

An Empirical Study of ML-Based Rolling-Shutter OCC Decoding on MCU-Class Receiver Platforms

Hongbo Liu, Juan F. Gutiérrez, Negar Halakou, and Andres Gomez

Institute of Computer and Network Engineering

Technische Universität Braunschweig

Braunschweig, Germany

{hongbo.liu, juan-felipe.gutierrez-gomez, negar.halakou, andres.gomez}@tu-braunschweig.de

Abstract—Rolling-shutter optical camera communication (OCC) has been studied as a camera-based approach for delivering information from indoor infrastructure to compact mobile platforms. Prior studies have separately examined machine-learning-based (ML-based) OCC decoding and OCC-enabled indoor mobile systems. However, deploying ML-based rolling-shutter OCC decoding on microcontroller-class (MCU-class) receiver platforms remains challenging under tight latency, energy, memory, communication, and toolchain constraints. This paper studies ML-based rolling-shutter OCC decoding as a deployment problem. We systematically compare three representative deployment solutions—one remote and two local—with TinyML as the local decoding approach. Our results show clear trade-offs among the studied solutions: an MCU implementation with a dedicated convolutional neural network (CNN) accelerator achieves 85% bit accuracy with 5.68 ms inference latency and 44.95 ms end-to-end pipeline latency, demonstrating onboard feasibility in an embedded mobile system, but at the cost of greater hardware dependence.

Index Terms—visible light communication (VLC), optical camera communication (OCC), rolling-shutter OCC, ML-based decoding, TinyML, MCU, deployment trade-offs

I. INTRODUCTION

Compact embedded and mobile platforms often operate in indoor environments where nearby infrastructure can provide short, local, and spatially bounded information. OCC is attractive for this setting because visible light sources are physically tied to locations and can act as local transmitters, while many embedded platforms already carry cameras. In this work, we use OCC as an infrastructure-to-platform information path for short optical messages, not as a replacement for general-purpose wireless networking [1].

Rolling-shutter OCC provides a practical reception mechanism for such camera-based optical messages. By exploiting the row-wise exposure of CMOS image sensors, temporal light modulation can be observed as spatial stripe patterns in captured frames [2].

Decoding rolling-shutter OCC patterns, however, becomes difficult on compact embedded receivers. The captured stripe pattern depends on exposure, sampling asynchrony, motion, blur, distance, background illumination, and the visible transmitter region. ML-based decoders can improve robustness under such non-ideal conditions, but they also introduce model size, memory footprint, runtime, energy, and toolchain costs. On MCU-class receiver platforms, these costs are not sec-

ondary implementation details; they determine whether the decoder can be used in a practical OCC-to-command pipeline.

This paper therefore treats ML-based rolling-shutter OCC decoding as a deployment problem. Given a learned decoder and a compact embedded receiver, the system designer must decide where the decoder should run. Remote execution can reduce onboard computation, but introduces wireless transfer cost, network latency, and host dependence. Local CPU execution preserves locality, but may exceed MCU-side latency and memory limits. Local accelerator-assisted execution can make onboard decoding feasible, but depends on model structure, supported operators, quantization format, memory layout, and deployment toolchain compatibility.

We study this trade-off through an empirical comparison of three representative deployment solutions: remote decoding, local TinyML decoding on a general-purpose MCU, and local decoding on an MCU with a dedicated CNN accelerator. Across the three solutions, the OCC decoding task is kept fixed: an 80×80 single-channel rolling-shutter OCC input is decoded into a 20-bit output and mapped to a command-level interface. The compact robot in our prototype is used only as a command-consumption endpoint; localization, navigation, and motion-control optimization are outside the scope of this work.

This paper makes three contributions. First, it formulates ML-based rolling-shutter OCC decoding as a deployment problem for MCU-class receiver platforms rather than only an accuracy-driven receiver-design problem. Second, it implements and compares three representative execution solutions for the same decoding task: remote execution, local CPU execution, and local accelerator-assisted execution. Third, it characterizes the resulting accuracy, latency, energy, communication, and hardware-dependence trade-offs. Our results show that CPU-only MCU inference is runnable but too slow for the studied pipeline, remote execution is dominated by wireless transfer cost, and accelerator-assisted local execution achieves 85.0% bit accuracy, 5.68 ms inference latency, and 44.95 ms end-to-end pipeline latency, demonstrating onboard feasibility at the cost of stronger platform and toolchain dependence.

The remainder of this paper is organized as follows. Section II reviews related work; Sections III and IV describe the system design and implementation; Sections V and VI present the experimental evaluation and analysis; and Section VII concludes the paper.

II. RELATED WORK

A. Rolling-Shutter and ML-Assisted OCC Decoding

Rolling-shutter OCC uses the row-wise exposure of CMOS image sensors to convert temporal light modulation into spatial stripe patterns within image frames [1], [2]. This enables commodity cameras to receive optical signals beyond the nominal frame rate, but also makes decoding sensitive to exposure, sampling asynchrony, transmitter size, distance, background illumination, blur, and receiver motion [2]–[4]. Traditional receivers often rely on handcrafted pipelines for region selection, stripe-profile extraction, thresholding, synchronization, and symbol reconstruction [3], [4]. These methods can be efficient under clear and stable stripe patterns, but their assumptions become fragile when the receiver moves or the stripe pattern is degraded.

ML-assisted OCC decoding has therefore been explored for symbol decoding, LED detection, threshold prediction, equalization, preamble detection, signal clock and exposure estimation, mobility-aware reception, and complex-background signal extraction [5]–[10]. These studies establish ML-based OCC decoding as prior work rather than the novelty of this paper. Our focus is different: we study how an ML-based rolling-shutter OCC decoder should be deployed under MCU-class receiver constraints, where memory, latency, energy, communication, and toolchain costs become part of the design problem.

B. Light-to-Camera Systems, TinyML, and Execution Placement

Visible light positioning (VLP), visible light communication (VLC), and related light-to-camera front ends have already been integrated into mobile robot systems for localization, navigation, mapping, calibration, and sensor fusion [11]–[14]. These systems show that optical camera links can operate on robotic platforms and with onboard-offboard processing paths. In contrast, this paper does not claim novelty in robot localization or navigation. We use the robot only as a command-consumption endpoint, while the studied object is the deployment of the ML-based rolling-shutter OCC decoder itself.

TinyML and embedded inference provide the platform context for local ML-OCC decoding. Frameworks such as TensorFlow Lite Micro show that neural-network inference can run on MCU-class devices, but also expose constraints in static memory allocation, operator support, quantization, kernel efficiency, data layout, and low-level runtime integration [15]. More broadly, mobile offloading, edge inference, and collaborative intelligence show that execution can be shifted between a mobile device and a more capable remote resource, with the best placement depending on computation cost, communication cost, network condition, hardware capability, and energy budget [16]–[18]. These works motivate our comparison among remote execution, constrained local MCU execution, and accelerator-assisted local MCU execution. The OCC-specific question studied here is how rolling-shutter

OCC decoding from camera-derived stripe patterns should be deployed on MCU-class receiver platforms.

III. SYSTEM DESIGN

We study ML-based rolling-shutter OCC decoding as a deployment problem on MCU-class receiver platforms. The optical task, decoder input-output interface, and command-level output are kept fixed, while the decoder execution placement is varied. The three evaluated placements are remote execution, local CPU execution on a general-purpose MCU, and local accelerator-assisted execution on an MCU with a dedicated CNN accelerator.

A. Scenario and Scope

The study setup consists of a modulated light source, a camera-equipped embedded receiver, and a compact robot command endpoint. The light source transmits a fixed-length optical payload. The receiver observes the source with a rolling-shutter camera, converts the captured stripe pattern into a decoder input, recovers the transmitted bit sequence, and maps the decoded result to a command output consumed by the robot controller.

The target task starts after the light source is visible to the receiver camera. Light-source discovery, robot localization, path planning, and motion-control optimization are outside the scope of this work. The robot is used only as a command-consumption endpoint; the design variable studied here is where and how the ML-based OCC decoder is executed.

B. Pipeline and Decoder Interface

Figure 1 shows the OCC-to-command pipeline. The light source emits a binary payload through visible-light modulation, and the rolling-shutter camera captures it as a stripe-like image pattern. The receiver then preprocesses the captured frame, runs the ML decoder, validates the decoded bits, maps them to a command, and sends the command to the robot controller.

Across all deployment placements, the decoder is treated as a component with a fixed interface. Its input is a compact single-channel representation of the rolling-shutter OCC pattern, and its output is a fixed-length OCC bit sequence. In our implementation, this interface is instantiated as an 80×80 single-channel input and a 20-bit output. The deployed binaries, runtimes, quantization formats, and memory layouts may differ across platforms, but the input semantics, output semantics, and downstream command mapping remain fixed.

C. Execution Placements

Remote execution. In the remote placement, the embedded receiver captures the OCC frame and sends camera-derived input to a remote host. The host executes the ML decoder and returns the decoded bit sequence to the receiver, which then performs validation and command output. This placement reduces onboard inference cost, but introduces a wireless communication path and host dependence.

Local CPU execution. In the local CPU placement, the receiver keeps the full decoding path onboard. After capture

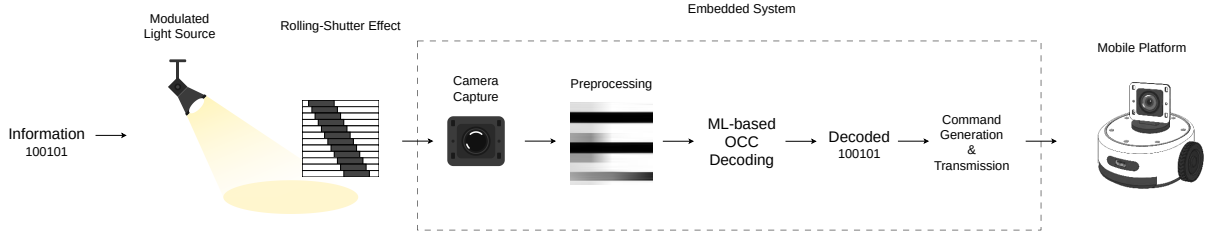


Fig. 1. Overview of the evaluated OCC-to-command pipeline. The receiver converts a rolling-shutter OCC signal into a command output through camera capture, preprocessing, ML-based decoding, and command transmission.

and preprocessing, the decoder runs on the general-purpose MCU through a TinyML runtime, followed by validation and command output. This placement removes the remote host from the inference path, but the decoder must fit the MCU’s memory, operator-support, and runtime constraints.

Local accelerator-assisted execution. In the accelerator-assisted placement, the receiver also keeps the decoding path local, but maps the computation-dominant inference stage to a dedicated CNN accelerator. The main MCU handles capture, preprocessing, validation, and command output, while the accelerator executes the decoder. This placement targets low-latency onboard inference, but depends on the accelerator’s supported layers, quantization format, memory layout, and deployment toolchain.

IV. IMPLEMENTATION

This section describes the prototype implementation used for the empirical comparison. We focus on implementation details that affect the measured OCC-to-command pipeline: receiver platforms, decoder runtimes, input formats, and command output.

A. Platforms and System Roles

The prototype consists of a modulated light source, two MCU-class camera receiver platforms, a remote host, and a compact robot command endpoint. Across the evaluated placements, the light source and command interface are fixed, while the receiver platform, runtime, and execution path vary.

The light source uses a high-power RGBW LED module driven by a custom voltage-controlled current driver. A TI DAC1282S085 digital-to-analog converter provides the modulation input, controlled by a TI CC3200 LaunchPad. The transmitter hardware and modulation setting are fixed throughout the evaluation. The command endpoint is a Pololu 3pi+ 2040 robot, which receives validated command-level output over UART. Localization, navigation, and motion planning are outside the scope of this implementation.

Table I summarizes the platform roles, and Figure 2 shows the implemented setup.

B. Camera Front Ends and Input Preprocessing

Both receiver platforms capture RGB565 frames at 160×120 pixels. The ESP32-S3-EYE uses an OV2640 camera through the Zephyr video interface and serves both the

TABLE I
PLATFORM ROLES IN THE IMPLEMENTATION.

Platform	Role
ESP32-S3-EYE	Remote capture; local CPU TinyML
MAX78000CAM02	Local CNN-accelerated decoding
Remote host	Off-board ML inference
Pololu 3pi+ 2040	UART command endpoint

remote and local CPU placements. The MAX78000CAM02 uses a GC0308 camera through the MAX78000 camera interface.

Camera parameters are configured through platform-specific drivers and are fixed during each measurement after obtaining a visible rolling-shutter stripe pattern. Because the OV2640 and GC0308 expose different low-level controls, the comparison does not assume physically identical camera settings across platforms.

Each captured frame is converted to the same decoder input: an 80×80 single-channel representation of the rolling-shutter OCC pattern. In the ESP32-S3 local CPU placement, preprocessing converts RGB565 pixels to grayscale, resizes the frame, normalizes values, and quantizes the input for TensorFlow Lite Micro. In the remote placement, the ESP32-S3-EYE transmits the 160×120 RGB565 frame to the host, which applies the preprocessing before inference. In the MAX78000 placement, preprocessing converts and resizes the frame, then packs it into the signed int8 layout required by the CNN accelerator.

C. Dataset and Decoder Preparation

The ML decoder is trained on a collected rolling-shutter OCC dataset with 500 manually labeled samples. Each sample is represented as an 80×80 single-channel OCC pattern with a 20-bit binary label. The dataset is split into approximately 80% training samples and 20% held-out test samples, with no data augmentation. We use this dataset to instantiate fixed decoder variants for deployment comparison, while broader robustness across unseen OCC conditions is left to larger multi-condition datasets.

The decoder is a compact CNN that maps the 80×80 input to a 20-bit binary sequence. The remote placement runs the decoder in the host software environment. The



Fig. 2. Implemented prototype setup with robot command-output validation.

ESP32-S3 placement uses a quantized TensorFlow Lite model compiled into firmware through TensorFlow Lite Micro. The MAX78000 placement uses an accelerator-compatible CNN variant generated through the AI8X/MAX78000 toolchain.

The deployed binaries differ across platforms because the runtimes, operator support, quantization formats, and memory layouts are platform-specific. However, all versions preserve the same decoding task: mapping a preprocessed rolling-shutter OCC image to a fixed-length bit sequence.

D. Runtime and Command Output

The three placements implement the same decoder interface but differ in where inference runs. In the remote placement, the ESP32-S3-EYE sends captured RGB565 frames to the remote host over Wi-Fi using UDP. The host reconstructs the frame, applies preprocessing, runs the ML decoder, and returns the decoded OCC bit string for command generation.

In the local CPU placement, the decoder is converted to a TensorFlow Lite Micro compatible format and compiled into the ESP32-S3 firmware. Inference runs directly on the MCU using a statically allocated tensor arena.

In the local accelerator-assisted placement, the decoder is adapted to the MAX78000 CNN accelerator toolchain. The firmware prepares the 80×80 input, invokes accelerator inference, retrieves the output logits, and converts them into OCC bits. This path must satisfy the accelerator's layer, quantization, input-layout, and memory-mapping constraints.

After decoding, the receiver maps the bit sequence to a command byte and sends it to the robot over UART. The same UART interface is used for all placements, so downstream command behavior does not depend on where the decoder runs.

V. EXPERIMENTAL EVALUATION

We evaluate ML-based rolling-shutter OCC decoding as an empirical deployment study on MCU-class receiver platforms. The evaluation compares three placements: remote execution on a host through ESP32-S3-EYE capture, local CPU execution on ESP32-S3, and local accelerator-assisted execution on MAX78000. All placements preserve the same task: an 80×80 single-channel OCC input is decoded into a 20-bit output and mapped to the same command interface. Accuracy is evaluated offline on the held-out dataset split, while latency and energy are measured on the running receiver pipelines using the Otii setup and platform-side timing markers.

Power and energy measurements are receiver-side; remote-host energy is excluded.

A. Decoder Footprint and Accuracy

Tables II and III report the deployed model footprint and bit-level accuracy; they confirm that all placements execute the same decoding task before latency and energy are compared.

TABLE II
ML DECODER MODEL FOOTPRINT IN THE EVALUATED DEPLOYMENT SOLUTIONS.

Deployment solution	Model footprint	Measurement note
Remote	359.5 KB	TFLite model
Local CPU	57.0 KB	TFLite model
Local accelerator	51.9 KB	CNN weight memory

TABLE III
BIT-LEVEL ACCURACY OF THE ML DECODER MODELS USED IN THE EVALUATED DEPLOYMENT SOLUTIONS.

Deployment solution	Bit accuracy	Note
Remote	90.4%	Host-side TFLite model
Local CPU	89.1%	TFLite model on ESP32-S3-EYE
Local accelerator	85.0%	AI8X model on MAX78000CAM02

B. Local Inference Latency

Table IV isolates local MCU-side inference latency, showing that CPU-only inference is runnable but remains far slower than accelerator-assisted execution.

TABLE IV
INFERENCE LATENCY OF MCU LOCAL DEPLOYMENT SOLUTIONS. PORTENTA H7 IS INCLUDED AS A SUPPLEMENTAL CPU-ONLY BENCHMARK ON A STRONGER MCU-CLASS PLATFORM.

Deployment solution	Latency
Local CPU	32.674 s
Portenta H7 CPU	11.083 s
Local accelerator	5.68 ms

C. Pipeline Latency and Energy

Table V reports complete OCC-to-command latency, average power, and energy, including camera capture, preprocessing, decoding or communication, postprocessing, and UART command output.

TABLE V

PIPELINE-LEVEL LATENCY, AVERAGE POWER, AND ENERGY. TOTAL LATENCY COVERS THE FULL OCC-TO-COMMAND CYCLE; THE PHASE BREAKDOWN REPORTS DOMINANT MEASURED PHASES ONLY, OMITTING SUB-RESOLUTION POSTPROCESSING AND UART-OUTPUT PHASES.

Deployment solution	Pipeline phase	Latency	Avg. power	Energy
Remote	Camera capture	234.73 ± 52.98 ms	471.08 ± 34.03 mW	108.8970 ± 24.4809 mJ
Remote	UDP transmission	1640.69 ± 1.68 ms	632.77 ± 1.92 mW	1038.1772 ± 3.4501 mJ
Remote	Waiting for remote processing	29.99 ± 29.42 ms	613.67 ± 22.72 mW	17.7793 ± 13.5895 mJ
Remote	UDP reception	5.31 ± 0.46 ms	640.68 ± 13.86 mW	3.3985 ± 0.2894 mJ
Remote	Total	1910.72 ± 53.00 ms	611.54 ± 5.00 mW	1168.2520 ± 24.7302 mJ
Local CPU	Camera capture	100.32 ± 5.38 ms	398.84 ± 1.33 mW	40.0075 ± 2.0022 mJ
Local CPU	Preprocessing	6.95 ± 0.23 ms	463.81 ± 4.83 mW	3.2211 ± 0.0987 mJ
Local CPU	Local inference	32674.22 ± 0.42 ms	488.80 ± 0.88 mW	15971.0999 ± 28.9007 mJ
Local CPU	Total	32781.49 ± 5.35 ms	488.52 ± 0.90 mW	16014.3285 ± 27.6855 mJ
Local accelerator	Camera capture	35.56 ± 0.50 ms	64.15 ± 0.12 mW	2.2811 ± 0.0338 mJ
Local accelerator	Preprocessing	3.71 ± 0.45 ms	65.00 ± 0.31 mW	0.2412 ± 0.0297 mJ
Local accelerator	Local inference	5.68 ± 0.47 ms	78.21 ± 3.46 mW	0.4441 ± 0.0475 mJ
Local accelerator	Total	44.95 ± 0.21 ms	66.01 ± 0.51 mW	2.9673 ± 0.0301 mJ

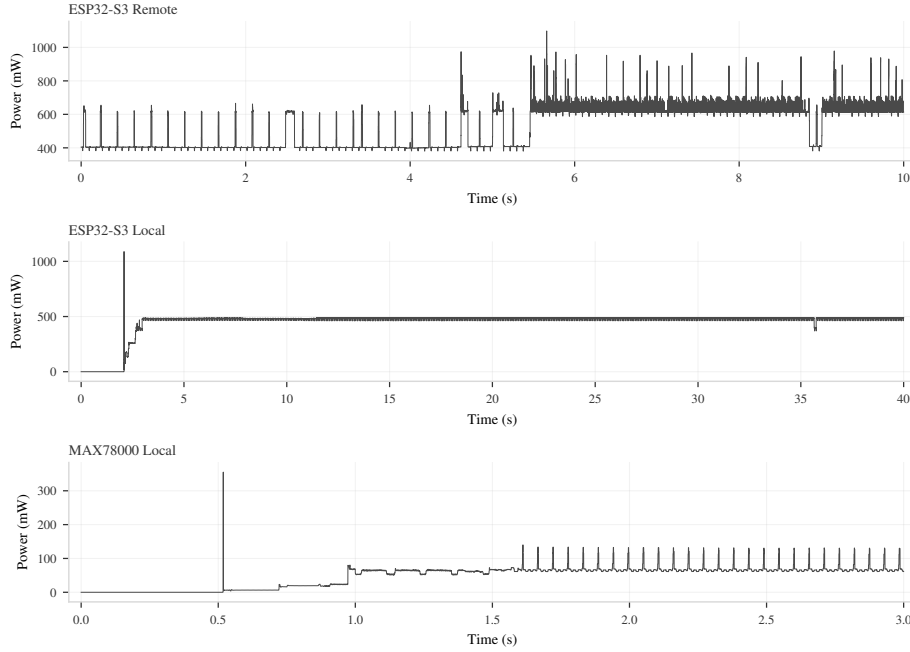


Fig. 3. Representative Otii power traces for remote, local CPU, and local accelerator-assisted execution. The traces illustrate the distinct runtime bottlenecks of the three pipelines; time and power axes are scaled independently for readability.

Figure 3 shows representative Otii power traces for the three placements, validating that the measurements come from continuous runtime loops.

The pipeline results show different bottlenecks: remote execution is dominated by wireless transfer, local CPU execution by MCU inference, and accelerator-assisted execution by capture and preprocessing after inference becomes short.

VI. ANALYSIS

The evaluation shows that execution placement changes the dominant cost of ML-based rolling-shutter OCC decoding. Remote execution avoids MCU-side inference but is dominated by wireless transfer and host dependence. Local CPU execution preserves locality, but inference dominates the OCC-to-command cycle even on the supplemental Portenta

H7 benchmark. Accelerator-assisted execution changes the feasible design space: the MAX78000 path reduces inference to a short pipeline stage, making onboard decoding feasible while exposing other costs such as camera capture and preprocessing.

These results show that decoding accuracy alone does not determine deployability. The relevant question is which placement provides a usable accuracy–latency–energy trade-off under embedded receiver constraints, as summarized in Figure 4. Remote execution provides flexibility but adds infrastructure dependence; CPU-only local execution provides portability but remains inference-bound; accelerator-assisted local execution provides the strongest latency result but increases platform and toolflow dependence.

The study should be read as a platform-realistic deployment

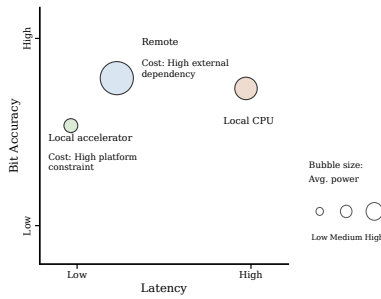


Fig. 4. Qualitative summary of deployment trade-offs among remote, local CPU, and local accelerator solutions.

comparison rather than a controlled model-equivalence or robustness benchmark. The three placements preserve the same decoding task, but platform-specific cameras, runtimes, quantization formats, and model variants can affect both accuracy and runtime; therefore, the reported values characterize the evaluated deployment configurations rather than placement alone. Accuracy is measured on a single held-out split of the collected 500-sample dataset, so the differences among decoder accuracies should not be over-interpreted, and broader robustness would require larger multi-condition datasets and repeated evaluation. The remote baseline transmits full 160×120 RGB565 frames over UDP; cropped, compressed, or preprocessed offloading would shift the communication cost. Finally, command-level reliability could be improved with full-sequence validation, rejection, or checksum mechanisms.

VII. CONCLUSION

This paper presented an empirical study of ML-based rolling-shutter OCC decoding on MCU-class receiver platforms by comparing remote, local CPU, and local accelerator-assisted execution. The results show that each placement exposes a different bottleneck: wireless transfer for remote execution, inference for CPU-only local execution, and camera capture and preprocessing when accelerator-assisted inference is no longer the dominant cost. In the evaluated prototype, accelerator-assisted execution achieves 85.0% bit accuracy, 5.68 ms inference latency, and 44.95 ms end-to-end latency, demonstrating practical onboard feasibility while increasing hardware and toolchain dependence. Overall, the results support treating ML-based rolling-shutter OCC decoding as a deployment problem in which accuracy, latency, energy, communication cost, and hardware constraints must be considered jointly.

ACKNOWLEDGEMENT

AI Acknowledgement. ChatGPT was used in the preparation of this manuscript for editing, grammar checking, and information retrieval. All LLM-assisted content was reviewed and verified by the authors. All substantive ideas, analyses, interpretations, and conclusions are the product and responsibility of the authors.

REFERENCES

- [1] N. Saeed, S. Guo, K.-H. Park, T. Y. Al-Naffouri, and M.-S. Alouini, "Optical camera communications: Survey, use cases, challenges, and future trends," *Physical Communication*, vol. 37, p. 100900, 2019.
- [2] H.-Y. Lee, H.-M. Lin, Y.-L. Wei, H.-I. Wu, H.-M. Tsai, and K. C.-J. Lin, "RollingLight: Enabling line-of-sight light-to-camera communications," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, 2015, pp. 167–180.
- [3] W. LIU and Z. Xu, "Some practical constraints and solutions for optical camera communication," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 378, no. 2169, p. 20190191, 2020.
- [4] B. W. Kim, J.-H. Lee, and S.-Y. Jung, "Transition-based data decoding for optical camera communications using a rolling shutter camera," *Curr. Opt. Photon.*, vol. 2, no. 5, pp. 422–430, 2018.
- [5] T. L. Pham, H. Nguyen, T. Nguyen, and Y. M. Jang, "A novel neural network-based method for decoding and detecting of the DS8-PSK scheme in an OCC system," *Applied Sciences*, vol. 9, no. 11, p. 2242, 2019.
- [6] K. Yu, J. He, and Z. Huang, "Decoding scheme based on CNN for mobile optical camera communication," *Appl. Opt.*, vol. 59, no. 23, pp. 7109–7113, 2020.
- [7] H. Nguyen, V. L. Nguyen, D. H. Tran, and Y. M. Jang, "Rolling shutter OFDM scheme for optical camera communication considering mobility environment based on deep learning," *Applied Sciences*, vol. 12, no. 16, p. 8269, 2022.
- [8] O. I. Younus, N. Bani Hassan, Z. Ghassemlooy, P. A. Haigh, S. Zvanovec, L. N. Alves, and H. L. Minh, "Data rate enhancement in optical camera communications using an artificial neural network equaliser," *IEEE Access*, vol. 8, pp. 42 656–42 665, 2020.
- [9] C. Jurado-Verdu, V. Guerra, J. Rabadan, and R. Perez-Jimenez, "Deep learning for signal clock and exposure estimation in rolling shutter optical camera communication," *Opt. Express*, vol. 30, no. 12, pp. 20 261–20 277, 2022.
- [10] R. Xiao, L. Zhao, F. Qian, L. Yang, and J. Han, "Practical optical camera communication behind unseen and complex backgrounds," in *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services*, 2024, pp. 113–126.
- [11] W. Guan, S. Chen, S. Wen, Z. Tan, H. Song, and W. Hou, "High-accuracy robot indoor localization scheme based on robot operating system using visible light positioning," *IEEE Photonics Journal*, vol. 12, no. 2, pp. 1–16, 2020.
- [12] W. Guan, L. Huang, S. Wen, Z. Yan, W. Liang, C. Yang, and Z. Liu, "Robot localization and navigation using visible light positioning and SLAM fusion," *Journal of Lightwave Technology*, vol. 39, no. 22, pp. 7040–7051, 2021.
- [13] J. Huang, S. Wen, W. Liang, and W. Guan, "VWR-SLAM: Tightly coupled SLAM system based on visible light positioning landmark, wheel odometer, and RGB-D camera," *IEEE Transactions on Instrumentation and Measurement*, vol. 72, pp. 1–12, 2023.
- [14] R. Amsters, E. Demeester, N. Stevens, and P. Slaets, "Calibration of visible light positioning systems with a mobile robot," *Sensors*, vol. 21, no. 7, p. 2394, 2021.
- [15] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang, P. Warden, and R. Rhodes, "TensorFlow Lite Micro: Embedded machine learning for TinyML systems," in *Proceedings of Machine Learning and Systems*, vol. 3, 2021, pp. 800–811.
- [16] E. Cuervo, A. Balasubramanian, D.-K. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, 2010, pp. 49–62.
- [17] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: elastic execution between mobile device and cloud," in *Proceedings of the Sixth Conference on Computer Systems*, 2011, pp. 301–314.
- [18] E. Li, Z. Zhou, and X. Chen, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," in *Proceedings of the 2018 Workshop on Mobile Edge Communications*, 2018, pp. 31–36.