2018 9th ACM/IEEE International Conference on Cyber-Physical Systems

Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems

Shinpei Kato*[‡§], Shota Tokunaga[†], Yuya Maruyama[†], Seiya Maeda[†], Manato Hirabayashi[‡],
Yuki Kitsukawa[‡], Abraham Monrroy[‡], Tomohito Ando[‡], Yusuke Fujii[§], and Takuya Azumi[†§¶]
[*] *Graduate School of Information Science and Technology, The University of Tokyo*
[†] *Graduate School of Engineering Science, Osaka University*
[‡] *Graduate School of Informatics, Nagoya University*
[§] *Tier IV, Inc.*
[¶] *JST, PRESTO*

Abstract—**This paper presents *Autoware on Board*, a new profile of Autoware, especially designed to enable autonomous vehicles with embedded systems. Autoware is a popular open-source software project that provides a complete set of self-driving modules, including localization, detection, prediction, planning, and control. We customize and extend the software stack of Autoware to accommodate embedded computing capabilities. In particular, we use DRIVE PX2 as a reference computing platform, which is manufactured by NVIDIA Corporation for development of autonomous vehicles, and evaluate the performance of Autoware on ARM-based embedded processing cores and Tegra-based embedded graphics processing units (GPUs). Given that low-power CPUs are often preferred over high-performance GPUs, from the functional safety point of view, this paper focuses on the application of Autoware on ARM cores rather than Tegra ones. However, some Autoware modules still need to be executed on the Tegra cores to achieve load balancing and real-time processing. The experimental results show that the execution latency imposed on the DRIVE PX2 platform is capped at about three times as much as that on a high-end laptop computer. We believe that this observed computing performance is even acceptable for real-world production of autonomous vehicles in certain scenarios.**

1. Introduction

Autonomous vehicles are becoming the next core of mobility infrastructures. They will improve our quality of life and social welfare significantly in the near future. This paradigm shift has encouraged automotive manufacturers to develop production-quality autonomous vehicles and even plan their mass production. Electronics and semiconductor manufacturers, as well as technology suppliers, are also eager to participate in strategic cross-modal alliances with automotive manufacturers. This alliance trend is due to the fact that applications of autonomous vehicles are truly diverse. Autonomous *individual owned* vehicles could reduce traffic accidents and mitigate traffic jams. Autonomous *mobility-as-a-service* vehicles, on the other hand, could provide more efficient logistics and passenger services. There are many other potential markets for autonomous vehicles, and technology transfer from academia to industry is now really needed.

Common functional pieces of autonomous vehicles often fall into sensing, computing, and actuation. Examples of sensing devices or sensors include cameras, laser scanners (a.k.a., LiDAR), milliwave radars, and GNSS/IMU. Using sensor data, autonomous vehicles perform localization, detection, prediction, planning, and control, which are core modules of computing intelligence. Classical algorithms are in favor of rule-based systems and pattern recognition, while recent data-driven approaches to new algorithms leverage machine learning and deep neural networks. More efficient approaches and algorithms are still extensively studied today. Actuation of autonomous vehicles, finally, require a by-wire controller to manipulate steering (angular velocity) and throttle (linear velocity). Towards real-world production, a secure gateway must be installed between the original CAN-bus controllers and the new by-wire controller to protect primitive vehicle manipulation from cyber-attacks. These technical challenges of autonomous vehicles indeed represent the domain of cyber-physical systems.

Despite the well-studied entire architecture and individual algorithms, not much is known about the detail of computation for autonomous vehicles. In particular, there are few case studies on embedded systems, concerning real problems of real vehicles. This is largely due to the fact that common libraries and platforms have not been provided for the community. To overcome this circumstance, we have developed open-source software, called Autoware [1], [2] which includes a rich set of software packages and libraries required for autonomous vehicles. Autoware is now widely used for research and development, and is probably the largest open-source project for self-driving technology.

Contribution: This paper aims at building Autoware with embedded systems while clarifying its performance in real environments. To this end, we customize and extend Autoware to accommodate with ARM-based embedded processing cores and Tegra-based embedded graphics processing units (GPUs) of DRIVE PX2, which is manufactured by NVIDIA Corporation for autonomous vehicles. To the best of our knowledge, this is the first work that clarified the performance of complete software stack for autonomous

0-7695-6378-3/18/$31.00 ©2018 IEEE
DOI 10.1109/ICCPS.2018.00035

IEEE computer society

vehicles with embedded systems. This paper is more interested in ARM cores than Tegra cores, because low-power CPUs are often preferred to high-performance GPUs from the functional safety point of view. It is true that GPUs are more promising solutions to break through a trade-off relation between power and performance of autonomous vehicles. Thus, we will explore more detailed advantages of GPUs for Autoware in future work.

Organization: The rest of this paper is organized as follows. Section 2 describes the system model of Autoware, including the system stack and underlying basic software. Section 3 presents the detail of Autoware with a particular emphasis on the modules that we make a significant implementation effort for DRIVE PX2. Section 4 evaluates the performance of Autoware on Board using DRIVE PX2. Related work is discussed in Section 5, and this paper concludes in Section 6.

2. System Model

Autonomous vehicles as cyber-physical systems can be abstracted into sensing, computing, and actuation modules, as shown in Figure 1. Sensing devices, such as laser scanners (LiDAR) and cameras, are typically used for self-driving in urban areas. Actuation modules handle steering and stroking whose twisted control commands that are typically generated by the path following module. Computation is a major component of self-driving technology. Scene recognition, for instance, requires the localization, detection, and prediction modules, whereas path planning is handled by mission-based and motion-based modules. Each module employs its own set of algorithms. The modules implemented in Autoware are discussed in Section 3.

Figure 1 shows the basic control and data flow for an autonomous vehicle. Sensors record environmental information that serves as input data for the artificial intelligence core. 3D maps are becoming a commonplace for self-driving systems, particularly, in urban areas as a complement to the planning data available from sensors. External data sources can improve the accuracy of localization and detection without increasing the complexity of the vehicle's algorithms. Artificial intelligence cores typically output values for angular and linear velocities, which serve as commands for steering and stroking, respectively.

2.1. System Stack

We designed a complete software stack for autonomous vehicles that are implemented with open-source software, as shown in Figure 2. The self-driving technology discussed herein is intended for vehicles that are driven in urban areas instead in freeways or highways. We contributed to the development of Autoware, a popular open-source software project developed for autonomous vehicles intended for using in urban areas. Autoware is based on Robot Operating System (ROS) and other well-established open-source software libraries, as shown in Figure 2. Section 2.2 provides a brief overview of ROS. Point Cloud Library (PCL) [3] is mainly used to manage LiDAR scans and 3D mapping data, in addition to performing data-filtering and visualization

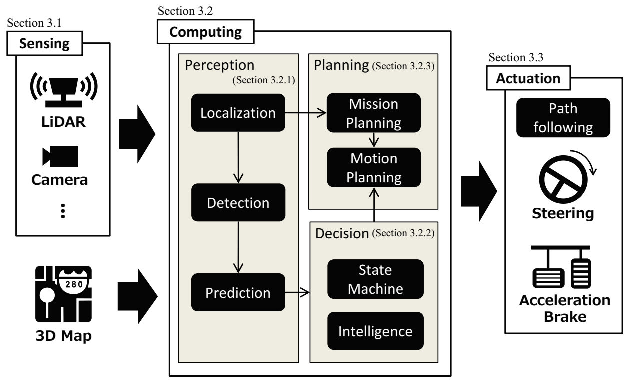


Figure 1. Basic control and data flow of autonomous vehicles.

functions. CUDA [4] is a programming framework developed by NVIDIA and is used for general-purpose computing on GPUs (GPGPU). To handle the computation-intensive tasks involved in self-driving, GPUs running CUDA are promising solutions for self-driving technology, though this paper does not focus on them. Caffe [5], [6] is a deep learning framework designed with expression, speed, and modularity in mind. OpenCV [7] is a popular computer vision library for image processing.

Autoware uses such tools to compile a rich set of software packages, including sensing, perception, decision making, planning, and control modules. Many drive-by-wire vehicles can be transformed into autonomous vehicles with an installation of Autoware. In several countries, successful implementations of Autoware have been tested for autonomous vehicles that are driven for long distances in urban areas. Recently, automotive manufacturers and suppliers have begun implementing Autoware as a baseline for prototype autonomous vehicles.

This section briefly introduces the hardware components that are generally installed in autonomous vehicles. Commercially available vehicles require minor modifications for a successful application of our software stack. This will enable us to control the vehicles using external computers through a secure gateway and install sensors on the vehicles. The vehicle, computers, and sensors can be connected by Controller Area Network (CAN) bus, Ethernet, and/or USB 3.0. However, the specifications of sensors and computers required for a particular application depend closely on specific functional requirements of the autonomous vehicle. Autoware supports range sensor, and we chose Velodyne HDL-32e LiDAR scanners and PointGrey Grasshopper3 cameras for our prototype system. Autoware also supports a range of processors. Many users run Autoware on desktops and laptops. The NVIDIA DRIVE PX2 is used herein though our previous study ported Autoware to another embedded computing platform, the Kalray Massively Parallel Processor Array (MPPA) [8].

2.2. Robot Operating System (ROS)

As mentioned earlier, Autoware is based on ROS [9], [10], which is a component-based middleware framework

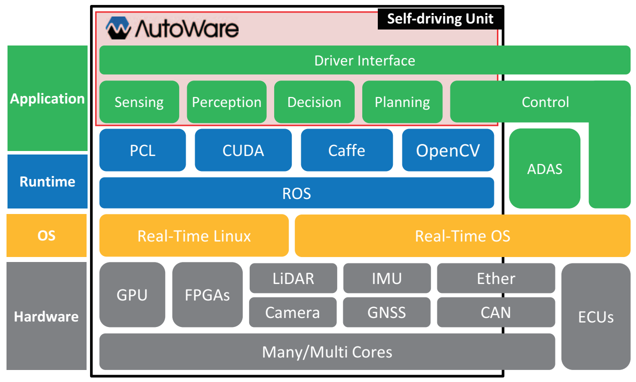


Figure 2. Complete software stack for autonomous vehicles using Autoware.

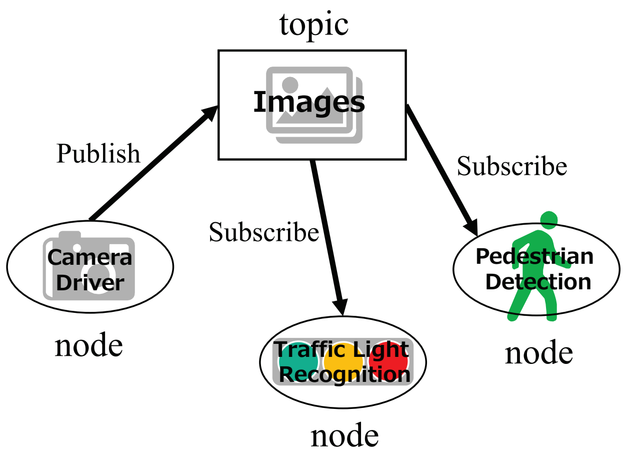


Figure 3. The publish/subscribe model in ROS.

developed for robotics research. ROS is designed to enhance the modularity of robot applications at a fine-grained level and is suitable for distributed systems, while allowing for efficient development. Since autonomous vehicles require many software packages, ROS provides a strong foundation for Autoware's development.

In ROS, the software is abstracted as *nodes* and *topics*. The *nodes* represent individual component modules, whereas the *topics* mediate inputs and outputs between *nodes*, as illustrated in Figure 3. ROS *nodes* are usually standard programs written in C++. They can interface with any other software libraries that are installed.

Communication among *nodes* follows a publish/subscribe model. This model is a strong approach for modular development. In this model, *nodes* communicate by passing *messages* via a *topic*. A *message* is structured simply (almost identical to structs in C) and is stored in .msg files. *Nodes* identify the content of the *message* in terms of the *topic* name. When a *node* publishes a *message* to a *topic*, another *node* subscribes to the *topic* and uses the *message*. For example, in Figure 3, the "Camera Driver" *node* sends *messages* to the "Images" *topic*. The *messages* in the *topic* are received by the "Traffic Light Recognition" and "Pedestrian Detection" *nodes*. *Topics* are managed using first-in, first-out queues when accessed by multiple *nodes* simultaneously. At the same time, ROS *nodes* can launch several threads implicitly; however, issues with real-time processing must be addressed.

ROS also includes an integrated visualization tool, RViz, and a data-driven simulation tool, ROSBAG. Figure 4 (a) shows examples of visualizations produced by RViz for perception tasks in Autoware. The RViz viewer is useful for monitoring the status of tasks. ROSBAG is a set of tools for recording and playing back ROS *topics*. It provides development environments for testing self-driving algorithms without hardware, thereby making development processes more efficient.

3. Autoware

This section describes the software stack of Autoware as highlighted in Figure 2, and emphasizes on the extensions required to adapt it to the DRIVE PX2 platform. The complete Autoware system can be analysed on its project repository [1]. It is important to note that Autoware is designed for such autonomous vehicles driven in urban areas. Therefore, additional modules might be required to adapt it for freeways or highways. Discussions regarding accuracy and optimization for each algorithm implemented in Autoware are beyond the scope of this paper.

3.1. Sensing

Autoware mainly recognizes road environments with the help of LiDAR scanners and cameras. LiDAR scanners measure the distance to objects by illuminating a target with pulsed lasers and measuring the time of the reflected pulses. Point cloud data from LiDAR scanners can be used to generate digital 3D representations of the scanned objects. Cameras are predominantly used to recognize traffic lights and extract additional features of the scanned objects.

To achieve real-time processing, Autoware filters and pre-processes the raw point cloud data obtained from the LiDAR scanners. As a case in point, voxel grid filtering is required to localize, map, and detect objects in the point cloud. This kind of filtering downsamples the point cloud, replacing a group of points contained in a cubic lattice grid to its centroid. Having this downsampled point cloud, the Normal Distributions Transform (NDT) algorithm [11] can obtain the relative position between two point clouds [12]. Autoware takes advantage of this result to get an accurate position between the 3D map and the LiDAR scanner mounted on the vehicle.

Data from other sensors such as radars, GNSS, and IMU can be used to refine the localization, detection, and mapping. Radars are already available in commercial vehicles and are often used for ADAS safety applications. GNSS and IMU sensors can be coupled with gyroscope sensors and odometers to improve the positioning information.

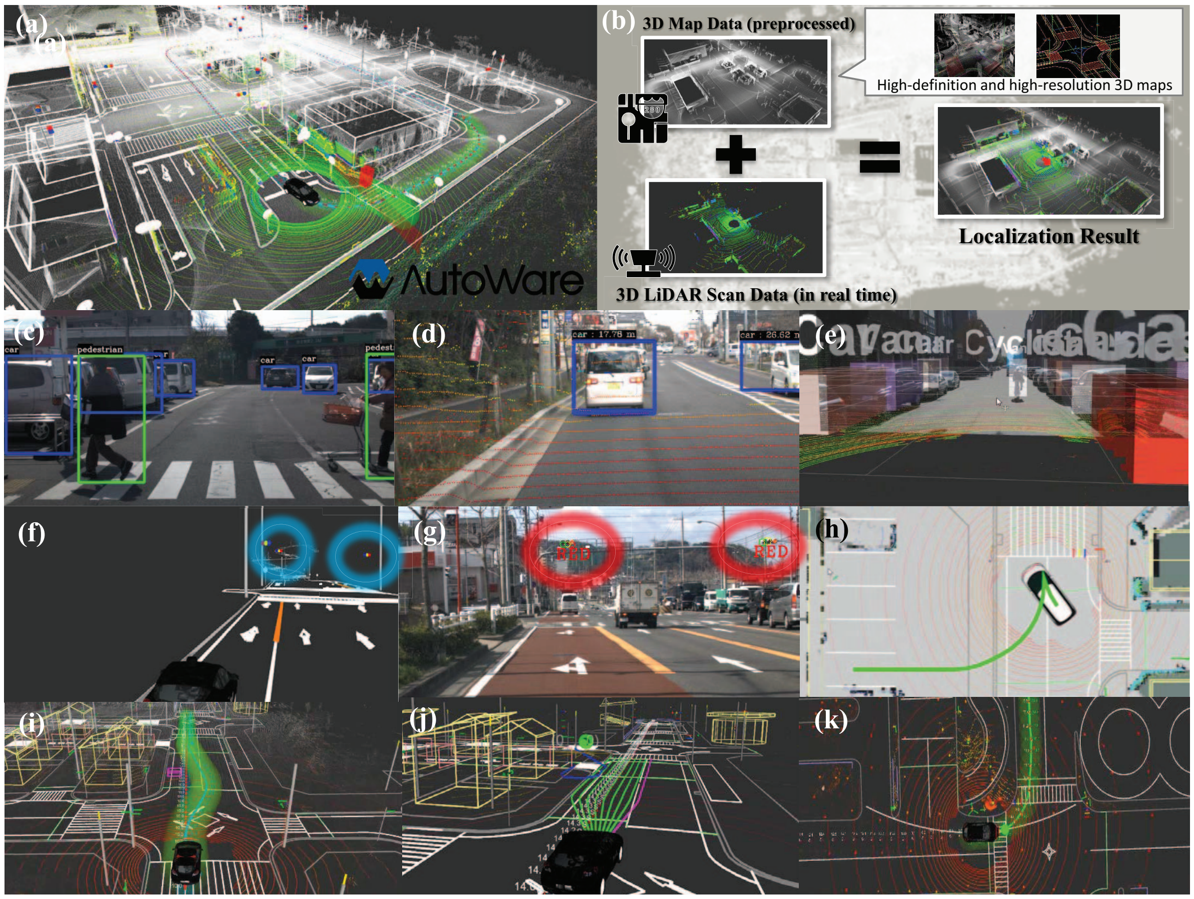

(a)
AutoWare
(b)
3D Map Data (preprocessed)
High-definition and high-resolution 3D maps
Localization Result
3D LiDAR Scan Data (in real time)
(c)
car
pedestrian
(d)
car : 17.78 m
car : 26.62 m
(e)
(f)
(g)
RED
(h)
(i)
(j)
(k)

Figure 4. Autoware's self-driving packages: (a) RViz visualization with high-definition and high-resolution geographical information, (b) NDT scan matching localization using a 3D map and a 3D LiDAR scan, (c) deep learning based object detection based on You only look once (Yolo2) algorithm, (d) projection of the 3D point-cloud data onto an image, (e) fusion for calibrated point clouds and images, (f) traffic light positions extracted from the 3D map, (g) traffic light recognition with the region of interest marked, (h) trajectory planning in a parking lot using hybrid-state A* search, (i) trajectory planning for object avoidance using hybrid-state A* search, (j) trajectory generation for object avoidance using the state-lattice algorithm, and (k) steering angular velocity calculations for path following using the pure pursuit algorithm.

3.2. Computing

The computing intelligence is an essential component for autonomous vehicles. Using sensor data and 3D maps, the vehicle can compute the final trajectory and communicate with the actuation modules. This section explains the core modules used for perception, decision-making, and planning.

3.2.1. Perception. Safety in autonomous vehicles is a high-priority issue. Therefore, the perception modules must calculate the accurate position of the ego-vehicle inside a 3D map, and recognize objects in the surrounding scene as well as the status of traffic signals.

Localization: The localization module is one of the crucial parts of the self-driving system. Many other perception modules rely on the location of the vehicle. The localization algorithms implemented in Autoware exploit scan matching between 3D maps and LiDAR scanners. The average error is in the order of a few centimeters. Furthermore, localization is also required to create 3D maps. This approach is often referred to as Simultaneous Localization And Mapping (SLAM). To create a map using this strategy, the matching process is performed against the previous LiDAR scan, instead of a 3D map. Once the transformation has been obtained, the current scan is transformed to the coordinate system of the previous one, and the point cloud is summed up. This is repeated continuously when building the map.

Autoware mainly uses the NDT [11] algorithm for localization. This is because the computational cost of the NDT algorithm is not dominated by the map size, thus enabling the deployment of high-definition and high-resolution 3D maps at larger scales.

Autoware also supports other localization and mapping algorithms such as the Iterative Closest Point (ICP) algorithm [13]. This allows Autoware users to select the best-suited algorithm for their applications.

Detection: Autonomous vehicles must detect the surrounding objects, such as other vehicles, pedestrians, and traffic signals. Having this information readily available, we can follow traffic rules and avoid accidents.

Autoware supports deep learning [14], [15] and traditional machine learning approaches [16] for object detection using Caffe and OpenCV. Figure 4 (c) shows the application of the Single Shot MultiBox Detector (SSD) [14] and the You only look once (Yolo2) algorithms [15]. Both are based on fully convolutional neural networks (CNN) architectures designed to provide fast computation. As previously mentioned, Autoware also includes a pattern recognition algorithm based on the Deformable Part Models (DPM) [16], which searches and scores the histogram of oriented gradients features of target objects in 2D images [17].

Perception in Autoware is mainly performed using point cloud data obtained from LiDAR scanners. The point cloud is pre-processed and segmented using the nearest neighbors algorithm [18]. This process calculates the Euclidean distance between points according to a distance threshold in 3D space. Once the point cloud is clustered, the distance between surrounding objects and the ego-vehicle can be calculated. Moreover, using the distance of each cluster in conjunction with the classified 2D image processing algorithms, objects can be tracked on a time to improve the perception information.

Assuming that localization in a 3D map is precise, Autoware implements an elegant scheme for traffic light recognition. Knowing the current position of the ego-vehicle in a map, and having an extrinsically and intrinsically calibrated camera and LiDAR scanner, the 3D map features can be projected to the image captured by the front camera of the vehicle, with the body rigid transformations across the camera, the LiDAR scanner, and the 3D map. Using this strategy, the region of interest (ROI) for traffic lights can be limited, reducing the computation time and increasing the accuracy of the traffic light state classifiers at the same time. Figures 4 (f) and (g) depict the result of this approach. In general, traffic light recognition is considered to be one of the most difficult problems associated with autonomous vehicles. Nonetheless, Autoware is able to accurately recognize traffic lights using 3D map features and a highly-reliable localization algorithm.

Prediction: Object detection algorithms are computed on each synchronized camera frame and point cloud scan. These results must be associated among the time, so the trajectories of moving objects can be estimated for the mission and motion planning modules.

In Autoware, Kalman filters and particle filters are employed to solve these problems. The Kalman filters are used under the assumption that the ego-vehicle is driving at a constant velocity while tracking moving objects [19]. The computational cost of the Kalman filters is lightweight, thereby making it suitable for real-time processing. Particle filters, in contrast, can be applied to non-linear tracking scenarios, which are appropriate for realistic driving [20]. Autoware uses both Kalman filters and particle filters, depending on the given scenario. They are used to track in both the 2D image plane and the 3D point cloud plane.

The detected and tracked objects in the 2D image can be combined with clustered and tracked objects obtained from the LiDAR scanners; this idea is also known as “sensor fusion” with projection and re-projection. The sensor fusion parameters are determined with the calibrated LiDAR scanners and cameras.

Point cloud can be projected onto the 2D image, thereby adding depth information to the image. Figures 4 (d) and (e) show the result of bounding box calculation to clustered 3D point cloud objects as projected to the camera image.

Another supported approach consists on back-projecting the detected and tracked objects on the image, onto the 3D point cloud coordinates, using the same extrinsic parameters. These reprojected positions can partially be used to determine the motion and mission plans.

3.2.2. Decision.

Once the obstacles and traffic signals are detected, the trajectories of other moving objects can be estimated. Mission planning and decision making modules use these estimated results to determine an appropriate direction to which the ego-vehicle should move.

Autoware implements an intelligent state machine to understand, forecast, and make decisions in response to the road status. Moreover, Autoware also allows persons in the ego-vehicle to supervise automation, overwriting the state determined by the planning module.

3.2.3. Planning.

The planning module generates trajectories following the output from the decision-making module. Path planning can be classified into mission and motion planning. Autoware makes a plan for global trajectories based on the current location and the given destination. Local trajectories, on the other hand, are generated by the motion planning module, along with the global trajectories. Available graph-search algorithms include hybrid-state A* [21], and trajectory generation algorithms [22] such as lattice-based methods [23]. A motion planning module suited to the decision-making module is selected according to the road scenario.

Mission planning: Depending on the driving states, such as lane changes, merges, and passing, the mission planner uses a rule-based system to determine path trajectories. Navigation from the current position to the destination is also included as part of mission planning. The high-definition 3D map information contains static road features that can be used for this mission planning function. In more complex scenarios, such as parking and rerouting, persons in the ego-vehicle could supervise the mission capability. In either case, once the global path is generated, the motion planning module is launched to make a plan for local trajectories.

The basic policy of the mission planner is to follow the center lines of the lanes over the route generated by the map navigation system, involving the high-definition 3D map information. The mission planner changes the lane only when the ego-vehicle needs to pass a preceding vehicle, or approaches an intersection followed by a turn.

Motion planning: The motion planner is in charge of generating local feasible trajectories along with the given global trajectories, taking into account the vehicle states, the drivable areas indicated by the 3D map, the surrounding objects, the traffic rules, and the desired goal.

When planning on unstructured environments, such as parking lots, graph-search algorithms, such as A* [24] and hybrid-state A* [21] are used to find the minimum-cost path to the goal. Figures 4 (h) and (i) show these methods in action. Although graph-search algorithms require considerable computing capabilities, they can even analyze complex scenarios. On the other hand, when navigating on structured environments, such as roads and traffic lanes, vertices are likely to be dense and irregularly distributed. This provides constrained options for feasible trajectories. For this reason, Autoware preferably uses the spatiotemporal lattice-based algorithms to adapt motion plans to urban environments [25]. State-of-the-art research has also encouraged implementation of these algorithms in previous autonomous vehicles [26]. Trajectories for obstacle avoidance and lane changes must be calculated in real time. The lattice-based algorithms [27], [25] are commonly affordable for real-time processing. Figure 4 (j) outlines the trajectories generated by the lattice planner and one trajectory is selected by the cost evaluation.

3.3. Actuation

Once local trajectories are determined, autonomous vehicles need to follow them.

Path following: The pure pursuit algorithm [28] is used to generate the actuation commands for the ego-vehicle. This method breaks down the path into multiple waypoints, which are discrete representations of the path. During every control cycle, the algorithm searches for the next closest waypoint in the ego-vehicle's heading direction. The waypoint is searched under a threshold distance. This reduces the abrupt changes in angle that might occur when turning or following a deviation path. As shown in Figure 4 (k), the velocity and angle of the following movement are set to the values that will move the vehicle to the selected waypoint following a predefined curvature.

Vehicle control: The target waypoint is continuously updated until the goal is reached. In other words, the ego-vehicle continues following these waypoints until it reaches the user-specified destination. However, if the steering and throttle of the ego-vehicle are not successfully controlled in accordance with the velocity and angle output produced by the pure pursuit algorithm, the ego-vehicle can temporarily deviate from the planned trajectory. To actuate the steering and throttle based on the velocity and angle commands, a PID controller is often applied. The parameters of the PID controller highly depend on the given vehicle platform. In certain scenarios where the parameters are not correctly tuned, the PID controller will not be able to control the vehicle stably. If localization is not accurate, even worse, this could cause gaps between the vehicle state and the outputs of the pure pursuit algorithm. In extreme cases, the ego-vehicle could collide with unexpected obstacles. To cope with these accidental scenarios, the path following module must ensure a minimum distance between the ego-vehicle and detected obstacles, overriding the planned trajectory. The motion planner presented in this paper indeed reacts to this scenario, by updating the waypoints in consideration of obstacles in the heading lane area.

TABLE 1. INFORMATION OF EACH EVALUATED NODE AND ITS TOPICS.

package	*Topic*	
	Publish	Subscribe
Points localizer	Current vehicle pose	Filtered 3D point-cloud data
Clustering	Cluster information	Filtered 3D point-cloud data
Image detector	Objects detected in the image	Image captured form a camera

4. Evaluation

This section provides a performance evaluation of Autoware on the DRIVE PX2 platform, as compared to that on a high-end computer. DRIVE PX2 is currently one of the most computationally capable embedded multicore platforms integrated with high-performance GPUs. We believe that the performance result obtained using the DRIVE PX2 platform is a highly useful contribution for the future development of autonomous vehicles.

The minimum set of perception packages required to operate autonomous vehicles includes the localization, clustering, and detection packages. The functional nodes incorporated in these packages require non-trivial computational costs. Due to a space constraint, performance evaluations of the planning and the control modules are omitted. In consequence, we evaluate the performance of those packages highlighted in Figure 5.

We define an end-to-end time as the total run-to-completion time taken for a specific given set of packages. Measuring this end-to-end time, we can also evaluate that the tested computing platforms meet timing requirements of sensing and actuation. The functional nodes highlighted by the red color in Figure 5 are those measured as the end-to-end time in this evaluation. Note that a series of measurements was conducted using ROSBAG recordings of real data obtained from LiDAR scanners and cameras mounted with our autonomous vehicle. In particular, we used a Velodyne VLP-16 LiDAR scanner and a PointGrey Grasshopper3 camera to obtain point cloud and image data, respectively. The LiDAR scanner produces about 30,000 points at 10 Hz, while the camera captures a three color channel image of 960x604 pixels at 20 Hz.

4.1. Experimental Setup

Experiments were conducted using Autoware v1.5.1, CUDA 8.0, and ROS Kinetic on two machines: DRIVE PX2 (AutoChauffeur) [29] and a high-end laptop computer with a GPU. The DRIVE PX2 platform integrates two NVIDIA TEGRA X2 processors [30] (referred as TEGRA A and TEGRA B, henceforth) and a single GPU. Table 2 summarizes the specifications of the DRIVE PX2 platform and the laptop computer.

4.2. Measurement

The execution time for each tested package and the end-to-end time were measured as the difference in time between the start and the end times of the corresponding process,

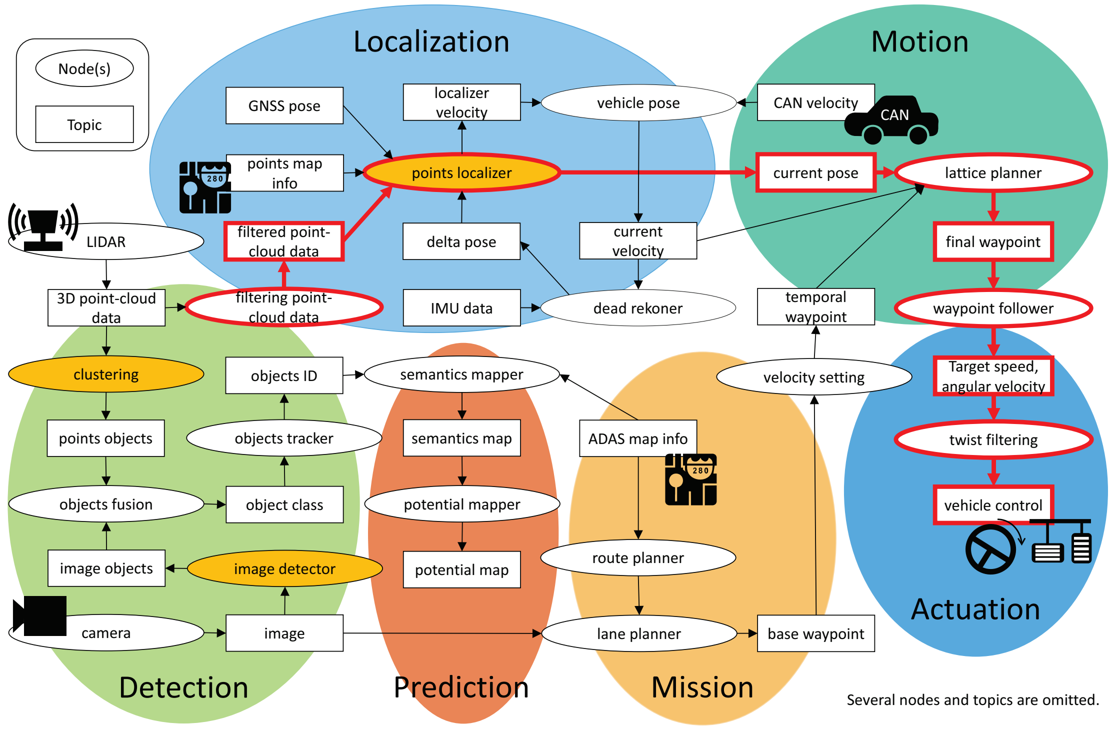


Figure 5. A package configuration diagram of Autoware. Some packages include multiple *nodes*.

TABLE 2. EXPERIMENTAL SETUP.

	CPU			Discrete GPU		Memory
	Model name	Cores	Frequency	Architecture	Cores	
The laptop computer	Intel Core i7-6700K	4	4.0 GHz	Maxwell	2,048	32 GB
DRIVE PX2 (AutoChauffeur)	2× ARMv8 Denver 2.0	2× 2	2.00 GHz	2× Pascal	2× 1152	2× 8 GB
	2× ARMv8 Coretex A57	2× 4	2.00 GHz			

which are acquired by the *timespec_get* function. The precision of this function is a nanoseconds order. The overhead of function calls was also considered. Each measurement was repeated 1,000 times, and the average performance was obtained.

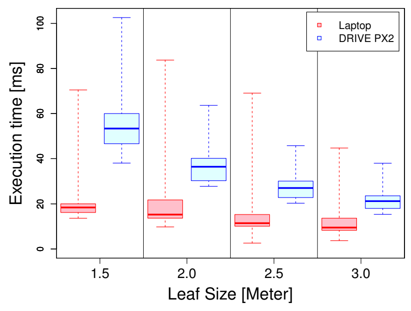
4.2.1. Execution time of target processes. The execution time for each tested package was obtained as the difference in time between the subscription to the *topic* shown in Table 1 and the publication of the computation result. In these measurements, only the tested package and a minimum set of necessary *nodes* were executed to avoid additional overhead and system load. The measurements conducted on the laptop computer executed all the relevant processes on the on-board CPU and GPU, whereas those conducted on the DRIVE PX2 platform distributed the relevant processes in such a way that the tested package was executed on the TEGRA A and the remaining *nodes* were executed on TEGRA B.

Points localizer: The points localizer package was implemented using the NDT algorithm as described in Section 3.1. The points localizer calculates the position and orientation of the autonomous vehicle using point cloud data downsampled by the voxel grid filter.

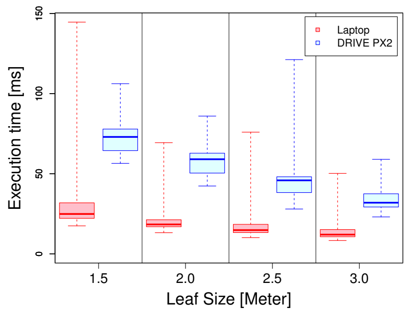
The execution times, when using point cloud data extending to 50, 100, and 200 meters around the autonomous vehicle, are shown in Figure 6. The horizontal axis of each figure represents the leaf size, which is the length of the side of the voxels. The result shows that larger leaf sizes reduce the execution time on both the DRIVE PX2 platform and the laptop computer. The execution time is also shorter when filtering the point cloud data out to only those located close to the LiDAR scanner position. These results indicate that the execution time decreases as the number of the voxels to be processed decreases.

However, if the leaf size is too large, the computation of the NDT algorithm does not converge, and as a result, the execution time could become very long due to many trials of scan matching, which are never converged. This means that localization is only reliable within a certain threshold of resolution for the point cloud data. Therefore, the execution time and the accuracy of localization must be balanced with appropriate parameters under practical requirements.

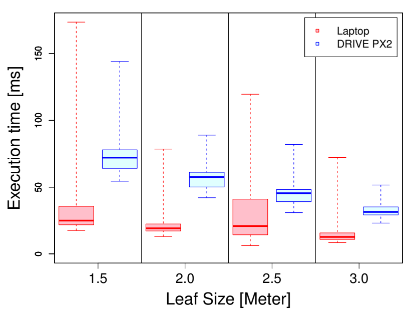
Clustering: The clustering package plays a segmentation of objects using point cloud data. Points above the vehicle and below the ground are ignored by specifying the parameters (clip_min_height and clip_max_height). In this evaluation, we set clip_min_height = -1.3 and



(1) Using point cloud data up to 50 meters.



(2) Using point cloud data up to 100 meters.



(3) Using point cloud data up to 200 meters.

Figure 6. Execution time of the points localizer package.

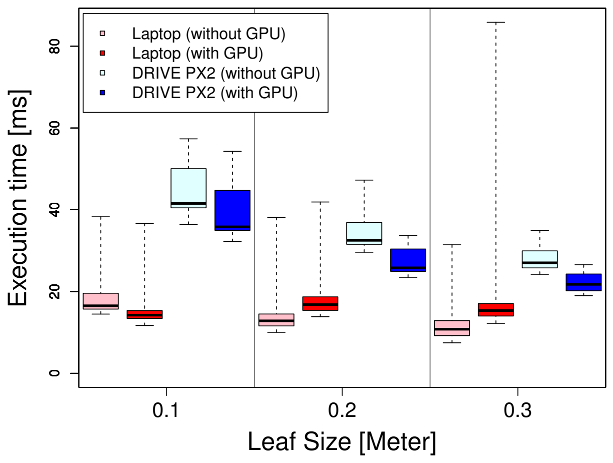


Figure 7. The execution time of the clustering package.

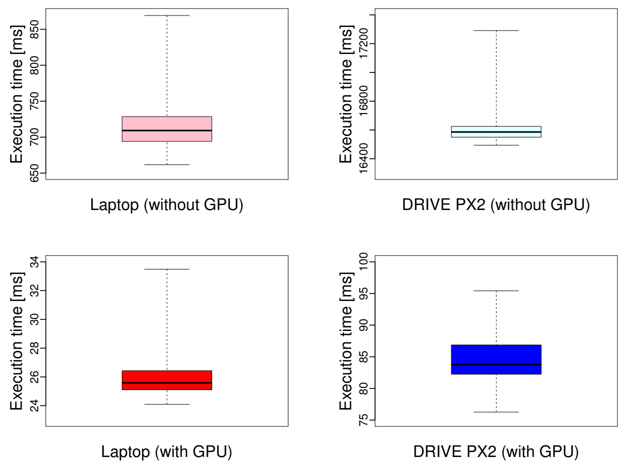


Figure 8. The execution time of the image detector package.

clip_max_height = 0.5.

We tested two cases: the clustering package was executed with and without using the GPU. Measurement results with the leaf size of 0.1, 0.2, and 0.3 meters are shown in Figure 7. On the DRIVE PX2 platform, the execution time decreased as the leaf size increased, similarly to the case for the laptop computer. Moreover, one can see that the presence of the GPU helped to reduce the execution time. In case that the laptop computer was used, the execution time was not influenced significantly by the presence of the GPU. We conjecture that the overhead for using the GPU was equivalent to the extra computation time required without the GPU. Therefore, in some cases, the GPU is not really desired to reduce the execution time. The parameters for the clustering package must also be configured appropriately, because the accuracy of clustering and the execution time has a trade-off relation.

Image detector: The image detector package is based on the SSD algorithm as described in Section 3.2.1. The measurement results with and without using the GPU are shown in Figure 7. Without using the GPU, the SSD algorithm required a longer execution time, meaning that the GPU is truly useful for this algorithm. The execution time on the DRIVE PX2 platform was approximately three times as much as that on the laptop computer. The SSD algorithm was executed using the 1152 cores on the DRIVE PX2 platform, whereas the laptop computer could use up to 2,048 cores. This indicates that the difference in the number of cores between the two platforms led to the difference in performance as well. Therefore, the execution time could be improved by distributing the computing workload across the TEGRA A and B processors for the DRIVE PX2 platform. The same is true of other state-of-the-art algorithms, such as Yolo2 [15]: the execution time on the DRIVE PX2 platform could be improved by load balancing.

4.2.2. End-to-end time. To measure the end-to-end time, the packages that are necessary for our autonomous vehicle were launched at the beginning. The execution times of a specific package set were measured and summed up. In these measurements, the leaf size of the points localizer was set to 2.0 meters, and the point cloud data extending up to 200 meters ahead was used. These conditions represent a similar workload model that would be required during a real self-driving scenario. As for measurements with the DRIVE PX2 platform, the image detector package and the clustering package were launched on TEGRA A, while the other packages were launched on TEGRA B to distribute the computational load.

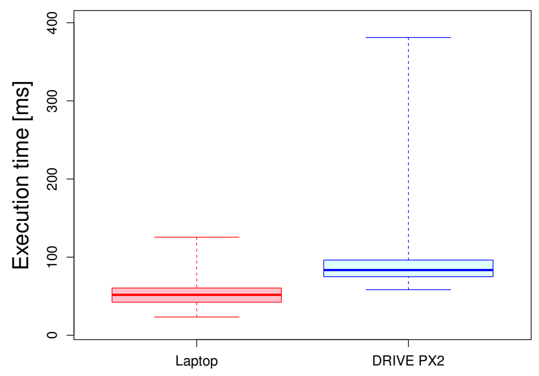


Figure 9. End-to-end time.

TABLE 3. CONFIGURATION OF EACH NODE.

Node	parameter	value
voxel_grid_filter	Voxel Leaf Size	2 m
	Measurement Range	200 m
ndt_matching	Resolution	1 m
lane_rule	Acceleration	1 m/s^2
lane_select	Distance Threshold to Neighbor Lanes	5 m
velocity_set	Deceleration	0.7 m/s^2
	Velocity Change Limit	10 km/h
waypoint_follower	Lookahead Ratio	2 m/s
	Minimum Lookahead Distance	6 m/s
twist_filter	Lateral Accel Limit	5 m/s^2

The measured end-to-end times are shown in Figure 9. The worst execution time observed on the DRIVE PX2 platform was about three times as much as that on the laptop computer. Our observation is that this difference is caused by the performance of CPU cores rather than the GPUs.

Based on the above results, we may conclude that the DRIVE PX2 platform provides the acceptable level of performance for Autoware installed to real autonomous vehicles, if the parameters of each functional node are configured appropriately. A demonstration movie of our autonomous vehicle powered by the DRIVE PX2 platform, with the configuration of Table 3, can be found at the following hyperlink: https://youtu.be/zujGfJcZCpQ.

5. Related Work

Self-driving technology has made a significant leap in both theory and practice since prototype autonomous vehicles shown off on the DARPA Urban Challenge projects [26], [31], [32], [33], [34]. Common features found on these prototype autonomous vehicles were 3D laser scanners widely used in addition to traditional cameras, radars, and GNSS/IMU. Today, many industrial companies aiming for urban-area self-driving have followed the achievement of the DARPA Urban Challenge projects [35], [36], [37], [38], [39]. Autoware is also expected to be deployed on autonomous vehicles in urban areas, rather than freeways and highways, and thus its design concept has been inspired by the DARPA Urban Challenge projects.

In terms of open-source software contributions, Apollo [40] and PolySync [41] can be compared to Autoware. Apollo was originated from the Baidu's alliance, it mainly depends on high-precision GNSS location and 3D perception. On the other hand, PolySync provides an integrated development environment for the backend framework of self-driving technology. These two projects are likely to consider a specific configuration for platforms and algorithms. Autoware differs in that it is designed and implemented for more general configurations, where users can choose platforms and algorithms according to their requirements. Autoware's general purposiveness and openness have allowed its integration, as the base for online education of self-driving technology presented by Udacity [38]. It also helped to produce autonomous vehicle platforms such as the one created by AutonomouStuff [42].

So far, none of the above alternative contributions have succeeded in revealing their technology with embedded systems. To the best of our knowledge, this paper is the first contribution that disclosed the complete software stack of autonomous vehicles on embedded systems, and quantified its performance using real hardware.

6. Conclusion

In this paper, we have presented *Autoware on Board*, a new profile of Autoware using embedded systems for autonomous vehicles. We clarified the integration feasibility of *Autoware on Board*, using the DRIVE PX2 platform. We also highlighted our implementation effort to accommodate with ARM-based embedded processing cores on the DRIVE PX2 platform, whereas only a few functional nodes of Autoware are offloaded to the Tegra-based GPUs. The experimental results demonstrated that the latency of an individual module of Autoware imposed on the DRIVE PX2 platform is maintained in the order of milliseconds. As compared to a high-end laptop computer, the latency was, at most, threefold.

The source code of *Autoware on Board* implemented through this paper has been merged into the master branch of the Autoware GitHub repository [1]. To the best of our knowledge, this is the first contribution that achieved a publicly-available embedded system profile for autonomous vehicles with quantified performance evaluation.

In future work, we plan to improve the design and implementation of Autoware for real-world production. We are also interested in the application of it to real-world services. To that end, this work can be optimized for further low-power and vehicular-grade platforms. Particular examples include the next generation of the DRIVE PX2 platform, which is expected to meet an order of TFLOPS performance under 30 W power, and an alternative solution without GPUs, called Kalray MPPA, which presumably achieves TFLOPS performance under 20 W power using many-core technology.

Acknowledgments

This work was partially supported by JST CREST Grant Number JPMJCR1683 and JST PRESTO Grant Number

JPMJPR1751, Japan.

References

[1] "Autoware: Open-source software for urban autonomous driving," https://github.com/CPFL/Autoware, accessed: 2018-02-14.

[2] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, "An Open Approach to Autonomous Vehicles," *IEEE Micro*, vol. 35, no. 6, pp. 60–68, 2015.

[3] "Point Cloud Library (PCL)," http://pointclouds.org/, accessed: 2018-02-14.

[4] "Compute Unified Device Architecture (CUDA)," https://developer.nvidia.com/cudazone, accessed: 2018-02-14.

[5] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[6] "Caffe," http://caffe.berkeleyvision.org/, accessed: 2018-02-14.

[7] "OpenCV," http://opencv.org/, accessed: 2018-02-14.

[8] Y. Maruyama, S. Kato, and T. Azumi, "Exploring Scalable Data Allocation and Parallel Computing on NoC-based Embedded Many Cores," in *Proc. of IEEE International Conference on Computer Design (ICCD)*, 2017.

[9] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *Proc. of IEEE International Conference on Robotics and Automation Workshop on Open Source Software*, vol. 3, no. 3.2, 2009, p. 5.

[10] "ROS.org," http://www.ros.org/, accessed: 2018-02-14.

[11] P. Biber and W. Straßer, "The normal distributions transform: A new approach to laser scan matching," in *Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 3, 2003, pp. 2743–2748.

[12] M. Magnusson, "The three-dimensional normal-distributions transform: an efficient representation for registration, surface analysis, and loop detection," Ph.D. dissertation, Örebro universitet, 2009.

[13] P. J. Besl and N. McKay, "A method for registration of 3-d shapes," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, pp. 239–256, 1992.

[14] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *Proc. of European Conference on Computer Vision (ECCV)*, 2016.

[15] J. Redmon and A. Farhadi, "YOLO9000: better, faster, stronger," *CoRR*, vol. abs/1612.08242, 2016, accessed: 2018-02-14. [Online]. Available: http://arxiv.org/abs/1612.08242

[16] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan, "Object detection with discriminatively trained part-based models," *IEEE transactions on pattern analysis and machine intelligence*, vol. 32, no. 9, pp. 1627–1645, 2010.

[17] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Proc. of IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 1, 2005, pp. 886–893.

[18] R. B. Rusu, "Semantic 3d object maps for everyday manipulation in human living environments," October 2009.

[19] R. E. Kalman *et al.*, "A new approach to linear filtering and prediction problems," *Journal of basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960.

[20] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking," *IEEE Transactions on signal processing*, vol. 50, no. 2, pp. 174–188, 2002.

[21] D. Dolgov, S. Thrun, M. Montemerlo, and J. Diebel, "Path planning for autonomous vehicles in unknown semi-structured environments," *The International Journal of Robotics Research*, vol. 29, no. 5, pp. 485–501, 2010.

[22] B. Nagy and A. Kelly, "Trajectory generation for car-like robots using cubic curvature polynomials," *Field and Service Robots*, vol. 11, 2001.

[23] H. Darweesh, E. Takeuchi, K. Takeda, Y. Ninomiya, A. Sujiwo, L. Y. Morales, N. Akai, T. Tomizawa, and S. Kato, "Open source integrated planner for autonomous navigation in highly dynamic environments," *Journal of Robotics and Mechatronics*, vol. 29, no. 4, pp. 668–684, 2017.

[24] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[25] M. McNaughton, C. Urmson, J. M. Dolan, and J.-W. Lee, "Motion planning for autonomous driving with a conformal spatiotemporal lattice," in *Proc. of IEEE International Conference on Robotics and Automation (ICRA)*, 2011, pp. 4889–4895.

[26] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer *et al.*, "Autonomous driving in urban environments: Boss and the Urban Challenge," *Journal of Field Robotics*, vol. 25, no. 8, pp. 425–466, 2008.

[27] M. Pivtoraiko, R. A. Knepper, and A. Kelly, "Differentially constrained mobile robot motion planning in state lattices," *Journal of Field Robotics*, vol. 26, no. 3, pp. 308–333, 2009.

[28] R. C. Coulter, "Implementation of the pure pursuit path tracking algorithm," CARNEGIE-MELLON UNIV PITTSBURGH PA ROBOTICS INST, Tech. Rep., 1992.

[29] "NVIDIA: DRIVE PX2," http://www.nvidia.com/object/drive-px.html, accessed: 2018-02-14.

[30] "NVIDIA: Parker," https://blogs.nvidia.com/blog/2016/08/22/parker-for-self-driving-cars, accessed: 2018-02-14.

[31] M. Montemerlo, J. Becker, S. Bhat, H. Dahlkamp, D. Dolgov, S. Ettinger, D. Haehnel, T. Hilden, G. Hoffmann, B. Huhnke *et al.*, "Junior: The stanford entry in the urban challenge," *Journal of field Robotics*, vol. 25, no. 9, pp. 569–597, 2008.

[32] J. Leonard, D. Barrett, J. How, S. Teller, M. Antone, S. Campbell, A. Epstein, G. Fiore, L. Fletcher, E. Frazzoli *et al.*, "Team mit urban challenge technical report," 2007.

[33] M. Buehler, K. Iagnemma, and S. Singh, *The DARPA urban challenge: autonomous vehicles in city traffic*. Springer, 2009, vol. 56, ch. The MIT - Cornell Collision and Why it Happened, pp. 509–548.

[34] M. Buehler, K. Iagnemma, and S. Singh, Eds., *The DARPA urban challenge: autonomous vehicles in city traffic*. Springer, 2009, vol. 56, ch. A Perception-Driven Autonomous Urban Vehicle, pp. 163–230.

[35] "Waymo," https://waymo.com/, accessed: 2018-02-14.

[36] "Toyota Research Institute," http://www.tri.global/toyota-research-institute-releases-video-guardian-and-chauffeur, accessed: 2018-02-14.

[37] "Uber," https://www.uber.com/cities/pittsburgh/self-driving-ubers/, accessed: 2018-02-14.

[38] "Udacity," https://www.udacity.com/self-driving-car, accessed: 2018-02-14.

[39] "Apollo," http://apollo.auto/, accessed: 2018-02-14.

[40] "Apollo: an open autonomous driving platform," https://github.com/ApolloAuto/apollo.

[41] "PolySync: tools and software infrastructure at the center of autonomous vehicle development," https://polysync.io/, accessed: 2018-02-14.

[42] "AutonomouStuff," https://autonomoustuff.com/, accessed: 2018-02-14.