(r_{k,j}) = W(t_0) + \mathsf{new}(t_0, r_{k,j}) - \mathsf{done}(t_0, r_{k,j}). \tag{4.3}
$$

As each task τ_i releases consecutive jobs with a minimum separation of T_i, $\mathsf{new}(t_0, r_{k,j})$ can be upper bounded by

$$
\begin{aligned}
\mathsf{new}(t_0, r_{k,j}) &\leq \sum_{i=1}^{n} \left(\left\lceil \frac{r_{k,j} - t_0}{T_i} \right\rceil \cdot B_i \cdot C_i \right) \\
&< \{\text{because } \lceil a \rceil < a + 1\} \\
&\quad \sum_{i=1}^{n} \left(\left(\frac{r_{k,j} - t_0}{T_i} + 1 \right) \cdot B_i \cdot C_i \right) \\
&= \{\text{rearranging}\}
\end{aligned}
$$

$$\left(r_{k,j} - t_0\right) \sum_{i=1}^{n} \frac{B_i \cdot C_i}{T_i} + \sum_{i=1}^{n} (B_i \cdot C_i)$$

$$= \{\text{by the definitions of } u_i \text{ and } U_{sum}\}$$

$$\left(r_{k,j} - t_0\right) U_{sum} + \sum_{i=1}^{n} (B_i \cdot C_i). \tag{4.4}$$

By Definition 4.3, $(t_0, r_{k,j}]$ being a busy time interval implies that at most $(H_{max} - h)$ G-threads in each of the g SMs are idle at any time instant in this time interval. That is, at least $g \cdot (m - H_{max} + h)$ G-threads are occupied executing work at any time instant in $(t_0, r_{k,j}]$. Therefore,

$$\mathsf{done}(t_0, r_{k,j}) \geq (r_{k,j} - t_0) \cdot g \cdot (m - H_{max} + h). \tag{4.5}$$

By (4.3), (4.4), and (4.5),

$$\begin{aligned}
W(r_{k,j}) &< W(t_0) + \left(r_{k,j} - t_0\right) U_{sum} + \sum_{i=1}^{n} (B_i \cdot C_i) \\
&\qquad - (r_{k,j} - t_0) \cdot g \cdot (m - H_{max} + h) \\
&= \{\text{rearranging}\} \\
&\quad (r_{k,j} - t_0) \cdot (U_{sum} - g \cdot (m - H_{max} + h)) \\
&\qquad + W(t_0) + \sum_{i=1}^{n} (B_i \cdot C_i) \\
&\leq \{\text{by (4.1) and } t_0 \leq r_{k,j}\} \\
&\quad W(t_0) + \sum_{i=1}^{n} (B_i \cdot C_i) \\
&\leq \{\text{by Lemma 4.1}\} \\
&\quad L_{max} \cdot (g \cdot m - H_{max}) + \sum_{i=1}^{n} (B_i \cdot C_i).
\end{aligned}$$

The lemma follows. □

The following theorem provides our response-time bound.

Theorem 4.3. *$\tau_{k,j}$ finishes the execution of all of its blocks by time $r_{k,j} + R_k$, where*

$$R_k = \frac{L_{max} \cdot (g \cdot m - H_{max}) + \sum_{i=1}^{n} (B_i \cdot C_i) - C_k}{g \cdot (m - H_{max} + h)} + L_k. \tag{4.6}$$

Proof. Since the EE queue is FIFO, we omit all jobs released after $r_{k,j}$ in the analysis. Thus, any workload executed at or after $r_{k,j}$ is from $W(r_{k,j})$. We also assume each block of $\tau_{k,j}$ executes for its worst-case workload C_k (if any of its blocks executes for less, $\tau_{k,j}$'s completion is not delayed).[6]

Let β^* denote the *last-finished* block of $\tau_{k,j}$. Then, the workload from other blocks or jobs at $r_{k,j}$ is $W(r_{k,j}) - C_k$. Let t^* denote the time instant at which β^* starts execution. Then, $[r_{k,j}, t^*)$ is a busy interval (else β^* would have executed before time t^*). Let $\mathsf{done}(r_{k,j}, t^*)$ denote the workload completed during the time interval $[r_{k,j}, t^*)$. Then, by Definition 4.3,

$$\mathsf{done}(r_{k,j}, t^*) \geq (t^* - r_{k,j}) \cdot g \cdot (m - H_{max} + h). \tag{4.7}$$

The workload C_k from β^* executes beyond time t^*, so $\mathsf{done}(r_{k,j}, t^*) \leq W(r_{k,j}) - C_k$. By (4.7), this implies $t^* \leq r_{k,j} + \frac{W(r_{k,j}) - C_k}{g \cdot (m - H_{max} + h)}$. At time t^*, β^* executes continuously for L_k time units. Thus, β^* finishes by time $r_{k,j} + \frac{W(r_{k,j}) - C_k}{g \cdot (m - H_{max} + h)} + L_k$. By Lemma 4.2, the theorem follows. □

Discussion. As noted in Section 4.2.2, the absence of shared-memory-induced blocking is assumed in the above analysis. This limitation could be eased by introducing blocking terms, but we leave this for future work. Alternatively, through offline profiling, one could restrict the per-SM G-thread count of m to some value m' such that, if only m' G-threads are used per SM, no shared-memory-induced blocking ever occurs. The analysis above could then be applied with m replaced by m'. While one might expect this analysis to be sustainable in the sense that m per-SM G-threads could really be used at runtime, we found a counterexample (Example 4.4) where increasing m' to m causes response times to increase. Thus, the restricted G-thread count of m' would actually have to be enforced. This could potentially be done by creating a never-ending kernel per SM that monopolizes $m - m'$ G-threads.

Example 4.4. A counterexample that shows this analysis is not sustainable with respect to increasing m is illustrated in Figures 4.7 and 4.8, where $\tau_1 = (1216, 8, 1, 608)$, $\tau_2 = (1920, 8, 6, 480)$, and $\tau_3 = (1536, 8, 1, 768)$. The response time of τ_3 increases from 2 to 5 as m increases from 2016 to 2048. ◊

Early releasing (see Sections 4.1 and 6.1.2) must be restricted for GPU tasks. Because the FIFO EE queue effectively prioritizes work by *actual* enqueueing times, uncontrolled early releasing can change priorities. Also, this can lead to a violation of the sporadic task model if consecutive jobs of the same task τ_i have

[6]Other jobs' blocks may or may not execute for their worst case.

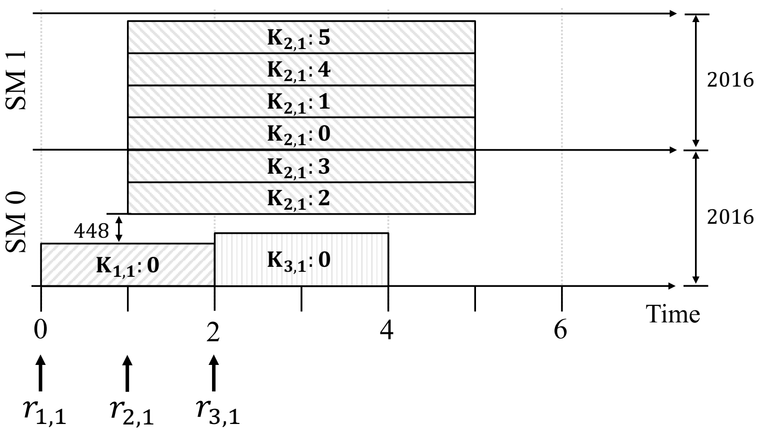


Figure 4.7: A possible schedule for Example 4.4 with $m = 2016$.

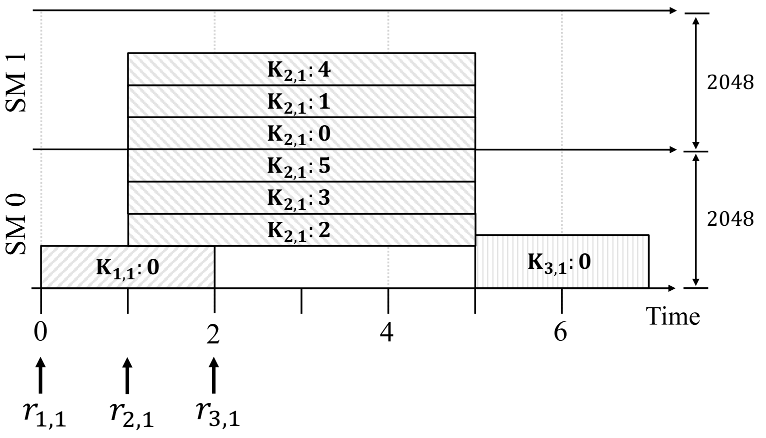


Figure 4.8: A possible schedule for Example 4.4 with $m = 2048$.

enqueueing times less than T_i time units apart. Thus, the early releasing of GPU tasks must be guarded to ensure that this minimum separation is maintained.

4.3 Chapter Summary

In this chapter, we presented a task model for GPU tasks. Using this model, we derived the first ever response-time bounds under the NVIDIA GPU scheduling rules. We explained how GPU-using workloads can be modeled into DAG tasks and how prior work on DAG scheduling can be leveraged to compute end-to-end response-time bounds for these DAG tasks.

In Chapter 5, we will transition from theory to practice, by first exploring pipelined and parallel execution for improving the throughput of CNN applications. In Chapter 6, we will demonstrate a case study that shows both such throughput improvement and bounded response times can be achieved for the OpenVX standard, and evaluate our analytical end-to-end response time bounds in comparison with prior works.

Acknowledgement. The description of the DAG model in Section 4.1 draws heavily from (Yang *et al.*, 2016a). I defined the task model, constructed the total utilization constraints, and developed the initial response-time bound analysis, which Kecheng Yang then reworked into the current version in Section 4.2.4.

CHAPTER 5: IMPROVE CNN FRAMEWORKS FOR AUTONOMOUS-DRIVING APPLICATIONS[1]

In this chapter, we report on the results of an industrial automotive design study focusing on the application of convolutional neural networks (CNNs) in computer vision for autonomous or automated-driving cars.

In Chapter 2, we reviewed that CNNs are essential building blocks for computer-vision applications that process camera images in diverse uses including object detection and recognition (cars, people, bicycles, signs), scene segmentation (finding roads, sidewalks, buildings), localization (generating and following a map), *etc*. Computer-vision applications based on CNN technology are essential because cameras deployed on cars are much less costly than other sensors such as lidar.

Computer vision in a resource-constrained setting. While cameras are cheap, computer-vision computational costs are not. Keeping overall monetary cost low is essential for consumer-market cars and requires effective management of computing hardware. Because state-of-the-art computer-vision approaches have been shown to take over 94% of the computational capacity in a simulated vehicle (Lin *et al.*, 2018), computer hardware costs can be significantly reduced by optimized implementations. Existing CNN frameworks have some amount of optimization with parallelism, but focus largely on processing a single stream of images in a parallel manner only if a CNN model contains parallel layers. The solutions considered in this study re-think the way CNN software is implemented so that effective resource management can be applied to CNN applications. Our study focuses on one of the most demanding computer-vision applications in cars, the timely detection/recognition of objects in the surrounding environment.

Industrial challenge. The major challenge we seek to address is to resolve the inherent tensions among requirements for throughput, response time, and accuracy, which are difficult to satisfy simultaneously.

[1]Contents of this chapter previously appeared in the following paper:

Yang, M., Wang, S., Bakita, J., Vu, T., Smith, F. D., Anderson, J. H., and Frahm, J.-M. (2019). Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In *Proceedings of the 25th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 305–317.

The sheer number and diversity of cameras necessary to provide adequate coverage and redundancy (for safety) drives throughput requirements. A typical configuration in today's experimental automated-driving vehicle includes ten or more cameras that cover different fields of view, orientations, and near/far ranges. Each camera generates a stream of images at rates ranging from 10 to 40 frames per second (FPS) depending on its function (*e.g.*, lower rates for side-facing cameras, higher rates for forward-facing ones). All streams must be processed simultaneously by the object detection/recognition application.

For this application, response time (latency from image capture to recognition completion) is time-sensitive because of its position on the critical path to control and actuation of the vehicle. Every millisecond that elapses during object recognition is a millisecond not available for effecting safe operation. Ideally, the camera-to-recognition latency per frame should not substantially exceed the inter-frame time of the input images (*e.g.*, 25 milliseconds for a 40 FPS camera).

Accuracy requirements for detection/recognition depend on a number of factors specific to the functions supported by a given camera. For example, accuracy may be much more important in streams from front-facing cameras than those from side-facing cameras. For most of the design alternatives we consider, our modifications have no effect on the accuracy of the computer-vision CNN. However, we do explore one particular instance of a throughput vs. accuracy trade-off of relevance in systems with many cameras (see Section 5.4).

To summarize, our challenge is to *re-think the design of CNN software to better utilize hardware resources and achieve increased throughput (number of simultaneous camera streams) without any appreciable increase in per-frame latency (camera to CNN output) or reduction of per-stream accuracy.*

Our hardware and software environment. In this study, we use the NVIDIA DRIVE PX Parker AutoChauffeur (previously called the 'DRIVE PX2' or just 'PX2') (NVIDIA, 2016a). This embedded computing platform is marketed by NVIDIA as the "first artificial-intelligence (AI) supercomputer for autonomous and semi-autonomous vehicles." The PX2 incorporates multicore processors along with multiple GPUs to accelerate computer-vision applications. With respect to such applications, we focus on a variant of the open-source CNN framework 'Darknet' configured to implement an object detection/recognition program called 'Tiny YOLO' (version 2, 'YOLO' for short) (Redmon and Farhadi, 2017; Redmon, 2016, 2017). While our study targets specific hardware and CNN software, our hardware and software choices are representative of the multi-core/multi-GPU hardware and CNN designs that are used and will continue to be used for automated driving.

Our design study. As stated earlier, the fundamental challenge we consider is to design an object-detection application that can support independent streams of images from several cameras. Due to the constraining nature of our hardware, effective utilization of every part of the system is essential. Our preliminary measurements (provided in Section 5.4) revealed that GPU execution time dominates the time spent processing each image in YOLO, yet one of the PX2's two GPUs[2] was left completely idle and the second was utilized at less than 40%. This result motivated us to explore system-level optimizations of YOLO as a path to higher throughputs.

Many CNN implementations similarly seem to have limited throughput and poor GPU utilization. As explained further in Section 5.2, the root of the problem lies in viewing processing as a single run-through of all the processing layers comprising a CNN. However, in processing an image sequence from one camera, each layer's processing of image i is independent of its processing of image $i+1$. If the intermediate per-image data created at each layer is buffered and sequenced, there is no reason why successive images from the same camera cannot be processed in different layers concurrently. This motivates us to consider the classic *pipelining* solution to sharing resources for increased throughput.[3] While pipelining offers the potential for greater parallelism, concurrent executions of the same layer on different images are still precluded. While allowing this may be of limited utility with one camera, it has the potential of further increasing throughput substantially when processing streams of images from multiple cameras (all of which are also independent). This motivates us to extend pipelining by enabling each layer to execute in *parallel* on multiple images.

We altered the design of Darknet to enable pipelined execution, with parallel layer execution as an option, in the YOLO CNN. We also implemented a technique whereby different camera image streams are combined into a single image stream by combining separate per-camera images into a single composite image. This compositing technique can greatly increase throughput at the expense of some accuracy loss. We also enabled the balancing of GPU work across the PX2's two GPUs, which have differing processing capabilities. The YOLO variants arising from these options and various other implementation details are discussed in Section 5.3. The options themselves are evaluated in Section 5.4.

[2]Technically, the PX2 provides four GPUs split between two systems on a chip (SoCs). We considered only one of these SoCs in our evaluation.

[3]For example, instruction pipelining in processors can fully utilize data-path elements and increase throughput.

Contribution. This chapter provides an effective solution for a challenging industry problem: designing CNN systems that can provide the throughput, timeliness, and accuracy for computer-vision applications in automated and autonomous driving. Through re-thinking how the YOLO CNN is executed, we show that its throughput can be improved by more than twofold with only a minor latency increase and no change in accuracy. We also show that throughput can be further improved with proper GPU load balancing. Finally, we show that the compositing technique can enable very high throughputs of up to 250 FPS on the PX2. While this technique does incur some accuracy loss, we show that such loss can be mitigated by redoing the "training" CNNs require. We claim these results are sufficiently fundamental to guide CNN construction for similar automotive computer-vision applications on other frameworks or hardware.

In order to understand these contributions in full, an understanding of relevant background information and related work is required. This we provide in the next two sections.

5.1 Background

In this section, we provide detailed descriptions of our hardware platform, the DRIVE PX2 (NVIDIA, 2016a), and the Darknet CNN framework. The object-detection application ("YOLO") that runs using this framework has been reviewed in Section 2.2.

5.1.1 NVIDIA DRIVE PX2

The PX2 is embedded hardware specifically designed to provide high-performance computing resources for autonomous operation on a platform with small size, weight, and power characteristics.

Architecture. The PX2 contains two identical and independent "Parker" systems on a chip (SoCs), called Tegra A and B, connected by a 1 Gbps Controller Area Network (CAN) bus. We show one of the identical halves of the PX2 in Figure 5.1. Each SoC contains six CPUs (four ARMv8 Cortex A57 cores, two ARMv8 Denver 2.0 cores), and an integrated Pascal GPU (iGPU). Each Parker SoC also connects to a discrete Pascal GPU (dGPU) over its PCIe x4 bus. While the iGPU and CPU cores share main DRAM memory, the dGPU has its own DRAM memory. Memory transfers between CPU memory and dGPU memory use the PCIe bus. *The remainder of this chapter, including the design descriptions and evaluation experiments, is based on using only one of the PX2's two independent systems.*

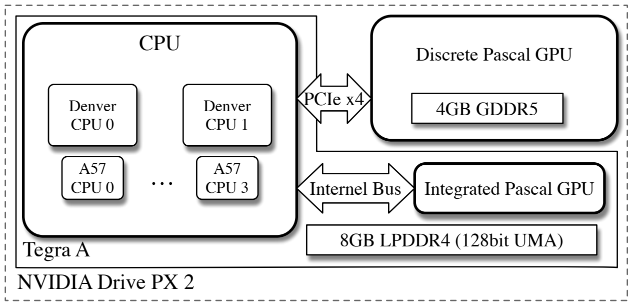


Figure 5.1: NVIDIA DRIVE PX2 architecture (showing one of the PX2's two identical and independent systems).

	Integrated GPU (iGPU)	Discrete GPU (dGPU)
Computing units	256 CUDA Cores	1152 CUDA Cores
Accessible memory	6668 MBytes	3840 MBytes
Memory bandwidth (Sundaram, 2016)	≈50 GBytes/s	≈80 GBytes/s
Shared L2 cache size	512 KBytes	1,024 KBytes

Table 5.1: Specifications of the PX2 iGPU and dGPU.

iGPU vs. dGPU. As previously mentioned, the PX2 has two types of GPUs—the iGPU and the dGPU—using the same Pascal architecture. While they share an architecture, they share little else. Consider their specifications as shown in Table 5.1. The dGPU has more cores, faster memory, and a larger L2 cache. The iGPU lags behind, advantaged only by its access to the larger, albeit shared, main memory DRAM banks.

The exact performance differences between the iGPU and the dGPU depend on the characteristics of GPU programs used in a specific application. We characterize these differences for YOLO in Section 5.4.

5.1.2 YOLO and Darknet

For most of the experiments in Section 5.4, we used the YOLO CNN as already trained on a widely used set of relevant training images (VOC 2007+12 (Everingham *et al.*, 2010, 2015)), so we did not modify the YOLO network architecture or training approach. This means that most of our work applies to any CNN constructed using Darknet, from YOLOv3 to AlexNet. This is extremely important as noted in (Lin *et al.*, 2018) because the state of the art rapidly changes for autonomous vehicles. Only one optional enhancement we consider, namely *image compositing*, requires retraining.

The Darknet framework is programmed in C and CUDA. All of our modifications and benchmarks are based on a proprietary fork of Darknet, so frame rates in Section 5.4 are not directly comparable to those from the open-source version.[4] Unfortunately, we are unable to open source our fork due to intellectual property management issues.

[4]Specifically, our base version of Darknet pre-loads frames and performs frame resizing in the main thread whereas the open-source version uses a separate thread.

5.2 Related Work

In the area of parallel CNN scheduling, frameworks we reviewed in Section 2.2.3 such as TensorFlow (Abadi *et al.*, 2016) already support *intra-model* CNN graphs. These graphs serve to enable "parallelism, distributed execution, compilation, and portability" in their applications (per their website). However, neither the original paper (Abadi *et al.*, 2016) nor their survey of computer-vision CNNs (Huang *et al.*, 2017) seems to consider anything besides parallelism inside one execution of a model that contains parallel layers. Their graphs seem mostly to serve to formalize the implicit dataflow graph present in any CNN. In contrast, our work deals with synthesizing an extra-model *shared CNN* by which to collapse multiple CNN instances into a single shared program. It is worth emphasizing that our efforts are designed to largely work alongside the vast body of existing CNN optimization and compression efforts such as DeepMon (Huynh *et al.*, 2017).

One other work similar to ours, *S^3DNN*, applied techniques such as kernel-level deadline-aware scheduling and batch processing (data fusion) that require additional middleware (Zhou *et al.*, 2018). We focus on higher-level parallelism by applying a graph-based architecture.

Some of our work here also considers ways to optimally partition YOLO between GPUs of different capabilities. The partitioning concept is not new—it was used by Kang *et al.* (2017) to dynamically balance CNN layers between a compute-impoverished system and a networked server in the Neurosurgeon system. We, however, use the technique to optimize solely for performance without the networking constraints that dominate that work.

5.3 System Design

In this section, we discuss some of our ideas for how to implement this shared CNN. First, we describe our overarching approach, which involves generalizing the concept of a CNN layer by combining groups of consecutive layers into "stages," examining opportunities for parallelism among stages ("pipelining"), and considering opportunities for parallelism inside each stage ("parallel-pipelining"). Second, our hardware platform provides the somewhat unique opportunity to examine GPU-work-assignment strategies across asymmetric GPUs, so we briefly consider that issue. Third, we propose a novel technique for realizing additional throughput gains in high-camera-count environments. Along the way, we also briefly touch on several other issues that naturally arise with the concept of a shared CNN. These include data handling, communication issues, and implicit synchronization. Before delving too deeply into the details, however, we first motivate why we are interested in this shared approach in the first place.

The obvious solution to support multiple cameras is simple: run multiple instances of the application—one process for each camera—and have them share the GPU(s), as shown in Figure 2.2. We reject this approach for two reasons. First, the high memory requirements of each network instance heavily limits the total number of instances (a maximum of six on our hardware platform). Second, running the models independently does not easily allow for fast synchronization across multiple cameras. Ideally, we would like each network and camera to process frames at the same rate, prioritizing work for networks that are lagging behind. However, this proves extremely challenging in practice as it requires modifying GPU scheduling behavior—a task still not satisfactorily solved by any research that we are aware of.

5.3.1 Enabling Parallelism

As previously noted, CNN layers and models are inherently independent and thus sharable—the only major complexities for us stem from Darknet's limited "bookkeeping" side-effects and from the need to correctly pipe intermediate data between layers. In this section, we explore ways of enabling parallelism in such a shared computation structure.

Decomposition into stages. As a precursor to introducing parallelism, we decompose a shared CNN into a succession of computational *stages*. Each stage is simply a subsequence of layers from the YOLO CNN shown in Figure 2.2, as illustrated in Figure 5.2. To correctly manage both frame data and the bookkeeping data required by YOLO, we buffer indices to this data using queues between stages as shown in Figure 5.2. Note that these queues are not necessarily index-ordered. Conceptually, we can view the data being indexed as being stored in infinite arrays, but in reality, these arrays are "wrapped" to form ring buffers. The size of the frame ring buffer is large enough to prevent any blocking when adding another frame to it. However, memory constraints limit the ring buffer for bookkeeping data to have only ten entries. *This implies that only ten frames can exist concurrently in the shared CNN at any time.* Note that Darknet ensures that the original layers execute correctly without further modification by us when provided with the appropriate bookkeeping data. Data buffering and synchronization in our pipelined implementation are provided by PGM^{RT}, a library created as part of prior work at UNC (Elliott *et al.*, 2014).

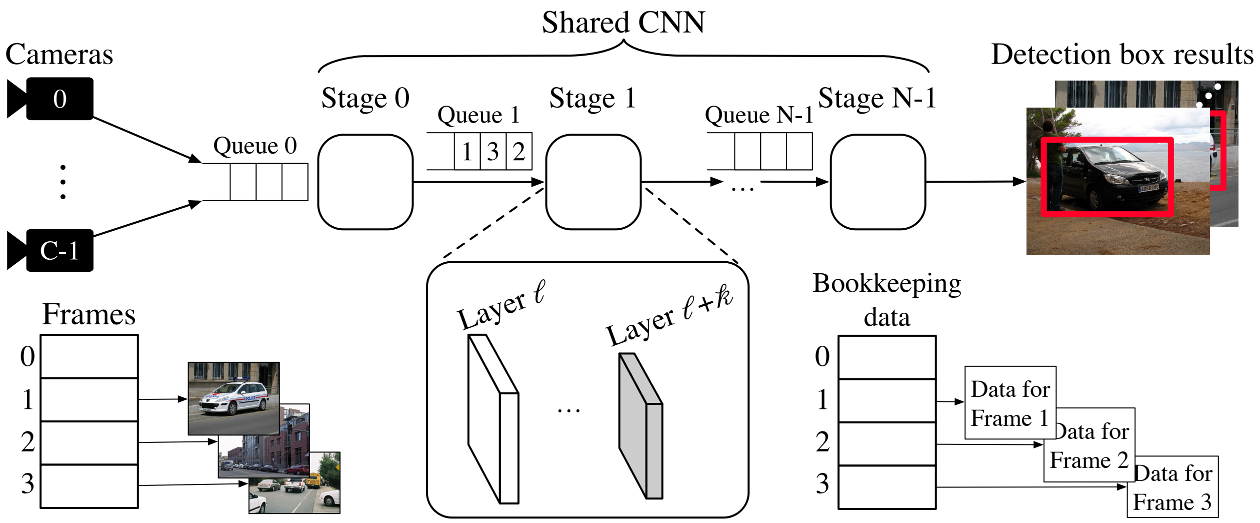


Figure 5.2: Sharing one CNN for multiple cameras.

Serialized execution. Before enabling parallelism, we first look at the scheduling of the default serialized execution of the unmodified YOLO through an example in Figure 5.3 (a). We will explain Figure 5.3 (b) and (c) shortly. Figure 5.3 (a) shows a single three-stage CNN that processes frames serially. At time t_0, the first stage (Stage 0) starts the initial preparation work on the CPU for frame f. This initial preparation work (for example, resizing the frame) requires Stage 0 to execute longer on the CPU than the following stages. Stage 0 launches its kernel, indicated by S0, at time t_1 into the single CUDA stream queue belonging to this CNN instance. Stage 1 launches another kernel, S1, at time t_1, and S2 is launched at time t_2. These kernels S0, S1, and S2 are executed in FIFO order and are completed at times t_4, t_5, and t_6 respectively. Since the intermediate data produced by the first kernels S0 and S1 are not needed for CPU computations, only the last stage suspends to wait for all kernels to complete (at time t_6). The use of intermediate data only by kernels running on the GPU is common to other CNN implementations. Because the processing of multiple frames is serialized by the single CNN, the GPU is often left idle when no kernel requests are in the stream queue.

Pipelined execution. We now explore different methods for enabling the parallel processing of stages. The first method we consider is to enable *pipelined execution*, *i.e.*, we allow successive stages to execute in parallel on different frames. We implement pipelined execution by defining a thread per stage,[5] which processes one frame by performing the computational steps illustrated in Figure 5.4 for a stage comprised of one convolution layer followed by one maxpool layer. Each stage's thread starts by waiting on the arrival of

[5]While observation 3.4 indicates that processes with MP(MPS) are superior to threads (MT), MPS is not supported in the NVIDIA DRIVE products.

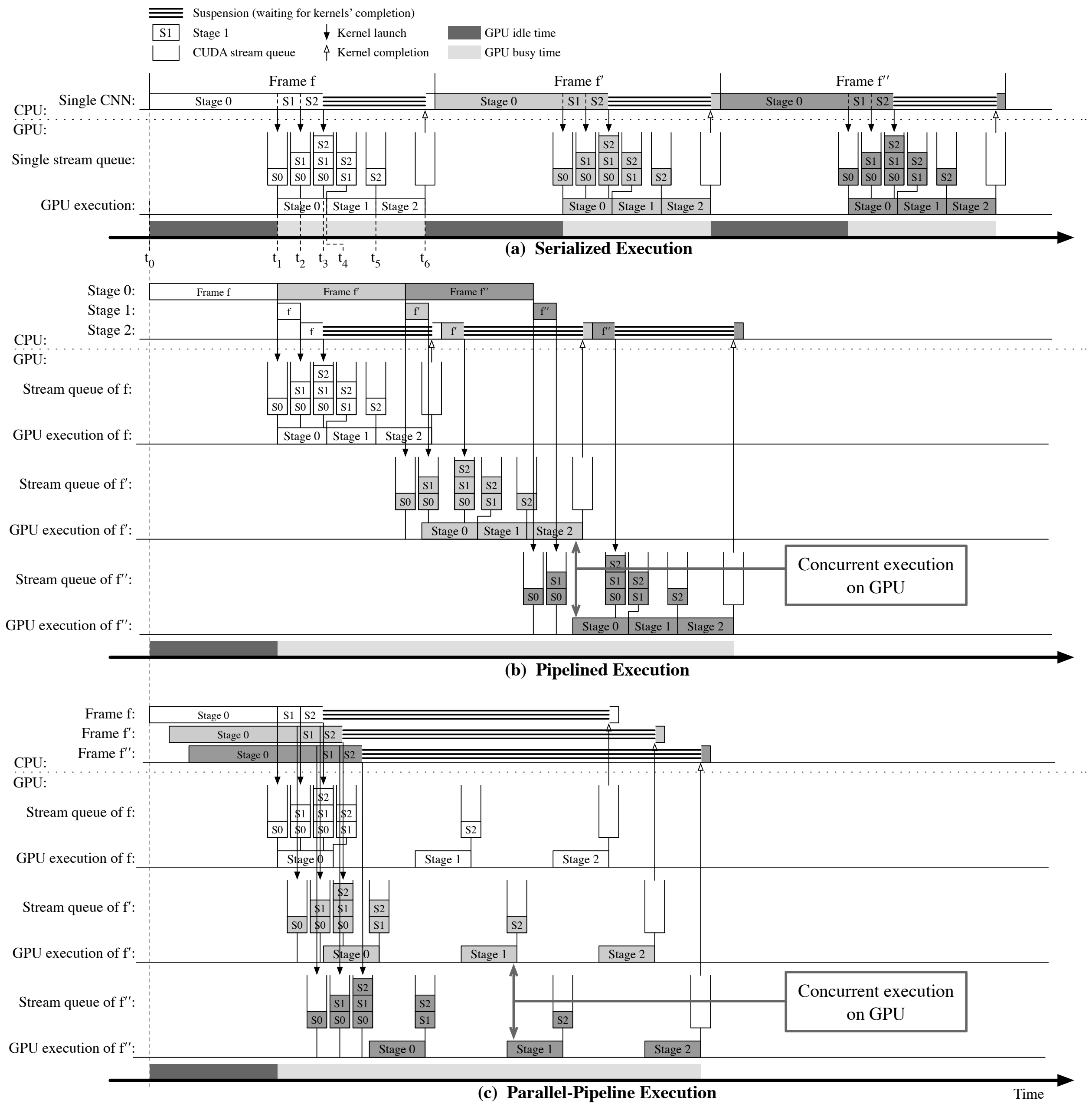


Figure 5.3: Scheduling of SERIAL, PIPELINE, and PARALLEL.

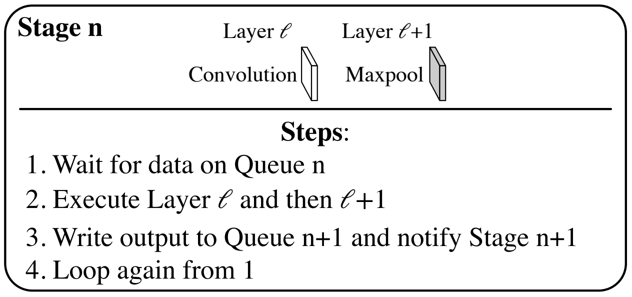

Stage n
Layer ℓ
Layer ℓ+1
Convolution
Maxpool
Steps:
1. Wait for data on Queue n
2. Execute Layer ℓ and then ℓ+1
3. Write output to Queue n+1 and notify Stage n+1
4. Loop again from 1

Figure 5.4: Stage representation for pipelined execution model.

an index from the previous stage to signal that stage's completion. Once this index becomes available, the thread starts processing by calling the original CNN layer functions with the appropriate bookkeeping data. After these layer functions return, the thread notifies the next stage of completion by pushing the index it just finished into the next queue. It then returns to waiting for input data again. *This approach requires no changes to the original YOLO layer functions.*

The resulting parallelism is illustrated in Figure 5.3 (b). For conciseness, we henceforth refer to this pipelined-execution variant of YOLO as simply "PIPELINE" and refer to the original unmodified YOLO as "SERIAL." In PIPELINE, multiple per-frame queues are created and pipeline stages can execute concurrently on CPU cores. This enables kernels from different frames to be queued and executed concurrently on the GPU when enough of its resources are available. One example of the concurrent GPU execution of kernels is noted in Figure 5.3 (b). The threads for stages are scheduled by the Linux scheduler `SCHED_NORMAL`. Threads may migrate between CPU cores or block when cores are not available. The concurrent executions of threads on the CPUs and kernels on the GPU should enable PIPELINE to provide higher throughput or lower latency. We evaluate this in Section 5.4. While the longer first stage causes some overall pipeline imbalance, we believe this is largely confined to the CPU. We further mitigate the influence of a pipeline imbalance by enabling thread parallelism within each stage as described next.

Parallel-pipeline execution. As seen in Figure 5.3, PIPELINE enables parallelism across stages but not within a stage. Given the independence properties previously mentioned, there is no reason why the same stage cannot be executed on different frames at the same time. To enable this, we consider a second parallel YOLO variant, called *parallel-pipeline execution*, or simply "PARALLEL" for brevity, that extends PIPELINE by allowing parallel executions of the same stage. The PARALLEL YOLO variant is realized by allocating a configurable number of "worker" threads in each stage, and a "manager" thread that is responsible for

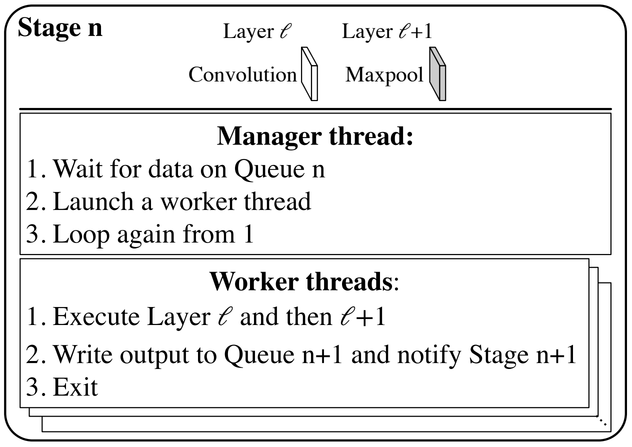

Stage n
Layer ℓ
Layer ℓ+1
Convolution
Maxpool
Manager thread:
1. Wait for data on Queue n
2. Launch a worker thread
3. Loop again from 1
Worker threads:
1. Execute Layer ℓ and then ℓ+1
2. Write output to Queue n+1 and notify Stage n+1
3. Exit

Figure 5.5: Stage representation for parallel-pipeline execution model.

dispatching worker threads. The worker threads can process different frames concurrently. This is illustrated in Figure 5.5 (once again) for a stage comprised of one convolution layer and one maxpool layer. The difference between PIPELINE and PARALLEL is illustrated in Figure 5.3 (c). In PARALLEL, more parallelism is enabled on both CPU cores and the GPU through allowing multiple threads of the same stage to execute in parallel. Recalling the scheduling rules revealed in Section 3.1, kernels are scheduled in order of their arrival times to the heads of their stream queues. In Figure 5.3 (c), these arrival times are such that kernel executions from different frames are interleaved, as shown.

With the scheduling shown in Figure 5.3, we can see that PIPELINE and PARALLEL achieve the purpose of fully utilizing the GPU resources (much less GPU idle time than SERIAL) as well as better utilizing the multiple CPU cores. We quantify the performance in Section 5.4.

5.3.2 Multi-GPU Execution

Recall from Section 5.1 that the PX2 has an iGPU and a dGPU with different capabilities. As seen in Figure 5.11, which we cover in detail later, the iGPU tends to execute layers substantially slower than the dGPU. However, notable exceptions (maxpool layers and the region layer) do exist, so the iGPU is still a useful computational resource. Given this observation, we enhanced our PIPELINE and PARALLEL implementations to allow a configurable distribution of stages across GPUs. While utilizing both GPUs may be beneficial, additional data-transfer and synchronization overheads can cause dual-GPU configurations to sometimes under-perform single-GPU ones. We analyze this trade-off experimentally in Section 5.4.2.

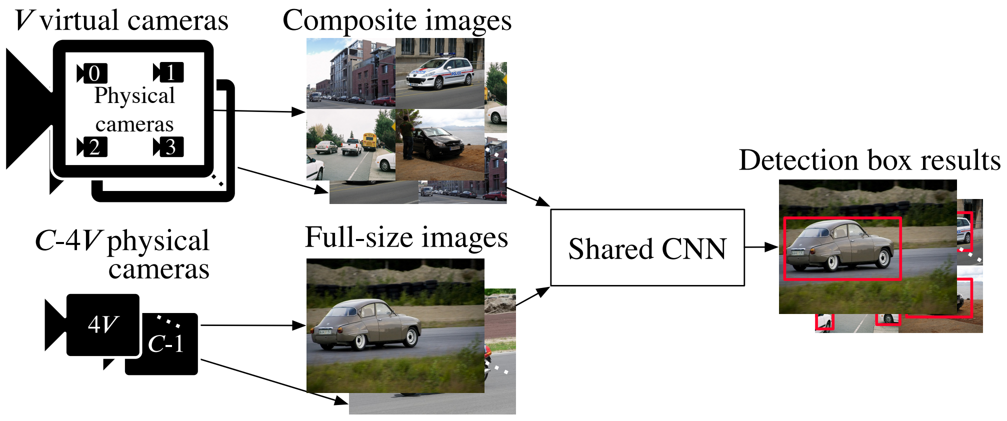


Figure 5.6: CNN extended for multi-camera composite images.

5.3.3 Accelerating Multiple Streams with Multi-Camera Composite Images

What if we could combine several physical cameras into a single "virtual" one, as illustrated in Figure 5.6, by combining independent per-camera images into a single composite image as illustrated? Such an approach could have a tremendous impact on throughput. For example, if all physical cameras were combined in groups of four, throughput could potentially be increased by fourfold. Whether this is acceptable would depend on how latency and accuracy are impacted.

To evaluate this approach, we added flexible support for compositing physical cameras to the YOLO variants already described. Note that these variants themselves are not impacted by this approach: the fact that the images to be processed are obtained by compositing does not change the processing to be done. Because of this, it is possible to have both composite and full images supported in the same system, as illustrated in Figure 5.6.

The compositing approach works by exploiting the flexibility of YOLO's neural network to process multiple frames as a single frame. This requires taking k^2 frames for some k, tiling them in a grid, and then passing that combined image through YOLO as a single input, as shown in Figure 5.6. *For best accuracy, this approach requires retraining the YOLO network on a set of suitably tiled images to produce new model weights.*

While compositing should yield a throughput improvement that is linear in the number of images in the grid, there are several additional factors to consider. First, frame arrival times from different cameras must be synchronized or the combining process will block waiting for all the images required to fill the grid. This could increase latency to the time between frames on the camera with the smallest frame rate. Second, this approach will reduce accuracy. Because our configuration of YOLO only accepts frames of 416 by 416 pixels, each frame in the grid must be downscaled by a factor proportional to the grid size. For detecting objects that are large relative to the frame size (*e.g.*, close vehicles), this compromise may be permissible, however that will not be true in all cases.

Additionally, we must stress that this approach only works for CNNs that reason locally. Using this approach in global-classification systems such as those used in scene understanding *will yield incorrect results*. While the YOLO authors believe their network learned to use global features, we did not observe this behavior in our analysis.

5.3.4 Avoiding Implicit Synchronization

The original Darknet framework uses the single default FIFO queue in CUDA for kernel requests. This works flawlessly for serialized execution by a single-threaded program, but falls apart when we want concurrent parallel-pipeline instances. Specifically, use of only the default queue causes destructive implicit synchronization on the GPU, which can significantly degrade performance. As suggested in Section 3.3, our approach mitigates that by using CUDA functions to create a unique FIFO queue for the kernel executions and memory transfers of each parallel-pipeline instance.

5.4 Evaluation

Our evaluation experiments were designed to answer a number of questions related to our challenge: How many cameras of differing frame rates (FPS) can be supported with the PX2 running a shared CNN? Can acceptable latency be maintained for each camera supported? How can we use the configuration options in PIPELINE and PARALLEL to increase the number of cameras supported? Are multi-camera composite images a viable option for supporting significant numbers of cameras?

The evaluation of multi-camera composite images must explicitly consider the effects on object detection/recognition accuracy. In all other evaluations of PIPELINE and PARALLEL using single-frame images, we verified that accuracy remained unchanged (as expected, since the CNN layer processing programs were unchanged).

We next describe aspects of the experimental methodology common to all experiments and then proceed to answer our questions. We first evaluate the frame rates enabled by the three basic execution models on single and multiple cameras. Then, we explore the effects of configurable options—stage granularity, the number of threads per stage, and the assignment of stages to GPUs. Finally, we explore the trade-off between throughput and accuracy with multi-camera composite images.

General experiment set-up. In each experiment, the CNN processes an aggregate of at least 10,000 frames. Latency is measured as the time difference between the arrival time of a frame at the first CNN layer and the time when the object-detection result for that image is ready. Because the system runs Linux and all its services, worst-case latency outliers can occur. Occasionally dropping a small number of frames when their processing time exceeds a latency threshold is acceptable for our automated-driving platform. Therefore, latencies are reported only for those in the 95^{th}-percentile of all measured values.

Unless otherwise stated, the default CNN configuration has one CNN layer per pipeline stage for both PIPELINE and PARALLEL and ten threads per stage for PARALLEL. In these default configurations, only the PX2's dGPU is used. The choice of these default configurations is explained as part of our configuration option evaluation. All the experiments were conducted on the NVIDIA Drive PX 2 with Linux kernel 4.9.38-rt25-tegra with the PREEMPT_RT patch, NVIDIA Driver 390.00, and CUDA 9.0, V9.0.225. CPU threads were scheduled under `SCHED_NORMAL`.

5.4.1 How Many Cameras can be Supported?

To answer this question we first consider processing one stream of images by the shared CNN. Cameras are characterized by their frame rates (FPS), and we emulate camera inputs to the CNN at rates common to cameras used in automated driving (10, 15, 30, 40 FPS). We also consider streams with much higher frame rates (up to 100 FPS) to represent aggregate frame rates from multiple cameras sharing the CNN. Our emulated camera is a program running on a separate thread that generates input frames to the shared CNN at a defined inter-frame interval corresponding to a given FPS (*e.g.*, 25 milliseconds for 40 FPS). Because latency is a critical metric, we present results as a plot of latency for different input camera rates and compare execution under SERIAL, PIPELINE, and PARALLEL.

PIPELINE supports 63 FPS, PARALLEL may support 71 FPS. Figure 5.7 shows the latency results. Our main criterion for supporting a frame rate is that latency does not become unbounded at that rate. Using this criterion, SERIAL (essentially the original YOLO) can support rates up to 28 FPS and PIPELINE can support up to 63 FPS. PARALLEL can also support 63 FPS with negligible latency reduction and up to 71 FPS

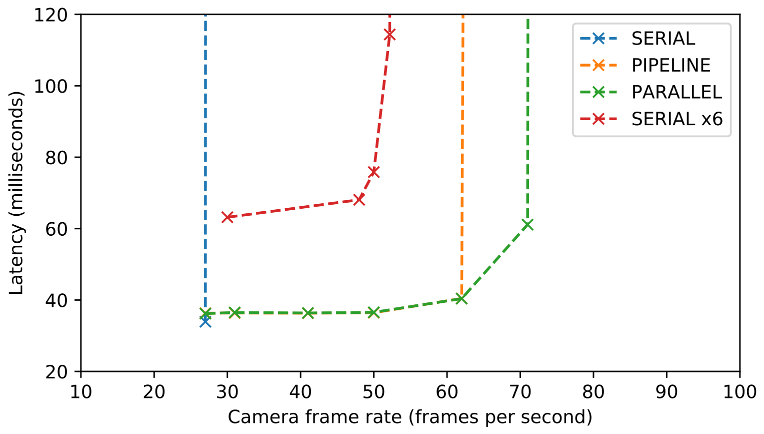


Figure 5.7: Latency of SERIAL, PIPELINE, and PARALLEL YOLO.

before latency becomes unbounded. Note that the latency at rates lower than 63 FPS is essentially constant indicating that 36 milliseconds is the minimum latency for YOLO (as seen on the *y*-axis of Figure 5.7). Latency becomes unbounded in any of these scenarios when either the GPU becomes overloaded or the CNN execution model cannot further utilize the available GPU capacity. The latency increase of PARALLEL at 71 FPS over PIPELINE at 63 FPS is 20 milliseconds. This may be acceptable in scenarios such as maneuvering in a parking lot. However, running multiple SERIAL instances (at most six due to memory constraints) can support at most 53 FPS with latency of 114 milliseconds before getting unbounded latency.

We attribute these results to the inter-stage GPU dispatch parallelism of PIPELINE and the additional intra-stage GPU dispatch parallelism of PARALLEL. These simultaneous dispatches allow us to more fully utilize the GPU and achieve a higher frame rate. Although some dispatch parallelism can be achieved by running multiple SERIAL instances, GPU resource usage is still limited because kernels from different process contexts cannot run concurrently in CUDA. As we mentioned above, concurrent kernel execution from different processes can be enabled with NVIDIA's MPS (Multi-Process Service), however, NVIDIA has not implemented this on the PX2.

These results are perhaps most useful as ways to estimate how many cameras with differing rates can share the CNN. As long as the aggregate rate from all cameras is less than the supported total rate, latency should not be compromised. For example, the results show that a single, shared SERIAL YOLO could not support a single 30 FPS camera on the PX2 hardware while PIPELINE and PARALLEL should easily support two 30 FPS cameras. Similarly, PIPELINE can support up to four 15 FPS cameras, or two at 15 FPS and one at 30 FPS, *etc*.

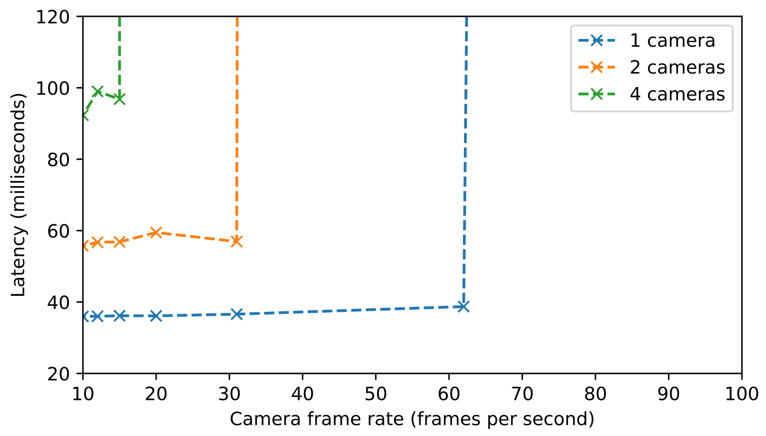


Figure 5.8: PARALLEL YOLO latency with different numbers of cameras.

Note that all these results are from using only one of the two independent computing systems on the PX2 board. Using both systems to run two independent CNN configurations should double the number of cameras supported by one PX2.

Multiple cameras are supported for aggregate frame rates. We verified that estimations based on the FPS of a single image stream are valid for multiple cameras. This experiment investigated whether the additional competition from multiple input streams to the shared CNN changed latencies. The results are shown in Figure 5.8 for one, two, and four cameras generating input to PARALLEL YOLO. The supported total camera frame rate is determined by the hardware capacity and the CNN framework execution model, thus it is constant with multiple cameras instead of one. That is, this data corroborates the estimate given above that supporting one 60 FPS camera is the equivalent of supporting two 30 FPS cameras, four 15 FPS cameras, *etc*. The increased latencies for two and four cameras seen on the y-axis in Figure 5.8 are an artifact of the experiment. The input images are processed in sequence, so longer latency measurements will be incurred when a frame starts being processed after frames that arrive at the same time from other cameras. These in-phase frame arrivals can be avoided in a real system by setting a different phasing for each camera (a topic for future work).

5.4.2 Can Our CNN Implementations Be Better Configured?

PIPELINE can be configured with multiple CNN layers aggregated into the pipeline stages, *i.e.*, configuring the granularity of each stage. PARALLEL can be configured in the same way, but also adds the option to adjust the number of threads assigned per stage, *i.e.*, configure the degree of parallelism within each stage. In

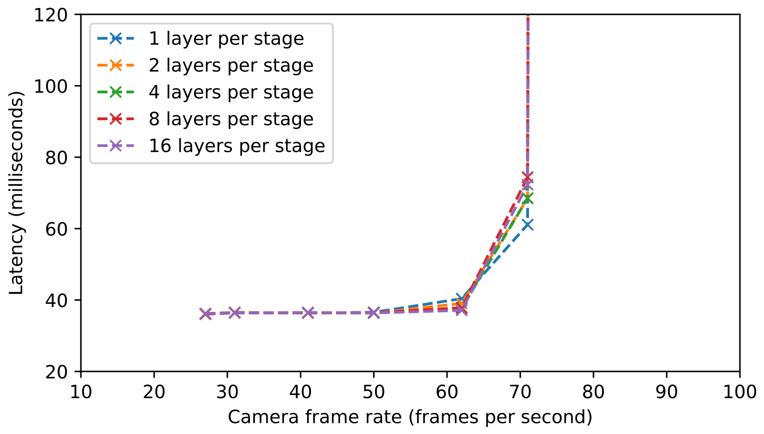


Figure 5.9: PARALLEL YOLO latency with different granularities (number of layers per stage).

addition, sequences of stages can be assigned to either the iGPU or the dGPU on the PX2 to avoid wasting any available GPU capacity. We evaluate these configuration options next.

Granularity of stages improves latency under heavy loads for PARALLEL. In Figure 5.9, we compare the latency results at frame rates from a single camera stream for PARALLEL with different stage granularities. We did not observe any impact of granularities on PIPELINE, so those results are not shown here.

Note that pipeline is effectively disabled by configuring a single stage containing all 16 layers. This approximates multiple instances of SERIAL that are running in one process context, a configuration not supported in the original framework.

As shown in Figure 5.9, latencies of different granularities up to a frame rate at most 63 FPS (the maximum load before latencies increase significantly, *e.g.*, the max steady state) are very close. While all the granularities can support a heavier processing load up to 71 FPS with an increase in latency, the finer-granularity stages have smaller latencies in this region. Because each stage is implemented by one manager thread and multiple worker threads, more stages with finer granularity enable more parallelism among stages on the CPU. Our design for data communication between stages introduces small overheads, which is reflected in the very small changes in the performance of different granularities when the frame rate is low. *A granularity of one layer per stage is a good configuration choice.*

More threads for PARALLEL improves latency under heavy loads. In Figure 5.10, we compare the latency results at frame rates from a single camera stream for PARALLEL with different numbers of threads. More parallelism is enabled with more threads per stage, so more frames can be processed in parallel by a stage. Having only one thread per stage is essentially the same as pipelined execution, so latency becomes

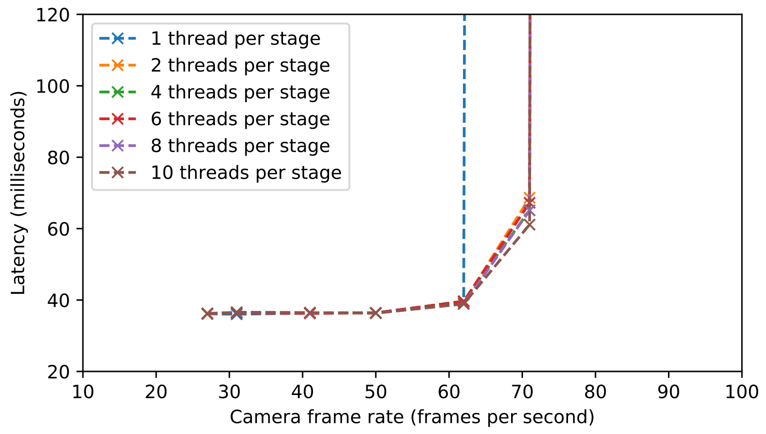


Figure 5.10: PARALLEL YOLO latency with different numbers of threads.

unbounded at 63 FPS as shown previously. Between 63 FPS and 71 FPS the latencies increase substantially but higher numbers of threads have somewhat lower latencies and a slower rate of latency increase with load. *A larger number of threads, 10 or so, is a good configuration choice.*

Overhead analysis and performance bottleneck. Enabling PIPELINE and PARALLEL certainly causes additional overheads. We evaluate the computational and spatial overheads with different configurations by comparing the CPU and main memory usage between SERIAL and PIPELINE or PARALLEL. In each experiment, process or thread instances share the six active CPU cores on the PX2. The maximum CPU capacity is 600%. As shown in Table 5.2, a single instance of SERIAL uses 92% of the CPU capacity, essentially one core's worth. Six instances of SERIAL use 536%, most of the available CPU resources. PIPELINE with one thread per stage uses 219% of the CPU capacity and 1,132 MB of memory. Compared to one SERIAL instance, these figures represent an increase of 127% in CPU capacity and 358 MB with respect to memory. These increases reflect added overheads for synchronization, scheduling, and data buffering. Moving to a PARALLEL implementation with ten threads per stage causes only a slight increase in CPU usage, from 219% to 239%, and a negligible increase in memory usage. We have shown in Section 5.4.1 that a significant increase in the supported frame rate is achieved with the PARALLEL and PIPELINE configurations compared to both one and six SERIAL instances. Under SERIAL, the GPU is around 60% utilized, while under PARALLEL and PIPELINE, it is nearly 100% utilized, shown from CUDA profiling results that are similar to Figure 5.3. This demonstrates that using more CPU resources in order to fully utilize the scarce resources of the GPU is a reasonable strategy.

	CPUs (%)	Memory (MB)
SERIAL	92	774
SERIAL x6	536	4,644
PIPELINE (single thread)	219	1,132
PARALLEL (10 threads)	239	1,136

Table 5.2: CPU and memory usage.

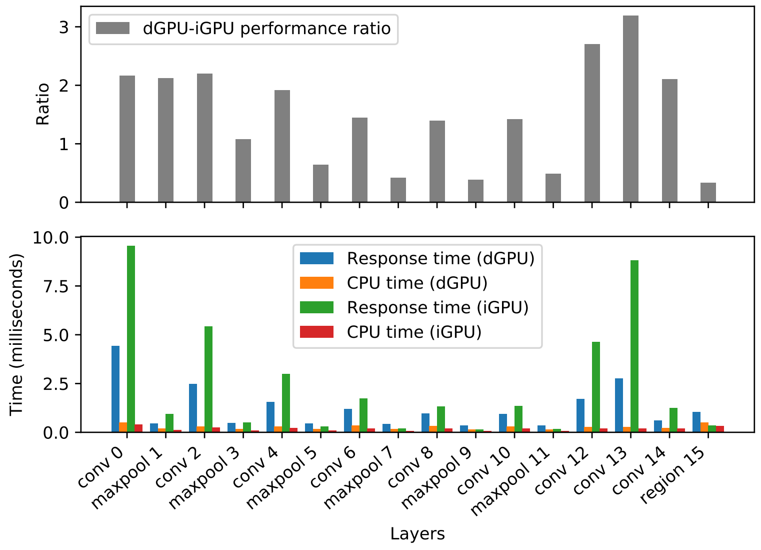


Figure 5.11: Per-layer execution times.

Multi-GPU execution improves latency under heavy loads We evaluate the latency performance of different GPU assignment strategies next to understand the impact of assigning some layers to the iGPU.

Layer performance on both GPUs. We first measured the performance of each layer on both GPUs. We executed SERIAL using each GPU individually and measured per-layer response times and per-layer CPU computation times. Results are shown in Figure 5.11, which consists of two parts that use a common *x*-axis; the top part depicts per-layer dGPU-iGPU performance ratios (obtained by dividing a layer's iGPU response time by its dGPU response time); the bottom part depicts the per-layer data just mentioned. As seen, convolution layers typically take longer on the iGPU, some two to three times longer. Convolution layers contain heavy matrix computations that are completed much faster with more CUDA cores. The maxpool and region layers have lower performance ratios because they have lighter computation loads. *These layers are good choices to be moved to the iGPU.*

GPU assignment strategy. To introduce as little latency degradation as possible while using both GPUs, our strategy is to move layers with low dGPU-iGPU performance ratios to the iGPU. Given the performance ratios at the top of Figure 5.11, we evaluate the GPU assignment strategies shown in Table 5.3. Each strategy

Strategy ID	dGPU	iGPU
S0	All	$\emptyset$
S1	$\{0,1,...,13\}$	$\{14,15\}$
S2	$\{0,1,...,4,12,13,14,15\}$	$\{5,6,...,11\}$
S3	$\{0,1,...,4,12,13\}$	$\{5,6,...,11,14,15\}$
S4	$\{0,1,...,7\}$	$\{8,9,...,15\}$

Table 5.3: GPU assignment strategies.

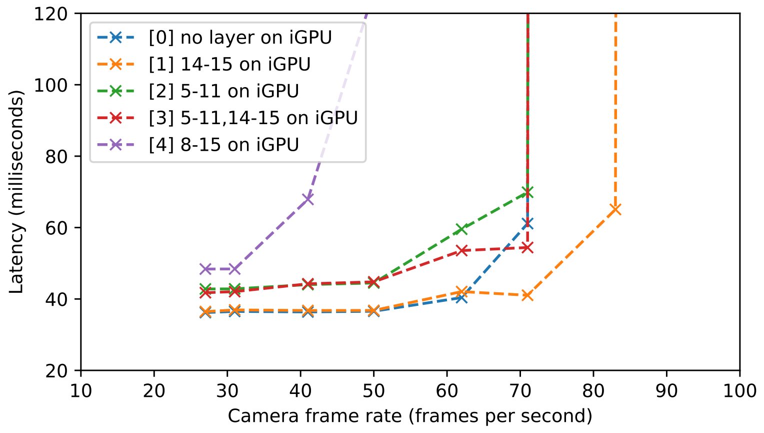


Figure 5.12: PARALLEL YOLO latency with different GPU assignments.

is characterized by the sequence of layer numbers assigned to each GPU. For example, strategy S1 assigns layers 0 to 13 to the dGPU and layers 14 and 15 to the iGPU.

Strategy S1 substantially reduces latency for PARALLEL under heavy loads. Figure 5.12 shows that, for PARALLEL, moving the last two CNN layers to the iGPU under strategy S1 improves latency. S1 provides support for a frame rate of 71 FPS with latency only slightly greater than a frame rate of 63 FPS when using only the dGPU. At a frame rate of 80 FPS, the latency is approximately the same as a frame rate of 71 FPS using only the dGPU. All other strategies considered either increase latencies or have negligible effect.

Strategies S1 and S3 provide some latency reduction for PIPELINE under heavy loads. In Figure 5.13, S1 provides support for a frame rate of 71 FPS in PIPELINE with latency only slightly greater than a frame rate of 63 FPS using only the dGPU. S3 for PIPELINE supports a frame rate of 71 FPS with reduced latency comparable to that for PARALLEL using only the dGPU. All other strategies considered either increase latencies or have negligible effect.

5.4.3 Are Multi-Camera Composite Images Effective?

Any combination of the features and execution models above fails to provide enough throughput at desired latencies for a vehicle equipped with more than four cameras at a frame rate of 30 FPS using both

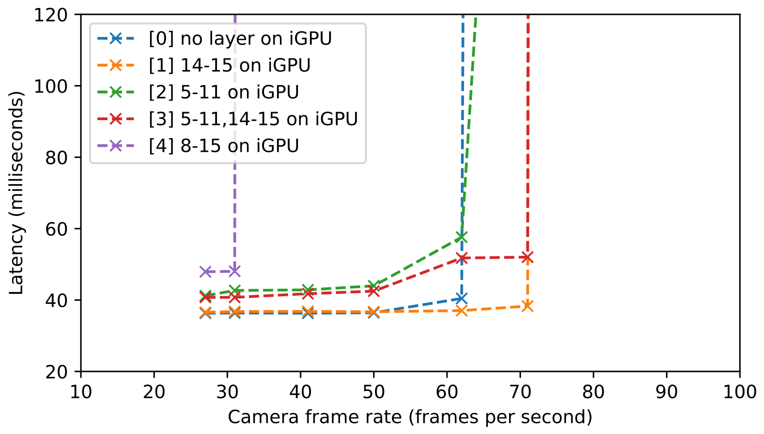


Figure 5.13: PIPELINE YOLO latency with different GPU assignments.

systems on the PX2 board. We propose to tackle this by leveraging the technique of compositing frames from multiple cameras into one image for processing. Although compositing might introduce a small loss of accuracy as the image resolution will be reduced, this can be acceptable for low-criticality cameras in certain circumstances. We devote this section to evaluate the effectiveness of the technique.

With a multi-camera composite, the throughput is multiplied by the number of images in the composite image. However, the frame concatenation introduces overhead. Also, because the frames are down-sampled as explained in Section 5.3.3, the object detection accuracy could be reduced. We evaluate the throughput, latency, and accuracy of the multi-camera composite image technique.

Experimental set-up. We focus on comparing YOLO's performance on full-size images and 2x2 composite images. The network is evaluated on a standard object-detection dataset called PASCAL VOC (Everingham *et al.*, 2015), using two types of tests: one of full-size images and one of composite images. For the full-size test, the original VOC test data is used. For the composite, we generate test images by randomly choosing four full-size VOC test images and combining them into a composite image with a 2x2 grid. Each VOC test image is used exactly once as a component of a composite image, hence, the size of the composite test set is four times smaller than the full-size one. This ensures that the total numbers of objects to be detected are the same in both tests. At runtime, full-size and composite images are both resized by YOLO to be 416x416 during processing.

Evaluation of detection results. Evaluation of the network's output is done using *mean average precision* (mAP)—a metric commonly used to evaluate object-detection algorithms (Everingham *et al.*, 2010). In loose terms, mAP characterizes a model's ability to have high recall and high precision. In other words, higher

mAP	Full-size Test	Composite Test
YOLO	55.63	36.46
Composite YOLO	57.64	48.29

Table 5.4: mAPs for full-size and composite tests with PASCAL VOC dataset.

mAP	Full-size Test	Composite Test
YOLO	63.66	44.91
Composite YOLO	66.20	56.21

Table 5.5: mAPs of object classes relevant to autonomous driving.[6]

mAP means more objects are detected and the detections and localizations are more accurate. A detailed explanation of mAP has been provided in Section 2.2.2.

Similar to other state-of-the-art detection networks, YOLO is designed to be scale-invariant, meaning it can detect objects at different scales and resolutions. However, when image resolutions decrease significantly (in our case, by a factor of four), the network's performance can start to degrade. As seen in Table 5.4, with a 2x2 grid setup, YOLO trained on the original VOC data can only perform at 36.46% mAP when inferring composite images, an almost 20% accuracy drop.

To mitigate this, we retrained the network on the VOC dataset using the same number of training images for both full-size and composite data. We refer to this retrained network as Composite YOLO, and the resulting mAPs are shown in Table 5.4. Moreover, since our primary applications are to autonomous driving, we also show in Table 5.5 the detection results for relevant object classes.[6] Overall, our results consistently indicate that with YOLO retrained on both composite and full-size data, the detection accuracy of composite images is improved without reducing that of full-size images. With the retrained composite network, we can provide a trade-off between a 10% accuracy drop and a fourfold increase of throughput.

Although the mAP decreases with composite images, it is important to remember the following. First, the focus of our work is on providing the option to trade-off between higher throughput and acceptably lower accuracy, and not on the magnitude of the accuracy itself. Second, YOLO represents a state-of-the-art detection network, but we are tackling a very challenging computer-vision problem with strictly limited resources. Third, real-life autonomous driving systems usually deploy tracking algorithms, which can use information of multiple frames to improve detection accuracy. Finally, as shown in (Redmon and Farhadi,

[6]Among 20 object classes, we deem the following eight to be irrelevant to the task of autonomous driving: airplane, boat, bottle, chair, dining table, potted plant, sofa, and TV monitor.

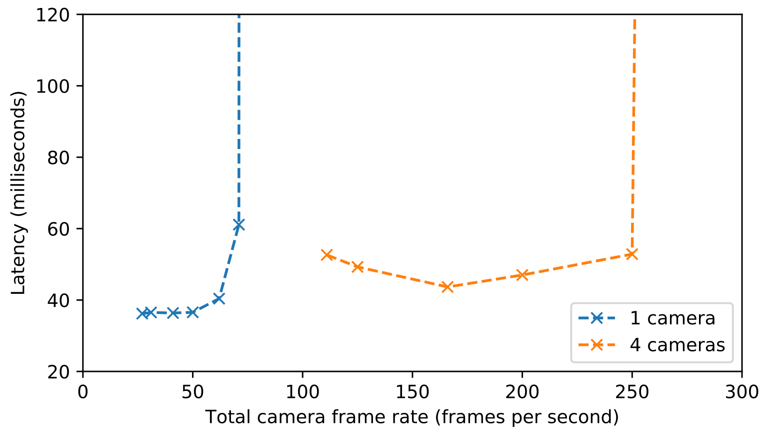


Figure 5.14: PARALLEL YOLO latency with multi-camera-image processing and different numbers of cameras.

2017), YOLO tends to perform better for bigger objects than for smaller ones. Thus, we expect the mAPs to be higher for nearby objects, which are usually prioritized in the case of automated driving.

A four-camera composite image supports 250 FPS (!) and a latency of 48 milliseconds. This is shown in Figure 5.14. Examining the *y*-axis Figure 5.14, observe that for lower respective frame rates, latency is higher in the four-camera scenario than in the one-camera scenario (about 45 vs. 36 milliseconds). This is due to the additional overhead incurred to combine multiple frames.

5.5 Chapter Summary

In this chapter, we presented the results of an industrial design study that addresses the challenge of re-designing CNN software to support multiple cameras with better throughput for automated-driving systems. We showed how to transform a traditional CNN framework to obtain pipelined and parallel-pipelined variants to better utilize hardware resources. We also examined the impact of properly load balancing GPU work across the GPUs of a multi-GPU hardware platform. These basic techniques resulted in an improvement in throughput of over twofold with only minor latency increases and no accuracy loss. In order to accommodate more cameras that these basic techniques allow, we proposed the technique of compositing multiple image streams from different cameras into a single stream. We showed that this technique can greatly boost throughput (up to 250 FPS) at the expense of some accuracy loss. We demonstrated that this accuracy loss can be partially mitigated through network retraining.

In the next chapter, we will apply more fine-grained graph scheduling to a computer-vision standard OpenVX and apply our schedulability analysis provided in Chapter 4 to derive end-to-end response-time bounds for GPU-using applications.

Acknowledgement. Most of the study described in this chapter was conducted during my internship with Shige Wang at General Motors. Thanh Vu conducted the retraining of the YOLO model with assistance from Cheng-Yang Fu and Akash Bapat.

CHAPTER 6: CASE STUDY[1]

In the preceding chapters, we developed theoretical analysis to derive end-to-end response-time bounds of GPU-using applications (in Chapter 4) and proposed practical techniques for improving throughput performance of a CNN-based computer-vision workload (in Chapter 5). In this chapter, we apply our analysis to a more fine-grained graph scheduling of a computer-vision standard, OpenVX (Khronos Group, 2020), and evaluate both analytical and empirical real-time performance of a real-world application.

To facilitate the development of computer-vision techniques, the Khronos Group has put forth a ratified standard called OpenVX. Although initially released only six years ago, OpenVX has quickly emerged as the computer-vision API of choice for real-time embedded systems, which are the standard's intended focus. Under OpenVX, computer-vision computations are represented as directed graphs, in which graph nodes represent high-level computer-vision functions, and graph edges represent precedence and data dependencies across functions. OpenVX can be applied across a diversity of hardware platforms. In our work, we consider its use on platforms where graphics processing units (GPUs) are used to accelerate computer-vision processing.

Unfortunately, OpenVX's alleged real-time focus reveals a disconnect between computer-vision researchers and the needs of the real-time applications where their work would be applied. In particular, OpenVX lacks concepts relevant to real-time analysis, such as priorities and graph invocation rates, so it is debatable as to whether it really does target real-time systems. More troublingly, OpenVX implicitly treats entire graphs as monolithic schedulable entities. This inhibits parallelism[2] and can result in significant processing-capacity loss in settings (like autonomous vehicles) where many computations must be multiplexed onto a common hardware platform.

[1]Contents of this chapter previously appeared in the following paper:

Yang, M., Amert, T., Yang, K., Otterness, N., Anderson, J. H., Smith, F. D., and Wang, S. (2018a). Making OpenVX really "real time". In *Proceedings of the 39th IEEE International Real-Time Systems Symposium*, pages 80–93.

[2]As discussed below, a recently proposed extension (Khronos Group, 2019a) enables more parallelism, but this extension is directed at throughput, not real-time predictability, and is not available in any current OpenVX implementation.

In prior work, our research group partially addressed these issues by proposing a new OpenVX variant in which individual graph nodes are treated as schedulable entities (Elliott *et al.*, 2015; Yang *et al.*, 2015). This variant allows greater parallelism and enables end-to-end graph response-time bounds to be computed. However, graph nodes remain as high-level computer-vision functions, which is problematic for (at least) two reasons. First, these high-level nodes still execute sequentially, so some parallelism is still potentially inhibited. Second, such a node will typically involve executing on both a CPU and a GPU. When a node accesses a GPU, it suspends from its assigned CPU. Suspensions are notoriously difficult to handle in schedulability analysis without inducing significant capacity loss.

In this chapter, we show that these problems can be addressed through more fine-grained scheduling of OpenVX graphs.

First, we show how to transform the *coarse-grained* OpenVX graphs proposed in our group's prior work (Elliott *et al.*, 2015; Yang *et al.*, 2015) to *fine-grained* variants in which each node accesses either a CPU or a GPU (but not both). Such transformations eliminate suspension-related analysis difficulties at the expense of (minor) overheads caused by the need to manage data sharing. Instead, the analysis in Chapter 4 can apply to our fine-grained OpenVX graphs to determine end-to-end graph response-time bounds.

Moreover, our transformation process exposes new potential parallelism at many levels. For example, because we decompose a coarse-grained OpenVX node into finer-grained schedulable entities, portions of such a node can now execute in parallel. Also, we allow not only successive invocations of the same graph to execute in parallel but even successive invocations of the same (fine-grained) *node*.

Next, we present the results of case-study experiments conducted to assess the efficacy of our fine-grained graph-scheduling approach. In these experiments, we considered six instances of an OpenVX-implemented computer-vision application called HOG (histogram of oriented gradients), which is used in pedestrian detection, as scheduled on a multicore+GPU platform. These instances reflect a scenario where multiple camera feeds must be supported. We compared both analytical response-time bounds and observed response times for HOG under coarse- vs. fine-grained graph scheduling. We found that bounded response times could be guaranteed for all six camera feeds only under fine-grained scheduling. In fact, under coarse-grained scheduling, just one camera could (barely) be supported. We also found that observed response times were substantially lower under fine-grained scheduling. Additionally, we found that the overhead introduced by converting from a coarse-grained graph to a fine-grained one had modest impact. These results demonstrate the importance of enabling fine-grained scheduling in OpenVX if real time is *really* a first-class concern.

6.1 OpenVX Graph Scheduling

In this section, we show how to transform coarse-grained OpenVX graphs to fine-grained variants.

6.1.1 Coarse-Grained OpenVX Graph Scheduling

In prior work (Elliott *et al.*, 2015; Yang *et al.*, 2015), the coarse-grained graph scheduling techniques *without* intra-task parallelism, are proposed for scheduling acyclic[3] OpenVX graphs using G-EDF,[4] with graph nodes implicitly defined by high-level OpenVX computer-vision functions. We call OpenVX graphs so scheduled *coarse-grained* graphs.

Given the nature of high-level computer-vision functions, the nodes of a coarse-grained graph will typically involve executing both CPU code and GPU code. Executing GPU code can introduce *task suspensions*, and under G-EDF schedulability analysis, suspensions are typically dealt with using *suspension-oblivious analysis* (Brandenburg and Anderson, 2010). This entails analytically viewing suspension time as CPU computation time and can result in significant processing-capacity loss.

6.1.2 Fine-Grained OpenVX Graph Scheduling

In this section, we propose a fine-grained scheduling approach for OpenVX graphs obtained by applying four techniques. First, to eliminate suspension-based capacity loss, we treat CPU code and GPU code as separate graph nodes. Second, to reduce response-time bounds, we allow intra-task parallelism. Third, to avoid non-work-conserving behavior and enable better observed response times, we allow early releasing. Finally, we use a scheduler (namely, G-FL—see below) that offers advantages over G-EDF. We elaborate on these techniques in turn below.

DAG nodes as CPU or GPU nodes. To handle data dependencies, CUDA provides a set of synchronization functions (Section 3.3), which are configured on a per-device basis to wait via spinning or suspending. In this dissertation, we consider only waiting by suspending because the kernel executions in the workloads of interest are too long for spinning to be viable. In our fine-grained scheduling approach, we avoid suspension-related capacity loss due to kernel executions by more finely decomposing an OpenVX graph so that each of

[3]As described in these prior works, cycles can be dealt with by relaxing graph constraints or by combining certain nodes into "super-nodes." Adapting these techniques to our context is beyond the scope of this dissertation.

[4]While G-EDF was the focus of (Yang *et al.*, 2015), in experiments presented in (Elliott *et al.*, 2015), real-time work was limited to execute on one socket of a multi-socket machine and thus was only globally scheduled within a socket.

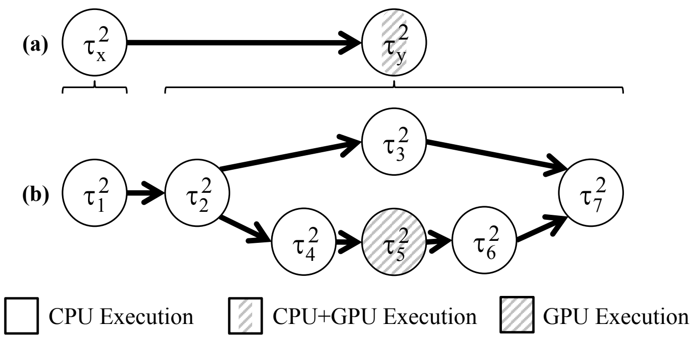


Figure 6.1: **(a)** Coarse- and **(b)** fine-grained representations of the same DAG G^2. τ_x^2 is simple sequential CPU code, so it is represented by one fine-grained task. τ_y^2 is more complex and consists of both CPU and GPU parts, some of which can execute in parallel.

its nodes is either a CPU node or a GPU node that executes a kernel. Additionally, we distinguish between regular CPU nodes and the necessary CPU work to launch a GPU kernel and await its results. In this chapter, we assume that copy operations are included in CPU nodes. In the workloads of interest to us, copies are short, so any resulting suspension-based capacity loss is minor. More lengthy copies could instead be handled as separate nodes, similarly to how we handle kernels.

In the rest of this section, we use a continuing example to illustrate various nuances of our fine-grained approach.

Example 6.1. Figure 6.1 (a) depicts a simple coarse-grained graph comprised of two tasks, τ_x^2 and τ_y^2. Figure 6.1 (b) shows a fine-grained representation of this same graph. Task τ_x^2 is a simple CPU-only computer-vision function and is represented by one fine-grained CPU task, τ_1^2. τ_y^2 is more complex, and its fine-grained representation consists of six tasks, $\tau_2^2, \cdots, \tau_7^2$, where τ_5^2 is a GPU task, and τ_4^2 and τ_6^2 are CPU tasks that launch the GPU kernel and await its completion, respectively.[5] $\Diamond$

An end-to-end response-time bound for a fine-grained graph can be obtained from per-node bounds as discussed in Section 4.1, with the copy operations in CPU nodes dealt with using suspension-oblivious analysis. For GPU nodes, we provided a new analysis for NVIDIA GPUs in Section 4.2.[6] Note that, in work on the prior coarse-grained approach (Elliott *et al.*, 2015; Yang *et al.*, 2015), locking protocols were used to preclude concurrent GPU access, obviating the need for such analysis.

[5]The synchronization call to await results may be launched before the GPU kernel has completed, but this overhead is extremely short.

[6]Response times for copies, if handled as separate nodes, are trivial to bound because they are FIFO-scheduled.

Example 6.2 (cont'd). Possible schedules for the graphs in Figure 6.1 are depicted in Figure 6.2. As in Example 4.1, successive jobs of the same task are shaded differently to make them easier to distinguish. Recall from Section 4.1 that all tasks in a graph share the same period; in these schedules, all periods are 5 time units, shown as the time between successive job release times (up arrows).

Figure 6.2 (a) depicts the graph's schedule as a single monolithic entity, as implied by the OpenVX standard. OpenVX lacks any notion of real-time deadlines or phases, so these are excluded here, as is a response-time bound. The depicted schedule is a bit optimistic because the competing workload does not prevent the graph from being scheduled continuously. Under monolithic scheduling, the entire graph must complete before a new invocation can begin. As a result, the second invocation does not finish until just before time 28.

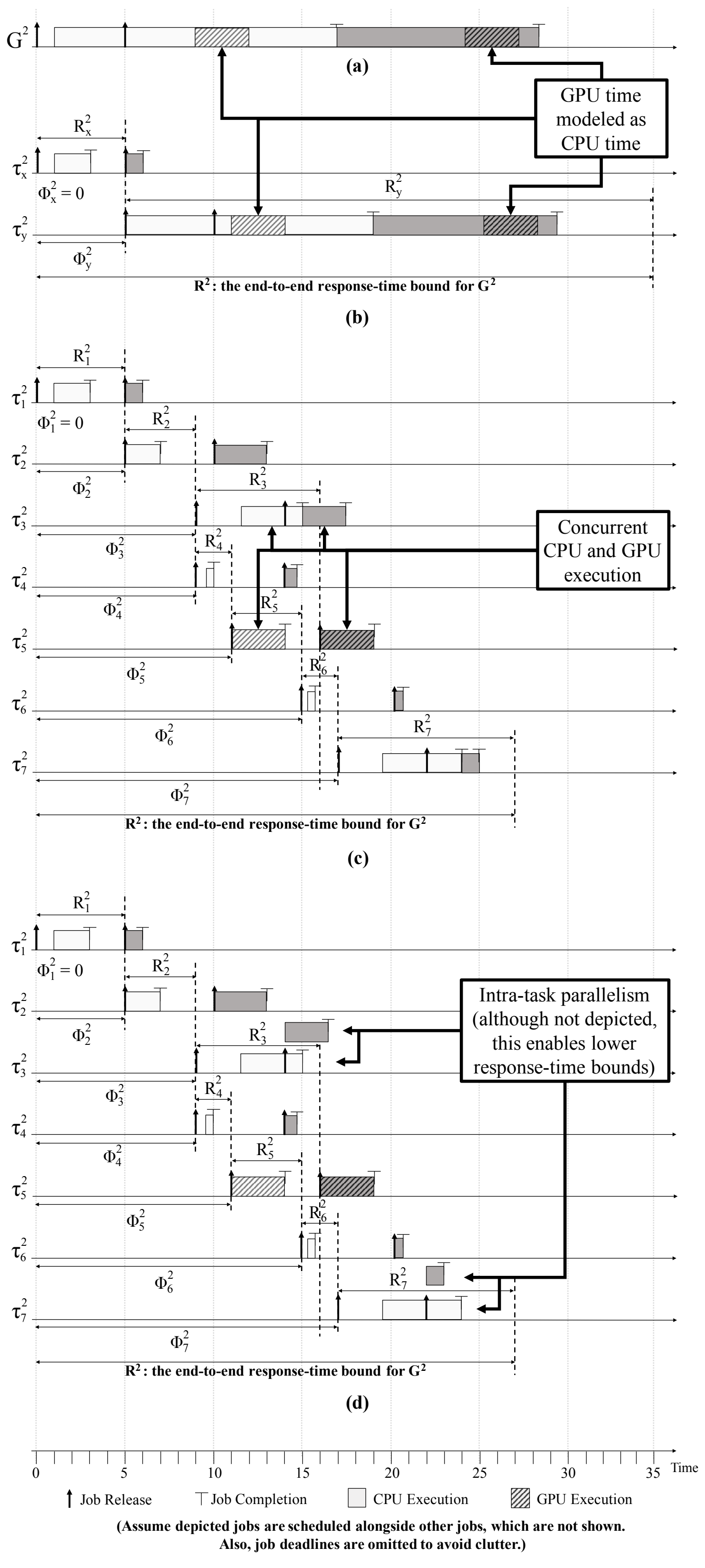
Figure 6.2 (b) depicts coarse-grained scheduling as proposed in prior work (Elliott *et al.*, 2015; Yang *et al.*, 2015), where graph nodes correspond to high-level computer-vision functions, as in Figure 6.1 (a). Nodes can execute in parallel. For example, $\tau^2_{y,1}$ and $\tau^2_{x,2}$ do so in the interval $[5,6)$. However, intra-task parallelism is not allowed: $\tau^2_{y,2}$ cannot begin until $\tau^2_{y,1}$ completes, even though its predecessor ($\tau^2_{x,2}$) is finished. Note that, under coarse-grained scheduling, GPU execution time is also analytically viewed is CPU execution time using suspension-oblivious analysis. This analytical impact is not represented in the figure.

Figure 6.2 (c) depicts a fine-grained schedule for the graph in Figure 6.1 (b), but without intra-task parallelism. In comparing insets (b) and (c), the difference is that nodes are now more fine-grained, enabling greater concurrency. As a result, $\tau^2_{7,2}$ completes earlier, at time 25. The detriments of suspension-oblivious analysis for GPU kernels are also now avoided. $\Diamond$

Intra-task parallelism. Our notion of fine-grained graph scheduling allows intra-task parallelism, *i.e.*, consecutive jobs of the same task may execute in parallel. Such parallelism can cause successive invocations of the same graph to complete out of order. This can be rectified via buffering.[7]

Example 6.2 (cont'd). Lower response times are enabled by intra-task parallelism, as depicted in Figure 6.2 (d). In this schedule, $\tau^2_{3,1}$ and $\tau^2_{7,1}$ are able to execute in parallel with $\tau^2_{3,2}$ and $\tau^2_{7,2}$, respectively, reducing the completion time of $\tau^2_{7,2}$ to time 23. Observe that $\tau^2_{7,2}$ completes before $\tau^2_{7,1}$, so some output buffering would be needed here. $\Diamond$

[7]The buffer size can be determined based on the calculated response-time bounds.



G^2
(a)
GPU time modeled as CPU time
R_x^2
τ_x^2
$\Phi_x^2 = 0$
R_y^2
τ_y^2
Φ_y^2
R^2: the end-to-end response-time bound for G^2
(b)
R_1^2
τ_1^2
$\Phi_1^2 = 0$
R_2^2
τ_2^2
Φ_2^2
R_3^2
τ_3^2
Concurrent CPU and GPU execution
Φ_3^2
R_4^2
τ_4^2
Φ_4^2
R_5^2
τ_5^2
Φ_5^2
R_6^2
τ_6^2
Φ_6^2
R_7^2
τ_7^2
Φ_7^2
R^2: the end-to-end response-time bound for G^2
(c)
Intra-task parallelism (although not depicted, this enables lower response-time bounds)
(d)
0
5
10
15
20
25
30
35
Time
Job Release
Job Completion
CPU Execution
GPU Execution
(Assume depicted jobs are scheduled alongside other jobs, which are not shown. Also, job deadlines are omitted to avoid clutter.)

Figure 6.2: Example schedules of the tasks corresponding to the DAG-based tasks in G^2 in Figure 6.1. **(a)** Monolithic scheduling. **(b)** Coarse-grained scheduling as in prior work. Fine-grained scheduling as proposed here **(c)** without and **(d)** with intra-task parallelism.

Allowing intra-task parallelism has an even greater impact on *analytically derived* response-time bounds (Erickson and Anderson, 2011), and as noted earlier (in Section 4.1, enables task utilizations to exceed 1.0.

Early releasing. Although omitted in Figure 6.2 for clarity, early releasing can decrease *observed* response times without affecting analytical bounds.

Example 6.2 (cont'd). Allowing $\tau_{7,1}^{2}$ to be early released once $\tau_{6,1}^{2}$ completes in Figure 6.2 (d) reduces the overall graph's completion time to just under 23 time units. ◊

G-FL scheduling. The approach in Section 4.1 applies to any GEL scheduler. As shown in (Erickson and Anderson, 2012; Erickson *et al.*, 2014), the global fair-lateness (G-FL) scheduler is the "best" GEL scheduler with respect to tardiness bounds. We therefore perform CPU scheduling using G-FL instead of G-EDF.[8]

Periods. An additional benefit of fine-grained scheduling is that it allows for shorter periods.

Example 6.2 (cont'd). The period used in Figure 6.2 seems reasonable in insets (c) and (d): notice that each job finishes before or close to its task's next job release. In contrast, in insets (a) and (b), response times could easily be unbounded. ◊

Recently proposed OpenVX extensions. The Khronos Group recently released the *OpenVX Graph Pipelining, Streaming, and Batch Processing Extension* (Khronos Group, 2019a), which enables greater parallelism in OpenVX graph executions. However, this extension is not available in any current OpenVX implementation and still lacks concepts necessary for ensuring real-time schedulability. While we have not specifically targeted this extension, an ancillary contribution of our work is to provide these needed concepts. In particular, the parallelism enabled by this extension's pipelining feature is actually subsumed by that allowed in our fine-grained graphs. Furthermore, the batching feature allows a node to process multiple frames instead of just one, potentially increasing computation cost; this could increase the node's utilization, possibly even exceeding 1.0. Introducing intra-task parallelism as we have done enables such nodes to be supported while still ensuring schedulability.

[8]As explained in Section 6.2, we actually consider two variants of G-FL, a "clustered" variant in which G-FL is applied on a per-socket basis on our test platform, and a fully global variant that is applied across all sockets.

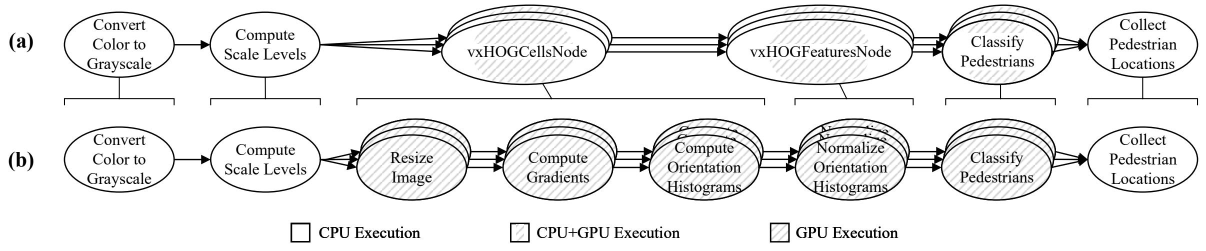


Figure 6.3: **(a)** Monolithic/coarse-grained and **(b)** fine-grained HOG DAGs. Our experiments used 13 scale levels; three are shown here. In (b), there are also two CPU nodes per kernel that launch that kernel and await its results. These nodes have short execution times and are omitted from much of our discussion for simplicity. However, they were fully considered in our evaluation.

6.2 Case Study

In this section, we detail the case-study experiments we performed, and compare our fine-grained DAG scheduling approach to monolithic and coarse-grained DAG scheduling.

6.2.1 Experimental Evaluation

All of our case-study experiments focused on the *Histogram of Oriented Gradients* (HOG) algorithm, a well-known CV technique for recognizing pedestrians in input images (Dalal and Triggs, 2005b).

Why HOG? The HOG algorithm is required by the OpenVX 1.2 specification,[9] ensuring its relevance in real-world graph-based CV. HOG is inherently a multi-stage technique: it calculates a directional *gradient* for each pixel, sorts gradients into histograms, normalizes lighting and contrast, and then performs classification. These computations are performed at multiple image-scale levels, using successively smaller versions of the original image. These steps require both CPU and GPU computations, meaning that HOG fits naturally into a DAG-based model.

HOG implementation and DAG variations. At the time of writing, no open-source implementations of version 1.2 of OpenVX exists, so we based our case-study experiments on the HOG implementation available in the open-source CV library *OpenCV*.[10] OpenCV provides CV functions, but does not structure computations as DAGs. To create DAGs, we split the OpenCV HOG code into separate functions, and

[9]`https://www.khronos.org/registry/OpenVX/specs/1.2/OpenVX_Specification_1_2.pdf`, Sec. 3.53.1.

[10]`https://docs.opencv.org/3.4.1/d5/d33/structcv_1_1HOGDescriptor.html`.

designated each function as a DAG node. We compared the response times of successively finer-grained notions of DAG scheduling, corresponding to monolithic, coarse-grained, and fine-grained HOG DAGs.[11]

The monolithic version of HOG corresponds to the type of DAG that one might specify using OpenVX, and consists of a single DAG of six types of nodes (three are replicated per scale level), as shown in Figure 6.3 (a). Implementing this DAG required the fewest changes to OpenCV code, as monolithic execution requires only a single thread to sequentially execute the six nodes' functions. Coarse-grained HOG uses the same DAG as monolithic HOG, but, as discussed in Section 6.1.1, each of the six nodes is a schedulable entity, with scheduling via G-EDF with early releasing. We also used G-EDF, but without early releasing, as a monolithic DAG scheduler.

In fine-grained HOG, several of the coarse-grained nodes are refined, as shown in Figure 6.3 (b). This DAG reflects our new fine-grained approach, where nodes are treated as tasks and the techniques (early releasing, intra-task parallelism, and G-FL scheduling) in Section 6.1.2 are applied.

Fine-grained DAG implementation. Implementing fine-grained HOG introduced a series of challenges. For example, intra-task parallelism required multiple instances of each DAG to ensure each node can have multiple jobs executing in parallel. Other challenges included priority points that varied (for launching GPU kernels and awaiting results), handling inter-process communication (IPC) between CPU and GPU nodes, enforcing guards on early releasing for GPU nodes, and computing task offsets from response-time bounds in order to run the experiments.

As in prior work (Elliott *et al.*, 2015), we used PGM$^{\text{RT}}$ (Elliott *et al.*, 2014) to handle IPC in the coarse- and fine-grained HOG variants. PGM$^{\text{RT}}$ introduces producer/consumer buffers and mechanisms that enable producer nodes to write output data and consumer nodes to suspend until data is available on all inbound edges.

Test platform. Our evaluation platform was selected to over-approximate current NVIDIA embedded offerings for automotive systems, such as the Drive PX2. This platform features a single NVIDIA Titan V GPU, two eight-core Intel CPUs, and 32 GB of DRAM. Each core features a 32-KB L1 data cache, a 32-KB L1 instruction cache, and a 1-MB L2 cache, and all eight cores on a socket share an 11-MB L3 cache. The system was configured to run Ubuntu 16.04 as an OS, using version 2017.1 of the LITMUS$^{\text{RT}}$ kernel (LITMUS$^{\text{RT}}$ Project, 2018), with hardware multi-threading disabled.

[11]Our source code is available online at `https://github.com/Yougmark/opencv/tree/rtss18`.

Overall computational workload. One would expect contention for hardware resources in many real-world use cases, such as an autonomous vehicle that processes data from multiple camera feeds. We approximated a "contentious" workload by executing six HOG processes on our hardware platform—the limit based on our platform's DRAM, CPU, and GPU capacity. This number of HOG processes makes ensuring bounded response times difficult without careful consideration of resource allocation. This scenario also reflects the very real possibility of executing at high platform utilization, as is commonly done in the automotive industry. To ensure consistency with the GPU scheduling rules in Section 4.2.1, we conducted all of our experiments using NVIDIA's MPS.

Video-frame-processing can potentially experience cache-affinity-loss issues under global scheduling. We therefore considered two variants of both G-EDF and G-FL: a truly global variant where any of the six DAGs can be scheduled on any of our platform's 16 CPUs, and a "clustered" variant where the six DAGs are partitioned between the machine's two sockets, with scheduling being "global" only within a socket. We refer to the latter variants as C-EDF and C-FL, respectively, where the "C" prefix stands for "clustered."

6.2.2 Results

Our experiments were designed to examine analytical response-time bounds and observed response times under the considered scheduling approaches. We also sought to examine the overhead required to support fine-grained scheduling.

Analytical bounds. To compute analytical response-time bounds, we first computed CPU WCETs and worst-case GPU workloads via a measurement process. All worst-case values were calculated as the 99th percentile of 30,000 samples obtained with all six DAGs executing together to cause contention. For each GPU task τ_i, we used NVIDIA's profiling tool `nvprof` to measure B_i and H_i, and instrumented the CUDA kernels to measure L_i *on the GPU* using the `globaltimer` performance-counter register. For HOG, $H_{max} = 256$ and $h = 64$. We measured CPU WCETs using Feather-Trace (Brandenburg and Anderson, 2007). For all DAGs, $T_i = 33ms$.

We computed fine-grained response-time bounds by using Theorem 4.3 in Section 4.2.4 for GPU nodes and Therom 2 from (Yang and Anderson, 2014b) for CPU nodes and by then applying the techniques in Section 6.1.2 to obtain an overall end-to-end bound. We tried computing analytical bounds for the coarse-grained (resp., monolithic) C-EDF and G-EDF variants using prior work (Yang *et al.*, 2015) (resp., (Devi and

	Monolithic G-EDF	Monolithic C-EDF	Coarse-Grained G-EDF	Coarse-Grained C-EDF	Fine-Grained G-FL	Fine-Grained C-FL
Analytical Bound (ms)	N/A	N/A	N/A	N/A	542.39	477.25
Observed Maximum Response Time (ms)	170091.06	243745.21	427.07	428.50	125.66	131.43
Observed Average Response Time (ms)	84669.47	121748.05	136.57	121.52	65.99	66.06

Table 6.1: Analytical and observed response times. A bound of N/A indicates unschedulability, so no bound could be computed.

Anderson, 2008)), but found these variants to be unschedulable.[12] These results are summarized in the first row of Table 6.1.

Observation 6.1. With respect to schedulablity, the monolithic and coarse-grained variants could not even come close to supporting all six cameras (*i.e.*, DAGs).

With respect to schedulability, the monolithic variants could not even support one camera, because the overall execution time of a single monolithic DAG far exceeds its period. The coarse-grained variants were only slightly better, being able to support just one camera (in which case the choice of variant, C-EDF vs. G-EDF, is of little relevance). In this case, adding a second HOG DAG increased GPU responses to the point of causing a CPU utilization-constraint violation. Note that the increase in CPU utilization is due to using suspension-oblivious analysis.

Observation 6.2. With respect to schedulability, both fine-grained variants were able to support all six cameras.

The better schedulability of the fine-grained variants largely resulted from scheduling shorter tasks with intra-task parallelism, though switching to a fair-lateness-based scheduler also proved beneficial. In particular, we found that the scheduler change reduced node response times by 0.1–9.9%. While this reduction is modest, it is still useful. Nonetheless, these reductions suggest that most of the schedulability improvements stemmed from increasing parallelism between both CPU and GPU tasks.

Observed response times. Figure 6.4 plots DAG response-time distributions, which we computed for all tested variants from measurement data. Corresponding worst- and average-case times are also reported in Table 6.1.

[12]In the original coarse-grained work (Elliott *et al.*, 2015; Yang *et al.*, 2015), locking protocols were used to preclude concurrent GPU accesses. We instead allowed concurrent accesses and used the analysis in Section 4.2, but the coarse-grained variants were still unschedulable.

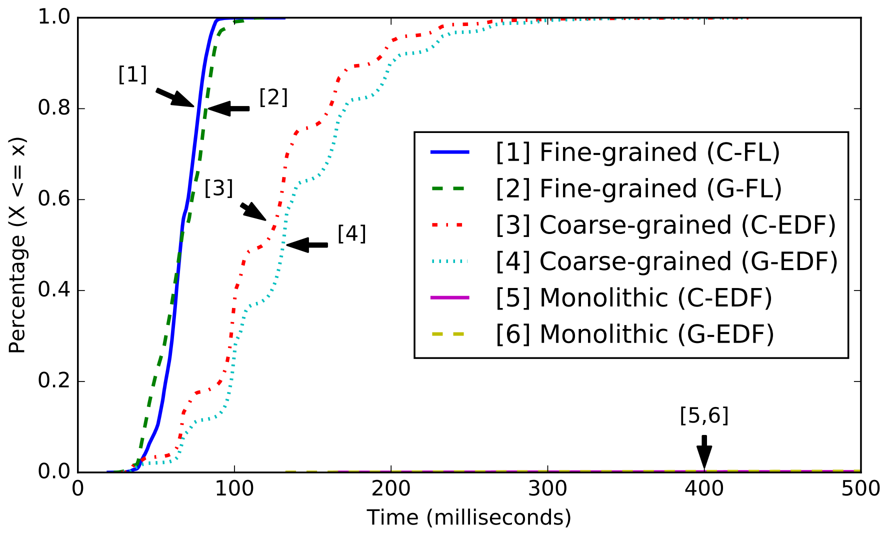


Figure 6.4: CDFs of response times for varying DAG granularity.

Observation 6.3. The average (resp., worst-case) observed response time under the fine-grained variants was around 66 ms (resp., 130 ms), which is substantially lower than the non-fine-grained variants. (For reference, the response time of an alert driver is reported to be around 700 ms (Green, 2000).)

This observation is supported by Figure 6.4 and Table 6.1. Note that the difference between clustered and global scheduling was not substantial. This is because the aggregate memory footprint of all frames concurrently being accessed under both variants tended to far exceed the size of the L3 cache.

Observation 6.4. The analytical fine-grained response-time bounds upper bounded the observed worst-case times.

This observation is supported by Figure 6.4 and Table 6.1. While the listed bounds of 477.25 ms and 542.39 ms in Table 6.1 may seem high, note that they are based on considering worst-case scenarios that may be unlikely to occur in practice (and they are still within the limit mentioned in (Green, 2000)). Moreover, the monolithic and coarse-grained variants were unable to guarantee *any* bounds when scheduling all six DAGs.

Observation 6.5. Observed response times exhibited lower variation under fine-grained scheduling.

This observation is supported by Figure 6.4. The fine-grained variances in this plot are several orders of magnitude less than the variances for the other variants.

Observation 6.6. Early releasing improved observed response times.

To verify this observation, we conducted additional experiments in which we disabled early releasing for the fine-grained G-FL variant. In these experiments, we found that early releasing reduced observed response times by 49%.

Overhead of DAG conversion. We estimated the overhead of converting from a coarse-grained DAG to a fine-grained one by comparing the computed WCET of every coarse-grained node with the sum of the computed WCETs of the fine-grained nodes that replace it. The total percentage increase across all nodes was deemed to be *overhead*.

Observation 6.7. The additional overhead introduced to support fine-grained scheduling had modest impact.

From our collected data, the total overhead was 14.15%.

6.3 Chapter Summary

In this chapter, we applied our analysis and implementation techniques from Chapters 4 and 5 to a fine-grained approach of decomposing and scheduling OpenVX graphs. To illustrate the efficacy of our proposed fine-grained approach, we presented an experimental case study. We saw in this study that our fine-grained approach enabled response-time bounds to be guaranteed and observed response times to be reduced. A notable aspect of our fine-grained approach is its crucial reliance on allowing intra-task parallelism, a feature forbidden in most conventional real-time task models.

CHAPTER 7: CONCLUSION

Moving forward, increased autonomy will come to mass-market vehicles, for which computer-vision-based perception systems are essential. Compared to lidars and radars, cameras are less expensive and uniquely capable of perceiving colors and textures. In accelerating computer-vision workloads, GPUs are key enablers that provide massive parallelism. As a number of disparate applications are required to ensure adequate coverage and redundancy offered by a large quantity and diversity of cameras, it becomes necessary for these applications to share increasingly powerful GPUs. This is especially important for embedded autonomous-driving systems, where resources should be fully utilized. Meanwhile, tasks that share GPUs for better throughput performance should also have guaranteed bounded response times in a safety-critical autonomous-driving system, and any induced accuracy loss must be tolerable.

This dissertation addresses the challenging issue of *how* GPUs can be shared between computer-vision applications to improve throughput performance while bounding response times and ensuring high per-camera-stream accuracy. We **(i)** investigated NVIDIA GPUs to reveal GPU scheduling rules and synchronization pitfalls, **(ii)** developed a task model of GPU tasks, analyzed response-time bounds for GPU tasks, **(iii)** designed and implemented CNN frameworks to exploit pipelined and parallel execution for throughput improvement, and **(iv)** conducted a case study of fine-grained graph scheduling for the computer-vision standard OpenVX, comparing both analytical and empirical results with prior work.

The results we obtained support the following thesis we presented in Chapter 1:

> *Capacity loss in GPU-using real-time task systems can be lessened through sharing GPUs. Such sharing can be guaranteed with bounded response times. Furthermore, throughput can be improved without compromising such guarantees by exploiting pipelining and parallelism in real-time tasks. These real-time guarantees of tasks on shared GPUs are enabled with a new model for GPU execution and corresponding schedulability analysis.*

In the following sections, we first summarize our results in Section 7.1, acknowledge others' contributions to these results in Section 7.2, then discuss future work in Section 7.3, and finally conclude in Section 7.4.

7.1 Summary of Results

In this section, we summarize our results that support the thesis above.

Investigation of NVIDIA GPUs. We first investigated NVIDIA GPUs to reveal GPU scheduling rules. In particular, we focused on the scheduling of kernel and memory-copy operations from multiple CUDA streams. These operations are allowed to execute concurrently when submitted from the same CUDA context, but are managed by undocumented scheduling rules. Unfortunately, the software and hardware stacks are closed-source. Therefore, we inferred the scheduling rules presented in Section 3.1 from conducting extensive experiments and examining their scheduling results. We defined the basic scheduling rules that specify how the kernel and memory-copy operations progress through a variant of a hierarchical FIFO structure and how the scheduling can be impacted by constrained resources, including GPU threads and shared memory. In addition to the basic scheduling rules, we considered additional features available to CUDA programmers that can impact scheduling, including the NULL stream, stream priorities, and streams in independent process address spaces.

In addition to the scheduling rules, we discussed one set of issues that leads to a surprising number of pitfalls when NVIDIA GPUs are used: synchronization. For example, documented sources of implicit synchronization may not occur, while some functions that cause implicit synchronization are neglected, and some functions even block future, unrelated CUDA tasks *on the CPU*. The resultant blocking can cause capacity loss, potentially leading to unbounded response times in task sets with high utilization. Ignoring these synchronization issues would be perilous in a safety-critical system where blocking must be anticipated and accounted for in analysis. We also noted that running CPU tasks in independent processes managed by MPS can overcome this issue.

In addition, we examined a fundamental trade-off for designing real-time tasks that use a GPU. Specifically, we evaluated the performance of multiple instances of various computer-vision tasks in different settings. In each setting, applications were executed either as threads that share the same address space or as OS processes both with or without MPS. We observed that the execution times of tasks running in multiple processes with MPS are more predictable. However, despite these benefits of MPS, it is currently not available for embedded platforms, such as the TX2.

These specific issues comprise the essential background for developing task systems where real-time design meets GPUs. We summarized them to provide guidance, recommendations, and pitfall warnings to both researchers and implementation practitioners.

Task model of GPU tasks. Based on the fundamental understanding from our investigation of NVIDIA GPUs, we developed a task model of GPU tasks, which expands on the sporadic task model (Mok, 1983) with additional parameters that characterize the parallel execution of GPU tasks under the CUDA programming model. Using this task model, we showed that NVIDIA GPU scheduling can inherently cause unnecessary idleness of its processing units, implying a fundamental capacity loss. Specifically, we showed that the capacity loss can be extreme without *intra-task parallelism*, which allows multiple jobs of a task to execute in parallel, but is forbidden in most conventional real-time task models. Moreover, GPUs can be left idle if the number of idle GPU threads on any one SM is insufficient for scheduling the next unassigned block. This implies that some capacity loss is inevitable when seeking to ensure response-time bounds for GPUs. We expressed such loss by providing a *total utilization bound*. We showed that this bound is tight with a counterexample, a task system that has total utilization slightly exceeding this bound but with unbounded response times.

Response-time bound analysis for tasks sharing GPUs. Our total utilization bound forms the schedulability condition for bounded response times of task systems where intra-task parallelism is allowed. Under this schedualbility condition, we derived response-time bounds for tasks sharing NVIDIA GPUs under the scheduling rules we inferred. By leveraging prior work on DAG scheduling analysis, we modeled GPU-using applications as DAGs in which each node is either CPU or GPU work. We then showed that an end-to-end response-time bound can be computed inductively for each DAG by scheduling its nodes such that they can be viewed as sporadic tasks and by then leveraging response-time bounds applicable to CPU tasks (Devi and Anderson, 2008; Liu and Anderson, 2011; Erickson and Anderson, 2012) and our response-time bounds for GPU tasks.

Design and implementation of CNN frameworks for throughput improvement. One of our goals was to improve the throughput performance of computer-vision frameworks to support adequate coverage and redundancy offered by an array of cameras. We addressed this challenge in an industrial design study by re-designing CNN software. We showed how to transform a traditional CNN framework to obtain pipelined and parallel-pipelined variants to better utilize hardware resources. Through re-thinking how the YOLO

CNN is executed, we showed that its throughput can be improved by more than twofold on the PX2 with only a minor latency increase and no change in accuracy. We also showed that the throughput can be further improved by properly load balancing work across multiple GPUs. Finally, to accommodate more cameras than these basic methods allow, we proposed the technique of compositing multiple images from different camera streams. We showed that this approach can greatly boost throughput (up to 250 FPS) at the expense of some accuracy loss. We demonstrated that this accuracy loss can be partially mitigated through retraining the neural network.

Evaluation of end-to-end bounds of OpenVX applications. We applied these implementation techniques and our response-time bound analysis to a computer-vision standard, OpenVX. Prior work partially addressed the issues of enabling real-time OpenVX by treating individual graph nodes as schedulable entities (Elliott *et al.*, 2015; Yang *et al.*, 2015). We proposed a more fine-grained scheduling of OpenVX graphs so that each node accesses either a CPU or a GPU (but not both). In this way, we were able to explore the parallelism that was inhibited in the coarse-grained scheduling from prior work. Applying our response-time bound analysis enables avoiding the capacity loss induced in handling suspensions for accessing GPUs. We conducted a case study using a HOG application for evaluating both analytical and empirical results in comparison with prior work. We showed that our fine-grained approach enabled response-time bounds to be guaranteed and observed response times to be reduced.

7.2 Acknowledgements

These results previously appeared in a series of papers that were produced through team efforts. I have listed my collaborators' contributions in detail in the end of each chapter, and also summarize them here.

The experiments that investigate NVIDIA GPUs were conducted using the framework that Nathan Otterness developed, the plotting scripts that Tanya Amert wrote, and the sample applications that I ported. Each of us conducted part of the experiments to reveal the scheduling rules and expose the list of pitfalls, also with Joshua Bakita's help.

Kecheng Yang reworked the response-time bound analysis into the current version.

The industrial study of improving throughput performance of CNN frameworks was conducted during my internship with Shige Wang at General Motors. Thanh Vu retrained the YOLO model with assistance from Cheng-Yang Fu and Akash Bapat.

Chapters 3–6 were largely drawn from our papers, which include text drafted and polished by my co-authors—especially Tanya Amert, Jim Anderson, Nathan Otterness, Don Smith, and Kecheng Yang.

7.3 Future Work

We now discuss future work involving sharing GPUs for real-time autonomous-driving applications.

Mitigate interference between GPU tasks. In this dissertation, we focused on the native scheduler in NVIDIA GPUs. GPU tasks could interfere while accessing constrained resources concurrently. Mitigating this interference could lead to better response times. As we reviewed in Section 2.3.4, many techniques have been applied to reduce interference between tasks running on a GPU, such as the management of cache, memory, and computing resources. Another technique is to prevent tasks from executing concurrently if their interference is estimated to be high through runtime profiling or GPU kernel characterization. Future research can determine whether these interference-mitigation techniques could be incorporated into our analysis when deriving response-time bounds.

Analyze the impacts of memory usage. We noted that the memory usage of GPU tasks impacts the scheduling in a way similar to the usage of computing resources. As we noted in Section 4.2.2, we assumed the absence of shared-memory-induced blocking because we have never observed such blocking on any platform for any workload in our experiments involving computer-vision workloads on NVIDIA GPUs spanning several years. In addition, we remarked that register usage can be limited with a compiler option. Nonetheless, the issue of incorporating the memory usage of GPU tasks into our analysis is an interesting challenge for the future.

Handle memory-copy operations. In the workloads of interest to us, memory copies are short, so we assumed that copy operations are included in CPU nodes as any resulting suspension-based capacity loss is minor. However, it is necessary to handle more lengthy memory copies as separate nodes. In addition, on platforms that have multiple CEs, memory copies of different transferring directions may need to be considered separately.

Explore other features in recent NVIDIA GPUs. In our investigation of NVIDIA GPUs, we mentioned other CUDA features that are rarely used. They include instruction-level preemption, dynamic parallelism, and CUDA Graphs. The instruction-level preemption could be beneficial for ensuring fairer scheduling that respects task priorities, compared to the default non-preemptive execution on NVIDIA GPUs. However, the cost of preemption overheads needs to be evaluated. Dynamic parallelism allows the submission of new work on a GPU, avoiding CPU-GPU communication overheads, but the scheduling behavior of the work launched from the GPU needs to be investigated when deriving response-time bounds. CUDA Graphs can structure a sequence of GPU kernels into graphs. Each graph of multiple kernels can be launched with a single CPU function call and thus launch overheads can be reduced. However, its underlying graph-related optimization may make it difficult to model the scheduling behavior of individual kernels. Also, the scheduling of multiple graphs needs to be investigated. While improving the performance of GPU applications, all these features cause challenging problems for real-time systems.

Extend the investigation of the graph-node granularity. In our case study, we showed the benefits of fine-grained graph scheduling. However, it may not be possible to decompose all applications into DAGs at such a fine-grained level, as it may result in overheads that outweigh the performance gain from the explored parallelism. To determine conditions for fine-grained scheduling to be beneficial, further investigation, such as an overhead-aware schedulability study, is necessary. We can also consider DAGs with a mix of both coarse-grained and fine-grained nodes. However, for such DAGs, it would be challenging to perform the response-time bound analysis. Meanwhile, a computer-vision framework that supports configurable node granularities is required.

Investigate other GPUs. We focused on NVIDIA GPUs because they dominate the market and are widely used in research. However, we also noted that their closed-source software and hardware stacks and incomplete documents have hindered research in many aspects. To generalize our results, we should extend our analysis and evaluation to other GPUs. For example, AMD GPUs share a similar programming model and hardware architecture. In addition, AMD provides an open-source software stack, which affords more opportunities to investigate GPU scheduling and implement additional scheduling and resource-management options for real-time systems.

7.4 Closing Remarks

In this dissertation, we have shown how GPUs can be shared for real-time autonomous-driving systems. Such sharing is motivated by the increase in computation requirements of these systems and the performance advancements of GPUs. Meanwhile, as GPUs are being applied in safety-critical systems, real-time certification of GPUs becomes important. Such certification will also be applicable to other accelerators. In deriving real-time guarantees for GPUs, we showed how an accelerator can be investigated, modeled, analyzed, and evaluated. We hope that our work will contribute to and encourage future research in sharing GPUs or other accelerators for real-time systems.

BIBLIOGRAPHY

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., *et al.* (2016). Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283.

Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., Schoinas, I., Uhlig, R., Vembu, B., and Wiegert, J. (2006). Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3).

Ackerman, E. (2017). Toyota's Gill Pratt on self-driving cars and the reality of full autonomy. *IEEE Spectrum*. `https://spectrum.ieee.org/cars-that-think/transportation/self-driving/toyota-gill-pratt-on-the-reality-of-full-autonomy`.

Adinets, A. (2014). CUDA dynamic parallelism API and principles. `https://devblogs.nvidia.com/cuda-dynamic-parallelism-api-principles/`.

Adriaens, J. T., Compton, K., Kim, N. S., and Schulte, M. J. (2012). The case for GPGPU spatial multitasking. In *Proceedings of the 2012 IEEE International Symposium on High-Performance Compute Architecture*, pages 1–12.

Agrawal, A., Modi, A. N., Passos, A., Lavoie, A., Agarwal, A., Shankar, A., Ganichev, I., Levenberg, J., Hong, M., Monga, R., *et al.* (2019). Tensorflow eager: A multi-stage, python-embedded dsl for machine learning. *arXiv preprint arXiv:1903.01855*.

Aguilera, P., Morrow, K., and Kim, N. S. (2014a). Fair share: Allocation of GPU resources for both performance and fairness. In *Proceedings of the 32nd IEEE International Conference on Computer Design*, pages 440–447.

Aguilera, P., Morrow, K., and Kim, N. S. (2014b). QoS-aware dynamic resource allocation for spatial-multitasking GPUs. In *Proceedings of the 19th Asia and South Pacific Design Automation Conference*, pages 726–731.

Akai, N., Morales, L. Y., Yamaguchi, T., Takeuchi, E., Yoshihara, Y., Okuda, H., Suzuki, T., and Ninomiya, Y. (2017). Autonomous driving based on accurate localization using multilayer lidar and dead reckoning. In *Proceedings of the 20th IEEE International Conference on Intelligent Transportation Systems*, pages 1–6.

Ali, W. and Yun, H. (2018). Protecting real-time GPU kernels on integrated CPU-GPU SoC platforms. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems*, pages 19:1–19:22.

Amazon AWS (2020). AWS and NVIDIA. `https://aws.amazon.com/nvidia/`.

AMD (2016). AMD I/O virtualization technology (IOMMU) specification. `http://developer.amd.com/wordpress/media/2013/12/48882_IOMMU.pdf`.

AMD (2016). ROCm, a new era in open GPU computing. `https://rocm.github.io/`.

Amert, T., Otterness, N., Yang, M., Anderson, J. H., and Smith, F. D. (2017). GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *Proceedings of the 38th IEEE International Real-Time Systems Symposium*, pages 104–115.

Anderson, J. H. and Srinivasan, A. (2000). Early-release fair scheduling. In *Proceedings 12th Euromicro Conference on Real-Time Systems*, pages 35–43.

Andersson, B., Baruah, S., and Jonsson, J. (2001). Static-priority scheduling on multiprocessors. In *Proceedings of the 22nd IEEE International Real-Time Systems Symposium*, pages 193–202.

Andersson, B. and de Niz, D. (2012). Analyzing global-EDF for multiprocessor scheduling of parallel tasks. In *Proceedings of the 2012 International Conference On Principles Of Distributed Systems*, pages 16–30.

Aravantinos, V. and Diehl, F. (2018). Traceability of deep neural networks. *arXiv preprint arXiv:1812.06744*.

Audsley, N. C., Burns, A., Richardson, M. F., and Wellings, A. J. (1991). Hard real-time scheduling: The deadline-monotonic approach. *IFAC Proceedings Volumes*, 24(2):127–132.

Ausavarungnirun, R., Ghose, S., Kayiran, O., Loh, G. H., Das, C. R., Kandemir, M. T., and Mutlu, O. (2015). Exploiting inter-warp heterogeneity to improve gpgpu performance. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 25–38.

Ausavarungnirun, R., Landgraf, J., Miller, V., Ghose, S., Gandhi, J., Rossbach, C. J., and Mutlu, O. (2017). Mosaic: a GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 136–150.

Ausavarungnirun, R., Miller, V., Landgraf, J., Ghose, S., Gandhi, J., Jog, A., Rossbach, C. J., and Mutlu, O. (2018). Mask: Redesigning the GPU memory hierarchy to support multi-application concurrency. *ACM SIGPLAN Notices*, 53(2):503–518.

Axer, P., Quinton, S., Neukirchner, M., Ernst, R., Döbel, B., and Härtig, H. (2013). Response-time analysis of parallel fork-join workloads with real-time constraints. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 215–224.

Badue, C., Guidolini, R., Carneiro, R. V., Azevedo, P., Cardoso, V. B., Forechi, A., Jesus, L., Berriel, R., Paixão, T., Mutz, F., *et al.* (2019). Self-driving cars: A survey. *arXiv preprint arXiv:1901.04407*.

Baek, I., Harding, M., Kanda, A., Choi, R. K., Samii, S., and Rajkumar, R. R. (2020). CARSS: Client-aware resource sharing and scheduling for heterogeneous applications. In *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 324–335.

Bakhoda, A., Yuan, G. L., Fung, W. W., Wong, H., and Aamodt, T. M. (2009). Analyzing CUDA workloads using a detailed GPU simulator. In *Proceesdings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177.

Baruah, S. (2014). Improved multiprocessor global schedulability analysis of sporadic DAG task systems. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, pages 97–105.

Baruah, S. (2015a). The federated scheduling of constrained-deadline sporadic DAG task systems. In *Proceedings of the 2015 Design, Automation and Test in Europe Conference and Exhibition*, pages 1323–1328.

Baruah, S. (2015b). Federated scheduling of sporadic DAG task systems. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium*, pages 179–186.

Baruah, S. (2015c). The federated scheduling of systems of conditional sporadic DAG tasks. In *Proceedings of the 2015 IEEE International Conference on Embedded Software*, pages 1–10.

Baruah, S. (2018). Resource-efficient execution of conditional parallel real-time tasks. In *Proceedings of the 2018 European Conference on Parallel Processing*, pages 218–231.

Baruah, S., Bonifaci, V., and Marchetti-Spaccamela, A. (2015). The global EDF scheduling of systems of conditional sporadic DAG tasks. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, pages 222–231.

Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A., Stougie, L., and Wiese, A. (2012). A generalized parallel task model for recurrent real-time processes. In *Proceedings of the 33rd IEEE International Real-Time Systems Symposium*, pages 63–72.

Basaran, C. and Kang, K.-D. (2012). Supporting preemptive task executions and memory copies in GPGPUs. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 287–296.

Bast, H., Delling, D., Goldberg, A., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D., and Werneck, R. F. (2016). Route planning in transportation networks. In *Algorithm engineering*, pages 19–80.

Bedford, D., Morgan, G., and Austin, J. (1996). Requirements for a standard certifying the use of artificial neural networks in safety critical applications. In *Proceedings of the 1996 International Conference on Artificial Neural Networks*, pages 1–6.

Behere, S. and Törngren, M. (2016). A functional reference architecture for autonomous driving. *Information and Software Technology*, 73:136–150.

Bensaou, B., Tsang, D. H., and Chan, K. T. (2001). Credit-based fair queueing (CBFQ): a simple service-scheduling algorithm for packet-switched networks. *IEEE/ACM Transactions on Networking*, 9(5):591–604.

Berezovskyi, K., Bletsas, K., and Andersson, B. (2012). Makespan computation for GPU threads running on a single streaming multiprocessor. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 277–286.

Berezovskyi, K., Bletsas, K., and Petters, S. M. (2013). Faster makespan estimation for GPU threads on a single streaming multiprocessor. In *Proceedings of the 18th IEEE Conference on Emerging Technologies & Factory Automation*, pages 1–8.

Berezovskyi, K., Guet, F., Santinelli, L., Bletsas, K., and Tovar, E. (2016). Measurement-based probabilistic timing analysis for graphics processor units. In *Proceedings of the 2016 International Conference on Architecture of Computing Systems*, pages 223–236.

Berezovskyi, K., Santinelli, L., Bletsas, K., and Tovar, E. (2014). WCET measurement-based and extreme value theory characterisation of CUDA kernels. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, pages 279–288.

Betts, A. and Donaldson, A. (2013). Estimating the WCET of GPU-accelerated applications using hybrid analysis. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 193–202.

BlackBerry (2020). QNX neutrino realtime operating system (RTOS). `https://blackberry.qnx.com/en/software-solutions/embedded-software/industrial/qnx-neutrino-rtos`.

Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., *et al.* (2016). End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*.

Bonifaci, V., Marchetti-Spaccamela, A., Stiller, S., and Wiese, A. (2013). Feasibility analysis in the sporadic DAG task model. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 225–233.

Brandenburg, B. (2011). *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, Chapel Hill, NC, USA.

Brandenburg, B. and Anderson, J. (2007). Feather-trace: A light-weight event tracing toolkit. In *Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 61–70.

Brandenburg, B. and Anderson, J. (2010). Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st IEEE International Real-Time Systems Symposium*, pages 49–60.

Breitenstein, M. D., Reichlin, F., Leibe, B., Koller-Meier, E., and Van Gool, L. (2009). Robust tracking-by-detection using a detector confidence particle filter. In *2009 IEEE 12th International Conference on Computer Vision*, pages 1515–1522.

Broggi, A., Buzzoni, M., Debattisti, S., Grisleri, P., Laghi, M. C., Medici, P., and Versari, P. (2013). Extensive tests of autonomous driving technologies. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1403–1415.

Brooker, G. M. (2007). Mutual interference of millimeter-wave radar systems. *IEEE Transactions on Electromagnetic Compatibility*, 49(1):170–181.

Buehler, M., Iagnemma, K., and Singh, S. (2007). *The 2005 DARPA grand challenge: the great robot race*, volume 36. Springer.

Buehler, M., Iagnemma, K., and Singh, S. (2009). *The DARPA urban challenge: autonomous vehicles in city traffic*, volume 56. Springer.

Burton, S., Gauerhof, L., and Heinzemann, C. (2017). Making the case for safety of machine learning in highly automated driving. In *Proceedings of the 2017 International Conference on Computer Safety, Reliability, and Security*, pages 5–16.

Calandrino, J. M., Leontyev, H., Block, A., Devi, U. C., and Anderson, J. H. (2006). LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 111–126.

Capodieci, N., Cavicchioli, R., Bertogna, M., and Paramakuru, A. (2018). Deadline-based scheduling for GPU with preemption support. In *Proceedings of the 39th IEEE International Real-Time Systems Symposium*, pages 119–130.

Capodieci, N., Cavicchioli, R., Valente, P., and Bertogna, M. (2017). Sigamma: Server based integrated GPU arbitration mechanism for memory accesses. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 48–57.

Carvalho, P., Cruz, R., Drummond, L. M., Bentes, C., Clua, E., Cataldo, E., and Marzulo, L. A. (2020). Kernel concurrency opportunities based on GPU benchmarks characterization. *Cluster Computing*, 23(1):177–188.

Carvalho, P., Drummond, L. M., Bentes, C., Clua, E., Cataldo, E., and Marzulo, L. A. (2017). Analysis and characterization of GPU benchmarks for kernel concurrency efficiency. In *Latin American High Performance Computing Conference*, pages 71–86.

Cavicchioli, R., Capodieci, N., and Bertogna, M. (2017). Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms. In *Proceedings of the 22nd IEEE International Conference on Emerging Technologies and Factory Automation*, pages 1–10.

Cavicchioli, R., Capodieci, N., Solieri, M., and Bertogna, M. (2019). Novel methodologies for predictable CPU-to-GPU command offloading. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems*, pages 22:1–22:22.

Chen, Q., Yang, H., Guo, M., Kannan, R. S., Mars, J., and Tang, L. (2017a). Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 17–32.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. (2015). Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*.

Chen, X., Kundu, K., Zhu, Y., Ma, H., Fidler, S., and Urtasun, R. (2017b). 3D object proposals using stereo imagery for accurate object class detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(5):1259–1272.

Cheng, C.-H., Diehl, F., Hinz, G., Hamza, Y., Nührenberg, G., Rickert, M., Ruess, H., and Truong-Le, M. (2018a). Neural networks for safety-critical applicationschallenges, experiments and perspectives. In *Proceedings of the 2018 Design, Automation and Test in Europe*, pages 1005–1006.

Cheng, C.-H., Huang, C.-H., Ruess, H., Yasuoka, H., *et al.* (2018b). Towards dependability metrics for neural networks. In *Proceedings of the 16th ACM/IEEE International Conference on Formal Methods and Models for System Design*, pages 1–4.

Cheng, X., Wang, P., and Yang, R. (2018c). Learning depth with convolutional spatial propagation network. *arXiv preprint arXiv:1810.02695*.

Cho, H., Seo, Y.-W., Kumar, B. V., and Rajkumar, R. R. (2014). A multi-sensor fusion system for moving object detection and tracking in urban driving environments. In *Proceedings of the 2014 IEEE International Conference on Robotics and Automation*, pages 1836–1843.

Chwa, H. S., Lee, J., Phan, K.-M., Easwaran, A., and Shin, I. (2013). Global EDF schedulability analysis for synchronous parallel tasks on multicore platforms. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 25–34.

Codevilla, F., Miiller, M., López, A., Koltun, V., and Dosovitskiy, A. (2018). End-to-end driving via conditional imitation learning. In *Proceedings of the 2018 IEEE International Conference on Robotics and Automation*, pages 1–9.

Crowe, S. (2019a). California's self-driving car reports are imperfect, but they're better than nothing. `https://www.theverge.com/2019/2/13/18223356/california-dmv-self-driving-car-disengagement-report-2018`.

Crowe, S. (2019b). Waymo autonomous vehicles leave apple in the dust. `https://www.therobotreport.com/waymo-autonomous-vehicles-apple/`.

Dai, H., Lin, Z., Li, C., Zhao, C., Wang, F., Zheng, N., and Zhou, H. (2018). Accelerate GPU concurrent kernel execution by mitigating memory pipeline stalls. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture*, pages 208–220.

Dalal, N. and Triggs, B. (2005a). Histograms of oriented gradients for human detection. In *Proceedings of the 2005 IEEE Conference on Computer Vision and Pattern Recognition*, pages 886–893.

Dalal, N. and Triggs, B. (2005b). Histograms of oriented gradients for human detection. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, volume 1, pages 886–893.

Davis, J. and Goadrich, M. (2006). The relationship between precision-recall and ROC curves. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 233–240.

De Kruijf, M. and Sankaralingam, K. (2011). Idempotent processor architecture. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 140–151.

Devi, U. and Anderson, J. (2005). Tardiness bounds for global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 330–341.

Devi, U. and Anderson, J. (2008). Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189.

Dickmanns, E. D., Behringer, R., Dickmanns, D., Hildebrandt, T., Maurer, M., Thomanek, F., and Schiehlen, J. (1994). The seeing passenger car "VaMoRs-P". In *Proceedings of the 1994 IEEE Intelligent Vehicles Symposium*, pages 68–73.

Dickmanns, E. D. *et al.* (1997). Vehicles capable of dynamic vision. In *Proceedings of 15th International Joint Conference on Artificial Intelligence*, pages 1577–1592.

Dickmanns, E. D. and Zapp, A. (1987). Autonomous high speed road vehicle guidance by computer vision. *IFAC Proceedings Volumes*, 20(5):221–226.

Dijkstra, E. W. *et al.* (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271.

Dong, Z., Liu, C., Gatherer, A., McFearin, L., Yan, P., and Anderson, J. (2017). Optimal dataflow scheduling on a heterogeneous multiprocessor with reduced response time bounds. In *Proceedings of the 29th Euromicro Conference on Real-Time Systems*, pages 15:1–15:22.

Doshi-Velez, F. and Kim, B. (2017). Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*.

Duato, J., Pena, A. J., Silla, F., Mayo, R., and Quintana-Ortí, E. S. (2010). rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proceedings of the 2010 International Conference on High Performance Computing & Simulation*, pages 224–231.

Eigen, D., Puhrsch, C., and Fergus, R. (2014). Depth map prediction from a single image using a multi-scale deep network. In *Advances in Neural Information Processing Systems*, pages 2366–2374.

Elliott, G., Kim, N., Erickson, J., Liu, C., and Anderson, J. (2014). Minimizing response times of automotive dataflows on multicore. In *Proceedings of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10.

Elliott, G., Ward, B., and Anderson, J. (2013). GPUSync: A framework for real-time GPU management. In *Proceedings of the 34th IEEE International Real-Time Systems Symposium*, pages 33–44.

Elliott, G., Yang, K., and Anderson, J. (2015). Supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms. In *Proceedings of the 36th IEEE International Real-Time Systems Symposium*, pages 273–284.

Enzweiler, M. and Gavrila, D. M. (2011). A multilevel mixture-of-experts framework for pedestrian classification. *IEEE Transactions on Image Processing*, 20(10):2967–2979.

Erickson, J. and Anderson, J. (2011). Response time bounds for G-EDF without intra-task precedence constraints. In *Proceedings of the 15th International Conference On Principles Of Distributed Systems*, pages 128–142.

Erickson, J. and Anderson, J. (2012). Fair lateness scheduling: Reducing maximum lateness in G-EDF-like scheduling. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 3–11.

Erickson, J., Anderson, J., and Ward, B. (2014). Fair lateness scheduling: Reducing maximum lateness in G-EDF-like scheduling. *Real-Time Systems*, 50(1):5–47.

Everingham, M., Eslami, S. A., Van Gool, L., Williams, C. K., Winn, J., and Zisserman, A. (2015). The PASCAL visual object classes challenge: A retrospective. *International Journal of Computer Vision*, 111(1):98–136.

Everingham, M., Van Gool, L., Williams, C. K., Winn, J., and Zisserman, A. (2010). The PASCAL visual object classes (VOC) challenge. *International Journal of Computer Vision*, 88(2):303–338.

Felzenszwalb, P. F., Girshick, R. B., McAllester, D., and Ramanan, D. (2009). Object detection with discriminatively trained part-based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1627–1645.

Fernandez, R. (2020). Self-driving vehicle makers say california is looking at the wrong data to measure their progress. `https://www.bizjournals.com/sanjose/news/2020/02/24/california-self-driving-car-disengagement-report.amp.html`.

Fonseca, J. C., Nélis, V., Raravi, G., and Pinho, L. M. (2015). A multi-DAG model for real-time parallel applications with conditional execution. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1925–1932.

Forsberg, B., Marongiu, A., and Benini, L. (2017). GPUguard: Towards supporting a predictable execution model for heterogeneous SoC. In *Proceedings of the 2017 Design, Automation & Test in Europe Conference & Exhibition*, pages 318–321.

Garcia-Garcia, A., Orts-Escolano, S., Oprea, S., Villena-Martinez, V., Martinez-Gonzalez, P., and Garcia-Rodriguez, J. (2018). A survey on deep learning techniques for image and video semantic segmentation. *Applied Soft Computing*, 70:41–65.

Giebel, J., Gavrila, D. M., and Schnörr, C. (2004). A bayesian framework for multi-cue 3D object tracking. In *Proceedings of the 2004 European Conference on Computer Vision*, pages 241–252.

Girshick, R. (2015). Fast R-CNN. In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, pages 1440–1448.

Girshick, R., Donahue, J., Darrell, T., and Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 580–587.

Giunta, G., Montella, R., Agrillo, G., and Coviello, G. (2010). A GPGPU transparent virtualization component for high performance computing clouds. In *Proceedings of the 2010 European Conference on Parallel Processing*, pages 379–391.

González, A., Vázquez, D., López, A. M., and Amores, J. (2016). On-board object detection: Multicue, multimodal, and multiview random forest of local experts. *IEEE Transactions on Cybernetics*, 47(11):3980–3990.

González, D., Pérez, J., Milanés, V., and Nashashibi, F. (2015). A review of motion planning techniques for automated vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 17(4):1135–1145.

Google Cloud (2020). Cloud GPUs. `https://cloud.google.com/gpu`.

Goswami, N., Shankar, R., Joshi, M., and Li, T. (2010). Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications. In *Proceedings of the 2010 IEEE International Symposium on Workload Characterization*, pages 1–10.

Green, M. (2000). "How long does it take to stop?" methodological analysis of driver perception-brake times. *Transportation Human Factors*, 2(3):195–216.

Gregg, C., Dorn, J., Hazelwood, K., and Skadron, K. (2012). Fine-grained resource sharing for concurrent GPGPU kernels. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism*, pages 1–6.

Guizzo, E. (2011). How google's self-driving car works. *IEEE Spectrum*, 18(7):1132–1141.

Gupta, K., Stuart, J. A., and Owens, J. D. (2012). A study of persistent threads style GPU programming for GPGPU workloads. In *Proceedings of the 2012 Innovative Parallel Computing*, pages 1–14.

Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V., and Ranganathan, P. (2009). GViM: GPU-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, pages 17–24.

Gupta, V., Schwan, K., Tolia, N., Talwar, V., and Ranganathan, P. (2011). Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *Proceedings of the 2011 USENIX Annual Technical Conference*, pages 3–17.

HajiRassouliha, A., Taberner, A. J., Nash, M. P., and Nielsen, P. M. (2018). Suitability of recent hardware accelerators (DSPs, FPGAs, and GPUs) for computer vision and image processing algorithms. *Signal Processing: Image Communication*, 68:101–119.

Harris, M. (2015). GPU pro tip: CUDA 7 streams simplify concurrency. `https://devblogs.nvidia.com/gpu-pro-tip-cuda-7-streams-simplify-concurrency/`.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107.

Hata, A. and Wolf, D. (2014). Road marking detection using lidar reflective intensity data and its application to vehicle localization. In *Proceedings of the 17th IEEE International Conference on Intelligent Transportation Systems*, pages 584–589.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Spatial pyramid pooling in deep convolutional networks for visual recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(9):1904–1916.

Herrera, A. (2014). NVIDIA GRID: Graphics accelerated VDI with the visual performance of a workstation. *Nvidia Corp*, pages 1–18.

Horga, A., Chattopadhyay, S., Eles, P., and Peng, Z. (2018). Measurement based execution time analysis of GPGPU programs via SE+GA. In *Proceedings of the 21st Euromicro Conference on Digital System Design*, pages 30–37.

Houdek, P., Sojka, M., and Hanzálek, Z. (2017). Towards predictable execution model on ARM-based heterogeneous platforms. In *Proceedings of the 26th IEEE International Symposium on Industrial Electronics*, pages 1297–1302.

Hu, Q., Shu, J., Fan, J., and Lu, Y. (2016). Run-time performance estimation and fairness-oriented scheduling policy for concurrent GPGPU applications. In *Proceedings of the 45th International Conference on Parallel Processing*, pages 57–66.

Huang, C., Wu, B., and Nevatia, R. (2008). Robust object tracking by hierarchical association of detection responses. In *European Conference on Computer Vision*, pages 788–801.

Huang, J., Rathod, V., Sun, C., Zhu, M., Korattikara, A., Fathi, A., Fischer, I., Wojna, Z., Song, Y., Guadarrama, S., *et al.* (2017). Speed/accuracy trade-offs for modern convolutional object detectors. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition*, pages 7310–7311.

Huang, Y.-J., Wu, H.-H., Chung, Y.-C., and Hsu, W.-C. (2016). Building a KVM-based hypervisor for a heterogeneous system architecture compliant system. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 3–15.

Huangfu, Y. and Zhang, W. (2017a). Static WCET analysis of GPUs with predictable warp scheduling. In *Proceedings of the 20th IEEE International Symposium on Real-Time Distributed Computing*, pages 101–108.

Huangfu, Y. and Zhang, W. (2017b). Warp-based load/store reordering to improve GPU time predictability. *Journal of Computing Science and Engineering*, 11(2):58–68.

Huangfu, Y. and Zhang, W. (2018). Estimating the worst-case execution time of the shared data cache in integrated CPU-GPU architectures. *Journal of Computing Science and Engineering*, 12(4):139–148.

Hughes, J. (2017). Car Autonomy Levels Explained. `http://www.thedrive.com/sheetmetal/15724/what-are-these-levels-of-autonomy-anyway`.

Hui, J. (2018). mAP (mean Average Precision) for object detection. `https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173`.

Hussain, R. and Zeadally, S. (2018). Autonomous cars: Research results, issues, and future challenges. *IEEE Communications Surveys & Tutorials*, 21(2):1275–1313.

Huynh, L. N., Lee, Y., and Balan, R. K. (2017). Deepmon: Mobile GPU-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 82–95.

Ibaez-Guzmn, J., Laugier, C., Yoder, J.-D., and Thrun, S. (2012). Autonomous driving: Context and state-of-the-art. In *Handbook of Intelligent Vehicles*, pages 1271–1310.

IEEE TCRTS (2020). Terminology and notation. `https://site.ieee.org/tcrts/education/terminology-and-notation/`.

ISO (2011). 26262: Road vehicles-functional safety. *International Standard ISO/FDIS*, 26262.

Jain, S., Baek, I., Wang, S., and Rajkumar, R. (2019). Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *Proceedings of the 25th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 29–41.

Janai, J., Güney, F., Behl, A., and Geiger, A. (2019). Computer vision for autonomous vehicles: Problems, datasets and state-of-the-art. *arXiv preprint arXiv:1704.05519v2*.

Janzén, J., Black-Schaffer, D., and Hugo, A. (2016). Partitioning GPUs for improved scalability. In *Proceedings of the 28th International Symposium on Computer Architecture and High Performance Computing*, pages 42–49.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, pages 675–678.

Jia, Z., Maggioni, M., Staiger, B., and Scarpazza, D. P. (2018). Dissecting the NVIDIA Volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*.

Jiang, X., Guan, N., Long, X., and Yi, W. (2017). Semi-federated scheduling of parallel real-time tasks on multiprocessors. In *Proceedings of the 38th IEEE International Real-Time Systems Symposium*, pages 80–91.

Jiang, X., Long, X., Guan, N., and Wan, H. (2016). On the decomposition-based global EDF scheduling of parallel real-time tasks. In *Proceedings of the 37th IEEE International Real-Time Systems Symposium*, pages 237–246.

Jochem, T. and Pomerleau, D. (1995). No hands across America. `https://www.cs.cmu.edu/~tjochem/nhaa/nhaa_home_page.html`.

Jog, A., Bolotin, E., Guz, Z., Parker, M., Keckler, S. W., Kandemir, M. T., and Das, C. R. (2014). Application-aware memory system for fair and efficient execution of concurrent GPGPU applications. In *Proceedings of the 2014 Workshop on General Purpose Processing Using GPUs*, pages 1–8.

Jog, A., Kayiran, O., Kesten, T., Pattnaik, A., Bolotin, E., Chatterjee, N., Keckler, S. W., Kandemir, M. T., and Das, C. R. (2015). Anatomy of GPU memory system for multi-application execution. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 223–234.

Jog, A., Kayiran, O., Pattnaik, A., Kandemir, M. T., Mutlu, O., Iyer, R., and Das, C. R. (2016). Exploiting core criticality for enhanced GPU performance. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, pages 351–363.

Kaehler, A. and Bradski, G. (2016). *Learning OpenCV 3: computer vision in C++ with the OpenCV library*. O'Reilly Media.

Kang, Y., Hauswald, J., Gao, C., Rovinski, A., Mudge, T., Mars, J., and Tang, L. (2017). Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News*, 45(1):615–629.

Kato, S., Lakshmanan, K., Kumar, A., Kelkar, M., Ishikawa, Y., and Rajkumar, R. (2011a). RGEM: a responsive GPGPU execution model for runtime engines. In *Proceedings of the 32nd IEEE International Real-Time Systems Symposium*, pages 57–66.

Kato, S., Lakshmanan, K., Rajkumar, R., and Ishikawa, Y. (2011b). TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the USENIX Annual Technical Conference*, pages 17–30.

Kato, S., McThrow, M., Maltzahn, C., and Brandt, S. (2012). Gdev: First-class GPU resource management in the operating system. In *Proceedings of the 2012 USENIX Annual Technical Conference*, pages 401–412.

Kayıran, O., Jog, A., Kandemir, M. T., and Das, C. R. (2013). Neither more nor less: optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 157–166.

Kendall, A., Hawke, J., Janz, D., Mazur, P., Reda, D., Allen, J.-M., Lam, V.-D., Bewley, A., and Shah, A. (2019). Learning to drive in a day. In *Proceedings of the 2019 IEEE International Conference on Robotics and Automation*, pages 8248–8254.

Kerr, A., Diamos, G., and Yalamanchili, S. (2009). A characterization and analysis of PTX kernels. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, pages 3–12.

Khronos Group (2019a). The OpenVX graph pipelining, streaming, and batch processing extension. `https://www.khronos.org/registry/OpenVX/extensions/vx_khr_pipelining/1.1/html/vx_khr_pipelining_1_1_0.html`.

Khronos Group (2019b). The OpenVX specification version 1.3. `https://www.khronos.org/registry/OpenVX/specs/1.3/html/OpenVX_Specification_1_3.html`.

Khronos Group (2020). OpenVX: Portable, power efficient vision processing. `https://www.khronos.org/openvx/`.

Kim, H., Patel, P., Wang, S., and Rajkumar, R. R. (2017). A server-based approach for predictable GPU access control. In *Proceedings of the 23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10.

Kim, H., Patel, P., Wang, S., and Rajkumar, R. R. (2018). A server-based approach for predictable GPU access with improved analysis. *Journal of Systems Architecture*, 88:97–109.

Kim, J., Kim, H., Lakshmanan, K., and Rajkumar, R. (2013). Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *Proceedings of the 4th ACM/IEEE International Conference on Cyber-Physical Systems*, pages 31–40.

Kocić, J., Jovičić, N., and Drndarević, V. (2018). Sensors and sensor fusion in autonomous vehicles. In *Proceedings of the 26th IEEE Telecommunications Forum*, pages 420–425.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105.

Kyriazis, G. (2012). Heterogeneous system architecture: A technical review. *AMD Fusion Developer Summit*, page 21.

Laina, I., Rupprecht, C., Belagiannis, V., Tombari, F., and Navab, N. (2016). Deeper depth prediction with fully convolutional residual networks. In *Proceedings of the 4th International Conference on 3D Vision*, pages 239–248.

Lakshmanan, K., Kato, S., and Rajkumar, R. (2010). Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 31st IEEE International Real-Time Systems Symposium*, pages 259–268.

Lázaro-Muñoz, A., González-Linares, J. M., Gómez-Luna, J., and Guil, N. (2017). A tasks reordering model to reduce transfers overhead on GPUs. *Journal of Parallel and Distributed Computing*, 109:258–271.

Leal-Taixé, L., Canton-Ferrer, C., and Schindler, K. (2016). Learning by tracking: Siamese CNN for robust target association. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 33–40.

Leontyev, H. and Anderson, J. (2007). Tardiness bounds for FIFO scheduling on multiprocessors. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 71–80.

Leontyev, H. and Anderson, J. H. (2010). Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Systems*, 44(1-3):26–71.

Levinson, J., Montemerlo, M., and Thrun, S. (2007). Map-based precision vehicle localization in urban environments. In *Robotics: Science and Systems*, volume 3, pages 1–8.

Levinson, J. and Thrun, S. (2010). Robust vehicle localization in urban environments using probabilistic maps. In *Proceedings of the 2010 IEEE International Conference on Robotics and Automation*, pages 4372–4378.

Li, A., van den Braak, G.-J., Kumar, A., and Corporaal, H. (2015a). Adaptive and transparent cache bypassing for GPUs. In *Proceedings of the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12.

Li, C., Song, S. L., Dai, H., Sidelnik, A., Hari, S. K. S., and Zhou, H. (2015b). Locality-driven dynamic GPU cache bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 67–77.

Li, D., Rhu, M., Johnson, D. R., O'Connor, M., Erez, M., Burger, D., Fussell, D. S., and Redder, S. W. (2015c). Priority-based cache allocation in throughput processors. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture*, pages 89–100.

Li, J., Agrawal, K., Lu, C., and Gill, C. (2013). Analysis of global EDF for parallel tasks. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 3–13.

Li, J., Ferry, D., Ahuja, S., Agrawal, K., Gill, C., and Lu, C. (2017). Mixed-criticality federated scheduling for parallel real-time tasks. *Real-Time Systems*, 53(5):760–811.

Li, J., Saifullah, A., Agrawal, K., Gill, C., and Lu, C. (2014). Analysis of federated and global scheduling for parallel real-time tasks. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, pages 85–96.

Lin, S.-C., Zhang, Y., Hsu, C.-H., Skach, M., Haque, M. E., Tang, L., and Mars, J. (2018). The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–766.

Lin, Z., Nyland, L., and Zhou, H. (2016). Enabling efficient preemption for simt architectures with lightweight context switching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 898–908.

LITMUS$^{\mathrm{RT}}$ Project (2018). `http://www.litmus-rt.org/`.

Liu, C. and Anderson, J. (2010). Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss. In *Proceedings of the 31st IEEE International Real-Time Systems Symposium*, pages 3–13.

Liu, C. and Anderson, J. H. (2011). Supporting graph-based real-time applications in distributed systems. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, volume 1, pages 143–152.

Liu, C. and Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 30(1):46–61.

Liu, J. W. (2000). *Real-Time Systems*. Prentice Hall.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., and Berg, A. C. (2016). SSD: Single shot multibox detector. In *Proceedings of the 2016 European Conference on Computer Vision*, pages 21–37.

Long, J., Shelhamer, E., and Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In *Proceedings of the 2015 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440.

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110.

Luettel, T., Himmelsbach, M., and Wuensche, H.-J. (2012). Autonomous ground vehicles—concepts and a path to the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1831–1839.

Maddern, W., Pascoe, G., Linegar, C., and Newman, P. (2017). 1 year, 1000 km: The Oxford RobotCar dataset. *The International Journal of Robotics Research*, 36(1):3–15.

Maia, C., Bertogna, M., Nogueira, L., and Pinho, L. M. (2014). Response-time analysis of synchronous parallel tasks in multiprocessor systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, pages 3–12.

Mao, M., Wen, W., Liu, X., Hu, J., Wang, D., Chen, Y., and Li, H. (2016). Temp: thread batch enabled memory partitioning for GPU. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6.

McAllister, R., Gal, Y., Kendall, A., Van Der Wilk, M., Shah, A., Cipolla, R., and Weller, A. (2017). Concrete problems for autonomous vehicle safety: Advantages of bayesian deep learning. In *Proceedings of the 25th International Joint Conferences on Artificial Intelligence*, pages 4745–4753.

McManus, C., Churchill, W., Napier, A., Davis, B., and Newman, P. (2013). Distraction suppression for vision-based pose estimation at city scales. In *Proceedings of the 2013 IEEE International Conference on Robotics and Automation*, pages 3762–3769.

Mei, X. and Chu, X. (2016). Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86.

Melani, A., Bertogna, M., Bonifaci, V., Marchetti-Spaccamela, A., and Buttazzo, G. (2016). Schedulability analysis of conditional parallel task graphs in multicore systems. *IEEE Transactions on Computers*, 66(2):339–353.

Melani, A., Bertogna, M., Bonifaci, V., Marchetti-Spaccamela, A., and Buttazzo, G. C. (2015). Response-time analysis of conditional DAG tasks in multiprocessor systems. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, pages 211–221.

Meltzer, R., Zeng, C., and Cecka, C. (2013). Micro-benchmarking the c2070. In *GPU Technology Conference*.

Menychtas, K., Shen, K., and Scott, M. L. (2013). Enabling OS research by inferring interactions in the black-box GPU stack. In *Proceedings of the 2013 USENIX Annual Technical Conference*, pages 291–296.

Menychtas, K., Shen, K., and Scott, M. L. (2014). Disengaged scheduling for fair, protected access to fast computational accelerators. *ACM SIGARCH Computer Architecture News*, 42(1):301–316.

Microsoft Azure (2020). GPU optimized virtual machine sizes. `https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-gpu`.

Milan, A., Rezatofighi, S. H., Dick, A., Reid, I., and Schindler, K. (2017). Online multi-target tracking using recurrent neural networks. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*, pages 4225–4232.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., *et al.* (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Mobus, R. and Kolbe, U. (2004). Multi-target multi-object tracking, sensor fusion of radar and infrared. In *Proceedings of the 20014 IEEE Intelligent Vehicles Symposium*, pages 732–737.

Mohan, A., Papageorgiou, C., and Poggio, T. (2001). Example-based object detection in images by components. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(4):349–361.

Mohri, M., Rostamizadeh, A., and Talwalkar, A. (2018). *Foundations of machine learning*. MIT press.

Mok, A. K.-L. (1983). *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology.

Montemerlo, M., Becker, J., Bhat, S., Dahlkamp, H., Dolgov, D., Ettinger, S., Haehnel, D., Hilden, T., Hoffmann, G., Huhnke, B., *et al.* (2008). Junior: The stanford entry in the urban challenge. *Journal of field Robotics*, 25(9):569–597.

Montgomery, W. D., Mudge, R., Groshen, E. L., Helper, S., MacDuffie, J. P., and Carson, C. (2018). America's workforce and the self-driving future: Realizing productivity gains and spurring economic growth. https://trid.trb.org/view/1516782.

Mujtaba, H. (2018). NVIDIA Drive Xavier SoC Detailed. https://wccftech.com/nvidia-drive-xavier-soc-detailed/.

Muller, U., Ben, J., Cosatto, E., Flepp, B., and Cun, Y. L. (2006). Off-road obstacle avoidance through end-to-end learning. In *Advances in Neural Information Processing Systems*, pages 739–746.

Nasri, M., Nelissen, G., and Brandenburg, B. B. (2019). Response-time analysis of limited-preemptive parallel DAG tasks under global scheduling. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems*, pages 21:1–21:23.

National Highway Traffic Safety Administration (2015). Critical reasons for crashes investigated in the national motor vehicle crash causation survey. Technical report. https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812115.

Nelissen, G., Berten, V., Goossens, J., and Milojevic, D. (2012). Techniques optimizing the number of processors to schedule multi-threaded tasks. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 321–330.

Nurvitadhi, E., Sim, J., Sheffield, D., Mishra, A., Krishnan, S., and Marr, D. (2016). Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic. In *Proceedings of the 26th International Conference on Field Programmable Logic and Applications*, pages 1–4.

NVIDIA (2009). Fermi computer architecture whiltepaper, v1.1.

NVIDIA (2012). Kepler GK110/210 computer architecture whiltepaper, v1.1.

NVIDIA (2016a). Get under the hood of Parker, our newest SoC for autonomous vehicles. https://blogs.nvidia.com/blog/2016/08/22/parker-for-self-driving-cars/.

NVIDIA (2016b). GP100 Pascal whitepaper, v1.1.

NVIDIA (2017a). Gv100 gpu hardware architecture in-depth, v1.1.

NVIDIA (2017b). Partner innovation: Accelerating automotive breakthroughs. http://www.nvidia.com/object/automotive-partner-innovation.html.

NVIDIA (2019a). CUDA C++ best practices guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

NVIDIA (2019b). CUDA C++ programming guide, v10.2.89. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

NVIDIA (2019c). CUDA toolkit documentation v10.2.89. http://docs.nvidia.com/cuda/.

NVIDIA (2019d). CUPTI: The CUDA profiling tools interface. https:docs.nvidia.com/cuda/cupti/index.html.

NVIDIA (2019e). Multi-process service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

NVIDIA (2020a). Embedded systems for next-generation autonomous machines. `https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/`.

NVIDIA (2020b). NVIDIA DRIVE AGX Developer Kit. `https://developer.nvidia.com/drive/drive-agx`.

NVIDIA (2020). Parallel thread execution ISA version 7.0. `https://docs.nvidia.com/cuda/parallel-thread-execution/index.html`.

Olah, C., Satyanarayan, A., Johnson, I., Carter, S., Schubert, L., Ye, K., and Mordvintsev, A. (2018). The building blocks of interpretability. `https://distill.pub/2018/building-blocks/`.

OpenCV (2020). OpenCV 4.0. `https://opencv.org/opencv-4-0/`.

Otterness, N., Yang, M., Amert, T., Anderson, J., and Smith, F. D. (2017a). Inferring the scheduling policies of an embedded cuda gpu. In *Proceedings of the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 47–52.

Otterness, N., Yang, M., Rust, S., Park, E., Anderson, J. H., Smith, F. D., Berg, A., and Wang, S. (2017b). An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *Proceedings of the 23th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 353–364.

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). GPU computing. *Proceedings of the IEEE*, 96(5):879–899.

Paden, B., Čáp, M., Yong, S. Z., Yershov, D., and Frazzoli, E. (2016). A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on Intelligent Vehicles*, 1(1):33–55.

Pai, S., Thazhuthaveetil, M. J., and Govindarajan, R. (2013). Improving GPGPU concurrency with elastic kernels. *ACM SIGARCH Computer Architecture News*, 41(1):407–418.

Papageorgiou, C. and Poggio, T. (2000). A trainable system for object detection. *International Journal of Computer Vision*, 38(1):15–33.

Papp, Z., Brown, C., and Bartels, C. (2008). World modeling for cooperative intelligent vehicles. In *Proceedings of the 2008 IEEE Intelligent Vehicles Symposium*, pages 1050–1055.

Park, J. J. K., Park, Y., and Mahlke, S. (2015). Chimera: Collaborative preemption for multitasking on a shared gpu. *ACM SIGARCH Computer Architecture News*, 43(1):593–606.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., *et al.* (2019). PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035.

Patole, S. M., Torlak, M., Wang, D., and Ali, M. (2017). Automotive radars: A review of signal processing techniques. *IEEE Signal Processing Magazine*, 34(2):22–35.

Phull, R., Li, C.-H., Rao, K., Cadambi, H., and Chakradhar, S. (2012). Interference-driven resource management for GPU-based heterogeneous clusters. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, pages 109–120.

Pomerleau, D. A. (1989). Alvinn: An autonomous land vehicle in a neural network. In *Advances in Neural Information Processing Systems*, pages 305–313.

Pomerleau, D. A. (1992). *Neural network perception for mobile robot guidance*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

Qamhieh, M., Fauberteau, F., George, L., and Midonnet, S. (2013). Global EDF scheduling of directed acyclic graphs on multiprocessor systems. In *Proceedings of the 21st International conference on Real-Time Networks and Systems*, pages 287–296.

Qamhieh, M., George, L., and Midonnet, S. (2014). A stretching algorithm for parallel real-time DAG tasks on multiprocessor systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, pages 13–22.

Qu, X., Soheilian, B., and Paparoditis, N. (2015). Vehicle localization using mono-camera and geo-referenced traffic signs. In *Proceedings of the 2015 IEEE Intelligent Vehicles Symposium*, pages 605–610.

Rainey, E., Villarreal, J., Dedeoglu, G., Pulli, K., Lepley, T., and Brill, F. (2014). Addressing system-level optimization with OpenVX graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 658–663.

Ramchandani, C. (1973). *Analysis of asynchronous concurrent systems by timed Petri nets.* PhD thesis, Massachusetts Institute of Technology.

Ranganathan, A., Ilstrup, D., and Wu, T. (2013). Light-weight localization for vehicles using road markings. In *Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 921–927.

Rasshofer, R. H. and Gresser, K. (2005). Automotive radar and lidar systems for next generation driver assistance functions. *Advances in Radio Science*, 3.

Ravi, V. T., Becchi, M., Agrawal, G., and Chakradhar, S. (2011). Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, pages 217–228.

Redmon, J. (2013–2016). Darknet: Open source neural networks in C. `http://pjreddie.com/darknet/`.

Redmon, J. (2017). YOLO: Real-time object detection. `https://pjreddie.com/darknet/yolov2/`.

Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788.

Redmon, J. and Farhadi, A. (2017). Yolo9000: better, faster, stronger. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition*, pages 7263–7271.

Renda, A. (2018). No need for speed when it comes to autonomous vehicles. `https://www.forbes.com/sites/washingtonbytes/2018/03/20/no-need-for-speed-when-it-comes-to-autonomous-vehicles`.

Rodvold, D. M. (1999). A software development process model for artificial neural networks in critical applications. In *Proceedings of the 1999 IEEE International Joint Conference on Neural Networks*, volume 5, pages 3317–3322.

Ross, P. E. (2017). The Audi A8: the world's first production car to achieve level 3 autonomy. *IEEE Spectrum*. `https://spectrum.ieee.org/cars-that-think/transportation/self-driving/the-audi-a8-the-worlds-first-production-car-to-achieve-level-3-autonomy`.

Rossbach, C. J., Currey, J., Silberstein, M., Ray, B., and Witchel, E. (2011). PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 233–248.

SAE (2018). *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. `https://doi.org/10.4271/J3016_201806`.

Saha, S. K., Xiang, Y., and Kim, H. (2019). STGM: Spatio-temporal GPU management for real-time tasks. In *Proceedings of the 25th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–6.

Saifullah, A., Ferry, D., Li, J., Agrawal, K., Lu, C., and Gill, C. D. (2014). Parallel real-time scheduling of DAGs. *Proceedings of the 2014 IEEE Transactions on Parallel and Distributed Systems*, 25(12):3242–3252.

Saifullah, A., Li, J., Agrawal, K., Lu, C., and Gill, C. (2013). Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49(4):404–435.

Salay, R., Queiroz, R., and Czarnecki, K. (2017). An analysis of ISO 26262: Using machine learning safely in automotive software. *arXiv preprint arXiv:1709.02435*.

Sallab, A. E., Abdou, M., Perot, E., and Yogamani, S. (2017). Deep reinforcement learning framework for autonomous driving. *Electronic Imaging*, 2017(19):70–76.

Saxena, A., Chung, S. H., and Ng, A. Y. (2006). Learning depth from single monocular images. In *Advances in Neural Information Processing Systems*, pages 1161–1168.

Saudo, I., Capodieci, N., Luis, J., Garcia, M., Marongiu, A., and Bertogna, M. (2020). Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective. In *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 213–225.

Schulter, S., Vernaza, P., Choi, W., and Chandraker, M. (2017). Deep network flow for multi-object tracking. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition*, pages 6951–6960.

Serrano, M. A., Melani, A., Bertogna, M., and Quiñones, E. (2016). Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions. In *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition*, pages 1066–1071.

Serrano, M. A., Melani, A., Kehr, S., Bertogna, M., and Quiñones, E. (2017). An analysis of lazy and eager limited preemption approaches under DAG-based global fixed priority scheduling. In *Proceedings of the 20th IEEE International Symposium on Real-Time Distributed Computing*, pages 193–202.

Sethia, A. and Mahlke, S. (2014). Equalizer: Dynamic tuning of GPU resources for efficient execution. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 647–658.

Shaffer, B. (2017). Why are automotive radar systems moving from 24GHz to 77GHz? `https://e2e.ti.com/blogs_/b/behind_the_wheel/archive/2017/10/25/why-are-automotive-radar-systems-moving-from-24ghz-to-77ghz`.

Shelhamer, E., Rakelly, K., Hoffman, J., and Darrell, T. (2016). Clockwork convnets for video semantic segmentation. In *Proceedings of the 2016 European Conference on Computer Vision*, pages 852–868.

Shi, J. *et al.* (1994). Good features to track. In *Proceedings of the 1994 IEEE Conference on Computer Vision and Pattern Recognition*, pages 593–600.

Shi, L., Chen, H., Sun, J., and Li, K. (2011). vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Transactions on Computers*, 61(6):804–816.

Spuri, M. and Buttazzo, G. C. (1994). Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 2–11.

Stuart, J. A. and Owens, J. D. (2009). Message passing on data-parallel architectures. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12.

Suhr, J. K., Jang, J., Min, D., and Jung, H. G. (2016). Sensor fusion-based low-cost vehicle localization system for complex urban environments. *IEEE Transactions on Intelligent Transportation Systems*, 18(5):1078–1086.

Sukkarieh, S., Nebot, E. M., and Durrant-Whyte, H. F. (1999). A high integrity IMU/GPS navigation loop for autonomous land vehicle applications. *IEEE transactions on robotics and automation*, 15(3):572–578.

Sun, W., Ricci, R., and Curry, M. L. (2012). GPUstore: harnessing GPU computing for storage systems in the OS kernel. In *Proceedings of the 5th Annual International Systems and Storage Conference*, pages 1–12.

Sundaram, S. (2016). DRIVE PX 2: Self driving car computer. `https://www.slideshare.net/shri123/drive-px-2`.

Suzuki, Y., Fujii, Y., Azumi, T., Nishio, N., and Kato, S. (2016). Real-time GPU resource management with loadable kernel modules. *IEEE Transactions on Parallel and Distributed Systems*, 28(6):1715–1727.

Suzuki, Y., Kato, S., Yamada, H., and Kono, K. (2014). GPUvm: Why not virtualizing GPUs at the hypervisor? In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 109–120.

Tanasic, I., Gelado, I., Cabezas, J., Ramirez, A., Navarro, N., and Valero, M. (2014). Enabling preemptive multiprogramming on GPUs. *ACM SIGARCH Computer Architecture News*, 42(3):193–204.

Tang, S., Andres, B., Andriluka, M., and Schiele, B. (2016). Multi-person tracking by multicut and deep matching. In *Proceedings of the 2016 European Conference on Computer Vision*, pages 100–111.

Texas Instrument (2018). TMS320C6000 DSP Library (DSPLIB). `https://wccftech.com/nvidia-drive-xavier-soc-detailed/://www.ti.com/tool/SPRC265`.

The Apache Software Foundation (ASF) (2020). Apache MXNet: A flexible and efficient library for deep learning. `https://mxnet.apache.org/`.

Thorpe, C., Hebert, M. H., Kanade, T., and Shafer, S. A. (1988). Vision and navigation for the carnegie-mellon navlab. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(3):362–373.

Thorpe, C., Herbert, M., Kanade, T., and Shafer, S. (1991a). Toward autonomous driving: the CMU Navlab. i. perception. *IEEE expert*, 6(4):31–42.

Thorpe, C., Herbert, M., Kanade, T., and Shafter, S. (1991b). Toward autonomous driving: the cmu navlab. ii. architecture and systems. *IEEE expert*, 6(4):44–52.

Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Hoffmann, G., *et al.* (2006). Stanley: The robot that won the darpa grand challenge. *Journal of field Robotics*, 23(9):661–692.

Tran, D., Bourdev, L., Fergus, R., Torresani, L., and Paluri, M. (2015). Learning spatiotemporal features with 3d convolutional networks. In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, pages 4489–4497.

Tzeng, S., Patney, A., and Owens, J. D. (2010). Task management for irregular-parallel workloads on the GPU. page 2937.

Ubal, R., Jang, B., Mistry, P., Schaa, D., and Kaeli, D. (2012). Multi2sim: a simulation framework for CPU-GPU computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 335–344.

Uijlings, J. R., Van De Sande, K. E., Gevers, T., and Smeulders, A. W. (2013). Selective search for object recognition. *International Journal of Computer Vision*, 104(2):154–171.

Ukidave, Y., Kalra, C., Kaeli, D., Mistry, P., and Schaa, D. (2014). Runtime support for adaptive spatial partitioning and inter-kernel communication on GPUs. In *Proceedings of the 26th IEEE International Symposium on Computer Architecture and High Performance Computing*, pages 168–175.

Ukidave, Y., Li, X., and Kaeli, D. (2016). Mystic: Predictive scheduling for gpu based cloud servers using machine learning. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium*, pages 353–362.

Ulmer, B. (1994). VITA II—active collision avoidance in real traffic. In *Proceedings of the 1994 IEEE Intelligent Vehicles Symposium*, pages 1–6.

Urmson, C., Anhalt, J., Bagnell, D., Baker, C., Bittner, R., Clark, M., Dolan, J., Duggins, D., Galatali, T., Geyer, C., *et al.* (2008). Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466.

Urmson, C., Ragusa, C., Ray, D., Anhalt, J., Bartz, D., Galatali, T., Gutierrez, A., Johnston, J., Harbaugh, S., "Yu" Kato, H., *et al.* (2006). A robust approach to high-speed navigation for unrehearsed desert terrain. *Journal of Field Robotics*, 23(8):467–508.

Viola, P. and Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Conference on Computer Vision and Pattern Recognition*, pages 511–518.

Volkov, V. and Demmel, J. W. (2008). Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–11.

Wang, G., Lin, Y., and Yi, W. (2010). Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In *Proceedings of the 2010 IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing*, pages 344–350.

Wang, Z., Yang, J., Melhem, R., Childers, B., Zhang, Y., and Guo, M. (2016). Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture*, pages 358–369.

Warren, M. E. (2019). Automotive lidar technology. In *Proceedings of the 2019 IEEE Symposium on VLSI Circuits*, pages C254–C255.

Waymo (2020). Waymo safety report – on the road to fully self-driving. `https://waymo.com/safety/`.

Wei, J., Snider, J. M., Kim, J., Dolan, J. M., Rajkumar, R., and Litkouhi, B. (2013). Towards a viable autonomous driving research platform. In *Proceedings of the 2013 IEEE Intelligent Vehicles Symposium*, pages 763–770.

Wen, Y., O'Boyle, M. F., and Fensch, C. (2018). MaxPair: enhance OpenCL concurrent kernel execution by weighted maximum matching. In *Proceedings of the 11th Workshop on General Purpose GPUs*, pages 40–49.

Wende, F., Cordes, F., and Steinke, T. (2012). On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering. In *Proceedings of the 2012 Symposium on Application Accelerators in High Performance Computing*, pages 74–83.

Williams, M. (1988). PROMETHEUS—the european research programme for optimising the road transport system in europe. In *IEE Colloquium on Driver Information*, pages 1/1–1/9.

Wolcott, R. W. and Eustice, R. M. (2014). Visual localization within lidar maps for automated urban driving. In *Proceedings of the 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 176–183.

Wong, H., Papadopoulou, M.-M., Sadooghi-Alvandi, M., and Moshovos, A. (2010). Demystifying GPU microarchitecture through microbenchmarking. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 235–246.

Wu, B., Chen, G., Li, D., Shen, X., and Vetter, J. (2015). Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 119–130.

Wu, B. and Nevatia, R. (2007). Detection and tracking of multiple, partially occluded humans by bayesian combination of edgelet based part detectors. *International Journal of Computer Vision*, 75(2):247–266.

Xiang, Y. and Kim, H. (2019). Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In *Proceedings of the 40th IEEE International Real-Time Systems Symposium*, pages 392–405.

X.org Foundation (2014). Memory mapped I/O trace. `https://nouveau.freedesktop.org://nouveau.freedesktop.org/wiki/MmioTrace/`.

X.org Foundation (2020). Nouveau: Accelerated open source driver for nVidia cards. `https://nouveau.freedesktop.org/`.

Xu, Q., Jeon, H., Kim, K., Ro, W. W., and Annavaram, M. (2016a). Warped-slicer: efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming. In *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture*, pages 230–242.

Xu, Y., Wang, R., Li, T., Song, M., Gao, L., Luan, Z., and Qian, D. (2016b). Scheduling tasks with mixed timing constraints in GPU-powered real-time systems. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–13.

Yang, K. and Anderson, J. (2014a). Optimal GEDF-based schedulers that allow intra-task parallelism on heterogeneous multiprocessors. In *Proceedings of the 12th IEEE Symposium on Embedded Systems for Real-Time Multimedia*, pages 30–39.

Yang, K. and Anderson, J. H. (2014b). Optimal GEDF-based schedulers that allow intra-task parallelism on heterogeneous multiprocessors. In *Proceedings of the 12th IEEE Symposium on Embedded Systems for Real-time Multimedia*, pages 30–39.

Yang, K., Elliott, G., and Anderson, J. (2015). Analysis for supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms. In *Proceedings of the 23rd International Conference on Real-Time Networks and Systems*, pages 77–86.

Yang, K., Yang, M., and Anderson, J. (2016a). Reducing response-time bounds for DAG-based task systems on heterogeneous multicore platforms. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pages 349–358.

Yang, K., Yang, M., and Anderson, J. H. (2016b). Reducing response-time bounds for DAG-based task systems on heterogeneous multicore platforms. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pages 349–358.

Yang, M., Amert, T., Yang, K., Otterness, N., Anderson, J. H., Smith, F. D., and Wang, S. (2018a). Making OpenVX really "real time". In *Proceedings of the 39th IEEE International Real-Time Systems Symposium*, pages 80–93.

Yang, M., Otterness, N., Amert, T., Bakita, J., Anderson, J. H., and Smith, F. D. (2018b). Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems*, pages 20:1–20:21.

Yang, M., Wang, S., Bakita, J., Vu, T., Smith, F. D., Anderson, J. H., and Frahm, J.-M. (2019). Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In *Proceedings of the 25th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 305–317.

Yurtsever, E., Lambert, J., Carballo, A., and Takeda, K. (2019). A survey of autonomous driving: common practices and emerging technologies. *arXiv preprint arXiv:1906.05113*.

Zhang, F., Stähle, H., Chen, G., Simon, C. C. C., Buckl, C., and Knoll, A. (2012). A sensor fusion approach for localization with cumulative error elimination. In *Proceedings of the 2012 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, pages 1–6.

Zhao, Z.-Q., Zheng, P., Xu, S.-t., and Wu, X. (2019). Object detection with deep learning: A review. *IEEE Transactions on Neural Networks and Learning Systems*, 30(11):3212–3232.

Zhong, J. and He, B. (2013). Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1522–1532.

Zhou, H., Bateni, S., and Liu, C. (2018). S^{3}DNN: Supervised streaming and scheduling for GPU-accelerated real-time DNN workloads. In *Proceedings of the 24th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 190–201.

Zhou, H., Tong, G., and Liu, C. (2015). GPES: A preemptive execution system for GPGPU computing. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 87–97.

Zhou, Y. and Tuzel, O. (2018). Voxelnet: End-to-end learning for point cloud based 3d object detection. In *Proceedings of the 2018 IEEE Conference on Computer Vision and Pattern Recognition*, pages 4490–4499.

Ziegler, J., Bender, P., Schreiber, M., Lategahn, H., Strauss, T., Stiller, C., Dang, T., Franke, U., Appenrodt, N., Keller, C. G., *et al.* (2014). Making bertha drivean autonomous journey on a historic route. *IEEE Intelligent Transportation Systems Magazine*, 6(2):8–20.

Zou, Z., Shi, Z., Guo, Y., and Ye, J. (2019). Object detection in 20 years: A survey. *arXiv preprint arXiv:1905.05055*.