

Shareable Camera Framework for Multiple Computer Vision Applications

Hayun Lee, Gyeonghwan Hong, Dongkun Shin

Department of Electrical and Computer Engineering, Sungkyunkwan University, Korea

lhy920806@skku.edu, redc7328@skku.com, dongkun@skku.com

Abstract— On IoT devices such as autonomous driving drones, computer vision jobs such as video recording, streaming and object detection use same camera frame. However, since these IoT devices are resource-constrained systems, they have two problems. First, these applications often do duplicated processing for the same camera raw frame. Second, scheduling between computer vision applications is difficult. In this paper, we propose a shareable camera framework that performs the tasks of computer vision applications. This framework converts the existing pipeline to a pipeline that does not have redundant processing based on the data flow whenever it receives a request from the applications. It also has a scheduling algorithm to guarantee quality-of-service of the applications in the resource-constrained systems. With the proposed framework, the IoT application developers can easily develop reliable computer vision applications that share a single camera simultaneously.

Keywords— Camera framework, component sharing, computer vision, IoT device, scheduling

I. INTRODUCTION

On IoT devices such as autonomous driving drones, computer vision tasks such as video recording, streaming and object detection use same camera frame. Deep neural networks (DNN) are generally used for computer vision tasks, which consume more significant computing resources such as CPU computation and memory than traditional computer vision tasks. In order to execute DNN for computer vision tasks, high-performance computing systems are required.

There are many researches on computer vision tasks for IoT devices with leveraging cloud offloading [1], [2]. The researches try to find optimization point of allocating such tasks on IoT devices or cloud servers, considering trade-off among accuracy, speed, and power consumption.

However, the cloud offloading approach has three problems. At first, excessive network traffic incurs significant networking cost. Second, since the wireless connection between IoT devices and cloud server is unstable, vision tasks are seldom available. Third, the unpredictable latency between IoT devices and cloud servers makes it hard to meet time constraint for time-critical vision tasks.

Various computer vision tasks run on recent IoT devices. In the example, as shown in Figure 1, four computer vision applications running on autonomous driving drone simultaneously; recording, snapshot, object detection, lane tracing and streaming. Recording is an application that stores

video recorded by its camera in the period of 5 minutes. Snapshot is an application that stores photo captured by the camera on other application's demand. Object detection is an application that detects bumps on the drone and triggers other operations if a bump is detected. It can execute the snapshot application or make obstacle avoidance commands. Lane tracing is an application that detects where the lane is and make tracing commands to follow the lane. At the same time, recording application transmits streaming video captured by the camera to the user's smartphones.

Previous research work mainly optimizes each computer vision task such as recording, object detection, lane tracing, etc. However, there are few researches on optimizing the system executing the multiple computer vision tasks simultaneously.

In order to run multiple computer vision applications on resource-constrained IoT devices, several optimization points should be considered to design the system. At first, resource usage of each application should be optimized. Since multiple computer vision applications often use same camera frame to do their jobs, the processing tasks of the applications can be duplicated. For example, compression jobs can be duplicated in recording applications and video streaming applications. Second, quality of service (QoS) (e.g., resolution, deadline, FPS) of each job should be maintained. For example, lane tracing and object detection are kinds of time-critical application, which result in fatal consequences if they fail to meet the deadlines. Third, since simultaneous access on camera device of multiple applications is usually unavailable, an additional software layer that multiple computer vision applications share the same camera frames is required.

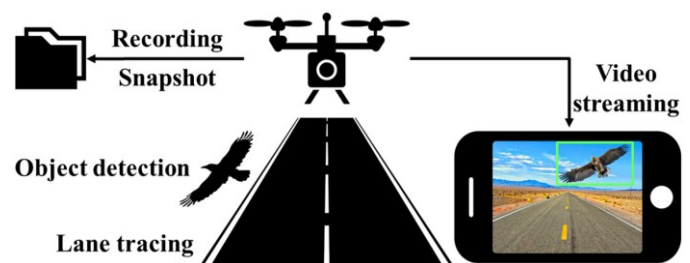


Figure 1. A scenario of autonomous driving drone running computer vision applications simultaneously

In this paper, we propose a shareable camera framework that optimizes the simultaneous execution of multiple computer vision applications. This framework has the following three contributions.

- This framework provides virtualization view in order that multiple computer vision applications share and use one camera device for their own purpose simultaneously.
- This framework dynamically optimizes the computation resource and memory resource by removing duplicated processing of multiple computer vision applications.
- This framework schedules the processing jobs of each computer vision applications to guarantee the QoS of the applications.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 provides the background of this work. Section 4 describes the design of the proposed framework. Section 5 offers details of the proposed framework implementation. Section 6 evaluates the proposed framework, and Section 7 contains the conclusion.

II. RELATED WORK

Recent work in the mobile cloud computing has optimized the performance of mobile computer vision tasks by offloading to cloud servers. Yang et al.[1] proposed a framework for partitioning execution of feature extraction into mobile part and cloud part. MCDNN[2] schedules execution on mobile device and cloud to achieve maximum accuracy within resource bounds for deep neural networks using a model catalog. However, these researches require the assumption that the cloud always exists. We designed it to run only on devices in cloud-less systems.

This work is inspired by Starfish[9]. Starfish caches results for vision library function calls and removes redundant computation and memory usage through memoization. However, Starfish focused on optimizing the performance of feature extraction such as [1] and [2] mentioned above. We designed not only feature extraction but also various computer vision applications such as recording, streaming and so on. In Starfish, buffer management and searching are required for memorization. However, our framework does not require searching because it uses a pre-configured pipeline for camera's streaming data.

TABLE 1. COMPONENTS OF COMPUTER VISION APPLICATIONS

Location	Component Name
Front	Scaling (Sca)
Middle	Converting Color Space (Con)
	Encoding (Enc)
	Muxing (Mux)
	Packetizing (Pac)
Back	Storing File (Sto)
	Streaming (Str)
	Feature Extraction (Fea)

III. BACKGROUND

As shown in Table 1, computer vision applications are composed of a series of common components. Following descriptions are the roles of each computer vision component.

- **Scaling:** component that changes the size of frame
- **Converting Color Space:** component that changes the color space of frame (e.g., I420 → RGB)
- **Encoding:** component that compresses the frame in various encoding methods (e.g., JPEG, H.264)
- **Muxing:** component that fuses video frame and audio data into one container (e.g., AVI, MP4)
- **Packetizing:** component that makes packet to be transmitted through network
- **Storing File:** component that stores the result data into a file
- **Streaming:** component that transmits a data stream to remote devices through IP protocol stacks such as TCP or UDP.
- **Feature Extraction:** component that extracts features from frame (e.g., lane tracing, object detection)

There are three locations that each computer vision task can be located; front, middle and back, as shown in Table 1. In this paper, the location of each component is fixed. For example, Scaling (*Sca*) component is always located in the front part and gets camera frame from the camera devices. On the other hand, making result is done by Storing File (*Sto*), Streaming (*Str*) or Feature Extraction (*Fea*).

TABLE 2. COMPONENT ORGANIZATIONS OF COMPUTER VISION APPLICATIONS

Computer Vision Applications	Component Organizations
Recording	Sca – Con – Enc – Mux – Sto
Snapshot	Sca – Con – Enc – Sto
Streaming	Sca – Con – Enc – Pac – Str
Feature Extraction	Sca – Con – Fea

Table 2 shows the examples of component organizations of computer vision applications. The applications use same camera frame as its input data. If the formats of input data and output data of the component are same, the components can be regarded as ones performing same operations. Therefore, sharing components that perform same operations results in same output data.

For example, as shown in Figure 2, there are two applications. One is a recording application that stores recorded video in the form of 720p-H.264-AVI, and another is a streaming application that transfers streaming video in the form of 720p-H.264-TCP. If components of two applications are not shared, two series of components run separately, as shown in Figure 2-(a). In this case, the two application's components have components of same data type, between Scaling component and Encoding component. Therefore, the duplicated components can be shared, as shown in Figure 2-(b). In this case, since components in the front part are shared, it consumes less computation cost and memory cost.

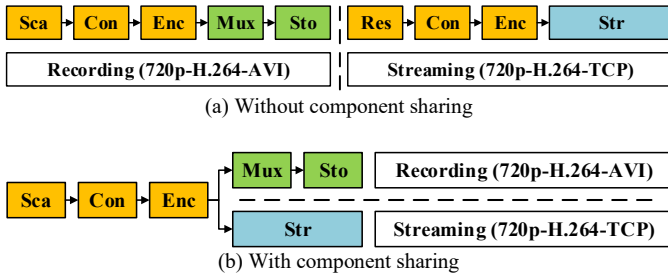


Figure 2. Example of two computer vision applications without and with component sharing

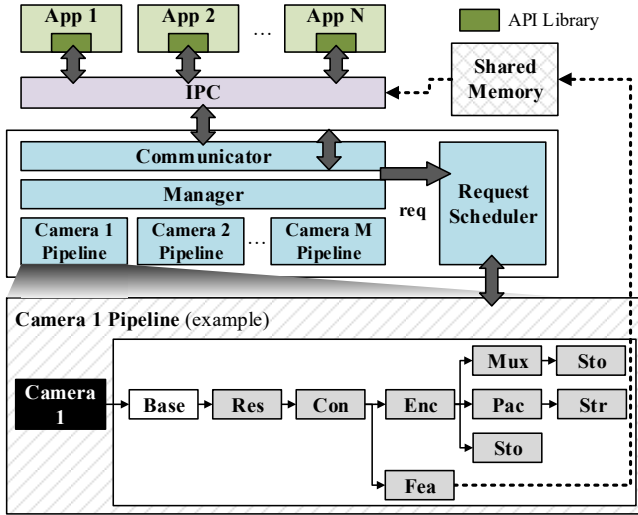


Figure 3. The architecture of shareable camera framework

IV. Design

Figure 3 shows the overall architecture of the shareable camera framework proposed in this paper. Following sub-chapters cover the detailed descriptions of its architecture.

A. Proxy Daemon

Shareable camera framework works as a proxy daemon that delegates the handling of request issued by computer vision applications. On launching the daemon, Manager initializes the pipeline composed of base components for all the cameras installed on the device. The base components mainly read camera frame from the camera devices. After that, Communicator waits for requests that will be issued by applications. When it receives a request, it delivers the contents of the request to the Manager. Since this framework delegates the processing tasks of each processing, multiple computer vision applications can share one camera and run simultaneously.

B. API Library

In order that the applications perform computer vision task, the applications should transmit the request to the framework through inter-process communication (IPC). This framework provides an application programming interface (API) library to communicate with the framework. This library transmits requests to the framework by some function calls. Applications can also transmit component's configurations

(e.g., resolution, FPS, compression format) to the framework as arguments of the function calls.

All the API function can be called in synchronous or asynchronous manner. In the case of synchronous call, the application waits for the task and receives return value. On the other hand, in the case of asynchronous call, applications just transmit the request to the framework and do not wait for the framework. Exceptively, Feature extraction APIs are always called in synchronous manner since it should retrieve shared memory information in synchronous manner. For Feature Extraction APIs, only pre-processing jobs are performed in the framework, feature extraction jobs are performed on application-side.

C. Dynamic Component Sharing

Framework Manager receives request issued by Communicator and transmits it to the Request Scheduler. Request Scheduler dynamically allocates components to the corresponding request of each camera pipeline and connects the components. At first, Request Scheduler checks whether components shareable with existing pipeline exist or not. If there are shareable components, the Request Scheduler does not allocate the shareable components and allocates the other components. Allocated components are connected to the last component of the series of shareable components. On the other hand, if there is not a shareable component, it allocates all the components that are required for the request and connect it to the base components.

The pipeline example in Figure 3 shows that all the computer vision applications mentioned in table 2 are in execution and sharing the pipeline. We assume that the component configurations of each application are the same for component sharing. Then the recording, snapshot, and streaming can be shared to the *Enc* component. In addition, feature extraction can be shared to the *Con* component.

Depending on the frame size of compression format required by the request, there can be few shareable components. If the device's computing resources are insufficient, it results in fatal consequences. In order to resolve the problem, computer vision application developers can write their applications to have as many shareable components as possible. In the systematic approach, it can be resolved by task scheduling with QoS configurations.

D. Task Scheduling

This framework has task scheduling algorithm to guarantee the QoS of computer vision applications. If the device is short of computing resources, the framework can adjust the configurations of the components to meet both resource limits and QoS of the applications. In this paper, we have defined the QoS of computer vision applications as a requirement of the applications that must be followed, such as resolution and deadline. This framework uses two following strategies for task scheduling algorithms.

1) Strategy 1: It shares more components while ensuring the QoS of requests except Recording. Since recording

application should store frames in the same format, Recording components are excluded from sharing target.

2) Strategy 2: It delays the frame completion time of Feature extraction component with ensuring the QoS of applications.

V. IMPLEMENTATION

Shareable camera framework reads camera frame and performs various components. Each component is implemented with the plugins of GStreamer[3]. In this framework, a component is composed of one or more than one plugin of GStreamer. This framework leverages tee plugin of GStreamer to ramify the non-shared parts.

This framework uses D-Bus[4] library for the IPC between the framework and applications. D-Bus provides stateful and reliable connections between the processes.

In this paper, we used deep learning open source framework Caffe[5] for the feature extraction applications. In the evaluation, we used pre-trained SqueezeNet[6] model to evaluate this framework.

API library is implemented as a form of C library. API library is composed of functions and configuration data structure on computer vision tasks. Other functions except feature extraction are implemented in both synchronous call and asynchronous call. In the case of feature extraction API, it returns the address and size of shared synchronously.

VI. EXPERIMENTS

We have implemented the shareable camera framework as mentioned in Section 5. As for IoT device, our target device is the Raspberry Pi 2 board[7], which includes a 900MHz quad-core ARM Cortex-A7 CPU, and 1GB RAM. As for the camera device, Raspberry Pi Camera Module V2[8] is connected to the target device, which produces 1080p images at 30fps. The CPU utilization mentioned in the experiment is the average of each core utilization. ‘No sharing’ means the original system that the component sharing is not applied, whereas ‘sharing’ means the proposed system that the component sharing is applied.

A. Impacts of Component Sharing

Figure 4 shows the result of using and not using component sharing in the example of Figure 2. Component sharing almost halves CPU utilization compared to the original system, because *Enc* component occupies most of the CPU utilization. However, memory usage also does not decrease by half, because unshared components, tee plugin, and camera framework except pipeline occupy memory space in addition to the shared components. After this, the experiment is carried out by changing the number of applications.

Figure 5 shows average CPU utilization and maximum memory usage according to the number of recording applications. ‘sharing (*X*)’ means that the components are shared up to *X* component. For no sharing and ‘sharing (*Sca/Con*)’, there is no result for six or more applications because there is a limit on the number of simultaneous accesses on H.264 encoders. However, the framework can

solve the problem of a limited number of components such as ‘sharing (*Enc*)’ and ‘sharing (*Mux*)’, because they use only one *Enc* component (e.g., H.264 encoder).

‘No sharing’ increases both CPU utilization and memory usage depending on the number of applications. However, when the newly allocated components are shared with *Sca* and *Con* components, the resource usage does not increase drastically compared to ‘no sharing’. In contrast, when the newly allocated components are shared with *Enc* and *Mux* components, CPU utilization and memory usage are almost constant regardless of the number of applications. This means that framework can dramatically reduce computation and memory usage when applications are handling the same frame size. This idea is adopted by the strategy 1 of task scheduling.

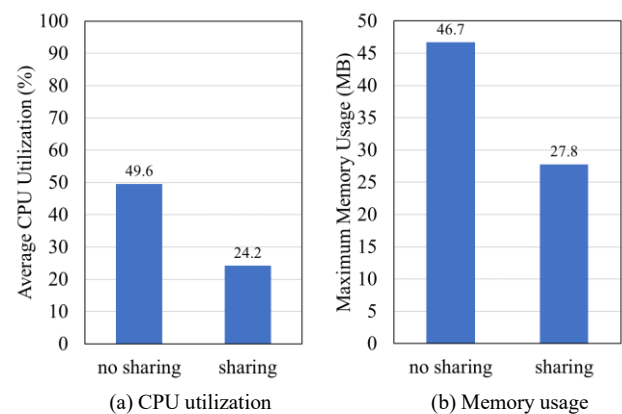


Figure 4. Performance evaluation for the example scenario of two computer vision applications as shown in Figure 2

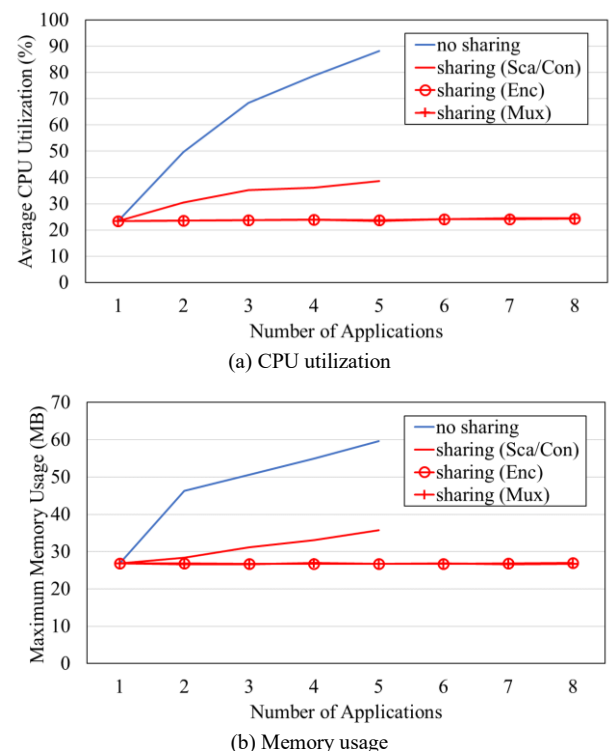


Figure 5. Resource usages of the framework running computer vision tasks for multiple recording applications

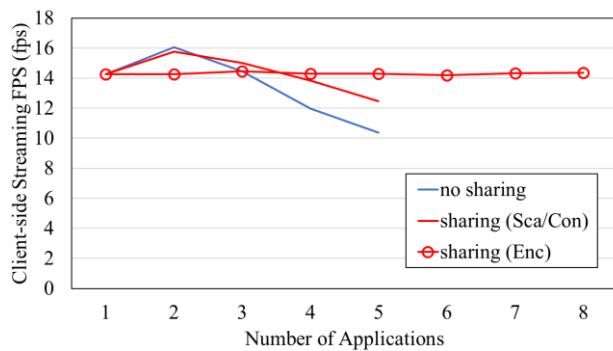


Figure 6. Performance of the framework running computer vision tasks for multiple streaming applications

Figure 5-(b) shows that the memory usage is constant when shared to the *Enc* component. This is because the new unshared components and tee plugins are small enough not to affect overall memory usage. Therefore, most of the memory space except the shared components in Figure 4 is occupied by the camera framework except pipeline.

The performance of the camera framework running *streaming* is similar to one running *recording*. Figure 6 shows the streaming FPS on the client side according to the number of *streaming* applications. The streaming FPS is important because it is an indicator of the quality of output from the consumer's perspective. The streaming FPS is 14.25 when the number of application is one. Both of 'no sharing' and 'sharing (Sca)' have increased by 16.07 and 15.78 respectively when the number of applications was two, and have decreased since then. However, 'sharing (Sca/Con)' has a higher FPS than 'no sharing'. Also, it shows that 'sharing (Enc)' maintains almost constant FPS.

In this experiment, 'sharing (Enc)' does not always guarantee high streaming FPS compared to other policies. It shows better performance when performing two or three jobs than performing one job with H.264 encoder. In other words, this result implies that low CPU utilization and memory usage do not necessarily guarantee the user's QoS.

B. Impacts of Task Scheduling

To verify the effectiveness of task scheduling, we executed the applications with the scenario in Table 3. The applications are predefined by the user with initial configuration and QoS. Figure 7 and 8 shows the result of the experiment. S_i means the initial state where component sharing is done. S_1 means that strategy 1 is applied, and S_{1+2} means that the combination of both strategy 1 and strategy 2 are applied.

In S_1 , QoS is satisfied even if frame size is set as 1080p for both *snapshot* and *streaming* because frame size is larger than 480p. When the frame size of two tasks is changed to 1080p, *Sca* component scaling to 720p and *Enc* component encoding H.264 are removed. As two components are removed, CPU utilization is reduced by 12.3%, memory usage is reduced by 30.3MB, and streaming FPS is increased by 1.8x.

TABLE 3. COMPONENT CONFIGURATIONS IN TASK SCHEDULING EXPERIMENTS SCENARIO

Computer Vision Applications	Initial Configuration	QoS
Recording	1080p-H.264-AVI	-
Snapshot (per 1sec)	720p-JPEG	480p ~
Streaming	720p-H.264-TCP	480p ~
Feature Extraction (Image classification)	480p	~ 3sec

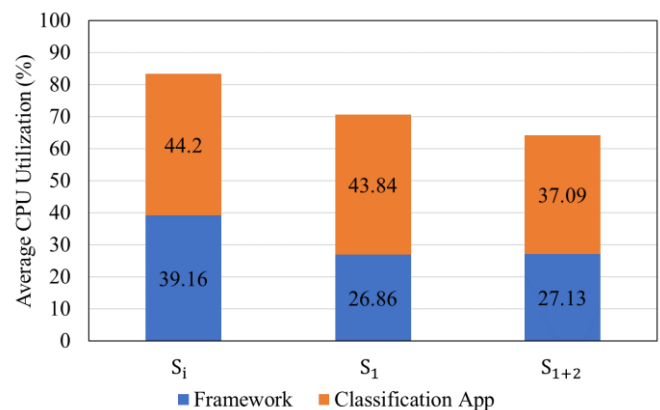


Figure 7. Average CPU utilization of task scheduling scenario

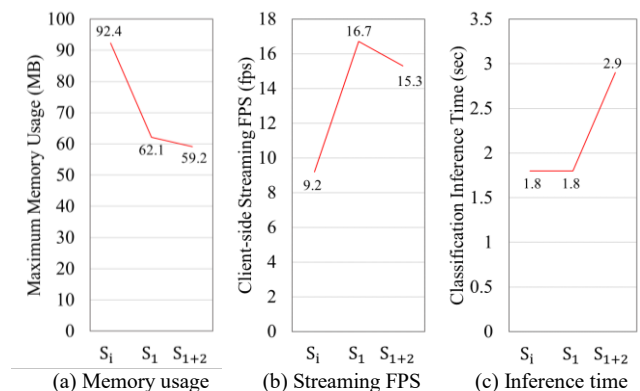


Figure 8. Resource usage and performance of task scheduling scenario

In S_{1+2} , *feature extraction* (image classification) was scheduled to be executed every 3 seconds while satisfying QoS. We have noticed the CPU utilization of the framework and classification application. In the former case, there is a little difference in CPU utilization. On the other hand, in the latter case, the CPU utilization was reduced by 6.75 percent with delay in inference time from 1.8 to 2.9 seconds.

Each strategy applied to S_1 and S_{1+2} was effective in reducing computation and memory usage while ensuring QoS. Especially, strategy 2 is effective in reducing system-wide computation rather than reducing memory usage.

VII. CONCLUSION

In this paper, we propose shareable camera framework that supports concurrent computer vision applications without sacrificing performance on resource-constrained systems such as IoT devices. It is accomplished by sharing components and scheduling them to ensure QoS of applications. Experimental results show that the proposed framework effectively reduces computation and memory usage.

However, this work has three limitations. Firstly, users should decide which components are shared manually to achieve most effective resource usage. Secondly, current scheduling algorithm is not optimal. Finally, we did not deal with issues when performing in real workload. In the future, we will address these limitations.

ACKNOWLEDGMENT

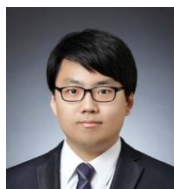
This research was supported by the MSIT(Ministry of Science and ICT), Korea, under the SW Starlab support program(IITP-2017-0-00914) supervised by the IITP(Institute for Information & communications Technology Promotion)

REFERENCES

- [1] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, "A framework for partitioning and execution of data stream applications in mobile cloud computing," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 4, pp. 23-32, Mar. 2013.
- [2] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints," in *Proc. MobiSys '16*, 2016, pp. 123-136.
- [3] (2017) GStreamer: open source multimedia framework. [Online]. Available: <https://gstreamer.freedesktop.org/>
- [4] (2017) dbus. [Online]. Available: <https://dbus.freedesktop.org/>
- [5] (2017) Caffe. [Online]. Available: <https://github.com/BVLC/caffe>
- [6] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5 MB model size." *arXiv preprint arXiv:1602.07360*, 2016.
- [7] (2017) Raspberry Pi 2 Model B. [Online]. Available: <https://www.raspberrypi.org/products/raspberry-pi-2-model-b/>
- [8] (2017) Camera Module V2. [Online]. Available: <https://www.raspberrypi.org/products/camera-module-v2/>
- [9] R. LiKamWa, and L. Zhong, "Starfish: Efficient concurrency support for computer vision applications." in *Proc. MobiSys '15*, 2015, pp. 213-226.



Hayun Lee received the B.S. degree in Computer Engineering from Sungkyunkwan University, Suwon, Korea, in 2017. He is currently M.S. student in the School of Information and Communication Engineering, Sungkyunkwan University. His research interests include IoT platform and edge machine learning.



Gyeonghwan Hong received the B.S. degree in computer engineering from Sungkyunkwan University, Korea in 2013. He is currently pursuing the Ph. D. degree in Embedded Software Laboratory from Sungkyunkwan University, Suwon, Korea. His research interests include embedded software, mobile system software, Internet of Things, web platform and edge deep learning.



Dongkun Shin received the B.S. degree in computer science and statistics, the M.S. degree in computer science, and the Ph.D. degree in computer science and engineering from Seoul National University, Korea, in 1994, 2000 and 2004, respectively. He is currently an Assistant Professor in the School of Information and Communication Engineering, Sungkyunkwan University (SKKU). Before joining SKKU in 2007, he was a senior engineer of Samsung Electronics Co., Korea. His research interests include embedded software, low-power systems, computer architecture, multimedia and real-time systems.