A Unified Cloud Platform for Autonomous Driving

Shaoshan Liu, PerceptIn
Jie Tang, South China University of Technology
Chao Wang and Quan Wang, Baidu
Jean-Luc Gaudiot, University of California, Irvine

Tailoring cloud support for each autonomous-driving application would require maintaining multiple infrastructures, potentially resulting in low resource utilization, low performance, and high management overhead. To address this problem, the authors present a unified cloud infrastructure with Spark for distributed computing, Alluxio for distributed storage, and OpenCL to exploit heterogeneous computing resources for enhanced performance and energy efficiency.

Clouds provide basic infrastructure support for autonomous driving including distributed computing, distributed storage, and heterogeneous computing. On top of this infrastructure are implemented essential applications such as data storage, simulation testing for new algorithm development, high-definition (HD) map generation, and offline deep-learning model training.[1] Efficient cloud platforms are needed to store and process the enormous amount of raw application data generated by an autonomous vehicle, which can exceed 2 Gbytes per second.

Tailoring cloud platforms to individual applications presents several problems:

- *Lack of dynamic resource sharing.* Cloud platforms designed for one application cannot be used by other applications even if one platform is idle while another is fully loaded.
- *Performance degradation.* Data that is shared across applications—for instance, a newly generated map used in driving simulation workloads—must be frequently copied from one distributed storage

0018-9162/17/$33.00 © 2017 IEEE

element to another, sharply reducing performance.
- *Management overhead*. Each specialized platform might require its own team of engineers to maintain.

To address these problems, we developed a unified cloud infrastructure to provide distributed computing and storage capabilities for autonomous driving (see Figure 1). We also built a heterogeneous computing layer to accelerate different kernels on GPUs or field-programmable gate arrays (FPGAs), improving performance and energy efficiency. We use Apache Spark for distributed computing,[2] Alluxio for in-memory storage,[3] and OpenCL for heterogeneous computing acceleration.[4] By combining the advantages of these three technologies, we can deliver a reliable, low-latency, and high-throughput autonomous driving cloud.

DISTRIBUTED COMPUTING

In building our distributed computing framework for autonomous driving, we had two options: the Hadoop MapReduce engine,[5] which has a proven track record, or Spark, an in-memory distributed computing framework that provides low latency and high throughput.

Specifically, Spark provides programmers with an API centered on a data structure called the *resilient distributed dataset* (RDD), a read-only multi-set of data items distributed over a cluster of machines maintained in a fault-tolerant way. Spark was a response to limitations in the MapReduce cluster computing paradigm, which forces a particular linear dataflow structure on distributed programs: MapReduce programs read input data from disk, map a function across the data, reduce the map's results, and store the reduction results on disk. In contrast, Spark's RDDs function as a working set for distributed programs that offer a restricted form of distributed shared memory. By using in-memory RDD, Spark can reduce the latency of iterative computation by several orders of magnitude.

To determine whether Spark would be a viable solution for autonomous driving, we assessed its ability to deliver the needed performance improvement. First, to verify its reliability, we deployed a 1,000-machine Spark cluster and stress-tested it for three months. This helped us to identify a few bugs in the system, mostly in memory management, that caused the Spark nodes to crash. After fixing these bugs, the system ran smoothly for several weeks with very few crashes. Second, to quantify performance, we ran numerous SQL queries on MapReduce and on a Spark cluster. With the same computing resources, Spark outperformed MapReduce by 5× on average. It took MapReduce more than 1,000 seconds but Spark only 150 seconds to complete an internal query performed daily at Baidu.

DISTRIBUTED STORAGE

After selecting a distributed computing framework, we next needed to decide on the distributed storage engine. Again, we faced two options: the Hadoop Distributed File System (HDFS),[5] which provides reliable persistent storage, or Alluxio, a memory-centric distributed storage system that enables reliable data sharing at memory speed across cluster frameworks.

Specifically, Alluxio utilizes memory as the default storage medium and delivers memory-speed read and write performance. However, memory is a scarce resource and thus we had to determine whether Alluxio would provide enough storage to store all the data. Fortunately, Alluxio has a tiered storage feature that makes it possible to manage multiple storage layers including memory, SSD, and HDD. Using tiered storage, Alluxio can store more data in the system simultaneously in deployments where memory capacity might be limited. Alluxio automatically manages blocks between all the configured tiers, so users and administrators need not manually manage data locations. In essence, memory constitutes the first-level cache, SSD the second level, HDD the third level, and persistent storage the last level.

In our cloud platform, Alluxio is co-located with the compute nodes and Alluxio serves as a cache layer to exploit spatial locality. As a result, the compute nodes can read from and write to Alluxio; Alluxio then asynchronously persists data into the remote storage nodes. Using this technique, we managed to achieve a 30× speedup compared to using the HDFS only.

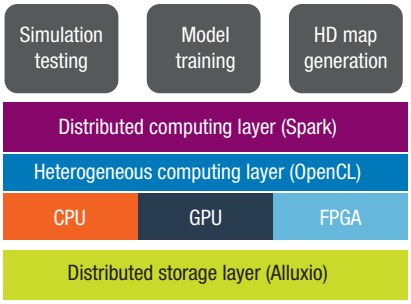

FIGURE 1. Unified cloud platform for autonomous driving. The platform combines the advantages of Apache Spark, Alluxio, and OpenCL to deliver reliable, low-latency, and high-throughput application support.

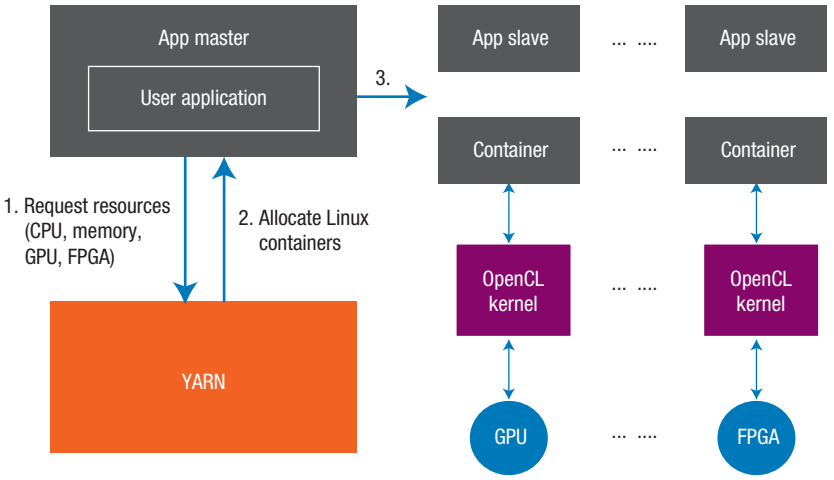

FIGURE 2. Distributed heterogeneous computing platform. When a Spark application is launched, it can request heterogeneous computing resources through Apache Hadoop YARN (1), which then allocates Linux containers to satisfy the request (2). Spark workers can host multiple containers, each of which might contain CPU, GPU, or FPGA computing resources (3).

HETEROGENEOUS COMPUTING

By default, Spark uses a generic CPU as its computing substrate. However, this might not be the best choice for certain types of workloads. For instance, GPUs inherently provide enormous data parallelism, which is highly suitable for high-density computations such as convolutions on images. We compared GPU versus CPU performance on convolution neural network (CNN)-based object-recognition tasks and found that GPUs easily outperform CPUs by a factor of 10 to 20 times. On the other hand, FPGAs are a low-power solution for vector computation, which is the core of most computer vision and deep-learning tasks. Utilizing heterogeneous computing substrates greatly improves performance as well as energy efficiency.

We faced two key challenges integrating these heterogeneous computing resources into our infrastructure: first, how to dynamically allocate different computing resources for different workloads; and second, how to seamlessly dispatch a workload to a computing substrate.

To address the first problem, we use Apache Hadoop YARN and Linux Containers (LXC) for job scheduling and dispatching (see Figure 2). YARN provides resource management and scheduling capabilities for distributed computing systems, enabling multiple jobs to share a cluster efficiently. LXC is an OS-level virtualization tool for running multiple isolated Linux systems (containers) on the same host. It allows isolation, limitation, and prioritization of CPU, memory, block I/O, network, and other resources. LXC makes it possible to effectively co-locate multiple virtual machines (VMs) on the same host with very low overhead. For example, our experiments showed that the CPU overhead of hosting Linux containers is 5 percent lower than running an application natively.

When a Spark application is launched, it can request heterogeneous computing resources through YARN, which then allocates Linux containers to satisfy the request. Spark workers can host multiple containers, each of which might contain CPUs, GPUs, or FPGAs. In this case, containers provide resource isolation to facilitate high-resource utilization as well as task management.

To solve the second problem, we needed a mechanism to seamlessly connect Spark with these heterogeneous computing resources. Because Spark uses a Java VM by default, the first challenge is to deploy workloads to the native space. Given Spark's RDD-centered programming interface, we developed a heterogeneous computing RDD that could dispatch computing tasks from the managed space to the native space through the Java Native Interface.

We also needed a mechanism to dispatch workloads to GPUs or FPGAs, for which we chose OpenCL due to its availability on different heterogeneous computing platforms. Functions executed on an OpenCL device are called *kernels*. OpenCL defines an API that allows programs running on the host to launch kernels on the heterogeneous devices and manage device memory.

SIMULATION TESTING

Our proposed unified cloud platform for autonomous driving supports several applications. Among these applications are distributed simulation tests for new algorithm deployment.

Before a new algorithm is qualified to deploy on an actual vehicle for

on-road testing, it must be thoroughly tested.[6] One simulation approach is to replay data through the Robot Operating System (ROS; www.ros.org) to identify problems. Testing new algorithms on a single machine would either take too long or provide insufficient test coverage. We therefore leveraged Spark to build a distributed simulation platform that lets us deploy a new algorithm on many compute nodes, feed each node with different chunks of data, and aggregate the test results.

To seamlessly connect ROS to Spark, we had to solve two problems. First, Spark by default consumes structured text data, but for simulations it must consume multimedia binary data recorded by the ROS such as raw or filtered sensor readings and bounding boxes of detected obstacles. Second, ROS must be launched in the native environment but Spark lives in the managed environment.

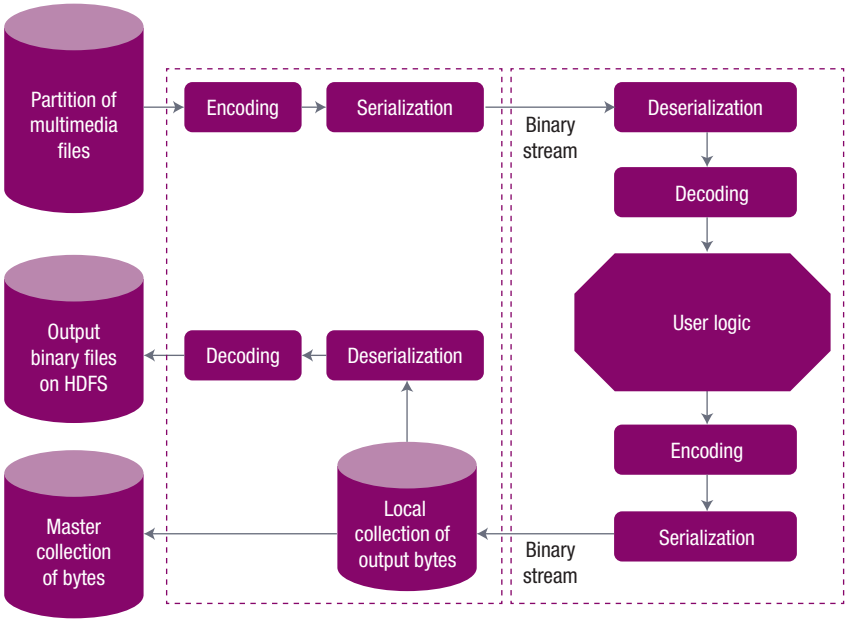

FIGURE 3. BinPipeRDD operation in a Spark executor. BinPipeRDD transforms multimedia binary data into a user-defined format and the output of the Spark computation into a byte stream for collect operations. The byte stream can in turn be converted into text or generic binary files in the Hadoop Distributed File System (HDFS) according to application needs and logic.

BinPipeRDD

Spark's original design assumes that inputs are in text format—for example, records with keys and values that are separated by space/tab characters or records separated by carriage-return characters. In binary data streams, however, each data element in a key/value field could be of any value.

To tackle this problem, we designed and implemented BinPipeRDD. Figure 3 shows how BinPipeRDD works in a Spark executor. First, partitioned multimedia binary files go through encoding and serialization to form a binary byte stream. All supported input formats including strings (for example, file name) and integers (for example, binary content size) are encoded into our uniform format, which is based on byte array. Serialization combines all byte arrays (each might correspond to one input binary file) into a single stream. Upon receiving that stream, the user program deserializes it and decodes it into an understandable format. The user program then performs the target computation ("user logic" in the figure), which ranges from simple tasks such as rotating a JPEG file by 90 degrees to relatively complex tasks such as detecting pedestrians given from LiDAR (light detection and range) sensor readings. The output is then encoded and serialized before being passed on in the form of RDD[Bytes] partitions. In the last stage, the partitions are returned to the Spark driver through a collect operation or stored in the HDFS as binary files.

With this process, binary data can be transformed into a user-defined format and the output of the Spark computation transformed into a byte stream for collect operations. The byte stream can in turn be converted into text or generic binary files in the HDFS according to application needs and logic.

Connecting Spark to the ROS

With BinPipeRDD, Spark could consume ROS bag data, but we needed a way to launch ROS nodes in Spark as well as a means for ROS nodes and Spark to communicate. One choice was to design a new form of RDD to integrate ROS nodes and Spark executors, but this would have involved changing the ROS and Spark interfaces. To avoid having to maintain different ROS versions, we opted to launch ROS and Spark independently, while

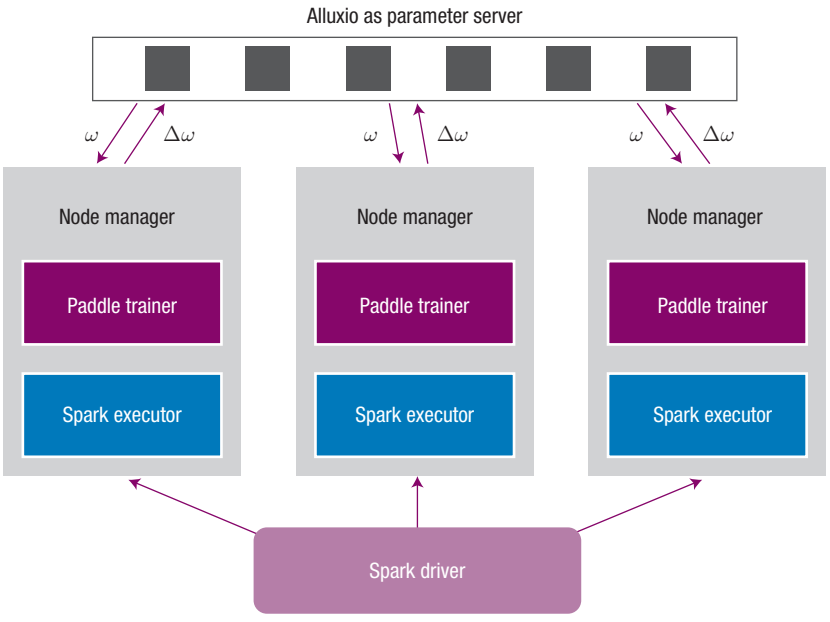

FIGURE 4. Training platform architecture for autonomous driving. A Spark driver manages all the Spark nodes, with each node hosting a Spark executor and a Paddle trainer. This architecture exploits data parallelism by partitioning all training data into shards so that each node independently processes one or more shards.

co-locating the ROS nodes and Spark executors and providing Linux pipes for them to communicate. Linux pipes create a unidirectional data channel for interprocess communication in which the kernel buffers data written to the pipe's write end until it is read from the pipe's read end.

System performance

As we developed the system, we continually evaluated its performance. First, we carried out basic feature-extraction tasks on one million images (total dataset size > 12 Tbytes). As we scaled from 2,000 to 10,000 CPU cores, the execution time dropped from 130 seconds to about 32 seconds, demonstrating extremely promising linear scalability. Next, we ran an internal replay simulation test set. It took about 3 hours to finish the simulation on a single Spark node but only about 25 minutes on 8 nodes, again demonstrating excellent potential scalability.

MODEL TRAINING

Another application our unified cloud infrastructure supports is offline model training. To achieve high performance, it provides seamless GPU acceleration as well as in-memory storage for parameter servers.

As autonomous driving relies on different deep-learning models, it is imperative to provide updates that will continuously improve the models' effectiveness and efficiency. Given the enormous amount of raw data generated, fast model training cannot be achieved using single servers. To address this problem, we developed a highly scalable, distributed deep-learning system using Spark and Baidu's Parallel Distributed Deep Learning (Paddle) platform (www.paddlepaddle.org). In the Spark driver, we can manage a Spark context and a Paddle context, and in each node, the Spark executor hosts a Paddle trainer instance. On top of that, we use Alluxio as a parameter server. With this system, we have achieved linear performance scaling, even as we add more resources, proving that the system is highly scalable.

Why Spark?

One might wonder why we use Spark as the distributed computing framework for offline model training, given that existing deep-learning frameworks all have distributed training capabilities. The main reason is that data preprocessing might consist of multiple stages—for example, ETL (extract, transform, and load) operations rather than simple feature extraction. Treating each stage as a standalone process results in extensive I/O to the underlying storage, such as the HDFS, and our tests revealed that this often becomes a bottleneck in the processing pipeline.

Spark buffers intermediate data in memory in the form of RDDs. The processing stages naturally form a pipeline without intensive remote I/O accesses to the underlying storage in between the stages. In this way, the system reads raw data from the HDFS at the beginning of the pipeline, then passes the processed data to the next stage in the form of RDDs, and finally writes the data back to the HDFS. This approach doubles, on average, system throughput.

Training platform architecture

Figure 4 shows the training platform architecture. A Spark driver manages

all of the Spark nodes, with each node hosting a Spark executor and a Paddle trainer. This architecture exploits data parallelism by partitioning all training data into shards so that each node independently processes one or more shards.

To synchronize the Spark nodes, at the end of each training iteration the system must summarize all the parameter updates from each node, perform calculations to derive a new set of parameters, and then broadcast the new set of parameters to each node so they can start the next training iteration. If we were to store the parameters in the HDFS, I/O would become a performance bottleneck. To alleviate this problem, we use Alluxio as our parameter server. As indicated earlier, Alluxio leverages in-memory storage to optimize I/O performance. With Alluxio, we observed an I/O performance gain of 5× compared to using the HDFS.

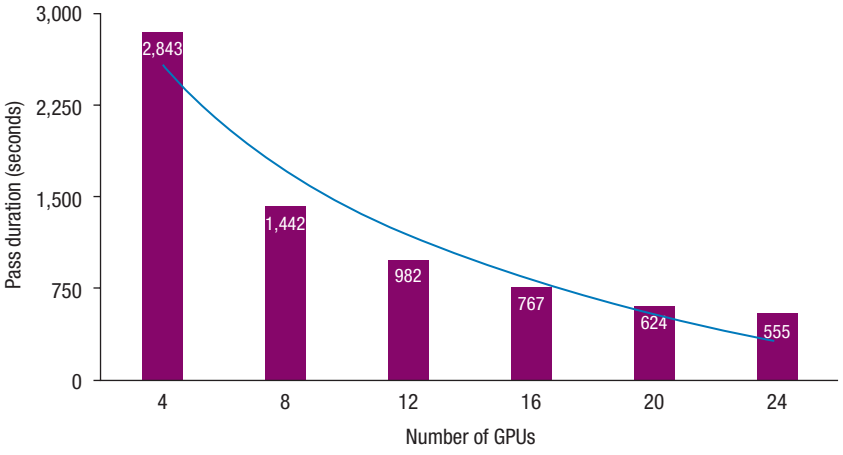

FIGURE 5. Distributed deep-learning model training latency per pass. As the number of GPUs was scaled, latency dropped almost linearly. This result showed that, with more data to train against, adding more computing resources could significantly reduce the training time.

Heterogeneous computing

Next, we explored how heterogeneous computing could improve the efficiency of offline model training. As a first step, we compared GPU and CPU performance in CNNs. Using an internal object-recognition model with the OpenCL infrastructure, we observed a 15× speedup using a GPU. The second step was to determine this infrastructure's scalability. On our machine, each node is equipped with one GPU card. As we scaled the number of GPUs, the training latency per pass dropped almost linearly (see Figure 5). This result showed that, with more data to train against, adding more computing resources could significantly reduce the training time.

HD MAP GENERATION

The third application our unified cloud infrastructure supports is HD map generation. This is a complex process that involves multiple stages, including raw data reading, filtering and preprocessing, pose recovery and refinement, point-cloud alignment, 2D reflectance map generation, HD map labeling, and outputting of the final map.[7,8] Spark's in-memory computing mechanism eliminates the need to store intermediate data on hard disk and thus makes it possible to connect all these stages into one job. Using Spark and heterogeneous computing, we reduced I/O between the pipeline stages and greatly accelerated the map production process.

HD maps

HD maps for autonomous driving have many layers of information. The bottom layer is a grid map of raw, LiDAR-generated elevation and reflection data about the environment, at about 5 cm × 5 cm granularity. As vehicles move, they compare newly connected LiDAR data against the grid map in real time, with initial position estimates provided by GPS and/or inertial measurement unit (IMU) to assist in precise self-localization. On top of the grid layer are several layers of semantic information. For instance, lane labels enable autonomous vehicles to determine whether they are in the correct lane and maintaining a safe distance from neighboring lanes. In addition, vehicles use traffic sign labels to determine the current speed limit and location of nearby signs in case the vehicle sensors fail to detect the signs.

Map generation in the cloud

To derive accurate vehicle position information, the HD map-generation process fuses raw data from multiple sensors.[9] For instance, wheel odometry and IMU data can be used to perform propagation—that is, to derive displacement of the vehicle within a fixed amount of time. GPS and LiDAR data can then be used to

ABOUT THE AUTHORS

SHAOSHAN LIU is chairman and cofounder of PerceptIn. His research focuses on computer architecture, big data platforms, deep-learning infrastructure, and robotics. Liu received a PhD in computer engineering from the University of California, Irvine. Contact him at shaoshan.liu@perceptin.io.

JIE TANG is an associate professor in the School of Computer Science and Engineering at South China University of Technology. Her research interests include computer architecture, autonomous driving, big data storage, and cloud computing. Tang received a PhD in computer science from the Beijing Institute of Technology. She is a member of IEEE. Tang is the corresponding author of this article. Contact her at cstangjie@scut.edu.cn.

CHAO WANG is a senior software architect in Baidu's Autonomous Driving Unit, focusing on distributed simulation testing. He received an MS in computer science from the University of Southern California. Contact him at wangchao30@baidu.com.

QUAN WANG is a principal architect in Baidu's Autonomous Driving Unit, focusing on high-definition map generation. He received a PhD in computer science from the University of Southern California. Contact him at wangquan02@baidu.com.

JEAN-LUC GAUDIOT is a professor in the Department of Electrical Engineering and Computer Science at the University of California, Irvine. His research interests include multithreaded architectures, fault-tolerant multiprocessors, and implementation of reconfigurable architectures. He received a PhD in computer science from the University of California, Los Angeles. Gaudiot is a Fellow of IEEE and the American Association for the Advancement of Science, and current president of the IEEE Computer Society. Contact him at gaudiot@uci.edu.

correct the propagation results to minimize errors.

The computation of map generation can be divided into three stages. First, simultaneous localization and mapping (SLAM) is performed on the vehicle's raw IMU, wheel odometry, GPS, and LiDAR data to derive the location of each LiDAR scan. Second, point-cloud alignment is performed, in which the independent LiDAR scans are stitched together to form a continuous map. Third, labels and other semantic information are added to the grid map. As with offline model training, we linked these stages together using Spark and buffered the intermediate data in memory. This approach achieved a 5× speedup compared to having separate jobs for each stage. Using our heterogeneous computing infrastructure also accelerated the most expensive map-generation operation, iterative closest point (ICP) point-cloud alignment,[10] by 30× by offloading core ICP operations to the GPU.

Distributed computing, distributed storage, and hardware acceleration through heterogeneous computing capabilities are all needed to support different autonomous-driving applications. Tailoring cloud support for each application would require maintaining multiple infrastructures, potentially resulting in low resource utilization, low performance, and high management overhead. We solved this problem by building a unified cloud infrastructure with Spark for distributed computing, Alluxio for distributed storage, and OpenCL to exploit heterogeneous computing resources for enhanced performance and energy efficiency. Our infrastructure currently supports simulation testing for new algorithm deployment, offline deep-learning model training, and HD map generation, but it has the scalability to meet the needs of new and emerging applications in this quickly evolving field.

ACKNOWLEDGMENTS

This work is partly supported by South China University of Technology Start-up Grant No. D61600470, Guangzhou Technology Grant No. 201707010148, and National Science Foundation (NSF) Grant No. XPS-1439165. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily represent the views of the NSF.

REFERENCES

1. S. Liu, J. Peng, and J.-L. Gaudiot, "Computer, Drive My Car!," *Computer*, vol. 50, no. 1, 2017, p. 8.
2. M. Zaharia et al., "Spark: Cluster Computing with Working Sets," *Proc. 2nd USENIX Conf. Hot Topics in Cloud Computing* (HotCloud 10), 2010, article no. 10.
3. H. Li et al., "Reliable, Memory Speed Storage for Cluster Computing Frameworks," *Proc. ACM Symp. Cloud Computing* (SOCC 14), 2014; doi:10.1145/2670979.2670985.
4. J.E. Stone, D. Gohara, and G. Shi,

"OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Eng.*, vol. 12, no. 3, 2010, pp. 66–73.
5. T. White, *Hadoop: The Definitive Guide*, 3rd ed., O'Reilly Media, 2012.
6. C. Basarke, C. Berger, and B. Rumpe, "Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence," *J. Aerospace Computing, Information, and Communication*, vol. 4, no. 12, 2007, pp. 1158–1174.
7. J. Levinson and S. Thrun, "Robust Vehicle Localization in Urban Environments Using Probabilistic Maps," *Proc. 2010 IEEE Int'l Conf. Robotics and Automation* (ICRA 10), 2010, pp. 4372–4378.
8. M. Schreiber, C. Knöppel, and U. Franke, "LaneLoc: Lane Marking Based Localization Using Highly Accurate Maps," *Proc. 2013 IEEE Intelligent Vehicles Symp.* (IV 13), 2013, pp. 449–454.
9. A. Geiger et al., "Vision Meets Robotics: The KITTI Dataset," *Int'l J. Robotics Research*, vol. 32, no. 11, 2013, pp. 1231–1237.
10. A.V. Segal, D. Haehnel, and S. Thrun, "Generalized-ICP," *Proc. 2009 Robotics: Science and Systems Conf.* (RSS 09), 2009; doi:10.15607/RSS.2009.V.021.