**Noname manuscript No.**
(will be inserted by the editor)

# ADBench: Benchmarking Autonomous Driving Systems

**Hamid Tabani · Roger Pujol · Miguel Alcon · Joan Moya · Jaume Abella · Francisco J. Cazorla**

Received: date / Accepted: date

**Abstract** Driven by the improvements in a variety of domains, autonomous driving is becoming a reality and today, industry aims at moving toward fully autonomous vehicles. High-tech chip manufacturers are designing high-performance and energy-efficient platforms in accordance with safety standard requirements. However, the software used to implement advanced functionalities in autonomous vehicles challenges real-time constraints on those platforms. Hence, there is a clear need for industry-level autonomous driving benchmarks to evaluate platforms and systems. In this paper, we propose ADBench, a benchmarking approach and benchmark suite for state-of-the-art autonomous driving platforms, in accordance with the key modules, structural design and functions of AD systems, building on several industry-level autonomous driving systems. The use of standard benchmarks facilitates the design, verification and validation process of autonomous systems.

**Keywords** Benchmarking · Autonomous Driving Systems · Safety-Critical Systems

## 1 Introduction

Self-driving vehicles are poised to dramatically change and revolutionize the automotive industry. Depending on the level of autonomy, autonomous driving is categorized into six levels, where "Level 0" which means no autonomy and "Level 5" is full autonomy, according to SAE International [1]. For more than a decade, we have seen cars featuring level one (driver assistance) and recently, cars with level two (partial automation). Even level three (conditional automation) are already in mass-production. However, moving toward higher levels of autonomy (e.g., levels 4 and 5 [1]) is challenging and requires

All Authors
Barcelona Supercomputing Center (BSC), Jordi Girona 29, 08034 Barcelona, Spain.
E-mail: firstname.lastname@bsc.es

to address several key aspects such as the demand for massive computation capacity, security and certification.

Critical real-time domains, as in the case of AD systems, are not only about the pursuit of absolute high-performance, but also about achieving sufficiently high performance levels subject to meeting specific requirements related to the certainty and predictability of the tasks' execution time (sometimes even at the expense of some performance loss). This occurs because in autonomous vehicles, once the vehicle is unable to respond within the specified time, it may cause a catastrophic event with fatalities. Therefore, AD systems, including their software stack, need to be designed in such a way that these requirements are fully satisfied [2,3,4,5].

Having a representative and comprehensive set of benchmarks can significantly improve the development process by easing the assessment of the system, as well as allow comparing fairly different design choices. This paper presents ADBench, the first set of automotive benchmarks representative of industry-level and modern AD systems whose main features are as follows.

**Representative and reproducible benchmarks**. We consider and include most of the common modules and functions including key functional safety and comfort features to provide a comprehensive set of benchmarks. These include the main modules of AD systems as well as driver-system interaction functions such as speech recognition command and controls, driver-assistance systems, and driver monitoring systems.

**Realistic and industry-level modules**. Many of the algorithms and modules used in AD systems already exist in a variety of domains. However, the complexity and dimensions of many of them widely changes across domains. ADBench provides realistic benchmarks in terms of functionality and complexity with respect to industry-level AD systems. Also ADBench targets realistic hardware platforms for its execution and provides benchmarks compatible with the latest heterogeneous hardware platforms for AD systems such as NVIDIA Drive Xavier [6] and NVIDIA Drive Pegasus [7].

**Flexibility to configure the benchmarks according to the hardware and software capabilities**. Most of the ADBench benchmarks are designed in a flexible manner to be configured easily according to the needs of the user. In most cases, the problem size can be decreased in two main dimensions. To change the problem size, we can change the input size – for example, the resolution of the input image in the Camera Object Detection – which will affect directly the computation time. On the other hand, for deep learning-based benchmarks, the deep learning model architecture, such as the type and number of layers, can be modified. This helps to diminish the problem size for hardware prototyping (e.g., RTL models). Also, it allows to configure the benchmarks for more powerful hardware to support a higher number of sensors and higher resolutions.

The rest of the paper is organized as follows: In Section 2, we present background on the state-of-the-art AD systems and their main modules. Section 3 presents our study and comprehensive analysis of various AD systems. Section 4 introduces the benchmarks in detail, and Section 5 provide some early analysis of some of the benchmarks. Finally, Section 6 reviews the related work and Section 7 concludes the paper.

## 2 Background on Autonomous Driving

In the past few years, various AD systems have started their development and most of them are still on-going projects aiming at reaching full autonomy (i.e. level 5). Although several of them are not accessible to the community, their main architecture and features are publicly available (e.g. Tesla autopilot [8]).

Among those AD systems, which are under development in top-tier tech companies and car manufacturers, we have access to NVIDIA drive program [9], Baidu's Apollo program [10] and Autoware [11], which are open-source and available for in-detail study. We have observed that key modules and the overall structure of the system, the use of heterogeneous sensors to perceive the objects, and many more details are quite similar across these systems. For the sake of simplicity and without lack of generality, in the rest of the paper, we focus on Baidu's Apollo AD framework and we describe its modules and functions in more detail.

### 2.1 Apollo AD Framework

Apollo [10] is an open-source autonomous driving platform developed and released by Baidu. It offers its users the opportunity to develop their own AD systems through on-vehicle and hardware platforms. Regarding its software implementation, Apollo, similarly to most state-of-the-art AD systems, consists of a set of main modules [12,13] (see Figure 1). Each of the modules implements a crucial functionality of autonomous vehicles. The main modules of Apollo are:

– **Perception:** identifies the area surrounding the autonomous vehicle by detecting objects, obstacles, and traffic signs,. It is considered the most critical and complex module of an AD system. Perception module fuses the output of several types of sensors such as LiDAR, radar, and camera to improve its accuracy.
– **Localization:** estimates where the autonomous vehicle is located, using various information sources such as GPS, LiDAR and IMU. State-of-the-art localization algorithms, including the one in Apollo, are capable of localizing the position of the vehicle at centimeter-level accuracy.
– **Prediction:** anticipates the future motion trajectories of the perceived obstacles.
– **Navigator:** tells the autonomous vehicle how to reach its destination via a series of lanes or roads.
– **Planning:** plans the spatiotemporal trajectory for the autonomous vehicle to take.

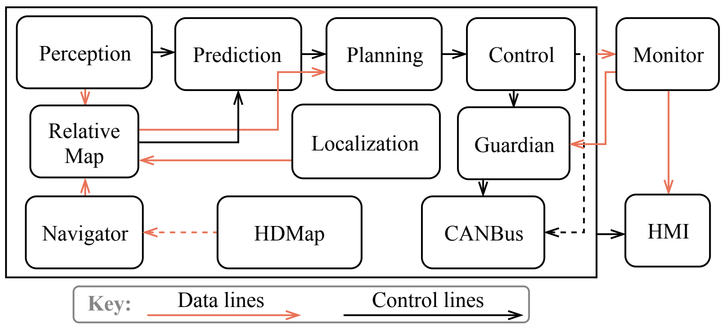

Fig. 1: Interaction between Apollo's modules [10].

- **Control:** executes the planned spatiotemporal trajectory by generating control commands such as accelerate, brake, and steering.
- **CanBus:** is the interface that passes control commands to the vehicle hardware. It also passes chassis information to the software system.
- **HD-Map:** is similar to a library. Instead of publishing and subscribing messages, it works as a query engine support, which provides ad-hoc structured information regarding the roads.
- **HMI** (Human Machine Interface): is a module for viewing the status of the vehicle, testing other modules and controlling the functioning of the vehicle in real-time.
- **Monitor:** is the surveillance system of all the modules in the vehicle, including hardware.
- **Guardian:** is a safety module that performs the function of an Action Center and intervenes should Monitor detect a failure.

For the sake of facilitating the installation and managing dependencies across numerous libraries, Apollo, similarly to other AD systems, is provided inside several Docker container images [14]. A container is a standard software unit that packages up code and all its dependencies so the entire application can run in a quick and reliable way in many different computing environments. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Each module is initialized by the Robot Operating System (ROS) [15], loading its parameters. These parameters are given to the module through configuration files or as flags in the command line. After that, the module calls the `Spin` function, which initializes one or more ROS *Spinners* before finishing the module initialization step. Then, the *Spinners* call their `Spin` functions, which execute all the callback functions that are triggered during runtime, until the client shuts down the module. A callback function is connected to a specific event, and it is triggered when this event occurs. In terms of ROS, a function can subscribe to an event (topic) and publish an event as well. Therefore, during normal operation, modules spin constantly awaiting for the

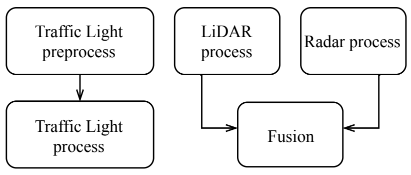

Fig. 2: DAG of the LiDAR configuration.

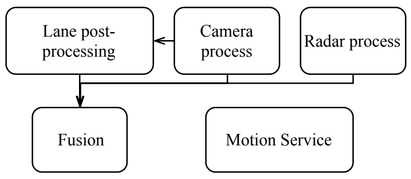

Fig. 3: DAG of the camera configuration.

events that trigger them (e.g. a new camera frame becomes available), and execute upon the occurrence of those events, further producing new events (e.g. an object list that will trigger other modules). Next, we illustrate the operation of a module with the perception module [16], which is, indeed, the most complex module.

**Perception Module**. It is in charge of the detection of the obstacles that surround the car. Its main functionality is to transform data from sensors (e.g., images from cameras, point clouds from LiDAR) into obstacles, thus, acquiring relevant information about them such as their position, size, orientation, etc.

Some of the large modules such as *Perception* are composed of several submodules. Submodules are very similar to modules. Their members share the same structure although each of them has a specific and totally different functionality. They also register callbacks to be triggered by messages or timers, and publish messages, thus communicating with other members or other components of Apollo.

The global configuration of input sensors for the Perception module can be represented as a direct acyclic graph (DAG). With this, Apollo offers the possibility of building customized configurations, according to the requirements and the available hardware. These DAGs, along with other parameters of the input sensors, are defined in a configuration file. Two different configurations are shown in Figures 2 and 3. In these DAGs, nodes correspond to different processes and each of them is responsible for completing a specific task. Arrows indicate data dependencies between nodes of each DAG. For instance, in Figure 3, *Fusion* requires the output data of *Lane post-processing*, *Camera process*, and *Radar process.* Although submodules mainly perform independent tasks, however, they communicate and exchange data. In fact, in most cases the system behaves in a pipeline fashion where the output of one module (submodule) is an input to the successor module (submodule).

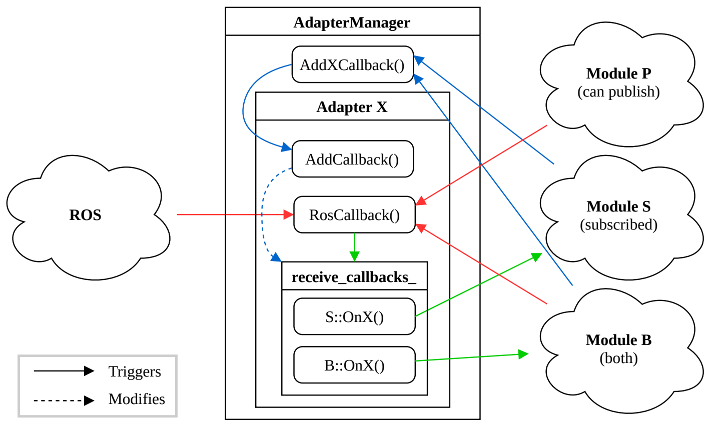

Fig. 4: Communication between ROS and modules using the Adapter Manager to subscribe and publish messages.

All modules use adapters to communicate with I/O and with other modules, and each adapter is attached to a certain topic, as Figure 4 shows. More precisely, it shows how ROS and modules communicate using the AdapterManager to subscribe and publish messages. As the blue arrows show, during the initialization phase of a module, such module adds its callbacks to the adapters of the topics that it is subscribed, through the AdapterManager. From the adapter's point of view, this phase finalizes when all callbacks related to the topic are stored in its receive_callbacks_vector. In this vector, the adapter saves the callback functions to trigger when an incoming message arrives. Then, when the module is running, if a message of that topic enters the system, via ROS (e.g. I/O) or other modules (red arrows), the adapter (with the RosCallback function) will trigger all functions inside the vector, as green arrows show. Note that the spinner created by the module is the one detecting the message, and the one running the function.

AD systems are usually structured similarly to the aforementioned approach. We have designed the ADBench benchmarks to process events from ROS bag files[1], thus decoupling them from other modules and the whole AdapterManager scheme. This infrastructure will help developers to focus on the modules themselves rather than on the complex communication system.

## 3 Analysis and Design Space Exploration

This section presents a *structural* analysis of AD frameworks, in contrast with previous section, which provided a *functional* description. We build our anal-

[1] A bag is a file format in ROS for storing ROS message data. They are typically created by a tool like *rosbag*, which subscribes to one or more ROS topics, and stores the serialized message data in a file as it is received. These bag files can also be played back in ROS to the same topics they were recorded from, or even remapped to new topics.

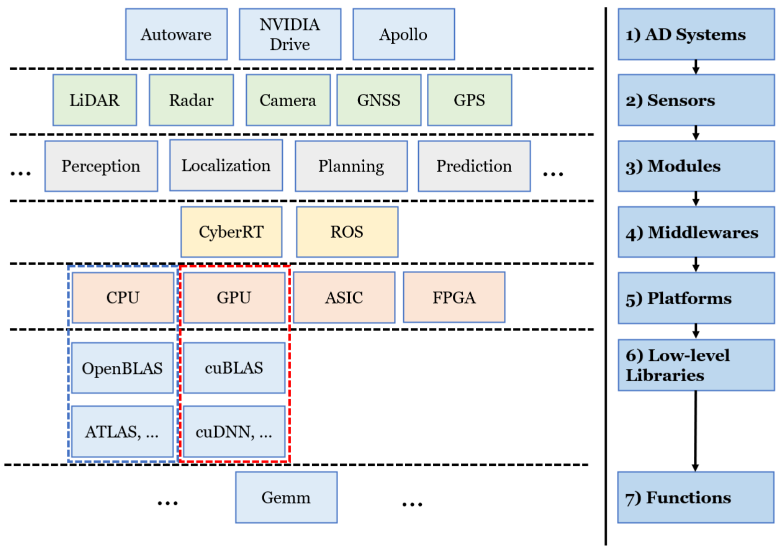

Fig. 5: Hierarchical structure of AD systems [17].

ysis studying several state-of-the-art AD systems for the sake of representativeness. We present a top-down hierarchical structure of the AD systems and then, we discuss each of the levels of this hierarchy in more detail by highlighting the key points of relevance for ADBench.

## 3.1 General Architecture of AD Systems

To build ADBench, we have done a thorough analysis on three different AD system which provide end-to-end software stack for autonomous vehicles, Autoware [11], Apollo [10], and NVIDIA Drive [9]. Our analysis confirms that the key functions and architecture of AD systems are very similar although implementation approaches and detailed algorithms may vary across different AD systems. Based on these analyses, we draw a hierarchical overview of AD systems, as shown in Figure 5, considering the following features:

- Input sensors which are needed to sense the surrounding of the vehicle.
- Driver libraries to use third-party sensors and units.
- Modular structure of the system.
- Use of a middleware in some AD systems with different purposes.
- The target hardware platform on which the overall system or specific modules run.
- Implementation of low-level and platform-dependent libraries.
- Use of graphical user-interface for user-system interaction.

As Figure 5 shows, we decompose the architecture of AD systems into several levels. Often these systems fuse the outcome of various types of sensors to increase the accuracy of the system. For instance, in the three AD systems that we study, the perception module uses laser sensors (LiDAR), ultra-sonic sensors (radar) and various types of cameras as shown in level 2 of Figure 5. An AD system is composed of several modules that are responsible to implement specific functions of the system. Level 3 shows some of these modules, which are common in AD systems. Modules may use specific drivers to communicate with various sensors in the system.

As discussed in Section 4, the interaction across modules is high, and it is managed through a middleware such as ROS [15] or CyberRT [10] (see level 4). Level 5 shows the target hardware platforms on which the modules run. In general, we have multicore processors that host the main functions and the middleware. AD systems also use high-end and powerful GPUs as well as specific accelerators to run some compute- and energy-intensive modules. Lastly, depending on the target hardware platform, the system may use platform-specific, low-level, and optimized libraries, which include very specific functions organized hierarchically, as shown from level 6 to 8 respectively.

ADBench has been made comprehensive in nature, and not only includes key modules and their variants, but also considers various implementations targeting different platforms and low-level libraries.

## 3.2 AD Systems Hierarchical Structure Breakdown

### *3.2.1 Sensors*

**Vision Sensors**. Building reliable vision capabilities is key for AD systems. By combining a variety of sensors, however, developers have been able to create a detection system that can see a vehicle's environment even better than human eyesight. The keys to this system are diversity, by using different types of sensors, redundancy and overlapping sensors, to ensure that object detection is accurate enough.

The three primary autonomous vision sensors are camera, radar and LiDAR. Together, they provide the car visuals of its surroundings and help detecting the speed and distance of nearby objects, as well as their three-dimensional shape. State-of-the-art AD systems use all the three types of vision sensors for the sake of reliability and accuracy.

**Localization Sensors**. Current localization approaches for AD systems involve satellite navigation systems, vehicle motion sensors, range sensors, and also include the vision sensors. For instance, the Inertial Measurement Unit (IMU) is a sensor capable of defining the movement of the vehicle along different axes. It calculates acceleration along the X, Y, Z axes, orientation, inclination, and altitude. A Global Navigation Satellite System (GNSS) is often used for positioning, whose information can be leveraged by several modules. For instance, Global Positioning System (GPS) or NAVSTAR are the US system for positioning.

Both *Perception* and *Localization* modules in AD systems use the latest sensors to provide high accuracy. This is taken into account in the benchmarks provided by ADBench.

### *3.2.2 Modules*

Modular design has plenty of well-known advantages for the development and maintenance of complex systems, as it is the case for AD systems. Modular design allows reducing the complexity of the development process with a divide-and-conquer strategy, eases the update and replacement of parts of the system, and simplifies debugging, organization and documentation, among other advantages.

For these reasons, AD systems, which are complex systems, are designed in a modular way. The benchmarks that are provided by ADBench reflect this modularity focusing on particular functionalities of AD systems.

### *3.2.3 Middleware*

Middleware is a core component of many customized and automated platforms. The middleware in AD systems is used to rapidly build applications, from advanced driver-assistance systems (ADAS) functions and complex Human-Machine Interfaces (HMIs) to full autonomy suites. The use of modular design can help the applications to be independent of the middleware, sensors and computing hardware. Apollo and Autoware use ROS [18] as middleware. ROS is responsible for the implementation of message passing and communications system across modules and sensors. The representative benchmarks in ADBench use ROS as middleware to communicates with the sensors' data and other modules of the system.

### *3.2.4 Hardware Platforms*

Hardware platforms are as important as software ones for AD systems. However, while software platforms in cars may experience upgrades during their lifetime, this is not generally the case for hardware platforms. For instance, software updates may occur to improve object detection with cameras or radars to increase accuracy and fix corner cases. However, such software, which may become more performance-hungry, needs to run efficiently and *timely* on the hardware platforms onboard as they are. Therefore, cars must be equipped with powerful and flexible hardware platforms.

In fact, the radical growth in performance required by AD systems has already propelled the developers and industry to use accelerators and specialized hardware platforms. Powerful and high-end GPUs are now part of most AD systems. Also, some vendors, such as Tesla, are designing their own specialized hardware to maximize the performance/energy ratio.

The use of GPUs or other hardware requires to adapt the software by modifying its implementation, or reimplementing some functions. To this end, some modules, such as *Perception*, which can highly benefit from parallelization, are implemented to run on GPUs or accelerators. Thus, several AD systems provide CUDA implementations for NVIDIA GPUs. Some of the ADBench benchmarks with high performance demands include CUDA and OpenMP implementations for NVIDIA GPUs and multicore CPUs respectively.

### *3.2.5 Low-level Platform-dependent Libraries*

Depending on the target hardware platform, a platform-dependent optimized library is used, which includes the majority of the required low-level functions implemented and optimized for that specific platform. For instance, *ATLAS* [19] and *openBLAS* [20] are two well-known libraries for CPUs, as well as *cuBLAS* [21] and *cuDNN* [22] for NVIDIA GPUs.

At this low level, different operations are performed by calling the corresponding functions of a library providing their required parameters. Such compute-intensive operations are linear algebra operations (mostly matrix operations) whose platform-dependent implementations are provided by the corresponding libraries. Normally, these implementations have been optimized with average performance in mind.

The result of our thorough analysis on different AD systems shows that the deep learning-based modules and submodules are the ones with heaviest computation loads. We have observed that their computations are translated into operations such as vector addition, scalar multiplication, dot product, linear combinations, and matrix operations with different dimensions. Deep learning frameworks do not implement those operations directly but, instead, they are built upon low-level libraries providing efficient implementations (mostly targeting optimized average performance) for different CPU and GPU target platforms. In particular, the frameworks considered in our analysis build upon one or several of the following low-level libraries (e.g., TensorFlow uses cuBLAS, TensorRT and cuDNN libraries):

– **BLAS** [23] (Basic Linear Algebra Subprograms) is a specification that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication.
– **OpenBLAS** [20] is an open source implementation of the BLAS API with many hand-crafted optimizations for specific processor types.
– **ATLAS** [19] (Automatically Tuned Linear Algebra Software) is a library for algebra targeting high-performance computing platforms.
– **Intel MKL** [24] (Intel Math Kernel) is a library of optimized math routines for science, engineering, and financial applications. Core math functions include BLAS, LAPACK, ScaLAPACK, sparse solvers, fast Fourier transforms, and vector math.
– NVIDIA's **cuBLAS** [21] library is a fast GPU-accelerated implementation of the standard basic linear algebra subroutines (BLAS).
– **cuDNN** [22] (NVIDIA CUDA Deep Neural Network) library is a GPU-accelerated library of primitives for deep neural networks. cuDNN provides

highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers, which ultimately build upon matrix operations.

– **TensorRT** [25] is a library for high-performance deep learning inference which includes a deep learning inference optimizer and runtime that delivers low-latency and high-throughput for deep learning inference applications, also building mostly on matrix operations.

ADBench employs some of these libraries for the implementation of the benchmarks depending on the target architecture. For instance, the *Camera-based Object Detection* benchmark employs cuDNN and OpenBLAS.

## 4 ADBench Benchmark Suite

### 4.1 Benchmarks

This section presents the 8 benchmarks in the proposed ADBench suite. We have designed all the benchmarks based on modules of the Apollo AD framework, presented in section 2.1. However, all of them are able to fully function without being in a full AD system. In other words, the benchmarks are designed by feeding all the data required from input files rather than co-runner modules and real sensors. This data is included in ADBench to run the tests, and was recorded from actual sensors and processes running in Apollo. Since we are testing performance and not correctness, some benchmarks can auto-generate random input data whenever data has no impact in performance (e.g. the sequence of operations in a matrix multiplication depends on the dimensions of the matrices but is independent of the actual values multiplied).

As discussed earlier, AD systems rely on a variety of sensors, such as cameras and LiDARs, which require deep learning algorithms to process their data. For this reason, some of the benchmarks utilize this kind of algorithms. The deep neural network models employed in such modules are similar to those in AD systems such as Apollo.

#### *4.1.1 Camera-based Object Detection*

This benchmark is in charge of obstacle and lane detection from the camera images. The object and lane detection process begins by cropping the image to the region of interest (ROI), if necessary, and resizing it to the required resolution to be the main Convolutional Neural Network (CNN) input. Then, the CNN's forward propagation process occurs, computing information about both object and lane detections. With the results of the forward propagation, it generates the 2-D bounding box (BB), the class probabilities of each of the objects, and the lane map (a matrix containing the probability that each pixel is part of a lane at its position in the image). If the user indicates it (by activating a flag), another similar process starts, however, with a different CNN which provides more details on the lanes and different lane maps, one per each possible lane. After that, the following processes are carried out in this order:

– 2-D BBs are used to generate 3-D BBs.
– The coordinates of the objects are transformed from camera to car space coordinates.
– Each detected object is associated with another detected object in the previous frame, if they match (tracking process).
– A Kalaman filter is applied to all matched objects.

As a result of the obstacle detection, it outputs a 3D rectangular cuboid of each obstacle, together with its relative velocity and direction, and its classification type (vehicle, truck, cyclist or pedestrian). For the lane detection, it outputs the lane map.

To sum up, the Camera-based Object Detector uses two CNNs, one for obstacle and lane detection (main CNN), and the other for detailed lane detection (secondary CNN). Both CNNs are based on YOLO [26]. The lane lines are detected by segmentation using the main CNN with some modifications. The secondary CNN can be activated, and it is used to provide longer lane lines in cases of either whole or broken lines. The difference between both lines is its final usage. While a lane mark segment is used for visual localization, a whole lane line is used for lane keeping.

#### *4.1.2 LiDAR-based Object Detection*

This benchmark is responsible for obstacle detection from 3D point cloud data from the LiDAR sensor. This process begins by filtering the LiDAR points that are outside the ROI, removing background objects. The ROI specifies the drivable area that includes road surfaces and junctions, which are retrieved from the High Definition (HD) Map.

These filtered point clouds are the input of the CNN segmentation process, which detects and segments out foreground obstacles, e.g. cars, trucks, bicycles, and pedestrians. Then, the obstacle segments (or clusters) are used to define a bounding box for each obstacle. Finally, once the detected obstacles are defined, they are compared with previously-detected obstacles for tracking purposes [27]. The output of this process is similar to the camera-based object detection without the lane information.

#### *4.1.3 Traffic Light Detection*

This benchmark is designed to provide accurate and comprehensive traffic light status using cameras. Typically, the traffic light has three states (colors): red, yellow and green. However, if the traffic light is not working, it might display the black color or show a flashing red or yellow light. Sometimes the traffic light cannot be found in the camera's field of vision and the module fails to recognize its status, so it is unknown. This benchmark covers all the five scenarios.

The detection starts with the camera selection. Using a single camera that has a constant field of vision is not enough to cover all the possible scenarios. This limitation is due to the fact that the height of the traffic lights or the width of crossing varies widely, and the perception range must be above 100 meters in order to take proper actions. Therefore, the selection is done building on the data about localization, calibration of camera results, and the HD-Map. The image of this camera is processed following these steps [28]:

1. **Rectify**. The projected position of the car, which is affected by the calibration, localization, and the HD-Map label, is not completely reliable. Therefore, a larger ROI, calculated using the projected light's position, is used to find the accurate BB for the traffic light. The goal of this step is to detect a traffic light BB in a ROI of the input image. The traffic light detection is implemented as a regular CNN detection task.
2. **Recognize**. In this step, the BB's color of the detected traffic lights are classified. This recognition is implemented as a CNN classification task. The class with maximum probability will be regarded as the light's status, if and only if the probability is large enough. Otherwise, the light's status will be set to black, which means that the status is not certain.
3. **Revise**. Since a traffic light can be flashing or shaded, and the previous step does not give a perfect result, the current status may fail to represent the real status. This step aims to correct the recognized color using sequential information across multiple frames.

### *4.1.4 Localization*

Two of the existing approaches for localization of autonomous vehicles are regarded as the most accurate, and thus are the most used ones. The first approach is Real Time Kinematic (RTK) based method, which incorporates GNSS and IMU information. The other approach, which provides centimeter-level accuracy and is implemented in this benchmark, builds on the Multi-Sensor Fusion (MSF) method, which incorporates GNSS, IMU, and LiDAR information [29,30].

The localization benchmark adaptively uses information from complementary sensors (GNSS, LiDAR, and IMU) to achieve high localization accuracy and resilience in challenging scenes, such as urban downtown, highways, and tunnels. The MSF method makes an innovative use of LiDAR intensity and altitude cues to significantly improve localization system accuracy and robustness. A GNSS RTK module utilizes the help of the MSF framework and achieves a better ambiguity resolution success rate. Reported results show that the approach can provide 5-10cm root mean square (RMS) accuracy and outperforms previous state-of-the-art systems. Therefore, this benchmark has been included in ADBench.

### *4.1.5 Planning*

The planning benchmark mimics the behavior of the planning module, which is in charge of generating a trajectory for the autonomous car that is safe, collision-free and comfortable to execute. To do so, it needs different kinds of information: data from the outside (e.g., perceived obstacles and traffic lights), the actual route of the car, and its localization and status such as position, velocity, acceleration, to name a few. Since during the ride the vehicle has to deal with different road conditions and driving scenarios, this method uses scenario-specific and holistic approaches for planning its trajectory. Apollo, for instance, focuses on three driving scenarios: following the lane, side passing a static obstacle, and retaking the ride after a stop signal. During the ride, this method detects the current scenario and selects the correct approach to deal with it from the vehicle status and other relevant information. First, a raw trajectory is generated and applies vehicle kinemetic model in the algorithm to create the raw trajectory with a series of distance equidistant points. The received raw trajectory is taken as an initial guess for optimization steps to iterate on. The generated result is a set of points that are not distributed evenly but are closer to each other near the turning while those on a linear path are more spread-out. This not only ensures better turns, but as time/space is fixed, the nearer the points, the slower the speed of the vehicle [31,32]. The planning benchmark is implemented using thread-level parallelism due to the nature of its design to improve the performance and efficiency.

### *4.1.6 Prediction*

Prediction is also one of the key functions of AD systems, which is in charge of anticipating the future motion trajectories of the perceived obstacles. Given the obstacles, their status (positions, headings, velocities, accelerations, etc.) and the vehicle localization, it generates the predicted trajectories with probabilities for each of the obstacles. It first predicts the path and the speed separately for any given obstacle, which is done using a Multi-Layer Perceptron (MLP) model. Then, using these predictions, the method generates the predicted trajectories of each of the obstacles. To this end, it offers six possible predictors to use depending on the type (e.g., bicycle, pedestrian, vehicle, etc.) and status (e.g., on the same lane) of the obstacle under analysis, which are as follows [33]:

- **Empty**: obstacles have no predicted trajectories.
- **Single lane**: obstacles move along a single lane in highway navigation mode. Obstacles that are not on lane will be ignored.
- **Lane sequence**: obstacle moves along the lanes.
- **Move sequence**: obstacle moves along the lanes by following its kinetic pattern.
- **Free movement**: obstacle moves freely.
- **Regional movement**: obstacle moves in a possible region.

### *4.1.7 Navigator*

Navigation module in AD systems generates high level navigation information based on requests from other modules. It uses map data, the routing start point, and the routing end point, to compute the passage lanes and roads. Usually, the routing start point is the autonomous vehicle current location. The *Navigator* benchmark mimics this behavior. Basically, it searches for an optimized path (i.e. shortest path or fastest depending on the configuration) between the start and end points and if successful, it provide the details of the path to the system. This process is very similar to graph search algorithms which are aiming at finding an optimal path.

### *4.1.8 Automatic Speech Recognition*

Automatic Speech Recognition (ASR) is the process of deriving the transcription of an utterance, given the speech waveform in form of word sequences. Understanding the speech, however, goes one step further, and gleans the meaning of the utterance in order to carry out the speaker's command. Speech recognition systems are one of the lately-introduced ways of human-machine interaction. ASR systems inside cars aim at increasing safety by allowing the driver to control a number of features of the car without using his/her hands or distracting his/her eyes from the road. ASR systems are considered as complex algorithms due to the challenging nature of the problem. Fortunately, deep learning significantly boosts the accuracy in ASR systems to human-level precision.

This benchmark provides an end-to-end ASR pipeline with medium-size vocabulary for AD systems based on deep learning. It is designed based on state-of-the-art ASR systems to be employed in AD systems with the possibility of modifying the algorithm and employing other models.

## 4.2 Measuring Metrics

One of the main use cases of the ADBench suite consists of assessing the performance of hardware platforms running AD algorithms. For this purpose, we measure the performance through the benchmarks' execution times mimicking the same function as in real AD systems. The hardware must be able to run each of the benchmarks within specified deadlines to be appropriate for real systems. Failing to perform the specified tasks and computations in time can cause catastrophic risks. The deadline for each benchmark varies depending on a number of parameters, which generally relate to system requirements. For instance, the object detection module needs to process the input frames produced by the cameras. A camera working at 30 frames per second (FPS) requires the object detection module to process each frame within 33 ms, whereas if the camera works at 60 FPS, then the limit would be 16 ms.

### 4.3 Implementation Details

Table 1: Implementation details of ADBench benchmarks for different hardware platforms.

| **Benchmarks** | **x86** | **ARM** | **CUDA** | **OpenMP** | **CUDA + Tensor Core** |
|---|---|---|---|---|---|
| Camera-based OD | ✓ | ✓ | ✓ | ✓ | ✓ |
| LiDAR-based OD | ✓ | ✓ | ✓ | ✓ | ✓ |
| TL Detection | ✓ | ✓ | ✓ | ✓ | ✓ |
| Localization | ✓ | ✓ | | | |
| Planning | ✓ | ✓ | | | |
| Prediction | ✓ | ✓ | | | |
| Navigator | ✓ | ✓ | | | |
| ASR | ✓ | ✓ | ✓ | ✓ | ✓ |

As discussed in Section 3.2.4, we designed the ADBench benchmarks to run on various hardware platforms, which in most cases require modified implementation rather than just using different compilers. Table 1 summarizes the implementation details of each of the benchmarks for x86 and ARM architectures, NVIDIA GPUs using CUDA and tensor cores for deep learning in the latest NVIDIA architectures, and OpenMP for multicore CPUs or GPUs.

### 4.4 Summary of Features

Overall, the ADBench benchmark suite has the following features:

- ADBench consists of 8 benchmarks that have been developed for x86 and ARM architectures. Among them, 4 benchmarks have been highly parallelized using OpenMP for multicore CPUs and also CUDA implementation for NVIDIA GPUs. In the development process of the ADBench suite, several optimization steps have been performed and the latest features of state-of-the-art architectures, such as vector units and tensor cores in GPUs are considered.
- The benchmarks cover a variety of domains in AD systems, from various forms of object detection to advanced localization algorithms and path planning.
- Some workloads with similar algorithm and logic, can be designed and employed in different ways. For instance, the *Camera-based Object Detection* benchmark can employ different DNN models for detecting object from camera data. Consequently, dimensions of the DNN layers, number of layers, performance and other requirements can be tuned as needed. ADBench benchmarks are designed in such a way that this diversity is considered and it is up to the user to configure the benchmark based on his/her needs.

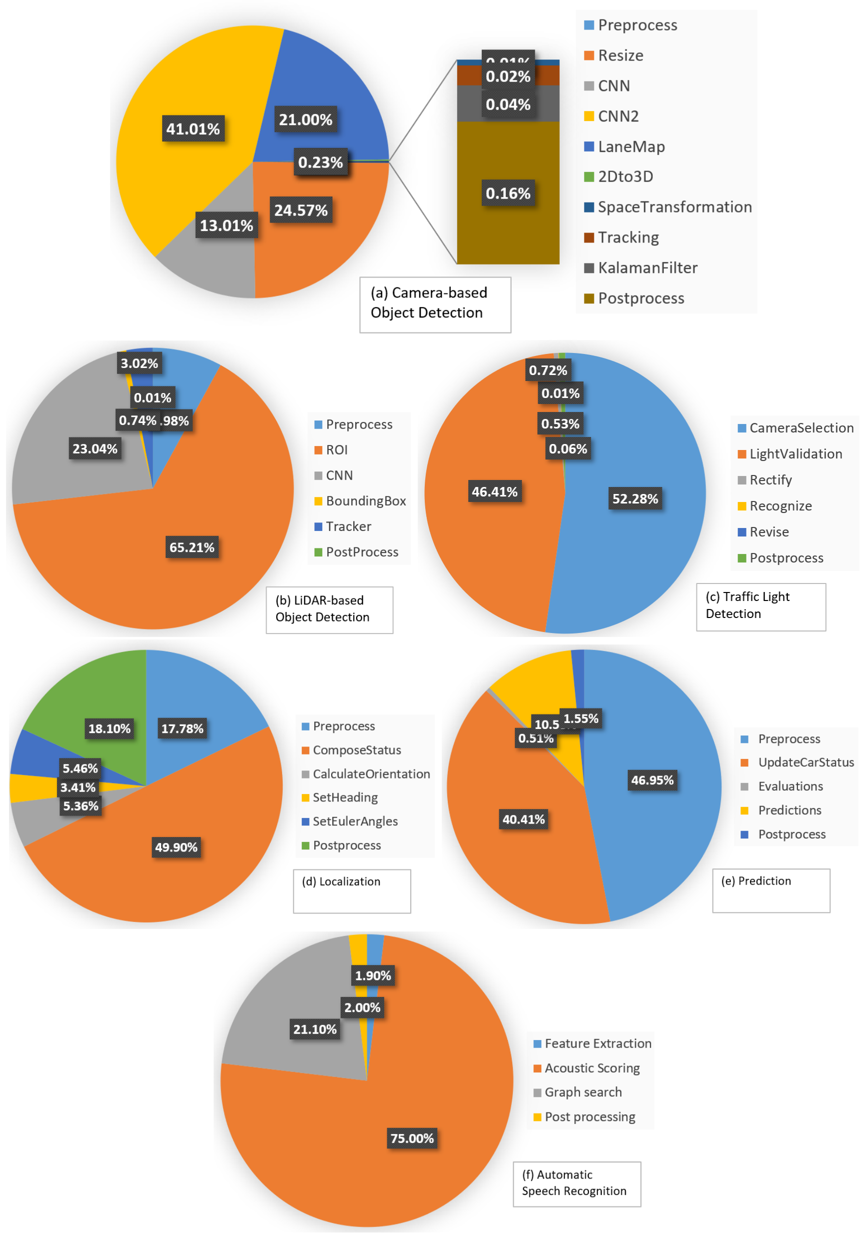

Fig. 6: Execution time breakdown of each benchmark.

## 5 Analysis Results

In this section, we first present the execution time breakdown of each benchmark to highlight the most time-consuming functions (submodules) of each benchmark. Then, as an illustrative example, we focus on more details of one of the most complex benchmarks, the camera-based object detection.

### 5.1 Experimental Platform

We test the ADBench suite on an x86 platform using an AMD Ryzen 7 1800X processor with 8 dual-threaded cores, 64 GB of DDR4 RAM at 2133 MHz, and a Pascal-based high-end GPU (the NVIDIA GeForce 1080 Ti with 3584 CUDA cores). The experimental platform resembles state-of-the-art automotive Systems on Chip (SoCs) targeting the automotive AD market. For example, the two variants of the NVIDIA Drive PX2 platform, AutoCruise and AutoChauffeur, have similar CPU and GPU configurations. The former comprises a single Tegra X2 SoC, which contains 4 ARM Cortex-A57 and 2 Denver cores, combined with an integrated Pascal GPU. The latter contains two Tegra X2 SoCs and 2 discrete Pascal-based GPUs. Moreover, the ARM A57 CPUs used in these platforms exhibit similar hardware complexity as that of the x86 cores in our platform, since both are superscalar, out-of-order CPUs, with several levels of cache.

### 5.2 Execution Time Breakdown

One of the first steps to understand and analyze the behavior of an application is to perform a profiling and obtain the execution time breakdown of its components. Profiling helps to understand where the time and resources are spent when running the application. This is a key step to understand the behavior of the application for further analysis and optimization. In this line, we perform a profiling analysis on the ADBench benchmarks and plot the execution time breakdown for each of the benchmarks, as shown in Figure 6. We used an x86 platform for illustration purposes, although the main trends hold for other platforms equipped with a high-performance GPU. As Figures 6 (a) and 6 (b) show, the CNN evaluation accounts for a significant fraction of the execution time despite it is highly parallelized and runs on the GPU. In the LiDAR-based object detection benchmark, the ROI selection process takes more than 65% of the total execution time.

In Figure 6 (c) camera selection and light validation processes are the ones that consume most of the time. In Localization, Figure 6 (d), the compose status, pre- and post-process are the functions where most of the time is spent on. As we see also in the Prediction benchmark, Figure 6 (e), the preprocess again takes a large fraction of the execution time (45%), which shows how important are these steps, so they are good candidates for any form of optimization.

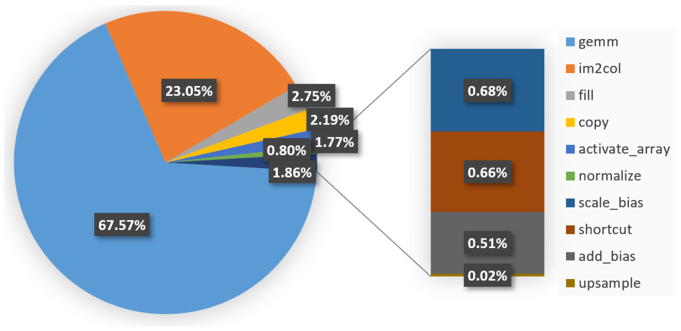

Fig. 7: Execution time breakdown for camera-based object detection CNN kernel.

Lastly, it is not surprising to see that in the ASR benchmark, Figure 6 (f), acoustic scoring takes 75% of the total execution time. Acoustic scoring in one of the steps that has the most significant effects on the accuracy of the ASR system. Therefore, usually sophisticated machine learning and compute-intensive models are employed to improve the accuracy of the ASR system. Note that we do not plot the breakdown of the execution time for the navigator and planning benchmarks due to the fact that they are not composed of several functions and, in fact, there are very few functions in charge of the entire process.

### 5.3 Camera-based Object Detection: GPU Kernels Breakdown

As shown in Figure 6, in object detection benchmarks, the convolutional neural network (CNN) evaluation is one of the key functions accounting for most of the execution time. For this reason, this function is executed on GPUs or specialized accelerators to exploit its huge parallelism. ADBench provides a CUDA kernel implementation using NVIDIA's highly optimized libraries (e.g., cuBLAS and cuDNN). Figure 7 shows the execution time breakdown of this function in the GPU. Due to the nature of CNNs, most of the computations are performed as general matrix multiply (GEMM) operations, as shown in the Figure. After the GEMM function, which consumes more than 67% of the execution time, the image-to-column transformation (im2col kernel), used in convolution to do matrix multiplication by laying out all patches into a matrix, is the most time-consuming kernel.

## 6 Related Work

There are multiple benchmark suites for different domains. However, to our knowledge, we lack a benchmark suite considering state-of-the-art AD systems and their various modules. ADBench aims at providing a comprehensive set of benchmarks for AD systems.

**Deep Learning-based Benchmark Suites**. With the dramatic success of deep learning-based approaches in the past years, various benchmark suites have been developed for different purposes. MLPerf [34] is a comprehensive framework designed focusing on inference accuracy/performance tradeoffs, performance metrics and evaluation, and also on the accelerators design for inference of machine learning workloads. Although some modules, such as object detection, are common in computer vision and AD systems, we focus on features that are more of the interest of AD systems rather than generic computer vision domain.

The EEMBC MLMark [35] is a machine learning benchmark designed to measure the performance and accuracy of embedded inference. The motivation for developing this benchmark grew from the lack of standardization of the environment required for analyzing ML performance. MLMark is targeted at embedded developers, and attempts to clarify the environment in order to facilitate, not just performance analysis of today's offerings, but tracking trends over time to improve new ML architectures.

**Automotive Suites**. EEMBC provides some benchmarks for some control functions of today's cars, but does not support AD systems functionalities. ADASMark [36] focuses on benchmarking a typical vision pipeline that may be used in ADAS platforms. Built on OpenCL, the pipeline may be distributed among CPUs, GPUs and DSPs. However, the main modules and key functions of AD systems are totally different and more complex than those of ADAS.

**Other Benchmarks and Datasets**. The work done in [37] proposes autonomous vehicle benchmarking with unbiased metrics such as fuel consumption, maintenance cost, or trip's time and distance. Its goal is to assess the vehicle's performance in terms of its capabilities to operate without manual driver intervention and its dependability on human input, taking into account the type of road conditions where it operates and the quality of the data. KITTI [38] and nuScenes [39], are benchmarks to test algorithms oriented to AD, such as Object detection algorithms. They contain data from a great variety of sensors in real vehicles. LG SVL [40] is an Autonomous Vehicle Simulator to test the response of an AD system. This is very useful to train and test AD frameworks in all possible cases. In contrast with these Benchmarks/-Datasets, the ADBench suite goal is to test the hardware's performance when running AD Software; therefore, they are used for different purposes.

To our knowledge, ADBench is the first benchmark suite specifically targeting advanced AD systems and can be a reference for future research and development in the community.

## 7 Conclusion

This paper introduces ADBench, the first benchmark suite for AD systems. ADBench focuses on industry-level and state-of-the-art AD systems and provides representative benchmarks targeting the main functions of AD systems. ADBench provides reproducible, realistic, and industry-level benchmarks, which are implemented according to latest automotive hardware platforms. The benchmarks are designed to be flexibly configured accordingly to the user's software and hardware requirements. Standard and representative benchmarks allow developers and designers to perform the analysis, verification and validation process of their solutions easier. Furthermore, ADBench facilitates the quantitative assessment and comparison of various hardware and software solutions.

**Acknowledgements** This work has been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P, the SuPerCom European Research Council (ERC) project under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773), and the HiPEAC Network of Excellence.

## References

1. P. Koopman, M. Wagner, Challenges in autonomous vehicle testing and validation, SAE Int. J. Trans. Safety 4 (2016) 15–24. doi:10.4271/2016-01-0128. URL https://doi.org/10.4271/2016-01-0128
2. M. Alcon, H. Tabani, L. Kosmidis, E. Mezzetti, J. Abella, F. J. Cazorla, Timing of autonomous driving software: Problem analysis and prospects for future solutions, in: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE, 2020, pp. 267–280.
3. M. Alcon, H. Tabani, J. Abella, L. Kosmidis, F. J. Cazorla, En-route: on enabling resource usage testing for autonomous driving frameworks, in: Proceedings of the 35th Annual ACM Symposium on Applied Computing, 2020, pp. 1953–1962.
4. H. Tabani, L. Kosmidis, J. Abella, F. J. Cazorla, G. Bernat, Assessing the adherence of an industrial autonomous driving framework to iso 26262 software guidelines, in: Proceedings of the 56th Annual Design Automation Conference 2019, ACM, 2019, p. 9.
5. S. Alcaide, L. Kosmidis, H. Tabani, C. Hernandez, J. Abella, F. J. Cazorla, Safety-related challenges and opportunities for gpus in the automotive domain, IEEE Micro 38 (6) (2018) 46–55.
6. M. Demler, Xavier simplifies self-driving cars., in: Microprocessors Report, The Linly Group, June 2017.
7. H. Tabani, F. Mazzocchetti, P. Benedicte, J. Abella, F. J. Cazorla, Performance analysis and optimization opportunities for nvidia automotive gpus, Journal of Parallel and Distributed Computing 152 (2021) 21–32.
8. T. Corp., Tesla autopilot, https://www.tesla.com/autopilot (2018).
9. NVIDIA, Self-driving safety report, https://www.nvidia.com/en-us/self-driving-cars/safety-report/ (2018).
10. Baidu, Apollo, an open autonomous driving platform., http://apollo.auto/ (2018).
11. The Autoware Foundation, Autoware: An open autonomous driving platform, https://github.com/CPFL/Autoware/ (2016).
12. ApolloAuto, Apollo 3.0 Software Architecture (2018). URL https://github.com/ApolloAuto/apollo/blob/master/docs/specs/Apollo_3.0_Software_Architecture.md
13. R. Pujol, H. Tabani, L. Kosmidis, E. Mezzetti, J. Abella, F. J. Cazorla, Generating and exploiting deep learning variants to increase heterogeneous resource utilization in the nvidia xavier, in: 31st Euromicro Conference on Real-Time Systems (ECRTS 2019), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
14. D. Merkel, Docker: lightweight linux containers for consistent development and deployment, Linux journal 2014 (239) (2014) 2.

15. M. Quigley, et al., ROS: an open-source robot operating system, in: ICRA workshop on open source software, Vol. 3, Kobe, Japan, 2009, p. 5.
16. ApolloAuto, Perception (2018).
URL https://github.com/ApolloAuto/apollo/blob/r3.0.0/docs/specs/perception_apollo_3.0.md
17. H. Tabani, R. Pujol, J. Abella, F. J. Cazorla, A cross-layer review of deep learning frameworks to ease their optimization and reuse, in: 2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC), IEEE, 2020, pp. 144–145.
18. M. Quigley et al., ROS: an open-source Robot Operating System, ICRA Workshop on Open Source Software.
19. R. C. Whaley, D. Aberdeen, M. Brett, N. Coult, T. Castaldo, M. Dittrich, D. Gaudet, K. Goto, J. Horner, C. Maguire, T. Mattox, H. Deitz, V. Nguyen, P. Strazdins, J. Ruhe, P. Soendergaard, C. Staelin, Automatically Tuned Linear Algebra Software (ATLAS), http://math-atlas.sourceforge.net/ (2018).
20. Z. Xianyi, W. Qian, W. Saar, Z. Chothia, C. Shaohu, L. Wen, et al., An optimized BLAS library (OpenBLAS), http://www.openblas.net/ (2020).
21. NVIDIA, cuBLAS, http://docs.nvidia.com/cuda/cublas/ (2021).
22. S. Chetlur et al., cudnn: Efficient primitives for deep learning, arXiv preprint arXiv:1410.0759.
23. L. Blackford et al., An updated set of basic linear algebra subprograms (blas), ACM Transactions on Mathematical Software 28 (2) (2002) 135–151.
24. Intel, Intel oneAPI Math Kernel Library: The fastest and most-used math library for Intel-based systems, https://software.intel.com/en-us/intel-mkl (2020).
25. NVIDIA, TensorRT: A platform for high-performance deep learning inference, https://developer.nvidia.com/tensorrt (2021).
26. J. Redmon, A. Farhadi, Yolov3: An incremental improvement, CoRR abs/1804.02767. arXiv:1804.02767.
URL http://arxiv.org/abs/1804.02767
27. ApolloAuto, 3D Obstacle Perception, https://github.com/ApolloAuto/apollo/blob/r3.0.0/docs/specs/3d_obstacle_perception.md (2018).
28. ApolloAuto, Traffic Light Perception, https://github.com/ApolloAuto/apollo/blob/master/docs/specs/traffic_light.md (2018).
29. ApolloAuto, Multi-sensor Fusion Localization, https://github.com/ApolloAuto/apollo/blob/r3.0.0/modules/localization/msf/README.md (2018).
30. G. Wan, X. Yang, R. Cai, H. Li, Y. Zhou, H. Wang, S. Song, Robust and precise vehicle localization based on multi-sensor fusion in diverse city scenes, in: 2018 IEEE International Conference on Robotics and Automation (ICRA), 2018, pp. 4670–4677.
31. D. Dolgov, S. Thrun, M. Montemerlo, J. Diebel, Path planning for autonomous vehicles in unknown semi-structured environments, The International Journal of Robotics Research 29 (5) (2010) 485–501.
32. ApolloAuto, Planning, https://github.com/ApolloAuto/apollo/tree/r3.5.0/modules/planning/README.md (2018).
33. ApolloAuto, Prediction, https://github.com/ApolloAuto/apollo/blob/r3.0.0/modules/prediction/README.md (2018).
34. V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, et al., Mlperf inference benchmark, in: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), IEEE, 2020, pp. 446–459.
35. EEMBC, Introducing the EEMBC MLMark Benchmark, https://www.eembc.org/mlmark/ (2019).
36. EEMBC, The ADASMark$^{TM}$ Benchmark: A Performance Measurement and Optimization Tool for Automotive Companies Building Next-Generation Advanced Driver-Assistance Systems (ADAS), 2019.
37. D. Paz, P.-j. Lai, N. Chan, Y. Jiang, H. I. Christensen, Autonomous vehicle benchmarking using unbiased metrics, arXiv preprint arXiv:2006.02518.
38. A. Geiger, P. Lenz, R. Urtasun, Are we ready for autonomous driving? the kitti vision benchmark suite, in: Conference on Computer Vision and Pattern Recognition (CVPR), 2012.

39. H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, O. Beijbom, nuscenes: A multimodal dataset for autonomous driving, arXiv preprint arXiv:1903.11027.
40. G. Rong, B. H. Shin, H. Tabatabaee, Q. Lu, S. Lemke, M. Možeiko, E. Boise, G. Uhm, M. Gerow, S. Mehta, et al., Lgsvl simulator: A high fidelity simulator for autonomous driving, arXiv preprint arXiv:2005.03778.