ECBA-MLI: Edge Computing Benchmark Architecture for Machine Learning Inference

Conference Paper · July 2022

DOI: 10.1109/EDGE55608.2022.00016

4 authors:

Mathias Schneider
OTH Amberg-Weiden
20 PUBLICATIONS **39** CITATIONS
SEE PROFILE

Ruben Prokscha
OTH Amberg-Weiden
7 PUBLICATIONS **4** CITATIONS
SEE PROFILE

Seifeddine Saadani
OTH Amberg-Weiden
8 PUBLICATIONS **5** CITATIONS
SEE PROFILE

Alfred Hoess
OTH Amberg-Weiden
36 PUBLICATIONS **122** CITATIONS
SEE PROFILE

ECBA-MLI: Edge Computing Benchmark Architecture for Machine Learning Inference

Mathias Schneider, Ruben Prokscha, Seifeddine Saadani, and Alfred Höß
Department of Electrical Engineering, Media Technology and Computer Sciences,
Ostbayerische Technische Hochschule
Amberg-Weiden
92224 Amberg, Germany
Email: {mat.schneider, r.prokscha, se.saadani, a.hoess}@oth-aw.de

Abstract—**Recent developments in Artificial Intelligence (AI) research enable new strategies for running Machine Learning (ML) models. Evaluating application data on a remote server used to be common practice. However, novel AI accelerators herald a paradigm shift by moving the inference step from the cloud closer to the application in the edge. This approach increases service availability while significantly reducing latency. Nevertheless, choosing the right target platform and model for inference is a challenge that depends on the use case and its non-functional requirements. In this work, we present an *Edge Computing Benchmark Architecture for Machine Learning Inference (ECBA-MLI)* which provides a universal, reproducible, and comparable solution for evaluating non-functional criteria such as latency and energy consumption for edge deployment scenarios. It further evaluates results for six state-of-art object detection models deployed on twenty-one different platform configurations.**

Index Terms—**Edge Computing, Benchmark, Machine Learning, Power Consumption, Inference**

I. Introduction

With the emergence of ubiquitously deployed smart sensors in the last decade, collecting and processing decentralized *Big Data* in a centralized manner was the common approach [1]. This tendency was further amplified by the progress of ML and in particular deep learning. More refined and sophisticated network architectures offer higher accuracy, but imply more demanding resource requirements. Accordingly, deployment often happened on computationally powerful platforms, exacerbating centralized architectures. Nevertheless, a push towards the edge is noticeable in recent years, mostly driven by application requirements, such as low-latency processing, high mobility with unreliable connection, and data privacy [2]. Accordingly, hardware vendors designed and marketed a variety of energy-efficient platforms, meeting common edge requirements to unlock the full potential of decentralized ML. Harnessing specialized accelerator hardware, ML inference at the edge is now tangible [3], and its potential is already evaluated in use cases such as Intelligent Transport Systems [4].

However, selecting a suitable edge device is still a challenging design decision involving multiple criteria and trade-offs [5]. Especially for ML, estimating the metrics of a model running on an edge device a priori is challenging, even if all static model and platform attributes (e.g., number of operation and processor frequency) are available. Thus, benchmarking is still a reliable indicator to determine metrics such as latency and power consumption empirically. Since a fair comparison between different scenarios requires a concise setup and evaluation approach, we introduce *Edge Computing Benchmark Architecture for Machine Learning Inference (ECBA-MLI)*. Our architecture targets to collect benchmarks for various model and platforms combinations in a unified way and allows to generate metrics for an objective comparison between different deployment scenarios. Thus, it supports developers during their design phase and could further enable new approaches for task offloading strategies in heterogeneous, collaborative edge networks.

The paper is structured as follows. In Section II, conventional concepts and implementations for edge computing benchmarks are presented. Subsequently, we define the scope of our benchmark comprising heterogeneous platforms in Section III. Section IV introduces a variety of state-of-the-art deep object detection models as well as an unified architecture for benchmarking edge inference. Based on this setup, we elaborate our benchmark architecture, conduct measurements, and apply our evaluation concept. Finally, a conclusion summarizes our findings and proposes potential future research.

II. Related Work

The *MLCommons* association addressed the demand for an independent and reliable ML benchmark for processing units, since vendor benchmarks for inference and training were often difficult to compare, resulting in MLPerf [8, 9]. However, due to the relative novelty of embedded hardware accelerators, and their continuous development of the accompanying software frameworks, the first implementation prioritized training and inference deployment scenarios on high performance computers. The recently introduced *MLPerf Tiny* [10] aims

PREPRINT - This paper has been accepted at the IEEE International Conference on Edge Computing (EDGE) 2022.

	RaspberryPi 4 Model B + Intel Movidius NCS2	Google Coral Dev Board	NVIDIA Jetson Nano	NVIDIA Jetson TX2	NVIDIA Jetson AGX Xavier
CPU	Quad-Core ARM Cortex A72	Quad-Core ARM Cortex A53	Quad-Core ARM Cortex A57 MPCore	Dual-Core NVIDIA Denver 1.5 + Quad-Core ARM Cortex A57 MPCore	8-core NVIDIA Carmel ARMv8
RAM	4 GB LPDDR4	1 GB LPDDR4	4 GB LPDDR4	8 GB LPDDR4	32 GB LPDDR4x
AI-Chip	Intel Movidius Myriad X (VPU)	Google Edge Coprocessor (TPU)	128 CUDA Cores (Maxwell GPU)	256 CUDA Cores (Pascal GPU)	512 CUDA Cores 64 Tensor Cores (Volta GPU)
Interface	USB 3.0	PCIe	PCIe	PCIe	PCIe
OPs	4 TOPs	4 TOPs	472 GFLOPs	1.33 TFLOPs	32 TOPs
TDP	7.6 W + 2 W (NCS2)	3.5 W + 2 W (TPU)	5 W/10 W	7.5 W/15 W	10 W/15 W/30 W
Toolkit	OpenVino	TensorFlow Lite	CUDA / TensorRT	CUDA / TensorRT	CUDA / TensorRT

TABLE I: Edge devices hardware specifications [2, 6, 7].

to fill this gap by benchmarking low-end embedded devices. All measurements fulfill a high quality standard and integrity, however the number of ML models is limited to representatives from certain domains, comprising image classification and segmentation, object detection, language processing, and anomaly detection.

Another promising approach was introduced in a paper by Nokia Bell Labs [11]. They tested a variety of edge platforms in combination with common model architectures for motion detection, audio processing and image recognition. Their benchmark monitored various metrics, including execution time, memory utilization and energy consumption. For evaluation, the measurements were divided into the three phases: model load, warm up and inference. By providing a benchmark of eight different models with a wide range of parameters and structures, the authors offer an opportunity to estimate how other models might perform on these platforms in future work.

Including the previous references, Varghese et al. provide a broader view on the state-of-art development in edge computing benchmarks and the relations to former benchmarks applicable in the field of high performance and cloud computing [12]. In their survey, a range of edge performance benchmarks were investigated. Besides the valuable representation of benchmark commonalities and their setups, the authors highlight observations and use them to shape a vision for the future research directions. These involve the following recommendations: Integration of measures for accelerators, inclusion of energy consumption as a criteria, and the usage of virtualization techniques for deployment of the benchmarks.

Following these proposals, this work presents a benchmark utility specifically designed to deal with hardware accelerators in embedded environments. The main focus is set on having a modular architecture, which can easily be extended by additional devices in the edge and adapted for a specific deployment scenario. Furthermore, our approach supports monitoring and benchmarking multiple devices in parallel. This will enable further research concerning real-time task scheduling and placement of entire data processing pipelines in edge and fog clusters.

III. Hardware Setup

In recent years, different vendors, i.a. Intel, Google and NVIDIA, developed specialized hardware solutions to realize accelerated and energy-efficient edge computing. This section introduces these platforms, as well as the incorporated accelerator hardware used in the preliminary test setup. An overview of all key specifications is provided by Table I.

A. Edge Devices and Accelerators

Running model inference efficiently on edge devices entails a set of highly optimized hardware and software components. Therefore, all vendors introduced coprocessors for ML tasks, which are connected via a high bandwidth interface to a carrier platform. The hardware is optimized for generating high Operations Per Second (OPs) while maintaining a low Thermal Design Power (TDP). This makes them ideal for edge processing, where the energy supply is often limited. Each kind of coprocessor implies the usage of a toolkit for mapping the model efficiently to the respective hardware architecture.

1) Intel Neural Compute Stick 2 (NCS2): Based on the Intel Movidus Myriad X-Vision Processing Unit (VPU) chip Architecture, the computing stick accelerates the inference of ML models. NCS2 has to be connected to a carrier platform via Universal Serial Bus (USB) and harnesses the Open Visual Inference and Neural network Optimization (OpenVINO) toolkit, providing its model optimizer for deployment and the *Myriad* inference engine for execution.

2) Google Coral: Google's solution uses a Tensor Processing Unit (TPU) which is carried out for various interfaces [13]. Besides different developer boards, a USB dongle variant, and adapter cards with mini Peripheral Component Interconnect Express (PCIe) and M.2 are available. Inference requires INT8 quantised Tensorflow Lite models which are further optimized by Google's *Edge TPU Compiler* for deployment.

3) NVIDIA Jetson: NVIDIA's approach for addressing edge inference is coupling an ARM CPU with an embedded version of their GPUs. The Jetson Xavier has additional tensor cores, which are able to accelerate 16 bit floating point operations. Currently tensor cores are part of NVIDIA's Turing, Volta and Ampere architecture. The Jetson Nano and TX2 utilizing previous GPU architectures are limited to Compute Unified Device Architecture (CUDA) units for inference. There are two common approaches to run a model on the embedded GPU. The first one is to use a ML framework leveraging CUDA instructions for GPU acceleration. The second way uses the TensorRT (TRT) SDK for model optimization. By applying TRT, the model is translated to *engines*, which are highly optimized to perform on the target platform. All platforms are tested with *Jetpack 4.6* installed, their respective maximum specified power mode, and with programmatically active fan cooling set to 50 %.

B. Power Monitor

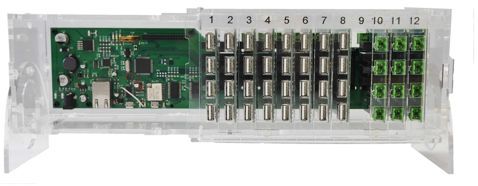


Fig. 1: Modular power monitor prototype for up to 48 devices.

Power consumption is a critical metric for edge inference. Especially battery driven setups require a low energy footprint. Thus, it is important to make a sensible comparison of the different devices in this regard. For this benchmark a custom power monitor is developed as depicted in Figure 1, empowered by a *STM32H750VB* MCU. Twelve module slots are connected via I²C which enable non-evasive measurements of current and supply voltage with a frequency of up to 50 Hz using INA220 sensors for up to four devices each. Modules can provide different voltage levels, depending on the external power supply, for monitoring heterogeneous clusters.

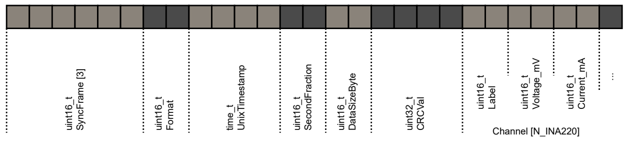


Fig. 2: PMS data frame structure.

Data packages are broadcasted after each cycle via Universal Asynchronous Receiver Transmitter (UART) in a compact format, as shown in Figure 2. The first three bytes are for synchronization purposes, while the format field enables versioning for detecting potential future firmware updates. Time stamps for each frame consists for the Unix Epoch Time and a fraction of a second (16 bit), which allows a maximum resolution of about 15 µs. As data payload, each frame includes a list of probes comprising sensor channel label, voltage, and current.

Consecutive data buffering and processing is realized by an external HTTP service. This service is hosted on a Raspberry Pi 3 Model B+ and receives measurements via UART from the power monitor. The web server provides a minimal interface to start, stop, and download the measurements for a channel. Recorded messages are validated using a Cyclic Redundancy Check (CRC). After finishing a measurement interval indicated by start and stop signals, data can be downloaded in JSON format.

IV. Software Setup

In order to distribute various ML tasks to a heterogeneous multiplatform appliance, several steps need to be taken into account. Figure 3 illustrates the processing steps for running device specific inference. The deployment of the benchmark is realized using Docker container virtualization shipping all required runtime libraries in distinct versions as indicated in the illustration. This creates a consistent environment, which can easily be modified and deployed in a multi-tenancy manner. It further allows to measure improvements introduced in updated runtime libraries.

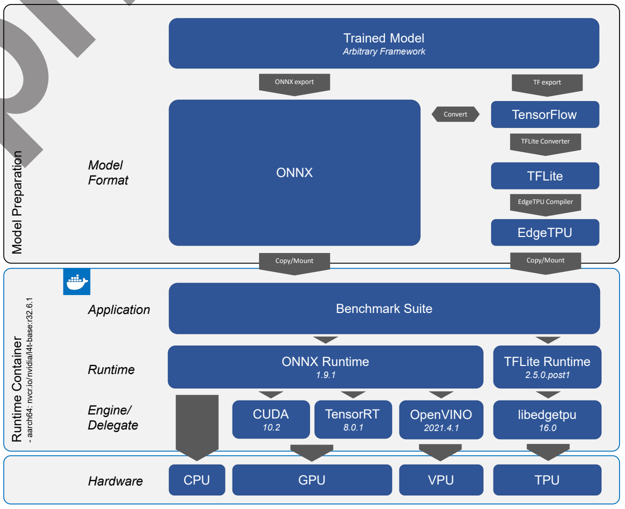


Fig. 3: Model conversion and deployment.

Execution of a ML tasks requires different base models, which are fetched from a model repository. The model accuracy is improved by utilizing transfer learning with an application specific dataset. To establish a unified interface for the subsequent steps, the models are converted beforehand into predefined frameworks.

A. Models

Table II lists object detection models used in this benchmark. They are selected due to their architecture, input size and layer depth.

	SSDMobileNetv2	TinyYolov3	TinyYolov4	Yolov4	Yolov5s	Yolov5l
ONNX	70.30	35.41	24.28	257.48	29.90	189.72
TFlite	6.95	9.45	6.92	65.65	10.68	53.57

TABLE II: Model candidate file size [MB].

1) SSDMobileNetv2: As the name suggests MobileNetv2 serves as the lightweight backbone for the object detector [14]. The base model for the ONNX conversion was taken from the TF1 object detection model zoo [15]. Since this base model could not be properly converted to TFLite, a model with the same label was taken from the Coral Edge TPU model zoo [16]. Both models have their input size set to 300 pixels squared.

2) TinyYolov3: The third adaptation of You Only Look Once (YOLO) [17] is widely used for object detection tasks. The smaller version (TinyYOLO) of the network is part of the model candidates. The model weights were taken from the original paper implementation and transferred to a TF-Keras model according to [18]. Afterwards models are converted to ONNX using *tf2onnx* [19], and to TFlite using the Tensorflow converter module applying post-training quantization. Additional, improvements, as All models have squared input size of 416 pixels.

3) YoloV4 and TinyYolov4: YOLOv4 [20] is the newest iteration from the original YOLO developers. It uses various micro optimizations in the model design and the training process to outperform existing models in both speed and accuracy. The benchmark considers both the full implementation, which was obtained from repository [21], and the tiny version from repository [22]. It is worth mentioning that the developers of the tiny model made additional modifications to the model to perform better on the Edge TPU. The models have a perception field of 300 pixels squared for the full model and 608 pixels squared for the tiny variant.

4) YoloV5s and YoloV5l: Contrary to the name, YOLOv5 is not related to original YOLO project, but is published by Ultralytics LLC [23]. YOLOv5 (v4.0) comes in four different versions, compromising between size and accuracy. The models are implemented in PyTorch. Scripts are available to export the trained model to ONNX. Additional effort is required to convert this model to TFLite via TF. The benchmark evaluates the small (*s*) and large (*l*) model. Both using an input size of 640 pixels squared.

B. Inter Framework Conversion

Due to the large number of ML frameworks, a unified architecture for inference should be employed. This requirement is addressed by leveraging the ONNX framework [24]. It enables the exchange of models between frameworks with the benefit of a common interface and format. An ONNX model can be used for inference by means of the ONNX runtime [25], allowing hardware vendors to implement standardized ONNX operations for their respective accelerator.

Since training models using the ONNX runtime is in an experimental stage, model conversion is performed after the training step. Therefore it comes with the disadvantage, that every framework has to be present for training various different models. Accordingly, the preparation for selecting and converting models for the benchmark is high and will be addressed in future benchmarks. Thereby, the goal will be to utilize solely pretrained model, reducing the necessity for additional modifications as described in Section IV-A.

As illustrated in Figure 3, model inference for TPU requires a separate branch in this pipeline. Therefore, it is necessary to transfer the model to Tensorflow and subsequently TFLite, before it is compiled by the EdgeTPU compiler. Within this step, additional quantization is performed and the mappings of operations to CPU and TPU are computed. The model is finally inferred using the TFLite Runtime in combination with the EdgeTPU delegate, which implements the interface to the TPU accelerator.

V. Edge Benchmark

Based on the introduced hardware and software setup, architecture and implementation of the benchmark suite are elaborated in this section. *ECBA-MLI's* design emphasizes on scaling and replicability to measure non-functional key figures in an stable environment. Further, we will present an evaluation strategy, harnessing obtained measurements to create a comparable and quantified characterization of a certain deployment.

A. Architecture and Implementation

Figure 4 provides a simplified overview of the implemented class architecture of the benchmark suite. It is built around the *Scenario* class, which marks the entry point for the benchmark. This class implements the concrete benchmark function and consolidates multiple measurement of the consecutive processing steps within the scenario. The benchmark suite differentiates scenarios:

- *Idle Consumption:* Device running without any additional load. This scenario allows to evaluate the net power consumption for a specific model on different devices.
- *Model Warm-up:* Model load from disk and creation of runtime session object, as well as a first inference on a single input sample. In this scenario overhead of running a model is determined. Since some model preparations are conducted during the very first inference of the model, it is beneficial to incorporate this measurement in the warm-up. It further avoids to distort measurements during the inference phase.

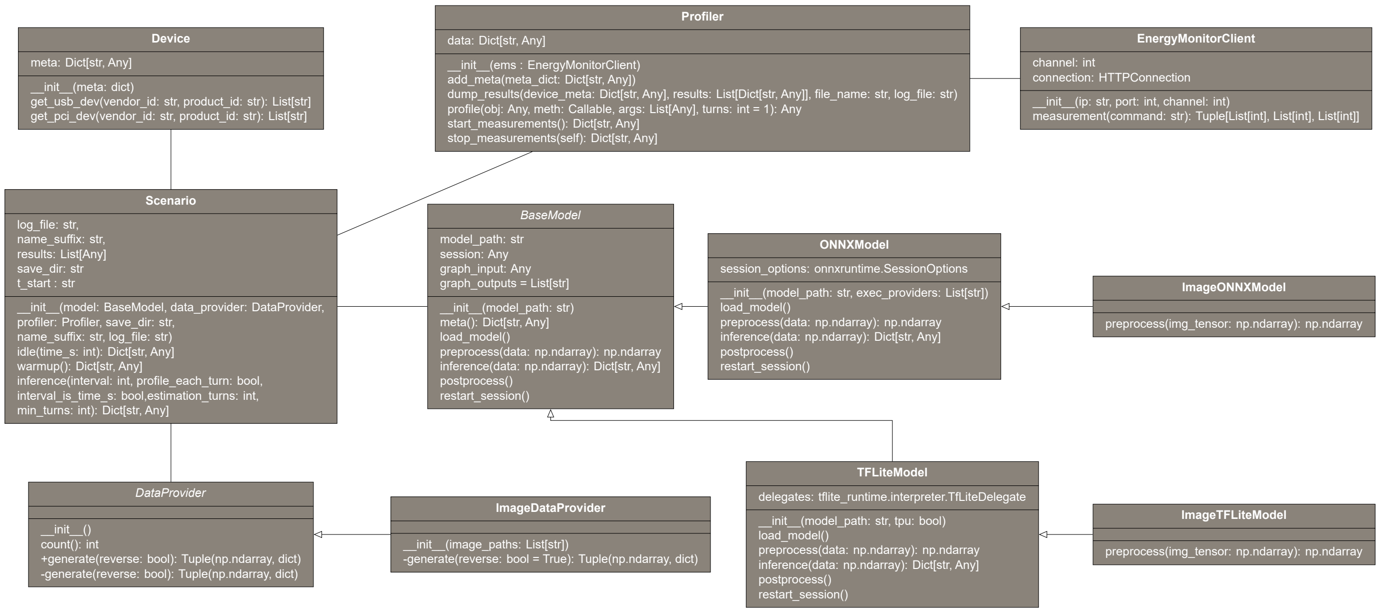

Device
meta: Dict[str, Any]
__init__(meta: dict)
get_usb_dev(vendor_id: str, product_id: str): List[str]
get_pci_dev(vendor_id: str, product_id: str): List[str]
Profiler
data: Dict[str, Any]
__init__(ems : EnergyMonitorClient)
add_meta(meta_dict: Dict[str, Any])
dump_results(device_meta: Dict[str, Any], results: List[Dict[str, Any]], file_name: str, log_file: str)
profile(obj: Any, meth: Callable, args: List[Any], turns: int = 1): Any
start_measurements(): Dict[str, Any]
stop_measurements(self): Dict[str, Any]
EnergyMonitorClient
channel: int
connection: HTTPConnection
__init__(ip: str, port: int, channel: int)
measurement(command: str): Tuple[List[int], List[int], List[int]]
Scenario
log_file: str,
name_suffix: str,
results: List[Any]
save_dir: str
t_start : str
__init__(model: BaseModel, data_provider: DataProvider,
profiler: Profiler, save_dir: str,
name_suffix: str, log_file: str)
idle(time_s: int): Dict[str, Any]
warmup(): Dict[str, Any]
inference(interval: int, profile_each_turn: bool,
interval_is_time_s: bool,estimation_turns: int,
min_turns: int): Dict[str, Any]
BaseModel
model_path: str
session: Any
graph_input: Any
graph_outputs = List[str]
__init__(model_path: str)
meta(): Dict[str, Any]
load_model()
preprocess(data: np.ndarray): np.ndarray
inference(data: np.ndarray): Dict[str, Any]
postprocess()
restart_session()
ONNXModel
session_options: onnxruntime.SessionOptions
__init__(model_path: str, exec_providers: List[str])
load_model()
preprocess(data: np.ndarray): np.ndarray
inference(data: np.ndarray): Dict[str, Any]
postprocess()
restart_session()
ImageONNXModel
preprocess(img_tensor: np.ndarray): np.ndarray
DataProvider
__init__()
count(): int
+generate(reverse: bool): Tuple(np.ndarray, dict)
-generate(reverse: bool): Tuple(np.ndarray, dict)
ImageDataProvider
__init__(image_paths: List[str])
-generate(reverse: bool = True): Tuple(np.ndarray, dict)
TFLiteModel
delegates: tflite_runtime.interpreter.TfLiteDelegate
__init__(model_path: str, tpu: bool)
load_model()
preprocess(data: np.ndarray): np.ndarray
inference(data: np.ndarray): Dict[str, Any]
postprocess()
restart_session()
ImageTFLiteModel
preprocess(img_tensor: np.ndarray): np.ndarray

Fig. 4: Benchmark framework class diagram.

- *Model Inference:* Multiple iterations of single input inference. In this scenario the stable, productive phase of the model is measured by running the single input inference with the same input data multiple times.

Benchmark results are enriched by meta information which is crucial feature of the benchmark suite, since they ensure replicability and comprehensibility of the tests. They are collected by multiple classes at different stages. This includes precise information about the device and its software configuration and is encapsulated in the *Device* class. Besides static information such as software versions of runtime dependent libraries, e.g. the installed NVIDIA driver version, it provides more dynamic information, e.g. whether accelerators connected via USB or PCI are available.

The *BaseModel* class provides interfaces for data processing and model inference. The vast variety of embedded accelerators require usage of different ML frameworks, depending on the platform in use. It is therefore beneficial to establish an unified interface which other modules can interact with as introduced in Section IV. For both runtimes, a subclass is implemented with a mandatory set of instructions. This set of methods includes model loading, data preprocessing, inference and postprocessing. The model class is injected with a separated *DataProvider* offering input data required for the respective model type under test.

A dedicated *Profiler* class wraps and monitors methods under test. The current implementation measures call duration and power measurements for the test interval. Latter implements client-server communication with the power monitoring service, executing the respective commands remotely and returning the returning the results. For the future, these measures could be further enriched, e.g. by recording additional device specific information such as CPU or RAM load.

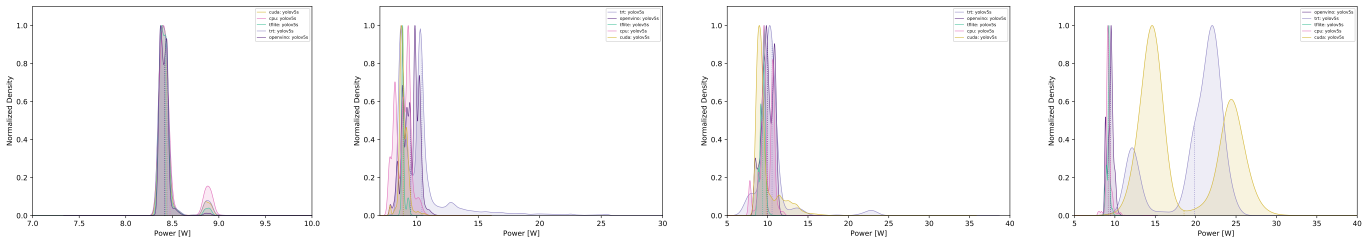
B. Measurements

The following formulas are utilized to calculate metrics used on the raw samples. Thereby, the subscript denotes the benchmark phase, respectively idle 0, load l, warmup inference w, and inference i. Equation (1) determines the average power consumption based on N samples of voltage $U(t)$ and current $I(t)$ recorded by the PMS.

$$\overline{P} = \frac{1}{N}\sum_{k=1}^{N} U(k)\cdot I(k) \tag{1}$$

Determining the execution time of an operation is conducted by collecting start and stop period. In particular, for processes that might be faster than the power measurement cycle, e.g., model inference, the operation is repeated r times to compute the average duration. As represented by Equation (2), a slight improvement is introduced by running test inferences prior to the actual measurement. This initial average probe duration $\overline{\tau}_{test}$ allows to estimate a proper number of repetitions r based on a given maximal test duration T_{max}. r has to exceed a minimum number of repetition r_{min} to ensure statistical significance. For our benchmark configuration T_{max} is set to 60 s and r_{min} to 100.

$$\tau = \frac{(t_N - t_1)}{r}, \text{ s.t. } \begin{cases} r = max\left(\left\lceil \frac{T_{max}}{\overline{\tau}_{test}} \right\rceil, r_{min}\right), & \text{for inference} \\ r = 1, \text{ else} \end{cases} \tag{2}$$



(a) Idle (b) Load (c) Warmup (d) Inference

Fig. 5: KDE plots per phase for YoloV5s benchmark measurements on NVIDIA Jetson Xavier AGX.

Name	RaspberryPi 4B	Coral Dev Board	NVIDIA Jetson Nano	NVIDIA Jetson TX2	NVIDIA Jetson AGX Xavier
$\overline{P}_{0,min}$	5.174	4.179	3.582	5.414	8.415
$\overline{P}_{0,mean}$	5.177	4.210	3.589	5.425	8.434
$\overline{P}_{0,max}$	5.179	4.243	3.597	5.440	8.488

TABLE III: $\overline{P}_0$ [W] range for platforms.

Based on these previous intermediate results, the consumed energy is calculated. Whereas Equation (3) is essential to estimate the overall platform energy consumption, Equation (4) allows to determine whether offloading a specific task to another already running device is more energy efficient. However, Equation (4) does not yet include wire resistance, which is essential when comparing two different devices. For the measured cases $\overline{P}$ is less than $\overline{P}_{0,min}$, $\Delta\overline{E}$ is corrected to 0.

$$\overline{E} = \overline{P} \cdot \tau \tag{3}$$

$$\Delta\overline{E} = \left(\overline{P} - \overline{P}_{0,min}\right) \cdot \tau \tag{4}$$

In the following paragraphs, the results of a benchmark run on all platforms is presented and a subset of the observed findings is discussed.

1) Idle Consumption: Idle power consumption of all devices is measured by running a sleep method for $\tau_0 = 100\,$s. This measurements include connected peripheral AI accelerators, whenever necessary. Accordingly, all carrier platforms are connected to NCS2 and EdgeTPU via USB, whereas the Google Coral Dev board has solely the NCS2 attached, since it uses the integrated TPU via PCIe for accelerated inference.

Figure 5a illustrates the power consumption Kernel-Density Estimates (KDEs). The KDEs are normalized to unity for their respective maximum peak to increase comparability. This representation will also be used to visualize other measurements in this work. Since the load method is independent of the hardware architecture, means of each distribution, indicated in dotted vertical lines, are nearly identical and the distribution's variance in the samples is low. A summary of the idle power consumption metrics for all platforms can be found in Table III.

2) Model Warm-up - Load: As part of the *Model Warm-up* scenario, load time for various models and their power consumption as presented in Figure 5b are recorded. This plot illustrates general characteristics that are also prominent for other platforms: model loaded to CPU require less power, followed by VPU and GPU-CUDA. In contrast, GPU-TRT has a higher variance with multiple peaks. These different power levels are presumably due to the initial computations to build the TRT engine for the loaded model. This online preparation of the engine results in large duration even on powerful platforms. Thus, it is recommended to activate the TRT engine cache, which is disabled during the benchmarks.

Detailed measurements for all deployable configurations are presented in Table IV. It summarizes latency and energy consumption for all measurements. As indicated, not all combinations can be executed, e.g. SSDMobileNetv2 cannot be loaded using the GPU-TRT provider, since building of the TRT engine fails. This behavior also applies to the Yolov4 model on the NVIDIA Jetson Nano. In contrast to other platform in this product group, the Jetson Nano is equipped with less memory which probably causes the failing TRT engine build. Similarly, on the Coral Dev Board, larger models are not supported by the VPU. Arguably, this as well can be caused by exceeding the lower memory constraints on this board (1 GB) in comparison to the other platforms, which is requisite for model deployment preparation using the OpenVINO Myriad engine provider implementation of the ONNX runtime.

3) Model Warm-up - Inference: During the warm-up scenario, the benchmark obtains measurements related to the first model inference. For all inference tests, the same image is used. As captured in Table V and Table VI, warm-up latency τ_w exceeds τ_i for most measurements. This observation is caused by lazy loading implementation for some processor deployment, for which additional preparation and memory allocation is conducted during the first inference instead of the loading phase. Whereas this behavior is less prominent for CPU inference, especially GPU-CUDA latency is increased by a significant margin in comparison with consecutive inferences due to on-going GPU memory allocation. This

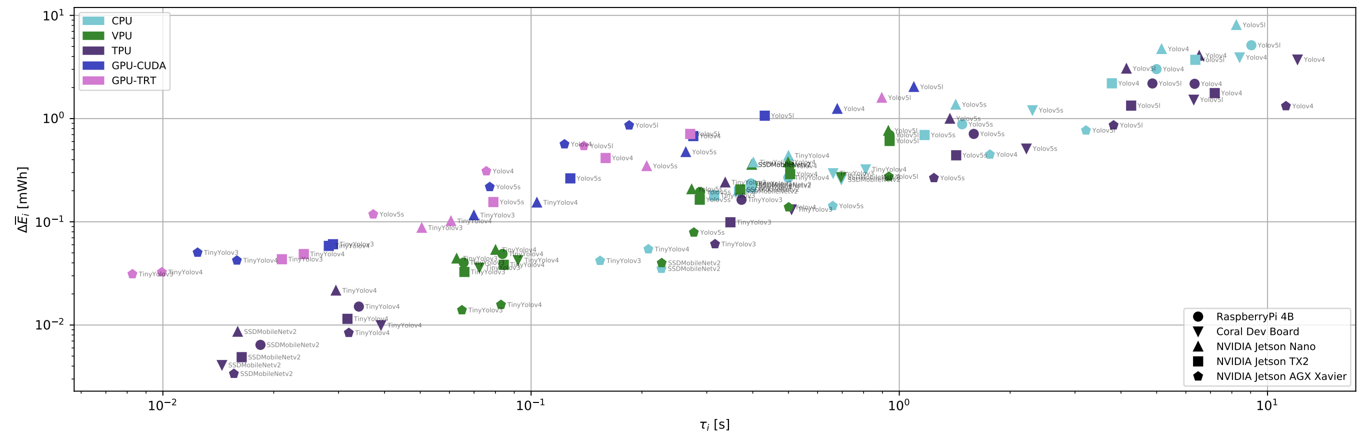


Fig. 6: Benchmark summary showcasing time spent and the necessary average net energy consumption per inference. Measurements include all possible deployments, combining carrier platform and available inference engines.

effect is further featured by the KDE depicted in Figure 5c, presenting an overall higher variance in the power consumption. It is worth noting that the TinyYolo4 using GPU-TRT measurement τ_w exceeds τ_l, which demonstrates the opposite behavior than the other models. It is presumably caused by TRT internal implementation details, since this effect occurs on all three GPU carrier platforms.

4) Inference: Several observation can be deduced by the results in Table VI and their visual representation in Figure 6. For models that are completely mapped to the TPU, including the SSDMobileNetv2 and TinyYolov4, τ_i is constant for all carrier platforms independent of their CPU. However, as soon as part of the models are partially executed on the CPU as indicated by the model structure after the EdgeTPU compiler step, its clock speed evidently becomes relevant for the overall latency. This underlines the importance of model optimization iterations to maximize computation executed on the TPU and is reflected by the high overall variance in latency and energy consumption.

Contrary effects can be examined for the VPU. In the case of the SSDMobileNetv2 model, inference times for VPU and CPU are nearly identical. Accordingly, it can be assumed that the model execution is automatically falling back to CPU. All other models however, require the same amount of time for VPU accelerated inference, independent of the respective carrier platform. This indicates that models are either completely inferred on the VPU or executed on CPU, implemented by the ONNX runtime explicitly configured to utilize the OpenVINO Myriad engine.

Whereas the SSDMobileNetv2, optimized for TPU, performs best on the TPU with up to 60 FPS, both TinyYolov3 and TinyYolov4 result in more than 10 FPS for all accelerated deployments (excluding TinyYolov3 on the TPU). If an application does not require more than 15 FPS, TPU and VPU are viable options in terms of latency, and outperform GPU execution by their higher energy efficiency. For Yolov4, Yolov5s, and Yolov5l model, GPU acceleration on more powerful platforms is required to reach higher rates than 5 FPS. It is worth noting that on the NVIDIA Jetson Nano, the VPU acceleration operates at approximately same latency, but requires less energy per inference in comparison to the GPU deployments.

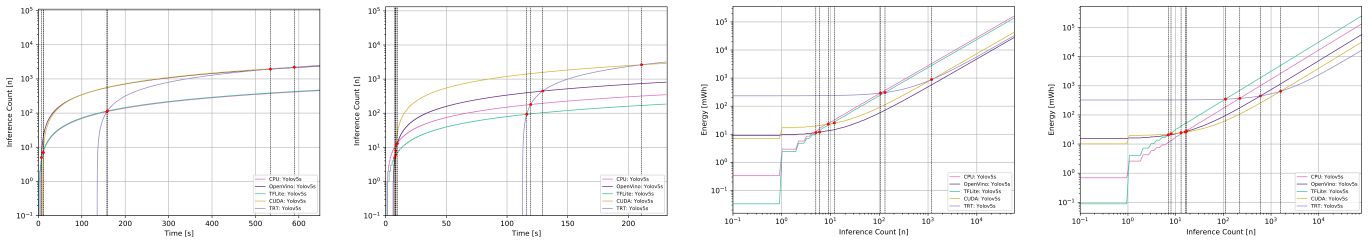
Contemplating TRT and CUDA measurements, they show that inference time is reduced by 18 % to 51 % for TRT, depending on the model and platform. Thereby, higher speedups are obtainable on the more powerful GPU architectures on the NVIDIA Jetson TX2 and AGX. As anticipated, this improvement in terms of latency comes with the drawback of a higher power consumption as illustrated in Figure 5d.

C. Evaluation

For evaluating model deployments, the obtained benchmark measures are harnessed to calculate performance profiles. These profiles reflect the behavior of the different phases and allow a quantitative comparison.

$$N(t) = \begin{cases} 0 & t < \tau_l + \tau_w \\ 1 + \left\lfloor \dfrac{t - (\tau_l + \tau_w)}{\tau_i} \right\rfloor & t \geq \tau_l + \tau_w \end{cases}, \text{ s.t. } t \in \mathbb{R}_{\geq 0} \quad (5)$$

To characterize short- and long-term throughput, Equation (5) allows to define functions to calculate the maximal amount of inferences that are possible for a specific model assuming that the model has not yet deployed to its platform. Since each deployment configuration defines their own functions, they can be easily compared. Their intersections define clear break-points that allow a ranking between scenarios as exemplarily illustrated in Figure 7a and 7b.



(a) $N_{Nano}(t)$ (b) $N_{AGX}(t)$ (c) $E_{Nano}(n)$ (d) $E_{AGX}(n)$

Fig. 7: Yolov5s model throughput and energy consumption profiles for different NVIDIA platforms.

$$E(n) = \begin{cases} E_l & n = 0 \\ E_l + E_w + (n-1)\, E_i & n \geq 1 \end{cases}, \text{ s.t. } n \in \mathbb{N}_0 \quad (6)$$

Complementary, Equation (6) allows to estimate the overall energy spent given a number of executed inferences, enabling to select an appropriate platform by comparing Figure 7c and 7d. In case both platforms are already online and available, e.g. as part of a temporary ad-hoc edge network, Equation (6) can be adapted to utilize the respective ΔE to calculate the criteria when task offloading between platforms becomes beneficial.

VI. Conclusion

Edge device and accelerators enable a novel direction to embed ML-driven data processing close to the application where a reliable connection to cloud-based AI is not viable. However, selecting the right model and platform to fulfill functional and non-functional requirements remains a challenging task for ML engineers. In this paper, we presented *ECBA-MLI*, a scalable benchmark architecture to approach this undertaking. An unified container-virtualized runtime setup ensures reproducible benchmarks for latency and power consumption during different phases of the model execution. We implemented the architecture for a variety of state-of-the-art edge devices and discussed significant observations for conventional object detection models. Based on these measurements, profiles were introduced, allowing to evaluate and compare optimal operating points for different deployment scenarios in quantified manner. For future work, we plan to integrate model accuracy measures as part of our benchmark architecture. This is a critical complement, since model preparation and optimization, such as quantization, impact the output of the model. In addition, task placement algorithm, harnessing the deployment profiles, will be developed to optimize workloads of entire data pipelines in a collaborative edge environment.

VII. Acknowledgment

This work has been financially supported by the AI4DI project. AI4DI receives funding within the Electronic Components and Systems For European Leadership Joint Undertaking (ESCEL JU) in collaboration with the European Union's Horizon 2020 Framework Programme and National Authorities, under grant agreement n° 826060.

References

[1] Rubén Casado and Muhammad Younas. "Emerging trends and technologies in big data processing". In: *Concurrency and Computation: Practice and Experience* 27.8 (2015), pp. 2078–2091.

[2] M. G. Sarwar Murshed et al. "Machine Learning at the Network Edge: A Survey". In: *ACM Comput. Surv.* 54.8 (2021).

[3] Xiaofei Wang et al. "Convergence of Edge Computing and Deep Learning: A Comprehensive Survey". In: *IEEE Communications Surveys Tutorials* 22.2 (2020), pp. 869–904.

[4] Mathias Schneider et al. "Open Traffic Data for Mobility-as-a-Service Applications - Architecture and Challenges". In: Sept. 2021, pp. 375–385.

[5] Sandeep Gupta. "Non-functional requirements elicitation for edge computing". In: *Internet of Things* 18 (2022), p. 100503.

[6] Albert Reuther et al. "Survey of Machine Learning Accelerators". In: *2020 IEEE High Performance Extreme Computing Conference (HPEC)* (Sept. 22, 2020), pp. 1–12. arXiv: 2009.00993. URL: http://arxiv.org/abs/2009.00993 (visited on 02/10/2021).

[7] *Harness AI at the Edge with the Jetson TX2 Developer Kit*. NVIDIA Developer. Mar. 7, 2017. URL: https://developer.nvidia.com/embedded/jetson-tx2-developer-kit (visited on 03/27/2022).

[8] Vijay Janapa Reddi et al. *MLPerf Inference Benchmark*. 2020. arXiv: 1911.02549.

[9] Peter Mattson et al. "MLPerf Training Benchmark". In: *Proceedings of Machine Learning and Systems*. Vol. 2. 2020, pp. 336–349.

[10] Colby Banbury et al. *MLPerf Tiny Benchmark*. 2021. arXiv: 2106.07597.

[11] Mattia Antonini et al. "Resource Characterisation of Personal-Scale Sensing Models on Edge Accelerators". In: *Proceedings of the First International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things - AIChallengeIoT'19*. the First International Workshop. 2019, pp. 49–55.

[12] Blesson Varghese et al. "A Survey on Edge Performance Benchmarking". In: *ACM Comput. Surv.* 54.3 (Apr. 2021). URL: https://doi.org/10.1145/3444692.

[13] *Google Announces New Coral products for 2020*. Electronics-Lab. Jan. 3, 2020. URL: https://www.electronics-lab.com/google-announces-new-coral-products-2020/ (visited on 09/21/2020).

[14] Mark Sandler et al. "MobileNetV2: Inverted Residuals and Linear Bottlenecks". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition. June 2018, pp. 4510–4520.

[15] *Tensorflow 1 Model Zoo*. GitHub. URL: https://github.com/tensorflow/models (visited on 04/27/2021).

[16] *Coral Edge TPU Model Zoo*. Coral. URL: https://coral.ai/models/ (visited on 12/02/2020).

[17] Joseph Redmon and Ali Farhadi. *YOLOv3: An Incremental Improvement*. 2018. arXiv: 1804.02767.

[18] *keras-yolo3*. https://github.com/qqwweee/keras-yolo3. 2018.

[19] *tf2onnx - Convert TensorFlow, Keras, Tensorflow.js and Tflite models to ONNX.* URL: https://github.com/onnx/tensorflow-onnx (visited on 03/30/2022).

[20] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. 2020. arXiv: 2004.10934.

[21] Katsuya Hyodo. *PINTO Model Zoo*. https://github.com/PINTO0309/PINTO_model_zoo. 2020.

[22] Hyeonki Hong. *tensorflow-yolov4*. https://github.com/hhk7734/tensorflow-yolov4. 2018.

[23] Glenn Jocher et al. *ultralytics/yolov5: v4.0 - nn.SiLU() activations, Weights &amp; Biases logging, PyTorch Hub integration*. Version v4.0. Jan. 5, 2021. URL: https://zenodo.org/record/4418161 (visited on 02/24/2021).

[24] Bai et al. *ONNX: Open Neural Network Exchange*. Oct. 6, 2020. URL: https://github.com/onnx/onnx (visited on 03/27/2022).

[25] *microsoft/onnxruntime*. Oct. 9, 2020. URL: https://github.com/microsoft/onnxruntime (visited on 10/09/2020).

Appendix

Device	Accelerator	τ_l [s]						$\overline{E}_l$ [mWh]						$\Delta\overline{E}_l$ [mWh]					
		SSDMobileNetv2	TinyYolov3	TinyYolov4	Yolov4	Yolov5s	Yolov5l	SSDMobileNetv2	TinyYolov3	TinyYolov4	Yolov4	Yolov5s	Yolov5l	SSDMobileNetv2	TinyYolov3	TinyYolov4	Yolov4	Yolov5s	Yolov5l
	CPU	24.188	0.162	0.135	1.217	0.275	1.014	42.782	0.276	0.229	2.201	0.477	1.829	8.020	0.042	0.034	0.452	0.082	0.372
Raspberry Pi 4B	VPU	24.138	3.061	0.270	15.607	5.805	43.907	39.886	5.546	0.438	29.532	10.908	85.816	5.197	1.148	0.050	7.101	2.566	22.715
	TPU	0.017	0.021	0.015	0.020	0.029	0.101	0.027	0.033	0.024	0.031	0.045	0.165	0.002	0.003	0.002	0.003	0.004	0.019
	CPU	38.253	0.124	0.131	1.986	0.427	1.516	51.270	0.162	0.171	3.174	0.570	2.216	6.861	0.018	0.019	0.868	0.074	0.457
Coral Dev Board	VPU	38.477	3.635	0.365	-	-	-	51.921	5.442	0.488	-	-	-	7.252	1.221	0.063	-	-	-
	TPU	0.014	0.023	0.032	0.028	0.034	0.110	0.018	0.029	0.041	0.034	0.044	0.147	0.002	0.002	0.004	0.002	0.004	0.019
	CPU	26.050	0.092	0.093	0.830	0.243	0.693	37.458	0.113	0.113	1.261	0.335	1.042	11.536	0.021	0.021	0.435	0.093	0.352
	VPU	26.128	2.775	0.171	14.460	5.585	48.735	37.792	4.383	0.232	24.499	9.137	82.768	11.792	1.622	0.062	10.109	3.580	34.271
NVIDIA Jetson Nano	TPU	0.016	0.017	0.035	0.023	0.028	0.088	0.018	0.022	0.041	0.027	0.033	0.105	0.003	0.005	0.006	0.004	0.005	0.017
	GPU-CUDA	-	4.191	4.145	5.672	4.426	5.325	-	6.702	6.632	9.022	7.114	8.566	-	2.531	2.508	3.378	2.710	3.267
	GPU-TRT	-	73.378	3.689	-	130.595	170.551	-	125.671	5.903	-	224.412	300.736	-	52.652	2.232	-	94.456	131.019
	CPU	33.657	0.117	0.109	0.911	0.279	0.820	56.714	0.193	0.180	1.615	0.468	1.438	6.094	0.018	0.015	0.244	0.048	0.205
	VPU	34.030	5.107	0.223	17.913	9.045	69.156	57.247	9.327	0.378	33.855	16.771	132.188	6.066	1.646	0.042	6.914	3.166	28.178
NVIDIA Jetson TX2	TPU	0.015	0.021	0.015	0.023	0.030	0.097	0.023	0.032	0.023	0.036	0.047	0.157	0.001	0.001	0.001	0.001	0.001	0.010
	GPU-CUDA	-	4.459	4.426	5.540	4.734	5.424	-	7.664	7.605	9.612	8.116	9.390	-	0.959	0.947	1.281	0.996	1.233
	GPU-TRT	-	69.315	3.030	184.374	99.531	118.742	-	145.498	5.237	399.115	208.653	258.901	-	41.249	0.680	121.819	58.960	80.314
	CPU	23.881	0.123	0.159	0.845	0.283	0.778	54.221	0.299	0.372	2.140	0.695	1.953	0.0*	0.013	0.001	0.165	0.034	0.134
	VPU	24.405	2.954	0.338	13.617	5.826	51.665	55.645	7.764	0.800	35.359	15.387	131.348	0.0*	0.860	0.009	3.527	1.768	10.580
NVIDIA Jetson Xavier AGX	TPU	0.011	0.022	0.014	0.025	0.036	0.129	0.028	0.053	0.035	0.062	0.089	0.327	0.002	0.003	0.002	0.004	0.005	0.025
	GPU-CUDA	-	3.926	5.354	4.885	4.109	4.736	-	9.755	15.488	12.082	10.175	11.681	-	0.579	2.974	0.663	0.569	0.611
	GPU-TRT	-	70.314	2.884	197.971	112.748	142.560	-	196.579	7.175	580.444	326.195	437.071	-	32.217	0.434	117.680	62.641	103.831

TABLE IV: Model load phase: average latency and delta energy consumption (*negative result is replaced by 0.0).

Device	Accelerator	τ_w [s]						$\overline{E}_w$ [mWh]						$\Delta\overline{E}_w$ [mWh]					
		SSDMobileNetv2	TinyYolov3	TinyYolov4	Yolov4	Yolov5s	Yolov5l	SSDMobileNetv2	TinyYolov3	TinyYolov4	Yolov4	Yolov5s	Yolov5l	SSDMobileNetv2	TinyYolov3	TinyYolov4	Yolov4	Yolov5s	Yolov5l
	CPU	0.429	0.420	0.521	5.062	1.543	8.460	0.827	0.823	1.018	10.625	3.150	17.900	0.211	0.219	0.269	3.349	0.933	5.742
Raspberry Pi 4B	VPU	0.429	0.069	2.691	0.520	0.313	0.951	0.777	0.125	4.900	1.048	0.617	1.977	0.160	0.026	1.032	0.301	0.168	0.610
	TPU	0.043	0.396	0.061	6.435	1.585	4.797	0.067	0.698	0.096	11.244	2.905	8.904	0.005	0.129	0.008	1.997	0.628	2.010
	CPU	0.733	0.683	0.840	8.389	2.358	13.824	1.059	1.043	1.271	13.437	3.711	22.238	0.208	0.249	0.296	3.698	0.973	6.189
Coral Dev Board	VPU	0.779	0.080	3.241	-	0.357	-	1.117	0.115	4.799	-	0.552	-	0.213	0.022	1.036	-	0.138	-
	TPU	0.037	0.522	0.061	12.111	2.207	6.304	0.046	0.706	0.077	17.578	3.040	8.708	0.004	0.099	0.006	3.518	0.478	1.389
	CPU	0.428	0.415	0.518	5.139	1.426	8.079	0.700	0.702	0.874	9.732	2.618	15.696	0.274	0.289	0.358	4.618	1.198	7.657
	VPU	0.424	0.064	2.561	0.509	0.280	0.944	0.704	0.095	4.002	0.840	0.457	1.636	0.282	0.031	1.453	0.334	0.178	0.696
NVIDIA Jetson Nano	TPU	0.039	0.400	0.059	6.582	1.450	4.194	0.044	0.609	0.061	10.713	2.385	7.220	0.004	0.211	0.003	4.163	0.943	3.046
	GPU-CUDA	-	2.675	3.137	8.238	5.204	11.185	-	4.451	5.432	17.768	9.986	26.167	-	1.789	2.310	9.570	4.808	15.037
	GPU-TRT	-	0.134	60.990	-	0.360	1.038	-	0.178	113.229	-	0.593	2.444	-	0.044	52.537	-	0.235	1.411
	CPU	0.402	0.333	0.419	3.839	1.192	6.399	0.757	0.626	0.793	7.855	2.397	13.124	0.152	0.125	0.163	2.080	0.605	3.501
	VPU	0.403	0.068	4.684	0.517	0.299	0.953	0.767	0.120	8.527	1.017	0.578	1.955	0.161	0.018	1.483	0.240	0.129	0.521
NVIDIA Jetson TX2	TPU	0.040	0.433	0.060	7.176	1.538	4.401	0.062	0.735	0.092	12.402	2.707	7.856	0.001	0.083	0.002	1.609	0.394	1.237
	GPU-CUDA	-	2.498	2.655	4.485	3.362	5.381	-	4.509	4.961	11.185	7.216	15.040	-	0.751	0.969	4.440	2.159	6.947
	GPU-TRT	-	0.028	47.275	0.204	0.089	0.300	-	0.051	105.544	0.619	0.195	1.008	-	0.008	34.443	0.311	0.062	0.556
	CPU	0.268	0.177	0.280	1.824	0.695	3.241	0.711	0.454	0.710	5.038	1.901	8.814	0.084	0.040	0.057	0.774	0.277	1.237
	VPU	0.268	0.071	3.021	0.516	0.297	0.951	0.702	0.184	7.579	1.442	0.824	2.700	0.076	0.018	0.517	0.235	0.129	0.478
NVIDIA Jetson Xavier AGX	TPU	0.037	0.376	0.074	11.505	1.541	4.366	0.093	0.973	0.183	28.082	4.020	11.212	0.005	0.095	0.011	1.189	0.418	1.005
	GPU-CUDA	-	2.759	3.953	4.198	3.527	4.831	-	7.075	11.814	12.861	9.629	15.891	-	0.626	2.573	3.049	1.385	4.597
	GPU-TRT	-	0.014	50.498	0.116	0.052	0.173	-	0.039	155.799	0.437	0.149	0.767	-	0.007	37.757	0.166	0.028	0.362

TABLE V: Model warmup phase: average latency and delta energy consumption.

Device	Accelerator	τ_i [s]						$\overline{E}_i$ [mWh]						$\Delta\overline{E}_i$ [mWh]					
		SSDMobileNetv2	TinyYolov3	TinyYolov4	Yolov4	Yolov5s	Yolov5l	SSDMobileNetv2	TinyYolov3	TinyYolov4	Yolov4	Yolov5s	Yolov5l	SSDMobileNetv2	TinyYolov3	TinyYolov4	Yolov4	Yolov5s	Yolov5l
	CPU	0.399	0.395	0.500	4.997	1.482	9.039	0.797	0.804	0.986	10.198	3.010	18.128	0.224	0.235	0.268	3.016	0.880	5.137
Raspberry Pi 4B	VPU	0.399	0.066	0.084	0.504	0.288	0.941	0.804	0.135	0.169	1.071	0.607	2.045	0.230	0.040	0.049	0.348	0.194	0.693
	TPU	0.018	0.373	0.034	6.340	1.593	4.869	0.033	0.699	0.064	11.283	3.000	9.187	0.006	0.163	0.015	2.172	0.711	2.189
	CPU	0.696	0.661	0.811	8.400	2.300	-	1.065	1.063	1.263	13.666	3.874	-	0.257	0.295	0.322	3.915	1.204	-
Coral Dev Board	VPU	0.695	0.072	0.092	-	-	-	1.077	0.120	0.149	-	-	-	0.271	0.036	0.042	-	-	-
	TPU	0.014	0.510	0.039	12.064	2.214	6.306	0.021	0.724	0.055	17.737	3.083	8.845	0.004	0.132	0.010	3.732	0.512	1.524
	CPU	0.398	0.402	0.500	5.156	1.423	8.228	0.753	0.777	0.935	9.857	2.783	16.268	0.357	0.377	0.438	4.727	1.367	8.080
	VPU	0.397	0.063	0.080	0.499	0.273	0.934	0.753	0.107	0.133	0.875	0.479	1.693	0.358	0.044	0.054	0.378	0.207	0.764
NVIDIA Jetson Nano	TPU	0.016	0.337	0.030	6.526	1.373	4.139	0.025	0.576	0.051	10.571	2.364	7.172	0.009	0.241	0.022	4.077	0.999	3.054
	GPU-CUDA	-	0.070	0.104	0.679	0.263	1.095	-	0.185	0.257	1.920	0.737	3.118	-	0.116	0.154	1.245	0.475	2.028
	GPU-TRT	-	0.050	0.061	-	0.206	0.896	-	0.138	0.162	-	0.552	2.486	-	0.088	0.102	-	0.347	1.594
	CPU	0.367	0.314	0.396	3.778	1.171	6.367	0.750	0.653	0.810	7.880	2.455	13.289	0.198	0.180	0.215	2.197	0.693	3.713
	VPU	0.370	0.066	0.084	0.505	0.287	0.941	0.762	0.132	0.165	1.050	0.596	2.023	0.206	0.033	0.038	0.290	0.164	0.607
NVIDIA Jetson TX2	TPU	0.016	0.348	0.032	7.189	1.427	4.262	0.030	0.622	0.059	12.579	2.588	7.747	0.005	0.099	0.011	1.766	0.441	1.336
	GPU-CUDA	-	0.029	0.028	0.276	0.128	0.431	-	0.104	0.101	1.092	0.456	1.715	-	0.061	0.058	0.677	0.264	1.067
	GPU-TRT	-	0.021	0.024	0.159	0.079	0.271	-	0.075	0.085	0.655	0.274	1.117	-	0.043	0.049	0.415	0.155	0.710
	CPU	0.226	0.154	0.208	1.763	0.660	3.214	0.563	0.402	0.541	4.570	1.684	8.282	0.035	0.042	0.054	0.449	0.142	0.769
	VPU	0.226	0.065	0.083	0.501	0.277	0.937	0.569	0.166	0.209	1.309	0.726	2.464	0.040	0.014	0.016	0.139	0.079	0.274
NVIDIA Jetson Xavier AGX	TPU	0.016	0.316	0.032	11.223	1.242	3.819	0.040	0.799	0.083	27.556	3.168	9.790	0.003	0.061	0.008	1.322	0.266	0.862
	GPU-CUDA	-	0.012	0.016	0.123	0.077	0.185	-	0.079	0.079	0.853	0.398	1.292	-	0.050	0.042	0.565	0.218	0.861
	GPU-TRT	-	0.008	0.010	0.076	0.037	0.139	-	0.050	0.056	0.486	0.205	0.868	-	0.031	0.033	0.310	0.118	0.543

TABLE VI: Model inference phase: average latency and delta energy consumption.