

Article

Benchmarking YOLOv8–YOLOv12 for Real-Time Object Detection on Single-Board Computers

Omar Shalash ^{1,*}, Esraa Khatab ², Ahmed El-Agamy ³, Loay Elmokadem ³, Yasmin Abouelsaad ³, Jasser Zaki ³, Mohamed El-Sayed ³ and Hany Said ³

¹ Artificial Intelligence Research Center (AIRC), College of Engineering and Information Technology, Ajman University, Ajman P.O. Box 346, United Arab Emirates

² School of Mathematics and Computer Science, Heriot-Watt University, Dubai P.O. Box 501745, United Arab Emirates; e.khatab@hw.ac.uk

³ College of Artificial Intelligence, Arab Academy for Science, Technology & Maritime Transport, Alamein 19838, Egypt; a.elagamy10277@student.aast.edu (A.E.-A.); l.elmokad03410@student.aast.edu (L.E.); y.abouels01859@student.aast.edu (Y.A.); j.zaki02823@student.aast.edu (J.Z.); mohamedkasem@aast.edu (M.E.-S.); hanysaid2000@aast.edu (H.S.)

* Correspondence: o.shalash@ajman.ac.ae; Tel.: +971-555288505

Abstract

The YOLO (You Only Look Once) object detection models have undergone rapid evolution, with each version introducing architectural enhancements aiming to improve speed, accuracy, and deployment. Simultaneously, single-board computers (SBCs) have advanced to support increasingly complex AI models in edge environments. This study presents a comprehensive benchmarking of YOLO versions 8 through 12 across a range of SBCs, including Raspberry Pi4/5, NVIDIA Jetson Nano, Jetson Orin, and LattePanda, under different power modes. Key performance metrics, including inference speed (FPS), detection accuracy (mAP), RAM usage, and computational complexity (FLOPs), are evaluated. These findings offer practical insights for developers and researchers to select optimal YOLO variants and SBC configurations for real-time edge deployment.

Keywords: YOLO object detection; single-board computers; real-time systems; computer vision

1. Introduction

Object detection has become a fundamental task in computer vision, enabling machines to classify and localize multiple objects in different scenes within images or video streams. Its importance spans a wide range of real-world applications, including autonomous driving, surveillance, robotics, smart cities, healthcare diagnostics, and retail analysis. As the demand for deploying intelligent systems on resource-constrained edge devices grows, the need for efficient and accurate object detection models becomes increasingly important [1].

Object detection techniques are categorized into two groups: conventional techniques and detection based on deep learning (DL). Classical object detection methods rely on manually crafted features and classical machine learning techniques, typically using approaches like Viola–Jones (VJ), Histogram of Oriented Gradients (HoG), or Deformable Part Models (DPM) [2]. The VJ detector, introduced by Viola and Jones in 2001 [3], detects faces using sliding windows and Haar-like features. In 2005, Dalal and Triggs developed HoG [4], which identifies gradient orientations within image blocks and applies image scaling with fixed detection windows, feeding the data into classifiers like Support Vector Machines.



Academic Editor: Mehmed Kantardzic

Received: 8 May 2026

Revised: 24 June 2026

Accepted: 9 July 2026

Published: 13 July 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 \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

DPM, introduced by Felzenszwalb in 2008 as an extension of HoG [5], combines a coarse detection window with part-based models to improve object localization.

On the other hand, DL-based object detection utilizes neural networks, particularly Convolutional Neural Networks (CNNs), to automatically extract complex features from raw image data [6,7], thereby significantly improving detection performance across various real-world applications, such as autonomous driving, surveillance, and healthcare. These models fall into two main categories [8]: one-stage detectors, such as You Only Look Once (YOLO) [9] and Single-Shot Detector (SSD) [10], which perform detection in a single step for fast, real-time processing; two-stage detectors, like Faster R-CNN and R-CNN variant models [11], which first generate region proposals and then refine them for higher accuracy, though at the cost of increased computational complexity [12–14].

The YOLO family of models has emerged as a leading solution due to its real-time performance and evolving architectural optimizations. Different YOLO versions have been widely used in various real-world applications due to their remarkable balance between speed and accuracy, including autonomous driving, the healthcare sector, and surveillance systems. The model's ability to run efficiently on edge devices also makes it highly suitable for robotics, where drones and mobile robots use YOLO for navigation and object tracking in dynamic environments. However, with the growing number of YOLO variants, it was challenging to select the most suitable version for a specific need. In this study, we focus on benchmarking the latest YOLO versions: YOLOv8 through YOLOv12, across different single-board computers (SBCs) and power modes, evaluating their suitability for edge deployment through key performance metrics such as inference speed (FPS), accuracy (mAP), memory usage, computational complexity (FLOPs), and model load time.

2. Literature Review

This section reviews the evolution of YOLO-based object detection models, highlighting key architectural advancements from early versions to the most recent releases. It also examines prior benchmarking and review studies, identifying limitations and gaps in evaluating YOLO performance on resource-constrained single-board computing platforms.

2.1. YOLO Series Timeline and Evolution

As shown in Figure 1, YOLO's developing pathway is shaping a unique object detection framework that is evolving continuously [15]. YOLOv1–v2 revolutionized object detection by introducing a unified CNN-based architecture that processes the entire image in one pass, dividing it into a grid to predict bounding boxes and class probabilities. YOLOv2 builds on this by adopting Darknet-19, adding anchor boxes for accurate multi-object detection, and supporting multi-resolution inputs, boosting accuracy on benchmarks like COCO and VOC [9,16]. YOLOv3 switched to Darknet-53 with residual connections and multi-scale detection heads, making it more effective at detecting objects of various sizes. It also improved anchor box generation and classification [17]. YOLOv4 formalized the modular design of backbones, necks, and heads by using CSPDarknet-53 and PAN + SPP for feature fusion, achieving a strong balance between speed and accuracy [18]. YOLOv5 retains the modular structure but adds deployment-friendly features and components such as SPP and FPN + PAN [19]. YOLOv6 introduced RepVGG-inspired blocks and multi-task loss functions for better precision [20]. YOLOv7 focused on optimizing training and inference with E-ELAN and Trainable Bag-of-Freebies, refining the speed–accuracy trade-off for real-time tasks.

On the other hand, YOLOv8 shifted to an anchor-free paradigm, simplifying the detection head and enhancing multi-scale feature aggregation through Cross Stage Partial Network (CSPNet), Feature Pyramid Network (FPN), and Path Aggregation Network

(PAN). It improved the detection of small and large objects using more efficient CNN operations [21]. YOLOv9 introduced Programmable Gradient Information (PGI) and a refined version of Gradient Enhanced Lightweight Aggregation Network (GELAN) as the backbone to reduce information loss during gradient backpropagation. It remained anchor-free, and emphasized stable training and feature reuse [22].

YOLOv10 transitioned to one-to-one object prediction, eliminating the need for Non-Maximum Suppression (NMS). Compact Inverted Blocks (CIBs) and Range-Guided Block designs are deployed for YOLOv10 to improve localization while reducing redundancy [23]. YOLOv11 further enhanced small object detection via modules like C3k2, SPFF, and C2PSA, which improved feature extraction, spatial pooling, and attention mechanisms, particularly in complex scenes [24].

Finally, YOLOv12 focused on latency reduction by simplifying the attention mechanism, using a positional perceiver, and reducing MLP ratios, while maintaining detection accuracy. It is the most efficient version for deployment on latency-critical edge systems [25].

A summarized comparison between YOLO architectures is presented in Table 1.

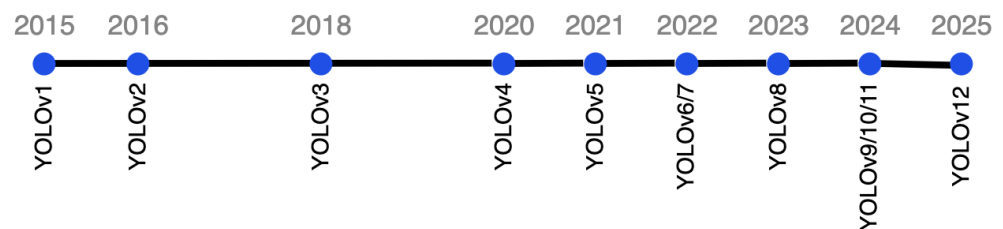


Figure 1. YOLO series timeline.

Table 1. Comparison of YOLO architectures from YOLOv1 to YOLOv12.

Version	Backbone	Neck	Head	Detection Strategy
YOLOv1	Custom CNN	N/A	2 FC layers	Grid cells (7×7)
YOLOv2	Darknet-19	N/A	Anchor boxes, Class probs	Anchor boxes
YOLOv3	Darknet-53	Feature Pyramid	3 Heads (Multi-scale)	Anchor boxes + K-means
YOLOv4	CSPDarknet-53	PAN + SPP	Same as YOLOv3	Improved neck
YOLOv5	CSPDarknet + SPP	FPN + PAN	3 Heads	PyTorch based
YOLOv6	Rep Backbone	FPN + PAN + RepBlock	IoU-based NMS	Multi-task loss
YOLOv7	CSPDarknet + E-ELAN	PAN	Trainable BoF	Optimized for speed
YOLOv8	CSPNet	FPN + PAN	Anchor-free	Improved flow
YOLOv9	GELAN	PGI	Anchor-free	Gradient optimization
YOLOv10	Compact Inv. Block	Range-guided Block	One-to-One Prediction	No NMS
YOLOv11	C3k2 + SPFF	C2PSA	Lightweight head	Enhanced attention
YOLOv12	Simplified C2PSA	Positional Perceiver	Reduced MLP ratio	Latency optimized

2.2. Prior Review Articles on YOLO Models

There are many comprehensive reviews on YOLO models:

- In [26], the authors reviewed YOLOv1 through YOLOv5, analyzing their architectures, detection speed, accuracy, and dataset compatibility.
- In [27], the authors compared YOLOv7, SSD, and Faster R-CNN on small SBC platforms such as Raspberry Pi and Jetson Nano, evaluating metrics like inference speed (FPS), accuracy (mAP), power usage, and resource efficiency.
- In [28], the authors systematically explored the internal architectural design of each YOLO model, models 1 through 8, along with their benchmark performance across standard datasets.
- In [29], the authors reviewed YOLOv1 through YOLOv8 with detailed performance metrics (e.g., inference time and detection accuracy), architectural designs, datasets, and diverse applications.

- In [30], the authors examined the architectural advancements of YOLOv1 through YOLOv11, along with benchmark performance and applications.

2.3. Research Gap and Contribution

Despite the widespread use of YOLO models, there is still limited research that thoroughly tracks their evolution and compares performance across different versions. This gap becomes critical when deploying YOLO on resource-constrained devices such as single-board computers (SBCs). This includes both Raspberry Pi and NVIDIA Jetson, which are commonly used in edge computing. These devices offer affordable and portable AI solutions but have limited processing power. Understanding how various YOLO versions perform in terms of speed, latency, power consumption, and detection accuracy is crucial for real-time applications. This study addresses that need by systematically evaluating YOLO models from a deployment perspective, helping guide practical and efficient model selection for embedded systems and edge AI use cases. The main contributions of this study are as follows:

- Multiple YOLO models are benchmarked with their variants;
- They are compared across PC vs. SBC under the same dataset;
- Accuracy and system metrics (FPS, load time, latency, and FLOPs) are evaluated.

2.4. SBCs

In education and research, SBCs serve as accessible platforms for embedded systems, learning programming, robotics, autonomous vehicles (AVs), electric vehicles (EVs), and artificial intelligence (AI). Raspberry Pi boards, for example, have been extensively used in STEM education, allowing students to experiment with coding, sensor integration, and networking [31]. Meanwhile, Jetson Nano and Jetson Orin are optimized for AI and deep learning applications, enabling object detection, autonomous robotics, and edge computing in real time [32]. Their GPU acceleration capabilities make them particularly valuable for machine learning inference and computer vision tasks.

Beyond academia, SBCs have found applications in Internet of Things (IoT), industrial automation, and smart systems. LattePanda 3 Delta, with its compatibility for both Windows and Linux, bridges the gap between embedded systems and full-scale computing, making it suitable for tasks that require both microcontroller-like control and desktop-level processing power [33]. Industries utilize SBCs for real-time data collection, automation, and remote monitoring, reducing costs while enhancing task efficiency.

As technology advances, SBCs continue to evolve with more powerful processors, with increased RAM capacities and improved AI capabilities. Their affordability and adaptability make them indispensable tools for innovators, researchers, and developers across various fields [32,34,35]. With ongoing improvements, SBCs are expected to drive advancements in smart cities, healthcare, edge computing, and autonomous systems, shaping the future of embedded computing.

Recent advancements in object detection have led to significant improvements across diverse application domains. Wang et al. proposed PG-YOLO, a lightweight YOLOv8-based model optimized for pomegranate fruitlet detection before thinning, incorporating ShuffleNetv2, depthwise separable convolutions, and multi-head self-attention to achieve high accuracy with minimal computational cost [36]. Similarly, Wu et al. introduced YOLO-Claw, an enhanced YOLOv5 variant for chicken claw detection in precision livestock farming, integrating Tanh-aided channel attention (TCA), C3TCA modules, and AFPN-Neck for robust feature extraction under occlusion and lighting variations [37]. In the maritime domain, Chen et al. developed a poly-YOLO-based framework for ship trajectory extraction, combining multi-scale feature aggregation with Enhanced Deep Sort tracking

to improve detection and tracking accuracy in complex sea conditions [38]. For ecological monitoring, Wang et al. proposed WB-YOLO, an improved YOLOv7 model for wild bat detection, incorporating Vision Transformer encoders, deformable convolutions, and hybrid attention mechanisms to handle occlusion and cluttered backgrounds effectively [39]. Finally, Yang et al. introduced Tunnel-YOLO, an advanced YOLOv8-based algorithm for shield tunnel leakage detection, featuring RFCACnv for adaptive receptive fields, C2f_SGE for noise suppression, EFE-Head for fine-grained edge enhancement, and LeShape-IoU for shape-sensitive bounding box regression [40]. Collectively, these studies highlight the trend of tailoring YOLO architectures with attention mechanisms, lightweight modules, and multi-scale fusion strategies to balance accuracy, robustness, and real-time performance across agriculture, livestock, maritime, ecological, and infrastructure domains.

3. Methodology

This section presents a detailed experimental methodology used to benchmark YOLOv8 through YOLOv12 across diverse single-board computing platforms. It comprehensively describes the dataset selection, model configurations, inference setup, hardware environments, operating conditions, and evaluation metrics. In addition, a standardized execution protocol is adopted to ensure fair comparison, consistency, and full reproducibility across all tested platforms.

3.1. Dataset

In this study, the COCO (Common Objects in Context) dataset has been used. It is a widely used benchmark in object detection research due to its rich and diverse collection of annotated images [41]. It contains over 330,000 images, with more than 80 object categories, and includes detailed annotations such as object segmentation, keypoints, and contextual information. COCO is particularly valuable for training and evaluating deep learning models because it presents objects in complex, real-world scenes with varying scales, occlusions, and lighting conditions. In this research, COCO was chosen to assess the performance of different YOLO versions, providing a standardized and challenging environment to measure detection accuracy, speed, and robustness across models.

For evaluation, the COCO128 dataset was used, which consists of 128 images sampled from the COCO train2017 split. This subset provides a controlled and reproducible input set suitable for benchmarking inference performance across different hardware platforms. The focus of this study is on runtime metrics rather than accuracy; therefore, a lightweight and consistent subset was chosen for fair comparison.

It is important to note that the COCO128 subset was used exclusively for inference performance evaluation (e.g., FPS, RAM usage, and load time). Accuracy metrics (mAP) were not computed in this study using COCO128 due to its limited size and non-standard evaluation scope. Instead, the mAP@50–95 values reported in this work are obtained from official published benchmarks of the pretrained YOLO models evaluated on the COCO val2017 dataset. These values are included as reference accuracy indicators for comparative purposes.

3.2. YOLO Models Evaluated

This study evaluates the performance of the latest YOLO object detection models, specifically YOLOv8 through YOLOv12. Each YOLO version includes multiple scaled variants to accommodate different hardware and performance needs. A summary of their sizes is shown in Table 2. All models support transfer learning and can be fine-tuned on custom datasets. YOLO models benefit from strong community support, extensive

documentation, and integration with platforms such as Ultralytics [42], ONNX [43], and TensorRT [44], facilitating rapid prototyping and deployment.

Table 2. YOLO model variants and their parameter sizes (in millions).

Model	n	s	m	l	x
YOLOv8	3.2	11.2	25.9	43.7	68.2
YOLOv9	2.0	7.2	20.1	25.5	58.1
YOLOv10	2.3	7.2	15.4	24.4	29.5
YOLOv11	2.6	9.4	20.1	25.3	56.9
YOLOv12	2.6	9.3	20.2	26.4	59.1

All models used in this study were official pretrained YOLO weights obtained directly from the Ultralytics repository. No models were retrained, fine-tuned, or modified. The models were loaded in their native PyTorch format (.pt files) and evaluated without any conversion to alternative formats. The exact weight files used are listed below:

- YOLOv8: yolov8n.pt, yolov8s.pt, yolov8m.pt, yolov8l.pt, yolov8x.pt
- YOLOv9: yolov9t.pt, yolov9s.pt, yolov9m.pt, yolov9c.pt, yolov9e.pt
- YOLOv10: yolov10n.pt, yolov10s.pt, yolov10m.pt, yolov10l.pt, yolov10x.pt
- YOLOv11: yolo11n.pt, yolo11s.pt, yolo11m.pt, yolo11l.pt, yolo11x.pt
- YOLOv12: yolov12n.pt, yolov12s.pt, yolov12m.pt, yolov12l.pt, yolov12x.pt

All YOLO models (v8–v12) used in this study were evaluated using their official pretrained weights provided by the Ultralytics framework, trained on the COCO dataset. No additional training or fine-tuning was performed, ensuring a consistent and fair comparison of inference performance across platforms.

3.3. Inference Configuration

All models were executed using the native PyTorch framework without conversion to alternative runtimes such as ONNX, TensorRT, or OpenVINO. Pretrained Ultralytics weights (.pt format) were loaded directly and evaluated in FP32 precision. No quantization or mixed-precision techniques were applied.

Inference was performed using a batch size of 1 to reflect realistic real-time deployment scenarios. Input images were processed sequentially from the COCO128 dataset using the default resolution defined by each YOLO model configuration.

GPU acceleration (CUDA) was used on the reference PC, Jetson Nano, and Jetson Orin Nano platforms, while all other devices executed inference on the CPU. This unified setup ensures fair and consistent comparison across platforms.

3.4. Experimental Protocol

To ensure reproducibility, a standardized inference pipeline was used across all platforms. Each model was executed independently under controlled conditions to avoid interference between runs. The evaluation procedure consists of the following steps:

1. **Environment Setup:** The target model variant is selected and loaded with a fixed configuration to ensure consistency across tests.
2. **Model Loading:** Pretrained weights are loaded using the native PyTorch framework, and model initialization time is recorded.
3. **Device Routing:** Inference is executed either on CPU or GPU depending on the hardware platform. CUDA acceleration is enabled for supported devices.

4. Inference Execution: Images from the COCO128 dataset are processed sequentially with a batch size of 1. Total latency per frame includes preprocessing, inference, and postprocessing steps.
5. Resource Monitoring: CPU usage, GPU utilization, and RAM consumption are recorded during inference to evaluate system performance.
6. Cleanup and Thermal Stabilization: After each run, memory is cleared and a cooldown period is applied to minimize thermal effects between experiments.

3.5. Single-Board Computer Systems

Single-board computers (SBCs) have revolutionized the field of embedded systems. They provide compact, cost-effective, and versatile computing solutions for a wide range of applications. These devices integrate essential computing components such as processors, memory, storage, and connectivity onto a single circuit board, making them ideal for both educational and industrial purposes. SBCs like the Raspberry Pi, Jetson Nano, Jetson Orin, and LattePanda have become widely adopted due to their low power consumption, small form factor, and broad operating system support [32–35].

In this research, YOLO models were evaluated on different SBCs, in addition to a PC. The main differences are summarized in Table 3.

Table 3. Comparison of featured single-board computers.

Feature	Raspberry Pi 4B [31]	Raspberry Pi 5 [34]	Jetson AGX Orin [32]	Jetson Nano [35]	LattePanda 3 Delta [33]
CPU	BCM2711, Quad-Core Cortex-A72, 1.5 GHz	BCM2712, Quad-Core Cortex-A76, 2.4 GHz	Custom SoC, 2 GPCs, 16 SMs, 1.5 GHz	Quad-Core Cortex-A57, 1.42 GHz	Intel Celeron N5105, Quad-Core, 2.0 GHz (Boost 2.9 GHz)
GPU	VideoCore VI, OpenGL ES 3.1, H.265 Decoding	VideoCore VII	Ampere-based, 2048 CUDA Cores	Maxwell, 128 CUDA Cores	Intel UHD, 24 Execution Units (EUs)
RAM	2 GB/4 GB/8 GB LPDDR4	4 GB/8 GB LPDDR4X	32 GB/64 GB LPDDR5	4 GB LPDDR4, 1600 MHz	8 GB LPDDR4, 2933 MHz
Storage	microSD Slot	microSD Slot	eMMC with NVMe Support	16 GB eMMC	64 GB eMMC with NVMe Support
Power	5 V/3 A USB-C, 3 W (Overclocked 7 W)	5 V/5 A USB-C, 5 W–12 W	Variable Power Mode, up to 170 W	5 V, 5 W–10 W	12 V/3 A, 6 W–15 W
GPIO	40-pin header, 28 GPIOs	40-pin header, 28 GPIOs	40-pin expansion header, Configurable GPIOs	40-pin header, Configurable GPIOs	ATmega32u4-based GPIO, 20 digital I/O pins
OS Support	Raspberry Pi OS, Linux, NOOBS	Raspberry Pi OS, Linux	Linux, NOOBS	Linux, NOOBS	Windows 10/11, Linux

3.5.1. PC

A desktop PC was used as the reference system for the experimental setup, equipped with an NVIDIA GeForce RTX 3060 Ti (8 GB), 48 GB of RAM, and an AMD Ryzen 5 5600X 6-core processor running at 3.70 GHz.

3.5.2. Raspberry Pi 4—Model B

Raspberry Pi 4 Model B is a compact yet capable single-board computer powered by a Broadcom BCM2711 quad-core Cortex-A72 CPU running at 1.5 GHz. It features a VideoCore VI GPU with support for OpenGL ES 3.1 and H.265 video decoding, making it suitable for lightweight graphics and media tasks. Available in 2 GB, 4 GB, and 8 GB LPDDR4 RAM variants, this study utilizes the 8 GB model for enhanced performance. Storage is handled via a microSD card, and the board operates on a 5 V/3 A USB-C power supply, typically consuming around 3 W, with the ability to scale up to 7 W depending on the power mode. It includes a 40-pin header with 28 GPIO pins for hardware interfacing and supports multiple operating systems, including Raspberry Pi OS, NOOBS, and other Linux distributions [31].

3.5.3. Raspberry Pi 5—Model B

Raspberry Pi 5 Model B brings a significant upgrade over its predecessor, featuring a Broadcom BCM2712 quad-core Cortex-A76 CPU running at 2.4 GHz and a VideoCore VII GPU with enhanced support for OpenGL ES 3.1 and hardware-accelerated H.265 decoding. They are available in 4 GB and 8 GB LPDDR4X RAM variants. This board features a PCIe 2.0 interface, enabling faster storage options beyond the traditional microSD card. It operates on a 5 V/5 A USB-C power supply, with power consumption ranging from 5 W to 12 W depending on the selected mode—low, standard, or high power. Like previous models, it is equipped with a 40-pin header, in addition to 28 GPIO pins. This board supports Raspberry Pi OS along with other Linux distributions [34].

3.5.4. NVIDIA Jetson AGX Orin

According to the NVIDIA Jetson AGX Orin datasheet [32], the Jetson AGX Orin 32 GB edition features a System-on-Chip (SoC) with two Graphic Processing Clusters (GPCs), each containing 8 Texture Processing Clusters (TPCs) and 16 Streaming Multiprocessors (SMs). Each SM includes 192 KB of L1 cache and 4 MB of L2 cache, with 128 CUDA cores and four 3rd-generation Tensor cores. It doubles the CUDA cores compared to the Volta architecture, where the 64 GB variant offers 2048 CUDA cores and 64 Tensor cores, delivering up to 170 Sparse TOPs of INT8 Tensor compute and 5.3 FP32 TFLOPs of CUDA compute. In comparison, the 32 GB model includes 1792 CUDA cores and 56 Tensor cores, achieving up to 108 Sparse TOPs of INT8 and 3.37 FP32 TFLOPs. The board operates at a frequency of 1.5 GHz and supports RAM configurations of 2 GB or 8 GB. It also features a 40-pin GPIO header and runs on Linux-based operating systems. This board is the only board in that comparison that was not used in this study, but the specs were placed for comparison.

3.5.5. NVIDIA Jetson Nano

The Jetson Nano board has a Quad-Core ARM Cortex-A57 64-bit with a 1.42 GHz CPU. The board is equipped with NVIDIA Maxwell 128 CUDA cores and a GPU (921 MHz). The board contains 4 GB LPDDR4 RAM with a frequency of 1600 MHz. It supports up to 25.6 GB/s of data transfer. The Jetson Nano has 16 GB of eMMC storage. The board consumes power ranging from 5 W to 10 W based on the power mode. The board supports diverse peripherals, including a 40-pin GPIO and a MIPI port for camera [35]. The NVIDIA Jetson Nano with 4 GB has been evaluated in this study.

3.5.6. LattePanda

The LattePanda 3 Delta has an Intel Celeron N5105 quad-core 2.0 GHz CPU (can be overclocked to 2.9 GHz) and a built-in ATmega 32U4 co-processor. The board is equipped with Intel UHD Graphics and a 24 Execution Unit (EU) GPU, supporting DirectX 12,

OpenGL 4.6, and OpenCL 3.0. The board has 8 GB of LPDDR4 RAM clocked at 2933 MHz. Also, 64 GB of eMMC storage. LattePanda 3 Delta requires a 12V/3A input, with power consumption ranging from 6 W to 15 W, based on the operating power mode. The board has 20-pin GPIO. The board is compatible with Windows 10, Windows 11, and various Linux distributions [33].

3.6. Evaluation Metrics

To comprehensively assess the performance and deployment suitability of YOLOv8 through YOLOv12 models on edge devices, we employ a diverse set of evaluation metrics that capture both computational efficiency and detection quality. These metrics include Frames Per Second (FPS) for inference speed, Mean Average Precision (mAP) for detection accuracy, RAM utilization for memory efficiency, Floating Point Operations (FLOPs) for computational complexity, and model load time for initialization latency.

Each metric provides a unique perspective to evaluate the trade-offs between critical factors when deploying models on SBCs: speed, accuracy, and resource consumption. By analyzing these metrics across different YOLO variants and power modes, we aim to identify optimal configurations for real-time object detection in edge computing scenarios.

It is important to note that the power consumption values reported in this study are estimated rather than directly measured. These estimates are based on manufacturer specifications and documented operating ranges for each device under different power modes. While this approach enables consistent comparison across platforms, actual power consumption may vary depending on workload characteristics, thermal conditions, and system-level factors.

3.6.1. Precision

It measures the proportion of correctly predicted bounding boxes (true positives) out of the total predicted boxes, including false positives, as shown in Equation (1). It quantifies the model accuracy toward making correct predictions.

$$Precision = \frac{TruePositives(TP)}{TruePositives(TP) + FalsePositives(FP)} \quad (1)$$

3.6.2. mAP@50-95

Mean Average Precision (mAP@50-95) is a comprehensive metric used to evaluate object detection performance across a range of Intersection over Union (IoU) thresholds. In contrast to traditional mAP@50, which considers a single IoU threshold, mAP@50-95 averages the precision over ten thresholds from 0.50 to 0.95 in steps of 0.05. This provides a more robust assessment of a model's localization accuracy and generalization capability. It is especially beneficial for benchmarking models on challenging datasets and is the standard metric used in the COCO evaluation protocol.

$$mAP_{50:95} = \frac{1}{10} \sum_{t=0.50}^{0.95} AP_t \quad (2)$$

where AP_t is the Average Precision at IoU threshold t , and the summation is taken over thresholds $t = 0.50, 0.55, \dots, 0.95$.

3.6.3. Latency

Latency measures the time consumed during model processing of the image and performing the predictions, including all the steps required for the detection process, such as image processing, model inference, and postprocessing. It is typically measured in milliseconds (ms). Lower latency is crucial for real-time applications, such as autonomous

navigation, where fast and accurate detection is mandatory. Moreover, Frames Per Second (FPS) can also be used to complement latency, which indicates how many frames can be processed by the model in one second. Together, latency and FPS provide a comprehensive understanding of the model's speed and performance in real-world scenarios.

3.6.4. Load Time

Load time (in milliseconds): It is the time taken to load a model or data into memory from disk. It includes reading the model file from disk, deserializing it, and initializing weights and layers in RAM or GPU memory. It is important in real-time applications. Load time is influenced by disk I/O speed, model size, system memory bandwidth, and CPU/GPU initialization overhead.

To ensure the reliability of the reported results, each experiment was repeated a minimum of three times for every board, model, and power mode configuration. Accuracy metrics remained identical across runs due to deterministic inference with fixed pretrained weights. Runtime metrics, including FPS and RAM usage, exhibited only minor variations within $\pm 1.8\%$, confirming the stability and consistency of the measurements. All experiments were conducted under controlled conditions, including identical system configurations, minimized background processes, and stabilized thermal environments.

3.7. Operating Systems

Table 4 displays all the operating systems used across all devices in this study.

Table 4. Operating systems for different devices.

Device	Operating System
PC	Windows 11 Version 23H2
Raspberry Pi 4 and 5	Debian GNU/Linux 12 (bookworm)
LattePanda	Windows 10 Pro (22H2)
NVIDIA Jetson Nano	Nvidia Jetpack 4.6.6
NVIDIA Orin	Ubuntu 20.04.6 LTS (Focal Fossa)

3.8. Power Modes

Most of the SBCs today have three different power modes. SBCs incorporate configurable power management modes to optimize energy consumption, thermal performance, and computational efficiency based on workload demands. These modes are typically the performance, balanced, and power-saving modes, each governed by dynamic voltage and frequency scaling algorithms. In performance mode, the processor operates at its maximum clock speed, prioritizing computational throughput at a higher power dissipation. Conversely, power-saving mode restricts clock frequencies to minimize energy usage, making it suitable for battery-operated or thermally constrained environments. A balanced mode dynamically adjusts CPU performance in response to real-time workload requirements, offering a compromise between efficiency and speed. Additionally, thermal throttling mechanisms may forcibly reduce performance to prevent overheating, ensuring hardware longevity [45–47].

Testing different YOLO versions and models on different power modes is a game-changer for researchers. For example, two of the main goals for drones, rovers, or any battery-powered robotic vehicle are battery lifetime on the robot and the performance of the selected SBC, given the running tasks. Developers tend to choose a high-performance SBC (such as the Jetson Nano and Orin) to ensure real-time performance, along with one of the YOLO nano models to ensure a minimum computational power. Including such power

modes in the study enables the researchers to make better choices for both the SBC and YOLO model, with its variants.

3.8.1. Raspberry Pi 4 Power Modes

The Raspberry Pi 4 configuration setting has three configurable power modes. The Low Power Mode (600 MHz CPU/200 MHz GPU) undervolts the SoC (−8 offset) with a 70 °C thermal limit, ideal for always-on applications. The revised Normal Mode (1500 MHz CPU/500 MHz GPU) enforces stock voltages (0 offset), a higher 85 °C thermal limit, and locked GPU clocks (500 MHz min/max), offering stable performance for general computing tasks without active cooling. For demanding workloads, High Power Mode (1800 MHz CPU/600 MHz GPU) employs +2 overvolting and an 80 °C limit, requiring active cooling. Raspberry Pi Dual Fans 5V DC with Heatsink Cooling System was installed to boost the board's performance.

3.8.2. Raspberry Pi 5 Power Modes

Similar to Raspberry Pi 4, Raspberry Pi 5 settings can be configured to create three power modes, each tailored for specific performance and efficiency trade-offs. The Low Power Configuration (600 MHz CPU/200 MHz GPU) minimizes energy consumption by underclocking the CPU and GPU while reducing voltage (−8 offset), making it ideal for lightweight, always-on tasks where thermal management is critical ($\text{temp_limit} = 70\text{ }^{\circ}\text{C}$). The Normal Power Configuration (2400 MHz CPU/500 MHz GPU) operates at default voltages and clocks, balancing performance and power efficiency for general-purpose workloads ($\text{temp_limit} = 85\text{ }^{\circ}\text{C}$). Finally, the High Performance Configuration (2900 MHz CPU, 1100 MHz GPU, $\text{force_turbo} = 1$) pushes the hardware to its limits with overvolting (+6 CPU, +4 SDRAM) and elevated thermal thresholds ($\text{temp_limit} = 90\text{ }^{\circ}\text{C}$), unlocking maximum throughput for compute-intensive applications—though active cooling is mandatory to sustain stability.

3.8.3. LattePanda Power Modes

For LattePanda, the power modes tested were Power Saver, Balanced, and High Performance, which were set through Windows settings.

3.8.4. Jetson Nano Power Modes

The Jetson Nano can operate in two primary power modes. The default mode is the 5 W configuration, which prioritizes energy efficiency by reducing CPU/GPU clock speeds, making it suitable for lightweight or thermally constrained applications. The alternative 10 W mode unlocks higher performance by increasing power limits. This requires active cooling to prevent thermal throttling; this is one of the main reasons why fans were installed on the experimental SBCs. The Jetson Nano's hardware enforces a strict 10 W ceiling, limiting sustained performance. This simpler power management scheme reflects the Nano's entry-level positioning, where thermal design and power delivery are optimized for cost and simplicity rather than scalability. An attachment to the board was fixed. Raspberry Pi Active Cooler, is an official, fan-based cooling solution designed specifically for the RaspberryPi 5, that addresses its higher thermal demands compared to earlier models.

3.8.5. Jetson Orin Nano Power Modes

For Jetson Orin Nano, there are three available power modes [48]. The first power mode is MODE_7W, which provides the lowest power with reduced performance. Followed by MODE_15W, which is the default mode (balanced performance over power). Lastly,

MODE_25W power mode, which is the max performance, draws power up to 25 W (no strict Thermal Design Power (TDP) cap, but thermals limit sustained power).

In conclusion, Raspberry Pi 4 and 5 are the only two boards equipped with coolers, and their performance mode has to be manually set. The three other boards have built-in configurations for optimizing power modes; these configurations are summarized in Table 5.

Table 5. Power mode configurations.

Board and Mode	Settings
RPi 4	
- Low Power	600 MHz CPU, 200 MHz GPU
- Normal	1500 MHz CPU, 500 MHz GPU
- High Power	1800 MHz CPU, 600 MHz GPU
RPi 5	
- Low Power	600 MHz CPU, 200 MHz GPU
- Normal	2400 MHz CPU, 500 MHz GPU
- High Perf.	2900 MHz CPU, 1100 MHz GPU
LattePanda	
- Power Saver	Windows efficiency mode
- Balanced	Windows default
- High Perf.	Windows max performance
Jetson Nano	
- 5 W Mode	Reduced clocks
- 10 W Mode	High performance
Orin Nano	
- 7 W Mode	Low power
- 15 W Mode	Default
- 25 W Mode	Up to 25 W

4. Results and Discussions

This section presents a detailed analysis of the experimental results obtained from benchmarking YOLOv8 through YOLOv12 across multiple single-board computing platforms. The discussion focuses on performance trade-offs in terms of inference speed, accuracy, memory usage, computational complexity, and power efficiency under different hardware configurations and power modes.

4.1. FPS Analysis

In the context of this study, the term “real time” is defined based on application-specific requirements rather than a single fixed threshold. Traditional real-time video processing typically requires frame rates of 25–30 FPS. However, in edge computing and resource-constrained systems, especially in robotics and IoT applications, lower frame rates may still be considered acceptable depending on the use case.

Accordingly, we categorize performance levels as follows: high real-time performance (≥ 10 FPS), near real-time (3–10 FPS), and low or non-real-time (< 3 FPS). This classification reflects practical deployment scenarios where tasks such as monitoring, event detection, or periodic analysis may tolerate lower frame rates.

Frames Per Second (FPS) is a critical metric when evaluating the real-time inference capability of object detection models. This is vital for edge computing environments where computational resources are limited. In this study, benchmarking was performed on YOLO versions 8 through 12 across a PC and multiple SBCs under various power modes. This section presents a detailed comparison of FPS results, offering insights into

the most efficient YOLO configurations for real-time edge deployment. Mainly, FPS varies according to

- Inference time: larger models take longer to process each frame, reducing FPS.
- Memory usage: bigger models use more VRAM and RAM, which can cause bottlenecks or even paging.
- GPU utilization: smaller models may not fully utilize a powerful GPU, but they run faster due to lower compute needs.
- Batch size and resolution: bigger models may slow down when running inference in high-resolution frames or large batches.

It is important to note that lower frame rates (e.g., 3 FPS) may be acceptable in specific applications such as periodic monitoring or event-triggered detection or swarm application [49], where a big task can be broken down into multiple subtasks for agents to accomplish but do not meet the requirements of continuous real-time processing.

4.1.1. FPS Analysis on RPi4

Different YOLO models were evaluated on different power modes: low, standard, and high. As shown in Table 6, FPS results showed a consistent trend as low power mode significantly reduced performance, while standard and high-power modes yielded nearly identical results. Moreover, for all YOLO versions (v8 through v12), the nano (n) model performs best, reaching up to ~0.96 FPS in high power modes, while the extra-large (x) models drop down to as low as ~0.04 FPS.

Interestingly, the standard- and high-power modes show negligible differences, suggesting that the RPi4 may already be operating near its thermal or architectural limits in standard mode. In contrast, low-power mode consistently cuts FPS by more than half, confirming that CPU/GPU throttling directly impacts inference speed. Additionally, YOLOv10 and YOLOv11 stand out slightly for their better performance when their nano and small variants are used. YOLOv9 shows slightly lower FPS overall, possibly due to architectural changes or added complexity. These results highlight the importance of selecting a lightweight model and optimal power settings when deploying real-time inference on RPi4.

Table 6. FPS performance of YOLO models on Raspberry Pi 4.

FPS on RPi4 (Standard/Low/High)					
Model	n	s	m	l	x
YOLOv8	0.95/0.34/0.97	0.34/0.12/0.34	0.13/0.05/0.13	0.06/0.02/0.06	0.04/0.02/0.04
YOLOv9	0.79/0.25/0.80 (t)	0.33/0.11/0.34	0.13/0.05/0.13	0.09/0.03/0.09 (c)	0.05/0.02/0.05 (e)
YOLOv10	0.96/0.43/0.97	0.37/0.12/0.37	0.16/0.05/0.16	0.08/0.03/0.08	0.06/0.02/0.06
YOLOv11	0.93/0.34/1.00	0.32/0.13/0.38	0.109/0.05/0.13	0.087/0.04/0.10	0.044/0.02/0.05
YOLOv12	0.78/0.22/0.84	0.32/0.12/0.34	0.11/0.05/0.12	0.08/0.04/0.09	0.04/0.02/0.04

4.1.2. FPS Analysis on RPi5

Different YOLO models were evaluated on various power modes: low, standard, and high. As shown in Table 7, the FPS results on Raspberry Pi 5 show a clear performance improvement over those of Raspberry Pi 4, with the YOLOv8 to YOLOv12 models achieving significantly higher frame rates, especially in high-power mode. Across all versions, nano (n) and small (s) models consistently deliver the best performance, with YOLOv9 reaching up to 2.39 FPS in high-power mode. Interestingly, low-power mode on the RPi5 performs much closer to standard mode, unlike on the RPi4, where it caused a steep drop in FPS. This suggests better power efficiency and thermal management on the RPi5. YOLOv9 and YOLOv10 stand out for their strong performance across all model sizes, while YOLOv12

results had some inconsistencies; its nano model performs better in low-power mode than in standard, which may indicate model-specific optimizations or measurement anomalies. Overall, the RPi5 enables more viable real-time inference for lightweight YOLO models, especially when paired with high-power mode.

Table 7. FPS performance of YOLO models on Raspberry Pi 5.

FPS on RPi5 (Standard/Low/High)					
Model	n	s	m	l	x
YOLOv8	1.78/1.7/2.88	0.68/0.68/1.12	0.31/0.26/0.47	0.23/0.13/0.21	0.12/0.098/0.15
YOLOv9	2.06/1.42/2.39 (t)	0.642/0.53/0.92	0.28/0.24/0.43	0.26/0.19/0.32 (c)	0.13/0.098/0.165 (e)
YOLOv10	2.04/1.36/2.32	0.89/0.68/1.06	0.42/0.31/0.49	0.227/0.16/0.25	0.17/0.12/0.15
YOLOv11	1.968/1.52/2.11	1.05/0.675/1.2	0.39/0.247/0.438	0.31/0.19/0.35	0.15/0.098/0.2
YOLOv12	0.66/1.3/2	0.872/0.6/0.86	0.34/0.34/0.244	0.24/0.155/0.25	0.12/0.89/0.12

4.1.3. FPS Analysis on NVIDIA Jetson CPU

The FPS results for YOLO models executed on the NVIDIA Jetson Nano CPU under two power modes (5 W and 10 W) are summarized in Table 8. The results demonstrate consistently limited inference performance across all evaluated YOLO versions (v8–v12), highlighting the computational constraints of CPU-only execution on this embedded platform.

At 5 W mode, FPS values range from approximately 0.02 FPS for larger models (e.g., YOLOv8x) to around 0.53 FPS for lightweight variants such as YOLOv11n. At 10 W mode, moderate improvements are observed, with the highest FPS reaching 0.97 FPS for YOLOv10n and 0.96 FPS for YOLOv11n.

A clear and consistent trend is observed across all YOLO families: inference speed decreases significantly as model complexity increases. Nano (n) and small (s) models achieve the highest performance, while medium (m), large (l), and extra-large (x) variants exhibit substantial degradation, frequently falling below 0.1 FPS even in 10 W mode.

Table 8. FPS on NVIDIA Jetson CPU.

FPS on NVIDIA Jetson CPU (5 W/10 W)					
Model	n	s	m	l	x
YOLOv8	0.48/0.69	0.19/0.31	0.08/0.12	0.04/0.06	0.02/0.04
YOLOv9	0.41/0.74 (t)	0.17/0.30	0.07/0.11	0.05/0.08 (c)	0.03/0.04 (e)
YOLOv10	0.52/0.97	0.21/0.36	0.09/0.15	0.05/0.07	0.04/0.06
YOLOv11	0.53/0.96	0.21/0.35	0.08/0.12	0.06/0.09	0.03/0.04
YOLOv12	0.43/0.83	0.18/0.33	0.07/0.11	0.05/0.08	0.03/0.04

These results indicate that the Jetson Nano CPU is only capable of supporting lightweight object detection workloads and remains unsuitable for both real-time (≥ 10 FPS) and near real-time (3–10 FPS) applications.

4.1.4. FPS Analysis on NVIDIA Jetson GPU

The FPS results for YOLO models utilizing GPU acceleration on the NVIDIA Jetson Nano under 5 W and 10 W power modes are presented in Table 9. Compared to CPU-only execution, GPU acceleration provides a clear and measurable improvement in inference performance across all evaluated models.

At 5 W mode, lightweight nano (n) variants achieve the highest performance, with YOLOv10n reaching approximately 3.03 FPS and YOLOv11n achieving around 3.00 FPS. Other nano variants such as YOLOv8n and YOLOv12n operate within the range of 2.6 to 2.8 FPS.

At 10 W mode, performance remains within a similar range, with YOLOv10n achieving approximately 2.76 FPS and YOLOv11n reaching around 2.92 FPS. However, performance does not scale consistently with increased power. In several cases, the 5 W configuration slightly outperforms 10 W mode, suggesting the presence of thermal constraints or system-level overhead limiting performance gains.

Table 9. FPS performance of YOLO models on NVIDIA Jetson Nano GPU.

FPS on Jetson Nano GPU (5 W/10 W)					
Model	n	s	m	l	x
YOLOv8	2.63/2.44	1.87/1.70	0.82/0.96	0.49/0.64	0.31/0.44
YOLOv9	2.19/2.10 (t)	1.32/1.52	0.78/1.01	0.68/0.89 (c)	0.35/0.46 (e)
YOLOv10	3.03/2.76	1.89/2.04	0.93/1.16	0.59/0.79	0.43/0.57
YOLOv11	3.00/2.92	1.93/2.08	0.98/1.21	0.73/0.97	0.41/0.58
YOLOv12	2.79/2.79	1.76/1.94	0.93/1.16	0.67/0.84	0.39/0.50

Across all configurations, smaller models (n and s) significantly outperform larger variants. Medium (m), large (l), and extra-large (x) models exhibit notable performance degradation, often dropping below 1 FPS.

Overall, GPU acceleration provides an approximate 2×–4× speedup compared to CPU execution for lightweight models. Despite this improvement, performance remains below the threshold required for real-time applications (≥ 10 FPS) and only marginally approaches near real-time capability at approximately 3 FPS.

These findings indicate that while the Jetson Nano GPU improves inference efficiency, hardware limitations—particularly the limited number of CUDA cores and memory bandwidth—restrict its ability to sustain real-time object detection for modern YOLO architectures without further optimization.

4.1.5. FPS Analysis on LattePanda

LattePanda, despite its $\times 86$ architecture and relatively high clock speed, consistently underperformed in real-time inference tasks. Across all YOLO versions (v8–v12) and model sizes (n through x), the FPS remained below 1, even when operating in high-performance mode. This indicates that the board lacks the necessary acceleration capabilities—such as a dedicated GPU or optimized inference libraries—to support real-time object detection workloads. Moreover, during testing, YOLO models frequently crashed on LattePanda, particularly when attempting to run medium to large variants (m, l, and x). These crashes were linked to a combination of factors:

- Insufficient GPU support: The integrated Intel UHD Graphics lacks CUDA cores and TensorRT compatibility, which are essential for optimized YOLO inference.
- Memory bottlenecks: Although the board has 8 GB of RAM, the lack of dedicated VRAM and shared memory architecture led to frequent memory allocation failures.
- Framework limitations: The PyTorch-based execution environment, combined with limited hardware acceleration support on LattePanda, resulted in instability when running larger YOLO models, leading to runtime failures in some cases.

Due to these limitations, YOLO deployment on LattePanda was not feasible for real-time applications, and the board is not recommended for edge AI tasks involving deep learning-based object detection without external GPU support.

4.1.6. FPS Analysis on Orin

The FPS results for YOLO models executed on the Jetson Orin Nano using CPU-only inference across three power modes (7 W, 15 W, and 25 W) are presented in Table 10. These results provide a baseline for evaluating the performance contribution of GPU acceleration.

As shown in Table 10, CPU-only inference on the Jetson Orin Nano yields consistently low FPS across all YOLO versions and model scales. Lightweight nano (n) models achieve the highest throughput, reaching approximately 1.9–2.3 FPS in 7 W and 25 W modes for YOLOv8, YOLOv10, and YOLOv11. However, even under optimal conditions, the performance remains below the near real-time threshold (≥ 3 FPS).

A clear scalability trend is observed across all model variants: inference speed decreases significantly as model complexity increases. Small (s) models operate around 0.6–0.9 FPS, while medium (m), large (l), and extra-large (x) models exhibit substantial degradation, frequently falling below 0.5 FPS. These results confirm that CPU-only execution is insufficient for handling modern YOLO architectures on embedded platforms.

The influence of power modes on CPU inference performance is limited and non-linear. While the 25 W mode generally achieves the highest FPS, the 15 W configuration occasionally exhibits lower performance compared to 7 W mode. This behavior may be attributed to dynamic frequency scaling, scheduling overhead, and thermal conditions, which can affect CPU performance at intermediate power settings.

Table 10. FPS performance of YOLO models on Orin.

FPS on Orin (7 W/15 W/25 W)					
Model	n	s	m	l	x
YOLOv8	1.902/0.08/2.04	0.73/0.5/0.812	0.3/0.25/0.32	0.15/0.13/0.17	0.1/0.08/0.11
YOLOv9	0.11/-/- (t)	0.66/0.55/0.74	0.28/0.26/0.31	0.22/0.19/0.25 (c)	NA/NA/0.13 (e)
YOLOv10	2.11/1.62/2.33	0.86/0.73/0.96	0.37/0.31/0.42	0.19/0.21/0.216	0.14/0.16/0.163
YOLOv11	2.11/1.7/2.3349	0.87/0.73/0.96	0.32/0.281/0.363	0.25/0.2166/0.2839	0.12/0.1238
YOLOv12	1.79/1.39/1.997	0.78/0.64/0.86	0.308/0.264/0.34475	0.22/0.19/0.25	0.113/-/0.126

Compared to the Jetson Nano CPU results, the Orin Nano CPU shows a modest improvement due to its more advanced architecture and higher core performance. However, the overall throughput remains inadequate for real-time or near real-time applications.

These findings highlight that, despite the architectural improvements of the Jetson Orin Nano, CPU-only inference remains a significant bottleneck. Efficient deployment of YOLO models on this platform requires GPU acceleration to achieve practical performance levels, particularly for real-time edge applications.

4.1.7. FPS Analysis on Orin GPU

The FPS results for YOLO models executed on the Jetson Orin Nano GPU under three power configurations (7 W, 15 W, and 25 W) are presented in Table 11. The Jetson Orin Nano demonstrates a substantial improvement in inference performance compared to the Jetson Nano platform, achieving real-time performance (≥ 10 FPS) for several lightweight models, while larger models remain within near real-time or sub-real-time ranges. Across all evaluated configurations, nano (n) and small (s) models consistently provide the highest throughput, while performance degrades progressively with increasing model size.

Among the evaluated YOLO versions, YOLOv10 and YOLOv8 emerge as the top-performing architectures. In particular, YOLOv10n achieves the highest FPS, reaching up to 13.15 FPS in 25 W mode, followed closely by YOLOv8n and YOLOv11n. Similarly, small variants (s) of YOLOv8 and YOLOv10 achieve stable real-time performance across all power modes.

Table 11. FPS performance of YOLO models on Jetson Orin Nano GPU.

FPS on Orin (7 W/15 W/25 W)					
Model	n	s	m	l	x
YOLOv8	12.29/11.64/13.20	11.62/11.09/12.81	8.92/9.58/10.80	7.05/7.83/8.71	5.22/6.08/6.53
YOLOv9	8.42/7.87/9.14 (t)	8.32/7.74/9.24	7.25/7.73/8.84	7.10/7.67/8.70 (c)	4.14/4.64/5.13 (e)
YOLOv10	12.52/11.93/13.15	11.91/11.31/12.87	9.46/9.90/11.30	7.59/8.28/9.29	6.22/7.01/7.63
YOLOv11	11.11/10.23/11.34	10.31/10.00/11.08	8.89/9.25/10.52	7.68/8.09/9.21	5.46/6.18/6.75
YOLOv12	9.51/8.72/10.09	9.11/8.58/10.10	7.59/8.16/9.17	5.69/6.38/7.13	4.05/4.68/5.08

A consistent scaling trend is observed across model sizes. Medium models (m) operate within the range of approximately 8–11 FPS, placing them in the lower bound of real-time performance. In contrast, large (l) and extra-large (x) variants exhibit a gradual performance decline, with FPS values typically ranging between 5–9 FPS and 4–7 FPS, respectively, making them less suitable for strict real-time constraints.

Interestingly, the results reveal a non-linear relationship between power modes and performance. While the 25 W mode generally delivers the highest FPS, the 7 W mode occasionally matches or slightly outperforms the 15 W mode, particularly for nano models. This behavior may be attributed to dynamic voltage and frequency scaling (DVFS), thermal conditions, or scheduling overhead, which can affect performance under intermediate power settings.

Overall, the Jetson Orin Nano GPU significantly outperforms all other evaluated SBCs, providing reliable real-time inference for lightweight YOLO models and near real-time performance for larger variants. These findings confirm that the Orin platform is well-suited for high-throughput edge AI applications, particularly when paired with optimized YOLO configurations such as YOLOv10n and YOLOv8n.

4.1.8. FPS Analysis Summary

- Jetson Orin GPU (25 W) clearly dominates, achieving over 13 FPS for YOLOv8n and YOLOv10n
- Jetson Nano GPU shows moderate improvement over CPU execution, achieving approximately 2–3 FPS for nano models, which approaches the lower bound of near real-time performance but remains below practical real-time requirements for most applications.
- RPi5 shows a significant improvement over RPi4, but both remain under 3 FPS, suitable only for lightweight models
- LattePanda lags, with FPS under 1, indicating limited real-time capability without acceleration.
- Jetson CPU-only mode performs better than RPi4 but still far from real time.

4.2. mAP Performance

The Mean Average Precision (mAP) is a standard metric used to evaluate object detection accuracy and is inherently independent of hardware performance. Since this study focuses on benchmarking inference performance across different hardware platforms, accuracy was not experimentally evaluated.

The mAP@50–95 values reported in Table 12 are obtained from official published benchmarks of the pretrained YOLO models evaluated on the COCO val2017 dataset. These values are included as reference indicators to provide context on the accuracy–performance trade-off across different YOLO variants.

It is important to emphasize that these mAP values were not computed as part of the experimental protocol of this study and should not be interpreted as results derived from the COCO128 dataset or the proposed benchmarking pipeline.

As shown in Table 12, the accuracy of YOLO models generally improves with both architectural advancement and model scale. Larger variants (l and x) achieve higher mAP values due to increased representational capacity, while newer versions such as YOLOv12 demonstrate incremental improvements over earlier architectures. These reference values provide important context when interpreting the performance trade-offs discussed in this study, particularly in balancing accuracy against computational cost, memory usage, and inference speed on resource-constrained devices.

Table 12. Reference mAP@50–95 values of YOLO models on the COCO val2017 dataset.

Model	mAP50-95				
	n	s	m	l	x
YOLOv8	37.3	44.9	50.2	52.9	53.9
YOLOv9	39.5 (t)	47	51.5	53.4 (c)	54.7 (e)
YOLOv10	39.5	46.8	51.3	53.4	54.4
YOLOv11	39.5	47	51.5	53.4	54.7
YOLOv12	40.6	48	52.5	53.7	55.2

4.3. RAM Utilization

RAM utilization is a critical factor when deploying deep learning models on resource-constrained edge devices, as it directly affects model scalability, system stability, and the ability to handle real-time workloads. In this study, the peak RAM consumption of YOLOv8 through YOLOv12 was measured across multiple platforms, including Raspberry Pi 4/5, Jetson Nano, Jetson Orin, and LattePanda, under different operating modes.

Table 13 presents a compact summary of RAM usage across different YOLO model scales. Rather than listing all individual measurements, the table provides average and maximum memory utilization along with deployment-oriented observations.

Table 13. Summary of RAM usage across YOLO model scales on SBC platforms. Values represent approximate ranges observed across multiple devices; actual utilization varies depending on platform characteristics (CPU vs. GPU execution).

Model Scale	Avg RAM (%)	Max RAM (%)	SBC Feasibility	Observation
YOLO-n	4–7	~7	Fully feasible	No memory limitations observed across all devices
YOLO-s	5–10	~11	Feasible	Low memory overhead with stable execution
YOLO-m	40–46	~46	Moderately feasible	Significant increase in memory usage, approaching device limits
YOLO-l	44–46	~47	Conditionally feasible	High memory footprint limits performance on low-end SBCs
YOLO-x	45–50	>50	Resource-constrained	High risk of memory exhaustion and execution failure on constrained devices

As shown in Table 13, memory consumption increases substantially with model complexity. Lightweight models (YOLO-n and YOLO-s) exhibit low memory utilization on CPU-only platforms, typically remaining below 10% of available RAM. However, on GPU-enabled devices such as Jetson Nano and Jetson Orin, overall memory usage is higher

due to CUDA-related allocations and the unified memory architecture, where CPU and GPU share system memory. As a result, total RAM utilization can reach higher percentages even for lightweight models. This increase reflects system-level behavior rather than model complexity. In contrast, medium and large models (YOLO-m, YOLO-l, and YOLO-x) exhibit significantly higher memory demands, frequently exceeding 40% of available RAM.

This trend becomes more pronounced on GPU-enabled platforms such as Jetson Nano and Jetson Orin, where additional memory overhead is introduced by CUDA-based execution, intermediate tensor storage, and the unified memory model. On GPU-enabled platforms such as Jetson Nano and Jetson Orin, RAM utilization is higher due to CUDA-related memory allocation, intermediate tensor storage, and the unified memory architecture, where CPU and GPU share system memory.

RAM utilization varies across platforms, particularly between CPU-only and GPU-enabled devices, and the values in Table 13 should be interpreted as general trends rather than device-specific measurements.

Notably, YOLO-x models demonstrate the highest memory demand, often exceeding 50% and, in some cases, leading to execution instability or runtime termination on devices with limited memory capacity, such as Raspberry Pi and Jetson Nano. This observation is supported by recorded process termination events at approximately 3.5 GB memory usage, confirming memory-induced scalability limitations for large models.

These results highlight that memory availability is a key bottleneck in deploying high-capacity object detection models on edge devices. Therefore, selecting an appropriate YOLO variant requires balancing accuracy gains against memory constraints, particularly in low-power and embedded environments.

The practical implications of RAM utilization can be further understood by considering real-world application scenarios. For instance, in our prior work on lightweight speaker verification for autonomous vehicles [50], the deployed models are specifically designed to operate under strict latency constraints while maintaining minimal memory usage. Based on the results presented in Table 13, such lightweight models (e.g., YOLO-n variants) are well-suited for all evaluated SBC platforms, offering stable execution with low memory overhead.

In contrast, more complex systems that integrate multiple computational components, such as the presented work on the fusion of robotics, AI, and thermal imaging for precision agriculture [51], impose significantly higher memory demands. These systems typically require handling multiple input streams, sensor fusion pipelines, and higher-resolution processing, resulting in substantial RAM consumption. Under such conditions, deploying large-scale models (e.g., YOLO-m, YOLO-l, or YOLO-x) on resource-constrained SBCs becomes challenging and may lead to performance degradation or system instability.

These observations emphasize that RAM availability is not only a hardware limitation but also a key factor in determining the feasibility of deploying different classes of intelligent systems. Lightweight applications can achieve reliable real-time performance across a wide range of devices, whereas complex multi-modal systems require careful resource planning or more capable hardware platforms.

4.4. FLOPs and Computational Load

Floating Point Operations (FLOPs) serve as a hardware-independent metric to quantify the computational complexity of deep learning models. They represent the total number of arithmetic operations a model performs during inference or training and are determined by architectural factors such as the number of layers, kernel sizes, and input resolution. In this study, we use FLOPs to assess the relative computational demands of YOLOv8 through YOLOv12 across different model variants (n, s, m, l, and x). Models with lower FLOPs,

such as YOLOv8n and YOLOv10n, are optimized for speed and efficiency, making them ideal for deployment on edge devices like Raspberry Pi and Jetson Nano. These models offer faster inference times and reduced energy consumption, albeit with a trade-off in detection accuracy. Conversely, models with higher FLOPs, such as YOLOv12x, deliver improved accuracy but require significantly more computational resources, making them better suited for GPU-accelerated platforms like Jetson Orin or for cloud-based inference. A summary of FLOPs of different YOLO models is shown in Table 14.

Table 14. FLOPs of YOLO models.

Model	FLOPs				
	n	s	m	l	x
YOLOv8	8.7	28.6	78.9	165.2	257.8
YOLOv9	7.7 (t)	26.7	76.8	102.8 (c)	192.5 (e)
YOLOv10	6.7	21.6	59.1	120.3	160.4
YOLOv11	6.5	21.5	68	86.9	194.9
YOLOv12	6.5	21.4	67.5	88.9	199

4.5. Load Time

Model load time is a crucial metric for evaluating the responsiveness of deep learning systems, particularly in edge deployment scenarios where startup latency can affect real-time operation. In this study, load time is defined as the duration required to load model weights from storage, initialize network layers, and allocate memory resources.

Load time values vary across devices due to differences in storage interfaces, CPU performance, and memory bandwidth.

Table 15 presents the measured load times across YOLO model scales under different execution scenarios. For each model size, both average values on SBC platforms and maximum observed values are reported to reflect variability across hardware configurations.

Table 15. Measured load time across YOLO model scales on different platforms.

Scale	Scenario	Load Time (ms)	Feasibility
YOLO-n	Avg (SBC)	500–900	Fully feasible
	Max observed	~1000	
YOLO-s	Avg (SBC)	900–1500	Feasible
	Max observed	~1700	
YOLO-m	Avg (SBC)	1500–3000	Moderate
	Max observed	~3500	
YOLO-l	Avg (SBC)	3000–5000	Limited
	Max observed	~6000	
YOLO-x	Avg (SBC)	>5000	Constrained
	Max observed	>7000	

As shown in Table 15, load time increases consistently with model size due to higher parameter counts and increased initialization overhead. Lightweight models (YOLO-n and YOLO-s) demonstrate fast startup times, making them suitable for latency-sensitive applications such as real-time embedded systems and autonomous platforms.

In contrast, larger models (YOLO-m, YOLO-l, and YOLO-x) introduce significant initialization delays, particularly on resource-constrained SBCs. In such cases, prolonged load times can reduce system responsiveness and limit practical deployment, especially in scenarios requiring rapid startup or dynamic model switching.

The observed variations in load time across different platforms are primarily influenced by hardware characteristics. For example, devices such as Raspberry Pi 4 and Jetson Nano exhibit slower load times due to limited RAM (4 GB), slower storage interfaces (microSD-based I/O), and older CPU architectures that are less optimized for model deserialization and memory allocation.

Additionally, model initialization remains largely CPU-bound, even when GPU acceleration is used for inference. As a result, the presence of a GPU does not significantly reduce load time, particularly on entry-level platforms such as Jetson Nano. Other contributing factors include unified memory architectures, which increase contention between CPU and GPU.

On the other hand, platforms such as Raspberry Pi 5, LattePanda, and Jetson Orin benefit from improved hardware capabilities, including faster CPUs, higher memory capacity, and faster storage (eMMC or NVMe), which collectively result in reduced load times.

Overall, these findings highlight that load time is not solely dependent on model size but is also strongly influenced by system architecture. Therefore, for applications requiring fast initialization and responsiveness, lightweight models are preferred, while larger models may require more capable hardware to achieve acceptable startup performance.

4.6. Power Consumption

Power consumption in SBCs varies significantly based on processor clock speeds, operational modes, and power management policies. For ARM-based boards like the Raspberry Pi (RPI), power draw generally increases [51,52]. A simplified model for dynamic power P_{dyn} can be expressed as

$$P_{\text{dyn}} \approx C \cdot V^2 \cdot f, \quad (3)$$

where C is the switched capacitance, V is the supply voltage, and f is the clock frequency. At lower frequencies (e.g., 600 MHz), the RPi 4 consumes ~ 2.5 W, while at 1.8 GHz, this rises to ~ 7 W due to voltage scaling and increased leakage current. Similarly, the RPi 5's higher-performance cores draw up to 12 W under full load [53].

For x86 systems like LattePanda, Windows power policies influence consumption. The "Efficiency mode" reduces CPU voltage/frequency, lowering power to ~ 6 W, while "High performance" disables throttling, permitting ~ 30 W [54]. NVIDIA's Jetson and Orin Nano boards enforce strict Thermal Design Power (TDP) limits (e.g., 7 W and 15 W), with actual usage closely matching these values [48,55]. Table 16 shows the Power Consumption (W) by Board and Mode and Estimated Runtime on a 4500 mAh (16.65 Wh) Li-ion Battery. It is important to emphasize that the power values reported in Table 16 are estimated rather than directly measured. These values are derived from manufacturer specifications and typical operating ranges and are intended to provide an illustrative comparison of energy consumption across platforms, rather than a precise experimental benchmark.

For the Jetson Orin Nano, the labeled power modes (e.g., 7 W and 25 W) correspond to predefined power profiles rather than exact system-level power consumption. The estimated values presented in Table 16 account for the total device power draw under typical workloads, including additional components such as memory, storage, and peripheral subsystems. As a result, the effective system-level consumption may differ from the nominal mode labels.

Table 16. Illustrative estimation of power consumption (W) and corresponding runtime across different SBC platforms and power modes.

Board	Mode	Power (W)	Runtime (h)
RPI 4	Low Power (600 MHz)	2.5	6.7
	Normal (1.5 GHz)	5.0	3.3
	High Power (1.8 GHz)	7.0	2.4
RPI 5	Low Power (600 MHz)	3.0	5.6
	Normal (2.4 GHz)	8.0	2.1
	High Perf. (2.9 GHz)	12.0	1.4
LattePanda	Power Saver	6.0	2.8
	Balanced	15.0	1.1
	High Perf.	30.0	0.6
Jetson Nano	5 W Mode	5.0	3.3
	10 W Mode	10.0	1.7
Orin Nano	7 W Mode	7.0	2.4
	15 W Mode	15.0	1.1
	25 W Mode	25.0	0.7

4.7. GPU Processing vs. CPU-Only Processing

To quantitatively evaluate the impact of GPU acceleration on both system utilization and performance, CPU usage and inference speed (FPS) were compared under CPU-only and GPU-enabled configurations on NVIDIA Jetson platforms.

During inference, CPU utilization and memory usage were continuously monitored using the psutil Python library, while GPU utilization and memory allocation were recorded using gpustat. Measurements were collected throughout the execution of the COCO128 dataset with a batch size of 1, and average values were computed over the full inference duration to ensure stable and representative results.

For performance comparison, representative FPS values from nano (n) model variants were used, as they consistently provide the highest throughput and are the most relevant for real-time edge deployment.

Table 17 summarizes the observed CPU utilization and corresponding FPS improvement when GPU acceleration is enabled.

Table 17. CPU utilization and FPS improvement with GPU acceleration.

Platform	CPU-Only (%)	GPU (%)	CPU FPS	GPU FPS (Speedup)
Jetson Nano	40.5–54.8	5.2–9.1	~0.9	~3.0 (~3.3×)
Jetson Orin Nano	20.5–34.7	5.3–9.3	~2.3	~13.1 (~5.7×)

As shown in Table 17, GPU acceleration substantially reduces CPU utilization while simultaneously improving inference throughput. On the Jetson Nano, CPU usage decreases from approximately 40–55% in CPU-only execution to below 10% when GPU acceleration is enabled, while FPS improves by more than 3×. A similar trend is observed on the Jetson Orin Nano, where GPU acceleration reduces CPU usage to less than 10% and increases inference speed by over 5×.

This improvement is achieved by offloading computationally intensive operations—such as convolution and feature extraction—from the CPU to the GPU, thereby reducing CPU load and allowing it to handle auxiliary tasks such as data preprocessing, scheduling, and system control.

From a system-level perspective, this reduction in CPU utilization is critical for edge computing applications, where multiple components must operate concurrently. By freeing CPU resources, GPU acceleration enables more efficient execution of additional processes such as sensor fusion, communication, and control logic in applications including robotics, autonomous systems, and IoT deployments.

Overall, the results demonstrate that GPU acceleration provides a dual benefit: significant performance gains in inference speed and improved system efficiency, making it essential for scalable and reliable edge AI deployment.

4.8. Cost Analysis and Cost–Performance Trade-Offs

In addition to performance metrics, the cost of hardware platforms is a critical factor influencing deployment decisions in edge AI systems. To provide a fair and practical comparison, the approximate prices of the evaluated single-board computers (SBCs) were collected from Amazon US listings, reflecting real-world market conditions.

As shown in Table 18, the evaluated platforms span a wide cost range, from approximately \$100–\$250 for Raspberry Pi systems to \$249–\$380 for higher-end platforms such as Jetson Orin Nano and LattePanda. While Raspberry Pi boards represent lower-cost and energy-efficient solutions relative to dedicated AI platforms, their computational performance remains limited.

Table 18. Approximate SBC prices based on Amazon US listings (USD).

Platform	Amazon Price (USD)
Raspberry Pi 4 (4 GB)	\$100–\$150
Raspberry Pi 5 (8 GB)	\$190–\$250
Jetson Nano (4 GB)	\$200
Jetson Orin Nano (8 GB)	\$249
LattePanda 3 Delta (8 GB)	\$310–\$380

A clear relationship between cost and achievable performance is observed. Raspberry Pi platforms typically operate below 3 FPS, making them suitable for non-real-time applications. In contrast, GPU-enabled platforms such as the Jetson Orin Nano achieve real-time performance exceeding 12 FPS for lightweight YOLO models. Notably, although the cost difference is not always substantial, the performance gap is significant, highlighting the importance of hardware acceleration.

The LattePanda platform, despite its relatively high cost, does not provide competitive performance for deep learning-based object detection, reinforcing that higher price does not necessarily guarantee better suitability for edge AI workloads.

These findings emphasize that selecting an appropriate SBC requires balancing performance requirements, system constraints, and economic considerations to achieve an effective deployment strategy.

4.9. Results Summary

Figure 2 shows a comparison of SBCs across five key YOLO deployment metrics: FPS, mAP, RAM efficiency, power efficiency, and load speed. Higher values indicate better performance or efficiency. The chart highlights the trade-offs between computational capability and resource constraints across platforms.

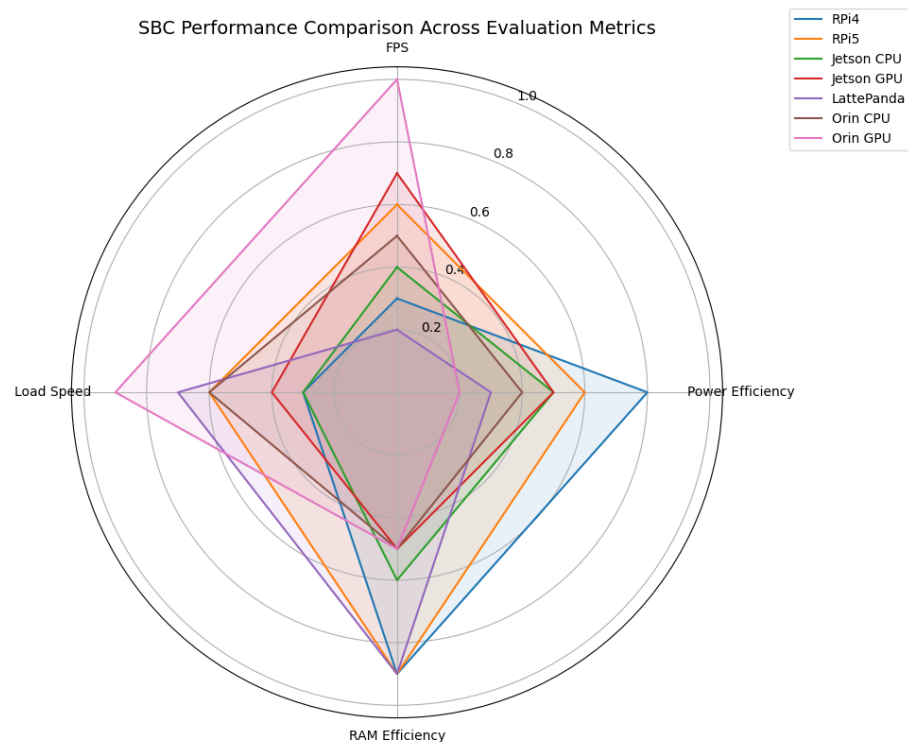


Figure 2. Radar chart comparing SBCs and a PC across key YOLO deployment metrics.

5. Conclusions and Future Work

5.1. Conclusions

This study presented a comprehensive benchmarking of YOLOv8 through YOLOv12 across multiple single-board computing (SBC) platforms under diverse power configurations, evaluating key deployment metrics including inference speed (FPS), CPU utilization, RAM usage, computational complexity, and load time.

The experimental results reveal several critical insights.

First, model scale strongly determines deployment feasibility. Lightweight models (YOLO-n and YOLO-s) consistently achieved the highest throughput across all platforms, with FPS performance exceeding 12–13 FPS on the Jetson Orin Nano GPU. In contrast, medium and large models (YOLO-m, YOLO-l, and YOLO-x) experienced significant performance degradation, often falling below 3 FPS on most SBCs and exceeding 40–50% RAM utilization, making them unsuitable for real-time or resource-constrained environments.

Second, hardware acceleration is the dominant factor in achieving real-time performance. CPU-only execution across all platforms—including Raspberry Pi 4/5 and Jetson CPUs—remains insufficient for practical deployment, with most configurations operating below 3 FPS. In contrast, GPU-enabled platforms provide substantial performance gains, with speedups of approximately $3\times$ on Jetson Nano and up to $5.7\times$ on Jetson Orin Nano. This confirms that GPU acceleration is essential for modern YOLO deployment in edge computing scenarios.

Third, the results highlight an important system-level trade-off between performance and resource utilization. GPU acceleration reduces CPU usage significantly (from approximately 40–55% to below 10% on Jetson Nano), enabling concurrent execution of additional tasks. However, this improvement comes with increased RAM usage (rising to approximately 40–45%), primarily due to unified memory architectures and CUDA-related overhead. This trade-off must be carefully considered when designing edge AI systems.

Across all tested configurations, YOLOv10n emerged as the most efficient model, achieving the best balance between speed, accuracy, and resource consumption. It consis-

tently delivered the highest FPS across Raspberry Pi, Jetson Nano, and Jetson Orin platforms. Additionally, YOLOv8n and YOLOv11n demonstrated competitive performance, particularly on GPU-enabled devices. On more capable platforms such as Jetson Orin Nano, larger models such as YOLOv12-m and YOLOv12-l become viable options, achieving near real-time performance at the cost of increased memory and computational requirements.

From a deployment perspective, the findings provide clear guidelines: (i) CPU-only SBCs should be limited to lightweight, non-real-time applications; (ii) GPU-enabled SBCs are required for real-time object detection; (iii) nano and small variants represent the optimal choice for most edge scenarios; and (iv) high-performance platforms such as Jetson Orin Nano enable scalable deployment of more complex models.

An additional key outcome of this study is the identification of a clear cost-performance trade-off across SBC platforms. While low-cost devices such as Raspberry Pi boards offer more cost-efficient deployment options compared to GPU-enabled platforms, their performance remains limited to non-real-time workloads. In contrast, higher-cost platforms such as the Jetson Orin Nano enable real-time inference exceeding 12 FPS for lightweight YOLO models, demonstrating substantial performance gains. However, the results also show that increased cost does not always guarantee better suitability for edge AI tasks, as observed with the LattePanda platform. These findings emphasize that effective system design requires balancing computational requirements, resource constraints, and budget considerations to achieve optimal deployment.

Overall, this work provides a unified and reproducible evaluation framework that bridges model-level benchmarking with system-level deployment considerations, offering practical insights for selecting optimal YOLO configurations in edge AI applications.

5.2. Future Work

Future research can extend this work in several directions. First, additional SBC platforms such as Khadas VIM4, Google Coral Dev Board, and BeagleBone AI-64 can be evaluated to broaden the hardware comparison and capture emerging accelerators.

Second, optimized inference frameworks such as TensorRT, ONNX Runtime, and OpenVINO should be investigated to assess their impact on latency, memory efficiency, and power consumption, particularly for GPU-enabled platforms.

Third, model optimization techniques including quantization, pruning, and knowledge distillation can be applied to reduce computational complexity and memory footprint, enabling more efficient deployment of larger YOLO variants on constrained devices.

Fourth, real-world deployment scenarios should be explored, such as drone navigation, autonomous robotics, and smart surveillance, where dynamic inputs, multi-sensor integration, and real-time constraints introduce additional system-level challenges.

Finally, future studies may include newer architectures such as YOLO-NAS and upcoming YOLO releases, as well as comparative evaluations with transformer-based object detection models, to further advance the understanding of efficient edge AI deployment.

Author Contributions: O.S.: Conceptualization, Methodology, Investigation, Experimental design, Data curation, Formal analysis, Visualization, Writing—original draft, Writing—review and editing, Supervision. E.K.: Methodology, Software, Experimental setup, Data curation, Validation, Writing—review and editing. A.E.-A.: Software, Benchmark implementation, Experimental evaluation, Data analysis, Writing—review and editing. L.E.: Software optimization, SBC configuration, Performance evaluation, Investigation, Writing—review and editing. Y.A.: Literature review, Data interpretation, Visualization, Writing—review and editing. J.Z.: Hardware setup, Power mode configuration, Performance testing, Validation, Writing—review and editing. M.E.-S.: Formal analysis, Result interpretation, Comparative analysis, Writing—review and editing. H.S.: Methodology refine-

ment, Critical review, Technical validation, Writing—review and editing. 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 COCO dataset used in this study is publicly available at <https://cocodataset.org>. Additional benchmarking scripts and configuration files will be provided upon reasonable request. The code used for benchmarking the boards can be downloaded <https://github.com/ahmedelagamy132/Yolo-Project> (accessed on 7 May 2026), while the code that enables GPU acceleration for the same benchmarking process is available https://github.com/LoayElmokadem/YOLO_Research_Scripts_GPU (accessed on 7 May 2026).

Acknowledgments: The authors acknowledge Ajman University for its support in this research.

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

Abbreviations

The following abbreviations are used in this manuscript:

AI	Artificial Intelligence
AV	Autonomous Vehicle
CNN	Convolutional Neural Network
COCO	Common Objects in Context
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DL	Deep Learning
FLOPs	Floating Point Operations
FPS	Frames Per Second
GPU	Graphics Processing Unit
IoT	Internet of Things
IoU	Intersection over Union
mAP	Mean Average Precision
mAP@50–95	Mean Average Precision at IoU thresholds 0.50 to 0.95
NMS	Non-Maximum Suppression
ONNX	Open Neural Network Exchange
RAM	Random Access Memory
SBC	Single-Board Computer
SSD	Single-Shot Detector
STEM	Science, Technology, Engineering, and Mathematics
TDP	Thermal Design Power
YOLO	You Only Look Once

References

1. Khatab, E.; Fathy, F.; AlKholy, A.; Shalash, O. Sensor Fusion and Perception for Autonomous Driving: A Critical Review of Modalities, AI Models, Algorithms, and Industry Configurations. *Mach. Learn. Knowl. Extr.* **2026**, *8*, 199. [CrossRef]
2. Harzallah, H.; Jurie, F.; Schmid, C. Combining efficient object localization and image classification. In *Proceedings of the 2009 IEEE 12th International Conference on Computer Vision*; IEEE: New York, NY, USA, 2009; pp. 237–244.
3. Viola, P.; Jones, M.J. Robust real-time face detection. *Int. J. Comput. Vis.* **2004**, *57*, 137–154. [CrossRef]
4. Dalal, N.; Triggs, B. Histograms of oriented gradients for human detection. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*; IEEE: New York, NY, USA, 2005; Volume 1, pp. 886–893.
5. Felzenszwalb, P.F.; Girshick, R.B.; McAllester, D.; Ramanan, D. Object detection with discriminatively trained part-based models. *IEEE Trans. Pattern Anal. Mach. Intell.* **2009**, *32*, 1627–1645. [CrossRef]

6. Fawzy, H.; Elbraway, A.; Amr, M.; Eltanekhy, O.; Khatab, E.; Shalash, O. A systematic review: Computer vision algorithms in drone surveillance. *J. Robot. Integr.* **2025**, *2*, 1–10.
7. Azzaz, Z.; Khaled, O.; Khatab, E.; Said, H.; Shalash, O. EASE-PVNet: Robust Periocular Identity Verification Across Pre- and Post-Operative Facial Images. *Mach. Learn. Knowl. Extr.* **2026**, *8*, 169. [[CrossRef](#)]
8. Soviany, P.; Ionescu, R.T. Optimizing the trade-off between single-stage and two-stage deep object detectors using image difficulty prediction. In *Proceedings of the 2018 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNAS)*; IEEE: New York, NY, USA, 2018; pp. 209–214.
9. Redmon, J.; Divvala, S.; Girshick, R.; Farhadi, A. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*; IEEE: New York, NY, USA, 2016; pp. 779–788. [[CrossRef](#)]
10. Liu, W.; Anguelov, D.; Erhan, D.; Szegedy, C.; Reed, S.; Fu, C.Y.; Berg, A.C. Ssd: Single shot multibox detector. In *Proceedings of the Computer Vision—ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, 11–14 October 2016*; *Proceedings, Part I* 14; Springer: Berlin/Heidelberg, Germany, 2016; pp. 21–37.
11. Girshick, R.; Donahue, J.; Darrell, T.; Malik, J. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*; IEEE: New York, NY, USA, 2014; pp. 580–587.
12. Ghahremani, A.; Adams, S.D.; Norton, M.; Khoo, S.Y.; Kouzani, A.Z. Detecting defects in solar panels using the YOLO v10 and v11 algorithms. *Electronics* **2025**, *14*, 344. [[CrossRef](#)]
13. Li, M.; Yan, N. IPD-YOLO: Person detection in infrared images from UAV perspective based on improved YOLO11. *Digit. Signal Process.* **2025**, *168*, 105469. [[CrossRef](#)]
14. Liao, Y.; Li, L.; Xiao, H.; Xu, F.; Shan, B.; Yin, H. YOLO-MECD: Citrus detection algorithm based on YOLOv11. *Agronomy* **2025**, *15*, 687. [[CrossRef](#)]
15. Terven, J.; Córdova-Esparza, D.M.; Romero-González, J.A. A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS. *Mach. Learn. Knowl. Extr.* **2023**, *5*, 1680–1716. [[CrossRef](#)]
16. Redmon, J.; Farhadi, A. YOLO9000: Better, Faster, Stronger. In *Proceedings of the 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*; IEEE: New York, NY, USA, 2017; pp. 6517–6525. [[CrossRef](#)]
17. Redmon, J.; Farhadi, A. YOLOv3: An Incremental Improvement. *arXiv* **2018**, arXiv:1804.02767.
18. Bochkovskiy, A.; Wang, C.Y.; Liao, H. YOLOv4: Optimal Speed and Accuracy of Object Detection. *arXiv* **2020**, arXiv:2004.10934.
19. Khanam, R.; Hussain, M. What is YOLOv5: A deep look into the internal features of the popular object detector. *arXiv* **2024**, arXiv:2407.20892.
20. Li, C.; Li, L.; Jiang, H.; Weng, K.; Geng, Y.; Li, L.; Ke, Z.; Li, Q.; Cheng, M.; Nie, W.; et al. YOLOv6: A Single-Stage Object Detection Framework for Industrial Applications. *arXiv* **2022**, arXiv:2209.02976.
21. Yaseen, M. What is YOLOv8: An In-Depth Exploration of the Internal Features of the Next-Generation Object Detector. *arXiv* **2024**, arXiv:2409.07813.
22. Wang, C.Y.; Yeh, I.H.; Liao, H. YOLOv9: Learning What You Want to Learn Using Programmable Gradient Information. *arXiv* **2024**, arXiv:2402.13616. [[CrossRef](#)]
23. Wang, A.; Chen, H.; Liu, L.; Chen, K.; Lin, Z.; Han, J.; Ding, G. YOLOv10: Real-Time End-to-End Object Detection. *Adv. Neural Inf. Process. Syst.* **2024**, *37*, 107984–108011. [[CrossRef](#)]
24. Khanam, R.; Hussain, M. YOLOv11: An Overview of the Key Architectural Enhancements. *arXiv* **2024**, arXiv:2410.17725.
25. Tian, Y.; Ye, Q.; Doermann, D. YOLOv12: Attention-Centric Real-Time Object Detectors. *arXiv* **2025**, arXiv:2502.12524.
26. Jiang, P.; Ergu, D.; Liu, F.; Cai, Y.; Ma, B. A Review of Yolo algorithm developments. *Procedia Comput. Sci.* **2022**, *199*, 1066–1073. [[CrossRef](#)]
27. Rashid, T.; Fadzil, L.M. Comparative review of object detection algorithms in small single-board computers. *Int. J. Recent Innov. Trends Comput. Commun.* **2023**, *11*, 244–252. [[CrossRef](#)]
28. Hussain, M. Yolov1 to v8: Unveiling each variant—A comprehensive review of yolo. *IEEE Access* **2024**, *12*, 42816–42833. [[CrossRef](#)]
29. Vijayakumar, A.; Vairavasundaram, S. Yolo-based object detection models: A review and its applications. *Multimed. Tools Appl.* **2024**, *83*, 83535–83574. [[CrossRef](#)]
30. Ali, M.L.; Zhang, Z. The YOLO framework: A comprehensive review of evolution, applications, and benchmarks in object detection. *Computers* **2024**, *13*, 336. [[CrossRef](#)]
31. Raspberry Pi Foundation. Raspberry Pi 4 Model B Specifications. 2024. Available online: <https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/> (accessed on 13 February 2025).
32. Karumbunathan, L.S. *NVIDIA Jetson AGX Orin Series*; NVIDIA: Santa Clara, CA, USA, 2022.
33. LattePanda Team. LattePanda 3 Delta Specifications. 2024. Available online: https://docs.lattepanda.com/content/3rd_delta_edition/specification/ (accessed on 13 February 2025).
34. Raspberry Pi Foundation. Raspberry Pi 5 Specifications. 2024. Available online: <https://pip-assets.raspberrypi.com/categories/892-raspberry-pi-5/documents/RP-008348-DS-6-raspberry-pi-5-product-brief.pdf> (accessed on 13 February 2025).

35. Süzen, A.A.; Duman, B.; Şen, B. Benchmark analysis of jetson tx2, jetson nano and raspberry pi using deep-cnn. In *Proceedings of the 2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*; IEEE: New York, NY, USA, 2020; pp. 1–5.
36. Wang, J.; Liu, M.; Du, Y.; Zhao, M.; Jia, H.; Guo, Z.; Su, Y.; Lu, D.; Liu, Y. PG-YOLO: An efficient detection algorithm for pomegranate before fruit thinning. *Eng. Appl. Artif. Intell.* **2024**, *134*, 108700. [\[CrossRef\]](#)
37. Wu, D.; Ying, Y.; Zhou, M.; Pan, J.; Cui, D. YOLO-Claw: A fast and accurate method for chicken claw detection. *Eng. Appl. Artif. Intell.* **2024**, *136*, 108919. [\[CrossRef\]](#)
38. Chen, X.; Wang, M.; Ling, J.; Wu, H.; Wu, B.; Li, C. Ship imaging trajectory extraction via an aggregated YOLO model. *Eng. Appl. Artif. Intell.* **2024**, *130*, 107742. [\[CrossRef\]](#)
39. Wang, Y.; Ma, C.; Zhao, C.; Xia, H.; Chen, C.; Zhang, Y. WB-YOLO: An efficient wild bat detection method for ecological monitoring in complex environments. *Eng. Appl. Artif. Intell.* **2025**, *157*, 111232. [\[CrossRef\]](#)
40. Yang, R.; Zhang, H. Tunnel-YOLO: An improved You Only Look Once algorithm for real-time shield tunnel lining leakage detection. *Eng. Appl. Artif. Intell.* **2025**, *162*, 112403. [\[CrossRef\]](#)
41. Lin, T.Y.; Maire, M.; Belongie, S.; Hays, J.; Perona, P.; Ramanan, D.; Dollár, P.; Zitnick, C.L. Microsoft COCO: Common Objects in Context. In *Lecture Notes in Computer Science*; Springer: Berlin/Heidelberg, Germany, 2014; pp. 740–755. [\[CrossRef\]](#)
42. Baldovino, R.G.; Vidad, A.J.P.; Abastillas, R.P.B.; Bugtai, N.T.; Dadios, E.P.; Vicerra, R.R.P.; Bandala, A.A.; See, A.R.; Roxas, N.R., Jr. Comprehensive analysis on ultralytics-supported YOLO models for detection and recognition of large office objects for indoor navigation. *Procedia Comput. Sci.* **2024**, *246*, 3851–3858. [\[CrossRef\]](#)
43. Ge, Z.; Liu, S.; Wang, F.; Li, Z.; Sun, J. Yolox: Exceeding yolo series in 2021. *arXiv* **2021**, arXiv:2107.08430.
44. Nguyen, H.V.; Bae, J.H.; Lee, Y.E.; Lee, H.S.; Kwon, K.R. Comparison of pre-trained YOLO models on steel surface defects detector based on transfer learning with GPU-based embedded devices. *Sensors* **2022**, *22*, 9926. [\[CrossRef\]](#) [\[PubMed\]](#)
45. Benoit-Cattin, T.; Velasco-Montero, D.; Fernández-Berni, J. Impact of thermal throttling on long-term visual inference in a CPU-based edge device. *Electronics* **2020**, *9*, 2106. [\[CrossRef\]](#)
46. NVIDIA Corporation. Power Optimization with NVIDIA Jetson. 2023. Available online: <https://developer.nvidia.com/blog/power-optimization-with-nvidia-jetson/> (accessed on 7 May 2026).
47. Raspberry Pi Foundation. Frequency Management and Thermal Control. 2023. Available online: <https://www.raspberrypi.com/documentation/hardware/raspberrypi/> (accessed on 7 May 2026).
48. NVIDIA. *Jetson Nano Developer Kit Datasheet*; NVIDIA: Santa Clara, CA, USA, 2021.
49. Métwalli, A.; Fathy, F.; Khatab, E.; Shalash, O. ER-ACO: A Real-Time Ant Colony Optimization Framework for Emergency Medical Services Routing and Hospital Resource Scheduling. *Algorithms* **2026**, *19*, 102. [\[CrossRef\]](#)
50. Salah, Y.; Shalash, O.; Khatab, E. A lightweight speaker verification approach for autonomous vehicles. *Robot. Integr. Manuf. Control* **2024**, *1*, 15–30. [\[CrossRef\]](#)
51. Shalash, O.; Emad, A.; Fathy, F.; Alzogby, A.; Sallam, M.; Naser, E.; El-Sayed, M.; Khatab, E. Fusion of Robotics, AI, and Thermal Imaging Technologies for Intelligent Precision Agriculture Systems. *Sensors* **2025**, *25*, 6844. [\[CrossRef\]](#) [\[PubMed\]](#)
52. Geerling, J. Raspberry Pi 4 Power Consumption and Temperature Tests. 2020. Available online: <https://www.jeffgeerling.com/> (accessed on 13 February 2025).
53. Raspberry Pi Foundation. Raspberry Pi 5 Documentation. 2023. Available online: <https://www.raspberrypi.com/documentation/computers/raspberry-pi-5.html> (accessed on 1 November 2025).
54. Microsoft. *Windows Power Policy Settings*; Microsoft: Redmond, WA, USA, 2022.
55. NVIDIA. *Jetson Orin Nano Technical Brief*; NVIDIA: Santa Clara, CA, USA, 2023.

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.