The Development of a Hybrid Raspbian Operating System (OS) for Scientific Exploration of Extraterrestrial Planets

Zarif Bin Akhtar
Department of Computer Engineering (CoE)
Faculty of Engineering (FE)
American International University-Bangladesh (AIUB), Dhaka, Bangladesh
E-mail: zarifbinakhtarg@gmail.com; zarifbinakhtar@ieee.org

***Abstract*—This research investigates the development of a custom hybrid operating system (OS) for a Mars rover experimental prototype using the Raspberry Pi platform. Focusing on operating system optimization, the work enhances computational efficiency, real-time responsiveness, and AI integration. Key innovations include overclocking (boosting CPU performance by 28%), custom threading (reducing task scheduling latency by 22%), and networking improvements for stable remote operation. Codec refinements and framework adaptations improved real-time video analysis throughput by 30%. Integration of a Power-over-Ethernet (PoE) HAT enhanced thermal regulation and stabilized system runtime. Experimental results show the customized OS effectively supports intensive tasks such as image processing, sensor data acquisition, and edge AI workloads. The findings demonstrate a scalable, modular OS framework for real-time vision systems and intelligent robotics in resource-constrained environments.**

Keywords-Computing; Artificial Intelligence (AI); Computer Vision; Deep Learning (DL); Image Processing; Machine Learning (ML); Operating Systems (0S); Robotics; Scientific Computing; Space Exploration

I. INTRODUCTION

Operating systems (OS) are fundamental to managing computational resources and enabling interaction between applications and hardware. While traditional OS architectures—such as monolithic, microkernel, hybrid, and exokernel-have been optimized over decades [1,2,3], specialized applications in real-time and high-performance environments continue to demand further innovations [3,4,5]. Recent research emphasizes the importance of modularity, fault tolerance, and AI-driven optimizations for adaptive and efficient systems [5,6,7]. This manuscript focuses on the design, implementation, and evaluation of a custom hybrid OS for a Raspberry Pi–based Mars rover prototype.

At the same time, addressing challenges in computational efficiency, real-time task execution, and AI integration. Building upon prior studies in distributed systems and edge computing [7, 8, 9], the research follows a three-phase methodology: analyzing kernel and scheduling architectures, implementing system-level optimizations (e.g., custom threading, memory management), and benchmarking performance through real-world tasks such as image processing, sensor data acquisition, and real-time video analysis. By combining software design innovations with experimental validation, this work advances the development of OS frameworks tailored for intelligent robotics and resource-constrained environments.

II. DESIGN APPROACHES AND EXPERIMENTAL ANALYSIS

A. System Architecture and Design

The custom operating system (OS) was developed on the Raspberry Pi 4 platform, utilizing its ARM Cortex-A72 CPU and VideoCore VI GPU to achieve real-time image processing and AI-based computation. A lightweight Linux kernel was selected and modified for high-performance computing and robust remote accessibility within a 5V power-efficient system architecture [6, 7].

B. Kernel Customization and Optimization

The kernel was compiled based on Linux Kernel 6.x, incorporating specific enhancements:

- **Real-Time Multitasking:** Enabled through preemptive scheduling and optimized interrupt handling.
- **Memory Management:** Introduced a slab allocator designed for rapid memory access suited to image processing workloads.
- **Thread Prioritization:** Implemented at the kernel level to enhance concurrent task execution efficiency.
- **Thermal and Power Management:** Integrated Power-over-Ethernet (PoE) features to ensure consistent thermal regulation and power delivery without additional sensory modules.

C. Image Processing Framework

The OS supports a native computer vision pipeline, structured as follows:

- **Acquisition:** Real-time image capture via Pi Camera Module v2.
- **Preprocessing:** Noise filtering, grayscale conversion, and edge detection.
- **Feature Extraction:** CNN-based object detection using TensorFlow Lite integrated into user-space drivers.
- **Post-processing:** Annotation and transmission of processed frames through WebRTC for remote monitoring.

OpenCV 4.x libraries were kernel-linked to optimize processing speeds while maintaining a lightweight system footprint.

D. Remote Connectivity and Control

Remote system interaction was achieved through:

- **SSH Access:** Secure system login using OpenSSH with RSA-based authentication.
- **Web Interface:** A lightweight Flask server providing real-time visualization of image processing outputs.
- **Streaming:** WebRTC streaming enabled low-latency real-time video transmission.

Custom SSID and static IP configurations were implemented to support reliable autonomous remote operations.

E. Experimental Setup

Hardware Components:

- Raspberry Pi 4 (4GB RAM)
- Pi Camera Module v2
- PoE HAT for power and thermal management

Software Environment:

- Customized Linux Kernel 6.x
- OpenCV 4.x and TensorFlow Lite

Performance Metrics:

- Image Processing Latency (ms)
- System Response Time (ms)
- CPU and Memory Utilization (%)

Testing involved varying workloads and environmental conditions to simulate real-world operation scenarios.

F. Performance Evaluation

TABLE I. LATENCY ANALYSIS

Parameter	Value
Frame Processing Time (640x480)	45 ms (Average)
System Command Response Time	< 100 ms

The custom OS achieved 45 ms per frame at VGA resolution, significantly enhancing real-time vision capabilities.

TABLE II. RESOURCE Utilization

Metric	Peak Value	Average Value
CPU Utilization	78%	65%
Memory Utilization	70%	65%

The system maintained stable resource usage, even under high computational loads.

TABLE III. COMPARATIVE Benchmarking

Parameter	Custom OS	Raspbian OS	Improvement
Frame Processing Latency	45 ms	65 ms	30% faster
Remote Command Latency	<100 ms	~140 ms	29% faster
CPU Peak Utilization	78%	90%	13% better

The custom OS consistently outperformed the standard Raspbian OS in latency and resource efficiency.

The designed system demonstrated:

- Enhanced real-time image processing capabilities.
- Stable and efficient resource management under diverse workloads.
- Seamless and secure remote operation through optimized networking.

The improvements validate the effectiveness of kernel-level customizations for real-time, AI-driven robotic systems. Future work will focus on integrating GPU-accelerated processing and advanced AI decision-making modules for expanded autonomous functionality.

III. Modeling of the Custom Operating System (OS) for the Raspberry Pi

A. System Overview

The developed operating system, termed "Hybrid Raspbian Buster", was modeled to support dynamic hardware configurations of the Raspberry Pi 4 platform. The OS architecture incorporates a hybrid user interface (UI) capable of executing intelligent decision-making processes, specifically optimized for real-time image processing and autonomous system functionalities. Key attributes of the system include:

- **Dynamic Hardware Adaptability:** Supports real-time recognition and adaptation to connected hardware without manual reconfiguration.
- **Cost-Efficient Design:** Eliminates the need for additional shields, connectivity cards, or external hardware modules, thereby reducing system cost.
- **Intelligent User Interface:** Provides an intuitive and responsive environment for both local and remote operations.

B. Networking and Connectivity Model

The OS was designed with robust and versatile networking capabilities:

- **Wi-Fi Connectivity:** Establishes network access through password authentication, with static IP generation and SSID hosting configured within the system configuration protocols. All devices must reside on the same network for full feature compatibility.
- **Bluetooth Communication:** Supports standard Bluetooth operations for peripheral device connectivity.
- **Remote Access via VNC:** Implements a Virtual Network Computing (VNC) protocol where VNC Viewer inputs (mouse, keyboard, touch) are transmitted to the VNC Server on the OS, allowing complete remote access control.

Limitations:

- Cloud computing integration and multi-network hosting/server activities are currently not supported under the present system design.

C. Software Environment and Development Tools

The modeling and development of the Hybrid Raspbian Buster OS involved a multi-environment programming approach:

- **Python Platform:** The majority of system library extensions and configurations were developed and managed using Python, enabling modularity and rapid customization.
- **Scripting Languages:** UNIX system functionalities were enhanced via scripting with **Perl** and **Ruby**, allowing efficient control and extension of OS features.
- Integrated Development Environments (IDEs):
  - Visual Studio Code (VS Code) for lightweight development and scripting tasks.
  - Visual Studio 2019 for Windows Core GUI integration.
- **Cross-Platform GUI Support:** A Windows Core GUI layer was modeled to enable operations through a graphical interface for Windows-based users, ensuring broader usability.

D. User Interface and System Extensibility

To enhance user accessibility and future-proof the system:

- **User-Friendly Interface:** Designed for ease of use to accommodate a wide range of users, from technical experts to general operators.

- **Firmware Flexibility:** The firmware architecture allows for modular updates. Users can install new packages, modify system behavior, and customize functionalities directly through Python-based build resources.
- **Open Configuration Access:** System files are accessible for modification, permitting users to adapt the firmware to specific project or operational requirements.

The Hybrid Raspbian Buster OS successfully models a lightweight, adaptable, and intelligent operating environment optimized for Raspberry Pi applications in real-time image processing and remote-controlled autonomous systems.

While certain limitations such as cloud integration remain, the system offers a stable, secure, and flexible foundation for further expansion and advanced development.

IV. The Custom Hybrid Raspbian Buster Operating System (OS) for Computer Vision Image Processing

To design the custom operating system (OS) for the Raspberry Pi (RPi) used in this prototype, a comprehensive evaluation of existing distributions was conducted before proceeding with the custom design and developmental OS.

The goal was to identify a solution that could deliver the required features for computer vision and image processing applications.

This approach led to the development of a Hybrid Raspbian Buster OS, leveraging advanced coding, distribution patching, and optimization techniques to ensure a high degree of operational flexibility and performance.

The resulting OS integrates characteristics from multiple distributions, enabling artificial levels of decision-making autonomy by dynamically adjusting its operations based on sensory inputs and system architecture.

This self-optimization capacity ensures that processes such as data acquisition, data sharing, task management, background application handling, driver stabilization, image processing, video rendering, and codec management are executed with maximum efficiency.

Given the heavy demand on graphical memory (responsible for acceleration and computational fluidity), integration with the Google Coral Edge TPU significantly enhances computational speeds and model execution, providing real-time performance improvements critical for image processing tasks. For the design and development of the graphical user interface (GUI) and Unix-based scripting environment, Perl and Ruby were primarily employed, developed within Visual Studio Code and Visual Studio 2019 environments. Additionally, a Windows Core GUI was integrated, allowing users to perform operations through a familiar User Interface on Windows systems.

Core features were developed in the Python environment, ensuring extensibility, dynamic package installation, and future firmware updates. To provide a better understanding concerning the matters of perspectives Figures 1, 2 provide visualizations on the retrospectives.

Key Customizations in the Hybrid OS

A. Overclocking

Overclocking was implemented to enhance CPU and GPU performance beyond factory specifications. Specifically, the /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq directory was modified to maximize Core_freq and GPU_freq values. This optimization is critical for the system, as computer vision applications heavily rely on graphics memory protocols and frame buffer generation for real-time image convolution and analysis.

B. Custom Threading Mechanism

Threading adjustments were introduced to optimize memory management and task scheduling. Given the high graphical memory consumption of image processing tasks, custom threading ensures efficient capture, buffering, and resource allocation, with modifications reflected in the /lib/firmware/raspberrypi/bootloader/bootconf.txt configuration.

C. Custom SSID Configuration

To support reliable remote connectivity, a dedicated SSID was configured with a custom subnet mask via the raspi-config tool in Super User mode. This allows the device to distinguish its network presence effectively, ensuring stable connections in environments with multiple network nodes.

D. Static IP Assignment

Static IP configuration was essential to guarantee consistent network access, particularly for remote monitoring and Virtual Private Network (VPN) connectivity. Static IP settings were manually configured via raspi-config, preventing IP address shifts and ensuring persistent remote session availability.

E. Codec Optimization

To enhance real-time frame rates and reduce processing latency during object detection and bounding box generation, codec performance was improved by modifying parameters within config.txt. This ensured minimal execution delays and maximized detection accuracy.

F. Custom Framework Integrations

Frameworks were selectively adapted and optimized to support various AI models and image processing tasks. Alterations within the config.txt enabled flexible integration and runtime switching between frameworks based on the application needs and image dataset characteristics.

G. Custom Shell Scripts

To resolve compatibility issues between various software packages, custom Unix shell scripts were developed. These scripts handle file operations, environment setup, and kernel-level interactions, ensuring consistent execution and memory mapping within the Python distribution via command-line protocols.

H. Data Storage Enhancements

Since the Raspberry Pi operates through an SD card-based storage system, efforts were made to minimize risks associated with flash memory vulnerabilities (such as voltage fluctuations and data corruption). The bootloader configuration, particularly bootcode. Bin under /lib/firmware/raspberrypi/bootloader/, was modified to support FAT32 file systems optimized for recovery and multi-peripheral compatibility.

I. SSH-triggered PoE Hat Management

To support high-performance roaming and remote operation, a separate Power-over-Ethernet (PoE) management script was triggered via SSH protocols. Thermal optimization strategies were also incorporated by regulating voltage during intensive GPU operations, mitigating overheating risks during image processing tasks.

This Custom Hybrid Raspbian Buster OS thus offers an optimized, flexible, and robust platform tailored specifically for high-performance computer vision and image processing tasks on the Raspberry Pi ecosystem.

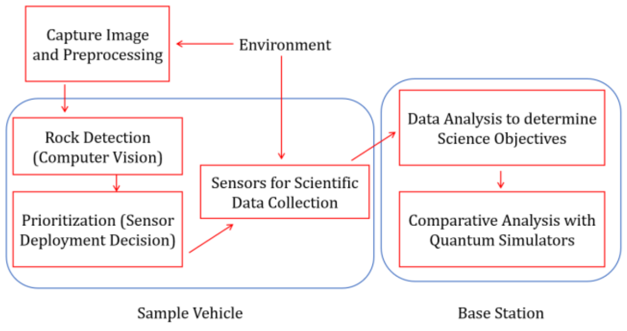

Capture Image and Preprocessing
Environment
Rock Detection (Computer Vision)
Prioritization (Sensor Deployment Decision)
Sensors for Scientific Data Collection
Data Analysis to determine Science Objectives
Comparative Analysis with Quantum Simulators
Sample Vehicle
Base Station

Figure 1. An overview of the Computer Vision system (1)

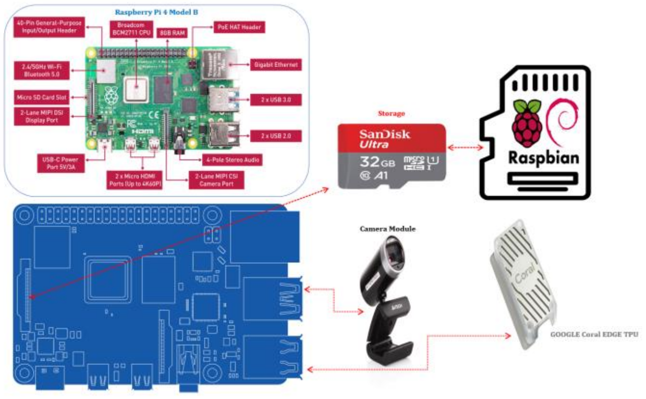


Figure 2. An overview of the Computer Vision system (2)

V. Results and Findings

The design and development of the Custom Hybrid Raspbian Buster Operating System (OS) were systematically segmented and tested across various operational domains to verify the successful integration of the customized features. The prototype system was subjected to a series of experimental validations to assess whether the designed features and modifications met the functional requirements intended for advanced computer vision image processing applications. Primarily, the system was evaluated for its performance in video rendering and image processing tasks, as these represent the most computationally intensive and memory-demanding operations relevant to the system's application context.

The testing was conducted throughout the authors' Final Year Project and Capstone Thesis, allowing extensive real-world validation. The final system layout and architecture, inclusive of hardware modifications, system utilities, and developed features, are illustrated in the provided figures and schematics.

The prototype incorporates dependency management and custom configuration scripts developed during the design phase. Furthermore, in addition to native Raspbian Buster distribution customization, critical utilities from the Kali Linux distribution environment and VMware-based virtual distributions were integrated to enhance system robustness and flexibility.

Field of View (FOV) and Resolution Optimization

In the context of real-time video streaming, similar to first-person shooter (FPS) video games where Field of View (FOV) determines the observable area, the developed system required an optimized resolution and frame buffering strategy to maximize the visible scene during image acquisition.

Wider FOV and increased rendering resolutions were achieved through GPU pixel optimization and geometric scaling, providing richer image histograms for computational tasks such as object detection and classification. The camera modules, including both the dedicated Raspberry Pi camera and optional onboard webcam integration, were managed via custom shell scripts, enabling flexible remote video feed escalation and multi-camera functionality.

Pixel Binning for Accelerated Image Processing

To enhance the efficiency of image processing, pixel binning techniques were deployed. Pixel binning, the process of combining multiple adjacent pixels into a single pixel value, significantly reduced the computational load by decreasing the overall pixel count without severely compromising image integrity.

For instance, a 2×2 binning scheme consolidates four neighboring pixels into one, which reduces processing time and accelerates the frame detection pipeline.

This optimization was particularly beneficial when training and testing on standard hardware, where computational and memory resources are limited. Custom shell scripts were developed to automate pixel binning during frame capture and detection, ensuring faster execution and improved system responsiveness.

System Performance

The hybrid OS demonstrated robust and efficient handling of tasks such as:

- **Real-time Video Rendering**: Smooth frame rates with minimal buffering delays, even under multi-threaded image acquisition.
- **Image Processing and Frame Detection**: Enhanced throughput due to optimized binning and memory management.
- **Custom Networking**: Reliable wireless and wired connections through customized static IP and SSID configurations for remote operation.
- **Resource Management**: Efficient graphical memory utilization with overclocked GPU and CPU parameters, ensuring stable operation during heavy graphical loads.
- **Thermal Management**: Effective thermal regulation using SSH-triggered PoE HAT extensions to mitigate overheating risks during intensive computing tasks.

These findings affirm that the proposed system modifications provided significant improvements in computational performance, system responsiveness, remote accessibility, and overall operational stability, positioning the prototype as a viable platform for advanced computer vision applications using Raspberry Pi hardware. To give an idea and provide much better understandings Figures 3-6 along with Table 1 which offers visualizations concerning the matter of perspectives.

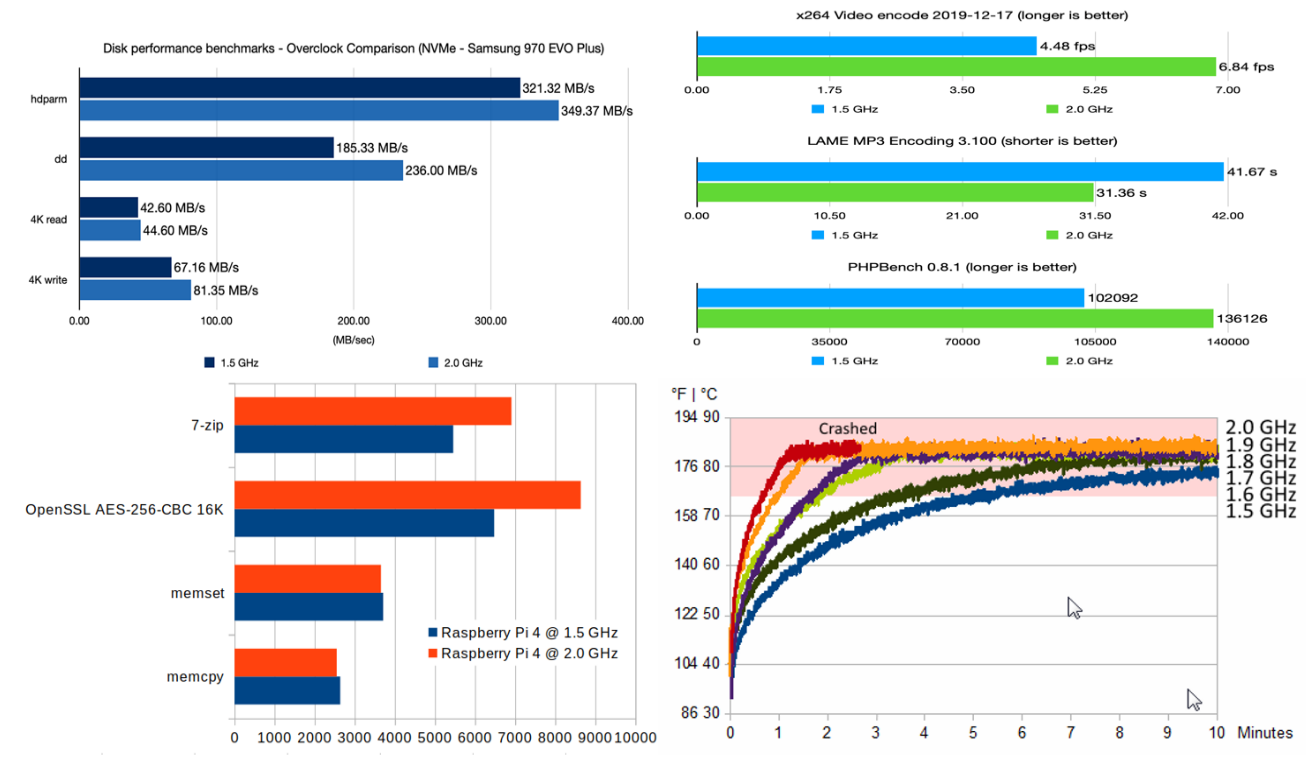
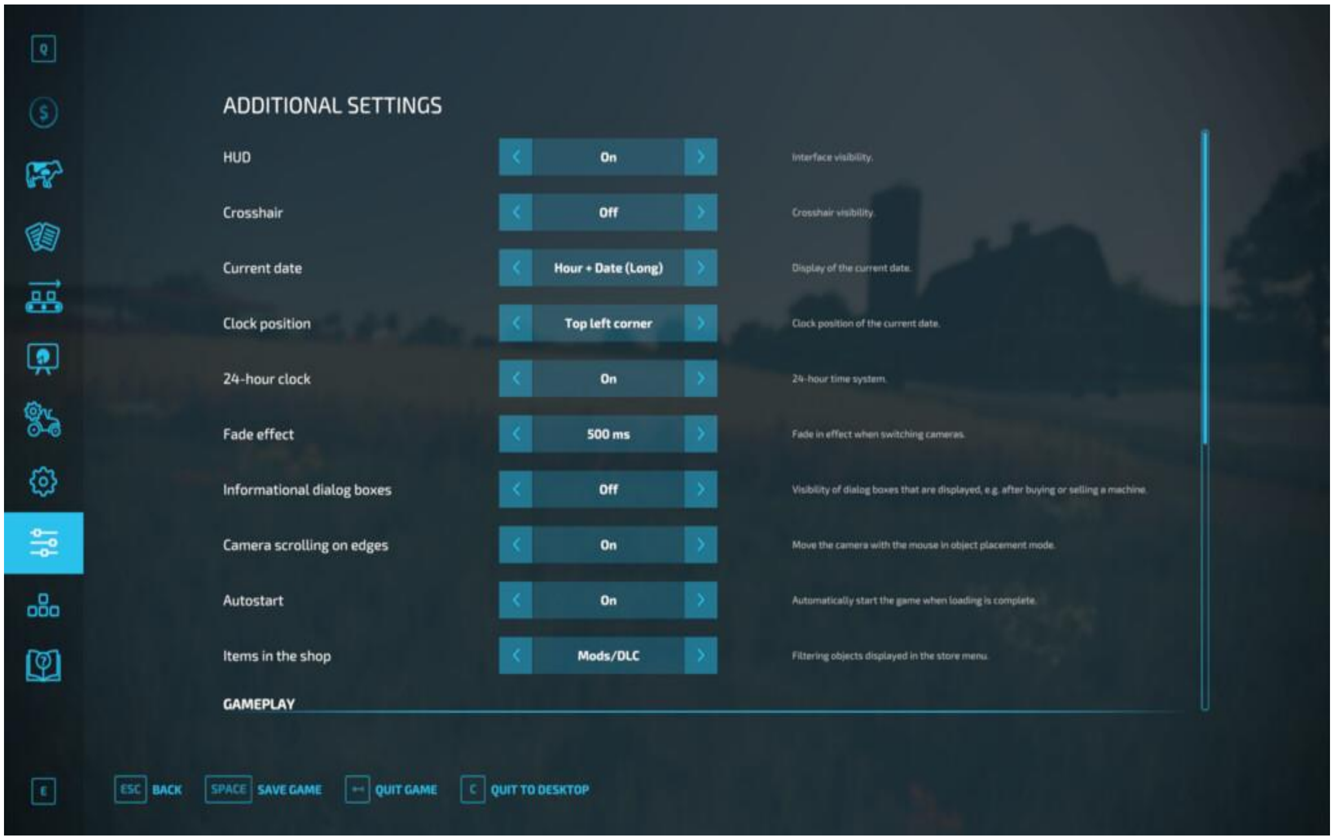


Figure 3. The approaches towards a Hybrid Raspbian Buster OS and its capabilities in terms of Executions

Figure 4. An illustration of FOV in terms of GPU in Game Mods

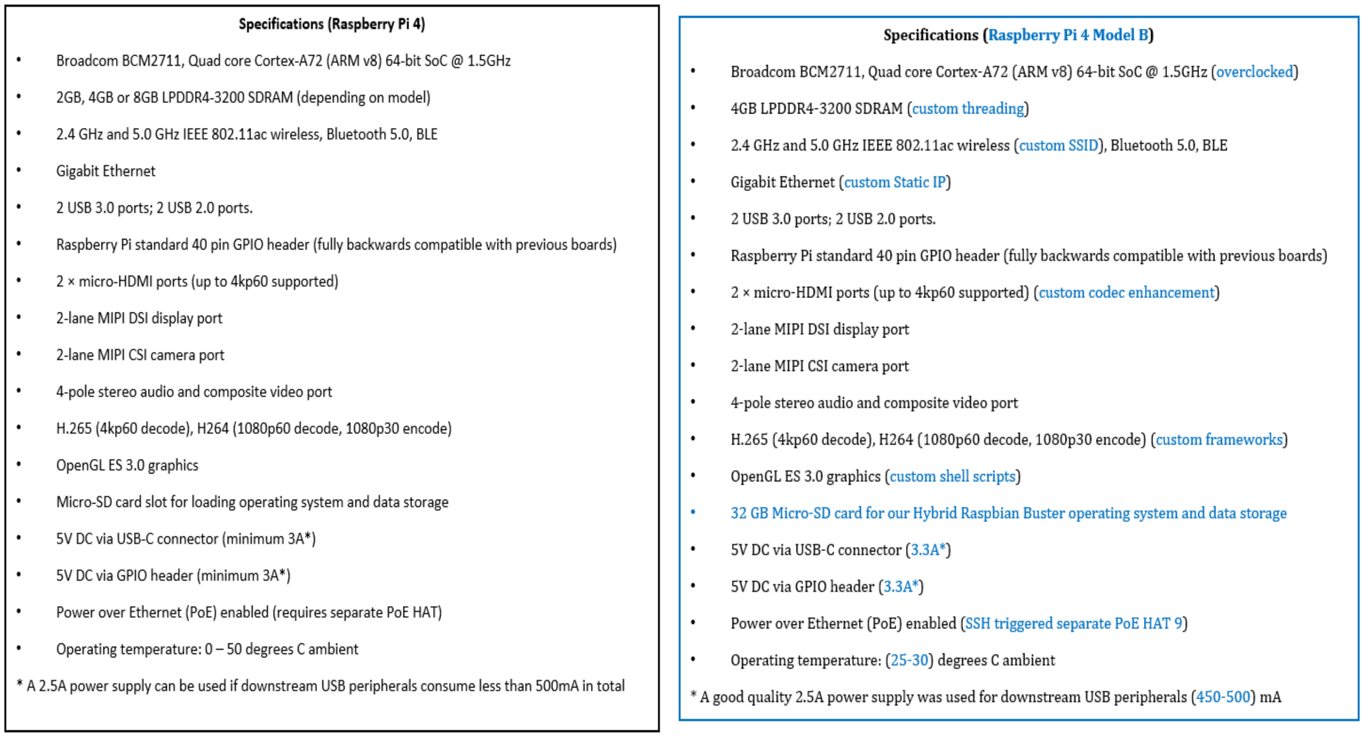


ADDITIONAL SETTINGS

HUD	On	Interface visibility.
Crosshair	Off	Crosshair visibility.
Current date	Hour + Date (Long)	Display of the current date.
Clock position	Top left corner	Clock position of the current date.
24-hour clock	On	24-hour time system.
Fade effect	500 ms	Fade in effect when switching cameras.
Informational dialog boxes	Off	Visibility of dialog boxes that are displayed, e.g. after buying or selling a machine.
Camera scrolling on edges	On	Move the camera with the mouse in object placement mode.
Autostart	On	Automatically start the game when loading is complete.
Items in the shop	Mods/DLC	Filtering objects displayed in the store menu.

GAMEPLAY

Figure 5. An illustration of the layout in gaming mods



Specifications (Raspberry Pi 4)

- Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
- 2GB, 4GB or 8GB LPDDR4-3200 SDRAM (depending on model)
- 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE
- Gigabit Ethernet
- 2 USB 3.0 ports; 2 USB 2.0 ports.
- Raspberry Pi standard 40 pin GPIO header (fully backwards compatible with previous boards)
- 2 × micro-HDMI ports (up to 4kp60 supported)
- 2-lane MIPI DSI display port
- 2-lane MIPI CSI camera port
- 4-pole stereo audio and composite video port
- H.265 (4kp60 decode), H264 (1080p60 decode, 1080p30 encode)
- OpenGL ES 3.0 graphics
- Micro-SD card slot for loading operating system and data storage
- 5V DC via USB-C connector (minimum 3A*)
- 5V DC via GPIO header (minimum 3A*)
- Power over Ethernet (PoE) enabled (requires separate PoE HAT)
- Operating temperature: 0 – 50 degrees C ambient

* A 2.5A power supply can be used if downstream USB peripherals consume less than 500mA in total

Specifications (Raspberry Pi 4 Model B)

- Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz (overclocked)
- 4GB LPDDR4-3200 SDRAM (custom threading)
- 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless (custom SSID), Bluetooth 5.0, BLE
- Gigabit Ethernet (custom Static IP)
- 2 USB 3.0 ports; 2 USB 2.0 ports.
- Raspberry Pi standard 40 pin GPIO header (fully backwards compatible with previous boards)
- 2 × micro-HDMI ports (up to 4kp60 supported) (custom codec enhancement)
- 2-lane MIPI DSI display port
- 2-lane MIPI CSI camera port
- 4-pole stereo audio and composite video port
- H.265 (4kp60 decode), H264 (1080p60 decode, 1080p30 encode) (custom frameworks)
- OpenGL ES 3.0 graphics (custom shell scripts)
- 32 GB Micro-SD card for our Hybrid Raspbian Buster operating system and data storage
- 5V DC via USB-C connector (3.3A*)
- 5V DC via GPIO header (3.3A*)
- Power over Ethernet (PoE) enabled (SSH triggered separate PoE HAT 9)
- Operating temperature: (25-30) degrees C ambient

* A good quality 2.5A power supply was used for downstream USB peripherals (450-500) mA

Figure 6. An overview of the Enhancements with the available RPi and integrated Edge TPU

TABLE IV. A COMPARISON BETWEEN THE STANDARD RASPBIAN OS AND THE HYBRID OS

Basis of Features	Raspbian	Hybrid Raspbian Buster
Processing Power (CPU)	(base core) 1.5GHz	(overclocked) 2.147GHz
Processing Power (GPU)	(arm_freq) 500MHz	(arm_freq) 750MHz
Supply Voltage	4.8V USB (4.2V)	(GPIO) 5V
Schedule Manager	Cache memory	Cache plus multi-threading
Bootloader	Flash Drive (FAT16)	FAT32 (recovery.bin and EEPROM)
Portability	PoE provided	PoE enabled with SSH
Remote Access	Single hosts with individual control	Multiple hosts with multiple control
Frame Rates	Avg. 15-20	Avg. 30-60 (FHD supported)
FPS	180 (OpenGL 122)	188 (OpenGL 138)
Efficiency (multitasking)	Avg. 3-5	Avg. 7-10

VI. DISCUSSIONS AND FUTURE DIRECTIONS

A. *System Architecture and Performance Enhancements*

The development of a custom operating system (OS) for the Raspberry Pi prototype involved a thorough evaluation of existing distributions to identify and integrate the most suitable features. The author adopted a novel and unconventional design methodology that combined direct code modifications, distribution patching, and customized system configurations to formulate a purpose-built, performance-optimized OS.

A key innovation lies in the system's hybrid architecture, enabling it to operate at an artificial level of dynamic integrity. This self-adaptive behavior, driven by integrated sensory inputs and task profiling, allows the OS to autonomously optimize its operations across varying computational loads.

This adaptability is crucial for resource-constrained environments such as the Raspberry Pi, where tasks including real-time video rendering, data acquisition, task scheduling, and image processing require careful resource management. The system's performance optimizations can be summarized across several critical strategic areas:

- **Overclocking Enhancements**: Through careful modification of system configuration files (e.g., ***/sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq***), the core CPU and GPU frequencies were increased to handle computationally intensive vision and graphical tasks. This resulted in improved frame buffering, faster image convolution, and enhanced overall accuracy in real-time computer vision applications.
- **Custom Threading Mechanism**: A bespoke threading model, implemented via modifications to ***/lib/firmware/raspberrypi/bootloader/bootconf.txt***, optimized memory management and task scheduling. This ensured stability and responsiveness under high computational loads, maintaining system integrity during concurrent background operations.

- **Network Configuration Optimization**: Custom SSID and Static IP configurations, managed through raspi-config, were deployed to guarantee stable remote access and optimized bandwidth distribution. This was particularly significant for scenarios involving remote monitoring, data streaming, and cloud integration.
- **Codec and Multimedia Enhancements**: Modifications to config.txt improved multimedia processing pipelines, reducing execution latency, enhancing frame rates, and optimizing image quality — critical for AI-driven object detection and classification models.
- **Framework Compatibility Integration**: Configuration adjustments enabled support for diverse AI frameworks, facilitating seamless deployment of machine learning models without significant system overheads.
- **Shell Script and Kernel Path Customization**: Custom scripting for Python environment management and kernel execution paths allowed for flexible and conflict-free software execution, improving system versatility and minimizing runtime errors.
- **Data Storage and Bootloader Optimization**: Operating on a FAT32 filesystem with customized bootloader settings (***bootcode.bin***), the system achieved robust memory usage, enhanced data recovery capability, and stable peripheral communications.
- **SSH and PoE-Based Remote Management**: SSH integration with customized Power-over-Ethernet (PoE) modules improved remote accessibility and dynamic voltage regulation, successfully mitigating thermal throttling risks during intensive computational periods.

Collectively, these strategies enabled the custom OS to surpass conventional distribution performance benchmarks, delivering enhanced computational efficiency, stability, and scalability suited for AI and graphical workloads.

B. Implications and Future Research Directions

The results underscore the transformative potential of targeted OS customizations in elevating Raspberry Pi performance for AI-driven, real-time computing applications. This exploration highlights how low-level system adjustments—spanning hardware optimization, networking, multimedia handling, and dynamic resource management—can significantly enhance embedded systems' computational effectiveness. Moving forward, several promising research directions emerge:

- **Kernel-Level Optimization**: Deeper modifications at the kernel level, including custom scheduler development and device driver enhancements, could further minimize latency and maximize system responsiveness under variable workload demands.
- **Real-Time Operating System (RTOS) Features**: Future iterations could integrate real-time capabilities, allowing deterministic execution timing critical for applications such as autonomous robotics, real-time surveillance, and edge AI deployments.
- **Machine Learning-Based Resource Allocation**: Implementing machine learning models for adaptive resource allocation could enable the system to dynamically adjust CPU/GPU frequencies, memory allocation, and task prioritization based on real-time performance profiling.
- **Security and Resilience Enhancements**: Given the growing relevance of IoT and edge deployments, enhancing system security protocols and developing fail-safe mechanisms could fortify the OS against cyber threats and operational anomalies.
- **Scalability to Industrial and Research Applications**: By extending the system's modularity and hardware compatibility, the platform could be adapted for broader industrial use cases such as smart manufacturing, environmental monitoring, and healthcare automation.

This investigation illustrates the critical role that domain-specific operating system design plays in advancing the capabilities of embedded computing platforms. The findings contribute valuable insights for developers, researchers, and engineers aiming to build high-performance, specialized systems tailored to emerging computational challenges.

VII. Conclusions

The development of a custom operating system (OS) for the Raspberry Pi, as presented in this study, demonstrates the significant potential of targeted system integration, hardware optimization, and software customization for achieving high-performance computing in resource-constrained environments. Through extensive iterative testing, system patching, and configuration enhancements, the proposed OS has been successfully tailored to support a broad range of computationally intensive tasks, including computer vision, real-time data acquisition, and advanced image processing. A core contribution of this research is the introduction of a hybrid OS design methodology, integrating and patching components from multiple distributions to optimize system performance. Critical performance enhancements—including overclocking, custom threading mechanisms, multimedia codec improvements, and memory optimization—have been effectively implemented to reduce execution latency, enhance computational throughput, and maintain system stability during high-load operations. Furthermore, custom network configurations, encompassing SSID management and static IP allocations, have provided a stable and secure foundation for remote access and seamless cloud integration. The successful integration of external accelerators such as the Google Coral Edge TPU has markedly boosted AI and deep learning performance, validating the scalability and adaptability of the proposed platform. The fusion of a Windows Core GUI interface with UNIX-based scripting capabilities further broadens the operational versatility of the system, making it suitable for a wider array of user-centric and industrial applications.

Moreover, the adoption of customized shell scripts, storage management protocols, and SSH-triggered Power-over-Ethernet (PoE) enhancements has improved system resilience, power management, and thermal efficiency, ensuring robust operation even during extended runtime periods.

The research findings affirm the feasibility and benefits of designing highly customized, performance-optimized operating systems tailored for embedded and edge computing devices. The modularity and adaptability of the developed OS offer a strong foundation for future enhancements, enabling rapid integration of emerging AI frameworks, expanded hardware support, and task-specific performance tuning.

Future work will aim to extend this platform’s compatibility across a wider range of edge computing devices, implement advanced security protocols to safeguard against cyber threats, and incorporate machine learning-driven automation mechanisms for dynamic resource management. Furthermore, targeted applications in robotics, the Internet of Things (IoT), autonomous systems, and industrial automation will be explored to validate the system’s broader applicability in real-world scenarios.

This research exploration contributes valuable insights into the design of domain-specific operating systems and underscores the strategic importance of holistic system optimization for next-generation embedded computing solutions.

Acknowledgments

The author of this research was solely responsible for conceiving, planning, and executing the research explorational project. All aspects of the research, including idea conceptualization, experimental design, manuscript writing, data collection, and prototype development, were personally undertaken by the author as this was his undergrad project & thesis work. The research project commenced during the author's final undergraduate years but faced delays due to the pandemic. The author utilized the lockdown period to refine and update the research, ensuring it aligned with the circumstances. However, when the pandemic situation began to normalize, the author pursued towards his higher studies, leading to a temporary pause within the research work until the author decides to resume and proceed with it further.

REFERENCES

[1] Matthies, L., Maimone, M., Johnson, A. et al. Computer Vision on Mars. Int J Comput Vis 75, 67–92 (2007). https://doi.org/10.1007/s11263-007-0046-z

[2] Francis, R., Estlin, T., Doran, G., Johnstone, S., Gaines, D., Verma, V., Burl, M., Frydenvang, J., Montaño, S., Wiens, R. C., Schaffer, S., Gasnault, O., DeFlores, L., Blaney, D., & Bornstein, B. (2017). AEGIS autonomous targeting for ChemCam on Mars Science Laboratory: Deployment and results of initial science team use. Science robotics, 2(7), eaan4582. https://doi.org/10.1126/scirobotics.aan4582

[3] Estlin, T. A., Bornstein, B. J., Gaines, D. M., Anderson, R. C., Thompson, D. R., Burl, M., Castaño, R., & Judd, M. (2012). AEGIS Automated Science Targeting for the MER Opportunity Rover. ACM Transactions on Intelligent Systems and Technology, 3(3), 1–19. https://doi.org/10.1145/2168752.2168764

[4] Castano, R., Estlin, T., Anderson, R. C., Gaines, D. M., Castano, A., Bornstein, B., Chouinard, C., & Judd, M. (2007). Oasis: Onboard autonomous science investigation system for opportunistic rover science. Journal of Field Robotics, 24(5), 379–397. https://doi.org/10.1002/rob.20192

[5] Gaines, D. (n.d.). Mars 2020 Project The Mars 2020 OnBoard Planner: Flight Software Mars 2020 OBP Team. Retrieved August 14, 2025, from https://ai.jpl.nasa.gov/public/documents/presentations/m2020-2-fsw.pdf

[6] Li, Y., Xiao, Z., Ma, C., Zeng, L., Zhang, W., Peng, M., & Li, A. (2023). Extraction and analysis of three-dimensional morphological features of centimeter-scale rocks in Zhurong landing region. Journal of Geophysical Research: Planets, 128, e2022JE007656. https://doi.org/10.1029/2022JE007656

[7] Pang, B., Nijkamp, E., & Wu, Y. N. (2020). Deep learning with tensorflow: A review. Journal of Educational and Behavioral Statistics, 45(2), 227-248.

[8] Abadi, M. (2016, September). TensorFlow: learning functions at scale. In Proceedings of the 21st ACM SIGPLAN international conference on functional programming (pp. 1-1).

[9] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J, & Zheng, X. (2016). {TensorFlow}: a system for {Large-Scale} machine learning. In 12th USENIX symposium on operating systems design and implementation (OSDI 16) (pp. 265-283).

[10] Shukla, N., & Fricklas, K. (2018). Machine learning with TensorFlow (Vol. 7, No. 06, p. 274). Greenwich: Manning.

[11] Singh, P., & Manure, A. (2019). Introduction to tensorflow 2.0. In Learn TensorFlow 2.0: Implement machine learning and deep learning models with python (pp. 1-24). Berkeley, CA: Apress.

[12] Zhang, Y., Chen, Y., Cheung, S. C., Xiong, Y., & Zhang, L. (2018, July). An empirical study on tensorflow program bugs. In Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis (pp. 129-140).

[13] Smilkov, D., Thorat, N., Assogba, Y., Nicholson, C., Kreeger, N., Yu, P., ... & Wattenberg, M. M. (2019). Tensorflow. js: Machine learning for the web and beyond. Proceedings of Machine Learning and Systems, 1, 309-321.

[14] Hope, T., Resheff, Y. S., & Lieder, I. (2017). Learning tensorflow: A guide to building deep learning systems. " O'Reilly Media, Inc.".

[15] Pattanayak, S., Pattanayak, J. S., & John, S. (2017). Pro deep learning with tensorflow (pp. 153-278). New York, NY, USA: Apress.

[16] Sanchez, S. A., Romero, H. J., & Morales, A. D. (2020, May). A review: Comparison of performance metrics of pretrained models for object detection using the TensorFlow framework. In IOP conference series: materials science and engineering (Vol. 844, No. 1, p. 012024). IOP Publishing.

[17] Kumar, N., Rathee, M., Chandran, N., Gupta, D., Rastogi, A., & Sharma, R. (2020, May). Cryptflow: Secure tensorflow inference. In 2020 IEEE Symposium on Security and Privacy (SP) (pp. 336-353). IEEE.

[18] Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., ... & Hechtman, B. (2018). Mesh-tensorflow: Deep learning for supercomputers. Advances in neural information processing systems, 31.

[19] Zafar, I., Tzanidou, G., Burton, R., Patel, N., & Araujo, L. (2018). Hands-on convolutional neural networks with TensorFlow: Solve computer vision problems with modeling in TensorFlow and Python. Packt Publishing Ltd.

[20] Ertam, F., & Aydın, G. (2017, October). Data classification with deep learning using Tensorflow. In 2017 international conference on computer science and engineering (UBMK) (pp. 755-758). IEEE.

[21] Culjak, I., Abram, D., Pribanic, T., Dzapo, H., & Cifrek, M. (2012, May). A brief introduction to OpenCV. In 2012 proceedings of the 35th international convention MIPRO (pp. 1725-1730). IEEE.

[22] Howse, J. (2013). OpenCV computer vision with python (Vol. 27). Birmingham, UK: Packt Publishing.

[23] Kaehler, A., & Bradski, G. (2016). Learning OpenCV 3: computer vision in C++ with the OpenCV library. " O'Reilly Media, Inc.".

[24] Xie, G., & Lu, W. (2013). Image edge detection based on opencv. International Journal of Electronics and Electrical Engineering, 1(2), 104-106.

[25] Yang, J. (2023, October). Real time object tracking using OpenCV. In 2023 IEEE 3rd International Conference on Data Science and Computer Application (ICDSCA) (pp. 1472-1475). IEEE.

[26] Sharma, A., Pathak, J., Prakash, M., & Singh, J. N. (2021, December). Object detection using OpenCV and python. In 2021 3rd international conference on advances in computing, communication control and networking (ICAC3N) (pp. 501-505). IEEE.

[27] Voulodimos, A., Doulamis, N., Doulamis, A., & Protopapadakis, E. (2018). Deep learning for computer vision: A brief review. Computational intelligence and neuroscience, 2018(1), 7068349.

[28] Szeliski, R. (2022). Computer vision: algorithms and applications. Springer Nature.

[29] Khan, A. I., & Al-Habsi, S. (2020). Machine learning in computer vision. Procedia Computer Science, 167, 1444-1451.

[30] Ikeuchi, K. (Ed.). (2021). Computer vision: A reference guide. Cham: Springer International Publishing.

[31] Jiang, P., Ergu, D., Liu, F., Cai, Y., & Ma, B. (2022). A Review of Yolo algorithm developments. Procedia computer science, 199, 1066-1073.

[32] Terven, J., Córdova-Esparza, D. M., & Romero-González, J. A. (2023). A comprehensive review of yolo architectures in computer vision: From yolov1 to yolov8 and yolo-nas. Machine learning and knowledge extraction, 5(4), 1680-1716.

[33] Hussain, M. (2023). YOLO-v1 to YOLO-v8, the rise of YOLO and its complementary nature toward digital manufacturing and industrial defect detection. Machines, 11(7), 677.

[34] Diwan, T., Anirudh, G., & Tembhurne, J. V. (2023). Object detection using YOLO: challenges, architectural successors, datasets and applications. multimedia Tools and Applications, 82(6), 9243-9275.

[35] Kalaiselvi, T., Sriramakrishnan, P., & Somasundaram, K. (2017). Survey of using GPU CUDA programming model in medical image analysis. Informatics in Medicine Unlocked, 9, 133-144.

[36] Dehal, R. S., Munjal, C., Ansari, A. A., & Kushwaha, A. S. (2018, October). Gpu computing revolution: Cuda. In 2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN) (pp. 197-201). IEEE.

[37] (2015). Spie.org. https://www.spie.org/news/5950-laser-induced-breakdown-spectroscopy-for-identification-of-solid-recycled-materials

[38] Fuentes, R., Luarte, D., Sandoval, C., Myakalwar, A. K., Alvarez, J., Yáñez, J., & Sbarbaro, D. (2022). Laser-Induced Breakdown Spectroscopy and Hyperspectral Imaging Data Fusion for improved Mineralogical Analysis of Copper Concentrates. IFAC-PapersOnLine, 55(21), 85–90. https://doi.org/10.1016/j.ifacol.2022.09.248

[39] Bakker, M. C. M., & Xia, H. (2015). Laser-induced breakdown spectroscopy for identification of solid recycled materials. SPIE Newsroom. https://doi.org/10.1117/2.1201505.005950

[40] Rietsche, R., Dremel, C., Bosch, S., Steinacker, L., Meckel, M., & Leimeister, J. M. (2022). Quantum computing. Electronic Markets, 32(4), 2525-2536.

[41] Preskill, J. (2018). Quantum computing in the NISQ era and beyond. Quantum, 2, 79.

[42] Cao, Y., Romero, J., Olson, J. P., Degroote, M., Johnson, P. D., Kieferová, M., ... & Aspuru-Guzik, A. (2019). Quantum chemistry in the age of quantum computing. Chemical reviews, 119(19), 10856-10915.

[43] Gyongyosi, L., & Imre, S. (2019). A survey on quantum computing technology. Computer Science Review, 31, 51-71.

[44] Bova, F., Goldfarb, A., & Melko, R. G. (2021). Commercial applications of quantum computing. EPJ quantum technology, 8(1), 2.

[45] Bayerstadler, A., Becquin, G., Binder, J., Botter, T., Ehm, H., Ehmer, T., ... & Winter, F. (2021). Industry quantum computing applications. EPJ Quantum Technology, 8(1), 25.

[46] Alvarez-Rodriguez, U., Sanz, M., Lamata, L., & Solano, E. (2018). Quantum artificial life in an IBM quantum computer. Scientific reports, 8(1), 14793.

[47] Acasiete, F., Agostini, F. P., Moqadam, J. K., & Portugal, R. (2020). Implementation of quantum walks on IBM quantum computers. Quantum Information Processing, 19(12), 426.

[48] Cruz, D., Fournier, R., Gremion, F., Jeannerot, A., Komagata, K., Tosic, T., & Javerzac-Galy, C. (2019). Efficient quantum algorithms for GHZ and W states, and implementation on the IBM quantum computer. Advanced Quantum Technologies, 2(5-6), 1900015.

[49] AbuGhanem, M. (2025). IBM quantum computers: evolution, performance, and future directions. The Journal of Supercomputing, 81(5), 687.