A Controlled Benchmark of Edge AI Accelerators for Traffic Object Detection

Master of Science in Technology Thesis
University of Turku
Department of Computing
Turku Intelligent Embedded and Robotic Systems (TIERS) Lab
2026
Ashish Shrestha

Supervisors:
MSc. Yasir Al-Ameri
MSc. Minh Nguyen
Prof. Tomi Westerlund

UNIVERSITY OF TURKU
Department of Computing

ASHISH SHRESTHA: A Controlled Benchmark of Edge AI Accelerators for Traffic Object Detection

Master of Science in Technology Thesis, 83 p.
Turku Intelligent Embedded and Robotic Systems (TIERS) Lab
July 2026

Stringent power and cost constraints have driven the adoption of purpose-built edge accelerators over conventional processors for real-time visual perception, yet architectural heterogeneity prevents direct comparison of published specifications for these accelerators. To resolve this, this thesis presents a controlled benchmark of four edge-inference paradigms: a fixed-function TPU (Coral Dev Board Mini), an SoC NPU (Khadas VIM3), an embedded GPU (NVIDIA Jetson Nano), and a reconfigurable FPGA (AMD Kria KV260). Under a unified evaluation protocol, an INT8 EfficientDet-Lite2 detector, trained on a seven-class BDD100K subset, was deployed across the platforms, with a YOLOv4-416 substitute on the GPU due to toolchain constraints. Each platform was measured on latency, accuracy, power, energy, and cost. Results demonstrate that no single paradigm dominates, and rated throughput is an unreliable performance predictor. Despite its 5.0 TOPS rating, the NPU was the slowest and least energy-efficient platform, yielding 0.71 frames per second and consuming 3.84 J per inference. Conversely, despite its lower 4.0 TOPS rating, the TPU ran 4.2 times faster per inference, delivered 6.8 times higher throughput per watt at 1.77 frames per second per watt, and at 91.99 € was the most cost-effective. The two fixed-function accelerators agreed within 0.001 mAP, confirming consistency. However, FPGA deployment collapsed accuracy from 0.2400 to 0.0233. A calibration ablation and bias-bypass experiment isolated this degradation to the per-tensor power-of-two quantization within the classification head rather than the hardware itself.

Keywords: FPGA, NPU, TPU, GPU, edge AI, object detection, INT8 quantization, benchmarking, EfficientDet, BDD100K

Contents

List of Figures

List of Tables

List of Acronyms

AP Average Precision.

APU Application Processing Unit.

ASIC Application-Specific Integrated Circuit.

AXI Advanced eXtensible Interface.

BDD100K Berkeley DeepDrive 100K.

BiFPN Weighted Bi-directional Feature Pyramid Network.

BRAM Block RAM.

CLB Configurable Logic Block.

CNN Convolutional Neural Network.

COCO Common Objects in Context.

CPU Central Processing Unit.

CUDA Compute Unified Device Architecture.

CV Coefficient of Variation.

DPU Deep-learning Processor Unit.

DSA Domain-Specific Architecture.

DSP Digital Signal Processor.

FN False Negative.

FP False Positive.

FP16 16-bit Floating Point.

FP32 32-bit Floating Point.

FPGA Field-Programmable Gate Array.

FPN Feature Pyramid Network.

FPS Frames Per Second.

GPGPU General-Purpose GPU.

GPP General-Purpose Processor.

GPU Graphics Processing Unit.

HDL Hardware Description Language.

INT16 16-bit Integer.

INT8 8-bit Integer.

IoU Intersection over Union.

LUT Look-Up Table.

mAP Mean Average Precision.

MBConv Mobile Inverted Bottleneck Convolution.

MPSoC Multi-Processor System-on-Chip.

MSRP Manufacturer's Suggested Retail Price.

NMS Non-Maximum Suppression.

NN Neural Network.

NPU Neural Processing Unit.

PMU Platform Management Unit.

PTQ Post-Training Quantization.

QAT Quantization-Aware Training.

ReLU Rectified Linear Unit.

RPN Region Proposal Network.

RPU Real-time Processing Unit.

RTL Register-Transfer Level.

SIMT Single-Instruction Multiple-Thread.

SM Streaming Multiprocessor.

SoC System-on-Chip.

SoM System on Module.

TOPS Tera Operations Per Second.

TP True Positive.

TPU Tensor Processing Unit.

UMA Unified Memory Architecture.

USB Universal Serial Bus.

VART Vitis AI Runtime.

VOC PASCAL Visual Object Classes.

VOE Vitis AI ONNX Runtime Engine.

XRT Xilinx Runtime.

1 Introduction

Deep learning has established itself as the dominant paradigm for visual perception, with object detection in particular underpinning applications ranging from advanced driver-assistance systems to mobile robotics [1]. These systems deploy perception models directly on the edge devices capturing the sensory data, rather than offloading inference to remote servers. On-device inference eliminates the network latency overhead that real-time control systems cannot tolerate, preserves data privacy by keeping sensitive imagery local, and sustains operational continuity where connectivity is intermittent. These models must therefore execute within strict latency bounds on hardware constrained by power, thermal, and cost budgets. Satisfying these requirements with general-purpose processors has proven increasingly untenable [2], [3]. In response, a class of specialized inference accelerators has emerged, trading the architectural generality of conventional processors for maximum computational efficiency on the highly parallel operations demanded by deep learning.

These accelerators fall into four broad paradigms, each representing a distinct architectural compromise. A fixed-function Application-Specific Integrated Circuit (ASIC), such as the Tensor Processing Unit (TPU), dedicates its silicon to a narrow set of operators and is highly efficient within that domain. A System-on-Chip (SoC) Neural Processing Unit (NPU) integrates a programmable accelerator alongside a host processor. A programmable Graphics Processing Unit (GPU) offers mature, general-purpose parallel computation. A Field-Programmable Gate Array (FPGA) reconfigures its logic to the

workload, trading raw execution efficiency for flexibility. These paradigms therefore differ not only in inference performance, but also in energy, cost, and the effort required to deploy a model.

The primary objective of this thesis is to benchmark these four acceleration paradigms for real-time traffic perception under a unified experimental framework, ultimately determining whether nominal hardware specifications accurately predict empirical performance. A single object detector, an 8-bit Integer (INT8) EfficientDet-Lite2 model [4], is trained once on a seven-class subset of the Berkeley DeepDrive 100K (BDD100K) driving dataset [5]. The trained model is then deployed on a representative board from each paradigm: a Coral Dev Board Mini, a Khadas VIM3, an NVIDIA Jetson Nano, and an AMD Kria KV260. Each board is measured under a common protocol spanning latency, detection accuracy, power, energy per inference, and cost.

1.1 Significance and Motivation

The significance of this thesis lies in supplying empirical, one-for-one evidence to guide the selection of an edge accelerator for object detection, a decision that vendor specifications and single-platform studies leave unresolved [6]. A practitioner confronts figures that are difficult to compare because they are typically obtained with different models, at different numerical precisions, under different measurement conditions, and reported through different metrics. The single figure most often quoted, rated throughput in Tera Operations Per Second (TOPS), describes peak arithmetic capability rather than the performance an accelerator delivers on a particular model. This thesis holds the workload fixed so that the observed differences can be attributed to the hardware and its toolchain. It addresses the following research questions:

1. How do the four accelerator paradigms (ASIC TPU, SoC NPU, GPU, and FPGA-based Deep-learning Processor Unit (DPU)) compare on latency, detection accu-

racy, power, energy per inference, and device cost when a single trained object detector is deployed on each under a common evaluation protocol, with documented platform-specific adaptations where the target toolchain does not support the primary model?

2. To what extent do accelerator specification-sheet metrics predict real-world suitability for a traffic-perception detection workload?

3. What deployment constraints and toolchain limitations arise when porting a single trained detector across four paradigms whose software stacks were not designed to interoperate?

1.2 Related Work

Benchmarking deep-learning inference on edge hardware is an active research area, driven by the spread of resource-constrained deployment in autonomous systems, robotics, and intelligent transportation. The literature relevant to this thesis sits at the intersection of three threads: the comparative benchmarking of heterogeneous edge accelerators, the deployment of object detectors for traffic perception on embedded devices, and the behavior of quantized detectors on integer-only hardware. Each prior study covers a subset of these threads; none combines the four accelerator classes, the single fixed model, and the traffic-perception dataset that this thesis evaluates under one protocol.

The most directly related work compares several accelerators under a common protocol, yet each covers only part of the design space. Baller et al. [7] introduced DeepEdgeBench, a widely cited framework that evaluates MobileNet and SSD-MobileNet variants across five boards, including the Google Coral Dev Board and the NVIDIA Jetson Nano, and measured inference time, power, and accuracy. For quantized models, the Coral board gave the best combination of speed and efficiency. The framework established a baseline methodology but covers neither an NPU-class SoC nor an FPGA, and it targeted

classification and lightweight detection rather than the EfficientNet family. Kang and Somtham [8] benchmarked YOLOv4-Tiny and SSD-MobileNet V2 on the exact Coral Dev Board Mini used in this thesis, alongside two Jetson devices on MS Common Objects in Context (COCO), and Lema et al. [9] extended the same GPU-and-TPU comparison to the YOLOv3, YOLOv5, and YOLOX detectors; neither work includes an NPU, an FPGA, or a traffic-domain dataset.

A further cluster centers on the EfficientDet family that this thesis deploys. Alqahtani et al. [10] benchmarked the EfficientDet-Lite family and SSD models, with and without an Edge TPU co-processor and on the Jetson Orin Nano, reporting improved throughput without accuracy loss. The study nonetheless lacks an SoC NPU, an FPGA, and a traffic-domain dataset. Broader comparisons by Minott et al. [11] and Tobiasz et al. [12] reinforce the recurring finding that no single device or model dominates across accelerator classes, leaving the NPU and FPGA corners comparatively unexplored.

Two studies approach the breadth that this thesis targets. Magalhães et al. [13] benchmarked embedded GPUs, a Coral TPU, and FPGA-based DPUs in one controlled study, running RetinaNet ResNet-50 and SSD ResNet-50 on an agricultural dataset at 32-bit Floating Point (FP32), 16-bit Floating Point (FP16), and INT8, and reporting the GPUs slowest, the FPGAs fastest, and comparable detection accuracy across all devices with no accuracy collapse on the DPU. It is the closest prior work, covering three of the four accelerator classes, yet it omits the SoC NPU and uses neither the EfficientDet family nor a traffic dataset; its uniform DPU accuracy is revisited below when the contrasting result of this thesis is interpreted. YOLOBench by Lazarevich et al. [14] is the only located peer-reviewed work that benchmarks the Khadas VIM3 NPU directly, measuring over 550 YOLO detectors across an x86 Central Processing Unit (CPU), an ARM CPU, a Jetson Nano GPU, and the VIM3 NPU. It drove the NPU through the Amlogic AML-NPU-SDK at 16-bit Integer (INT16) rather than through the TIM-VX TensorFlow Lite delegate at INT8, and it evaluated YOLO rather than EfficientDet. It therefore fixed a

latency baseline for the exact NPU silicon used here while leaving the software path and model of this thesis unexamined. Systematic reviews of edge-AI deployment [15] and of deep-learning hardware accelerators [16] identify the same structural gap, a shortage of holistic, hardware-aware benchmarking that couples a fixed model and dataset to genuinely heterogeneous accelerators under unified latency, accuracy, and energy metrics.

The second thread narrows to the traffic-perception application that motivates this thesis. Bulut et al. [17] evaluated road-scene detectors for traffic-safety applications on edge hardware, characterizing the accuracy and latency trade-offs of constrained deployment, but on a single target and with models pretrained on general-purpose data. The dataset used here, BDD100K [5], is a large and deliberately diverse driving dataset that has become a standard benchmark for autonomous-driving perception; yet the overwhelming majority of BDD100K work reports state-of-the-art Mean Average Precision (mAP) on high-end GPUs rather than deployment efficiency on low-power accelerators. The closest connection between BDD100K and embedded deployment is Ortiz-Castelló et al. [18], whose resource-constrained evaluation compares forward-pass time against a baseline rather than deploying on dedicated accelerators with power and energy profiling. No located study benchmarks a BDD100K-trained model across heterogeneous dedicated edge accelerators with unified accuracy, latency, and power metrics. The disjunction is structural rather than accidental. The computer-vision community that uses this dataset optimizes mAP on GPUs, whereas the embedded-systems community that benchmarks accelerators gravitates toward COCO-pretrained models that carry published baselines and avoid a full fine-tuning pipeline. The two rarely meet. This thesis sits deliberately at their intersection.

Because every platform in this thesis runs the detector in reduced precision, the literature on integer quantization both informs the method and frames one of its central results. The foundational integer-arithmetic-only scheme of Jacob et al. [19] formalized the affine mapping between real and integer values that enables integer-only inference. Subsequent

analyses located where this mapping breaks down. Wu et al. [20], in NVIDIA's empirical study, showed that per-tensor quantization is particularly damaging for networks built on depthwise-separable convolutions and singled out EfficientNet-style backbones once batch normalization is folded. Nagel et al. [21] attributed the same vulnerability to depthwise weight ranges that are so wide that a single per-tensor scale cannot represent them accurately. At the task level, Niu et al. [22] confirmed that post-training quantization suffers a severe accuracy drop on object detection and proposed task-loss-guided remedies. On the deployment side, Cantero et al. [23] benchmarked SSD, CenterNet, and EfficientDet variants across five quantization levels on a VeriSilicon Vivante NPU and on the Edge TPU, documenting that inference time cannot be predicted from model size and that the deeper levels required by the accelerators introduce accuracy and correctness risks, including conversion failures for EfficientDet. Their study includes no GPU or FPGA and does not use the TIM-VX delegate or a traffic dataset.

Taken together, this literature establishes that the severe accuracy degradation of a quantized one-stage detector on a strictly per-tensor integer datapath is a documented and well-understood phenomenon. The collapse observed in this thesis, when EfficientDet-Lite2 is quantized for the Kria DPU, is consistent with Wu et al. and Nagel et al. on depthwise-separable layers and with Niu et al. on detection-head sensitivity. The contrast with the dense-convolution RetinaNet of Magalhães et al. [13], which showed no such collapse on the same class of DPU, isolates the architecture rather than the accelerator as the sensitive element. Earlier work established this mechanism in controlled quantization research; this thesis appears to be the first to isolate and characterize it on a deployed one-stage detector running on real edge hardware, including the documented unavailability of the standard remedies, per-channel quantization and quantization-aware training, within that toolchain.

These threads converge on a single gap, summarized in Table 1.1. No prior peer-reviewed study spans all four accelerator classes (the ASIC Edge TPU, the SoC NPU,

Table 1.1: Coverage of the most directly comparable prior work against the dimensions of this thesis. A checkmark denotes that a dimension is addressed, "–" that it is not, and "∼" partial coverage. The accelerator classes are the TPU, NPU (an SoC accelerator such as the VeriSilicon Vivante NPU), GPU, and FPGA. "EffDet" denotes the EfficientDet family and "BDD100K" the driving dataset. No prior row covers all four accelerator classes together; this thesis is the only complete row.

	Accelerator class						
Study	TPU	NPU	GPU	FPGA	EffDet	BDD100K	Power
Baller et al. [7]	✓	–	✓	–	–	–	✓
Kang & Somtham [8]	✓	–	✓	–	–	–	✓
Lema et al. [9]	✓	–	✓	–	–	–	✓
Alqahtani et al. [10]	✓	–	✓	–	✓	–	✓
Cantero et al. [23]	✓	✓	–	–	✓	–	–
Magalhães et al. [13]	✓	–	✓	✓	–	–	✓
Lazarevich et al. [14]	–	✓	✓	–	–	–	–
Ortiz-Castelló et al. [18]	–	–	∼	–	–	✓	–
This thesis	✓	✓	✓	✓	✓	✓	✓

the GPU, and the FPGA-based DPU) with one model under a unified evaluation protocol. Magalhães et al. come closest with three classes but omit the NPU; Cantero et al. pair the NPU with the Edge TPU but omit the GPU and FPGA, and the remaining EfficientDet and YOLO-centered studies each cover at most two or three. This thesis is the only complete row, and to the author's knowledge, it additionally provides the first empirical characterization of EfficientDet-Lite2 INT8 inference on the Amlogic A311D NPU through the TIM-VX delegate.

1.3 Contribution

The contributions of this thesis are as follows:

1. Constructed a controlled cross-paradigm benchmark in which a single INT8 detector, EfficientDet-Lite2, trained once on a seven-class BDD100K subset, was deployed and evaluated under one protocol across an Edge TPU, an Amlogic A311D NPU, a Maxwell GPU, and an FPGA, covering latency, accuracy, power, energy, and cost. To the author's knowledge, no prior work measures a single trained object

detector across all four of these accelerator paradigms under one protocol with this metric set.

2. Deployed INT8 object detection on the A311D NPU through the open-source VeriSilicon TIM-VX TensorFlow Lite delegate, compiled from source. To the author's knowledge, this is the first reported detection-accuracy result obtained through this open-source delegate path on this NPU, distinct from prior latency-only benchmarking conducted through the closed vendor software stack using INT16 arithmetic.

3. Produced the first heterogeneous edge benchmark, to the author's knowledge, of a detector trained on the traffic-domain BDD100K dataset, such that the reported accuracy indexes real driving-perception difficulties rather than the generic recognition that a general-purpose pretrained model would reflect.

4. Demonstrated empirically that rated throughput is a weak predictor of delivered performance. Despite its higher theoretical capacity (5.0 TOPS against the Edge TPU's 4.0 TOPS), the NPU proved to be the slowest and least energy-efficient platform, whereas the Edge TPU executed inference roughly 4.2 times faster and delivered 6.8 times the throughput per watt.

5. Documented a deployment boundary on the Maxwell TensorRT stack, where EfficientDet-Lite2 INT8 could not be converted to a TensorRT engine through any of four routes, with the model's custom post-processing operator being the common obstruction on the JetPack 4.6/TensorRT 8.0 stack evaluated here. Combined with Maxwell's lack of a hardware integer dot-product path, this marks a concrete and reproducible compatibility limit for this detector family on this GPU generation.

6. Isolated a quantizer-induced accuracy collapse on the FPGA's DPU, where per-tensor power-of-two INT8 quantization reduced mAP_{50} from 0.2400 to 0.0233. A

calibration ablation and a surgical bias bypass localized the cause to the quantization of the classification head's weights rather than the hardware or the model, isolating a known quantization sensitivity in a new model, toolchain, and hardware context.

7. Established a reproducible deployment path for the detector on the FPGA through the runtime-compiled ONNX route, after every host-side compiler path of the standard flow rejected the model, recording the non-standard bring-up so that the deployment, and hence the inclusion of this paradigm in the benchmark, can be reproduced.

1.4 Structure

The thesis continues in Chapter 2, which establishes the technical background, covering object detection, the EfficientDet architecture, integer quantization, and the four accelerator paradigms together with their toolchains. Chapter 3 sets out the study design, including the controlled-variable framework, the rationale for the model and the four platforms, and the evaluation methodology. Chapter 4 documents the implementation end to end, from dataset preparation and training through the four platform deployments and the uniform measurement harness. Chapter 5 reports the measurements and interprets each against the research questions. Finally, Chapter 6 concludes the work, discusses the findings in relation to prior work, and outlines directions for future research.

2 Background

This chapter establishes the technical foundations required to understand the benchmarking methodology and experimental design of this thesis. The central objective is to evaluate four heterogeneous edge-AI hardware platforms (Google Coral Dev Board Mini Edge TPU, Khadas VIM3 NPU, NVIDIA Jetson Nano GPU, and AMD Kria KV260 FPGA) for real-time traffic perception using the EfficientDet-Lite2 object detection model trained on the BDD100K dataset.

2.1 Hardware Accelerators for Edge AI

Edge-AI hardware for mobile robots and traffic perception operates under strict constraints on latency, power, and thermal dissipation; hardware accelerators address this by offloading compute-intensive operations to dedicated units optimized for a narrow task, achieving throughput and energy efficiency that General-Purpose Processor (GPP)s cannot match [24], [25], [26]. The use of accelerators introduces a fundamental trade-off between flexibility and efficiency: fixed-function designs such as ASICs and NPUs deliver the highest efficiency within their target domain, while programmable GPUs and reconfigurable FPGAs offer broader applicability at some cost in efficiency [24], [27].

This trade-off motivates the four-platform comparison in this thesis. Each platform occupies a different position along the flexibility-efficiency spectrum: the Coral Edge TPU and VIM3 NPU are fixed-function accelerators optimized for INT8 inference, the Jetson Nano's GPU offers programmability through Compute Unified Device Architec-

ture (CUDA), and the Kria KV260 FPGA provides reconfigurable acceleration via a soft DPU core. Figure 2.1 illustrates this positioning. Understanding these architectural differences is essential for interpreting the benchmarking results.

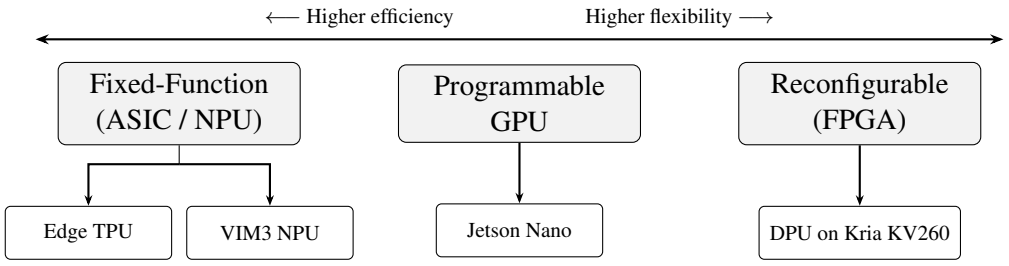


Figure 2.1: Taxonomy of hardware accelerators benchmarked in this thesis, positioned along the flexibility-efficiency spectrum. The Edge TPU Coral Dev Board Mini and A311D NPU (Khadas VIM3) are fixed-function ASIC/NPU accelerators; the Jetson Nano represents a programmable GPU, and the Kria KV260 provides reconfigurable FPGA acceleration. Adapted from [28].

2.1.1 GPU Architecture

Graphics Processing Units (GPUs) were originally developed to accelerate graphics rendering but have evolved into powerful parallel processors extensively employed in modern deep learning and autonomous systems. Programmable shader hardware and, from 2007, the CUDA programming model transformed GPUs into a General-Purpose GPU (GPGPU) platform, simplifying their use for non-graphics workloads [29], [30].

Modern GPUs achieve high throughput by organizing thousands of scalar cores into Streaming Multiprocessor (SM)s (NVIDIA) or Compute Units (AMD), each with dedicated shared memory, a register file, and an L1 cache. These GPUs exploit thread-level parallelism to hide memory latency rather than minimize it, dedicating more die area to arithmetic units than to control logic, a design well-suited to the dense matrix arithmetic of deep learning, where training datasets and model sizes necessitate sustained throughput [29], [31].

The NVIDIA Jetson Nano brings these architectural features into a compact SoC design, combining a quad-core ARM Cortex-A57 CPU with a 128-core Maxwell GPU and

4 GB of 64-bit LPDDR4 memory [32]. Since the CPU and GPU share a Unified Memory Architecture (UMA), data copy overhead is eliminated, reducing latency compared to discrete systems. In this thesis, the Jetson Nano serves as the programmable baseline against which the fixed-function and reconfigurable accelerators are compared. For the YOLOv4-416 workload used in this study, optimization via TensorRT (FP16) enables the Maxwell architecture to attain the throughput and deterministic response required for real-time traffic scenarios (see Section 2.6.3).

2.1.2 Domain-Specific Accelerators and ASICs

The end of single-threaded performance gains has made GPPs increasingly inefficient for workloads with high arithmetic intensity. Domain-Specific Architecture (DSA)s are specialized hardware units that address this by implementing highly optimized datapaths and memory architectures that enable significantly higher performance than GPPs [28]. ASICs sit at the far end of the specialization spectrum; these are chips designed for one specific task, delivering exceptional performance and energy efficiency at the cost of complete inflexibility. A well-known example is the Google TPU, purpose-built to accelerate neural network operations using a systolic array of multiply-accumulate units that propagate data between adjacent processing elements, minimizing memory bandwidth demand and maximizing arithmetic throughput [33]. The principal limitations of ASICs are their high upfront design costs and long development cycles [34].

Edge TPU and Neural Processing Units

Two of the four platforms benchmarked in this thesis use fixed-function neural accelerators that follow ASIC design principles, differing mainly in their supported operations, memory layout, and deployment toolchain.

The **Google Edge TPU** (Coral Dev Board Mini) is a purpose-built ASIC designed to run TensorFlow Lite models efficiently at the edge. It operates at a rated power of 2 W

with a peak performance of 4 TOPS (2 TOPS/W) [35]. A key architectural constraint is the requirement for full INT8 quantization; any operations using floating-point arithmetic fall back to the host CPU (quad-core ARM Cortex-A35), incurring a significant performance penalty. This constraint motivated the choice of EfficientDet-Lite2 which exports cleanly to INT8 with only the custom Non-Maximum Suppression (NMS) operator falling back to the host CPU (Section 2.4). The Coral Dev Board Mini also includes a MediaTek 8167s SoC with a quad-core ARM Cortex-A35 running at 1.3 GHz and 2 GB of LPDDR3 memory, making it the most resource-constrained platform in this benchmark.

The **Amlogic A311D NPU** (Khadas VIM3) is a heterogeneous SoC integrating a 5 TOPS neural processing unit alongside a hexa-core CPU (4× Cortex-A73 + 2× Cortex-A53) and an ARM Mali-G52 MP4 GPU [36]. The Khadas VIM3 development board houses this SoC, and its integrated NPU is referred to interchangeably as the A311D NPU (by chip) or VIM3 NPU (by board) throughout this thesis. The NPU handles INT8 and UINT8 quantized models through the Amlogic Neural Network (NN) SDK. In this study, EfficientDet-Lite2 is deployed via the TFLite runtime using the TIM-VX delegate, which directly interfaces with the VeriSilicon Vivante VIPNano-QI.7120 NPU. This deployment bypassed the standard Amlogic SDK, which lacks the required libraries, in favor of the TIM-VX backend built from source to ensure support for all required operations (see Section 2.6.2).

2.1.3 Reconfigurable Accelerators

Reconfigurable accelerators offer full control over hardware design and can be reprogrammed to implement different functionalities after manufacturing using a Hardware Description Language (HDL). The most widely used form is the FPGA, which contains a vast number of configurable logic gates [37].

FPGA Architecture

The fundamental distinction between FPGA-based acceleration and the GPU's Single-Instruction Multiple-Thread (SIMT) model is the shift from a control-flow to a spatial data-flow execution paradigm [38]. While GPUs hide latency through thread-level parallelism, FPGAs eliminate instruction-fetch overhead by mapping the algorithm's computational graph directly onto the hardware fabric.

An FPGA's architecture consists of a two-dimensional array of Configurable Logic Block (CLB)s, the fundamental building blocks. As shown in Figure 2.2, each CLB contains Look-Up Table (LUT)s that can implement any Boolean function of their inputs, flip-flops for state storage, and multiplexers for signal routing [39]. Modern FPGAs also include Digital Signal Processor (DSP) slices for efficient arithmetic, Block RAM (BRAM) for on-chip storage, high-speed serial transceivers, and hard processor cores in SoC FPGAs [40]. By configuring these blocks into specialized, deep pipelines, the FPGA can process data streams with deterministic and bounded latency, a critical property for real-time traffic perception workloads.

Modern SoC FPGAs, such as the AMD Kria KV260, use specialized soft IP cores known as DPUs. The DPU acts as a programmable overlay optimized for tensor operations, transforming the reconfigurable fabric into a DSA. This allows high-level models to be deployed via the Vitis AI stack, benefiting from custom hardware datapaths without manual Register-Transfer Level (RTL) design.

The Kria KV260

Among AMD's FPGA platforms, the Kria KV260 Vision AI Starter Kit is designed for vision and smart-camera applications such as smart-city, security, machine vision, and retail analytics, built on the K26 System on Module (SoM) employing the Zynq UltraScale+ MPSoC EV architecture. This heterogeneous SoC integrates an ARM Cortex-A53-based Application Processing Unit (APU), a dual-core ARM Cortex-R5F-based Real-time Pro-

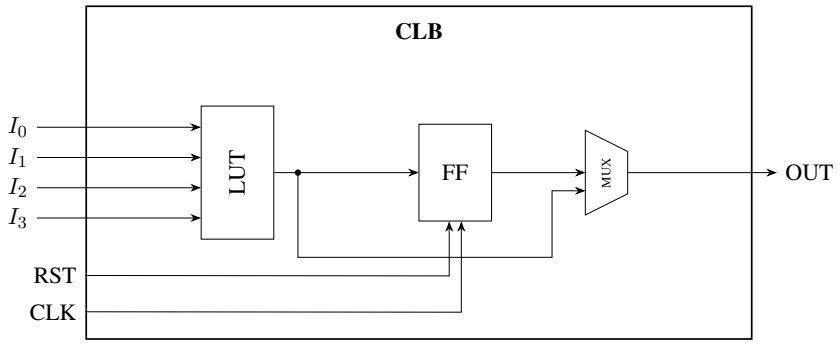


Figure 2.2: Simplified internal architecture of a CLB: the block employs a 4-input LUT ($I_0, \ldots, I_3$) to implement arbitrary combinational logic, which can be routed either directly to the output or through a Flip-Flop (FF) for edge-triggered storage. A multiplexer (MUX) determines the final output path, allowing the cell to function as either an asynchronous combinational element or a registered sequential component. Adapted from [39].

cessing Unit (RPU), a Platform Management Unit (PMU), an ARM Mali-400 MP2 GPU, and programmable logic fabric, all on a single chip [41]. Figure 2.3 shows a simplified block diagram of this architecture.

The board provides a vision-oriented I/O environment: one GbE RJ45 Ethernet port, four Universal Serial Bus (USB) 3.0/2.0 ports, one Pmod 12-pin connector, one Raspberry Pi camera interface, three MIPI sensor interfaces with an OnSemi API302 image signal processor, one DisplayPort 1.2a output, and one HDMI output [41].

In this thesis, the KV260's programmable logic is used to instantiate a DPU that executes EfficientDet-Lite2. The FPGA-based approach represents the most complex deployment path in this study, requiring model quantization and compilation into an `.xmodel` instruction binary via the Vitis AI compiler [42]. A notable architectural risk is operator coverage because the DPU is a fixed-instruction processor; thus, any layer not supported by the DPU instruction set must be offloaded to the APU, which can introduce latency bottlenecks.

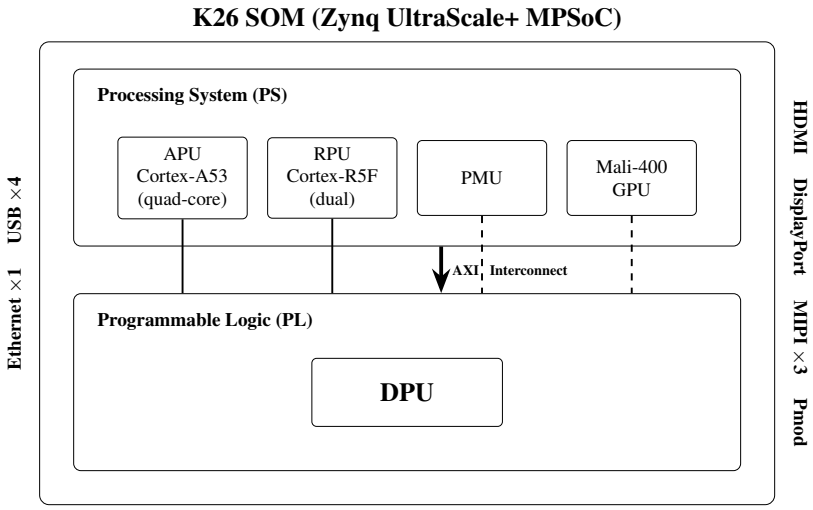


Figure 2.3: Architectural block diagram of the Kria KV260. The K26 SoM consists of a Processing System (PS) and Programmable Logic (PL). The PS contains the APU (Cortex-A53), RPU (Cortex-R5F), PMU, and Mali GPU. The PL contains the FPGA fabric on which the DPU is instantiated. Arrows from the APU and RPU represent active data paths to the PL via on-chip Advanced eXtensible Interface (AXI) interfaces; the dashed arrows from the PMU and Mali GPU represent the supervisory and graphics interface connections respectively. The DPU executes the compiled model (`.xmodel`) for accelerated inference. Adapted from [41].

2.2 Deep Learning Foundations

Deep learning has enabled significant advances in perception tasks that are central to autonomous systems. This section provides the theoretical foundations for the neural network architectures used in this thesis, with emphasis on the concepts needed to understand EfficientDet-Lite2's design choices.

2.2.1 Neural Networks

At its core, a NN is a computational model composed of interconnected units called neurons, organized in layers. A typical architecture consists of an input layer, one or more hidden layers, and an output layer. Depth influences the level of abstraction the network can model, while width affects information flow during forward propagation [43]. A single neuron computes a weighted sum (pre-activation z) followed by a non-linear activation function f:

$$z = \sum_{i=1}^{n} w_i x_i + b, \quad y = f(z) \tag{2.1}$$

where w_i are the weights, x_i the inputs, b a bias term. The function $f(z)$ represents a non-linear activation such as the Rectified Linear Unit (ReLU), which enables the network to learn complex, non-linear mappings. Together, these weights and biases form the model's learnable parameters, optimized during the training phase to minimize a defined objective function [44].

2.2.2 Training: Forward and Backward Propagation

Forward propagation passes inputs through the network layer by layer to produce predictions, and the discrepancy between predictions and ground truth is measured by a loss function $\mathcal{L}$. The learnable parameters are then optimized by backward propagation [45], which computes the gradient of the loss with respect to each parameter using the chain rule and updates them with a gradient-based optimizer such as SGD or Adam [44], [46].

2.2.3 Convolutional Neural Networks

While fully connected networks can in principle learn any function, they tend to perform poorly with high-dimensional inputs such as images. Convolutional Neural Network (CNN)s address this through local connectivity, weight sharing, and spatial pooling [47], dramatically reducing the parameter count. Stacking convolutional layers enables learning hierarchical features: edges in early layers, textures in middle layers, and object parts in deeper layers [48].

To ensure efficiency on resource-constrained edge hardware, modern architectures employ batch normalization, residual connections, and depthwise separable convolutions. The latter, introduced by MobileNet [49], factorizes a standard convolution into a depthwise convolution followed by a pointwise (1×1) convolution. This reduces cost to roughly

$\frac{1}{N} + \frac{1}{D_K^2}$ of a standard convolution, where N is the number of output channels and D_K is the kernel size. For a standard 3×3 kernel, this yields a $7\times$ to $9\times$ cost reduction depending on the channel depth. Depthwise separable convolutions are a core building block of the Efficientdet-Lite2 backbone used in this thesis, enabling deployment on edge accelerators while maintaining compatibility with INT8 quantization, as explored in Section 2.4.

2.3 Object Detection

Object detection is the computer-vision task of localizing and classifying multiple objects within an image, producing a set of bounding boxes, each associated with a class label and a confidence score. This task is fundamental to traffic perception: a self-driving or driver-assistance system must simultaneously detect and localize vehicles, pedestrians, and traffic signals in real time [50].

2.3.1 Two-stage vs. One-stage Detectors

Two-stage detectors first generate region proposals and then classify each proposal. Faster R-CNN exemplifies this with a Region Proposal Network (RPN), achieving high accuracy at a computational cost that often makes it unsuitable for edge deployment [51]. One-stage detectors directly predict bounding boxes and class probabilities in a single forward pass. YOLO and SSD pioneered this approach, trading some accuracy for significantly lower latency [52], [53]. EfficientDet, the model family used in this thesis, is a one-stage detector specifically designed to maximize detection accuracy under computational constraints [4].

2.3.2 Anchor-Based vs. Anchor-Free Detection

Anchor-based detectors predefine reference boxes (anchors) at various scales and aspect ratios, with the neural network learning to adjust its predictions based on these an-

chors [50]. Anchor-free detectors, such as CenterNet, predict bounding boxes directly as keypoints or distances while avoiding the anchor hyperparameter tuning that anchor-based methods require [54]. EfficientDet-Lite2 is an anchor-based detector offering a good balance between accuracy, simplicity, and compatibility with INT8 quantization. Its predictable memory access patterns and structure offer robust performance in power-constrained hardware accelerators like Edge TPU and Xilinx DPU.

2.3.3 Multi-Scale Feature Fusion

A detector trained for traffic perception must recognize objects whose apparent size varies enormously, from a distant traffic light occupying a small number of pixels to a bus filling most of the frame. A single convolutional feature map cannot represent this range well: deep layers carry strong semantic information but have a coarse spatial resolution, whereas shallow layers preserve fine spatial detail but have weak semantics. Multi-scale feature fusion addresses this by combining feature maps from several depths so that the prediction heads can reason about objects at every scale.

The Feature Pyramid Network (FPN) introduced the now-standard solution: a top-down pathway that propagates semantically rich features from the deepest levels back down to the higher-resolution shallow layers, joined by lateral connections [55]. PANet extended this with an additional bottom-up pathway, allowing strong localization features from shallow layers to propagate upward [56]. NAS-FPN later used neural architecture search to discover irregular cross-scale topologies, trading interpretability for further accuracy gains [57].

EfficientDet implements this with the Weighted Bi-directional Feature Pyramid Network (BiFPN) [4]. BiFPN makes three changes relative to the plain FPN. First, nodes that have only a single input edge are removed since they contribute little to feature fusion. Second, an extra skip connection is added between the original input and the output node at the same scale, fusing more features at little additional cost. Third, and most impor-

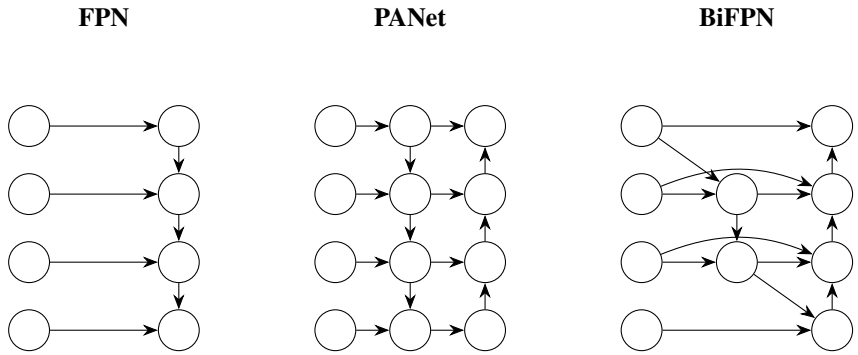


Figure 2.4: Cross-scale connectivity of three feature-fusion networks. FPN uses a single top-down pathway, PANet adds a bottom-up pathway, and BiFPN removes single-input nodes, adds same-scale skip connections (curved edges) and treats the bidirectional block as repeatable, while learning a weight for each input edge via Equation (2.2). Adapted from [4].

tantly, BiFPN treats each top-down and bottom-up pass as a repeatable block so that the whole bidirectional structure can be stacked several times to deepen feature fusion. The number of repeats is set by the compound scaling rule discussed in Section 2.4.

A further observation motivated the "weighted" part of the name. When feature maps of different resolutions are fused, most earlier networks simply add them together, implicitly assuming that each input contributes equally. Because the inputs originate at different scales, they do not, in general, contribute equally to the fused output. BiFPN therefore attaches a learnable weight to each input, normalized so that the weights sum to one. EfficientDet adopts the *fast normalized fusion* form, as shown below:

$$O \;=\; \sum_{i} \frac{w_i}{\varepsilon + \sum_j w_j} \;\cdot\; I_i, \tag{2.2}$$

where I_i is the i-th input feature map, $w_i \geq 0$ is its learnable fusion weight (kept non-negative by a ReLU applied after each update), and $\varepsilon = 10^{-4}$ is a small constant added for numerical stability. The normalized weight $w_i/(\varepsilon + \sum_j w_j)$ lies in $[0, 1]$ and expresses how much each scale is trusted at that fusion node. This weighting is computationally far cheaper than the softmax-based alternative while achieving comparable accuracy [4]. Figure 2.4 contrasts the connectivity of FPN, PANet, and BiFPN.

2.4 The EfficientDet-Lite2 Object Detector

The preceding sections introduced the building blocks (convolutional backbones, one-stage anchor-based detection, and multi-scale fusion) on which the detector used throughout this thesis is built. This section assembles those blocks into its primary detector, EfficientDet-Lite2, and explains the design choices that make it a natural fit for heterogeneous edge accelerators. The intent is to hold a single model constant so that differences in the measured results can be attributed to the hardware and its toolchain rather than to a change of architecture. This is achieved on three of the four platforms; the Coral Edge TPU, the VIM3 NPU, and the Kria KV260 DPU, all of which run the same EfficientDet-Lite2 network. The Jetson Nano is a deliberate and documented exception: as Section 2.6 details, none of the available conversion paths could bring EfficientDet-Lite2 through the Jetson's TensorRT toolchain, so a YOLOv4-416 network was deployed in its place. Therefore, the Jetson accuracy figure is not directly comparable to the other three platforms.

EfficientDet is a family of one-stage detectors that pairs an EfficientNet backbone with a BiFPN feature network and shared prediction heads [4]. EfficientDet-Lite is a mobile and IoT-oriented derivative released by the TensorFlow team, comprising five variants, Lite0 through Lite4, that trade accuracy for latency and model size [58]. The Lite series was created by adapting the original architecture so that it exports cleanly to a fully integer TensorFlow Lite model, which is the property required by the fixed-function accelerators in this study.

2.4.1 EfficientNet-Lite Backbone

The backbone of EfficientDet-Lite2 is EfficientNet-Lite2, a feature extractor built from Mobile Inverted Bottleneck Convolution (MBConv) blocks that rely on the depthwise separable convolution described in Section 2.2.3. Compared with the original Efficient-

Net, the Lite backbone applies three modifications aimed at edge deployment [58], [59]: the squeeze-and-excitation blocks are removed because they are poorly supported by several mobile accelerators; every Swish activation is replaced with ReLU6; and the stem and head are held fixed while the network is scaled, keeping the scaled models small. EfficientNet-Lite2 contains roughly 6.1 million backbone parameters [59].

Swish is a smooth, unbounded activation whose output range is hard to capture with a single fixed-point scale, whereas ReLU6 clips activations to the bounded interval $[0, 6]$, which maps far more cleanly onto an 8-bit integer grid and significantly improves the quality of Post-Training Quantization (PTQ) [59]. Hence, the replacement of Swish by ReLU6 is a quantization decision, not just an operator support convenience. This choice is one of the main reasons EfficientDet-Lite2 survives full INT8 quantization on most platforms, and it is directly relevant to the quantization behavior analyzed in Section 2.5.

2.4.2 BiFPN and Prediction Heads

Features extracted by the backbone at several scales are fused by the BiFPN of Section 2.3.3, and the fused multi-scale features are then passed to two small, fully convolutional prediction heads that are shared across all pyramid levels. The class head predicts, for every anchor at every spatial location, the probability that an object of each class is present, and the box head regresses the offsets from each anchor to the nearest ground-truth box. Because both heads are shared across scales, their parameter cost is modest and independent of the number of pyramid levels.

EfficientDet-Lite2 is an anchor-based detector. Each spatial location on each pyramid level is associated with a set of reference boxes (anchors) of predefined scales and aspect ratios, and the box head predicts a correction relative to the matched anchor rather than absolute coordinates. The decoded prediction is therefore the anchor displaced and rescaled by the regressed offsets. Recovering the final boxes at inference time requires applying these offsets to the known anchor grid and then suppressing duplicate detections

with NMS. In the TensorFlow Lite export used here, the NMS stage is emitted as a single custom operator. This single operator turns out to be consequential on more than one platform: on the fixed-function accelerators, it falls back to the host CPU, and on the Jetson Nano it cannot be parsed by the TensorRT toolchain at all, which is the root of the conversion failure discussed in Section 2.6.

2.4.3 Focal Loss and the Classification-head Bias

One-stage detectors evaluate the class head densely, at tens of thousands of anchors per image, the overwhelming majority of which correspond to the background. This extreme foreground-background imbalance is the central difficulty that prevented early one-stage detectors from matching two-stage accuracy. Focal loss [60] resolves it by reshaping the standard cross-entropy so that easy, well-classified examples are down-weighted and training focuses on the sparse set of hard examples:

$$\mathrm{FL}(p_t) \;=\; -\,\alpha_t\,(1-p_t)^{\gamma}\,\log(p_t), \qquad p_t = \begin{cases} p & \text{if the anchor is foreground,} \\ 1-p & \text{otherwise,} \end{cases} \tag{2.3}$$

where p is the model's estimated probability for the class, $\gamma \geq 0$ is the focusing parameter (typically $\gamma = 2$) that controls how strongly easy examples are suppressed, and α_t is a class-balancing weight (typically $\alpha = 0.25$). EfficientDet inherits this loss from its RetinaNet-style detection head.

Focal loss carries an initialization detail that is decisive for the present work. To stop the vast number of background anchors from generating a large, destabilizing loss in the first training iterations, the bias of the final convolution in the class head is initialized so that every anchor is predicted as foreground with only a small prior probability π [60]:

$$b \;=\; -\log\!\left(\frac{1-\pi}{\pi}\right), \qquad \pi = 0.01 \;\Rightarrow\; b \approx -4.6. \tag{2.4}$$

The classification logit produced by the head is therefore offset by a large negative baseline of about -4.6 per output channel, and a confident "object" prediction is expressed as a comparatively small positive deviation that lifts the sigmoid output above this $\pi \approx 0.01$ floor. After training, the class logits that distinguish an object from the background occupy a narrow band sitting on top of a strongly negative bias. This characteristic of the head was, early in the present work, suspected of causing the integer quantization failure observed on the Kria DPU. As Section 2.5.5 explains, the bias is real and was measured precisely (its mean matches Equation (2.4) to two decimal places), but a controlled bias-correction experiment demonstrated it is *not* the dominant cause of the accuracy collapse. The relevant interaction is instead between the per-tensor quantizer of that hardware and the depthwise-separable structure of the head, which is examined in Section 2.5.5 and analyzed empirically in Chapter 5.

2.4.4 Compound Scaling and the Lite2 Configuration

EfficientDet scales the input resolution together with the depth and width of the backbone, the BiFPN, and the prediction heads through a single compound coefficient, instead of scaling components independently [4]. Lite2 is the third step on this scaling ladder: it operates on a 448×448 input, uses an EfficientNet-Lite2 backbone, and reaches a reference COCO-mAP of about 34% while remaining small enough for edge deployment. Table 2.1 situates Lite2 within the full Lite family and quantifies the trade-off that motivates the choice.

Lite2 was selected for this thesis as the accuracy-cost sweet spot of the family. The smaller Lite0 and Lite1 variants reduce latency but sacrifice the detection of small, distant traffic objects that dominate the BDD100K benchmark, while the larger Lite3 and Lite4 variants exceed the memory and compute budget of the most constrained platform

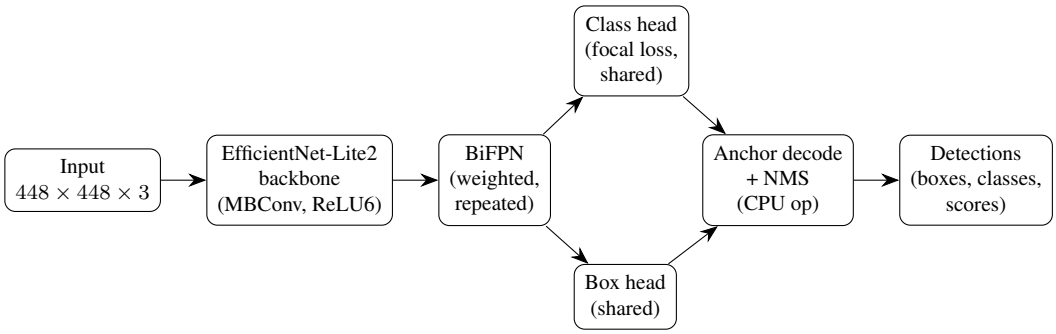


Figure 2.5: Simplified EfficientDet-Lite2 pipeline architecture as deployed in this thesis. A $448 \times 448 \times 3$ input is processed via an EfficientNet-Lite2 backbone (MBConv, ReLU6), and multi-scale features are fused using a weighted, repeated BiFPN. Symmetrical, shared convolutional heads then extract predictions; the class head incorporates focal loss and the negative bias of Equation (2.4). Because fixed-function edge hardware typically lacks acceleration for post-processing layers, anchor decoding and NMS are executed on the host CPU before emitting the final bounding boxes, classes and confidence scores.

considered here. Equally important, EfficientDet-Lite2 exports as a fully INT8-quantized TFLite model in which only the custom NMS operator falls back to the host CPU (see Section 2.4.2).

Alternative mobile detectors were considered and rejected during the model selection. SSD-MobileNet V2 could not be used because the `tflite-model-maker` training pipeline supports only the EfficientDet-Lite family and not SSD-MobileNet, and SSD-MobileNet additionally reports lower detection accuracy (around 22% versus EfficientDet-Lite2's $\sim 34\%$ COCO-mAP). The MediaPipe Model Maker path was likewise constrained to the same supported architectures. EfficientDet-Lite2 was therefore the only candidate that combined edge-oriented design, native full-INT8 TFLite export, and a supported fine-tuning pipeline. The complete architecture, comprising backbone, BiFPN, and shared heads, is summarized in Figure 2.5.

The table makes the trade-off quantitative. Across the family, mAP rises monotonically from roughly 26% to 42%, but the integer model grows more than fourfold and the reference latency by almost seven times. Lite2 is the first point on this ladder to clear the $\sim 34\%$ mAP mark while keeping the integer model near 7 MB, which is why it was adopted as the common detector for the three platforms that run EfficientDet-Lite2.

Table 2.1: The EfficientDet-Lite family, showing the accuracy-cost trade-off that motivates the choice of Lite2 (highlighted). Input resolution follows the official EfficientDet-Lite specification; the integer-model size, reference CPU latency and COCO-2017 validation mAP are the published TensorFlow Lite Model Maker figures [58], [61]. The latency column is a single mobile-CPU reference (Pixel 4, four threads) intended only for relative comparison across variants; it is *not* measured on the accelerators benchmarked in this thesis, whose results appear in Chapter 5.

Variant	Input	INT8 size (MB)	CPU latency (ms)	COCO mAP (%)
Lite0	320×320	4.4	37	25.69
Lite1	384×384	5.8	49	30.55
Lite2	$\mathbf{448 \times 448}$	**7.2**	**69**	**33.97**
Lite3	512×512	11.4	116	37.70
Lite4	640×640	19.9	260	41.96

2.5 Model Optimization and Quantization

Deploying a detector on a resource-constrained accelerator requires reducing the numerical precision of its weights and activations from FP32 to more efficient formats. Every platform in this thesis operates in such a reduced-precision regime: The Edge TPU and the A311D NPU *require* full INT8 quantization, the Kria DPU is an integer-only processor [62], and the Jetson GPU uses FP16 to maximize throughput. Reduced-precision inference is therefore not an optional optimization but the common deployment substrate of this study, and the specific quantization scheme enforced by each toolchain ultimately explains several of the experimental outcomes. This section establishes the theory, and Chapter 5 reports the consequences.

2.5.1 Why Quantize

An FP32 format provides far more dynamic range and precision than the weights and activations of a trained network inherently require. At any given layer, the values typically span a narrow range that INT8 and a single scaling factor can represent accurately [62]. Replacing FP32 with INT8 yields a roughly fourfold reduction in model size and memory traffic and allows arithmetic to run on dense integer units. For images, videos, and other sampled-data workloads, the degradation on the predictive accuracy is typically nominal,

often under one percent [62]. Though as this thesis demonstrates, the loss can be severe when a model's value distribution is misaligned with the quantizer.

2.5.2 Uniform Affine Quantization

The dominant scheme is uniform affine quantization, which maps a real value r to an integer q through a scale $S > 0$ and an integer zero-point Z [19]:

$$q \;=\; \mathrm{clip}\big(\mathrm{round}\big(\tfrac{r}{S}\big) + Z,\; q_{\min},\; q_{\max}\big)\,, \qquad \hat{r} \;=\; S\,(q - Z), \tag{2.5}$$

where $\hat{r}$ is the value recovered after quantization, and $[q_{\min}, q_{\max}]$ is the integer range, e.g. $[-128, 127]$ for signed INT8. The scale S sets the spacing of the integer grid, and the zero-point Z aligns the integer 0 with the real 0. Two regimes are distinguished by the zero-point. In *asymmetric* quantization, Z may be non-zero, so an arbitrary real range $[r_{\min}, r_{\max}]$ can be mapped onto the full integer range; this is well suited to the non-negative output of a ReLU activation. In *symmetric* quantization, $Z = 0$ and the range is forced to be symmetric about zero, which is the natural choice for signed weight tensors and is the scheme used by integer accelerators such as the DPU.

2.5.3 Quantization Granularity

A second design axis is granularity, the level at which a separate (S, Z) pair is chosen. *Per-tensor* quantization uses one scale for an entire weight or activation tensor, whereas *per-channel* (per-axis) quantization assigns a separate scale to each output channel of a convolution, so channels whose weights span very different magnitudes each receive an appropriately sized grid, and small-magnitude channels are not swamped by the largest ones. This distinction matters for the present work because the granularity supported by each toolchain differs, and a model whose accuracy depends on per-channel resolution can collapse when forced onto a per-tensor grid.

2.5.4 PTQ, Calibration and QAT

Quantization parameters can be obtained in two ways. PTQ takes an already-trained float model and estimates the activation ranges by forwarding a small, unlabeled *calibration* set, typically a few hundred representative images, through the network [62]. It is fast and requires no retraining and is the path used for every platform in this thesis. Quantization-Aware Training (QAT) instead simulates quantization during training so the weights adapt to the integer grid; it usually recovers more accuracy but requires a full training pipeline and labeled data, and is the recommended fallback when PTQ loses more than one to two percent of accuracy [63]. The choice of calibration set size and the clipping method used to set activation ranges (for example minimizing mean-squared error, or a non-overflow criterion) materially affect the result, and these settings are reported with the experiments in Chapter 5.

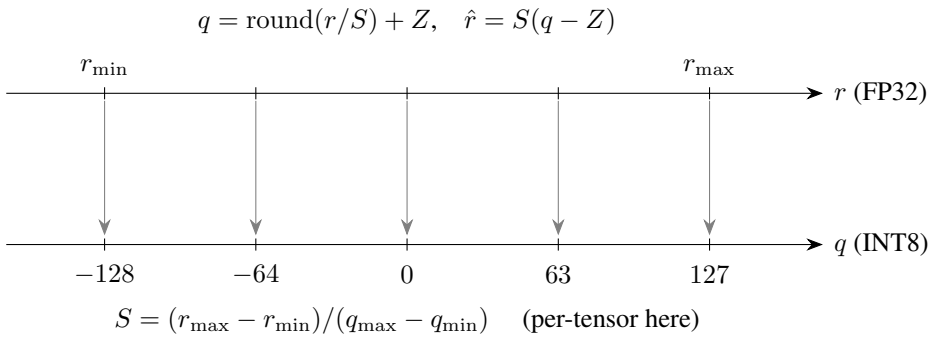


Figure 2.6: Uniform affine quantization maps a continuous FP32 range onto a finite INT8 grid via Equation (2.5). The scale S fixes the grid spacing and the zero-point Z aligns integer and real zero. A single scale per tensor (per-tensor quantization) must cover the entire range of the tensor.

2.5.5 Hardware-Specific Schemes and the DPU Constraint

The accelerators in this thesis do not all implement the same quantizer, and the differences are exactly where the cross-platform results diverge. The TensorFlow Lite full-integer scheme used by the Edge TPU and the A311D NPU employs per-channel symmetric quantization for convolution weights and per-tensor asymmetric quantization for

activations, with arbitrary floating-point scales [64]. The Vitis AI DPU, by contrast, is a more constrained integer processor. For the DPU target, the AMD Quark quantizer currently does not support per-channel quantization, so weights must be quantized per tensor, and its *power-of-two* mode constrains every scale to the form $S = 2^{-f}$, so that the runtime can replace a multiplication by a hardware bit shift [65]. This per-tensor, power-of-two scheme was introduced to make fixed-point inference amenable to such hardware, and with trained quantization thresholds, it attains near-floating-point accuracy even on depthwise-separable networks such as MobileNet [66]. That accuracy depends on QAT; under the post-training calibration used for every platform in this thesis, the weights are never retrained, so they have no opportunity to adapt to the coarse grid.

This combination of per-tensor granularity together with power-of-two scales is efficient but coarse (Figure 2.7).

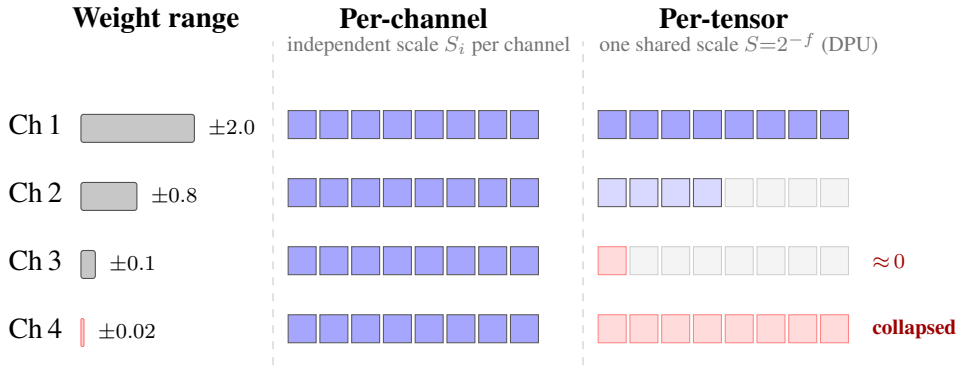


Figure 2.7: Per-channel versus per-tensor quantization applied to a depthwise convolutional layer. The left panel (Weight range) shows the weight magnitude of four illustrative output channels spanning two orders of magnitude (from ± 2.0 to ± 0.02), representative of the per-channel variation observed in the depthwise-separable layers of EfficientDet-Lite2. Blue squares denote usable INT8 quantization levels, light-blue squares partially-used levels, grey squares wasted grid steps, and red squares levels that collapse to zero. Under per-channel quantization (center panel) each channel receives its own scale S_i fitted to its weight range, preserving full INT8 resolution for all channels. Under per-tensor quantization (right panel) the Vitis AI Quark quantizer enforces a single power-of-two scale $S{=}2^{-f}$ fitted to the largest channel; channels with small weights (Ch 3, Ch 4) obtain at most one representable level, so their activations collapse to zero or near-zero. This channel collapse is the dominant cause of the accuracy degradation reported in Chapter 5.

It is poorly matched to the depthwise-separable structure that pervades EfficientDet-Lite2, including its classification head. Depthwise convolutions apply one filter per input channel, and the per-channel weight statistics of such layers vary enormously since a

single channel can span a dynamic range many times larger than its neighbors. A per-channel quantizer assigns each channel its own scale and absorbs this variation, which is the standard remedy; a per-tensor quantizer must cover every channel with one scale, and a power-of-two constraint coarsens it further. Under such a scale, the small-magnitude channels are rounded away, and a depthwise layer can collapse an entire output channel to zero. The standard references on integer quantization identify per-tensor quantization of depthwise-separable and EfficientNet-style networks as the case where accuracy loss becomes pronounced or catastrophic [20], [63], and severe PTQ degradation is established in the detection literature [22].

Existing literature indicates that this degradation is architectural rather than an inherent limitation of the DPU or of focal loss. A prior cross-platform study deployed a focal-loss detector (RetinaNet with a ResNet-50 backbone, which utilizes standard dense convolutions) on the same class of DPU alongside a GPU and an Edge TPU, reporting essentially equal accuracy across all three devices with no collapse [13]. The contrast isolates the depthwise-separable head, rather than the accelerator or the loss function, as the sensitive element. The focal-loss bias of Section 2.4.3 was correctly identified as a potential factor; however, executing a bias-bypass experiment in this study yielded only marginal accuracy improvements. The same insufficiency of bias-level corrections appears in the quantization literature, where post-quantization bias correction alone does not restore floating-point accuracy on per-tensor-quantized depthwise networks [21].

The standard remedies, per-channel quantization and QAT, are therefore the appropriate fixes in principle, but neither is available within the scope of this work. Per-channel quantization is unsupported by this DPU's toolchain [65], and QAT requires retraining the detector with simulated quantization, which falls outside the thesis scope. This thesis empirically isolates the failure of EfficientDet-Lite2 on the DPUCZDX8G and identifies QAT as the supported remedy. The methodology used to report KV260 accuracy under quantization collapse is given in Chapter 5.

2.6 Deployment Toolchains and Runtime

Each platform deploys the detector through a distinct software stack, and because this thesis isolates the hardware by keeping the model constant whenever possible, the toolchain constitutes part of the experimental apparatus. This section details how the trained model is converted, quantized, and executed on each platform, and highlights the toolchain decisions that directly influence the empirical results. The two fixed-function accelerators share the same INT8 TFLite EfficientDet-Lite2 model through different runtimes, while the FPGA recompiles that same network into a hardware instruction binary. The GPU represents the exception, where the EfficientDet-Lite2 toolchain failed on this architecture, necessitating a substitute model, as detailed below.

2.6.1 Edge TPU Compiler and PyCoral (Coral Dev Board Mini)

The Coral Dev Board Mini executes the model on a Google Edge TPU, a fixed-function ASIC that executes only fully INT8-quantized TFLite operators. Deployment has two stages. The model is first quantized to full integer through TensorFlow Lite PTQ, then passed to the Edge TPU compiler, which maps the supported operators onto the accelerator and delegates the remaining to the host CPU [67], [68]. Any operator that the Edge TPU cannot execute, here the custom NMS post-processing operator, is delegated to the quad-core Cortex-A35 host. Since all subsequent operators also fall back to the host, the placement of the first unsupported operator becomes a significant performance consideration. At runtime, inference is driven through the PyCoral API, a thin Python layer over the Edge TPU runtime. Output tensors must be explicitly copied from the runtime's internal buffers before further processing, as the underlying memory is not guaranteed to persist. The Edge TPU's strict INT8 requirement is the reason EfficientDet-Lite2, which quantizes cleanly thanks to its ReLU6 activations, was a deliberate architectural choice.

2.6.2 TIM-VX Delegate (Khadas VIM3)

The Khadas VIM3 carries an Amlogic A311D with a VeriSilicon Vivante NPU. The vendor's standard Amlogic SDK lacked the libraries required for the operators in EfficientDet-Lite2, so deployment in this thesis instead uses TIM-VX, VeriSilicon's tensor interface module [69], which is exposed to TensorFlow Lite as an external delegate. The delegate (`libvx_delegate.so`, built against `libtim-vx.so`) was compiled from source to obtain support for the required operators and delivered the quantized TFLite graph directly to the NPU, bypassing the Amlogic stack. To the best of the author's knowledge, no peer-reviewed benchmarking study names the TIM-VX delegate as the deployment path for this class of NPU; the VIM3 results here therefore represent the first empirical characterization of EfficientDet-Lite2 INT8 on the Amlogic A311D NPU through this delegate, a point developed in Chapter 5. Despite a nominal rating of 5 TOPS against the Edge TPU's 4 TOPS [35], [36], the NPU proves markedly slower on this model.

2.6.3 TensorRT (NVIDIA Jetson Nano)

The Jetson Nano serves as the programmable GPU baseline. The deployed model is optimized with NVIDIA TensorRT, which treats inference as a compilation problem: it parses the network, fuses compatible layers into single CUDA kernels, selects precision (here FP16), auto-tunes the kernel implementation for the target Maxwell GPU, manages tensor memory dynamically, and serializes the result into an optimized engine that is loaded at runtime [70]. The original intent was to run the same EfficientDet-Lite2 model on the Jetson, but every conversion path from the INT8 TFLite model to a TensorRT engine failed on the stack installed on the board, the JetPack 4.6 (L4T 32.6.1), which provides TensorRT 8.0.1. The custom NMS operator of Section 2.4 is not understood by TensorRT, and the intermediate ONNX routes were each blocked by an operator or attribute that this TensorRT build does not accept. Independently of the conversion question, the Maxwell generation provides no fast integer dot-product instruction, which

was introduced only from the Pascal generation, so an INT8 engine would not have been hardware-accelerated on this GPU even had it built, which is why an FP16 substitute represents the GPU paradigm. A YOLOv4-416 network was therefore deployed in TensorRT FP16 as a substitute. FP16 retains far more dynamic range than INT8 and avoids the quantization-induced degradation discussed in Section 2.5, at the cost of running in 16- rather than 8-bit arithmetic. Because this is a different model on a different dataset, the Jetson accuracy figure is reported separately and is not placed on the same accuracy axis as the other three platforms; the systematic conversion-failure analysis that led to this decision is presented in Chapter 5.

2.6.4 Vitis AI: Quantizer, Compiler and VART (Kria KV260)

The Kria KV260 demands the most involved deployment path. Its programmable logic is configured with a DPU, a soft instruction-set overlay for tensor operations, and the model must be recompiled into a DPU instruction binary rather than merely converted. AMD's Vitis AI provides this flow through three stages [42]: the *Optimizer* optionally prunes the network; the *Quantizer* converts it from floating point to INT8 using a calibration set; and the *Compiler* fuses operators (for example, convolution, batch normalization, and ReLU) and emits a deployable `.xmodel` for the DPU. At runtime, Vitis AI Runtime (VART), layered on the Xilinx Runtime (XRT), provides the C++/Python interface that loads the `.xmodel` and schedules inference on the DPU, while any operator that the DPU does not support is offloaded to the ARM APU. Figure 2.8 shows the Vitis AI environment. As established in Section 2.5.5, the DPU's per-tensor, power-of-two quantizer is the defining element of this stack and the origin of the accuracy behavior examined in Chapter 5.

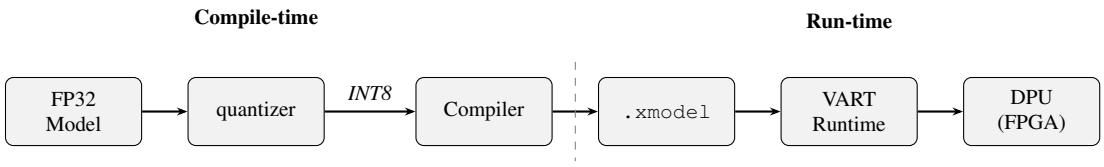


Figure 2.8: Simplified Vitis AI toolchain for the Kria KV260. At compile time, a floating-point model is quantized to INT8 by the Vitis AI Quantizer and compiled into a deployable `.xmodel` by the Vitis AI Compiler. At run time, VART loads the `.xmodel` and schedules execution on the DPU implemented in the FPGA PL. Adapted from [42].

2.7 The BDD100K Dataset

The model is trained and evaluated on BDD100K, a large and deliberately diverse driving dataset released by Yu et al. [5]. It comprises 100,000 driving video clips of forty seconds each, collected from more than 50,000 rides across several regions and a wide spread of weather conditions, times of day, and scene types, from city streets to highways. A single key-frame, the frame at the tenth second of each clip, is annotated for the image tasks, and the dataset is split into 70,000 training, 10,000 validation, and 20,000 test images [5]. For object detection, the key-frames carry bounding-box annotations across ten road-object categories, including car, truck, bus, person, rider, traffic light, traffic sign, bicycle, motorcycle, and train.

BDD100K was chosen for this thesis for two reasons. First, its geographic, environmental, and weather diversity makes it a far more demanding and realistic benchmark for traffic perception than curated datasets recorded under uniform conditions. This is the operating environment that an edge accelerator deployed in roadside or in-vehicle infrastructure must withstand. Second, to the author's knowledge, no prior peer-reviewed cross-accelerator benchmarking study evaluates this class of edge hardware on a model trained on BDD100K; most comparable works use COCO- or ImageNet-pretrained models. The use of a BDD100K-trained, traffic-domain model across heterogeneous accelerators is therefore one of the distinguishing methodological choices of this work.

In this work, the detection problem is reduced to seven traffic-relevant classes: *car*, *truck*, *pedestrian*, *t_sign* (traffic sign), and the traffic-light state split into *tl_red*, *tl_green*,

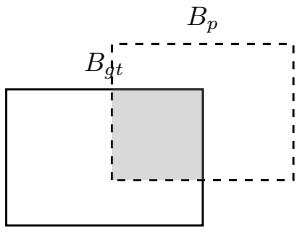
and *tl_yellow*. This mapping merges the native person and rider categories into a single pedestrian class, drops four non-traffic-critical categories (bus, train, bicycle, and motorcycle), and expands the single traffic-light category into three color states, reflecting the signal-state discrimination requirements of traffic perception. The detector is fine-tuned on a sampled training subset of $33,288$ key-frames and evaluated on a fixed validation subset of $4,993$ key-frames, held identical across the three platforms that run EfficientDet-Lite2 (Coral, VIM3, and KV260), so that the comparison reflects hardware and toolchain behavior alone. The class definition also exposes a dataset property that recurs throughout the results: the *tl_yellow* class is severely under-represented, and its Average Precision (AP) is near zero identically on architecturally different accelerators. This behavior isolates the cause as class imbalance in the dataset rather than any hardware or quantization artifact. The precise class mapping and evaluation protocol are reported with the experimental configuration in Chapter 5.

2.8 Benchmarking Metrics

A foundational premise of this thesis is that no single number captures the suitability of an edge accelerator for real-time traffic perception. A platform that is fast but inaccurate is unsuitable for safety-critical deployment, while an accurate but power-hungry platform is unsuitable for thermally or cost-constrained infrastructure. The platforms are therefore evaluated along three primary axes: detection accuracy, inference performance, and energy efficiency, as defined in the following subsections.

2.8.1 Detection Accuracy

Detection fidelity is measured with mAP, the standard object-detection metric. Its foundation is the Intersection over Union (IoU) between a predicted bounding box B_p and a ground-truth bounding box B_{gt}:



$$\text{IoU} = \frac{|B_p \cap B_{gt}|}{|B_p \cup B_{gt}|}$$

Figure 2.9: IoU, as defined in Equation (2.6). The ground-truth bounding box B_{gt} is represented with a solid outline and the predicted bounding box B_p with a dashed outline. The shaded region is their intersection; IoU is this area divided by the area of their union. A predicted box is counted as a TP when $\text{IoU} \geq 0.50$, following the single-threshold protocol of the VOC benchmark [72].

$$\text{IoU} \;=\; \frac{\text{area}(B_p \cap B_{gt})}{\text{area}(B_p \cup B_{gt})}\,. \tag{2.6}$$

A detection is counted as a True Positive (TP) if its IoU with a ground-truth box of the correct class exceeds a threshold, and as a False Positive (FP) otherwise. A ground-truth object with no matching detection is a False Negative (FN). From these, precision and recall follow:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \qquad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \tag{2.7}$$

The precision-recall curve for each class is obtained by evaluating precision and recall across the full range of confidence thresholds. AP is the area under that curve, and mAP is the AP averaged over all classes.

The COCO benchmark protocol averages mAP over ten IoU thresholds from 0.50 to 0.95 in steps of 0.05, and additionally reports mAP_{50} and mAP_{75} as secondary metrics [71]. This thesis reports mAP_{50} throughout, which counts a detection as correct whenever $\text{IoU} \geq 0.50$, following the single-threshold PASCAL Visual Object Classes (VOC) convention [72] adopted widely in the cross-platform benchmarking literature.

mAP_{50} is reported alongside the per-class AP breakdown because the per-class view reveals how quantization and platform differences affect small or rare traffic classes differently from large, common ones. A concrete instance of this is the *tl_yellow* class, examined in Section 2.7.

2.8.2 Inference Performance

Inference performance is reported as latency and throughput, distinguishing two latency figures because they answer different questions. *Invoke latency* is the time the accelerator spends executing the model's inference call alone, and it characterizes the raw hardware. *End-to-end latency* additionally includes image pre-processing, host-device transfer, and post-processing (anchor decoding and NMS), and it characterizes the deployable pipeline. Both are reported in milliseconds. Throughput is reported as Frames Per Second (FPS),

$$\mathrm{FPS} \;=\; \frac{1000}{t_{\text{latency}}\,[\mathrm{ms}]}\,. \tag{2.8}$$

evaluated on the end-to-end figure for the realistic case and on the invoke figure where the raw accelerator rate is of interest. Reporting both latencies prevents a fast accelerator with an expensive CPU-side post-processing stage (for example one that runs NMS on the host) from appearing better than it is in practice.

2.8.3 Power and Energy Efficiency

For battery-powered or thermally limited edge deployment, the energy spent per inference is as consequential as speed. Power is reported in watts and energy per inference in joules. Voltage and current are never reported in isolation. Average power P is measured at the board input during sustained inference, and the energy per inference is calculated as follows:

$$E \;=\; P \times t_{\text{inference}}, \tag{2.9}$$

with P in watts and $t_{\text{inference}}$ in seconds, giving E in joules. Energy per inference is a more informative efficiency metric than instantaneous power because a platform that draws more power but finishes inference sooner may still be the more energy-efficient choice overall. Power is measured with an external instrument or on-board sensor appropriate to

each platform, which covers total system draw and not only the accelerator alone.

2.8.4 Price-to-Performance

Finally, because deployment decisions are made under a budget, the platforms are also compared on a price-to-performance basis, relating the device's purchase cost to its achieved throughput and energy efficiency. This metric is reported as a summary table in Chapter 5 and places the technical measurements in the practical context of selecting hardware for traffic-perception infrastructure.

2.9 Summary

This chapter established the technical foundations for the experimental work that follows. It positioned the four accelerators along the flexibility-efficiency spectrum, introduced the deep-learning and object-detection concepts underpinning the EfficientDet-Lite2 detector, and detailed that model's backbone, BiFPN, and focal-loss classification heads. It then developed the theory of INT8 quantization and explained why the Kria DPU's mandatory per-tensor, power-of-two scheme is poorly matched to the depthwise-separable structure of EfficientDet-Lite2. This is a known quantization failure mode; its standard remedies, per-channel quantization and QAT, fall outside the toolchain constraints of this work. However, the cause is isolated and the collapse is quantified in Chapter 5. The platform-specific toolchains, including the documented substitution of YOLOv4-416 on the Jetson Nano, the BDD100K dataset, and the benchmarking methodology covering detection accuracy, inference performance, energy efficiency, and price-to-performance, complete the experimental apparatus. The next chapter describes the study design, including the controlled-variable framework and evaluation methodology.

3 Design

Chapter 2 established the technical foundations of this study, covering each accelerator paradigm, the detection model, the quantization schemes, and the dataset. This chapter sets out the study's design. It explains how those pieces are assembled into a single controlled experiment and why each design decision was made. The execution details, including the platform-specific toolchains, the deployment steps, and the problems resolved along the way, are deferred to Chapter 4.

3.1 Design Overview

The study addresses a single question: among four fundamentally different ways of accelerating a neural network, which is best suited for real-time traffic perception at the edge when speed, accuracy, power, and cost are weighed together? The detection task, the trained model, the evaluation data, the ground truth, and the way every metric is computed are therefore held constant across all four boards. The hardware is the only factor that changes between platforms.

This controlled approach follows the methodology established for cross-platform edge benchmarking, in which one shared workload is run across heterogeneous devices under a single measurement protocol [7]. Section 1.2 sets out how this study differs from prior work; the present chapter describes the design that implements it.

Figure 3.1 shows the resulting framework: one model trained once, deployed through four native toolchains, and scored under a single evaluation protocol. The sections that

follow take each element in turn. Section 3.2 covers the detection model, Section 3.3 the dataset and task, Section 3.4 the four hardware platforms, and Section 3.5 the evaluation methodology. Section 3.6 then sets out two points where the shared-model ideal could not be fully met.

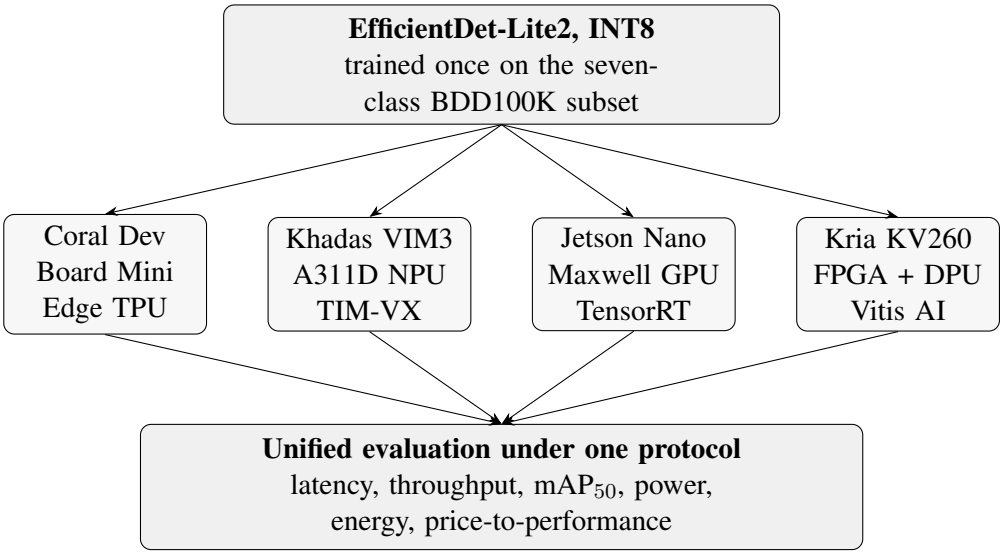


Figure 3.1: Design of the benchmarking framework. A single EfficientDet-Lite2 model, quantized to INT8 precision and trained once on the seven-class subset of BDD100K, is deployed to four hardware platforms through their respective native toolchains: the Edge TPU compiler on the Coral Dev Board Mini, the VeriSilicon TIM-VX delegate on the Khadas VIM3 NPU, TensorRT on the NVIDIA Jetson Nano GPU, and the Vitis AI flow on the AMD Kria KV260 FPGA. Every platform is then evaluated under one protocol on latency, throughput, mAP at an IoU threshold of 0.50 (mAP$_{50}$), power, energy per inference, and price-to-performance. The hardware and its native toolchain are the only variables in the experiment.

3.2 Detection Model

Selecting a shared detection model was a central design decision driven by strict deployment constraints. The model had to be natively designed for edge inference rather than scaled down from a server architecture. It also required exporting to a full-integer INT8 TensorFlow Lite graph to satisfy the Edge TPU's strict 8-bit arithmetic limits while exporting to a form intended for deployment across the four toolchains.

EfficientDet-Lite2 meets these constraints. The EfficientDet architecture [4] was adapted into the mobile-oriented EfficientDet-Lite family, which exports natively to INT8 Ten-

sorFlow Lite and has been deployed and evaluated on the Edge TPU in this form [23]. Among the five variants, Lite2 represents the most suitable operating point for this task. Lite0 and Lite1 run at lower input resolutions of 320 and 384 pixels and give up the detection accuracy that matters for small objects, such as distant signs and lights. Lite3 and Lite4 are larger, at roughly 11 and 20 megabytes once quantized, and both exceed the 8 MB of on-chip static memory on the Edge TPU described in Section 2.1. A model that does not fit on-chip must be streamed from external memory during inference, which erodes the throughput that the accelerator is designed to provide. Lite2 sits at a 448-pixel input and roughly 7 MB once quantized, which fits within the on-chip budget while keeping enough resolution for the seven-class task. The full architecture, including the EfficientNet-Lite backbone and the weighted bidirectional feature pyramid, is described in Section 2.4.

Two alternatives were evaluated against the core constraint: a single fine-tuned checkpoint must export to INT8 TensorFlow Lite and deploy across toolchains. Neither succeeded. SSD MobileNetV2 trained cleanly but operated at a lower input resolution and, in the available tooling, lacked a full cross-platform INT8 export path. MediaPipe Model Maker generated runtime-coupled models incompatible with the remaining toolchains. Consequently, EfficientDet-Lite2 was selected as the sole architecture satisfying all criteria from a single checkpoint.

The exported model carries one property that shapes the evaluation. The TensorFlow Lite graph emits a fixed maximum of 25 detections per image, set by its post-processing layer. This caps recall in dense traffic scenes that contain more than 25 objects. The cap applies identically on every platform, so it does not bias the comparison between the boards. It is stated here because it bounds the absolute accuracy figures reported in Chapter 5. Reduced-precision inference is likewise a property of every platform in the study, since each accelerator's toolchain requires it in some form. The way each toolchain implements that quantization is not identical, and that difference proves to have

a measurable effect on accuracy on one platform, examined in Chapter 5 and set out in design terms in Section 3.6.

3.3 Dataset and Detection Task

The dataset defines the detection task and is therefore kept identical across platforms. The study uses BDD100K [5], introduced in Section 2.7. It serves as an authentic foundation for this work because its imagery originates from actual, unstaged driving sequences rather than curated or synthetic environments. As a result, the accuracy metrics gathered reflect the uncontrolled, real-world conditions that an autonomous perception system must contend with.

Rather than the full BDD100K label set, the detector is trained on a focused subset of seven classes: red, green, and yellow traffic lights, pedestrians, cars, trucks, and traffic signs. These are the objects whose presence and state most directly govern near-term control decisions. Traffic lights are split by color because a perception system needs the signal state and not merely the fact that a light is present. A detector that reports a light without its color gives the downstream system almost nothing to act on. The remaining BDD100K categories were left out to keep the model focused on the classes that matter for this task and to avoid diluting training on objects that are either rare in the data or peripheral to the perception decision.

Training and evaluation use a prepared subset of BDD100K, filtered to the seven target classes and converted to the Pascal VOC annotation format that TensorFlow Lite Model Maker ingests. The subset is split into a training partition and a fixed validation partition of $4{,}993$ images. That validation set and its ground-truth annotations are the single source of truth for accuracy. Every platform that runs this model is scored against the same images and the same labels, which is what allows the per-class and overall accuracy figures in Chapter 5 to be compared directly between them.

3.4 Hardware Platforms

The four platforms are chosen to represent four different ways of accelerating a neural network in hardware, rather than four variations of one approach. The Coral Dev Board Mini carries the Edge TPU, a fixed-function inference ASIC that executes INT8 operations on a systolic array. The Khadas VIM3 carries the Amlogic A311D with an integrated NPU. The Jetson Nano uses a general-purpose Maxwell-class GPU. The Kria KV260 is a reconfigurable FPGA running a soft DPU. Each platform is described in detail in Section 2.1, and the four boards are shown together in Figure 3.2.

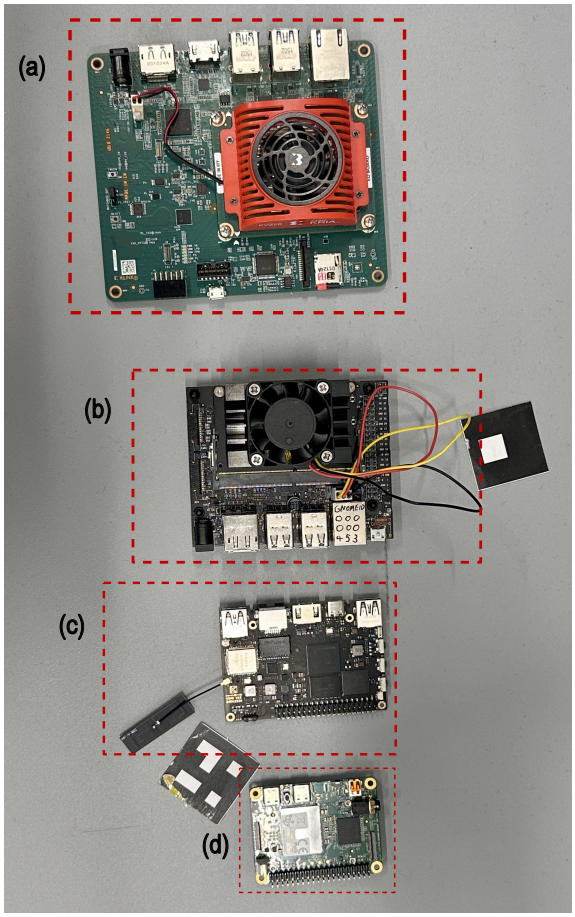


Figure 3.2: The four edge accelerator platforms evaluated in this thesis: (a) the AMD Kria KV260 (FPGA-based DPU), (b) the NVIDIA Jetson Nano (GPU), (c) the Khadas VIM3 (NPU), and (d) the Coral Dev Board Mini (Edge TPU).

Placed together, these four span the flexibility-efficiency range that defines the edge-accelerator landscape. The Edge TPU sits at the fixed, efficient end, featuring purpose-

built hardware for one class of operation which operates at minimal power cost. The FPGA sits at the flexible end, where the datapath is reconfigurable but incurs an overhead. The NPU and the GPU occupy the middle ground. Comparing across this range rather than within a single paradigm allows the study to address the question of which acceleration approach suits real-time traffic perception, in lieu of simply capturing a marginal throughput advantage between near-identical competitors.

The four are all commercially available single-board platforms aimed at edge deployment, but they do not share a price point. The Kria KV260 has the highest purchase price among the four. This spread is the reason the study reports price-to-performance alongside speed and efficiency. A board that is faster but considerably more expensive is not obviously the better choice for a cost-constrained deployment, and the comparison is only complete if price is one of its axes.

3.5 Evaluation Methodology

Because no single metric dictates an accelerator's viability for real-time traffic perception, this study evaluates speed, accuracy, power consumption, and device cost together. While precise definitions are established in Section 2.8, this section explains why each is measured and how fairness is maintained across platforms.

Speed is measured in two ways because the two answer different questions. Invoke-only latency measures the accelerator alone, with the model already loaded and the input already prepared, isolating the raw computational capability of the hardware. End-to-end latency times the full per-frame pipeline, including reading the image, resizing it to the model input, running inference, and decoding the output, and reflects what a deployed system experiences frame to frame. Reporting only device-level latency would overstate practical utility and obscure pipeline costs, while reporting only end-to-end latency would blur hardware differences behind largely identical preprocessing. Both are therefore re-

ported, and throughput in frames per second is derived from each. This separation of a tight repeated-inference measurement from a realistic full-pipeline measurement follows the timing methodology of Baller et al. [7].

Accuracy is measured as mean average precision at an IoU threshold of 0.50 (mAP_{50}), the established Pascal VOC detection protocol [72]. A detection counts as correct when its box overlaps the ground-truth box by at least half, which is a reasonable threshold for a perception system that must localize an object well enough to act on it. Across the platforms that run the shared model, the IoU threshold, the confidence score cut-off, the evaluation logic, and the ground-truth annotations are identical. This is the core fairness control of the study. Because the same network is scored on the same images under the same rules, any difference in their measured accuracy traces to how each platform's toolchain compiled and quantized the model, and to nothing else.

Power and energy are measured at the board input during sustained inference and reported as average power in watts and as energy per inference in joules. Measuring at the board input captures total system draw rather than an accelerator-only figure isolated from the host and peripherals, which is what a real deployment pays for. Voltage and current are never reported on their own since neither describes consumption without the other. Energy per inference, the product of average power and inference time, is the figure that compares platforms fairly on efficiency. A board that draws more power but finishes much sooner can be more efficient per frame than a slower, lower-power one, and only the energy figure makes that visible. The instruments used on each board are described in Chapter 4.

Cost enters as a price-to-performance figure, which relates each board's purchase cost to its measured throughput. As Section 3.4 notes, the platforms differ in price, so this figure reflects which board delivers the most capability per unit of spend, not merely which is fastest or most efficient.

3.6 Design Constraints and Deviations

The design holds the model constant across platforms wherever the hardware allows it. Two of the four boards did not allow it fully. Both deviations are reported here because the way they are handled is itself a design decision.

The Jetson Nano could not run EfficientDet-Lite2. As documented in Chapter 4, converting the model to a TensorRT engine failed across four separate conversion paths because its custom NMS operator is unsupported by TensorRT 8.0 on JetPack 4.6. Rather than dropping the GPU paradigm from the study, YOLOv4-416 compiled as a TensorRT half-precision engine is used as the Jetson's reference workload. This has a direct consequence for the comparison. The Jetson runs a different model on a different dataset, so its accuracy figure is not comparable to the mAP measured on the other three boards. The Jetson is therefore reported as a reference point for GPU latency and power, and its accuracy is deliberately kept out of the cross-platform accuracy comparison. Its speed and energy figures remain directly informative about what a Maxwell-class GPU delivers in this form factor. Handled this way, the substitution is not only a limitation but also a finding. That the fastest accelerator in the study is the one platform unable to run the target model shows that hardware capability does not guarantee deployability, a distinction that is itself a practical consideration in edge hardware selection.

The Kria KV260 runs EfficientDet-Lite2, but its DPU requires the model to be quantized with power-of-two scales, a more constrained scheme than the TensorFlow Lite quantization used on the Edge TPU and the NPU. Under that scheme, the classification head quantizes in a way that collapses detection accuracy far below the model's baseline, while latency and power are unaffected because the hardware still runs the network at full speed. The study treats this as a result to be explained rather than a result to be suppressed. The latency and power of the KV260 are reported as authoritative measurements of the FPGA paradigm. This collapse is confined to the DPU quantization stage, which Chapter 5 isolates and quantifies. The thesis documents this as a known class of quantization

sensitivity on this hardware and model combination, where the model has not previously been characterized on this class of DPU.

Both deviations reflect a point that the design accepts from the outset. The four platforms do not implement the same quantizer, and that difference is one of the things the study exists to measure. The TensorFlow Lite scheme used on the Edge TPU and the NPU keeps per-channel weight scales and arbitrary floating-point scaling factors [64]. The DPU on the FPGA constrains every scale to a power of two so that the runtime can replace a multiplication with a hardware bit shift [65]. These are properties of the hardware, described in Section 2.5, and the study measures their consequences rather than treating any one scheme as the reference.

3.7 Summary

This chapter has set out a controlled benchmarking design. A single object-detection model, EfficientDet-Lite2, trained on seven traffic-relevant classes of BDD100K, defines the common workload for the study. The model is deployed on the Edge TPU, NPU, and FPGA, while the GPU uses YOLOv4-416 as a substitute model because of toolchain constraints. Every board is evaluated on the same fixed validation set, under the same accuracy protocol, and on the same set of metrics covering speed, accuracy, power, and cost, with only the hardware and its toolchain changing between them. Two platforms required deviations from the shared model: the Jetson Nano because the model could not be converted for its runtime, and the Kria KV260 because the required quantization scheme degrades accuracy. Both are reported transparently and handled so that they inform the comparison rather than distort it. The next chapter describes how this design was carried out on each platform.

4 Implementation

The previous chapter defined *what* was held constant and *why*. This chapter describes its execution: the training and export of the shared model, its deployment across the four platforms using their respective toolchains, and the issues resolved to ensure reliable measurement. The emphasis throughout is on the steps that are not obvious from the toolchain documentation because those steps are what make the results in Chapter 5 reproducible. The numerical results themselves (latency, throughput, accuracy, power, and energy) are reported and interpreted in Chapter 5; this chapter establishes how they were produced.

All host-side work (dataset preparation, model export, ONNX conversion, quantization and compilation, and the evaluation harness) was carried out on a single workstation running Ubuntu 22.04, and each accelerator board was driven over the network from that workstation. The hardware of each platform is described in Section 2.1, the software and toolchain stack actually used to deploy the model on each is summarized in Table 4.1, and the remainder of the chapter expands each row in turn.

4.1 Model Training and Export

The detector is the one element shared by every platform (Section 3.2), so training and export are described once here, and the resulting artifacts are reused unchanged downstream. The training configuration is summarized in Table 4.2.

Table 4.1: Software and toolchain stack used to deploy the detector on each platform. Hardware is described in Section 2.1.

Platform	Operating system	Runtime / framework	Precision
Coral Dev Board Mini	Mendel Linux (Eagle)	Edge TPU runtime + PyCoral [73]	INT8 (TFLite PTQ)
Khadas VIM3	Ubuntu 20.04 (kernel 4.9)	TFLite + TIM-VX delegate (source-built) [69]	INT8 (TFLite PTQ)
Jetson Nano	JetPack 4.6 (Ubuntu 18.04, CUDA 10.2, cuDNN 8.2, TensorRT 8.0.1.6)	TensorRT [70]	FP16
Kria KV260	Ubuntu 22.04, `x07` firmware; Vitis AI 3.5	ONNX Runtime + Vitis AI EP / VART [42]	INT8 (Quark XINT8)

4.1.1 Dataset Preparation

The BDD100K detection labels are distributed as a single large JSON file, which the training pipeline cannot consume directly. A preparation script converted the annotations into the per-image Pascal VOC XML format expected by the trainer, filtering the ten native categories down to the seven traffic-relevant classes defined in Section 2.7 and writing paired image and annotation directories for the training and validation partitions. Sampling and the train/validation split used a fixed random seed so that the split is reproducible, yielding the $33,288$-image training partition and the fixed $4,993$-image validation partition used as the single source of truth for accuracy.

An initial training run was discarded after a label-conversion defect. It silently dropped every non-traffic-light annotation and misparsed the traffic-light color, which BDD100K stores as a two-element attribute rather than a plain string. The corrected run (`run3`), the source of `model_run3.tflite`, fixed both issues, and all results in this thesis use it.

4.1.2 Training

Fine-tuning used the `tflite-model-maker` object-detection pipeline [74], which wraps the EfficientDet-Lite training loop and is the only pipeline that exports the family to fully integer TensorFlow Lite (Section 3.2). The model specification selected

the `efficientdet-lite2` architecture initialized from the published feature-vector checkpoint, with the maximum number of training instances per image raised to accommodate the dense BDD100K scenes. The remaining hyperparameters are listed in Table 4.2; validation data was deliberately not passed to the trainer, and the consequence of that choice for checkpoint selection is addressed next.

Table 4.2: Training and export configuration for the shared detector.

Parameter	Value
Detector	EfficientDet-Lite2
Input resolution	$448 \times 448 \times 3$ (`UINT8`)
Initialization	EfficientDet-Lite2 feature-vector checkpoint
Training pipeline	`tflite-model-maker` v0.4.3 (Python 3.8, TensorFlow 2.8.4)
Training hardware	rented cloud GPU, NVIDIA A40
Dataset	BDD100K, seven-class subset
Training / validation images	33,288 / 4,993
Epochs	75
Batch size	24
Fine-tuning	whole model (`train_whole_model=True`)
Validation during training	none (manual checkpoint selection)
Selected checkpoint	`ckpt-59`, training loss 0.3700
Exported model	`model_run3.tflite`, INT8, 7.1 MB, max. 25 detections

4.1.3 Checkpoint Selection

Because no validation set was supplied during training, the pipeline performed no automatic best-checkpoint tracking, and every epoch wrote a checkpoint with no built-in criterion for choosing among them. The training-loss trajectory was therefore inspected manually from the training log, and the checkpoint at the loss minimum was selected; later epochs showed a small loss increase as the learning rate decayed toward zero. The selected checkpoint (Table 4.2) defines `model_run3`.

4.1.4 Export to TensorFlow Lite and ONNX

The selected checkpoint was exported with training disabled, producing the fully INT8-quantized TensorFlow Lite model `model_run3.tflite` (7.1 MB; $448 \times 448 \times 3$

`uINT8` input). The graph exposes four output tensors produced after NMS, with a fixed maximum of 25 detections per image, as noted in Section 3.2. This single file is the artifact deployed to the Coral and VIM3 platforms and recompiled for the Kria DPU.

Two additional conversions were required for the FPGA path, which consumes ONNX rather than TensorFlow Lite. A floating-point ONNX graph (`model_run3_fp32.onnx`) was produced from the SavedModel with `tf2onnx` [75]. It takes a `float32` input and exposes the ten raw feature-pyramid output tensors with the anchor decode and NMS removed, which is the form the DPU quantizer and compiler require. A second ONNX graph was converted directly from the INT8 TensorFlow Lite model, retaining the four post-NMS outputs. This graph is the integer-equivalent of the deployed model and is used in Section 4.6 as the CPU accuracy reference for the FPGA platform.

4.2 Coral Dev Board Mini (Edge TPU)

The Coral path is the most direct of the four because the model was authored for it. The INT8 TensorFlow Lite model was passed through the Edge TPU Compiler [68], which maps the supported operators onto the accelerator and leaves the remainder on the host CPU. The model compiled as a single segment, with only the custom NMS operator left to run on the quad-core Cortex-A35 host, as anticipated in Section 2.6. Inference was driven through the PyCoral API [73].

One implementation detail was critical for runtime correctness: PyCoral returns references to the internal interpreter buffers that are not guaranteed to persist across inference calls. Deferring access to the detection outputs led to corrupted or overwritten values. This was resolved by copying each output tensor immediately after the invocation, before any post-processing, ensuring that subsequent decoding operates on stable host memory.

With this adjustment, the Coral pipeline ran reliably without further modification.

The only host-side issue encountered concerned buffer-lifetime management, rather than model conversion, operator compatibility, or quantization. This outcome is itself significant: the vendor-native deployment path imposes minimal porting effort, establishing a baseline against which the complexity of the other platforms can be compared.

4.3 Khadas VIM3 (A311D NPU)

The VIM3 executes the same INT8 TensorFlow Lite model, but access to the NPU is provided through the VeriSilicon TIM-VX delegate rather than the vendor Amlogic SDK, which lacks support for several operators required by EfficientDet-Lite2 (Section 2.6). Deployment therefore used the external delegate for TensorFlow Lite [69], compiled from source `libvx_delegate.so`, built against `libtim-vx.so` to ensure full operator coverage. The delegate passes the quantized graph directly to the VeriSilicon Vivante NPU.

A practical constraint governed all measurement runs on this platform. The NPU driver initializes only when attached to an active controlling terminal; detaching the process (for example, with `nohup`) prevents the driver from bringing up the NPU. Consequently, all inference and benchmarking were executed within a persistent terminal session (`screen`) maintained for the duration of each run.

Aside from building the delegate and accommodating this runtime constraint, no model-level modifications were required. The same INT8 artifact used on Coral executed unchanged on the Vivante NPU, confining the porting effort to the runtime stack rather than the model.

4.4 NVIDIA Jetson Nano (Maxwell GPU)

The Jetson Nano is the documented exception to the shared-model design (Section 3.6). This section records both the conversion failure that forced the substitution and its deploy-

ment, because the failure analysis informs the answer to Research Question 3 in Chapter 6. The substitution raises an immediate question of comparability, and the governing principle adopted here is stated before the details. Where a platform cannot run the shared detector, the platform-intrinsic axes of latency, power, and energy remain directly comparable, because they characterize what the hardware delivers in this form factor. Only the accuracy axis is affected by the change of model and dataset, and it is therefore reported separately and is never placed on the same scale as the three EfficientDet-Lite2 platforms. Retaining the Jetson on these axes therefore serves two distinct purposes: its latency, power, and energy quantify what a Maxwell-class GPU delivers on a representative detector, while the conversion failure itself, treated in Section 4.4.1, records that peak capability did not translate into deployability for this workload.

4.4.1 The EfficientDet-Lite2 Conversion Failure

The intent was to execute the same EfficientDet-Lite2 model on the Jetson through TensorRT [70]. The board was flashed with JetPack 4.6 (L4T 32.6.1, Ubuntu 18.04, CUDA 10.2, cuDNN 8.2.1), which supplies TensorRT 8.0.1, and that TensorRT build could not be made to accept the model. The following four independent conversion paths were attempted on this stack, and each failed:

1. **Direct TensorFlow Lite → TensorRT.** Blocked by the model's custom NMS post-processing operator, a TensorFlow Lite custom operator with no TensorRT equivalent.

2. **TensorFlow Lite → ONNX via `tflite2tensorflow` → TensorRT.** Blocked by an unimplemented `QUANTIZE` operator in the conversion.

3. **SavedModel → ONNX (opset 16) via `tf2onnx` [75] → TensorRT.** Blocked by a `Reshape` `allowzero` attribute that the available TensorRT build does not accept (the opset exceeded parser support).

4. **SavedModel → ONNX (opset 13) via `tf2onnx` → TensorRT.** Blocked by a `GatherND` operator that the available TensorRT build does not support in the configuration required by the graph.

The common root is the custom NMS operator. It cannot cross the TensorRT boundary directly, and the intermediate ONNX routes attempting to express it each introduce an operator or attribute rejected by the TensorRT 8.0 build. A newer runtime (TensorRT 8.2.1, available for this board through the JetPack 4.6.x point release) was not evaluated, so the conversion failure reported here is specific to the JetPack 4.6/TensorRT 8.0 stack. However, the hardware imposes a separate and version-independent limit: the Maxwell generation lacks a fast integer dot-product path, so even a successful INT8 conversion would not have achieved acceleration on this GPU. The artifacts from all four attempts were retained, and the failure is examined in relation to Research Question 3 in Chapter 6.

4.4.2 YOLOv4-416 Deployment

A YOLOv4-416 detector was deployed as a TensorRT half-precision (FP16) engine, which suits the strengths of the Maxwell architecture and avoids the integer path that the hardware does not accelerate. YOLOv4-416 was chosen because it is a representative real-time detector with a mature, Maxwell-compatible TensorRT path, which keeps the comparison focused on object detection rather than dropping to a less demanding task such as classification. The engine was built and run using the `jkjung-avt/tensorrt_demos` open-source framework with its YOLO layer plugin [76]; the input resolution was 416×416 at a batch size of one. Deploying on the Maxwell target required several platform-specific fixes. The architecture flags were pinned to `compute_53`/`sm_53`, a conflicting host `cmake` Python package was removed before the ONNX dependency would install, and the external storage used for the evaluation images was reformatted to a filesystem that the board would mount cleanly. Before every timed run, the board was placed in its maximum-performance mode with

`jetson_clocks`, locking the GPU and CPU clocks so that thermal throttling did not contaminate the latency measurements. Runs taken before this lock was applied were discarded.

Because a different model was evaluated on a different dataset, the Jetson accuracy figure was computed on COCO-val2017 and is reported separately in Chapter 5. The Jetson's speed and energy figures remain directly informative about what a Maxwell-class GPU delivers in this form factor.

4.5 AMD Kria KV260 (FPGA + DPU)

The Kria path is the most involved of the four. The model is not merely converted but recompiled into a DPU instruction stream, and the runtime stack required substantial diagnosis before it would execute on the accelerator. The four parts below follow the order in which they must be performed: host-side quantization and on-board compilation, board bring-up, correct decoding of the raw model output, and characterization of the accuracy behavior that the toolchain's quantizer produces. Figure 4.1 shows the KV260 as deployed in this study, connected via Ethernet and USB for inference execution and data transfer.

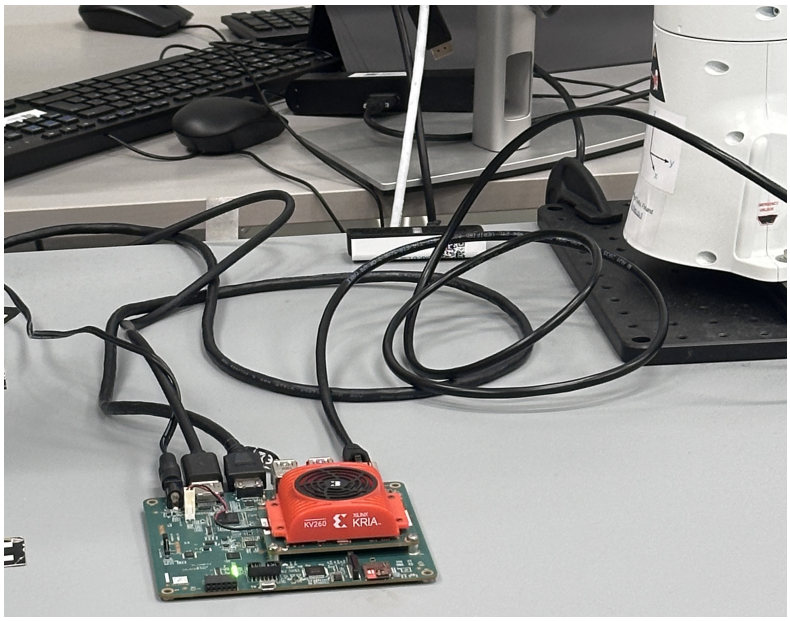


Figure 4.1: The AMD Kria KV260 Vision AI Starter Kit as deployed in this benchmark. The board integrates a Zynq UltraScale+ Multi-Processor System-on-Chip (MPSoC) carrying a DPU clocked at 300 MHz for hardware-accelerated INT8 inference. Ethernet, USB, and display connections are visible; the active cooling fan on the SoM manages thermal load during sustained inference.

Both the operating system and the inference path on the Kria followed from a single overriding requirement that the same EfficientDet-Lite2 model runs on every platform so that the comparison remains controlled. Substituting a more readily compiled model on this board would have simplified the engineering but introduced a second variable, so the shared EfficientDet-Lite2 model was retained (Section 3.5). The board was run under Ubuntu 22.04, the certified operating system that AMD supports for the KV260, which supplies the Python userspace that the shared evaluation harness depends on. A bespoke PetaLinux image would have demanded a separate build toolchain and would not have altered the quantization behavior reported in Section 4.5.4, since that behavior is a property of the quantizer and not of the operating system.

The inference path followed the same constraint. The execution was driven through the Vitis AI ONNX Runtime path for compilation to a `.xmodel`. The classic route failed for this model through two distinct paths: the frozen-graph format was rejected by the on-host compiler due to missing weight names, and the ONNX-to-Keras conversion

required by the TensorFlow2 compiler path failed before reaching the compiler because `onnx2keras` could not produce a consistent NHWC graph for this architecture. The ONNX Runtime path was the only route that ran the same model on the DPU, at the cost of the additional bring-up steps described below.

4.5.1 Quantization and Compilation

The compile-time flow follows the Vitis AI stack described in Section 2.6. The floating-point ONNX graph from Section 4.1.4 was quantized to INT8 with the Vitis AI quantizer (AMD Quark) [42], [65]; the configuration is summarized in Table 4.3. The DPU target requires symmetric INT8 with per-tensor, power-of-two scales, which is the constraint analyzed in Section 2.5; however, per-channel quantization is not available for this target [65]. Calibration consumed a set of pre-processed BDD100K images supplied as NumPy arrays named after the model's input tensor, in the network's native layout, a format requirement that the quantizer enforces and that raw image files do not satisfy. The reported configuration uses a 500-image calibration set, the largest in the ablation reported in Chapter 5. As that ablation shows, calibration-set size has a negligible effect on accuracy (mAP50 of 0.0216 at 200 images against 0.0233 at 500), which is itself evidence that the accuracy collapse is structural rather than calibration-limited. This produces the integer-quantized graph `model_run3_quark_500.onnx`. The transition from quantization to deployment then follows the Vitis AI ONNX Runtime Engine (VOE) runtime path rather than host-side compilation. In this flow, the quantized graph is not pre-compiled on the host. It is deployed to the board as an ONNX file, and the Vitis AI execution provider compiles its DPU-eligible subgraph to a `.xmodel` instruction binary at the first session load, caching the result and stamping it with the board's DPU fingerprint.

Table 4.3: Quantization and calibration configuration for the Kria DPU. The granularity and scale constraints are properties of the target and are analyzed in Section 2.5.

Setting	Value
Quantizer	Vitis AI quantizer (AMD Quark 0.11.1)
Data type	INT8 (XINT8); symmetric weights and activations
Scale	power-of-two, $S = 2^{-f}$
Granularity	per-tensor (per-channel unsupported for the DPU)
Calibration method	MinMSE
Calibration sizes (ablation)	50, 200, 500 images
Deployed model	500-image calibration (`model_run3_quark_500.onnx`)
Compiler target	`DPUCZDX8G_ISA1_B4096`
DPU fingerprint	`0x101000056010407` (`x07` firmware)
Runtime	ONNX Runtime 1.16.0 + Vitis AI EP / VART

A toolchain defect dominated this stage for much of the project. The board's on-chip compiler aborted with a stack-smashing error when compiling a graph as large as EfficientDet-Lite2, which forced a fragile reliance on reusing a previously cached `.xmodel`. The defect was resolved by a firmware update (designated `x07`) supplied by the supervisor, after which compilation completed normally. All authoritative KV260 measurements were taken on the `x07` firmware.

4.5.2 Board Bring-up and Runtime

Executing the compiled model on the DPU under the ONNX Runtime Vitis AI execution provider required a specific and order-sensitive bring-up. The essential steps, in order, are: (i) recreate the OpenCV shared-library symlinks, which must be done after every reboot; (ii) load the DPU overlay as the normal user with the runtime path passed on the command line; (iii) elevate to root, set the library path, and activate the runtime environment; and (iv) confirm that the Vitis AI execution provider is registered before launching the benchmark. Deviating from this order (in particular, loading the overlay as the root user, or setting the runtime path through `os.environ` rather than on the command line) leaves the execution provider unregistered. The exact commands are given in Listing 4.1.

Two runtime behaviors are worth noting. First, the `libvaip_custom_op_DPU.so` plugin failed to load until the two OpenCV symlinks in step 1 were created because the

Listing 4.1: KV260 DPU bring-up sequence (x07 firmware).

```
# 1. Recreate the OpenCV symlinks (as the normal user; needed
   every reboot)

ln -sf /usr/lib/aarch64-linux-gnu/libopencv_imgproc.so.4.5.4d
/usr/lib/libopencv_imgproc.so.406
ln -sf /usr/lib/aarch64-linux-gnu/libopencv_core.so.4.5.4d
/usr/lib/libopencv_core.so.406

# 2. Load the DPU overlay as the normal user, with XILINX_XRT set
   inline

sudo XILINX_XRT=/usr
/usr/local/share/pynq-venv/bin/python3 -c
"from␣pynq_dpu␣import␣DpuOverlay;␣DpuOverlay('dpu.bit')"

# 3. Become root, set the loader path, activate the runtime
   environment

sudo su
export LD_LIBRARY_PATH=/usr/lib
source /usr/local/share/pynq-venv/bin/activate

# 4. Verify the provider, then run the benchmark

python3 -c "import␣onnxruntime␣as␣ort;␣
   print(ort.get_available_providers())"

# Expect VitisAIExecutionProvider in the list, then run the
   benchmark script
```

runtime expected the `.406` sonames corresponding to the installed 4.5 libraries. Second, on the `x07` firmware, the first model load may emit an apparent XRT device error, after which inference proceeds on the DPU; this is an internal recovery within the runtime and not a cache-rebuild cycle, and no manual reload is required.

4.5.3 Post-processing and Anchor Decoding

Unlike the TensorFlow Lite deployments, the quantized ONNX graph running on the DPU (Section 4.5.1) emits the ten raw feature-pyramid tensors with no anchor decode and no NMS (Section 4.1.4); therefore, the entire detection post-processing had to be

reconstructed on the host. The decisive subtlety was the anchor format. The anchors recovered from the model are in center form (c_y, c_x, h, w), not corner form, and the box-regression outputs are applied as offsets to that center with no variance scaling:

$$c'_y = c_y + b_0\, h, \qquad c'_x = c_x + b_1\, w, \tag{4.1}$$

$$h' = h\, \exp(b_2), \qquad w' = w\, \exp(b_3), \tag{4.2}$$

where (b_0, b_1, b_2, b_3) is the regressed offset for the anchor. This decode was verified to reproduce the floating-point TensorFlow Lite output box-for-box on a reference image, which confirmed both the format and the absence of any variance term. Instrumenting the raw output further showed that several thousand candidate boxes per image reached the suppression stage; a top-K pre-NMS filter ($K = 500$) was added before NMS to keep host post-processing tractable without affecting the retained detections on the validation images. The complete host-side procedure is given in Algorithm 1.

Algorithm 1 KV260 host-side detection post-processing

Require: raw DPU outputs (box-regression and class-logit maps per pyramid level); anchor set $\mathcal{A}$ in center form (c_y, c_x, h, w); score threshold τ; NMS IoU threshold ν; pre-NMS cap $K = 500$

Ensure: retained detections as (box, class, score) triples

concatenate the per-level maps into anchor-aligned arrays of regressions and class logits
$D \leftarrow \emptyset$
for each anchor $(c_y, c_x, h, w) \in \mathcal{A}$ with regression (b_0, b_1, b_2, b_3) and class logits ℓ **do**
    $c'_y \leftarrow c_y + b_0 h \quad c'_x \leftarrow c_x + b_1 w$ ▷ center offset
    $h' \leftarrow h\, e^{b_2} \quad w' \leftarrow w\, e^{b_3}$ ▷ log-space size
    convert (c'_y, c'_x, h', w') to corner form (y_1, x_1, y_2, x_2)
    $s \leftarrow \max_k \sigma(\ell_k) \quad c \leftarrow \arg\max_k \sigma(\ell_k)$ ▷ sigmoid head
    if $s \geq \tau$ **then**
        append $(y_1, x_1, y_2, x_2, c, s)$ to D
    end if
end for
retain the K highest-scoring entries of D ▷ pre-NMS filter
$R \leftarrow \mathrm{NMS}(D, \nu)$
return R

4.5.4 The Accuracy-Collapse Investigation

With the DPU executing and the post-processing verified, detection accuracy nonetheless collapsed far below the model's baseline. The mechanism (per-tensor, power-of-two quantization of the depthwise-separable classification head) is established in Section 2.5; this section records the procedure of the investigation and the remediations attempted, while the quantified results are tabulated and interpreted in Chapter 5.

The collapse was first characterized with a calibration-set ablation at 50, 200, and 500 images, which showed that enlarging the calibration set did not recover accuracy and that the loss was structural rather than a calibration artifact. A sequence of toolchain-level remediations was then attempted, each ruled out for a definite reason. Unsigned-integer activations (`QUINT8`) were rejected outright by the DPU compiler. Excluding the classification-head nodes by name failed silently because model optimization renames nodes before quantization, so the exclusion list no longer matched. Excluding them as subgraphs caused the compiler to fail on an anchor-reshape mismatch. Cross-layer equalization made accuracy worse rather than better. The reason is specific and instructive. Its high-bias-absorption step is skipped for the classification head because that head uses a sigmoid rather than a ReLU activation, so equalization removes useful scale from the convolutional weights without a compensating bias adjustment.

A controlled bias-correction experiment then isolated the cause. The focal-loss classification bias of Section 2.4, although real and measured, was directly corrected; accuracy barely moved, which establishes that the bias is not the dominant cause and that the collapse originates in the per-tensor power-of-two quantization of the classification-head convolution weights. Ultimately, because the toolchain remediations described above failed to yield a deployable hybrid configuration, no partial accuracy recovery could be achieved on-device.

This behavior also dictates the methodology for reporting the KV260, which is set out with the rest of the evaluation in Section 4.6. The platform's latency and power are mea-

sured from the model running on the DPU, while its detection accuracy is reported from the integer-equivalent graph evaluated on the host CPU, because the DPU's quantizer, not the network or the FPGA's compute capability, is what degrades the accuracy.

4.6 Measurement and Evaluation Harness

The metric definitions and the fairness controls are given in Section 2.8 and Section 3.5. This section records how each quantity was actually measured, so that the figures in Chapter 5 are reproducible.

4.6.1 Latency and Throughput

Two latency figures were collected on every platform. Invoke latency times only the accelerator's inference call, with the model already loaded and the input already prepared; it is measured over a fixed number of repeated calls after a warm-up phase that excludes one-time engine and cache initialization. End-to-end latency times the full per-frame pipeline (image read, resize to the model input, inference, and host-side decode and NMS) over the validation images. Both are summarized by their median, and the throughput in frames per second is derived from each, following Section 2.8. Reporting the median rather than the mean keeps the figures robust to the occasional outlier introduced by host scheduling. Algorithm 2 sets out the measurement procedure.

Algorithm 2 Per-platform latency measurement

Require: loaded model M; validation images $\mathcal{V}$; timed-invoke count N; warm-up count W

Ensure: median invoke latency, median end-to-end latency, and the throughput derived from each

prepare one fixed model input x
for $i = 1$ to W **do**
    run $M(x)$ ▷ warm-up; timing discarded
end for
$T_{\text{inv}} \leftarrow \emptyset$
for $i = 1$ to N **do**
    $t_0 \leftarrow \text{now}()$; run $M(x)$; $t_1 \leftarrow \text{now}()$
    append $t_1 - t_0$ to T_{inv}
end for
$T_{\text{e2e}} \leftarrow \emptyset$
for each image $v \in \mathcal{V}$ **do**
    $t_0 \leftarrow \text{now}()$
    $x_v \leftarrow \text{resize}(\text{read}(v))$; $y \leftarrow M(x_v)$; decode and suppress y
    $t_1 \leftarrow \text{now}()$; append $t_1 - t_0$ to T_{e2e}
end for
return $\text{median}(T_{\text{inv}})$, $\text{median}(T_{\text{e2e}})$, and the frames per second derived from each

4.6.2 Power and Energy

Power was measured at the board input during sustained inference and reported as average power in watts and energy per inference in joules; voltage and current are never reported in isolation. The instrument differs by platform, as listed in Table 4.4; this difference is carried forward as a methodological note in Chapter 5. Energy per inference, the product of average power and inference time, is the figure that compares the platforms fairly on efficiency, because a board that draws more power but finishes sooner can still be the more efficient choice per frame.

Table 4.4: Power measurement instrument used on each platform. The differing methodology is noted as a limitation in Chapter 5.

Platform	Power instrument
Coral Dev Board Mini	ATORCH J7-C external in-line USB meter
Khadas VIM3	ATORCH J7-C external in-line USB meter
Jetson Nano	on-board INA3221 via `tegrastats` (total-input rail)
Kria KV260	INA260 sensor

4.6.3 Accuracy

Detection accuracy is reported as mAP at an IoU threshold of 0.50 (mAP_{50}), per Section 2.8. The per-class AP breakdown is retained to expose how quantization and platform differences affect rare classes such as `tl_yellow`. For the three platforms running EfficientDet-Lite2, accuracy was computed on the identical $4,993$-image BDD100K validation subset, the core fairness control established in Section 3.5. As noted in Section 4.4.2, the Jetson YOLOv4-416 figure was computed on COCO-val2017 using the COCO API and is reported separately. For the KV260, following Section 4.5.4, latency and power come from the DPU execution, while mAP_{50} is taken from the integer-equivalent ONNX graph evaluated on the host CPU. This CPU-evaluated figure aligns closely with the Coral and VIM3 results, serving as the model's intrinsic accuracy reference for this dataset.

4.7 Summary

This chapter has described how the single EfficientDet-Lite2 model was trained and exported, and how it was deployed on each of the four platforms. It also detailed the issues that had to be resolved to obtain trustworthy measurements. On the Coral, the key fix required copying the output buffers. VIM3 deployment relied on a source-built TIM-VX delegate and was constrained by the requirement for an active terminal session. On the Jetson Nano, repeated conversion failures led to the substitution of YOLOv4-416. The

Kria KV260 involved a more complex path, including host-side quantization and on-board compilation, an order-sensitive board bring-up sequence, and a center-form anchor decoding. Further investigation traced the observed accuracy collapse to the DPU's per-tensor power-of-two quantizer. The software stack, training configuration, DPU quantization settings, and power instrumentation are summarized in Tables 4.1, 4.2, 4.3, and 4.4. The host-side post-processing and measurement procedures are given as Algorithms 1 and 2. It also set out how latency, power, energy, and accuracy were measured on each board, including the split by which the KV260's speed and power are reported from the DPU while its accuracy is reported from the host CPU equivalent. The measurements produced by these procedures are reported and compared in the next chapter.

5 Experimental Results

This chapter reports the measurements and interprets them in the context of the research questions from Section 1.1. All metrics and extraction procedures strictly follow the methodologies established in Chapters 3 and 4.

Two points govern how every figure below is read. First, the Jetson Nano ran a different detector on a different dataset (Section 4.4.2), so its accuracy is reported separately and is never placed on the same axis as the three platforms that ran the shared EfficientDet-Lite2 model; its speed and energy figures remain directly informative about a Maxwell-class GPU. Second, the Kria KV260 figures follow the split established in Section 4.5.4, with latency and power measured from the model running on the DPU and detection accuracy reported from the integer-equivalent graph evaluated on the host CPU.

5.1 Latency and Throughput

Table 5.1 reports the two latency figures and the throughput derived from each. Invoke latency is the median over a fixed series of repeated inference calls on a prepared input. End-to-end latency is the median over the validation images and includes the full per-frame pipeline. Throughput in frames per second is the reciprocal of the end-to-end median.

Table 5.1: Latency and throughput for each platform. Invoke and end-to-end (E2E) latencies are medians, in milliseconds. Throughput in FPS is the reciprocal of the end-to-end median. The Coefficient of Variation (CV) is for the repeated-inference series. The Jetson Nano figures are for YOLOv4-416 and are comparable on speed but not on accuracy.

Platform	Invoke (ms)	E2E (ms)	Throughput (FPS)	CV
Coral Dev Board Mini	326.87	375.61	2.66	3.53%
Khadas VIM3	1374.12	1407.35	0.71	0.01%
Jetson Nano	209.99	230.59	4.34	0.50%
Kria KV260	192.71	335.40	2.98	6.39%

The two latency measures place the platforms in different orders. On invoke latency, the bare accelerator call on prepared input, the Kria KV260 is fastest at 192.71 ms, ahead of the Jetson Nano at 209.99 ms, the Coral at 326.87 ms, and the VIM3 last at 1374.12 ms. The reconfigurable fabric and the GPU therefore lead on raw computation. On end-to-end throughput, the order changes: the Jetson Nano is the fastest at 4.34 FPS, followed by the KV260 at 2.98 FPS, the Coral at 2.66 FPS, and the VIM3 at 0.71 FPS. The reordering is driven entirely by host-side work. On the KV260, the whole detection post-processing is reconstructed on the host (Algorithm 1), and the gap between its invoke and end-to-end medians, roughly 143 ms, is the largest of any platform and is what drops it below the Jetson on a per-frame basis.

One qualification applies to the invoke column and tempers the raw-computation ranking. On the KV260, the quantized DPU graph emits the ten raw feature-pyramid tensors with neither anchor decode nor NMS, so its invoke time covers only the accelerator forward pass, whereas the Coral and VIM3 TensorFlow Lite graphs carry the NMS operator inside the model and therefore include it in their invoke. The invoke figures are thus not directly comparable, and the end-to-end column, which subjects every platform to the identical pipeline, is the sounder basis for per-frame comparison.

The VIM3 is an order of magnitude slower than the others on both measures, with its inference call alone exceeding the full pipeline of any other board. Against this, it is by a wide margin the most deterministic, with a CV of 0.01% across the repeated-inference series because its NPU runs at a fixed clock with no dynamic frequency scaling.

The Jetson is also highly stable at 0.50%, with its clocks locked to maximum during measurement. The Coral shows moderate variation at 3.53%, attributable to the inference output crossing the USB and the custom NMS operator running on the host CPU. The KV260 shows the highest variation at 6.39%.

Cold-start behavior, the cost of the first inference relative to steady state, differs sharply across platforms and matters for duty-cycled deployment. The penalty is smallest on the Jetson, where the first inference costs an additional 19.3% because the precompiled TensorRT engine reloads quickly. Coral has a penalty of 92.0%, as the Edge TPU cache initializes. On the KV260, the dominant start-up cost is model loading, which took 6.62 seconds before any inference ran, while the first-inference penalty within a session was small. The penalty is largest on the VIM3, where the first inference took roughly 9.3 seconds against a steady-state inference of about 1.37 seconds, because the NPU shader binary is loaded from disk on first use. Figure 5.1 summarizes the steady-state throughput.

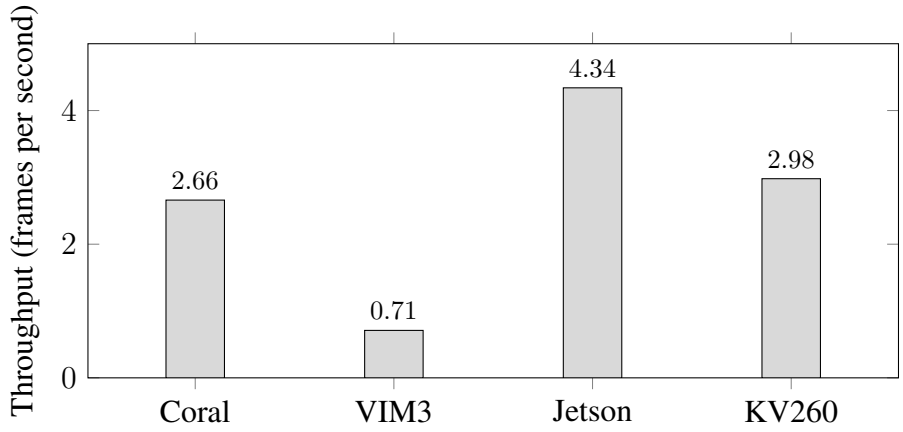


Figure 5.1: End-to-end throughput in FPS for each platform, computed as the reciprocal of the real-image end-to-end median latency. The Jetson Nano figure is for YOLOv4-416 and is comparable on speed but not on accuracy. Higher is better.

5.2 Detection Accuracy

Detection accuracy is reported as mAP at an IoU threshold of 0.50 (mAP_{50}). For the three platforms that ran EfficientDet-Lite2, accuracy was computed on the identical $4,993$-image BDD100K validation subset against the same ground truth, with a score threshold of 0.25 and the same evaluation logic. Table 5.2 reports the overall figure and the per-class AP.

Table 5.2: Per-class AP and overall mAP at an IoU threshold of 0.50 (mAP_{50}) on the BDD100K validation subset, for the three platforms that ran EfficientDet-Lite2. Ground-truth (GT) instance counts are shown. The Kria KV260 column is the host-CPU integer-equivalent graph, following Section 4.5.4.

Class	GT instances	Coral AP	VIM3 AP	KV260 (CPU) AP
car	50,001	0.5210	0.5209	0.5405
t_sign	17,194	0.2860	0.2853	0.2941
truck	1,948	0.3126	0.3128	0.3460
pedestrian	6,410	0.1948	0.1943	0.2076
tl_green	5,711	0.1678	0.1666	0.1721
tl_red	3,324	0.1190	0.1147	0.1199
tl_yellow	239	0.0000	0.0000	0.0000
mAP_{50}		**0.2288**	**0.2278**	**0.2400**

The three platforms cluster tightly, at 0.2288 on the Coral, 0.2278 on the VIM3, and 0.2400 on the KV260 host-CPU path, a spread of 0.012. The per-class figures agree closely as well, for example, a car AP between 0.52 and 0.54 on all three. This tight clustering is the central confirmation of the fairness control of the study. The minimal residual differences trace back to how each toolchain compiled and quantized the model, rather than to the hardware.

The per-class pattern follows class frequency. The car class, at 58.9% of all ground-truth instances, scores the highest. The rarer and smaller classes score lower, with pedestrians near 0.20 and the traffic lights between 0.11 and 0.18. The yellow traffic light was not detected above the score threshold on any platform, returning an AP of 0.0000. With only 239 ground-truth instances, 0.28% of the total, the training signal is insufficient for reliable detection. The identical result on the Coral and the VIM3 confirms that this is a limitation of the model and dataset rather than a property of any one platform. The

absolute accuracy near 0.23 reflects the difficulty of BDD100K and its class imbalance, and it remains consistent with the aim of the study, which is to characterize deployment constraints rather than to maximize accuracy.

The Jetson Nano achieved an mAP_{50} of 0.700 on COCO-val2017 with YOLOv4-416. This figure is reported separately and is not comparable to the values in Table 5.2, because it was produced by a different model on a different dataset with eighty classes rather than seven. It indexes the accuracy of a TensorRT half-precision deployment on this GPU and nothing more.

On the KV260, the model running on the DPU through the Quark INT8 path returned an mAP_{50} of only 0.0233, against the 0.2400 of the integer-equivalent graph on the host CPU. The collapse originates in the per-tensor, power-of-two quantization of the depthwise-separable classification head, the mechanism established in Section 2.5, and not in the FPGA's compute or in the network itself. Table 5.3 reports the calibration ablation. Enlarging the calibration set from 50 to 500 images raised the DPU accuracy only from 0.0000 to 0.0233, an order of magnitude below the baseline throughout, which shows the collapse to be structural rather than an artifact of calibration-set size. This toolchain-level behavior is examined further in Chapter 6.

Table 5.3: Effect of calibration-set size on Kria KV260 DPU accuracy, measured as mAP at an IoU threshold of 0.50 (mAP_{50}). The host-CPU integer-equivalent graph is shown for reference.

Configuration	mAP_{50}
DPU, 50-image calibration	0.0000
DPU, 200-image calibration	0.0216
DPU, 500-image calibration	0.0233
Host-CPU integer-equivalent graph	0.2400

5.3 Power and Energy

Power was measured at the board input during sustained inference and is reported as average power in watts. Energy per inference is the product of average inference power

and end-to-end latency, in joules. Table 5.4 reports both, together with throughput per watt.

Table 5.4: Idle and inference power in watts (W), throughput per watt in frames per second per watt (FPS/W), and energy per inference in joules (J) for each platform. Energy per inference is the product of inference power and end-to-end latency. The Jetson Nano figures are for YOLOv4-416. Power was measured with differing instruments, noted as a limitation in Section 5.6.

Platform	Idle (W)	Inference (W)	FPS/W	Energy/inference (J)
Coral Dev Board Mini	0.80	1.50	1.77	0.56
Khadas VIM3	2.40	2.73	0.26	3.84
Jetson Nano	1.95	6.66	0.65	1.54
Kria KV260	4.31	5.50	0.54	1.84

The Coral is the most efficient platform by a clear margin, at 1.77 FPS/W and 0.56 J per inference, the lowest energy of all four. Its fixed-function Edge TPU draws the least power during inference, 1.50 W, and its moderate frame rate is sufficient to provide the best efficiency. The VIM3 is the least efficient, at 0.26 FPS/W and 3.84 J per inference, roughly 6.8 times the Coral's energy, despite its NPU carrying a higher rated throughput of 5.0 TOPS against the Edge TPU's 4.0. This inversion shows that rated throughput is a poor predictor of delivered efficiency for this workload. The VIM3's very low frame rate dominates the energy calculation, and its small inference-power increment over idle, 0.33 W, does not compensate. The Jetson and the KV260 fall in between, at 0.65 and 0.54 FPS/W and 1.54 and 1.84 J per inference, respectively.

Energy per inference is the metric that compares the platforms fairly. The Jetson illustrates this. It draws 6.66 W during inference, roughly five times the Coral, yet its high throughput holds its energy per inference to 1.54 J, well below the VIM3's 3.84 J. The KV260 carries the highest idle power of the four, 4.31 W, which reflects the static power of the FPGA fabric and is incurred whether or not the accelerator is busy, a structural cost that an always-on FPGA deployment cannot avoid. Figure 5.2 summarizes energy per inference.

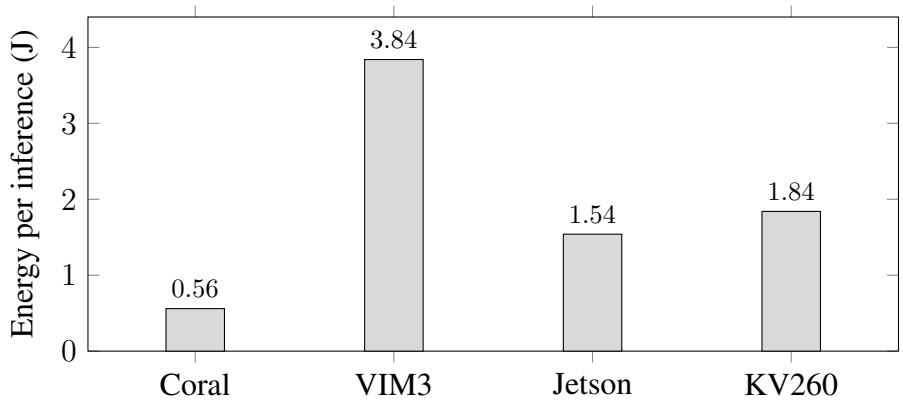


Figure 5.2: Energy per inference in joules (J) for each platform, computed as inference power times end-to-end latency. The Jetson Nano figure is for YOLOv4-416 and is comparable on energy but not on accuracy. Lower is better.

5.4 Price to Performance

The four platforms differ in purchase cost, so the comparison is completed by relating each board's price to its measured throughput.[1] Table 5.5 reports the cost per frame per second, where a lower figure is better.

The cost-per-frame ranking differs from the latency-based ranking. The Jetson Nano records the lowest cost per frame at $27.67\,€$. This figure is not directly comparable, as the platform is discontinued, its Manufacturer's Suggested Retail Price (MSRP) is sourced from third-party listings, and its throughput derives from the YOLOv4-416 substitute rather than the shared model. Setting the Jetson aside, the Coral Dev Board Mini is the most cost-efficient platform across the monetary metrics, with an $91.99\,€$ purchase price, $34.56\,€$ per frame, and $31.98\,€$ per year over three years. The VIM3 has a similar purchase price at $155.48\,€$, but its much lower throughput raises its cost per frame to $218.68\,€$, the highest of the four. The KV260 carries the highest purchase price at $229.08\,€$ and the highest three-year operating cost at $81.17\,€$ per year, while its cost per

[1] Device prices are sourced from official manufacturer listings (Coral [77], Khadas [78], AMD [79]) and were originally quoted in USD. Amounts are converted to EUR at the ECB euro reference rate prevailing at the time of data collection (May 2026, approximately 1 USD = 0.92 EUR). The Jetson Nano price is from third-party reseller listings as the product is discontinued.

frame of 76.85 € is second only to the VIM3. Rated accelerator throughput, therefore, does not reliably predict cost efficiency, since the VIM3 carries a higher nominal specification than the Coral yet has the highest cost per frame of the four.

Table 5.5: Hardware MSRP, end-to-end throughput in FPS, cost per frame per second, estimated annual electricity cost, three-year annualized operating cost, and market availability. MSRP values are approximate vendor list prices converted to € from USD; see footnote in the section text for sources and exchange-rate methodology. Annual electricity cost assumes continuous operation at inference power, computed as inference power $\div 1000 \times 0.10$ €/kWh. The three-year operating cost amortizes the MSRP over an assumed three-year service life and adds annual electricity. The Jetson Nano figure uses YOLOv4-416 throughput and indexes GPU capability rather than the shared task.

	Coral	VIM3	Jetson	KV260
MSRP (€)	∼91.99	∼155.48	∼120	∼229.08
Throughput (FPS)	2.66	0.71	4.34	2.98
Cost per FPS (€)	34.56	218.68	27.67	76.85
Electricity (€/yr)	1.32	2.39	5.83	4.81
3-yr operating cost (€/yr)	31.98	54.22	45.83	81.17
Availability	Available	Available	Discontinued	Available

From a cost-efficiency perspective, the Coral is the most efficient option, while the VIM3 remains competitive only on initial purchase cost. The KV260's higher cost reflects its reconfigurable FPGA fabric, which is not fully utilized by this workload. While platform-specific development timelines were not formally logged, the mature, well-documented toolchains of both the Coral and VIM3 compound their advantages over the more complex FPGA development life-cycle.

5.5 Cross-Platform Synthesis

The four platforms were chosen to span the flexibility-to-efficiency range of edge acceleration (Section 3.4), and the measurements reflect that spread. The fixed-function Edge TPU on the Coral is the most energy-efficient and cost-effective of the platforms that ran the shared model, and because the model was authored for it, its deployment was also the most straightforward. The reconfigurable FPGA on the KV260 delivers the lowest invoke latency but carries the largest host-side end-to-end overhead, the highest

purchase cost, the highest idle power, and a toolchain whose quantizer collapsed detection accuracy on the DPU. The NPU on the VIM3 is the most deterministic of the four; yet, it is an order of magnitude slower than the others and the least energy-efficient, despite the highest rated throughput. The GPU on the Jetson is the fastest on raw throughput but could not run the shared model at all and is represented by a substitute workload.

No single platform leads on every axis, which is the substantive answer to Research Question 1, posed in Section 1.1. For real-time traffic perception at the edge, where sustained throughput, energy, and cost weigh jointly, the fixed-function Edge TPU is the best-balanced of those evaluated. The detailed implications for selecting an accelerator are drawn together in Chapter 6.

5.6 Threats to Validity

Several limitations qualify these results. Power was measured with three different instruments: an ATORCH J7-C inline USB meter for the Coral and the VIM3, the on-board INA3221 read through `tegrastats` for the Jetson, and the on-board INA260 for the KV260. All measure power at the board input during sustained inference, which mitigates but does not eliminate the systematic uncertainty in the cross-platform power comparison. The invoke latencies are not strictly one-for-one, as set out in Section 5.1, because the KV260 DPU graph excludes the anchor decode and NMS that the Coral and VIM3 graphs include. The end-to-end figures, which apply the identical pipeline to every platform, therefore carry the cross-platform comparison, while the invoke figures indicate raw accelerator throughput only. The Jetson ran YOLOv4-416 on COCO rather than the shared model on BDD100K, a substitution forced by the conversion failure of Section 4.4.1, so its accuracy is not comparable, and its speed and energy reflect a heavier network. The KV260 latency and power come from the DPU, while its accuracy comes from the host CPU, a split justified because the DPU's quantizer, not the hardware, de-

grades the accuracy.

The fixed maximum of 25 detections per image, common to all shared-model platforms (Section 3.2), bounds recall in dense scenes equally across them, so it does not bias the comparison, although it does limit the absolute accuracy reported here. This accuracy is modest, with only the *car* class exceeding an AP of 0.5, and its robustness therefore warrants examination. Two properties of the measurements establish that the figures are a faithful reflection of the model and dataset. The first is cross-platform agreement: the per-class results in Section 5.2 coincide across three independently implemented inference stacks, the Edge TPU runtime on the Coral, the source-built TIM-VX delegate on the VIM3, and ONNX Runtime on the KV260 host CPU, to within approximately 0.03 AP on every class. These stacks share no inference or compilation code, so their convergence on a common per-class profile is a strong indicator that the training, export, and evaluation paths are implemented consistently. The second is internal structure: the accuracy varies monotonically with class frequency, the most frequent class scoring highest, while the rarest one, the yellow traffic light at 239 instances, goes undetected. This is the characteristic profile of class imbalance on a long-tailed dataset, and the magnitude is in line with the same architecture reaching roughly 0.2850 mAP at this IoU threshold on COCO under pretrained conditions, against the harder conditions of the BDD100K subset. Taken together, the figures are well explained by dataset imbalance and task difficulty, which is consistent with the study's aim of characterizing deployment behavior rather than maximizing accuracy. The agreement of the per-class accuracy across three independent platforms is itself a guaranty of the result's reproducibility.

Latency is summarized by the median over 500 runs or images, and the low CV on most platforms supports the stability of these medians, although the KV260's higher variation indicates more run-to-run spread. The calibration ablation spanned fifty to five hundred images, and the flat response across that range indicates a structural collapse rather than a calibration-size artifact, though larger calibration sets were not evaluated.

5.7 Summary

This chapter has reported benchmark results across latency, accuracy, power, and cost for four edge platforms. The three platforms that ran the shared model achieved near-identical detection accuracy, confirming the fairness of the experimental design. The yellow traffic light went undetected on every platform that ran the shared model, which is a limitation of the model and dataset rather than of the hardware.

6 Conclusion

This thesis evaluated the four principal paradigms of edge inference acceleration via a rigidly controlled object-detection benchmark. By deploying a shared INT8 EfficientDet-Lite2 detector across an Edge TPU, an Amlogic A311D NPU, a Maxwell-class GPU, and a KV260 FPGA-DPU, this thesis establishes a unified empirical comparison of latency, detection accuracy, power, energy per inference, and device cost. The three research questions posed in Section 1.1 are answered in turn below.

Research Question 1, which asks how the four paradigms compare under a common evaluation protocol, is answered by the cross-platform comparison of Chapter 5. No single paradigm leads on every axis. The fixed-function Edge TPU was the most energy-efficient, the most cost-effective of the platforms that ran the shared model, and the most straightforward to deploy. The FPGA-DPU achieved the lowest bare-inference latency but carried the largest host-side overhead, the highest idle power, and the highest cost, and its toolchain collapsed detection accuracy on the DPU. The NPU was, by a wide margin, the most deterministic platform; yet it was an order of magnitude slower than the others and the least energy-efficient. The GPU led on raw throughput but could not execute the shared model and was represented by a substitute workload. Detection accuracy clustered tightly across the three shared-model platforms, at mAP_{50} of 0.2288, 0.2278, and 0.2400, confirming that the comparison isolated the hardware and its toolchain rather than uncontrolled differences in the workload. The study thereby extends prior cross-platform benchmarks that compare fewer paradigms [13] or evaluate image classification

rather than a shared detector [7]. By holding a single detector and protocol fixed across all four paradigms using the traffic-domain BDD100K dataset, this benchmark ensures that the evaluative metrics index real-world driving-perception difficulties rather than generic object recognition.

Research Question 2, which asks whether an accelerator's specification-sheet metrics predict its real-world suitability, is answered in the negative. Rated throughput did not predict delivered performance. The platform with the higher rated figure, the NPU at 5.0 TOPS against the TPU's 4.0 TOPS, was the slowest and least energy-efficient of the four on this workload, while the TPU completed the inference call roughly 4.2 times faster and delivered roughly 6.8 times the throughput per watt. A peak arithmetic rating reveals nothing about how completely a toolchain maps a given model onto the accelerator, nor about how much computation falls back to the host. An accelerator for an edge perception task is therefore best selected based on measured end-to-end figures for the intended model class rather than on the single figure that vendors advertise.

Research Question 3, which asks what deployment constraints and toolchain limitations arise when a single detector is ported across paradigms whose software stacks were not designed around it, is answered by the deployment record of Chapter 4 and the results of Chapter 5. The detector could not be converted to a TensorRT engine on the Maxwell-class GPU through any of the four attempted routes, its custom NMS post-processing operator being unsupported by TensorRT 8.0. The FPGA toolchain's per-tensor power-of-two quantizer collapsed detection accuracy on the DPU, from mAP_{50} 0.2400 on the integer-equivalent host graph to 0.0233 on the accelerator. The sensitivity of depthwise-separable detection heads to coarse quantization is documented in the literature, so the phenomenon is not itself new. The contribution here is its isolation. A calibration ablation and a surgical bias bypass together identify the quantization of the classification-head weights as the dominant cause. Deployment also exposed routes absent from standard practice: a runtime-compiled path on the FPGA after the host-side flow rejected the

model, and the open-source TIM-VX delegate built from source on the NPU. The latter departs from prior work, which benchmarked this NPU through the closed vendor stack at INT16 arithmetic and reported latency only [14], whereas this study reports detection accuracy through the open delegate at INT8.

These conclusions are bounded by several limitations, which Chapter 5 treats in detail. Because the shared detector could not be deployed on the GPU, the Jetson Nano was measured on a substitute model evaluated on a general-purpose dataset; its accuracy is therefore not comparable to that of the shared-model platforms. Power was measured with three different instruments on a common basis of board-input measurement during sustained inference. On the FPGA, latency and power were taken from the model running on the DPU while accuracy was taken from the host CPU path. The deployment and quantization findings are specific to one detector family and its toolchains, and whether they generalize to other architectures remains an open question.

Taken together, the findings show that the questions that determine an edge deployment (whether a model can be deployed at all, at what accuracy, and at what energy per inference) are answered by measurements on the intended model and hardware, not by the specifications that nominally describe them. For real-time traffic perception at the edge, the fixed-function TPU was the best-balanced of the platforms evaluated, though no paradigm dominated across every axis.

6.1 Future Work

The findings open several directions. The accuracy collapse on the DPU invites QAT [66], or toolchain support for finer-grained quantization schemes, as routes to recovering detection accuracy on fixed-function integer hardware; establishing whether either suffices would turn the documented negative result into a deployable one. Broadening the study to further detector families and other platforms would test how far the present conclu-

sions generalize and whether the weak predictive value of rated throughput holds across workloads. The measured differences in throughput, efficiency, and cost also suggest a heterogeneous deployment in which inputs are routed between accelerators according to confidence or load, an architecture this work motivates but does not evaluate, and which would require its own power and latency accounting before it could be recommended. Finally, the open-source delegate path characterized here for the NPU could be evaluated on other models, extending the reproducible, vendor-independent route beyond the single detector studied here.

References

[1] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing", *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1738–1762, Aug. 2019, ISSN: 1558-2256. DOI: 10.1109/jproc.2019.2918951.

[2] J. M. Shalf and R. Leland, "Computing beyond moore's law", *Computer*, vol. 48, no. 12, pp. 14–23, 2015. DOI: 10.1109/MC.2015.374.

[3] M. M. Waldrop, "The chips are down for moore's law", *Nature*, vol. 530, no. 7589, pp. 144–147, Feb. 2016, ISSN: 1476-4687. DOI: 10.1038/530144a.

[4] M. Tan, R. Pang, and Q. V. Le, "Efficientdet: Scalable and efficient object detection", in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 10 778–10 787. DOI: 10.1109/CVPR42600.2020.01079.

[5] F. Yu et al., "Bdd100k: A diverse driving dataset for heterogeneous multitask learning", Jun. 2020, pp. 2633–2642. DOI: 10.1109/CVPR42600.2020.00271.

[6] V. Kamath and R. A., "Performance analysis of the pretrained efficientdet for real-time object detection on raspberry pi", in *2021 International Conference on Circuits, Controls and Communications (CCUBE)*, 2021, pp. 1–6. DOI: 10.1109/CCUBE53681.2021.9702741.

[7] S. P. Baller, A. Jindal, M. Chadha, and M. Gerndt, "Deepedgebench: Benchmarking deep neural networks on edge devices", in *2021 IEEE International Conference*

on Cloud Engineering (IC2E), 2021, pp. 20–30. DOI: 10.1109/IC2E52221.2021.00016.

[8] P. Kang and A. Somtham, "An evaluation of modern accelerator-based edge devices for object detection applications", *Mathematics*, vol. 10, p. 4299, Nov. 2022. DOI: 10.3390/math10224299.

[9] D. G. Lema, R. Usamentiaga, and D. F. García, "Quantitative comparison and performance evaluation of deep learning-based object detection models on edge computing devices", *Integration*, vol. 95, p. 102 127, 2024, ISSN: 0167-9260. DOI: 10.1016/j.vlsi.2023.102127.

[10] D. K. Alqahtani, M. A. Cheema, and A. N. Toosi, "Benchmarking deep learning models for object detection on edge computing devices", in *Service-Oriented Computing*, W. Gaaloul, M. Sheng, Q. Yu, and S. Yangui, Eds., Singapore: Springer Nature Singapore, 2025, pp. 142–150, ISBN: 978-981-96-0805-8. DOI: 10.1007/978-981-96-0805-8_11.

[11] D. Minott, S. Siddiqui, and R. J. Haddad, "Benchmarking edge ai platforms: Performance analysis of nvidia jetson and raspberry pi 5 with coral tpu", in *SoutheastCon 2025*, 2025, pp. 1384–1389. DOI: 10.1109/SoutheastCon56624.2025.10971592.

[12] R. Tobiasz, G. Wilczyński, P. Graszka, N. Czechowski, and S. Łuczak, *Edge devices inference performance comparison*, 2023. DOI: 10.48550/arXiv.2306.12093.

[13] S. C. Magalhães, F. N. d. Santos, P. Machado, A. P. Moreira, and J. Dias, "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. 116, p. 105 604, 2023. DOI: 10.1016/j.engappai.2022.105604.

[14] I. Lazarevich, M. Grimaldi, R. Kumar, S. Mitra, S. Khan, and S. Sah, *Yolobench: Benchmarking efficient object detectors on embedded systems*, 2023. DOI: 10.48550/arXiv.2307.13901.

[15] R. Cordova-Cardenas, D. Amor, and Á. Gutiérrez, "Edge ai in practice: A survey and deployment framework for neural networks on embedded systems", *Electronics*, vol. 14, no. 24, 2025. DOI: 10.3390/electronics14244877.

[16] C. Silvano et al., *A survey on deep learning hardware accelerators for heterogeneous hpc platforms*, 2023. DOI: 10.48550/arXiv.2306.15552.

[17] A. Bulut, F. Ozdemir, Y. S. Bostanci, and M. Soyturk, "Performance evaluation of recent object detection models for traffic safety applications on edge", in *Proceedings of the 2023 5th International Conference on Image Processing and Machine Vision*, ser. IPMV '23, Macau, China: Association for Computing Machinery, 2023, pp. 1–6, ISBN: 9781450397926. DOI: 10.1145/3582177.3582178.

[18] V. O. Castelló, I. S. Igual, O. del Tejo Catalá, and J.-C. Perez-Cortes, "High-profile vru detection on resource-constrained hardware using yolov3/v4 on bdd100k", *J. Imaging*, vol. 6, no. 12, p. 142, 2020. DOI: 10.3390/jimaging6120142.

[19] B. Jacob et al., "Quantization and training of neural networks for efficient integer-arithmetic-only inference", in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713. DOI: 10.1109/CVPR.2018.00286.

[20] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation", 2020. DOI: 10.48550/arXiv.2004.09602.

[21] M. Nagel, M. v. Baalen, T. Blankevoort, and M. Welling, "Data-free quantization through weight equalization and bias correction", in *Proceedings of the IEEE/CVF*

International Conference on Computer Vision (ICCV), Oct. 2019, pp. 1325–1334. DOI: 10.48550/arXiv.1906.04721.

[22] L. Niu, J. Liu, Z. Yuan, D. Yang, X. Wang, and W. Liu, "Improving post-training quantization on object detection with task loss-guided Lp metric", 2023. DOI: 10.48550/arXiv.2304.09785.

[23] D. Cantero, I. Esnaola-Gonzalez, J. Miguel-Alonso, and E. Jauregi, "Benchmarking object detection deep learning models in embedded devices", *Sensors*, vol. 22, no. 11, 2022, ISSN: 1424-8220. DOI: 10.3390/s22114205.

[24] C. Kachris, "A survey on hardware accelerators for large language models", *Applied Sciences*, vol. 15, no. 2, p. 586, Jan. 2025, ISSN: 2076-3417. DOI: 10.3390/app15020586.

[25] B. Peccerillo, M. Mannino, A. Mondelli, and S. Bartolini, "A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives", *Journal of Systems Architecture*, vol. 129, p. 102 561, Aug. 2022, ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2022.102561.

[26] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey", *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017, ISSN: 1558-2256. DOI: 10.1109/jproc.2017.2761740.

[27] M. Vaithianathan, M. Patil, S. Ng, and S. Udkar, "Comparative study of fpga and gpu for high-performance computing and ai", *ESP International Journal of Advancements in Computational Technology (ESP-IJACT)*, vol. 1, no. 1, pp. 37–46, May 2023, Accessed: June, 2026. DOI: 10.56472/25838628/IJACT-V1I1P107.

[28] J. L. Hennessy and D. A. Patterson, “A new golden age for computer architecture”, *Commun. ACM*, vol. 62, no. 2, pp. 48–60, Jan. 2019, ISSN: 0001-0782. DOI: 10.1145/3282307.

[29] J. Nickolls and W. J. Dally, “The gpu computing era”, *IEEE Micro*, vol. 30, no. 2, pp. 56–69, Mar. 2010, ISSN: 0272-1732. DOI: 10.1109/mm.2010.41.

[30] W. J. Dally, S. W. Keckler, and D. B. Kirk, “Evolution of the graphics processing unit (gpu)”, *IEEE Micro*, vol. 41, no. 6, pp. 42–51, Nov. 2021, ISSN: 1937-4143. DOI: 10.1109/mm.2021.3113475.

[31] T. M. Aamodt, W. W. Lun Fung, and T. G. Rogers, *General-Purpose Graphics Processor Architectures*. Springer International Publishing, 2018, ISBN: 9783031017599. DOI: 10.1007/978-3-031-01759-9.

[32] *Jetson Nano*, en, Accessed: May, 2025. [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano.

[33] N. P. Jouppi et al., *In-datacenter performance analysis of a tensor processing unit*, 2017. DOI: 10.48550/arXiv.1704.04760.

[34] W. J. Dally, Y. Turakhia, and S. Han, “Domain-specific hardware accelerators”, *Commun. ACM*, vol. 63, no. 7, pp. 48–57, Jun. 2020, ISSN: 0001-0782. DOI: 10.1145/3361682.

[35] Coral, *Edge TPU performance benchmarks*, https://coral.ai/docs/edgetpu/benchmarks/, Accessed: June, 2026, 2020.

[36] Khadas, *Khadas VIM3 product brief*, https://www.khadas.com/vim3, Accessed: June, 2026, 2020.

[37] I. Kuon, R. Tessier, and J. Rose, “Fpga architecture: Survey and challenges”, *Found. Trends Electron. Des. Autom.*, vol. 2, no. 2, pp. 135–253, Apr. 2008, ISSN: 1551-3939. DOI: 10.1561/1000000005.

[38] A. Habermaier and A. Knapp, "On the correctness of the simt execution model of gpus", in *Programming Languages and Systems*. Springer Berlin Heidelberg, 2012, pp. 316–335, ISBN: 9783642288692. DOI: 10.1007/978-3-642-28869-2_16.

[39] U. Farooq, Z. Marrakchi, and H. Mehrez, "Heterogeneous Architectures Exploration Environments", in *Tree-Based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization*, Springer, May 2012, pp. 85–122, ISBN: 978-1-4614-3593-8. DOI: 10.1007/978-1-4614-3594-5_4.

[40] Y. Al-Ameri, "FPGA-Based Hardware Accelerators for Deep Learning in Mobile Robotics", en, M.S. thesis, University of Turku, Department of Computing, 2024. [Online]. Available: https://www.utupub.fi/server/api/core/bitstreams/e1c26063-cce0-4b03-8947-e2c75275f917/content.

[41] *Product Details • Kria KV260 Vision AI Starter Kit Data Sheet (DS986) • Reader • AMD Technical Information Portal*, Accessed May, 2025. [Online]. Available: https://docs.amd.com/r/en-US/ds986-kv260-starter-kit/Product-Details.

[42] AMD Xilinx, *Vitis AI documentation*, https://xilinx.github.io/Vitis-AI/3.5/html/index.html, Version 3.5; Accessed: June, 2026, 2023.

[43] Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang, "The expressive power of neural networks: A view from the width", in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17, Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6232–6240, ISBN: 9781510860964. DOI: 10.48550/arXiv.1709.02540.

[44] J. Schmidhuber, "Deep learning in neural networks: An overview", 2014. DOI: 10.48550/ARXIV.1404.7828.

[45] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors", *Nature*, vol. 323, pp. 533–536, Oct. 1986. DOI: 10.1038/323533a0.

[46] S. S. Haykin, *Neural networks and learning machines*, en, 3. ed. New York Munich: Prentice-Hall, 2009, ISBN: 978-0-13-147139-9.

[47] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791.

[48] Y. Bengio, A. Courville, and P. Vincent, *Representation learning: A review and new perspectives*, 2012. DOI: 10.48550/ARXIV.1206.5538.

[49] A. G. Howard et al., "Mobilenets: Efficient convolutional neural networks for mobile vision applications", 2017. DOI: 10.48550/arXiv.1704.04861.

[50] Z. Li et al., "When object detection meets knowledge distillation: A survey", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 8, pp. 10 555–10 579, 2023. DOI: 10.1109/TPAMI.2023.3257546.

[51] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks", 2015. DOI: 10.48550/arXiv.1506.01497.

[52] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection", 2015. DOI: 10.48550/arXiv.1506.02640.

[53] W. Liu et al., "Ssd: Single shot multibox detector", 2015. DOI: 10.48550/arXiv.1512.02325.

[54] X. Zhou, D. Wang, and P. Krähenbühl, *Objects as points*, 2019. DOI: 10.48550/arXiv.1904.07850.

[55] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection", in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 936–944. DOI: 10.1109/CVPR.2017.106.

[56] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia, "Path aggregation network for instance segmentation", in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 8759–8768. DOI: 10.1109/CVPR.2018.00913.

[57] G. Ghiasi, T.-Y. Lin, and Q. V. Le, "Nas-fpn: Learning scalable feature pyramid architecture for object detection", in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 7029–7038. DOI: 10.1109/CVPR.2019.00720.

[58] Google TensorFlow, *Easier object detection on mobile with tensorflow lite*, https://blog.tensorflow.org/2021/06/easier-object-detection-on-mobile-with-tf-lite.html, Accessed: June, 2026, Jun. 2021.

[59] TensorFlow, *Efficientnet-lite*, https://github.com/tensorflow/tpu/tree/master/models/official/efficientnet/lite, 2026.

[60] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection", in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 2999–3007. DOI: 10.1109/ICCV.2017.324.

[61] Google AI Edge / MediaPipe, *EfficientDet-Lite model specifications*, https://developers.google.com/edge/mediapipe/solutions/vision/object_detector?utm_source=chatgpt.com, Accessed: June, 2026.

[62] AMD/Xilinx, *Developing a model — Vitis AI quantization*, https://xilinx.github.io/Vitis-AI/3.5/html/docs/workflow-model-development.html.

[63] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A white paper on neural network quantization", 2021. DOI: 10.48550/arXiv.2106.08295.

[64] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper", Google, Tech. Rep., 2018. DOI: 10.48550/arXiv.1806.08342.

[65] Advanced Micro Devices, *Full list of quantization configuration features — amd quark documentation*, https://quark.docs.amd.com/latest/onnx/appendix_full_quant_config_features.html, Accessed: June, 2026, 2025.

[66] S. R. Jain, A. Gural, M. Wu, and C. H. Dick, "Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks", in *Proceedings of Machine Learning and Systems*, 2020. DOI: 10.48550/arXiv.1903.08066.

[67] Coral, *TensorFlow models on the Edge TPU*, https://coral.ai/docs/edgetpu/models-intro/, Accessed: June, 2026, 2022.

[68] Coral, *Edge TPU Compiler*, https://coral.ai/docs/edgetpu/compiler/, Accessed: June, 2026, 2022.

[69] VeriSilicon, *TIM-VX: VeriSilicon tensor interface module for OpenVX*, https://github.com/VeriSilicon/TIM-VX, Accessed: June, 2026, 2021.

[70] NVIDIA, *NVIDIA TensorRT documentation*, https://docs.nvidia.com/deeplearning/tensorrt/, Accessed: June, 2026, 2021.

[71] T.-Y. Lin et al., "Microsoft coco: Common objects in context", vol. 8693, Apr. 2014, ISBN: 978-3-319-10601-4. DOI: 10.1007/978-3-319-10602-1_48.

[72] M. Everingham, L. Van Gool, C. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge", *International Journal of Computer Vision*, vol. 88, pp. 303–338, Jun. 2010. DOI: 10.1007/s11263-009-0275-4.

[73] Coral, *Run inference on the Edge TPU with Python*, https://coral.ai/docs/edgetpu/tflite-python/, Accessed: June, 2026, 2020.

[74] Google AI Edge, *Object detection with tensorflow lite model maker*, https://ai.google.dev/edge/litert/libraries/modify/object_detection, Accessed: June, 2026, 2025.

[75] ONNX Community, *tf2onnx: Convert TensorFlow, Keras, TensorFlow.js and TFLite models to ONNX*, https://github.com/onnx/tensorflow-onnx, Accessed: June, 2026, 2024.

[76] J. Jung, *tensorrt_demos: Tensorrt yolov3/yolov4 and other demos*, https://github.com/jkjung-avt/tensorrt_demos, Accessed: June, 2026, 2021.

[77] Coral. "Dev board mini". Accessed: June, 2026. [Online]. Available: https://coral.withgoogle.com/products/dev-board-mini.

[78] Khadas, *VIM3 Basic Single Board Computer*, Product page, Accessed: June, 2026, 2026. [Online]. Available: https://www.khadas.com/vim3.

[79] *Kria KV260 Vision AI Starter Kit*, Accessed: May, 2025. [Online]. Available: https://www.amd.com/en/products/system-on-modules/kria/k26/kv260-vision-starter-kit.html.