**Regular Paper**

# Real-time Performance of Embedded Platform for Autonomous Driving Using Linux and a GPU

Shota Mizuno†, Shinya Honda‡*, Ryo Kurachi‡**, and Naoya Chujo***

†Graduate School, Aichi Institute of Technology, Japan
‡Graduate School, Nagoya University, Japan
***Aichi Institute of Technology, Japan
{†b16727bb,***ny-chujo}@aitech.ac.jp
*honda@ertl.jp
**kurachi@nces.is.nagoya-u.ac.jp

***Abstract*** - Autonomous driving systems and advanced driver assistance systems (ADAS) have been developed to improve driving safety. These systems are becoming increasingly complicated, and the computing power they require continues to increase. Software platforms, such as Automotive Grade Linux, are being developed to handle the complexity of autonomous driving, and graphical processing units (GPUs) are being used in these systems for tasks such as recognition of driving environments, deep learning, and localization. However, to the best of our knowledge, the real-time performance of Linux combined with a GPU for ADAS has not yet been reported.

In this study, we developed lane keeping and collision avoidance systems to evaluate the real-time performance of Linux and a GPU. The experimental results show that the real-time performance of the system could be improved using widely available versions of Linux without software customization. Also, although the GPU execution speed was sufficiently high, camera image capture was relatively slow and created a bottleneck in the system.

***Keywords***: Autonomous driving, Advanced driver assistance systems, Linux, Real-time performance, GPU

## 1 INTRODUCTION

In recent years, autonomous driving systems and advanced driver assistance systems (ADAS) have been developed to improve driving safety [1], [2]. In fact, the first cars equipped with autonomous driving level 3, as defined by the Society of Automotive Engineers [3], is scheduled to be released in 2018 [4]; this level allows the vehicle to assume control of safety-critical functions. In both autonomous driving systems and ADAS, a large number of sensors, such as cameras, radar, light detection sensors, and ultrasonic sensors, are used to recognize various aspects of the driving environment [5]. The driving system must be capable of simultaneously processing a large amount of information from these sensors in real time in order to operate at highway speeds and respond to the environment on any roadway. For this reason, two challenges are common to these systems: complexity of the software and demand for computing power.

Regarding software complexity, standard software platforms are already being developed. For instance, Automotive Grade Linux [6] is being developed by many automakers and suppliers for in-vehicle infotainment systems. The abundant device drivers, application programming interfaces, and libraries within Linux are expected to reduce the development cycle and cost of new systems. Furthermore, there have been efforts to improve the real-time performance of Linux using RTLinux [7], which is a Linux kernel augmented by a dedicated patch. Here, real-time is defined as a time constraint for processing by a specified deadline, and it is the most important property required for in-vehicle systems.

To increase the computational speed of ADAS hardware, GPUs are increasingly being adopted. GPUs are necessary for various autonomous driving system tasks, such as recognition of vehicles and pedestrians using advanced image processing, deep learning, and localization. The development of semiconductor technology has dramatically accelerated the advancement of GPUs and given them a massively parallel architecture. In particular, GPUs seem to be essential for improving the recognition performance of ADAS using deep learning.

However, to the best of our knowledge, evaluation of the real-time performance of Linux and GPUs for autonomous driving systems has not yet been reported, despite its importance. In our study, two autonomous driving platforms were considered [8]. One platform was for evaluation of the real-time performance of an autonomous system using Linux, and the other platform was used to evaluate the performance of Linux with a GPU. We examined the factors affecting the real-time performance of systems developed on these platforms. In addition, the real-time processing capabilities of the embedded GPU were verified for the second platform. By evaluating these platforms, we aim to obtain guidelines for implementing software for autonomous driving systems and ADAS. Another objective is to evaluate the potential of embedded GPUs and understand processing bottlenecks.

This paper is organized as follows. Section 2 describes the two platforms based on the general architecture for autonomous driving systems, describes the elements comprising these platforms, and explains the systems used for evaluation. Section 3 discusses experiments conducted to evaluate the platform using Linux and a real-time operating system (RTOS) and determine its real-time performance. In Section 4, we present experiments using a platform based on Linux,

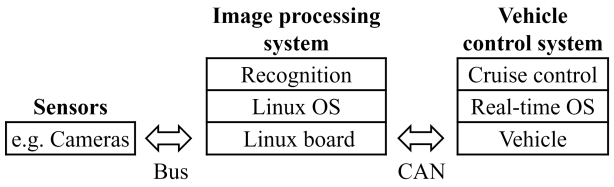



**Figure 1:** Minimum architecture for autonomous driving system

the RTOS, and a GPU. Experimental considerations are discussed in Section 5, and Section 6 provides conclusions.

## 2 PLATFORMS FOR AUTONOMOUS DRIVING

In this section, the evaluation platforms for autonomous driving are explained.

### 2.1 Minimum Architecture for Autonomous Driving

Autonomous driving systems and ADAS normally contain a number of different sensors, most commonly onboard cameras, which are indispensable for recognizing lanes, pedestrians, and signs. Use of open-source libraries enables reduction of cost and development cycle, and for image processing, function libraries such as OpenCV [9] are available for Linux. Therefore, we decided to use Linux for image processing in both platforms.

Figure 1 shows the minimum architecture for an autonomous driving system, which consists of three parts, an onboard camera, an image processing system, and a vehicle control system. On the basis of this architecture, two platforms were developed for this study. The features of these two platforms are given in Table 1.

Both platforms A and B use Linux and an RTOS, but only Platform B is equipped with a GPU. In Platform A, relatively lightweight processing is performed, whereas in Platform B, advanced image processing requiring substantial computing power is performed.

### 2.2 RoboCar®1/10 for AP

RoboCar®1/10 for automotive platform (AP) (hereafter "RoboCar") [10], shown in Fig. 2, is a 1:10 scale miniature version of an actual car. It has a Renesas V850 central processing unit (CPU) with a TOPPERS/ATK2 OS [11] RTOS, which was developed for automotive systems. In this study, RoboCar provided the base platform, and its CPU was used for the vehicle control system. The basic configuration of RoboCar is given in Tab. 2.

### 2.3 Platform A Using Linux

This section introduces the platform using Linux. The architecture developed for Platform A is shown in Fig. 3.

A Raspberry Pi 2 Model B was adopted as the image processing system. This system consists of a small single-board computer that runs a Linux-based OS and has a quad-core Arm Cortex-A7 processor and 1 GB of memory (DDR-SDR-

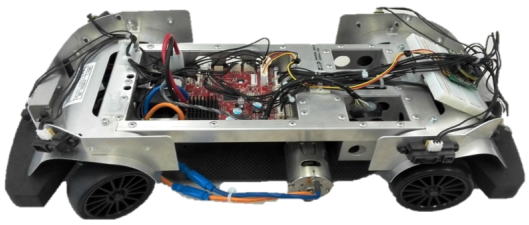


**Figure 2:** RoboCar®1/10 for AP

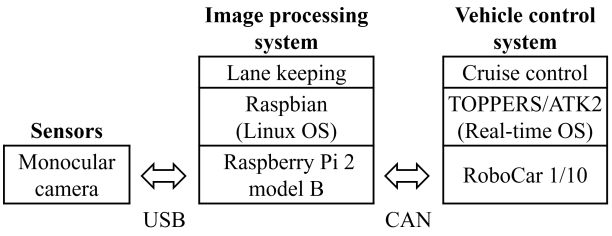



**Figure 3:** Architecture of Platform A

AM). The Raspberry Pi and Robocar were connected via a controller area network (CAN). A monocular camera (iBUFFALO BSW- 20KM11) with a resolution of 640 × 480 and a maximum frame rate of 30 frames per second (fps) was connected to the image processing system with a universal serial bus (USB). Figure 4 is a photograph of Platform A.

### 2.4 Platform B Using Linux with GPU

This section introduces the platform equipped with a GPU running Linux. The architecture of this platform is shown in Fig. 5.

In this platform, an NVIDIA Jetson TX1 module [12] was used for image processing to execute more demanding stereographic image processing. The Jetson TX1 is a card-sized board with an NVIDIA Tegra X1 chip running at 10 W. The chip has 256 CUDA cores for the GPU and a quad-core Arm Cortex-A57 CPU with a 2-MB L2 cache and a 4-GB random access memory (LPDDR4). The Jetson TX1 OS is Linux for Tegra (L4T) R23.2 provided by NVIDIA. Figure 6 shows the Jetson TX1 module, and Fig. 7 is a photograph of Platform B.

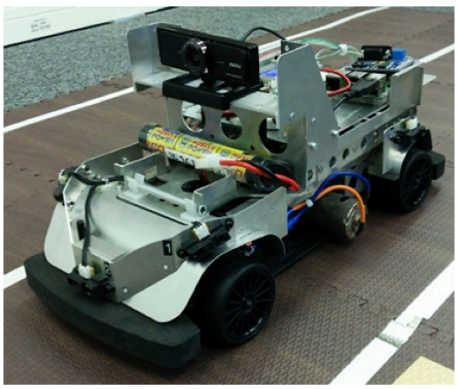


**Figure 4:** Platform A: RoboCar with monocular camera and Raspberry Pi

**Table 1:** Features of the two platforms

| System | Platform A | Platform B |
|---|---|---|
| **Sensors** | iBUFFALO BSW20KM11<br>monocular camera | Stereolabs ZED<br>stereo camera |
| **Image processing** | Raspberry Pi 2 Model B<br>CPU: 4× Arm Cortex-A7<br>GPU: none | NVIDIA Jetson TX1<br>CPU: 4× Arm Cortex-A57<br>GPU: 256× CUDA core |
| **Vehicle control** | ZMP RoboCar®1/10 for AP | |

**Table 2:** Basic configuration of RoboCar

| | |
|---|---|
| **Dimensions** | 429×195×212 mm |
| **Weight** | 1.8 kg |
| **Internal sensors** | Gyro<br>Accelerometer<br>5× Rotary encoder |
| **External sensors** | 8× Infrared range-finding sensor |
| **Battery** | 7.2-V NiMH battery |
| **CPU** | Renesas V850 |
| **OS** | TOPPERS/ATK2 |

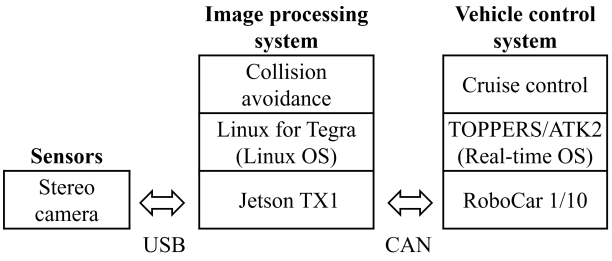



**Figure 5:** Architecture of Platform B

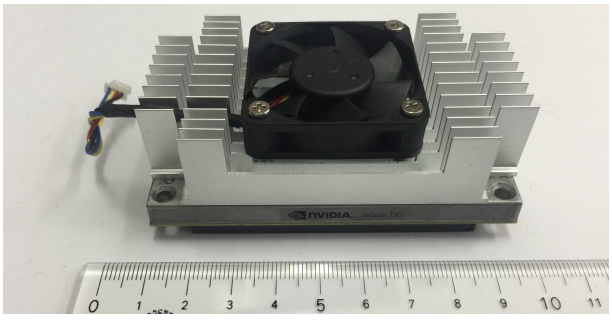


**Figure 6:** NVIDIA Jetson TX1 (with centimeter scale)

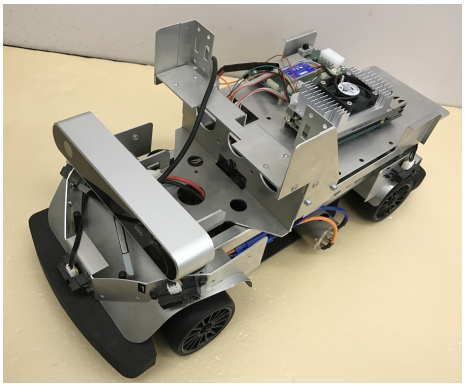


**Figure 7:** Platform B: RoboCar with stereo camera and Jetson TX1 GPU

**Table 3:** Key features of ZED Stereo Camera

| | | |
|---|---|---|
| **Dimensions** | | 175×30×33 mm |
| **Lens** | **Field of view** | 110° |
| | **F-number** | 2.0 |
| **Depth** | **Depth range** | 0.5 20 m |
| | **Stereo baseline** | 120 mm |
| **Resolution** | **HD1080** | 1920×1080 / 30Hz |
| | **HD720** | 1280×720 / 60 Hz |
| | **WVGA** | 672×376 / 100Hz |

In this setup, the Jetson TX1 module and RoboCar were connected via CAN using an Auvidea J120 carrier board [13]. A USB 3.0 Stereolabs ZED stereo camera was connected to the Jetson TX1 board; this camera obtains images useful for both image processing tasks evaluated in this study. Key features of ZED stereo camera are shown in Fig. 3.

Three of this camera's image resolutions were used in this study: wide video graphics array (WVGA) (672 × 376), HD7-20 (1,280 × 720), and HD1080 (1,920 × 1,080).

## 2.5 CUDA

The Compute Unified Device Architecture (CUDA) is a parallel programming language and model developed by NVIDIA [14], [15] that is available in the Jetson TX1 module. CUDA is supported by OpenCV, a library for computer vision functions that was used to develop the image processing program for both platforms.

In CUDA, the TX1 CPU is treated as the host and its GPU as the device. The term "kernel" refers to functions that run on the device, and a kernel consists of numerous "threads." All threads run the same code, and in image processing, one

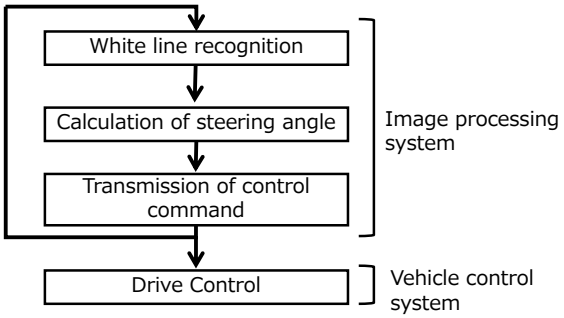



**Figure 8:** Flow diagram of lane keeping system

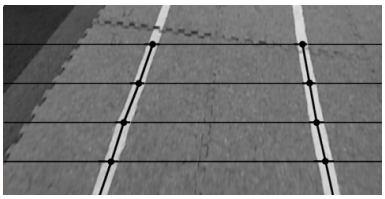


**Figure 9:** Example of white line recognition

thread corresponds to one pixel. First, a program executed on the host side is activated. Next, the device loads the kernel program, and the host passes the generated data to the device and "kicks" it (to instruct the host to start the kernel). The device executes the kernel and asynchronously returns the obtained result to the host. The processing on the GPU side can be synchronized, but in this study, unless otherwise noted, processing on the GPU side was executed asynchronously.

## 2.6 Lane Keeping System

Lane keeping [16] is a basic driving support system, and its configuration is simple. A lane keeping system consisting of white line recognition and steering control was implemented in Platform A to verify the real-time performance of Linux. Figure 8 is a flow diagram of the lane keeping system, and Fig. 9 shows an example of white line recognition.

The camera image is used by the image processing system to recognize white lines. Then, the image processing system uses the recognition result to calculate the steering angle required to run along the white line and transmits it to the vehicle control system.

In this study, the target time for white line recognition was set to 30 ms, which is generally regarded as real-time performance.

## 2.7 Collision Avoidance System

Collision avoidance systems are being adopted by many manufacturers. The sensors used in collision avoidance systems include millimeter-wave and infrared radar sensors and monocular and stereo cameras. We adopted collision avoidance as the task for evaluating Platform B using the GPU because collision avoidance requires more computing power, in comparison to that needed for lane keeping. Figure 10 is a flow diagram of the collision avoidance system.

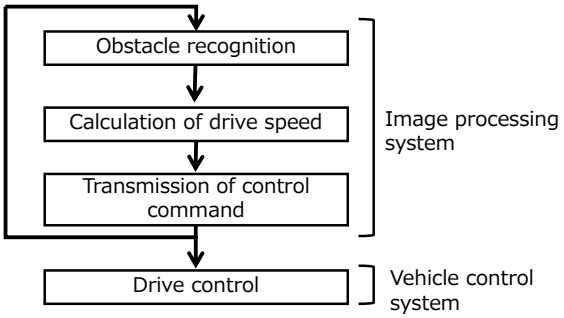
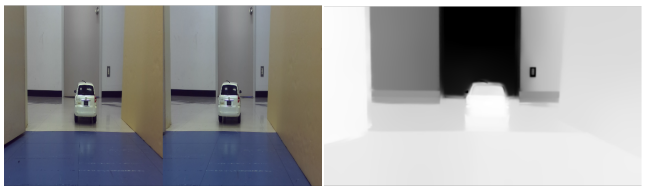



**Figure 10:** Flow diagram of collision avoidance system



**(a)** Stereo image **(b)** Depth map

**Figure 11:** Example of obstacle recognition with a stereo camera

In this system, the GPU generates a depth map from the stereo image and calculates the driving speed, which is transmitted to the vehicle control system via the CAN. When an obstacle is recognized in the depth map, the vehicle is stopped to avoid collision. An example of obstacle recognition is shown in Fig. 11, where the depth map expresses nearer as brighter.

It is said that "assuming car speed of 80 Km/Hr, the stopping distance reduces from 55m to 45m as frame-rate goes from 10fps to 15fps (for systems as of today) to 30 fps (for system in future)." [17]. In this study, the target time of collision avoidance system was set to 33 ms which is calculated from 30 fps.

# 3 PLATFORM A: REAL-TIME PERFORMANCE WITH LINUX

In this section, we report the evaluation of the real-time performance of Linux used for lane keeping with Platform A.

## 3.1 Lane Keeping System on Platform A

The iBUFFALO BSW20KM11 monocular camera used for sensing in Platform A was attached to the top of the RoboCar and output VGA ($640 \times 480$) images at 30 fps. One cycle of image processing and response is shown in Fig. 12.

In the experiments, Platform A navigated a round course created using a tile mat and white tape. The lane keeping processing time was measured for each cycle, which started with image capture and ended with transmission of steering

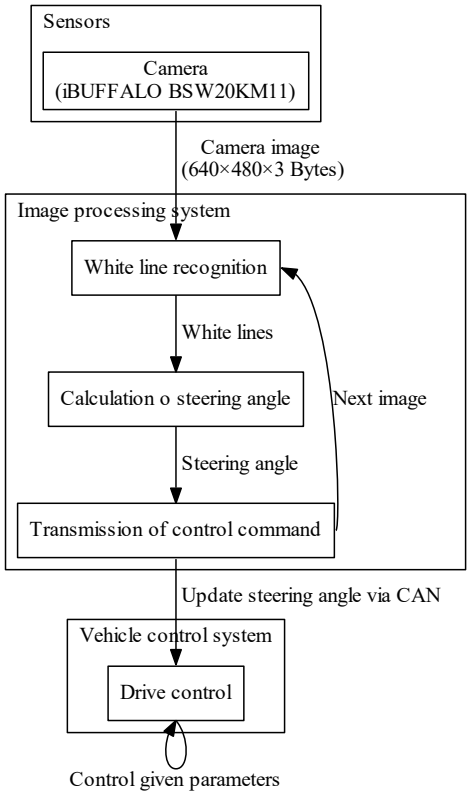



**Figure 12:** Process flow in lane keeping system on Platform A

instructions via CAN. The clock rate of the Raspberry Pi was set to 900 MHz.

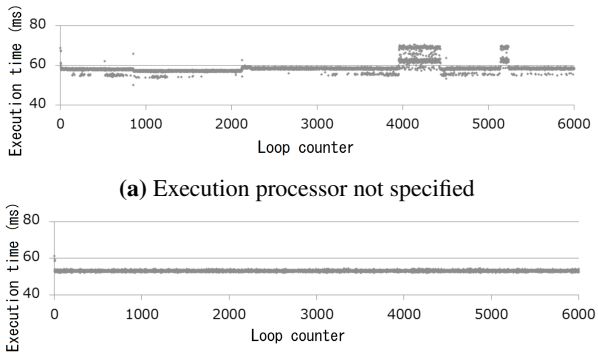
## 3.2 Processor Affinity for CPU Selection

Processor affinity is a property used to specify which CPU executes each process. In Linux, the OS normally automatically assigns an execution processor based on this property and the priority of the process. If the process can be executed by another processor, the execution may sometimes be reassigned, which delays the process while the cache is copied.

In this study, the affinity was set to execute the lane keeping program on core 2 of the Raspberry Pi using the taskset command in Linux. The execution times with and without the specification of the execution processor were measured for 10,000 frames. The results are shown in Fig. 13.

As shown in Fig. 13a, when the execution processor was not specified, lane keeping was typically executed within 60 ms, although it sometimes increased to approximately 70 ms. We investigated the cause and found that the occasional delay was caused by neither dynamic clock changes, a temperature rise, nor the software algorithm. With these factors ruled out, we concluded that the delay was caused by the reassignment of the execution processor. As shown in Fig. 13b, this delay is suppressed when the execution processor is specified.

The average execution time and worst-case execution time (WCET) are given in Tab. 4. The average execution time




**(a)** Execution processor not specified




**(b)** Execution processor specified

**Figure 13:** Execution time with and without specification of the execution processor

**Table 4:** Average execution time and WCET with and without specification of execution processor

| **Execution time** | Not specified | Specified |
|---|---|---|
| **Average** | 59.4 ms | 53.1 ms |
| **WCET** | 78.1 ms | 61.3 ms |

and the WCET were both lower when the execution processor was specified than when it was not specified, demonstrating that the WCET can be improved by specifying the execution processor.

During normal processing in a Linux OS, a timer interrupt occurs at regular intervals. When the interrupt occurs, the scheduler examines the process state and determines the priority of the processes to be executed. Because the priority of a process that is using a CPU for a long time decreases with time, the timer interrupt enables the execution of other waiting processes. With this mechanism, Linux maintains the even execution of processes.

When lane keeping was executed without options, the priority for this process is the same as that for other processes. Therefore, lane keeping execution may occasionally be postponed by the scheduler. In addition, if the execution time of one cycle of the process is long, the scheduler is activated during the execution, and other higher-priority processes are executed. Then, the original process is delayed until it becomes executable again. This is thought to be the reason for the occasional execution delay when no processor is specified in Linux.

## 3.3 Real-time Process and RTLinux

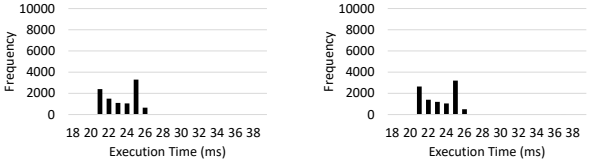
Next, we enhanced Linux for real-time performance and evaluated the results. RTLinux was developed to improve the real-time performance of Linux. In the real-time process, the execution of the processor is given higher priority.

RTLinux is obtained by augmenting the Linux kernel by applying the RT-Preempt patch [18] provided by the Linux community. Various other patches exist for this purpose, including Xenomai and real-time application interface [19], but these were not used in this study.

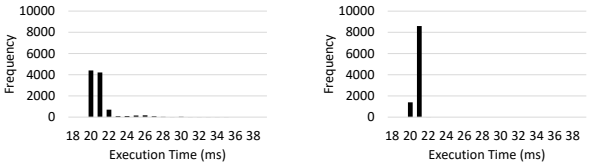
RTLinux was implemented on Platform A, the lane keeping experiment was repeated, and the normal and real-time exe-

**Table 5:** Kernel and process conditions for each real-time (RT) case

| Application | a | b | c | d |
|---|---|---|---|---|
| **Kernel** | Normal | Normal | RT | RT |
| **Process** | Normal | RT | Normal | RT |




**(a)** Linux, normal process **(b)** Linux, real-time process




**(c)** RTLinux, normal process **(d)** RTLinux, real-time process

**Figure 14:** Histograms of lane keeping execution times for four real-time cases

cution times were compared. When using RTLinux, the time required for the camera to capture the image was abnormally long. This was likely because the device driver was updated when the RT- Preempt patch was applied. Therefore, the time required for image capture was not included in the execution time. Experiments were conducted for four cases with each possible combination of real-time or normal kernels and real-time or normal processes. The conditions for the four cases are given in Tab. 5.

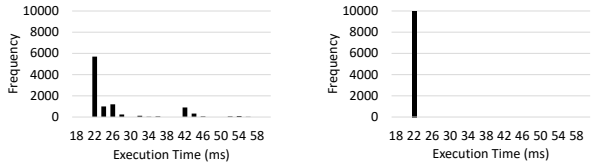
Histograms of the lane keeping execution times obtained for all four cases are shown in Fig. 14, and the corresponding average execution times and WCETs are given in Tab. 6.

When the program was executed as a normal process on RTL- inux, the execution time was unstable in comparison with the normal process executed on the normal Linux kernel. However, when it was executed as a real-time process on RTLinux, the execution time was stable, and the delay was greatly suppressed because the delay for dispatching the process is decreased by RTLinux.

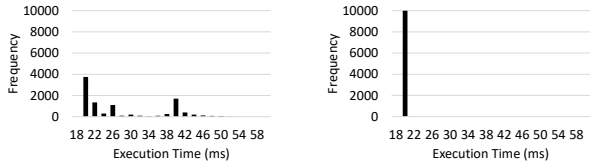
Applying the RT-Preempt patch reduced the time to start the scheduler. The real-time performance of the lane keeping program deteriorated in the kernel with the RT-Preempt patch applied without options. When the process was executed as a normal process, the execution state is thought to switch more frequently. When the process was executed as a

**Table 6:** Average execution times and WCETs for four lane keeping real-time cases

| Execution time | a | b | c | d |
|---|---|---|---|---|
| **Average** | 26.8 ms | 20.5 ms | 20.5 ms | 20.3 ms |
| **WCET** | 40.5 ms | 21.5 ms | 52.2 ms | 21. 7ms |




**(a)** Linux, normal process **(b)** Linux, real-time process




**(c)** RTLinux, normal process **(d)** RTLinux, real-time process

**Figure 15:** Histograms of lane keeping execution time for four real-time cases with CPU load

**Table 7:** Average execution times and WCETs for four lane keeping real-time cases with CPU load

| Execution time | a | b | c | d |
|---|---|---|---|---|
| **Average** | 25.9 ms | 20.4 ms | 27.4 ms | 19.1 ms |
| **WCET** | 67.4 ms | 21.4 ms | 57.3 ms | 20.6 ms |

real-time process on RTLinux, it was preferentially executed by one of the cores, even if the execution core was not specified. The real-time performance was also improved in normal Linux by executing it as a real-time process.

## 3.4 Real-time Process and RT Linux with CPU Load

In the experiments discussed so far, only a single user process was executed. However, in an actual system, multiple processes are executed simultaneously. Therefore, the delay in executing processes was examined while applying a background processing load. For the same four cases listed in Tab. 5, a CPU load was applied using the stress command and processing was performed for 10,000 periods. Histograms of the lane keeping execution times are shown in Fig. 15 for the four real-time cases with CPU load, and the corresponding average execution times and WCETs are given in Tab. 7.

According to Figs. 15a and 15c, the fluctuation of the execution cycle was larger in both kernels when the process was executed as a normal process. As shown in Tab. 7, the average execution time for the lane keeping program executed as a normal process in the normal Linux kernel (case a) was 25.9 ms, and the WCET was 67.4 ms. In this case, the variation in the execution cycle was not evenly distributed but was concentrated in two ranges: 22–26 ms and 42–44 ms. In contrast, when executed as a real-time process in the normal Linux kernel (case b), the average execution time and WCET were 20.4 ms and 21.4 ms, respectively. Thus, executing the program as a real-time process stabilized the execution time.

When a CPU load was applied and the program was exe-

cuted as a normal process, the execution cycle greatly fluctuated in both kernels. However, when it was run as a real-time process with an applied CPU load, the execution time was stabilized for both kernels. The execution time was stabilized by assigning the process a high priority; however, since an actual system contains multiple high-priority processes, it is a matter which process is prioritized.

# 4 PLATFORM B: REAL-TIME PERFORMANCE USING GPU WITH LINUX

This section discusses the evaluation of the real-time performance with Platform B using Linux and a GPU.

## 4.1 Lane Keeping System on Platform B

Image processing for the lane keeping program was implemented on the Jetson TX1 module in place of the Raspberry Pi; however, lane keeping was found to be too lightweight to use the GPU processor. The execution time for the lane keeping program was 5.4 ms (excluding image capture time) when using the TX1 CPU, and the time increased to 8.1 ms when using the TX1 GPU. Data transfer between the CPU and GPU occupied 1.7 ms.

Because lane keeping is a lightweight process and the overhead of the data transfer is larger than the execution time reduction achieved by using the GPU, the TX1 CPU executed the lane keeping task on Platform B.

## 4.2 Collision Avoidance System on Platform B

A stereo camera was adopted for the collision avoidance system because it is the sensor that is best able to both detect the lane and identify obstacles. The collision avoidance system configuration using the stereo camera and Jetson TX1 module is shown in Fig. 16.

When capturing WVGA image, the image data is captured by processor of image processing system. The depth map (672×376 ×1 Bytes) is generated and used on GPU to calculate the drive speed of RoboCar. Finally, drive speed is send to vehicle control system via CAN.

In this experiment, the RoboCar was located on a miniature straight course. By bringing an obstacle closer, we confirmed that the collision avoidance system was working. The image processing time for collision avoidance was measured for each cycle, which began with image capture and ended with transmission of instructions via the CAN. The Jetson TX1 clock rate of was 1.9 GHz. The experiment was performed with three different resolutions.

## 4.3 Dependence on Image Resolution

Simple pedestrian detection has already been put to practical use, but high-resolution images will be necessary to improve the detection distance and estimation of pedestrian movement direction. When 8-mega-pixel images are used, it is possible to detect pedestrians at distances of up to 200 m [17].

Figure 17 shows the processing speeds for obstacle recognition experiments conducted using stereo images at three

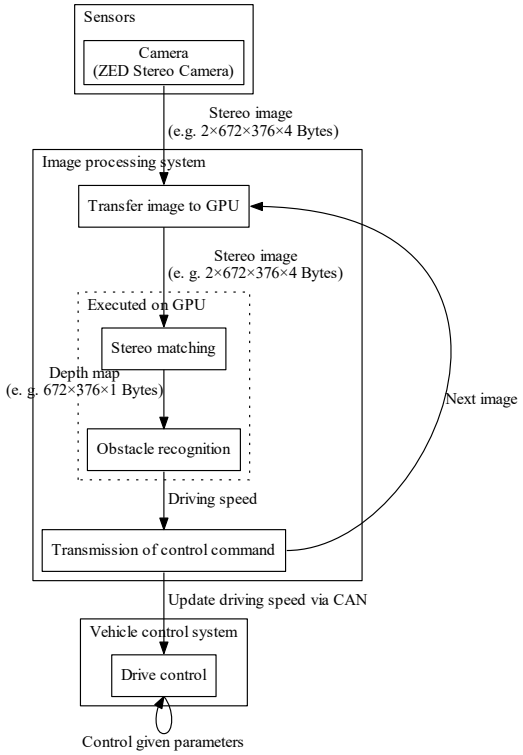



**Figure 16:** Process flow in collision avoidance system on Platform B

**Table 8:** Resolution-dependent stereo image sizes and processing speeds for WVGA

| Resolution | Image size | Processing speed |
|---|---|---|
| WVGA | 1.0× | 1.0× |
| HD720 | 3.7× | 2.2× |
| HD1080 | 8.3× | 4.0× |

resolutions. The processing speeds for WVGA, HD720, and HD1080 were 68, 31, and 17 fps, respectively. Table 8 gives the processing speeds for the three resolutions relative to that of WVGA.

Although the image sizes at resolutions of HD720 and HD-1080 were, respectively, 3.7 and 8.3 times that of WVGA, the processing speeds were only 2.2 and 4.0 times that of WVGA, respectively. This shows that the processing parallelization in the GPU was utilized. However, most of this execution time was spent on capturing the camera image. Figure 18 shows the breakdown of the synchronously measured execution time of the collision avoidance system.

In the case of WVGA, the CPU execution time for one frame was approximately 15 ms, and approximately 10 ms of that time was spent on image capture. The GPU completed processing in 8 ms, and this processing time did not increase much at higher resolutions. Thus, processing in the TX1 CPU, particularly with regard to the capture of camera images, is a performance bottleneck.

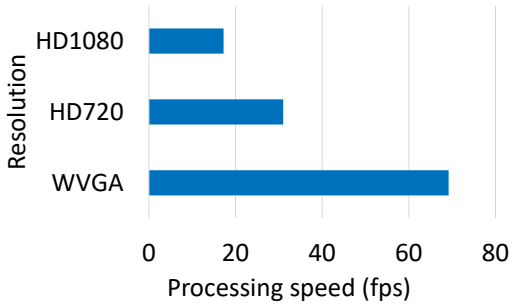



**Figure 17:** Comparison of stereo image processing speeds at three different resolutions

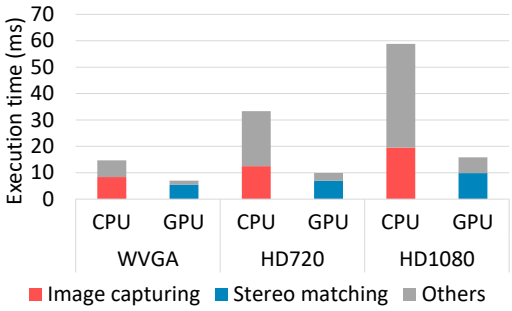
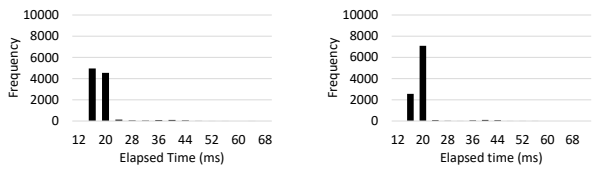



**Figure 18:** Execution time breakdown for collision avoidance system

## 4.4 Processor Affinity for CPU Selection

Experiments were also conducted to determine the processor affinity for Platform B. The recognition process was fixed to core 2 of the Jetson TX1 board, and the program was run as both with and without processor specification. The elapsed time per processing cycle was measured since processing on GPU performed asynchronously. The resulting elapsed time histograms are shown in Fig. 19.

The elapsed times were concentrated near 16 and 20 ms for both the normal and real-time processes. In addition, when the collision avoidance program was executed as a real-time process, the elapsed time increased to nearly 60 ms in approximately 0.02% of the frames. This is because the program used in the experiment consisted of several subprocesses and sometimes other subprocesses were prioritized, causing execution to be postponed.




**(a)** Execution processor not specified

**(b)** Execution processor specified

**Figure 19:** Histogram of the collision avoidance elapsed time with and without specification of the execution processor

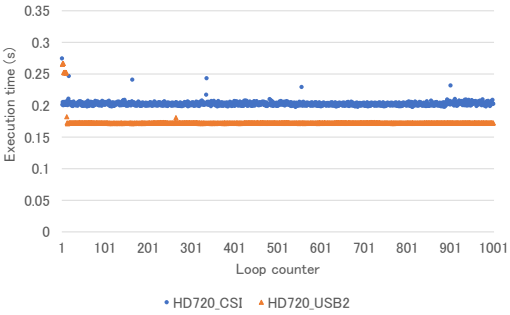



**Figure 20:** Comparison of capture execution time for USB and CSI camera interfaces

## 4.5 Camera Interfaces

In the experiments so far, cameras were connected via USB, and these cameras were the bottleneck in the system. Therefore in this section, we compared the USB camera interface and camera serial interface (CSI) [20].CSI is a camera/processor interface that is commonly used in embedded systems.

We measured and compared the time for capturing ZED stereo camera images between the USB and CSI camera interfaces with the Raspberry Pi 2. Figure 20 shows the execution time for capturing HD720 images for each interface.

Average execution times using the CSI and USB interfaces were 0.2 s and 0.17 s, respectively. Thus, in this experiment, use of the CSI interface did not speed up image capture.

# 5 CONSIDERATIONS

The experimental results obtained in this study show that the real-time performance of autonomous driving software programs implemented in Linux can be improved by specifying the execution core without optimizing the programs. Executing programs as real-time processes was also effective in both the normal Linux and RTLinux kernels. Furthermore, by applying the RT patch to the Linux kernel, it was possible to stabilize the execution time.

By executing the top-priority process as a real-time process, real-time performance can be secured even in Linux, which is a general-purpose OS. Therefore, it is expected that the wealth of software assets developed in Linux can be used, which leads to a reduction of development cycle time and cost.

It was verified that these methods are effective when executing one process, but an actual system will likely host multiple high-priority processes. Therefore, it is necessary to design the priorities of processes and system calls and verify the system performance with sufficient test vectors.

With regard to the GPU hardware used for image processing system in these experiments, the operation executed on the GPU side was sufficiently fast and the real-time performance was considered to be satisfactory. As GPU technology continues to develop, the image resolution and frame rate obtained using a GPU are expected to improve. However, the transfer time between the CPU and GPU on the TX-1 board and the image capture process remain as bottlenecks. Without an improved high-speed transfer channel and a high-speed interface with the camera, it is difficult to take advantage of the GPU's potential.

For comparatively lightweight computations, such as white line recognition, processing can be performed using only a CPU as in Platform A. The GPU usage method adopted for Platform B makes it possible to perform basic arithmetic operations with only the CPU.

## 6 CONCLUSION

In this study, we developed two platforms for autonomous driving and evaluated the real-time performance of image processing for lane keeping and collision avoidance systems using Linux. We found that allowing Linux to assign the processor decreased the real-time performance of both systems. Using the Linux taskset command to assign execution to a specific processor avoided the delay caused by the cache copying required for reassignment. We also found that executing the process as a real-time process on RTLinux can stabilize the execution time even when other processes are running. Additionally, we found that an embedded GPU provided process execution at sufficiently high speeds, but the overall speed was constrained by the speed of camera image capture, which was a major performance bottleneck in the system.

Future work will include considering a more complex system, such as a robot operating system, to verify the performance of a practical autonomous driving system using the advanced features evaluated here.

## ACKNOWLEGMENTS

The authors thank Zenzo Furuta (Graduate school, Aichi Institute of Technology, until May 2017) for experimental support.

## REFERENCES

[1] C. Urmson, et al. "Autonomous driving in urban environments: Boss and the urban challenge," Journal of Field Robotics Vol. 25, No. 8, pp. 425-466, (2008).

[2] A. Lindgren and F. Chen, "State of the art analysis: An overview of advanced driver assistance systems (adas) and possible human factors issues," Human factors and economics aspects on safety pp. 38-50, (2006).

[3] SAE International, "AUTOMATED DRIVING", http://www.sae.org/misc/pdfs/automated_driving.pdf (referred October 10, 2017).

[4] Audi newsroom, "Audi piloted driving", https://media.audiusa.com/models/piloted-driving (referred October 10, 2017).

[5] J. Levinson, et al. "Towards fully autonomous driving: Systems and algorithms," Intelligent Vehicles Symposium (IV), 2011 IEEE, pp. 163-168, (2011).

[6] Linux Foundation, "AUTOMOTIVE GRADE LINUX", https://www.automotivelinux.org/ (referred May 27, 2017).

[7] Linux Foundation, "Real-Time Linux", https://wiki.linuxfoundation.org/realtime/start.html (referred May 27, 2017).

[8] S. Mizuno, S. Honda, R. Kurachi, D. Hayakawa and N. Chujo, "Evaluation of Embedded Platform for Autonomous Drive using Linux and GPU," Proceedings of International Workshop on Informatics, pp. 53-58, (2017).

[9] G. Bradski and Adrian Kaehler, "Learning OpenCV: Computer vision with the OpenCV library," O'Reilly Media, Inc., (2008).

[10] ZMP Inc, "1/10 scale RoboCar(r) 1/10", http://www.zmp.co.jp/products/robocar-110 (referred May 27, 2017).

[11] TOPPERS Project, "TOPPES/ATK2", https://www.toppers.jp/atk2.html (referred May 27, 2017).

[12] NVIDIA, "Embedded Systems", http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html (referred October 10, 2017).

[13] AUVIDEA, "j120", https://auvidea.com/j120/ (referred October 10, 2017).

[14] NVIDIA, "CUDA Zone", https://developer.nvidia.com/cuda-zone (referred May 27, 2017).

[15] D. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture," ISMM, Vol.7, pp.103-104, (2007).

[16] R. Bosch, "BOSCH Automotive Handbook," the 8th Edtion, (2011).

[17] EETimes, "ADAS Front Camera: Demystifying Resolution and Frame-Rate", http://www.eetimes.com/author.asp?section_id=36&doc_id=1329109 (referred May 27, 2017).

[18] S. Arthur, C. Emde, and N. Mc Guire, "Assessment of the realtime preemption patches (RT-Preempt) and their impact on the general purpose performance of the system," Proceedings of the 9th Real-Time Linux Workshop, (2007).

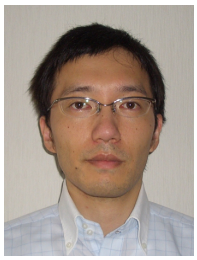
[19] A. Barbalace, et al. "Performance comparison of VxWorks, Linux, RTAI, and Xenomai in a hard real-time application", IEEE Transactions on Nuclear Science Vol. 55, No. 1, pp. 435-439, (2008).

[20] MIPI Alliance, "Camera and Imaging", https://mipi.org/specifications/camera-and-imaging (referred October 10, 2017).

(Received October 20, 2017)
(Revised December 28, 2017)

**Shota Mizuno** received the B.I. degree in Information Science from Aichi Institute of Technology, Japan in 2016. He is now a master course student of Aichi Institute of Technology, Japan. His research interest includes embedded system and automotive electronics.

**Shinya Honda** received his Ph.D. degree in the Department of Electronic and Information Engineering, Toyohashi University of Technology in 2005.From 2004 to 2006, he was a researcher at the Nagoya University Extension Course for Embedded Software Specialists.In 2006, he joined the Center for Embedded Computing Systems, Nagoya University, as an assistant professor, where he is now an associate professor.His research interests include system-level design automation and real-time operating systems.He received the best paper award from IPSJ in 2003. He is a member of ACM, IEEE, IPSJ, IEICE, and JSSST.



**Ryo Kurachi** is a Designated Associate Professor at Center for Embedded Computing Systems, Nagoya University. He graduated from Tokyo University of Science with undergraduate majors in applied electronics. After a few years working at AISIN AW CO., LTD. as a software engineer, he received his master's degree from Tokyo University of Science in 2007, followed by his Ph.D. in information science from the Nagoya University in 2012. His research interests includes embedded systems and real-time systems. Within that domain, he has investigated topics such as in-vehicle networks and real-time scheduling theory and embedded systems security. He is a member of IPSJ and IEICE.

**Naoya Chujo** received his B.E. degree in applied physics and his M.S. degree in information science and his Ph.D. degree in electrical engineering from Nagoya University in 1980, 1982 and 2004. He joined Toyota Central R&D Labs. in 1982. Since 2010, He is a professor at Aichi Institute of Technology. His research interests are in the area of embedded system and automotive electronics. He is a member of IEEE, the Information Processing Society of Japan (IPSJ), the Institute of Electronics, Information and Communication Engineers (IEICE),the Institute of Electrical Engineers of Japan (IEEJ), and Informatics Society.