Chauffeur: Benchmark Suite for Design and End-to-End Analysis of Self-Driving Vehicles on Embedded Systems

BISWADIP MAITY, SAEHANSEUL YI, DONGJOO SEO, and LEMING CHENG,
University of California, Irvine, USA
SUNG-SOO LIM and JONG-CHAN KIM, Kookmin University, Korea
BRYAN DONYANAVARD, San Diego State University, USA
NIKIL DUTT, University of California, Irvine, USA

Self-driving systems execute an ensemble of different self-driving workloads on embedded systems in an end-to-end manner, subject to functional and performance requirements. To enable exploration, optimization, and end-to-end evaluation on different embedded platforms, system designers critically need a benchmark suite that enables flexible and seamless configuration of self-driving scenarios, which realistically reflects real-world self-driving workloads' unique characteristics. Existing CPU and GPU embedded benchmark suites typically (1) consider isolated applications, (2) are not sensor-driven, and (3) are unable to support emerging self-driving applications that simultaneously utilize CPUs and GPUs with stringent timing requirements. On the other hand, full-system self-driving simulators (e.g., AUTOWARE, APOLLO) focus on functional simulation, but lack the ability to evaluate the self-driving software stack on various embedded platforms. To address design needs, we present Chauffeur, the first open-source end-to-end benchmark suite for self-driving vehicles with configurable representative workloads. Chauffeur is easy to configure and run, enabling researchers to evaluate different platform configurations and explore alternative instantiations of the self-driving software pipeline. Chauffeur runs on diverse emerging platforms and exploits heterogeneous onboard resources. Our initial characterization of Chauffeur on different embedded platforms – NVIDIA Jetson TX2 and Drive PX2 – enables comparative evaluation of these GPU platforms in executing an end-to-end self-driving computational pipeline to assess the end-to-end response times on these emerging embedded platforms while also creating opportunities to create application gangs for better response times. Chauffeur enables researchers to benchmark representative self-driving workloads and flexibly compose them for different self-driving scenarios to explore end-to-end tradeoffs between design constraints, power budget, real-time performance requirements, and accuracy of applications.

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2021. This work was partially supported by NSF grant CCF-1704859. This work was also supported by the MSIT (Ministry of Science, ICT), Korea, under the High-Potential Individuals Global Training Program)(2019-0-01607) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation).
Authors' addresses: B. Maity, S. Yi, D. Seo, L. Cheng, and N. Dutt, Donald Bren School of Information and Computer Sciences, University of California, Irvine, Irvine, CA, 92697, USA; emails: {maityb, saehansy, dseo3, lemingc}@uci.edu, dutt@ics.uci.edu; S.-S. Lim, School of Computer Science, Kookmin University, Seoul, Korea; email: ssllim@kookmin.ac.kr; J.-C. Kim, Department of Automobile and IT Convergence, Kookmin University, Seoul, Korea; email: jongchank@kookmin.ac.kr; B. Donyanavard, Department of Computer Science, San Diego State University, 5500 Campanile Drive, San Diego, CA, 92182, USA; email: bdonyanavard@sdsu.edu.

1539-9087/2021/09-ART74 $15.00
https://doi.org/10.1145/3477005

CCS Concepts: • **Computer systems organization** → **Embedded systems**; • **Hardware** → *Power and energy*; • **Computing methodologies** → Graphics systems and interfaces;

Additional Key Words and Phrases: Autonomous vehicles, benchmark suite, self-driving vehicles

ACM Reference format:
Biswadip Maity, Saehanseul Yi, Dongjoo Seo, Leming Cheng, Sung-Soo Lim, Jong-Chan Kim, Bryan Donyanavard, and Nikil Dutt. 2021. Chauffeur: Benchmark Suite for Design and End-to-End Analysis of Self-Driving Vehicles on Embedded Systems. *ACM Trans. Embedd. Comput. Syst.* 20, 5s, Article 74 (September 2021), 22 pages.
https://doi.org/10.1145/3477005

1 INTRODUCTION

Self-driving systems have seen an increasing demand in the recent decade fueled by investments from the industry as well as traction within research communities. For example, *Autonomous transportation* could reduce accidents and time spent in traffic by reducing human intervention while driving. *Warehouse robots* work alongside humans to increase the safety and efficiency of a factory, and have motivated advancements in both algorithmic and hardware technology. The cyber-physical functionalities in self-driving systems are achieved by executing an ensemble of autonomy applications on embedded systems in an end-to-end manner. While the orchestration of the end-to-end workloads is already extremely complex, providing functional guarantees and performance constraints becomes extremely challenging due to the stringent safety requirements. System designers of feasible self-driving solutions critically require a benchmark suite that represents the self-driving workloads and enables exploration, optimization, and end-to-end evaluation on different self-driving platforms. To further cater to the increased proliferation and diversity of self-driving applications, the benchmark suite must enable flexible and seamless configuration of self-driving scenarios. However, architecting such systems on resource-constrained low-power embedded platforms remains an open research challenge, as there remains a disconnect between academic research and real-world constraints for deployment.

Platform designers typically use existing CPU and GPU embedded benchmark suites to identify resource bottlenecks and tune and/or design future hardware. However, these existing CPU and GPU benchmark suites typically (1) consider isolated applications that do not represent the end-to-end workload pipeline, (2) are not sensor-driven, and (3) utilize synthetic applications that are not representative of self-driving workloads, and therefore, cannot simultaneously utilize CPUs and GPUs with stringent timing requirements. At the application level, application designers typically use full-system self-driving simulators (e.g., AUTOWARE, APOLLO) that focus on functional simulation and are often oblivious to the platforms on which the software executes; thus, it is not feasible to deploy application stacks with performance requirements in embedded hardware to meet/explore Size, Weight, Area and Power (SWAP) constraints. We seek to bridge the gap between algorithms and platforms for self-driving applications to aid both system and platform designers to evaluate and explore practical self-driving solutions.

To this end, we present Chauffeur, the first open-source end-to-end benchmark suite for self-driving vehicles with configurable representative workloads. Chauffeur consists of publicly available popular implementations of state-of-the-art real-world self-driving vehicle algorithms. Rather than focusing on a single software stack for an end-to-end use case, Chauffeur enables flexible composition of realistic workloads comprising multiple components of the perception-control pipeline. Chauffeur is easy to configure and run, enabling researchers to evaluate different platform configurations and explore alternative instantiations of the self-driving software pipeline. We illustrate

the use of Chauffeur through analysis of a typical full-system self-driving application behavior executing on diverse embedded NVIDIA platforms. The comparative evaluation of these GPU platforms for executing representative workloads enables system designers to assess the implications of executing the sensing-perception-planning-control pipeline on these emerging embedded platforms. In particular, we demonstrate the utility of Chauffeur by evaluating the schedulability of self-driving workloads on shared resources. Chauffeur thus enables researchers to benchmark representative self-driving workloads and flexibly compose them for different self-driving scenarios to explore end-to-end tradeoffs between design constraints, power budget, real-time performance requirements, and accuracy of applications.

The main contributions of this paper are summarized as follows:

(1) Chauffeur, a benchmark suite for embedded systems that allows hardware designers, system software engineers, and researchers to profile and characterize **configurable end-to-end** pipelines of self-driving workloads easily.
(2) Characterization of Chauffeur applications on two exemplar ARM-based embedded platforms from NVIDIA comprising of Pascal embedded GPUs: (1) Jetson TX2 (GPU with shared memory), and (2) Drive PX2 (GPU with discrete memory).
(3) Exemplar Chauffeur use case demonstrating how system designers can exploit parallel execution on multicore systems to minimize the end-to-end delay of Chauffeur applications.
(4) Public release of source code[1] along with complete documentation and profiling scripts to easily download and run Chauffeur.

Organization: The rest of the paper is organized as follows. Section 2 introduces the self-driving software pipeline and the limitations of existing state-of-the-art benchmarks for designing self-driving systems. Section 3 presents details of the Chauffeur benchmark suite with an emphasis on the representative software modules required for self-driving vehicles. Section 4 describes the target embedded platforms on which the benchmark suite is tested. Section 5 evaluates the performance of the Chauffeur benchmarks and the architectural implications for hardware designers. Section 6 demonstrates an example Chauffeur use-case of end-to-end delay minimization. Finally, Section 7 concludes the paper.

2 MOTIVATION

Designing self-driving vehicles requires the orchestration of an ensemble of different self-driving workloads working in an end-to-end manner, guaranteeing functional and performance constraints. Figure 1 shows the task graph of a generic end-to-end self-driving software pipeline. Based on this representative task graph, we investigate and formalize the requirements of a benchmark suite for self-driving vehicles geared toward system designers and researchers.

2.1 End-to-end Pipeline for Self-driving Vehicles

A self-driving system must plan and follow a trajectory to reach a given destination using real-time sensor information [12, 46, 54]. Consider Figure 1 where we present a generic software pipeline for operating a self-driving vehicle. The pipeline shown here is implementation-independent and captures the logical computation blocks (tasks) and data flow pipeline (relation between tasks). We can divide the pipeline into four stages: Sensing, Perception, Planning, and Actuation. (1) Sensing is composed of three tasks associated with collecting real-time data from hardware sensors: Camera grabber, LIDAR/RADAR sensing, and CANbus polling. These tasks are responsible for reading raw sensor data connected using either (a) Automotive Ethernet [21], or (b) Controller

[1] https://github.com/duttresearchgroup/Chauffeur.

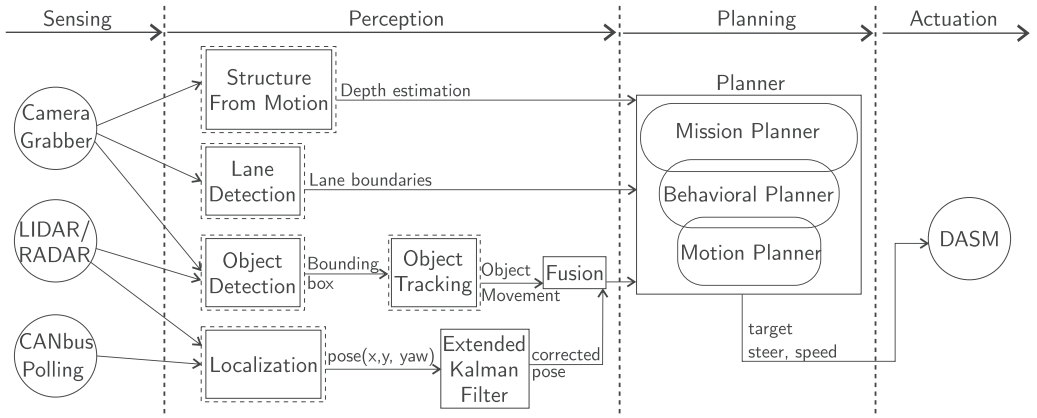


Fig. 1. Software components and flow of information in a self-driving vehicle. Interfaces with sensors and actuators are marked with a circle.

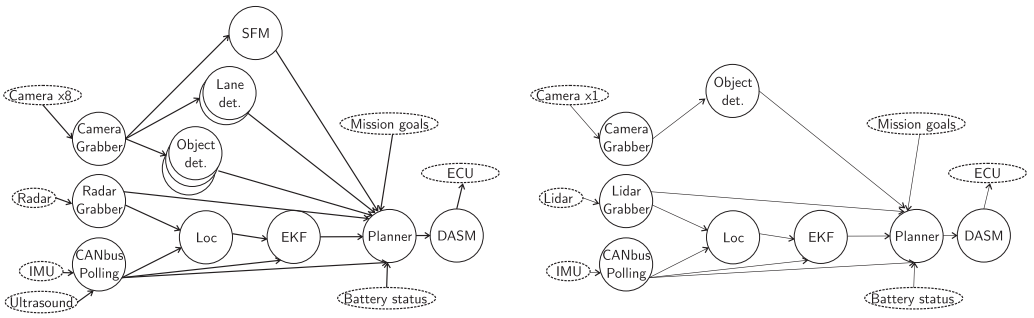
Area Network (CAN bus). (2) The Perception stage uses the real-time raw information to extract relevant environmental knowledge for decision-making. The tasks involved are (a) depth estimation (Structure From Motion), (b) detection of lane boundaries (Lane Detection), (c) bounding and identification of surrounding objects (Object Detection), (d) tracking of detected objects, and (e) position estimation (Localization). More details on these tasks are present in Section 3. (4) In the Planning stage, the planner uses the perception information to determine the future trajectory by setting the target steer and speed. (5) Finally, the Drivers Assistance System Module (DASM) uses CANbus to perform the hardware actuation and control the vehicle.

The generic self-driving software pipeline in Figure 1 can be instantiated in different ways. Vendor implementations vary in (a) the number and types of sensors, (b) the number of instances of software tasks, and (c) the underlying hardware on which the stack is running. For instance, Tesla's Full Self-Driving (FSD) computer [49], is equipped with eight cameras and uses a radar distance sensor along with multiple ultrasound sensors surrounding the car (Figure 2(a)) for autonomous transportation. On the other hand, self-driving warehouse robots have different requirements: 8 cameras surrounding the vehicle are excessive, and vendors often deploy a LIDAR sensor instead of cameras [50]; and Lane-Detection is no longer required. An example pipeline representing such a scenario is shown in Figure 2(b). Although the task graph has a different instantiation, the end-to-end pipeline will still need to be designed effectively for the hardware on which it is running.

2.2 Requirements of a Benchmark Suite for Self-driving Vehicles

Designing self-driving vehicles requires cross-layer collaboration from researchers in diverse domains, including hardware designers, system software architects, and application developers. Running a full-stack end-to-end simulation is often time-consuming and infeasible on resource-constrained embedded boards. Instead, we focus on flexibly composing a computational pipeline of critical software components representing real autonomous driving workloads. To support the flexible exploration of end-to-end configurations, we identify the following requirements for a benchmark suite for self-driving vehicles:

Self-driving workloads. Self-driving workloads consist of sensor-driven, resource-intensive applications with real-time safety-critical requirements[39]. In Section 3.1, we describe the applications which constitute a typical self-driving scenario. Future systems require a redesign



(a) Example instance for urban driving scenario. (b) Example instance for warehouse robotics scenario.

Fig. 2. Implementation-specific instances of the generic self-driving pipeline with different tasks.

of the hardware and heterogeneous resources to meet these applications' real-time performance requirements.

Configurable end-to-end pipeline. The performance and efficiency of autonomous driving systems require researchers to analyze the end-to-end pipeline: sensing, perception, planning, and actuation. This requires studying different end-to-end pipelines. Very few works [25] [15] [53] [29] [55] have been able to study the system performance of end-to-end pipelines as the monolithic software stacks are too complex to deploy on an embedded platform. To the best of our knowledge, there is no configurable end-to-end pipeline that can be used to study self-driving workloads.

Embedded Runtime. A key challenge in designing self-driving vehicles is that resource-hungry applications consume a lot of power. This is critical for battery-powered vehicles where users need high mileage operation from a single charge. Several optimizations have been made for such platforms (e.g., using shared physical memory between the CPU and GPU). However, real-time performance requirements, energy constraints, and safety are extremely hard to satisfy simultaneously. While trading off performance for energy can cause problems in the vehicle's safe operation, having energy optimizations as an afterthought leads to a poor design. For practical deployment, it is essential to evaluate these workloads on resource-constrained platforms.

Heterogeneity. Integrated GPUs are already ubiquitous. Future systems trend is to meet the real-time performance requirements by accelerating the bottlenecks in applications either through GPUs or dedicated on-device hardware accelerators for specific kernels. Consequently, applications must support multiple, heterogeneous resources (e.g., CPU version, GPU version) and utilize any available resource at runtime. Fickenscher et al. in [16] have analyzed automated code generation for self-driving algorithms on these heterogeneous platforms with the help of Domain-Specific Language (DSL).

Diverse platforms. As applications embrace heterogeneity, vendors have developed various platforms to support their runtime. Notably, heterogeneous multi-cores augmented with graphics processing units (GPUs) as accelerators show promise to meet real-time self-driving workloads' performance requirements. NVIDIA strongly advocates such an approach through their series of embedded platforms like Jetson TX2 (integrated GPU with shared physical memory, not aimed for safety-critical applications) and Drive PX2 (discrete GPU, provides dual-modular redundancy features for safety-critical applications). A benchmark suite should not only be representative of the real-time workloads but also support these types of diverse embedded platforms.

Research support. Open-source benchmarks that are easy to download and run are critical to engage and support research in this area. Chauffeur would enable hardware designers and system researchers to go beyond full stack driving simulators that are difficult to run on low-resource boards. We provide the infrastructure to instrument and derive the performance implications on state-of-the-art systems efficiently.

2.3 Related Work: Limitations of Existing Benchmark Suites

Table 1 summarizes the status of existing benchmark suites, showing how they fail to meet all of the requirements discussed above, while showing the capabilities of Chauffeur; we expand on Table 1 below.

Traditional embedded benchmark suites such as PARSEC [10], MiBench [19], SPEC [24] include programs from different domains like computer vision, media processing, enterprise servers, animation physics. These benchmarking suites mainly focus on computer architecture and system/hardware design. However, they fail to capture the implications of highly data-intensive applications having stringent performance requirements that constitute a typical self-driving scenario. These benchmarks either focus solely either on CPU workloads (e.g., PARSEC) or on GPU workloads (e.g., Rodinia [14]). The narrow scope fails to capture the correct ratio of heterogeneous resource utilization in self-driving vehicles.

CAVBench is a benchmark suite targeted towards evaluating autonomous driving computing system performance [51] in a connected vehicle setting. It focuses on six workloads: SLAM, object detection, object tracking, battery diagnostics, speech recognition, and edge video analysis. CAVBench analyzes the execution time breakdown for each application and the Quality of Service (QoS)—Resource Utilization (RU) curve (QoS-RU curve). The QoS-RU curve is used to calculate the matching factor (MF) between the application and the computing platform on autonomous vehicles. CAVBench serves as an initial artifact to study edge computing systems for autonomous driving but fails to present a holistic view of the end-to-end self-driving scenario.

Autoware [27] is a popular open-source full-stack driving simulator that is expected to be deployed on autonomous vehicles. Autoware is based on Robot Operating System (ROS) and other well-established open-source software libraries. However, full-stack simulators typically require powerful platforms (e.g., recommended system for evaluating Autoware is an 8-core X866 CPU with 32GB of main memory, which is infeasible for embedded platforms). Researchers have redesigned Autoware to customize the software stack to run on NVIDIA DRIVE PX2 computing platform to study self-driving workloads [28]. However, such customizations restrict hardware designers to studying a single workload (one specific implementation) rather than different algorithms for the same task. Significant customization is required to port such a complex stack to emerging embedded platforms.

This problem is further exacerbated in **industry-standard** autonomous driving software frameworks like **Apollo** [6]. These frameworks are developed with the application (self-driving vehicle) in mind. However, operating such stacks on low-resource embedded hardware requires careful system design by researchers to account for architectural implications [32]. Hardware designers and system researchers find it cumbersome and time-consuming to set up an end-to-end driving stack to study hardware/architectural implications.

Chauffeur incorporates the features highlighted in Table 1 using a set of benchmarks aimed at hardware designers and system researchers. It comprises state-of-the-art representative applications from the domain of self-driving vehicles and targets low-resource embedded systems. Chauffeur is open-source and easy to download and run, enabling quick analysis of system implications of a configurable end-to-end self-driving pipeline.

Table 1. Popular Benchmark Suites and the Key Challenges Addressed

Features	Traditional benchmarks [10, 19, 24]	CAVBench [51]	Autoware [27, 28]	Apollo [2]	Chauffeur
Self-driving workloads		✓	✓	✓	✓
Configurable end-to-end pipeline					*
Embedded Runtime	✓		✓		✓
Heterogeneity		✓	✓	✓	✓
Diverse Platforms	✓				✓
Supports research	✓	✓	✓	✓	✓

(* = uniquely addressed by Chauffeur).

3 CHAUFFEUR: BENCHMARK SUITE FOR DESIGNING SELF-DRIVING VEHICLES

Chauffeur is a benchmark suite comprising some essential applications for designing self-driving vehicles, with the ability to expand with additional applications. We categorize the (initial ten) applications in Chauffeur into four stages: (1) Sensing, (2) Perception, (3) Planning, and (4) Actuation. Sensing applications receive sensory information from different communication buses and share it with the rest of the pipeline. Perception applications collect the raw information shared by sensing applications and extract relevant knowledge used for decision making [41]. Perception provides a contextual understanding of the vehicle's environment (e.g., what the different objects are, location of the objects, road signs, traffic cones), and the vehicle's position with respect to the environment. The planning application develops and continuously updates the vehicle's trajectory to achieve the higher-level goals of the user (e.g., driving from Redmond to Seattle) while following the rules of the road. Finally, actuation applications control the vehicle and execute the planned actions generated by the previous stage. The applications defined in Chauffeur are not vendor/implementation-specific and apply to different self-driving use cases (e.g., autonomous driving versus warehouse robots). Table 2 summarizes these applications, inputs and outputs, and captures the relationship among the applications.

3.1 Description of Self-driving Application Categories

Camera Grabber. Visual sensors (e.g., CMOS camera) are a vital component to enable perception about the environment in self-driving vehicles. Typically, visual sensors generate a lot of data and require a high-bandwidth communication bus. This requirement is incredibly stringent when interfacing multiple cameras simultaneously, as shown in Figure 2(a). Car manufacturers use automotive ethernet to transfer such high volume data with very low latency and meet real-time performance requirements. The sensing application *camera grabber* is a representative workload for such processes. The camera grabber is responsible for receiving the packets from the network and then placing them on the main memory for the following stages.

LIDAR/RADAR. Although cameras are reliable and relatively cheap to produce, perception solely using camera data is non-trivial as it relies on black-box neural networks. Traditionally, car manufacturers use detection and ranging sensors (e.g., LIDAR, RADAR) to detect surrounding objects and calculate distances. Distance sensors assist in hazard detection and range-finding in features like adaptive cruise control (ACC) and automatic emergency braking (AEB). These sensors are imperative during adverse weather and lighting conditions and are still prevalent in modern vehicles.

Table 2. Applications in a Typical Self-driving Vehicle; Highlighting Inputs and Outputs and How Different Applications are Related

Applications	Stage	Input	Output
Camera Grabber	Sensing	Packets over Automotive Ethernet	(S1) Image frames in the shared memory
LIDAR/RADAR	Sensing	Packets over Automotive Ethernet	(S2) Point cloud in the shared memory
CAN bus pooling	Sensing	Messages (Frames) over CAN bus	(S3) Sensed information in shared memory
Structure From Motion	Perception	(S1)	(P1) Depth estimation
Lane Detection	Perception	(S1)	(P2) Lane Boundaries
Object Detection	Perception	(S1), (S2)	(I1) Bounding Box
Object Tracking	Perception	(I1)	(I2) Object movement
Localization	Perception	(S2)	(I3) position and orientation pose(x, y, yaw)
Extended Kalman Filter	Perception	(I3), (S3)	(I4) Corrected pose
Fusion	Perception	(I2), (I4)	(P3) Fused object and vehicle location
Path planner	Planning	(P1), (P2), (P3)	(A1) Spatio-temporal trajectory
DASM	Actuation	(A1)	Steer, Brake

CAN bus Polling. The Controller Area Network (CAN bus) is another integral interface found in automobiles that automobile engineers use to interface with the vehicle's hardware. CAN bus is typically used for low-volume data transfers with high reliability. It serves the following purposes: (1) reading the current status of the vehicle (e.g., Odometer value), (2) interfacing with CAN bus sensors (e.g., Inertial Measurement Unit (IMU)), and (3) controlling the steer and speed of the vehicle.

Structure From Motion (SFM). SFM is a perception application that aims to reconstruct three-dimensional structures from a sequence of two-dimensional moving images [22]. SFM uses the subsequent images to triangulate the 3D position of objects in the environment.

Lane Detection. Lane detection is a perception application that aims to detect road boundaries (lane line markings) and estimate the vehicle pose with respect to the detected lines using visual sensors on the vehicle. The application includes the localization of the road, the determination of the relative position between vehicle and road, and the analysis of the vehicle's heading direction [8].

Object Detection. Vision-based object detection is a perception application that is one of the primary prerequisites for self-driving vehicles. Distance and ranging sensors (e.g., LIDAR, RADAR) alone are not sufficient to meet the requirements of self-driving. For example, RADAR sensors, albeit working in adverse environmental conditions, do not produce a high precision output. On the other hand, information from LIDAR sensors, albeit extremely precise, are too complicated to process and prohibitively expensive. Modern object detection applications use neural networks along with visual sensor data to identify objects in the area surrounding the vehicle by drawing bounding boxes and classifying the object inside each bounding box.

Object Tracking. Given some objects of interest marked in a frame, the object tracking application locates the objects in subsequent frames in the video [23]. Object tracking is a part of the

perception stage, and it tracks objects as they move in the environment. It also allows the self-driving vehicle to estimate the motion of objects and predict how they will move in the subsequent frames.

Localization. Localization is a perception application that works closely with the environmental perception using visual sensors to identify the vehicle's position within the perceived environment. Global Positioning System (GPS) is the most commonly used localization system used in the vehicle industry. Although cheap and easily accessible, GPS suffers from poor reliability and accuracy and is not a good candidate for localization applications in self-driving vehicles [31]. Researchers have developed advanced sensors (e.g., RADAR, LIDAR, Visual sensors) that can further be fused to provide more robust, accurate, and reliable localization used in modern vehicles.

Extended Kalman Filter (EKF). The result of localization (using distance sensor like RADAR) is prone to drift over time and can be noisy. A Kalman filter is an excellent candidate for combining the distance information with other vehicle-status information (e.g., IMU, Odometer, GPS) and handling such disturbances. Kalman filter fuses multiple data sources and performs continuous prediction (for missing data) and correction (for drifting data) on the localization results. The EKF application is the non-linear implementation of the Kalman filter [42].

Fusion. The fusion task helps combine object information with the car's location on the fly. The information is sourced by pre-processing raw data from different sensors (e.g., cameras, different types of RADAR, LIDAR) and fused synchronously. The fusion process involves transformation of different coordinate systems and updating the environment maps periodically in real-time. These tasks have inherent data parallelism and are good targets for hardware acceleration [17].

Path Planner. The planning stage is responsible for understanding higher-level goals from the user (e.g., Navigate from Seattle to Redmond) and convert them to purposeful decisions to achieve the higher-level goals while avoiding obstacles [41]. The complexity of this stage compels a hierarchical design by partitioning the software into layers: (1) Mission planning, (2) Behavioral planning, and (3) Motion planning. *Mission planning* computes the global trajectory based on the current location and the target destination along with stops and which roads can be taken to achieve this application (e.g., avoid freeways). *Behavioral planning* generates local objectives (e.g., Change lanes, overtake) to interact with other agents on the path and follow the rules of the road. *Motion planning* takes the local objectives and generates the control plan to actuate the steering and speed. Although most recent works [7, 44, 47] follow some implementation of the planner hierarchy, the exact partitioning within the planner often varies between implementation.

Drivers Assistance System Module (DASM). The DASM application performs the final actuation stage in the pipeline. The local objectives of the path planner (through motion planner) are realized through a PID controller. The PID controller actuates the speed and steering based on the velocity and angle commands and can automatically use the odometer and IMU feedback to maintain the targets set by the path planning stage.

3.2 Why is Chauffeur Useful

The lack of a complete collection of an open-source and configurable set of autonomous vehicle applications motivated us to develop Chauffeur. The existing benchmarks [2, 27, 28, 51] are focused on a rigid stack without a configurable end-to-end pipeline. For example, Lin et al. [32] identifies the architectural implications of autonomous driving by creating an infrastructure for an end-to-end pipeline. However, such pipelines are specific to one implementation with very specific workloads. Traditionally application developers focus on the application stack, whereas platform

Table 3. Implementations Used in Chauffeur

Implementation	Application	Dataset	Data Size and Type
ROS	Camera Grabber	application-specific	variable
ROS	LIDAR/RADAR	application-specific	variable
OpenMVG [36]	Structure From Motion	provided	360° 5376 × 2688 color images
Jetson Inference [38]	Object Detection	cuda-lane-detection	30FPS h.264 1280 × 720
darknet-ros [11]	Object Detection	KITTI Odometry Dataset [18]	1392 × 512 color images
lidar-tracking [40]	Object Detection & Tracking	KITTI Odometry Dataset	3D Velodyne point cloud 100k points per frame
LaneNet [33, 52]	Lane Detection	tusimple-benchmark	1280 × 720 color images
cuda-lane-detection [26]	Lane Detection	provided	1280 × 720 30FPS h.264 video
FLOAM [20]	Localization	KITTI Odometry Dataset	3D Velodyne point cloud 100k points per frame
orb-slam-3 [13]	Localization	KITTI Odometry Dataset	1392 × 512 greyscale images
EKF [45]	Extended Kalman Filter	provided	radar and lidar pose estimates
Hybrid A* [30]	Path planner	provided	2D obstacle map

developers determine if a given computing platform is competent for the application stack. Thus, hardware is an afterthought. Our Chauffeur approach is the opposite: provide a set of benchmarks that are representative applications that enable hardware designers and system software engineers to observe system implications and plan future hardware. The application implementations we chose for Chauffeur are presented in Table 3.

Chauffeur enables opportunities for multiple research directions:

- Chauffeur enables researchers to compare different implementations of the same application (e.g., *jetson-inference* vs *darknet* for object detection) in the pipeline and observe their implications on the underlying system.
- Chauffeur is configurable and is capable of launching diverse applications as well as multiple instantiations of the same application. Configurability can potentially help identify bottlenecks in different instantiations of the self-driving pipeline (Figure 1) that are good candidates for accelerators.
- Chauffeur enables researchers to study different pipelines and explore optimization techniques; It opens the door for future runtime policies to govern the system and have dynamic policies if the pipeline changes (e.g., number of cameras changed leading to a heavier implementation of object detection).
- Chauffeur enables researchers to explore workload partitions across different resources (e.g., CPU and GPU) and understand the implications of data dependencies between tasks.
- Chauffeur is easy to configure and run and is accompanied by dockerized cross-compiling infrastructure. This helps researchers easily deploy Chauffeur on emerging embedded systems.
- End-to-end analysis: Despite a significant reduction in inference times using neural networks, the current end-to-end delay in self-driving systems is not satisfactory [25]. Chauffeur provides a configurable environment for end-to-end delay optimization not only in the perception stage but also considering sensing, planning, and actuation stages as illustrated in Section 6.
- Convenient evaluation: Currently, several works rely on synthetic data due to the complexity of setting up driving simulators (e.g., Saifullah et al. in [43] used an empty for-loop to measure energy consumption for their proposed methods). Chauffeur will serve as a real-world benchmark suite that researchers can use to compare results.

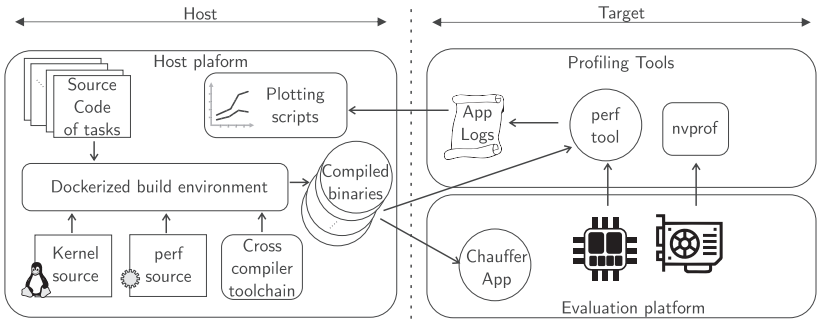


Fig. 3. Tool flow for using Chauffeur suite.

- Future versions will support integrations with real sensors and simulators, automated instantiation of the end-to-end pipeline, as well as connected, networked autonomous systems.

One of the goals that we pursue with the Chauffeur benchmark suite is to identify the system implications of the representative applications on emerging embedded platforms. We offer insights obtained by profiling the applications under different configurations (e.g., integrated vs shared main memory, different degrees of parallelism). We believe this will serve as a good starting point for researchers to design computing platforms for future self-driving vehicles.

4 EVALUATION FRAMEWORK USING CHAUFFEUR

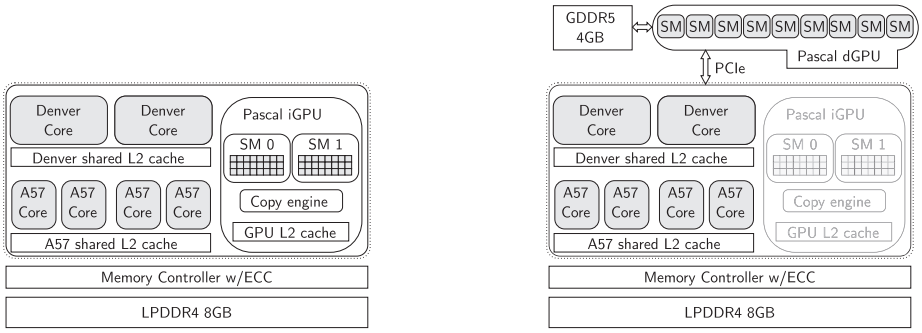
A significant challenge faced by system designers when using existing software stacks to analyze end-to-end performance bottlenecks and exploration of different platform configurations is the complexity of configuring and running them on embedded platforms. Chauffeur overcomes these barriers through a tool flow that enables researchers to analyze and evaluate different platforms quickly. We describe the Chauffeur tool flow and present an example of this tool flow for evaluating end-to-end performance evaluation.

4.1 Tool Flow

Figure 3 illustrates the Chauffeur tool flow across the host and target platforms. Users have two options to compile applications for the target platform: (1) directly compile the source on the board, or (2) use cross-compilation. We provide a dockerized build environment that builds the source code of applications and includes necessary dependencies. The application binaries are then deployed on the target platform and profiled using different tools. We use *perf* for IPC and CPU performance counters, *nvprof* and *NSight Systems* for GPU profiling. The tools are used to understand the implications of the end-to-end pipeline on the system. We include the scripts used for compiling, executing, and profiling as part of the repository. The compilation, deployment, and profiling steps are currently not completely automated and must be performed in a step-by-step fashion, manually, as explained in the repository.

4.2 Sample Experimental Evaluation platforms

We illustrate the use of Chauffeur to comparatively evaluate application execution on two exemplar embedded hardware platforms from NVIDIA: (a) NVIDIA Jetson TX2 platform, and (b) NVIDIA Drive PX2 platform. These emerging embedded platforms are widely adopted in many self-driving use cases (e.g., warehouse robotics, Tesla's Autopilot Hardware 2.0). Due to the



(a) Jetson TX2 Architecture (figure adapted from [5]). Parker SoC includes integrated GPU (iGPU) with shared main memory.

(b) Drive PX2 Architecture: Parker SoC with discrete Pascal GPU connected with PCIe. iGPU is not used for experiments.

Fig. 4. Architecture of exemplar NVIDIA evaluated platforms.

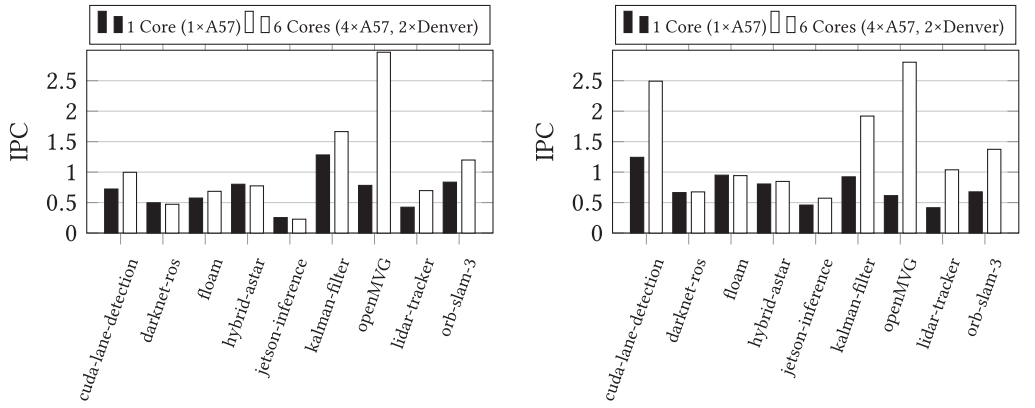
widespread adoption of these platforms, prototype design and product performance evaluation are effortless as researchers can compare different policies against the same hardware.

Figure 4(a) shows the architecture of the Jetson TX2. The TX2 consists of a Parker system on a chip (SoC) with two super Denver (NVidia proprietary) cores and four big A57 (ARM) cores. The Parker SoC includes an integrated Pascal GPU (iGPU) with two Streaming Multiprocessors (128 cores each). The CPU and GPU share 8GB LPDDR4 main memory. The Linux version used is 4.9.140-tegra, and the CUDA runtime library version is 10.0. Figure 4(b) shows the architecture of the Drive PX2. Like the TX2, the PX2 has a Parker SoC with two super Denver cores and four big A57 cores. We do not use the on-chip Pascal iGPU (grayed-out in the figure) in our experiments, as the Drive PX2 has a more powerful discrete GPU (dGPU). As a result, the entire 8GB LPDDR4 main memory is dedicated to the CPU. The dGPU consists of nine Streaming Multiprocessors (128 cores each), and is connected to the Tegra SoC using a PCIe bus. The dGPU also has a dedicated 4GB of GDDR5 memory. The Linux version used is 4.9.80-rt61-tegra, and the CUDA runtime library version is 9.2. Although the Drive PX2 supports dual modular redundancy by providing two instances of the described hardware architecture, we do not use the second Parker SoC/dGPU for our experiments. We run all experiments in maximum performance mode by disabling the dynamic frequency scaling of CPU cores and GPU.

The experimental platforms encompass diverse memory layout, and GPU compute capability. The shared main memory of Jetson TX2 creates two challenges at runtime: (1) memory space is a limiter for parallel applications when the end-to-end pipeline is considered, (2) memory contention between CPU and GPU workloads creates a memory bandwidth bottleneck. The discrete GPU on the Drive PX2 can alleviate challenges (1) and (2) in some cases. Still, the cost of memory copies before launching and after finishing kernel execution may outweigh the benefits. We explore some of these challenges by analyzing the Chauffeur applications that form the end-to-end self-driving pipeline.

5 CHARACTERIZATION OF CHAUFFEUR APPLICATIONS

To demonstrate the utility and flexibility of Chauffeur, we characterize Chauffeur applications and observe the performance implications of these Chauffeur applications on the two exemplar NVIDIA embedded platforms, to generate takeaways that can guide the exploration of different end-to-end self-driving pipelines. First, we report the utilization of the compute micro-architecture.



(a) Jetson TX2.

(b) Drive PX2.

Fig. 5. Comparison of *instructions per cycle* (IPC) (averaged across all cores) across different NVIDIA embedded platforms.

Then, we identify the resource bottleneck for each application. Finally, we present the power breakdown among the different resources. Identification of performance and power bottlenecks of Chauffeur applications serve as guidance for future optimizations.

5.1 Performance of Chauffeur Applications

Our first goal is to observe how well applications can utilize platform CPU's micro-architecture. We use *instructions per cycle* (IPC) to show on average how many instructions the CPU can retire in each clock cycle. Improving IPC directly translates to lowering execution time for the application, critical for performance in self-driving applications. The Cortex A57 cores in our exemplar NVIDIA platforms (Figure 4) contain a 3-wide decoder front-end for fetching instructions. Meanwhile, the Denver ("super") cores are implemented by NVIDIA and have a 7-wide decoder width. The maximum theoretical IPC for A57 cores is 3, and Denver cores are 7. Applications can exhibit a low IPC for various reasons: (a) the processor pipeline is not able to fetch enough instructions for the execution stage, (b) incorrect speculations, or (c) not enough resources to retire instructions (e.g., not enough cores, high cache misses). Typically IPC > 1 indicates that the application is instruction-bound (bottle-necked by code execution on CPU cores). IPC < 1 shows some resource (e.g., Memory, GPU) is stalling code execution, and further investigation is required to identify the bottle-necked resource.

Figure 5 shows the observed IPC of Chauffeur applications. The periodic applications are operated in a data-ready mode to discount any idle periods. Figure 5(a) is the result of execution on Jetson TX2 and Figure 5(b) is the result of execution on Drive PX2. We make the following observations: (1) average IPC for six cores (1.4) > average IPC for one core (0.8). While this confirms the intuition that applications, in general, perform better with more cores, we explore effective speedup from parallelization in more detail in Section 5.2. (2) Some applications (e.g., openMVG, kalman-filter, orb-slam-3) have much better IPC when the super cores are enabled. Therefore, it is better to map them to super cores instead of big cores. (3) Object detection applications suffer from the lowest average IPC across all core configurations (average jetson-inference IPC is 0.42, average darknet-ros IPC is 0.61). We explore this in detail in Section 5.3.

Takeaways: *(1) We need to investigate the degree of parallelism of high IPC applications (cuda-lane-detection, openMVG) (2) For the applications with a low IPC but running on a GPU, we need a full system analysis for identifying if GPU is the actual bottleneck. (3) For the remaining applications, we need to look into the memory access behavior.*

5.2 Effective Speedup From Parallelism

Applications with large inputs and working sets are typically good candidates for exploiting parallelism. We explore Chauffeur applications' ability to exploit CPU parallelism by increasing the number of online CPU cores. Figure 6 shows the results in terms of speedup. We make the following observations: (1) All applications gain speedup when increasing from 1 to 2 cores. For applications that are not explicitly parallelized (e.g., hybrid-star, jetson-inference, kalman-filter), this benefit comes from multicore execution thanks to reduced contention even in single-threaded implementations. (2) Certain applications (e.g., kalman filter, lidar-tracker) perform better when supercores are enabled (increasing from 4 to 5 cores). They can benefit from supercores' powerful floating-point units or larger L1 cache; (3) Some applications (e.g., OpenMVG, lidar-tracker) show linear speedup by increasing from 1 to 6 cores. This results from parallelization (OpenCV multi-threaded APIs for lidar-tracker and OpenMP for OpenMVG). (4) openMVG (SFM) experiences up to 3.3× speedup on Jetson TX2 and up to 3.9× speedup on Drive PX2 when compared to a single-core execution. The application operates on large images, is multi-threaded, and data-parallel. Hence it benefits a lot from multiple cores. However, the current implementation does not leverage the GPU. (5) jetson-inference (Object detection) experiences no speedup on the Jetson TX2 and up to 1.3× speedup on the Drive PX2. Although it spends 52% time executing the decoder thread on CPU, a large number of *memcpy* operations limits the degree of parallelism. We further investigate the difference in speedup across platforms in Section 5.3. (6) darknet-ros (Object detection) does not show any speedup with an increasing number of cores. We conclude it is not core bound. Candidate backend bottlenecks for darknet-ros include memory and GPU (further investigated in Section 5.3). (7) floam (Localization) is a compute-intensive application with the current implementation running only on CPU. The execution time to process one input on Drive PX2 is around ≈150ms and is not affected much by increasing cores. However, on Jetson TX2, we see it starts poorly (≈250ms for one core), improves with more cores (≈106ms for four cores), but performs worse with super cores. We performed a finer-grained analysis and found the source: a high number of branch mispredictions in super cores (≈16 Million/s) as opposed to big cores (≈13 Million/s).

Takeaways: (1) *floam (Localization) and openMVG (Structure from Motion) implementations are CPU (core) bound in the micro-architecture pipeline. (2) Further study is required for the remaining perception applications to identify their bottleneck. (3) Bigger cores with wider instruction decode paths are not always a better choice for running applications, as bad speculations in modern CPUs can cause severe degradation in performance.*

5.3 Architectural Implications of GPU Applications

GPU-accelerated applications achieve speedup by utilizing hundreds of small cores. Typically, perception applications are compute-intensive and promising targets for GPU acceleration. However, application developers are often unaware of details of the GPU architecture and cannot exploit the full potential of the device. Since GPU micro-architecture varies across platforms, designers often need to perform several optimizations before an application can meet the performance constraints.

From the previous section, we observed that although perception applications typically process large amounts of data, some of the worst-performing apps are object detection (darknet-ros,

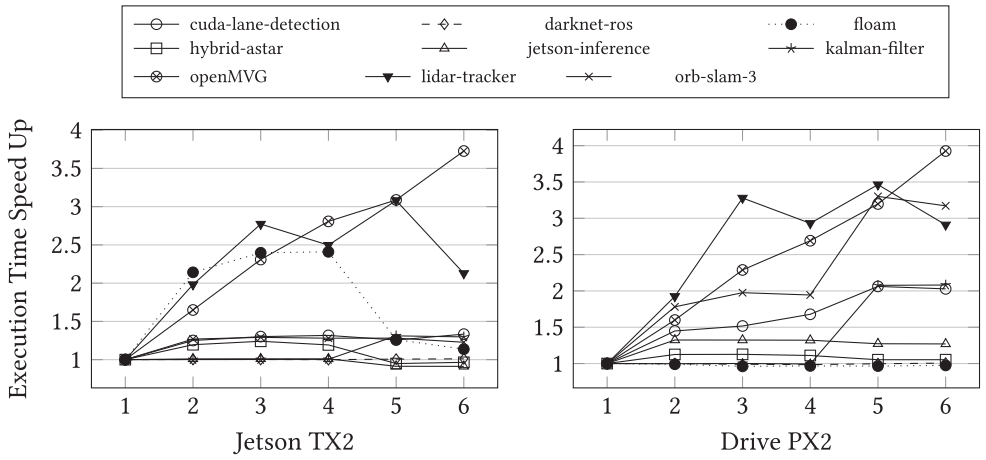


Fig. 6. Speed-up of application execution time with increasing number of online cores.

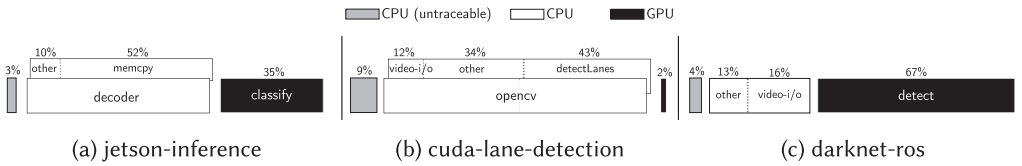
jetson-inference) and lane detection.[2] Therefore we analyze GPU implementations of these applications to identify the resource bottleneck for the applications running on the NVIDIA CUDA cores. The GPU on the two platforms, Jetson TX2 and Drive PX2, have different compute capacities. Unlike ARM PMU counters, which can be profiled at a very fine granularity, GPUs typically lack such fine-grained performance counters, presumably because of the overwhelming number of cores. Instead, there are aggregated counters for a group of cores or for all the cores. We leverage two tools for GPU profiling: (1) NVIDIA Nsight Systems, which gives performance-breakdown of the various threads along with a detailed timeline of the resource utilization; and (2) nvprof, which reports performance-breakdown by the various kernels (as opposed to threads). In embedded platforms, nvprof can only report aggregated metrics at the end of the execution.

We analyze the following applications: (1) jetson-inference, which takes a video as input and detects objects in each frame using a deep neural network (DNN). While processing DNN is a heavy operation, other operations in the application, such as decoding and grayscaling, are also present. (2) cuda-lane-detection, that uses the Hough transform to detect lanes in images. (3) darknet-ros, a ROS based version of another popular DNN object detection application. We use ROS to enable periodic execution of applications and separate the sensor grabber processes from the data processing, which we profile. This incurs a slight overhead (<2%) of inter-process communication, which we believe represents actual communication from sensor grabber to the periodic invocation of processing applications.

Figure 7 presents an overview of the major (most time-consuming) threads obtained by full-system profiling of the GPU applications on Jetson TX2. The first row shows major components' share of execution time, and the second row represents a breakdown of each component in detail (where applicable).

5.3.1 jetson-inference. (Figure 7(a)) We make the following observations: (1) jetson-inference involves a lot of memory operations (*memcpy*), which account for 52% of the execution time. (2) Although classification uses CUDA GPUs, it only accounts for 35% of execution time. (3) If higher

[2]Although we include Lanenet, a popular lane detection application, we were unable to profile it with any of the NVIDIA tools. Thus we do not have it in the analysis.



(a) jetson-inference (b) cuda-lane-detection (c) darknet-ros

Fig. 7. Nsight Profiling Results on Jetson TX2 showing % of time spend on GPU vs. CPU for running threads.

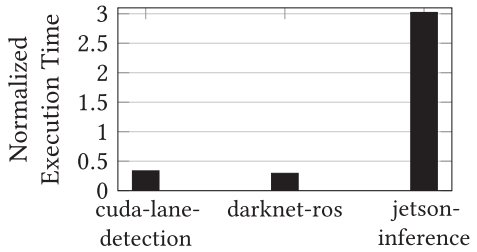


Fig. 8. Execution time of GPU kernels on Drive PX2 normalized to Jetson TX2.

FPS is required for object detection, optimizing the decoder rather than tuning the GPU kernel is better. **Takeaway:** *jetson-inference is memory bound.*

5.3.2 cuda-lane-detection. (Figure 7(b)) We make the following observations: (1) cuda-lane-detection is extremely CPU-heavy and spends about 97% of computation time in CPU. (2) cuda-lane-detection utilizes all six cores in the system very well with multi-threaded image operations in OpenCV. Section 5.2 confirms the utilization by the high degree of parallelism. (3) Only the Hough Transform is designed to use GPU, leaving the GPU cores highly underutilized. (4) We performed a finer-grained analysis and found that the GPU kernel hardly uses DRAM bandwidth nor saturate occupancy. **Takeaway:** *Although cuda-lane-detection uses GPU kernels, it is CPU bound. Developers didn't use GPU code for OpenCV; as a result, 89% of execution time is spent in OpenCV, which could be offloaded to GPU.*

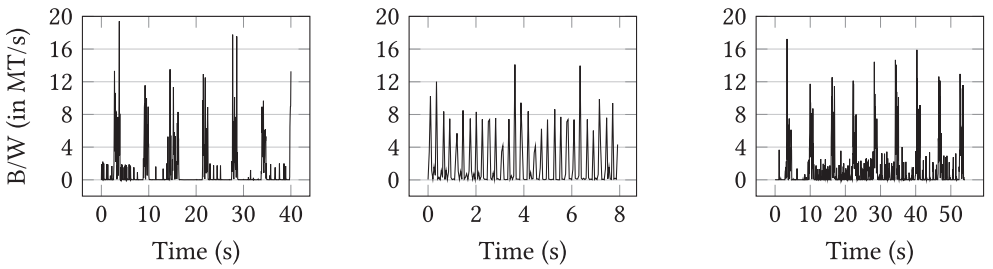
5.3.3 darknet-ros. (Figure 7(c)) We make the following observations: (1) darknet-ros creates dedicated threads to detect objects in each frame. (2) The overhead of ROS (part of others block in Figure 7(c)) is extremely negligible (< 2%). Meanwhile, unlike the previous applications, the overhead of profiling (still <3%) has increased slightly due to extra management performed when a thread is created/destroyed. (3) The thread creation and destroy overhead could be insightful when designing timing-sensitive real-time systems as it will create jitter in terms of scheduling and context switches. (4) darknet-ros is well-designed to utilize the GPU. However, the execution is not well optimized, causing the GPU to be the bottleneck in the execution time. We further investigate the energy implications of darkent-ros in Section 5.5. **Takeaway:** *darknet-ros is GPU bound. It is reasonable to optimize the GPU kernel or to design a system with a more efficient hardware accelerator to achieve better performance.*

We performed the above performance analysis on the Jetson TX2. We further compare these applications' performance across the different NVIDIA platforms in Figure 8. While cuda-lane-detection and darknet-ros confirm the intuition that a bigger dGPU in Drive PX2 performs better than a smaller iGPU in Jetson TX2, the increased execution time of jetson-inference seems an anomaly at first. Careful analysis shows jetson-inference supports multiple versions of the TensorRT library. TX2 supports a newer version (6.0.1.10) of TensorRT which facilitates faster FP16

Table 4. Comparison of Memory Access and Main Memory (DRAM) Bandwidth (B/W) of Chauffeur Applications on the Jetson TX2

Transfer (MT/s)	idle	cuda-lane-detection	darknet-ros	floam	hybrid-astar	jetson-inference	kalman-filter	lanenet-lane-detection	lidar-tracking	OpenMVG	orb-slam-3
Avg Access	9	717	120	534	499	199	1348	537	934	1852	1346
Peak Access	55	2703	944	1703	1318	1110	2820	2353	1512	7225	2088
Avg B/W	0	1	1	4	1	3	0	1	10	15	18
Peak B/W	0	19	14	25	16	17	13	17	21	89	67

Measured numbers are only from CPU performance counters and do not consider memory traffic from GPU. Unit is *million-transfers/sec (MT/s).*



(a) cuda-lane-detection (b) darknet-ros (c) lanenet-lane-detection

Fig. 9. Main-memory access pattern of selected Chauffeur applications from CPU cores. Applications demonstrate memory accesses phases.

operations, while PX2 (having TensorRT version 4.0.0.8) does not. FP16 can boost both arithmetic and memory operations compared to FP32. As a result, the GPU kernels are three times faster on the TX2 platform, leading to a 1.3× overall speedup as seen in Section 5.2 on PX2.

5.4 Main Memory Access by CPU

Table 4 shows the memory access characteristics of Chauffeur applications. We conduct these experiments on Jetson TX2 will all six cores active using MARS framework [37]. We make the following observations: (1) Although applications issue many memory requests, most accesses are served by the cache hierarchy (L1 and L2 cache). Thus, memory access rates are much higher than main memory bandwidth rates. (2) openMVG has the highest memory access rate and main memory peak bandwidth utilization. The high number of memory requests is a result of data-parallelism (as shown in Section 5.2) as openMVG exploits all cores. The CPU cores' fast and parallel computation on large inputs leads to numerous memory accesses, which increases the main memory bandwidth. (3) Several applications report extremely low average main memory bandwidth. This can be explained by looking at the memory access patterns over time (Figure 9). These applications have phases of memory accesses, causing a surge of memory bandwidth requirement followed by a computation phase (in CPU/GPU). **Takeaways:** *Memory access patterns are highly dynamic and very hard to model at design time. Runtime policies need to observe the memory access patterns and avoid overlapping memory phases between applications to reduce contention when running the end-to-end pipeline.*

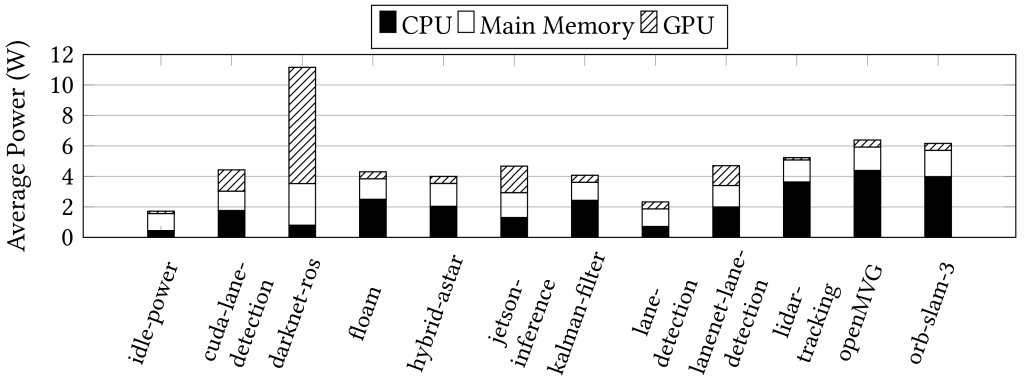


Fig. 10. Power breakdown for Jetson TX2 using onboard I2C power sensors.

5.5 Power Profile of Applications

The average instantaneous power is a direct representation of utilization of the onboard resources and demonstrates opportunities for future optimizations. Figure 10 shows the instantaneous power breakdown of Chauffeur applications on the Jetson TX2. We could not provide a comparative study due to the lack of power sensors in the Drive PX2 platform. We make the following observations: (1) darknet-ros reports the highest GPU power consumption of 7.6W on the iGPU. We observed in Section 5.3.3 that the performance bottleneck of darknet-ros is GPU (67% execution time spent on GPU). (2) openMVG reports the highest CPU power of consumption of 4.4W on the iGPU. This is expected, as we observed in Section 5.2 that openMVG exhibits a high degree of parallelism (up to 3.7× speedup) with the increase in cores, and the reported results are for six cores. (3) Memory requires a more profound investigation for optimizing power. Chauffeur applications are a good target for accuracy/power tradeoffs, as some sensors might be more relevant than others depending on the scenario.

Takeaways: *(1) darknet-ros justifies hardware acceleration. (2) openMVG justifies GPU acceleration. (3) Approximate memory techniques [35], and efficient memory management techniques [9, 34] should be explored to reduce memory power.*

6 EXEMPLAR SCENARIO: USING CHAUFFEUR FOR END-TO-END PERFORMANCE CONSTRAINT EVALUATION OF AUTONOMOUS URBAN DRIVING

Chauffeur is also essential when analyzing and optimizing the end-to-end performance of autonomous driving systems. Figure 1 shows the generic end-to-end application pipeline for self-driving systems for which we have characterized individual applications using Chauffeur. Chauffeur is an ensemble of interdependent real-world applications that researchers can utilize to evaluate the end-to-end self-driving pipeline as a whole. To this end, we further demonstrate a utility of Chauffeur by using observations and takeaways made in previous sections for end-to-end performance analysis of the autonomous urban driving scenario shown in Figure 2(a).

When periodic applications are scheduled on multicore CPUs, timing requirement failures may occur due to unpredictable inter-core interference. The severity of interference for complex workloads can prove infeasible to anticipate accurately. Recently, RT-gang [4] was proposed to address this complexity. In RT-gang scheduling, applications are grouped into multiple periodic gangs, limiting concurrent scheduling to a single gang. In other words, an application is allowed to run in parallel only with a predefined set of other applications. Using RT-gang, system designers focus only on the interference between gang members, reducing the complexity of analysis. RT-gang is

Table 5. Example Per-input Execution Time of Applications with Single Core in Milliseconds

	Camera Grabber(1)	Lidar Grabber(2)	CAN bus polling(3)	Structure from Motion(4)	Object Detection(5)	Lane Detection(6)	Localization (7)	EKF (8)	Planning (9)	DASM (10)
TX2	1.0	10.9	0.6	326.5	272.0	37.3	256.9	2.67	63.0	10.0
DPX2	1.0	10.9	0.6	439.0	102.5	45.2	147.2	3.9	70.0	10.0

Table 6. Gang Partitioning Using Table 6: Gang Configurations, Gang Periods, and Min. End-to-end Response Time

	gang1	gang2	gang3	gang p_1	gang p_2	gang p_3	Min. E2E(ms)
TX2	4,5,6,7,9	1,2,3,8,10	-	399.6	59.6	-	1956.0
DPX2	4,5,6,7	9	1,2,3,8,10	734.1	293.1	66.8	2455.3

also useful with GPU applications. GPU execution time is difficult to bound because of the limited control over proprietary hardware. Existing CPU-GPU inference minimization techniques [1, 3] could be even more effective on bounded inference provided by RT-gang. However, current techniques to form gangs do not consider a key performance metric in periodic systems: end-to-end response time, i.e., the latency from sensors to actuators, which is derived from gang periods. Intuitively, applications with similar execution times should be grouped together in order to minimize core idle time. However, if dependent applications are in the same gang, the gang needs consecutive runs to propagate the data as the producer and consumer applications are running concurrently, resulting in an increased end-to-end response time.

Chauffeur is an excellent candidate for the evaluation of such end-to-end performance constraints. The diverse workloads in Chauffeur make it perfect for measuring end-to-end response time while considering the data dependencies among applications. To illustrate the potential of Chauffeur, we evaluate various gang configurations of the task graph in Figure 2(a), instantiated with exactly one instance of each application. We use the application execution times shown in Table 5 to calculate gang periods and end-to-end response time. We observe that the perception stage is the biggest bottleneck in both Jetson TX2 and Drive PX2. Although the perception stage remains the biggest bottleneck in the pipeline, fine-grained bottleneck analysis is dependent on the pipeline under study and the evaluation platform. Note that the sensing and actuation stage values in Table 5 are an estimation due to the lack of ready sensors on the platforms.

Because gang partitioning is an NP-hard problem, we devise a simple greedy algorithm to create gangs verified as optimal through brute force on a smaller set of applications. We report the resulting gang partitioning in Table 6. We can see that differences in execution time of individual applications between platforms (e.g., 112 ms for SFM) can result in a 500 ms difference in end-to-end response time. Using Chauffeur, we can obtain estimates of the execution times of self-driving applications and evaluate the end-to-end response time on different systems. We can further improve response time by exploiting parallel execution of applications, i.e., assigning multiple cores to an application. Mapping an application to multiple cores can decrease the response time but potentially impact overall response time by limiting resources available to other applications. We pick SFM as a target for parallelization because it proved beneficial in our effective speedup analysis (Figure 6).

Using the speedup information from Section 5.2, we vary the number of cores assigned to SFM and apply the same greedy algorithm. As a result, we further minimize the response time on each platform when an additional core is allocated to SFM. Figure 11 shows the minimum end-to-end response time with varying numbers of cores assigned to SFM. We make the following

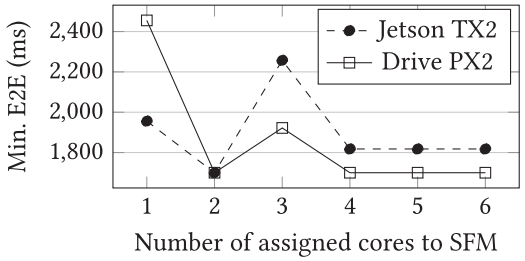


Fig. 11. Minimum end-to-end response time with varying numbers of cores to SFM.

observations: (1) Although SFM's execution time decreases with more cores, that is not always the case with the end-to-end response time. (2) After the allocation of 2 cores to SFM on Jetson TX2, a further increase in the number of cores leads to worse end-to-end minimum response times. (3) When a higher number of cores are allocated (3-6) to SFM, the Drive PX2 offers a lower minimum end-to-end response time with the same strategy.

We demonstrated that using Chauffeur, we can better understand the end-to-end response time of self-driving applications on different systems. Chauffeur provides representative workloads, an effortless workflow of environment setup, and profiling scripts to make these kinds of evaluations possible with minimal effort from researchers. Chauffeur enables researchers to benchmark representative self-driving workloads and flexibly compose them for different self-driving scenarios to explore end-to-end tradeoffs between design constraints, power budget, performance requirements, and accuracy of applications.

7 CONCLUSIONS

We presented Chauffeur, an open-source benchmark suite with representative applications for the entire self-driving perception-control pipeline. Chauffeur includes a tool flow that supports compilation and execution for two exemplar embedded platforms: the Nvidia Jetson TX2 and the Nvidia Drive PX2, and can easily be adapted to other platforms. Using these exemplar embedded platforms, we perform an analysis of system behavior of the Chauffeur applications and derive takeaways on the performance and power bottlenecks of the CPU, GPU, and memory behavior that can be flexibly used to analyze and explore different end-to-end self-driving pipeline configurations. We demonstrate the utility of Chauffeur by using measured behavior to analyze the real-time schedulability of the end-to-end self-driving workload on a resource-constrained battery-powered platform for an autonomous urban driving scenario. Based on our observations, it is likely that effective deployment of self-driving workloads will ultimately necessitate custom hardware [48], but designers can identify bottlenecks and candidates for optimization if realistic algorithms are evaluated on real embedded platforms. Chauffeur provides an essential toolkit for self-driving vehicle designers, and we plan to maintain its relevance and utility by continuing to update the representative inputs, algorithms, implementations, and workload composition in future versions.

REFERENCES

[1] H. Aghilinasab, W. Ali, H. Yun, and R. Pellizzoni. 2020. Dynamic memory bandwidth allocation for real-time GPU-based SoC platforms. *IEEE TCADICS* (2020), 3348–3360.

[2] M. Alcon, H. Tabani, L. Kosmidis, E. Mezzetti, J. Abella, and F. J. Cazorla. 2020. Timing of autonomous driving software: Problem analysis and prospects for future solutions. In *Proc. IEEE RTAS*. 267–280.

[3] W. Ali and H. Yun. 2018. Protecting real-time GPU kernels on integrated CPU-GPU SoC platforms. In *Proc. ECRTS*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 19:1–19:22.

[4] W. Ali and H. Yun. 2019. RT-Gang: Real-Time gang scheduling framework for safety-critical systems. In *Proc. IEEE RTAS*. 143–155.
[5] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith. 2017. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *Proc. IEEE RTSS*. 104–115.
[6] Apollo. 2019. An open autonomous driving platform, source-code and manuals. https://github.com/ApolloAuto/apollo.
[7] S. Aradi. 2020. Survey of deep reinforcement learning for motion planning of autonomous vehicles. *IEEE TITS* (2020), 1–20.
[8] A. A. Assidiq, O. O. Khalifa, M. R. Islam, and S. Khan. 2008. Real time lane detection for autonomous vehicles. In *Proc. CCCE*. 82–88.
[9] S. Bateni, Z. Wang, Y. Zhu, Y. Hu, and C. Liu. 2020. Co-optimizing performance and memory footprint via integrated cpu/gpu memory management, an implementation on autonomous driving platform. In *Proc. IEEE RTAS*. 310–323.
[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC benchmark suite. In *Proc. of PACT*. ACM Press.
[11] M. Bjelonic. 2016–2018. YOLO ROS: Real-Time Object Detection for ROS. https://github.com/leggedrobotics/darknet_ros.
[12] H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom. 2020. nuscenes: A multimodal dataset for autonomous driving. In *Proc. IEEE/CVF CVPR*. 11621–11631.
[13] C. Campos, R. Elvira, J. J. G. Rodriguez, J. M. M. Montiel, and J. D. Tardos. 2021. ORB-SLAM3: An accurate open-source library for visual, visual-inertial, and multimap SLAM. *IEEE T-RO* (2021), 1–17.
[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. IEEE IISWC*. 44–54.
[15] Z. Chen and X. Huang. 2017. End-to-end learning for lane keeping of self-driving cars. In *Proc. IEEE IV*. 1856–1860.
[16] J. Fickenscher, F. Hannig, and J. Teich. 2019. DSL-based acceleration of automotive environment perception and mapping algorithms for embedded CPUs, GPUs, and FPGAs. In *Proc. of ARCS*. 71–86.
[17] J. Fickenscher, J. Schlumberger, F. Hannig, J. Teich, and M. E. Bouzouraa. 2018. Cell-based update algorithm for occupancy grid maps and hybrid map for ADAS on embedded GPUs. In *Proc. of DATE*. 443–448.
[18] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun. 2013. Vision meets robotics: The KITTI dataset. *The International Journal of Robotics Research* 32, 11 (2013), 1231–1237.
[19] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. IEEE IISWC*. 3–14.
[20] W. Han and Y. Zhang. 2019. Fast LOAM (Lidar Odometry And Mapping). https://github.com/wh200720041/floam.
[21] P. Hank, S. Müller, O. Vermesan, and J. Van Den Keybus. 2013. Automotive ethernet: In-vehicle networking and smart mobility. In *Proc. DATE*. EDA Consortium, 1735–1739.
[22] R. Hartley and A. Zisserman. 2004. *Multiple View Geometry in Computer Vision* (second ed.). Cambridge University Press, ISBN: 0521540518.
[23] D. Held, S. Thrun, and S. Savarese. 2016. Learning to Track at 100 FPS with deep regression networks. In *Proc. ECCV*. 749–765.
[24] J. L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* 34, 4 (2006), 1–17.
[25] W. Jang, H. Jeong, K. Kang, N. Dutt, and J. C. Kim. 2020. R-TOD: Real-time object detector with minimized end-to-end delay for autonomous driving. In *Proc. IEEE RTSS*. 191–204.
[26] Jonaspfab and Danielebp. 2019. CUDA Implementation of a Hough Transform based Lane Detection algorithm. https://github.com/jonaspfab/cuda-lane-detection.
[27] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada. 2015. An open approach to autonomous vehicles. *IEEE Micro* 35, 6 (2015), 60–68.
[28] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi. 2018. Autoware on Board: Enabling autonomous vehicles with embedded systems. In *Proc. ACM/IEEE ICCPS*. 287–296.
[29] J. Kim, A. Rohrbach, T. Darrell, J. Canny, and Z. Akata. 2018. Textual explanations for self-driving vehicles. In *Proc. ECCV*. 563–578.
[30] K. Kurzer. 2016. *Path Planning in Unstructured Environments: A Real-time Hybrid A* Implementation for Fast and Deterministic Path Generation for the KTH Research Concept Vehicle*. Master's thesis. KTH, Integrated Transport Research Lab, ITRL.
[31] S. Kuutti, S. Fallah, K. Katsaros, M. Dianati, F. Mccullough, and A. Mouzakitis. 2018. A Survey of the State-of-the-Art localization techniques and their potentials for autonomous vehicle applications. *IEEE TOTJ* 5, 2 (2018), 829–846.
[32] S. C. Lin, Y. Zhang, C. H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars. 2018. The architectural implications of autonomous driving: Constraints and acceleration. *ACM SIGPLAN Notices* 53, 2 (2018), 751–766.
[33] Y. Luo. 2020. LaneNet-Lane-detection. https://github.com/MaybeShewill-CV/lanenet-lane-detection.

[34] B. Maity, B. Donyanavard, and N. Dutt. 2020. Self-aware memory management for emerging energy-efficient architectures. In *Proc. IGSC*. IEEE, 1–8.
[35] B. Maity, M. Shoushtari, A. M. Rahmani, and N. Dutt. 2020. Self-adaptive memory approximation: A formal control theory approach. *IEEE ESL* 12, 2 (2020), 33–36.
[36] P. Moulon, P. Monasse, R. Perrot, and R. Marlet. 2016. Openmvg: Open multiple view geometry. In *Proc. Springer RRPR*. 60–74.
[37] Tiago Mück, Bryan Donyanavard, Biswadip Maity, Kasra Moazzemi, and Nikil Dutt. 2021. MARS: Middleware for Adaptive Reflective Computer Systems. arXiv:2107.11417 [cs.DC]
[38] NVIDIA. 2021. Hello AI World Nvidia Jetson. https://github.com/dusty-nv/jetson-inference.
[39] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, and S. Wang. 2017. An Evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *Proc. IEEE RTAS*. 353–364.
[40] P. Palanisamy. 2019. Multiple-Object-Tracking-from-Point-Clouds. https://doi.org/10.5281/zenodo.3559186.
[41] S. D. Pendleton, H. Andersen, X. Du, X. Shen, M. Meghjani, Y. H. Eng, D. Rus, and M. H. Ang. 2017. Perception, planning, control, and coordination for autonomous vehicles. *Machines* 5, 1 (2017).
[42] M. I. Ribeiro. 2004. Kalman and extended kalman filters: Concept, derivation and properties. *Institute for Systems and Robotics* 43 (2004), 46.
[43] A. Saifullah, S. Fahmida, V. P. Modekurthy, N. Fisher, and Z. Guo. 2020. CPU energy-aware parallel real-time scheduling. In *Proc. ECRTS*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2:1–2:26.
[44] W. Schwarting, J. Alonso-Mora, and D. Rus. 2018. Planning and decision-making for autonomous vehicles. *Annual Review of Control, Robotics, and Autonomous Systems* 1, 1 (2018), 187–210.
[45] J. Shannon. 2019. Extended Kalman Filter. https://github.com/jeremy-shannon/CarND-Extended-Kalman-Filter-Project.
[46] X. Song, P. Wang, D. Zhou, R. Zhu, C. Guan, Y. Dai, H. Su, H. Li, and R. Yang. 2019. Apollocar3d: A large 3d car instance understanding benchmark for autonomous driving. In *Proc. IEEE/CVF CVPR*. 5452–5462.
[47] A. Taha and N. AbuAli. 2018. Route planning considerations for autonomous vehicles. *IEEE ComMag* 56, 10 (2018), 78–84.
[48] E. Talpes, D. D. Sarma, G. Venkataramanan, P. Bannon, B. McGee, B. Floering, A. Jalote, C. Hsiong, S. Arora, A. Gorti, and G. S. Sachdev. 2020. Compute solution for tesla's full self-driving computer. *IEEE Micro* 40, 2 (2020), 25–35.
[49] The Verge. 2019. Tesla FSD chip. https://www.theverge.com/2019/4/22/18511594/tesla-new-self-driving-chip-is-here-and-this-is-your-best-look-yet.
[50] Y. Wang, W. Chao, D. Garg, B. Hariharan, M. Campbell, and K. Q. Weinberger. 2019. Pseudo-lidar from visual depth estimation: Bridging the gap in 3d object detection for autonomous driving. In *Proc. IEEE/CVF CVPR*. 8445–8453.
[51] Y. Wang, S. Liu, X. Wu, and W. Shi. 2018. CAVBench: A benchmark suite for connected and autonomous vehicles. In *Proc. IEEE/ACM SEC*. 30–42.
[52] Z. Wang, W. Ren, and Q. Qiu. 2018. LaneNet: Real-Time Lane Detection Networks for Autonomous Driving. arXiv:1807.01726 [cs.CV]
[53] H. Xu, Y. Gao, F. Yu, and T. Darrell. 2017. End-to-end learning of driving models from large-scale video datasets. In *Proc. IEEE/CVF CVPR*. 2174–2182.
[54] J. Xue, J. Fang, T. Li, B. Zhang, P. Zhang, Z. Ye, and J. Dou. 2019. Blvd: Building a large-scale 5d semantics benchmark for autonomous driving. In *Proc. IEEE ICRA*. 6685–6691.
[55] S. Yi, T. Kim, J. Kim, and N. Dutt. 2021. Energy-Efficient adaptive system reconfiguration for dynamic deadlines in autonomous driving. In *IEEE ISORC*. 96–104.

Received April 2021; revised June 2021; accepted July 2021