Journal of Telecommunication, Electronic and Computer Engineering
ISSN: 2180 – 1843 e-ISSN: 2289-8131 Vol. 17 No. 1 (2025) 19-28
jtec.utem.edu.my
DOI: https://doi.org/10.54554/jtec.2025.17.01.003

Energy Optimized YOLO: Quantized Inference for Real-Time Edge AI Object Detection

Hwee Min Chiam[1], Yan Chiew Wong[1]*, Ranjit Singh Sarban Singh[2], T. Joseph Sahaya Anand[3]
[1]*Faculty of Electronics and Computer Technology and Engineering, Universiti Teknikal Malaysia Melaka (UTeM), 76100 Durian Tunggal, Melaka, Malaysia.*
[2]*School of Engineering and Technology, Sunway University, Selangor, Malaysia.*
[3]*School of Computing, MIT Vishwaprayag University, Solapur, 413255, India.*

Article Info

Article history:
Received Jul 1st, 2024
Revised Jan 27th, 2025
Accepted Mar 11th, 2025
Published Mar 19h, 2025

Index Terms:
Object Detection
Real-time
Edge Device
Quantization
FPGA
Jetson Nano
YOLO

Abstract

Efficient real-time object detection is a critical requirement in edge computing applications, such as smart surveillance, where resource constraints pose significant challenges. Existing deep learning methods often struggle to balance accuracy and efficiency, particularly when deployed on hardware with limited computational resources. This work focuses on developing a quantized object detection system utilizing advanced deep learning models to improve inference performance on edge devices, Zedboard and Jetson Nano. The Zedboard, an FPGA platform without GPU acceleration, executes a quantized YOLOv3-tiny model with ultra-low power consumption of 2.2W but requires over 3 seconds per inference, making it unsuitable for real-time applications. In contrast, the Jetson Nano, running an optimized YOLOv7-tiny model with FP16 quantization and GPU acceleration, achieves a processing speed of 38 FPS with mAP of 46.3%, while maintaining a low power consumption of 5.1W. Based on the results, this work presents a practical solution for real-time object detection in resource-constrained environments by demonstrating the benefits of combining quantized deep learning models with GPU acceleration. Future work could focus on fine-tuning models for specific applications, such as traffic monitoring, to improve the detection of vehicles, pedestrians, and traffic signs in dynamic environments.

*Corresponding Author: ycwong@utem.edu.my

I. INTRODUCTION

Nowadays object detection systems are essential for various applications, such as autonomous vehicles for environment perception, human-computer interfaces, and video surveillance systems [1] [2]. These applications require real-time detection performance, but accurately detecting objects in real-time can be challenging. Besides, the process of identifying objects in images or videos is difficult due to various issues such as partial occlusion, background clutter, variations in scale, viewpoint, lighting, and appearance. Object characteristics can also vary significantly from instance to instance, complicating the detection process.

There are numerous deep learning methods available today, but none of them are efficient when handling videos or images with many features, as recognizing objects with low latency becomes increasingly difficult. Besides, implementing deep learning systems that run on hardware is challenging since system performance, resources, and the power efficiency of the hardware employed are interdependent and impact each other [3]. This difficulty has prompted extensive study into the development of hardware accelerators to improve system performance while consuming less power.

The most important metrics for embedded machine learning include accuracy, energy, throughput, latency, and cost [4]. These must be applied to an appropriate dataset to ensure accuracy. When it comes to Deep Neural Networks (DNNs) the processor must be able to support networks of different number of channels, number of layers, number of filters and filter sizes. This requirement increases data movement and hence computation, which is the main constraint for energy efficiency, as data movement is more expensive than computation. Choosing the appropriate weight precision is one of the vital steps, as it involves balancing between the DNNs' performance and implementation costs. Higher weight precision decreases the quantization inaccuracies, whereas lower precision increases processing speed, decreases circuit complexity, decreases circuit size and lowers power consumption. One approach to resolving this trade-off is to identify the 'minimal level of precision' required to solve a given problem [5].

Hence, the main purpose of this work is to develop a quantized model and run inference through hardware accelerators to enhance the processing capabilities of edge devices, particularly in handling the complex computations required for real-time object detection. By integrating these hardware accelerators, it is possible to offload computationally intensive tasks from the edge device's main processor, thereby improving overall system efficiency and enabling real-time performance.

This work contributes to the field by demonstrating how quantization and hardware acceleration can be effectively combined to optimize object detection on edge devices, offering a practical approach for deploying deep learning models in resource-constrained environments. Moreover, the framework developed can be further fine-tuned for specific use cases, such as traffic monitoring or smart surveillance, to enhance detection accuracy and performance for these applications. By targeting real-world scenarios, such as dynamic environments with varying traffic patterns or complex surveillance footage, future iterations of this work can adapt the model to achieve even higher levels of precision and efficiency. This versatility demonstrates the potential for the developed system to be deployed across a wide range of edge applications, expanding the utility and impact of AI-driven object detection systems in real-time, resource-constrained settings.

II. LITERATURE REVIEW

A. Object Detection

The field of object detection in deep learning includes two main algorithmic approaches: Two-Stage and One-Stage methods [6]. Two-Stage methods, such as R-CNN, Faster-RCNN, and the SSD series, use external proposal generators to produce anchor boxes, whereas One-Stage methods like the YOLO-series generate bounding boxes directly from the image, enhancing speed and making it suitable for real-time applications. Modern research work on One-Stage detectors shows that a neural network generates a set of bounding boxes superimposed on the feature maps at different scales, ratios, and positions [7]. One-stage detectors are more efficient in terms of computational time, but their precision decreases when compared with Two-stage detectors [8].

B. Model Quantization

Over the past few years, quantization has emerged as an effective method for in-depth training of neural networks with less complexity [9]. Quantization reduces mode size in terms of memory requirements and enables faster computations in most processors [10].

Kalenichenko et al. introduced an 8-bit integer post-training quantization method, which converts a trained model to use 8-bit integers for weights and activations, thereby enhancing efficiency and reducing model size with minimal loss of accuracy [11]. They also proposed quantization-aware training, a technique designed for smaller models, where quantization effects are simulated during the training process, allowing the model to learn and adapt to lower precision. This approach typically results in better accuracy and performance for the quantized model compared to post-training quantization alone. In research conducted by Li et al., it was demonstrated that a 4-bit model can closely match the performance of a 32-bit floating-point version, even in mobile-friendly networks [12]. They introduced a fully quantized network (FQN) for object detection. FQN quantizes network weights and activations using low-bit fixed-point arithmetic and employs fine-tuning techniques to optimize them.

C. Object Detection at Edge

Deploying object detection models like YOLOv4, which utilizes the CSPDarknet53 backbone based on CNN algorithm, poses substantial issues for edge devices due to large model sizes and the demand for memory and computational resources [9]. To address this, hardware accelerators, such as GPU and FPGA, have been suggested to expedite the CNN processing [13]. While most popular Python-based machine learning frameworks use 32-bit floating-point numbers to compute model outputs during inference, switching from higher bit floating-point to smaller bit size is a viable solution to reduce model’s size while maintaining reasonable accuracy, enabling deployment on resource-constrained devices [14].

Li et al. proposed a CNN-based FPGA accelerator using Intel Arria 10 for real-time object detection, achieving 25 FPS with 73.6 mAP, demonstrating superior power efficiency compared to traditional GPUs and CPUs [15]. Their approach demonstrated the feasibility of using mixed fixed-point data representation to maintain accuracy while enhancing throughput with parallel architectures.

Aguiar et al. addressed grape bunch detection using quantized deep learning models deployed on low-cost Tensor Processing Units, achieving a mAP of 66.96% on vineyard images, demonstrating the feasibility of real-time agricultural applications [16]. Choe et al. conducted a benchmark analysis of NVIDIA Jetson platforms for 3D object detection, emphasizing TensorRT's role in boosting inference speed and reducing resource utilization [17]. Jetson NX and AGX platforms were shown to handle complex tasks efficiently, making them suitable for robotic applications requiring real-time processing. Abdulghafoor et al. developed a real-time object detection and tracking algorithm using Nvidia Jetson TX2 and DeepStream SDK, integrating models like SSD and YOLO [18]. Their results showed high accuracy and speed, with SSD excelling in precision and YOLO offering faster processing.

Divakar and Pavani proposed a CNN-based object detection system accelerated by an FPGA, achieving 82 FPS with AlexNet architecture [19]. Their study highlighted the efficiency of FPGA hardware accelerators for real-time object detection, emphasizing their advantages in speed, energy efficiency, and parallel processing capabilities.

In this work, the YOLO series will be implemented as the object detection model on FPGA and GPU platforms. By performing model quantization techniques, including the post-training quantization, and designing a hardware accelerator for running object detection tasks, this work aims to optimize the single-frame inference time to enhance the real-time performance.

III. RESEARCH METHODOLOGY

A. Deployment on Zedboard

In this work, the Zedboard development kit with the Zynq-7000 All Programmable SoC is chosen for its FPGA capabilities [20]. It meets key requirements such as low cost, flexibility, and low power consumption, making Zynq-based kits an ideal choice for developing customs, programmable systems that fulfill object detection tasks and inference needs on edge devices. To start the process, it is necessary to prepare the object detection model. In this case, since YOLO series pretrained model is used, no additional training is required. The overall deployment method that deployment on Zedboard and running inference is illustrated in Figure 1.

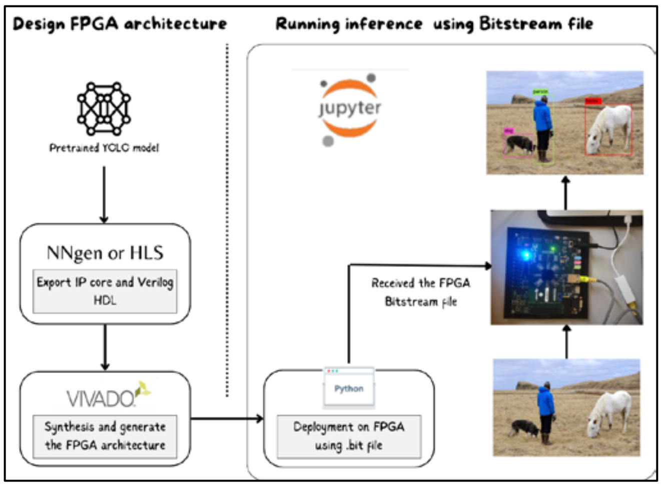

Design FPGA architecture
Running inference using Bitstream file
Pretrained YOLO model
NNgen or HLS
Export IP core and Verilog HDL
VIVADO
Synthesis and generate the FPGA architecture
jupyter
Received the FPGA Bitstream file
Python
Deployment on FPGA using .bit file

Figure 1. Overall flow of deployment of object detection model and running inference on Zedboard

1) *Generation of IP core*

The development of an IP core is a critical step in implementing an object detection model on FPGA hardware. This process includes creating a specialized IP core designed to accelerate object detection. It involves translating the model into a hardware description that can be deployed on an FPGA, thereby leveraging the technology's parallel processing capability and efficiency. By developing an IP core, the goal is to improve processing speed and optimize resource usage, ensuring that the object detection system meets the strict requirements of real-time, low-latency applications.

There are two methods for implementing the IP core, which are using NNgen framework and Xilinx HLS.

a) *NNgen Framework*

The software used in this approach is NNgen, and the object detection model used is YOLOv3-tiny. NNgen is an open-source, domain-specific high-level synthesis compiler, implemented in Python, that generates processing hardware circuits [21].

The NNgen framework converts the YOLOv3-tiny, trained model (developed in PyTorch) into an Open Neural Network Exchange (ONNX) file, which is then transformed into NNgen format. The framework performs quantization by converting the floating-point weight parameters into integers using NNgen's full-range quantization function. During quantization, it is necessary to properly truncate values by right-shifting to prevent overflow in calculation results across network layers. The quantized number range can be calculated using the following formula:

$$(N_{max}, N_{min}) = [-2^{k-1}, 2^{k-1}] \qquad (1)$$

where: N_{max} = Maximum integer value in the range
N_{min} = Minimum integer value in the range
k = Number of quantization bits

At the final stage, NNgen generates a Verilog HDL description and an IP-core package (IP-XACT) of the CNN accelerator, based on the custom model specification.

b) *Xilinx HLS Software*

For this approach, the YOLOv2 object detection model is implemented. The IP core generation is performed using Vivado HLS 2018.2 software, following a systematic process. First, it is required to understand the YOLOv2 model algorithm and its architecture to represent it in C/C++ code. During implementation, quantization is performed to convert floating-point values into fixed-point values, reducing data precision while optimizing efficiency. Next, HLS converts the C/C++ source code into RTL code, making it suitable for FPGA implementation. Following synthesis, the design is exported in IP catalog format using the export design command, resulting in the generation of the IP core.

The IP core generated is then integrated into the target FPGA system using tools such as Vivado Design Suite. Proper configuration of connections and interfaces is necessary to enable communication between system components.

2) *Integration of IP core with the processing unit*

The IP core generated using NNgen or Xilinx HLS serves as the entry point for integration in Xilinx Vivado software. Block design is implemented in Vivado IP Integrator to connect the generated IP core with the CPU core of Zedboard. For communication between the CPU and IP cores, Advanced eXtensible Interface (AXI) interconnects are used. This technique is known as hardware acceleration, as computational tasks are offloading from the main processor to the IP core for execution on Zedboard. The block design for YOLOv2 and YOLOv3-tiny is shown in Figure 2.

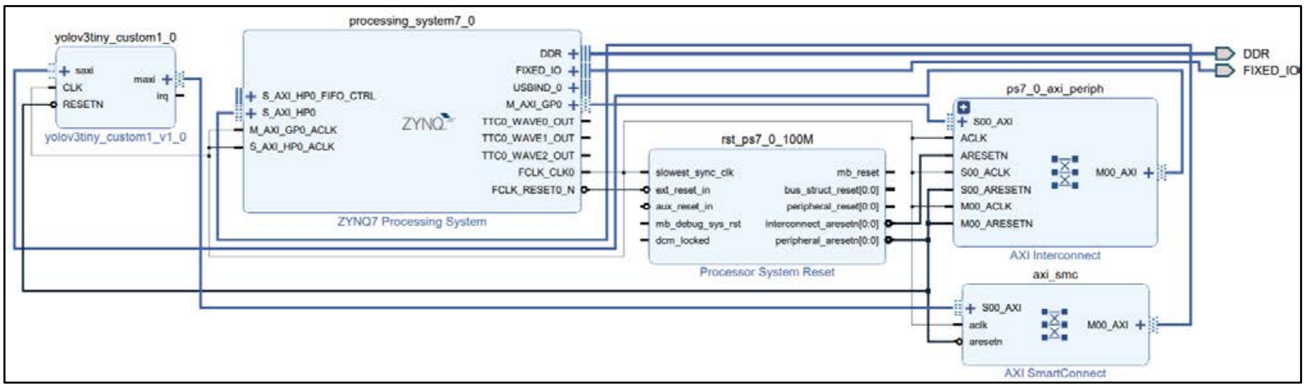
At this stage, the bitstream is generated, and the block design, along with the hardware configuration, is exported for further use in PYNQ for object detection. The hardware configuration for the FPGA accelerator equipped with the YOLOv3-tiny, dedicated circuit is completed, successfully integrating the hardware accelerator with the FPGA board. This marks the final step before deploying and evaluating the object detection system.

3) *Deployment of object detection model in Zedboard*

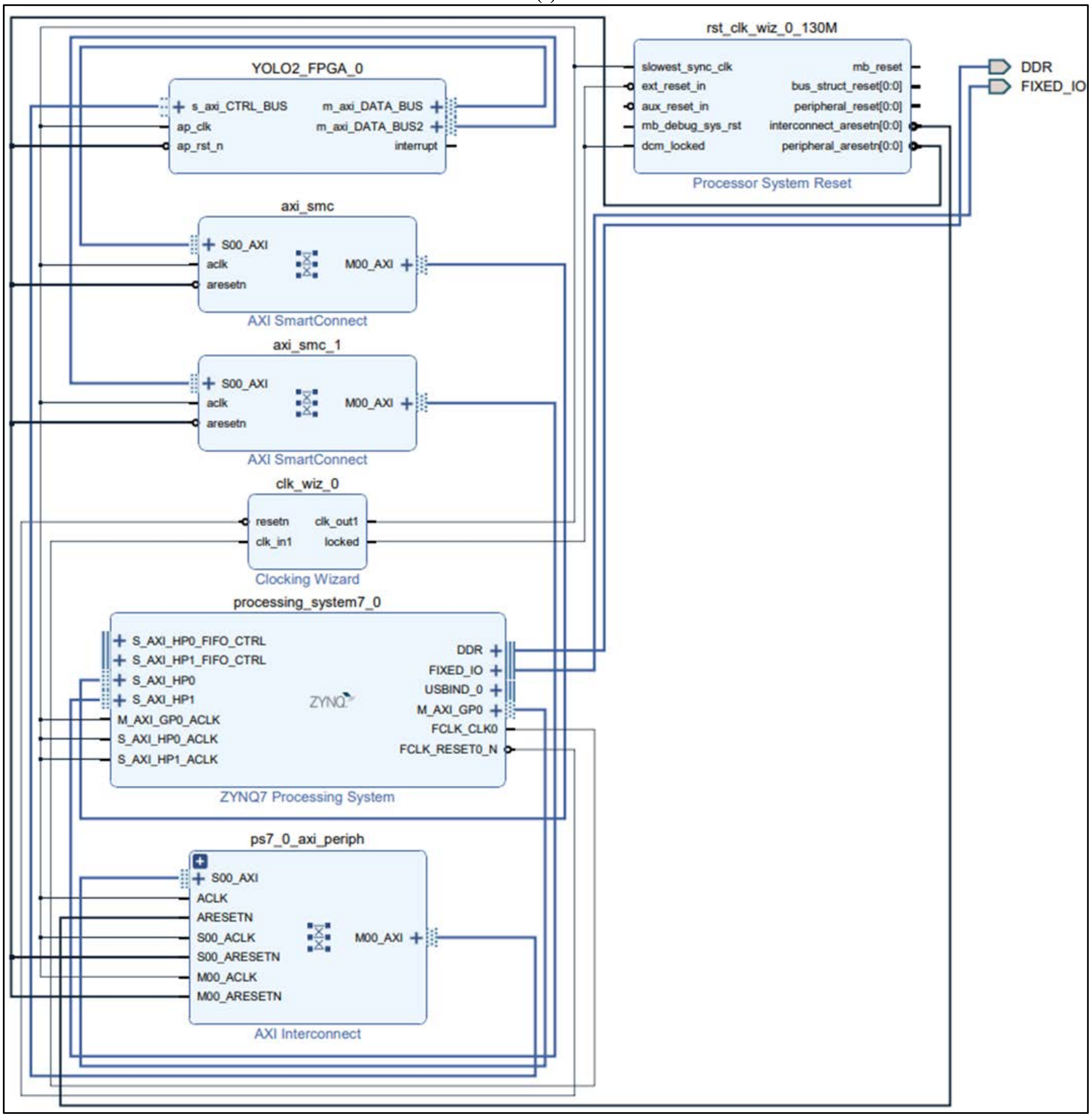
To deploy the object detection model circuit developed in Xilinx Vivado, the PYNQ framework and Jupyter Notebook are utilized. The Zedboard must be set up with the PYNQ environment, which is achieved by preparing an SD card with the required files. The necessary hardware setup includes SD cards (for PYNQ OS installation), ZedBoard, Ethernet cable, Ethernet adapter, and power supply. The hardware connections for the ZedBoard setup is as shown in Figure 3.

Figure 3. Connection of necessary hardware with ZedBoard

The generated bitstream (.bit) file and hardware (.hdf) file from Xilinx Vivado software are used to perform inference in the PYNQ environment using Python language. This integration allows for seamless hardware acceleration of the object detection model on the Zedboard, leveraging the efficiency and flexibility of the PYNQ framework and Python-based development.



(a)



(b)

Figure 2. Block design to integrate the custom IP core and Zedboard’s CPU core (a) YOLOv3-tiny IP core (b) YOLOv2 IP core

B. Deployment on Jetson Nano

The NVIDIA Jetson Nano equipped with a CUDA-capable GPU was chosen for its advanced AI capabilities [22]. With TensorRT support, a compact design and low power consumption, it offers efficient and real-time AI processing. Supporting popular deep learning frameworks, Jetson Nano is an ideal choice for edge computing applications [9]. TensorRT is an SDK for deep learning inference, developed by NVDIA to optimize neural network models for deployment on NVDIA GPUs. It generates an optimized runtime engine which increases throughput and decreases latency. The runtime engine is then used for executing object detection tasks on the GPU. Figure 4 shows the overall flow of building a TensorRT engine from YOLOv7 models and running inference.

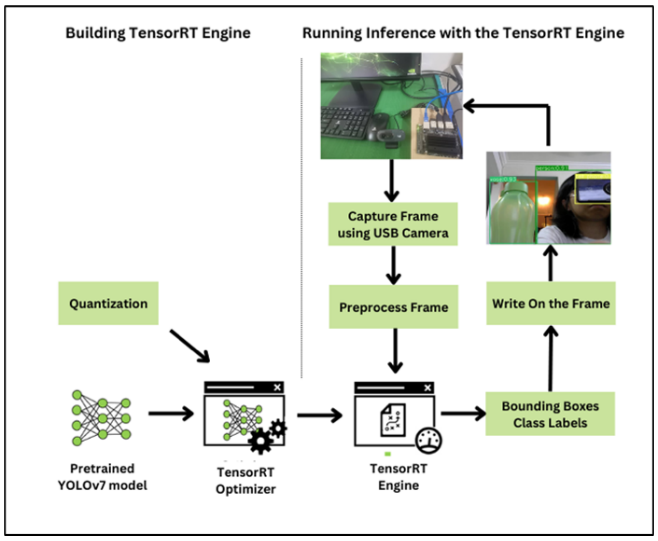


Figure 4. Overall flow of inference optimization for object detection task in Jetson Nano.

1) Model Conversion

TensorRTx is a popular open-source library that utilizes the TensorRT network definition API to build network structures for accelerating deep learning models [22]. The first step involves extracting the pretrained YOLOv7 model's parameters from PyToch to generate a binary weights file stored in hexadecimal format, which will be used in TensorRTx.

To run deep learning algorithms efficiently on edge devices, models have to be compressed into lighter versions. Typically, NVIDIA has its own deep learning toolkit, TensorRT, which compiles the trained model into a format that better utilizes the hardware in the edge device through means like quantization and hardware-optimized layers. In this work, post-training quantization is performed using the TensorRT capability. However, due to Jetson Nano's inability to support 8-bit signed integer (INT8) quantization, models are converted into half-precision, 16- bit floating point (FP16) format.

The optimized model is then converted into an engine file. The implementation of the engine using TensorRTx involves several steps, including creating a builder, configuring the engine, and serializing the model. Before generating the runtime engine, the configuration file must be modified to set the model's parameter based on requirements such as quantization precision, input size and confidence threshold for object detection. The evaluated models, based on different precision and input sizes, are tabulated in Table 1.

Table 1
Models with Different Configurations

Model	Precision	Input Size
YOLOv7-tiny	FP16	416x416
	FP16	640x640
	FP32	640x640
YOLOv7	FP16	640x640
	FP32	640x640

2) Running Inference

The created serialized engine file needs to be read and deserialized to run the inference. The process of running inference can be implemented using C++ language or Python language. TensorRT engine performs inference on input data by preparing input tensors, performing necessary pre-processing steps, executing inference on the GPU, retrieving output tensors in terms of bounding box and class labels generated by the model, and performing post-processing such as non-suppression and drawing bounding box for visualization.

This process continues for real-time object detection on single frames. The input frames are captured using a USB camera, preprocessed such as normalization, resizing and format conversion and then fed into the engine file for inference.

3) Model Performance Evaluation

To gain insight into different aspects of model's capabilities, key metrics analyzed include Mean Average Precision (mAP), Frames Per Second (FPS), and Resource Utilization. The object detection models were evaluated on the NVIDIA Jetson Nano Developer Kit, which features a 128-core Maxwell GPU and a quad-core ARM Cortex-A57 CPU.

(a) Mean Average Precision (mAP) and Frames Per Second (FPS)

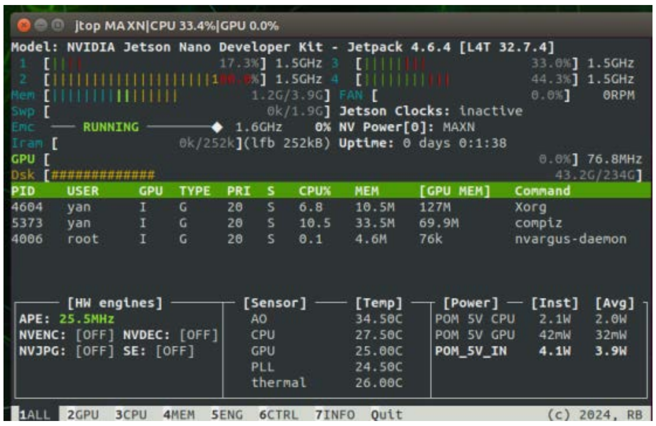
The object detection model was evaluated on the 2017 MS COCO Validation dataset, which consists of 5000 images of varying sizes with ground truth annotations in COCO format [23]. The COCO evaluator, part of the pycocotools Pythoon API, was used for performance assessment. In COCO mAP calculations, a 101-point interpolated mAP definition is used. For COCO, mAP is the average AP over multiple IoU (the minimum IoU to consider a positive match). The mAP@0.5:0.95 corresponds to the average AP for IoU thresholds from 0.5 to 0.95 (with a step size of 0.05), while mAP@0.5 represents the mAP score at an IoU threshold of 0.5.

In this work, "accuracy" specifically refers to the mAP metric, which evaluates object detection performance by measuring how well predicted bounding boxes match the ground truth at various IoU thresholds. This should not be confusing with general classification accuracy, which measures the percentage of correct predictions in classification tasks.

Frames Per Second (FPS) measures how many images the object detection model can process per second. In this work, FPS is obtained by running inference in real-time and measuring the inference time per frame, hence it is also called single frame processing. FPS is calculated as the inverse of the inference time.

b) Resource Utilization

The monitoring and evaluation of resource utilization were conducted using the Python package jetson-stats, which provides an easy-to-use API for NVIDIA's tegrastats. The jtop command was executed to launch an interface for real-time statistics monitoring, as shown in Figure 5. The monitored statistics included power consumption, CPU/GPU utilization, and inference latency. Monitoring was performed while running the model's inference with a batch size of 1. To ensure accurate measurement and maximize performance, the inference was executed on the maximum 10W power mode. Additionally, no other software was running during the measurement, and only the necessary peripherals (monitor, keyboard, mouse, and Ethernet cable) were connected.



```
jtop MAXN|CPU 33.4%|GPU 0.0%
Model: NVIDIA Jetson Nano Developer Kit - Jetpack 4.6.4 [L4T 32.7.4]
 1  [||||                17.3%] 1.5GHz 3  [||||||||              33.0%] 1.5GHz
 2  [||||||||||||||||||||100.0%] 1.5GHz 4  [|||||||||||           44.3%] 1.5GHz
Mem [|||||||||||||||      1.2G/3.9G] FAN [                       0.0%]    0RPM
Swp [                       0k/1.9G] Jetson Clocks: inactive
Emc ── RUNNING ──────◆ 1.6GHz     0% NV Power[0]: MAXN
Iram [              0k/252k](lfb 252kB) Uptime: 0 days 0:1:38
GPU [                                                         0.0%] 76.8MHz
Dsk [#############                                         43.2G/234G]
PID    USER    GPU  TYPE  PRI  S   CPU%   MEM     [GPU MEM]  Command
4604   yan     I    G     20   S   6.8    10.5M   127M       Xorg
5373   yan     I    G     20   S   10.5   33.5M   69.9M      compiz
4006   root    I    G     20   S   0.1    4.6M    76k        nvargus-daemon

 ──── [HW engines] ────── [Sensor] ──── [Temp] ── [Power] ─ [Inst]  [Avg]
 APE: 25.5MHz               AO           34.50C    POM 5V CPU  2.1W    2.0W
 NVENC: [OFF] NVDEC: [OFF]  CPU          27.50C    POM 5V GPU  42mW    32mW
 NVJPG: [OFF] SE: [OFF]     GPU          25.00C    POM_5V_IN   4.1W    3.9W
                            PLL          24.50C
                            thermal      26.00C
1ALL  2GPU  3CPU  4MEM  5ENG  6CTRL  7INFO  Quit                (c) 2024, RB
```

Figure 5. Interface of jtop for statistics monitoring

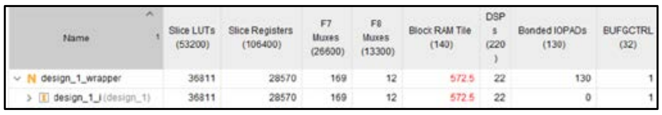
IV. RESULTS AND DISCUSSION

The effectiveness of deploying YOLO object detection models on the chosen hardware platforms, including Zedboard and Jetson Nano, was evaluated. Comparisons were made between different models and hardware configurations to highlight the strengths and weaknesses of each approach.

A. Zedboard

For Zedboard, performance evaluation was conducted using two object detection models, YOLOv2 with 16-bit integer (INT16) quantization and YOLOv3-tiny with 16-bit floating-point (FP16) quantization. The evaluation considered resource utilization, power consumption, and clock frequency, which were analyzed using the synthesis results from Xilinx Vivado software. Additionally, processing time was assessed using the PYNQ environment of Zedboard.

Both models were trained on the COCO dataset, with YOLOv2 pre-trained on 80 classes, and YOLOv3-tiny re-trained on a single class (person detection) due to resource constraints, as shown in Figure 6. However, the Block RAM (BRAM) limitation posed a significant challenge, hindering the development of YOLOv3-tiny from synthesis to implementation and bitsream generation for deployment.



Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Block RAM Tile (140)	DSPs (220)	Bonded IOPADs (130)	BUFGCTRL (32)
design_1_wrapper	36811	28570	169	12	572.5	22	130	1
design_1_i (design_1)	36811	28570	169	12	572.5	22	0	1

Figure 6. Pre-trained YOLOv3-tiny model with BRAM limit exceeded in Xilinx Vivado Synthesis

The resource utilization for both models is summarized in Table 2. The resources evaluated include Look-Up tables (LUTs), Flip-Flops (FFs), Block RAMs (BRAMs), and Digital Signal Processing blocks (DSPs).

From Table 2, both YOLOv2 and YOLOv3-tiny models exhibit almost similar usage of LUTs, FFs, and BRAMs, indicating that the overall logic and memory demands of the two models are relatively comparable. However, YOLOv2 utilizes significantly more DSPs (48.18%) than YOLOv3-tiny (9.09%), primarily due to its higher arithmetic requirements for multi-class detection. In contrast, YOLOv3-tiny's single-class detection requires fewer arithmetic operations, leading to lower DSO utilization.

Table 2
Resource Utilization for Different Models in Zedboard

Resource	YOLOv2	YOLOv3-tiny
LUT (53,200)	29,608 (55.65%)	33,589 (63.14%)
FF (106,400)	28,752 (27.02%)	27,025 (25.40%)
BRAM (140)	100.50 (71.79%)	93 (66.43%)
DSP (220)	106 (48.18%)	20 (9.09%)

The power consumption, clock frequency, and FPGA process time for each model are presented in Table 3.

Table 3
Power Consumption and Performance for Different Models in Zedboard

Performance	YOLOv2	YOLOv3-tiny
Power (W)	2.294	2.177
Clock Frequency (MHz)	130	100
FPGA Process Time (s)	2.67	3.71

From Table 3, both models have similar power consumption, with YOLOv2 consuming 2.294W and YOLOv3-tiny consuming 2.177W. However, YOLOv3-tiny has a longer FPGA processing time (3.71 seconds) compared to YOLOv2 (2.67 seconds). This indicates that YOLOv3-tiny is less efficient in terms of processing speed on the FPGA. The difference in processing speed can be attributed to the deployment via NNgen, as opposed to YOLOv2's deployment through the HLS method.

In conclusion, deploying YOLOv3-tiny on ZedBoard using NNgen proved to be less efficient compared to YOLOv2 using HLS. Although both models utilize similar FPGA resources, the longer processing time of YOLOv3-tiny suggests that Ngen deployment is suboptimal for real-time applications. Further, neither model is suitable for real-time object detection on edge devices due to their high FPGA processing time. These findings highlight the need for further optimization or alternative hardware solutions to achieve real-time performance for object detection tasks on resource-constrained edge devices.

B. Jetson Nano

The performance evaluation of object detection models on the Jetson Nano focused on three key metrics: mean average precision (mAP), frames per second (FPS), and resource utilization.

The model's effectiveness is primarily determined by its inference time when performing an object detection task. To evaluate the efficiency of the proposed optimization method using TensorRT, the optimized model is compared with the baseline model. For this study, YOLOv7-tiny model with a 640x640 input size and FP32 precision implemented in PyTorch serves as the baseline. The optimized versions include TensorRT YOLOv7-tiny models, where inference is implemented using either C++s or Python language. The results are tabulated in Table 4.

Table 4
YOLOv7-tiny Inference Times using Different Frameworks

Model	Inference Method	Inference Time
YOLOv7-tiny	PyTorch	112
	TensorRT Python	90
	TensorRT C++	88

From Table 4, it clearly shows that using TensorRT significantly accelerates the YOLOv7-tiny model's inference time compared to PyTorch. Furthermore, additional improvements are observed when using TensorRT with C++, which achieves a slightly better inference time (88 ms) compared to TensorRT with Python (90 ms). This indicates that leveraging TensorRT and transitioning the inference process to C++ can significantly enhance the model's performance, enabling faster and more efficient object detection.

a) Mean Average Precision (mAP) and Frames Per Second (FPS)

The next evaluation focuses on models with different input sizes and quantization precision, generated using TensorRT. Table 5 shows the inference times comparison of different configurations of YOLOv7 models that the inference process is generated either using C++ or Python. Table 6 summarizes the results of evaluation on mAP and FPS metrics at different Intersection over Union (IoU) thresholds. The trade-offs between accuracy and speed are further illustrated in Figure 7, highlighting the impact of quantization and input size on model accuracy and inference speed.

Table 5
Comparison of Inference Time for Different Configuration on YOLOv7 Models

Model	Precision	Size	Inference Time	
			C++	Python
YOLOv7-tiny	FP16	416x416	39	26
	FP16	640x640	58	62
	FP32	640x640	88	90
YOLOv7	FP16	640x640	326	330
	FP32	640x640	555	559

Table 6
Comparison of Mean Average Precision (mAP) and Frames Per Second (FPS) for Different Configuration on YOLOv7 Models

Model	Precision	Size	mAP@0.5	mAP@0.5:0.95	FPS
YOLOv7-tiny	FP16	416x416	46.3%	30.9%	38
	FP16	640x640	50.7%	34.0%	16
	FP32	640x640	50.8%	34.0%	11
YOLOv7	FP16	640x640	46.8%	64.6%	3
	FP32	640x640	46.8%	64.6%	1

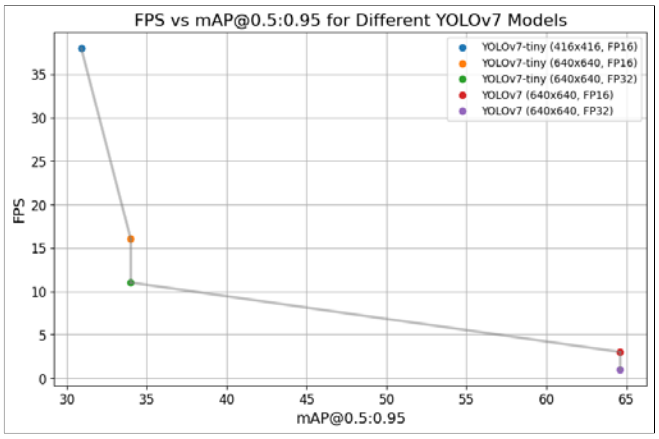


Figure 7. Plot of FPS vs mAP@0.5:0.95 for different YOLOv7 models

From Table 5 and Table 6, it is evident that smaller models result in shorter inference time, higher speed and lower accuracy. The inference process implemented in C++ consistently outperforms Python in terms of speed. Besides, the results show that reducing the floating-point precision from FP32 to FP16 does not significantly impact model accuracy, highlighting the efficiency of TensorRT quantization.

The trade-offs shown in Figure 7 underscore the inherent challenge of balancing speed and accuracy in real-time object detection systems. The choice between YOLOv7-tiny and YOLOv7 hinges on the specific application requirements. Applications prioritizing real-time processing may prefer YOLOv7-tiny for its superior FPS performance. Conversely, tasks demanding higher detection precision may lean towards YOLOv7 despite its lower FPS. The flexibility of TensorRT in handling different configurations highlight its crucial role in optimizing inference time tailored to specific application needs, enabling efficient deployment across a spectrum of use cases.

b) Resource Utilization

The evaluations of resource utilization include power consumption, CPU utilization, and GPU memory consumption, and latency for different YOLOv7 model configurations implemented using TensorRT framework.

Table 7
Comparison of Resource Utilization for Different Configuration on YOLOv7 Models

Model	Precision	Size	Power (W)	CPU (%)	GPU Memory	Latency (ms)
YOLOv7-tiny	FP16	416x416	5.1	64.9	251M	26
	FP16	640x640	7.0	68.6	265M	58
	FP32	640x640	7.7	65.8	327M	88
YOLOv7	FP16	640x640	8.6	37.3	553M	328
	FP32	640x640	9.0	52.0	892M	553

Table 7 provides a comprehensive summary of key performance metrics, including power consumption, CPU utilization, GPU memory usage, and latency. However, to offer a more visual representation and facilitate a comparative analysis, Figure 8 presents the plot of performance and resource utilization for different YOLOv7 models.

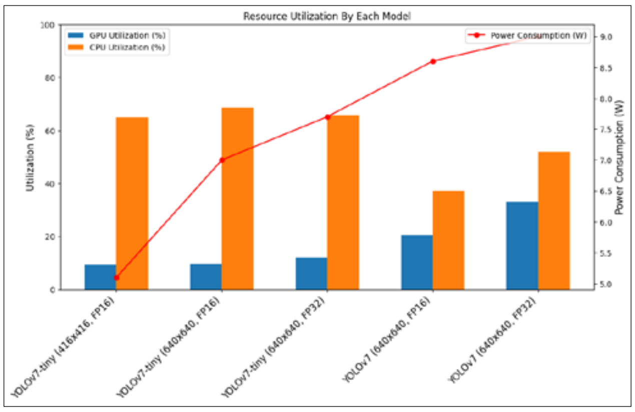


Figure 8. Plot of performance and resource utilization for different YOLOv7 models

From Table 7, it is evident that YOLOv7-tiny models consume less power than YOLOv7 models due to their lighter architecture. Power consumption increases with both input size and quantization precision. In terms of CPU and GPU utilization, YOLOv7-tiny models show higher CPU utilization because their simpler design allows the CPU to handle more tasks. In contrast, YOLOv7 models use more GPU resources, as their more complex structure benefits from the parallel processing power of the GPU. Latency is lower in YOLOv7-tiny models, making them more suitable for real-time applications on edge devices.

In summary, YOLOv7-tiny models, especially with FP16 precision, strike a good balance between power consumption, latency, and resource usage, making them ideal for real-time applications on edge devices with limited resources. YOLOv7 models, on the other hand, offer higher accuracy but require more power and computational resources, making them more suitable for applications where accuracy is a higher priority and resource constraints are less of a concern.

It clearly shows that the results highlight the trade-offs between model complexity, precision, and performance in Figure 8. YOLOv7-tiny models, particularly at FP16 precision, offer a good balance between power consumption and resource utilization, making them suitable for real-time applications on resource-constrained devices. On the other hand, YOLOv7 models, while providing higher accuracy, demand significantly more power and computational resources, making them better suited for applications where higher accuracy is critical and resource availability is not a primary constraint.

c) Comparison with Previous Work

The performance comparison of different hardware accelerators for object detection models reveals significant variations in accuracy, frames per second (FPS), and power consumption, as shown in Table 8. The model for comparison in this work is the tiny version of YOLOv7 with 416x416 input size and FP16 quantization.

Based on Table 8, although the proposed method does not achieve the highest accuracy and FPS with the lowest power consumption, it still outperformed the previous works in terms of overall efficiency. This advantage is attributed to the balanced trade-offs offered by the proposed method. Compared to FPGA-based solutions, such as the Xilinx ZCU706 and Intel Arria 10, the proposed GPU-based approach on the Jetson Nano demonstrates competitive performance in both accuracy and FPS. Specifically, it achieves a significantly higher FPS than other ASIC-based and GPU-based implementations, such as the Google Coral AI Development Board TPU with SSD MobileNet-V1 and Jetson TX2 with Fast R-CNN, making it well-suited for real-time applications. Additionally, the power consumption of the proposed method is comparable to other GPU-based solutions, ensuring energy-efficient operation.

Table 8
Model's Performance Comparison Table

Citation	Hardware Accelerator	Platform	Model	Precision	Accuracy	FPS	Power (W)
[24]	FPGA	Xilinx ZCU706	SSDLite-Mobile-NetV2	INT8	20.3	65	9.9
[15]	FPGA	Intel Arria 10	YOLO V2	INT8-16	73.6	25	27.2
[25]	ASIC	Coral Dev. Board TPU	RetinaNet ResNet-50	-	53.94	4	4.52
[16]	ASIC	Google Coral AI Dev. Board TPU	SSD MobileNet-V1	INT8	66.96	10	2.5
[17]	GPU	Jetson Nano	YOLO V3 Tiny	FP16	67.3	4	5.5
[18]	GPU	Jetson TX2	Fast R-CNN	-	45.32	20	-
[26]	GPU	NVIDIA Titan X	RefineDet512	FP32	81.8	24	-
[27]	GPU	NVDIA Tesla V100	YOLO V5s	FP16	36.8	45	-
[19]	GPU	NVDIA Jetson AGX Xavier	YOLOX-DarkNet53	FP16	47.4	50	-
[28]	GPU	NVDIA Jetson AGX Xavier	YOLO V4	FP16	23.45	3	0.07
This work	GPU	Jetson Nano	YOLO V7 Tiny	FP16	46.3	38	5.1

However, the NVIDIA Titan X may outperform the proposed method if further optimization techniques, such as quantization are applied. This is because the NVIDIA Titan X is a more powerful GPU compared to the Jetson Nano's GPU, offering superior computational capabilities. By leveraging these capabilities and incorporating advanced optimization techniques, the model's performance in terms of accuracy and FPS could be significantly improved.

In conclusion, the comparison highlights the suitability of the proposed method for real-world deployment, providing a practical balance between performance and power efficiency. Additionally, it highlights the potential for further improvements when deployed on more powerful hardware.

d) Running Inference in Jetson Nano

The results of running real-time object detection in the graphical user interface (GUI) of the Jetson Nano are depicted in Figure 9. The GUI enables access to a live camera feed, facilitating real-time object detection in captured images. Additionally, images captured from the camera feed were saved as image files for further analysis and reference.

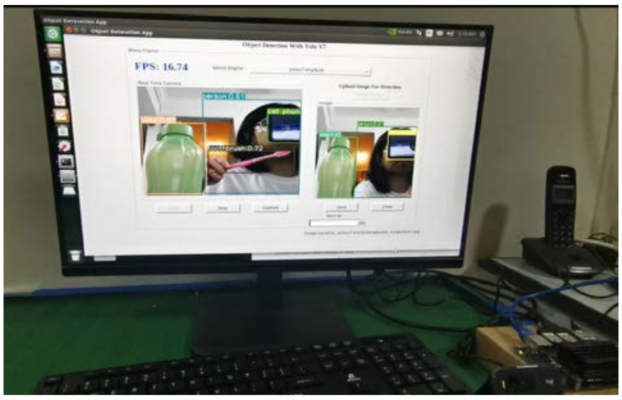


Figure 9. Result of running real-time object detection in the GUI of the Jetson Nano

This result demonstrates the practical implementation of real-time object detection on the Jetson Nano platform using live camera feeds, showcasing its ability to efficiently process live camera feeds and perform real-time inference with high responsiveness.

V. CONCLUSION

In this study, a quantized object detection model was developed and deployed on edge computing platforms, specifically Zedboard and Jetson Nano, to address the challenges of efficient real-time object detection in resource-constrained environments. While The Zedboard was capable of executing quantized models, it faced limitations in processing speed, making it less suitable for real-time applications and emphasizing the need for further optimization in FPGA-based systems. In contrast, the Jetson Nano, utilizing FP16 quantization and GPU acceleration, demonstrated significant improvements in both inference speed and real-time performance. Experimental results using the YOLOv7-tiny model (416x416 input) achieved 38 FPS with an mAP of 46.3%, showcasing a balance between speed and accuracy. These findings underscore the effectiveness of combining quantization with GPU acceleration for optimizing deep learning models on edge devices.

However, this work has certain limitations, primarily due to hardware constraints and the trade-offs between performance and model accuracy. The available edge computing devices have limited processing power, memory, and storage capacity, restricting the complexity and size of the machine learning models that can be deployed. While hardware acceleration improves processing speed and efficiency, it may come at the cost of reduced accuracy, particularly in complex real-world scenarios. Optimizing the system for real-time processing often requires techniques like quantization or model simplification, which can further impact object detection accuracy. Thus, balancing hardware constraints with performance requirements for effective real-time object detection remains a significant challenge in the development of efficient systems for edge computing environments.

This research lays the groundwork for future advancements in real-time object detection on edge computing devices. Future work could focus on fine-tuning models for specific applications, such as traffic monitoring or smart surveillance, to enhance detection accuracy and performance. Additionally, further exploration of advanced quantization techniques and hybrid accuracy methods could improve system performance while maintaining accuracy in dynamic environments. The integration of these models into fully automated systems with user-friendly graphical user interfaces (GUI) could open up practical applications across various industries, addressing real-time object detection challenges in resource-constrained environments.

ACKNOWLEDGMENT

The authors acknowledge the technical and financial support by the Ministry of Higher Education, Malaysia, under research grant no. FRGS/1/2024/ICT02/UTEM/02/3, and Universiti Teknikal Malaysia Melaka (UTeM).

CONFLICT OF INTEREST

Authors declare that there is no conflict of interest regarding the publication of the paper.

AUTHOR CONTRIBUTION

The authors confirm contribution to the paper as follows: study conception and design: Yan Chiew Wong, Hwee Min Chiam; data collection: Hwee Min Chiam; analysis and interpretation of findings: Hwee Min Chiam, Yan Chiew Wong; draft manuscript preparation: Hwee Min Chiam, Ranjit Singh Sarban Singh, T. Joseph Sahaya Anand. All authors had reviewed the findings and approved the final manuscript.

REFERENCES

[1] Y. Daga and S. Meena, “Applications of human activity recognition in different fields: a review,” in Proc. of 2022 IEEE 9th Uttar Pradesh Section International Conference on Electrical, Electronics and Computer Engineering (UPCON), 2022.
[2] G. L. Nguyen et al., “A low-cost real-time IoT human activity recognition system based on wearable sensor and the supervised learning algorithms,” Measurement, vol. 218, 2023.
[3] A. Kayid, Y. Khaled and M. Elmahdy, “Performance of CPUs/GPUs for deep learning workloads,” Research Report, German University in Cairo, 2018.
[4] V. Sze, Y. -H. Chen, J. Einer, A. Suleiman and Z. Zhang, “Hardware for machine learning: challenges and opportunities,” in Proc. of 2017 IEEE Custom Integrated Circuits Conference (CICC), 2017.
[5] P. S. J. Zhu, “FPGA implementations of neural networks – a survey of a decade of progress,” in Field Programmable Logic and Application, Springer, Berlin, Heidelberg, 2003.
[6] E. Kevin et al., “Robust physical-world attacks on deep learning visual classification,” in Proc. of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018.
[7] J. Huang et al., “Speed/accuracy trade-offs for modern convolutional object detectors,” in Proc. of 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 3296-3297.

[8] V. Ndonhong, A. Bao and O. Germain, “Wellbore schematics to structured data using artificial intelligence tools,” in Proc. of Offshore Technology Conference, 2019.

[9] A. Setyanto, T. B. Sasongko, M. A. Fikri and I. K. Kim, “Near-edge computing aware object detection: a review,” IEEE Access, vol. 12, pp. 2989-3011, 2024.

[10] R. G. Suri and Niranjan, “Object detection at the edge: off-the-shelf deep learning capable devices and accelerators,” Procedia Computer Science, vol. 205, pp. 239-248, 2022.

[11] B. J. Kalenichenko, K. Skirmantas, C. Bo, Z. Menglong, T. Matthew, H. Andrew, A. Hartwig and Dmitry, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,”, Conference on Computer Vision and Pattern Recognition, 2017.

[12] R. Li, Y. Wang, F. Liang, H. Qin, J. Yan and R. Fan, “Fully quantized network for object detection,” in Proc. of 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2019, pp. 2805-2814.

[13] A. Kyriakos, V. Kitsakis, A. Louropoulos, E. -A. Papatheofanous, I. Patronas and D. Reisis, “High performance accelerator for CNN applications,” in Proc. of 2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS), 2019.

[14] P. Jawandhiya, “Hardware design for machine learning,” International Journal of Artificial Intelligence and Applications, vol. 9, pp. 63-84, 2018.

[15] S. Li, Y. Luo, K. Sun, N. Yadav and K. Choi, “A novel FPGA accelerator design for real-time and ultra-low power deep convolutional neural networks compared with titan X GPU,” IEEE Access, 2020.

[16] A. S. Aguiar et al., “Grape bunch detection at different growth stages using deep learning quantized models,” Agronomy, vol. 11, 2021.

[17] C. Choe, M. Choe and S. Jung, “Run your 3D object detector on NVIDIA jetson platforms: a benchmark analysis,” Sensors, vol. 23, pp. 4005, 2023.

[18] N. H. Abdulghafoor and H. N. Abdullah, “Real-time moving objects detection and tracking using deep-stream technology,” Journal of Engineering Science and Technology, vol. 16, pp. 194-208, 2021.

[19] P. Divakar, and V. Pavani, “Implementation of an object detection system using convolutional neural networks,” International Journal of Research Publication and Reviews, vol. 5, no. 3, pp. 3865–3868, 2024.

[20] AVNET, “ZedBoard,” Accessed: Jun. 11, 2024, [Online.] Available: http://www.zedboard.org/product/zedboard.

[21] Konica Minolta, "Technology for hardware implementation of AI algorithms - technology,” Accessed: Feb. 23, 2024, [Online.] Available: https://research.konicaminolta.com/en/technology/tech_details/nngen/

[22] NVIDIA, “TensorRT documentation,” Accessed: June. 11, 2024, [Online.] Available: https://docs.nvidia.com/deeplearning/tensorrt/index.html.

[23] A. Veit, T. Matera, L. Neumann, J. Matas and S. Belongie, “COCO-text: dataset and benchmark for text detection and recognition in natural images,” arXiv: Computer Vision and Pattern Recognition, 2016.

[24] H. Fan et al., “A real-time object detection accelerator with compressed SSDLite on FPGA,” in Proc. of 2018 International Conference on Field-Programmable Technology (FPT), 2018.

[25] M. S. Costa, N. Filipe, M. Pedro, P. M. António and D. Jorge, “Benchmarking edge computing devices for grape bunches and trunks detection using accelerated object detection single shot multibox deep learning models,” Engineering Applications of Artificial Intelligence, vol. 117, no. 0952-1976, 2023.

[26] S. Zhang, L. Wen, X. Bian, Z. Lei and S. Li, “Single-shot refinement neural network for object detection,” in Proc. of 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2018, pp. 4203-4212.

[27] D. Dlužnevskij, P. Stefanovič and S. Ramanauskaitė, “Investigation of YOLOv5 efficiency in iPhone supported systems,” Baltic Journal of Modern Computing, vol. 9, 2021.

[28] P. Ruiz-Barroso, F. M. Castro and N. Guil, “Real-time unsupervised object localization on the edge for airport video surveillance,” Pattern Recognition and Image Analysis, pp. 466–478, 2023.