engineering proceedings

MDPI

Proceeding Paper

Evaluation of Real-Time Performance and Computational Load in Edge–Host Split Processing for Multi-Camera Gaze Estimation†

Yuya Kato and Masayuki Numao *

Department of Information and Computer Science, Faculty of Engineering, Kyoto Tachibana University, Kyoto 607-8175, Japan; a601023112@st.tachibana-u.ac.jp

* Correspondence: numao@tachibana-u.ac.jp

† Presented at the 6th International Conference on Electronic Communications, Internet of Things and Big Data (ICEIB 2026), New Taipei, Taiwan, 24–26 April 2026.

Abstract

Gaze estimation is used in many fields, but maintaining real-time performance becomes difficult when multiple gaze estimation streams are processed simultaneously. Therefore, we investigate how edge–host split processing affects the overall frame rate of a multi-camera gaze estimation system based on Gaze360. Using a Raspberry Pi 5 as the edge device and a graphics processing unit-equipped host personal computer, we defined four split points and evaluated their performance under virtual multi-camera load. The results showed that intermediate split configurations provided the best balance between communication reduction and edge-side computation, enabling stable real-time operation than centralized processing or excessively deep edge-side preprocessing.

Keywords: gaze estimation; edge computing; split processing; real-time performance; multi-camera system; load distribution

1. Introduction

Gaze estimation is important for human–computer interaction, behavior analysis, virtual and augmented reality, and educational support. By estimating where a user is looking, a gaze estimation system can capture visual attention and user engagement. Recent advances in deep learning have improved the accuracy of gaze estimation in unconstrained environments, and Gaze360 is a representative model that estimates gaze from facial image sequences under diverse head poses and lighting conditions [1].

However, it is still difficult to apply such a high-accuracy model to a real-time multi-camera environment. When multiple edge devices continuously send image data to a host personal computer (PC), both computational load and communication traffic increase. Even if the host has a graphics processing unit (GPU), centralized processing can become a bottleneck, and the overall frame rate decreases below the level required for real-time use. Therefore, practical gaze estimation systems must consider not only estimation accuracy but also the balance among edge-side processing, host-side processing, and communication cost.

Edge–host split processing is proposed to solve this problem. In this method, part of the pipeline is executed on the edge device, and the remaining part is executed on the host PC. If the split point is chosen appropriately, communication overhead can be reduced while computational load is distributed more effectively. For gaze estimation, this is especially important because the pipeline contains multiple stages, including image

Check for updates

Academic Editors: Teen-Hang Meen, Shu-Han Liao and Cheng-Fu Yang

Published: 19 October 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

capture, face detection, facial cropping, normalization, temporal tensor construction, and neural network inference [2].

In this study, we investigate split processing for a Gaze360-based gaze estimation system composed of a Raspberry Pi 5 and a host PC with an NVIDIA GeForce RTX 4050 Laptop GPU. We define multiple split points in the processing pipeline and evaluate how each split point affects system performance. The edge device transmits different forms of intermediate data, such as raw Joint Photographic Experts Group (JPEG) frames, facial crops, normalized face tensors, and temporal tensors, and the host executes the remaining processing stages. In addition, instead of preparing many physical edge devices, we emulate a multi-camera environment by duplicating one input stream into N virtual tasks on the server side. Using this setup, we evaluate the system by using the overall frame rate to clarify the split point that best balances communication reduction and edge-side processing overhead under real-time conditions.

2. Materials and Methods

2.1. Gaze360 and Split Points

In this study, we used a Gaze360-based gaze estimation pipeline and investigated how the system behavior changes depending on the split point between the edge device and the host PC. Gaze360 estimates gaze from facial image sequences, and its processing flow includes image acquisition, face detection, facial cropping, normalization, temporal tensor construction, and neural network inference [1]. Because these stages have different computational and communication costs, the position of the split point can significantly affect the real-time performance of the entire system.

We defined four split points, denoted as split indices 0 to 3. At split index 0, the edge device only captures the camera frame and transmits the raw frame as a JPEG image. At split index 1, the edge device performs face detection and facial cropping, and only the cropped facial image is transmitted. At split index 2, the edge device additionally performs preprocessing and transmits a normalized face tensor of size (3, 224, 224). At split index 3, the edge device constructs a temporal tensor of size (7, 3, 224, 224), which is directly used for Gaze360 inference on the host side. Thus, as the split index increases, the amount of computation assigned to the edge device increases, whereas the amount of transmitted data is reduced. Figure 1 illustrates the processing pipeline and the transmitted intermediate representation at each split point.

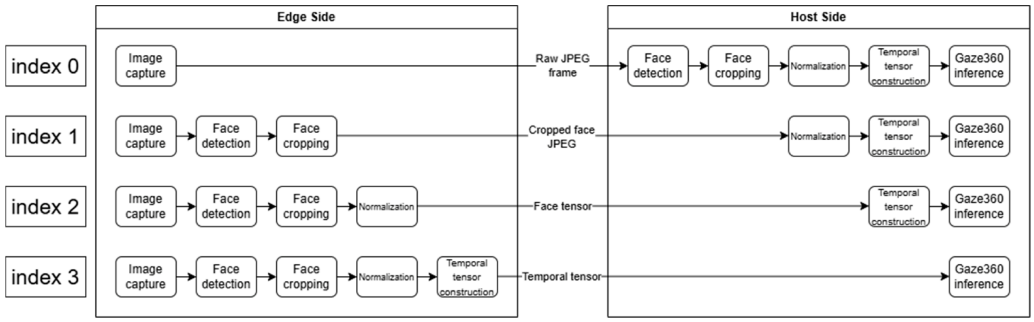


Figure 1. Definition of split points and transmitted intermediate data in the Gaze360-based pipeline. As the split index increases, more preprocessing is executed on the edge device, while the amount of transmitted data is reduced. Arrows indicate the flow of processing and transmitted data.

2.2. System Architecture

The experimental system consisted of an edge device, a host PC, and a universal serial bus (USB) camera. The edge device was a Raspberry Pi 5 (Raspberry Pi Ltd., Cambridge, UK) equipped with a 2.4 GHz quad-core Cortex-A76 CPU, a VideoCore VII GPU, and

8 GB LPDDR4X memory. The host PC (XL7C-R45; Thirdwave Corporation, Tokyo, Japan) had a 13th Gen Intel Core i7-13700H CPU (Intel Corporation, Santa Clara, CA, USA), an NVIDIA GeForce RTX 4050 Laptop GPU (NVIDIA Corporation, Santa Clara, CA, USA), and 16 GB of memory. An Elecom UCAM-C310FBBK HD 720p web camera (ELECOM Co., Ltd., Osaka, Japan) was used for image acquisition. The edge device and the host PC were connected through a wired local area network (LAN), and the camera was connected to the Raspberry Pi via USB.

The client program running on the Raspberry Pi generated intermediate data according to the selected split index and sent it to the host PC. The server program executed the remaining stages of the pipeline and produced the gaze output. The implemented communication protocol was the Transmission Control Protocol with a length-prefixed packet format, and each packet included the split index, a timestamp, metadata, and payload data. On the host side, the Gaze360 model was loaded with PyTorch 2.5.1 (CUDA 12.1) and executed on a compute unified device architecture when GPU was available. Face detection was performed with MediaPipe 0.10.14 on both devices, and image processing used OpenCV (version 4.12.0 on the host PC and 4.13.0 on the Raspberry Pi). The server program ran on Python 3.12.7 under Windows 11, and the client program ran on Python 3.11.10 under Raspberry Pi OS (64-bit, based on Debian 13).

2.3. Virtual Multi-Camera Load Emulation

Instead of preparing many physical edge devices, we emulated a multi-camera environment by duplicating one received input stream into multiple virtual tasks on the server side. More specifically, for each physically received frame or intermediate representation, the server replicated the same data into N virtual processing tasks. Among them, the task with replica_id = 0 was treated as the original task, while the additional replicated tasks were used only to impose computational load and emulate concurrent multi-camera processing. This design allowed us to control the virtual number of edge inputs while keeping the hardware configuration simple and reproducible.

This virtual-load setting was important for evaluating the scalability of split processing under multi-camera conditions. Because the same intermediate representation was processed repeatedly at the host side, the experiment focused on the effect of the host-side computational burden and the interaction between communication reduction and edge-side preprocessing. In this study, N was interpreted as the virtual number of concurrent input streams rather than the number of physically deployed Raspberry Pi devices. Figure 2 shows the overall system architecture and the server-side duplication process used for virtual multi-camera load emulation.

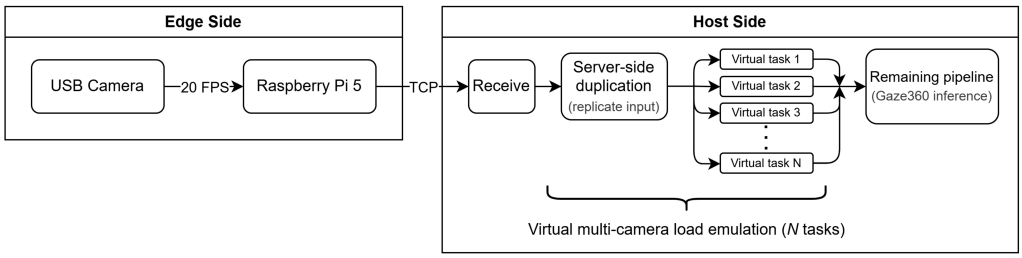


Figure 2. Overall system architecture and virtual multi-camera load emulation. One input stream from the edge device is duplicated into N virtual tasks on the host side to emulate concurrent multi-camera processing. Arrows indicate the flow of data, and the brace indicates the virtual tasks generated by server-side duplication.

2.4. Evaluation Metrics

The primary evaluation metric adopted in this study was the overall frame rate of the system under each split configuration. To discuss the real-time capability of the entire pipeline, we also referred to auxiliary metrics such as edge-side processing behavior, host-side processing latency, and transmission characteristics. On the server side, the implementation separately recorded the physical receive rate, the processing completion rate for the original input, and the processing completion rate for all replicated tasks. These values were used to interpret how efficiently the system sustained virtual multi-input processing under increasing load.

In addition, the real-time requirement was interpreted using the 20 frames per second (FPS) camera setting. Since a 20 FPS stream provides a new frame every 50 ms, processing slower than this interval can degrade real-time behavior. In the experiments, if a frame could not be processed before the arrival of the next input, that frame was discarded. This policy was adopted to prioritize real-time operation over complete frame preservation.

2.5. Experimental Conditions

The evaluation was conducted for a split index of 0 to 3. The client captured frames from the camera and generated the corresponding intermediate representation according to the selected split point. The server then executed the remaining stages of the pipeline. The camera frame rate was set to 20 FPS, and the virtual number of concurrent inputs was varied by changing N. For each split configuration, the system behavior was observed under increasing virtual load to examine how the split point influenced the balance between edge-side computation and host-side processing cost.

In the implementation, indices 4 and 5 were included for simple CPU-side testing; however, the evaluation in this paper focused only on the split index of 0 to 3, which correspond to the actual split processing stages of the Gaze360-based pipeline. In this setting, index 0 represents centralized host-side processing after frame transmission, whereas index 3 represents the deepest edge-side preprocessing before host-side inference. Therefore, the evaluated range covers the main trade-off region between communication reduction and edge-side computational overhead. Table 1 shows the hardware configuration and the main experimental conditions.

Table 1. Hardware configuration and experimental conditions.

Item	Description
Edge device	Raspberry Pi 5; 2.4 GHz quad-core Cortex-A76 CPU; VideoCore VII GPU; 8 GB LPDDR4X
Host PC	Intel Core i7-13700H; NVIDIA GeForce RTX 4050 Laptop GPU; 16 GB memory
Camera	Elecom UCAM-C310FBBK HD 720p
Network	wired LAN
Protocol	TCP with a length-prefixed packet format
Real-time setting	20 FPS input; 50 ms per frame threshold
Frame policy	Discard a frame if processing cannot finish before the next input arrives
Evaluation target	Split index 0–3; virtual load N by server-side duplication

2.6. Use of Generative AI Tools

The manuscript was originally written in Japanese by the authors. DeepL Translator (free web version; DeepL SE, Cologne, Germany) was used to translate the text into English, and ChatGPT (GPT-5.4 Thinking; OpenAI, San Francisco, CA, USA) was used to improve the grammar, phrasing, and readability of the English text and to format the manuscript according to the conference template. Both tools were used in April 2026. In addition, ChatGPT was used to generate code for selected functions of the experimental programs

based on the experimental design developed by the authors; all generated code was reviewed and modified by the authors before use. Generative AI tools were not used for the research idea and design, experimental design, data collection, analysis and interpretation of the results, creation of figures and tables, or the scientific content of the manuscript (arguments, discussion, and conclusions).

3. Results

3.1. Edge-Side Processing Characteristics

The edge-side processing behavior differed depending on the split index. For split index 0 to 2, the Raspberry Pi maintained processing performance close to the camera input rate of 20 FPS, indicating that image capture, face detection, cropping, and normalization could be executed on the edge device without causing a critical bottleneck. In contrast, split index 3 imposed a larger edge-side burden because the Raspberry Pi had to construct a temporal tensor before transmission. In the selected dataset, the Pi-side total latency at index 3 was approximately 53.1 ms, and the camera-side frame rate decreased to about 18.9 FPS. These results indicate that deeper preprocessing on the edge device can reduce host-side work, but excessive edge-side computation may directly limit the sustainable throughput of the entire system.

3.2. Host-Side Latency Under Virtual Load

Host-side latency increased as the virtual load N became larger, but the increase strongly depended on the split index. In the GPU-added experiment sheet adopted in this paper, all split configurations remained below 50 ms at $N = 5$. At $N = 10$, the average host-side latency was 71.3 ms for index 0, 24.5 ms for index 1, 30.4 ms for index 2, and 23.4 ms for index 3. This shows that index 0 exceeded the real-time threshold earlier than the other configurations, while index 1 and index 2 still maintained a sufficient margin at this load level. At $N = 11$, the host-side latency rose to 71.4, 69.6, 59.0, and 50.3 ms for the indices of 0 to 3, respectively, indicating that the host-side advantage of index 1 and index 2 was lost when the virtual load increased further. At $N = 15$, all configurations exceeded 50 ms. Figure 3 plots the host-side latency as a function of N for each split index.

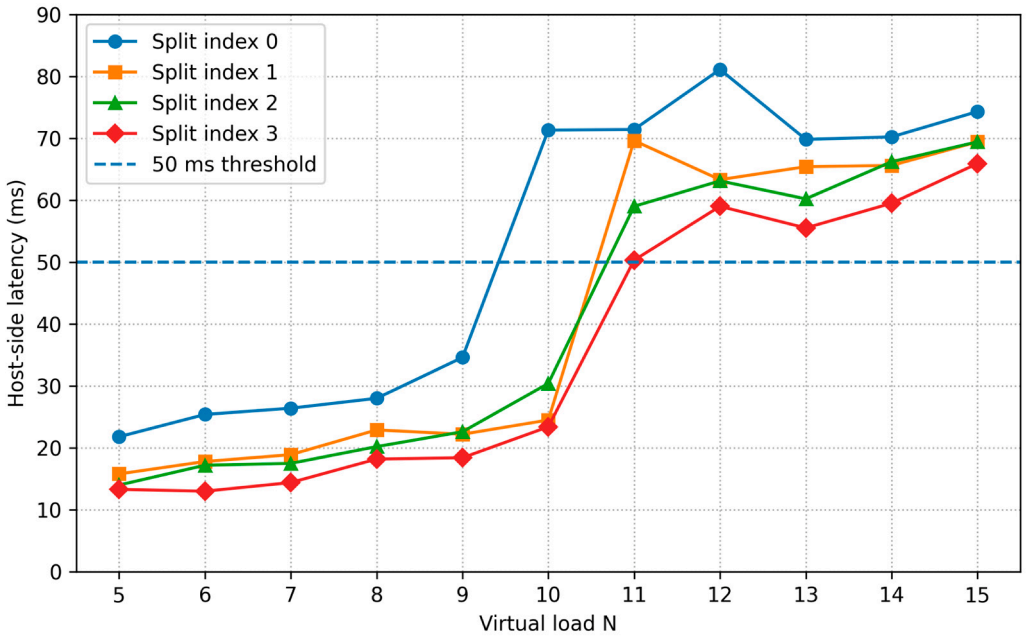


Figure 3. Host-side latency versus virtual load N for split index 0 to 3. The dashed line indicates the 50 ms real-time threshold.

3.3. Frame Rate and Maximum Supported N

Because the camera operated at 20 FPS, one new frame arrived every 50 ms. Therefore, the overall frame rate could be maintained only when both the edge-side and host-side constraints were satisfied. Under this interpretation, index 0 could no longer sustain the target real-time behavior once the host-side latency exceeded 50 ms around $N = 10$, meaning that its practically supported load was limited to approximately $N = 9$. Index 1 and index 2 maintained host-side latency below 50 ms up to $N = 10$ and preserved edge-side operation close to 20 FPS, indicating that these two configurations provided the best balance for sustaining the overall frame rate. As shown in Figure 3, index 0 degraded earlier than the intermediate split configurations as N increased.

Although index 3 showed relatively low host-side latency at $N = 10$, its edge-side total latency was already about 53.1 ms. This means that the host side alone did not determine the overall performance for index 3. In other words, index 3 reduced the communication and host-side burden, but this benefit was offset by the higher preprocessing cost on the Raspberry Pi. As a result, the best split positions were not the shallowest or deepest ones, but the intermediate configurations, namely index 1 and 2.

4. Discussion

The experimental results suggest that the split point must be selected by considering both communication reduction and edge-side computational overhead, rather than minimizing only one of them. In the developed system, split index 0 relied heavily on host-side processing after transmission of the raw frame, and this configuration lost its real-time advantage as the virtual load increased. In contrast, split index 1 and 2 reduced the amount of transmitted data and part of the host-side preprocessing cost without imposing excessive computation on the Raspberry Pi. This explains why these intermediate split positions provided the most favorable balance for sustaining the overall frame rate.

The behavior of split index 3 clarifies this trade-off. Although deeper preprocessing reduced the amount of data sent to the host and could keep host-side latency relatively low, the Raspberry Pi had to construct a temporal tensor before transmission. In our measurements, this edge-side cost increased the Pi-side latency to about 53.1 ms and reduced the edge-side frame rate to about 18.9 FPS, which means that host-side improvement alone was not sufficient to guarantee better overall performance. Therefore, a deeper split is not always preferable, even when it appears beneficial from the host-side perspective.

The main evaluation target of this study was the overall frame rate under virtual multi-camera load, rather than only host-side latency. The server implementation separately monitored physical receive rate, original-task processing rate, and processing completion rate for all replicated tasks, and this design is consistent with evaluating throughput under virtual concurrent inputs. From this viewpoint, the best split point is the one that preserves stable throughput under increasing N, not necessarily the one with the lowest latency at a single stage.

This study has limitations. First, the multi-camera condition was emulated by duplicating one input stream into N virtual tasks on the server side, rather than by deploying many physical Raspberry Pi devices. This design was useful for controlled and reproducible load evaluation, but it did not fully reproduce the effects of independent camera timing, network contention among multiple real devices, or device-to-device variation. Second, the analysis was limited to the split index of 0 to 3, and deeper or more adaptive split strategies were not examined. This necessitates more experiments with multiple physical edge devices, evaluation under different camera settings and network conditions, and dynamic control of the split position according to the current latency or throughput of the system.

https://doi.org/10.3390/engproc2026159018

5. Conclusions

In this study, we evaluated edge–host split processing for a Gaze360-based gaze estimation system using a Raspberry Pi 5 and a host PC. We defined four split points in the processing pipeline and investigated their influence on system performance under virtual multi-camera load. The evaluation focused mainly on the overall frame rate, while host-side latency and edge-side processing behavior were used as supporting indicators.

The results showed that the intermediate split positions, namely split index 1 and split index 2, provided the most effective balance between communication reduction and edge-side computational overhead. Split index 0 degraded earlier because of the high host-side processing cost after raw-frame transmission, whereas split index 3 increased the burden on the Raspberry Pi due to temporal tensor construction before transmission. These findings indicate that neither the shallowest nor the deepest split was optimal for maintaining real-time performance, and that intermediate split processing was the most practical choice in the evaluated setting.

In this study, the multi-camera condition was emulated by duplicating one input stream into N virtual tasks on the server side, rather than by deploying multiple physical edge devices. Further study is required to experiment with multiple physical edge devices, evaluation under different network and camera conditions, and adaptive control of the split point according to the current system state. Nevertheless, the developed framework enabled reproducible evaluation of throughput under controlled load.

Author Contributions: Conceptualization, Y.K. and M.N.; methodology, Y.K.; software, Y.K.; validation, Y.K. and M.N.; formal analysis, Y.K.; investigation, Y.K.; resources, M.N.; data curation, Y.K.; writing—original draft preparation, Y.K.; writing—review and editing, Y.K. and M.N.; visualization, Y.K.; supervision, M.N.; project administration, M.N. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are not publicly available.

Acknowledgments: During the preparation of this manuscript/study, the authors used DeepL Translator (free web version; DeepL SE, Cologne, Germany; accessed April 2026) for the purposes of translating the Japanese draft into English, and ChatGPT (GPT-5.4 Thinking; OpenAI, San Francisco, CA, USA; accessed April 2026) for the purposes of improving the grammar, phrasing, and readability of the English text, formatting the manuscript according to the conference requirements, and generating code for selected functions of the experimental programs. The authors have reviewed and edited the output and take full responsibility for the content of this publication.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Kellnhofer, P.; Recasens, A.; Mukhoti, S.; Turan, C.; Paredes, A.; Isola, P.; Torralba, A. Gaze360: Physically Unconstrained Gaze Estimation in the Wild. In Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), Seoul, Republic of Korea, 27 October–2 November 2019; pp. 6912–6921. [CrossRef]
2. Kang, Y.; Kim, J.; Kim, H. Latency and Cost-Efficient Split Point Identification for Multi-Tier DNN Partitioning. *IEEE Access* **2024**, *12*, 90253–90268. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.