DeepEdgeBench: Benchmarking Deep Neural Networks on Edge Devices*

*Note: This the preprint version of the accepted paper at IC2E'21

Stephan Patrick Baller*, Anshul Jindal*, Mohak Chadha*, Michael Gerndt*
*Chair of Computer Architecture and Parallel Systems, Technische Universität München, Germany
Garching (near Munich), Germany
Email: {stephan.baller, anshul.jindal, mohak.chadha}@tum.de, gerndt@in.tum.de

***Abstract*—EdgeAI (Edge computing based Artificial Intelligence) has been most actively researched for the last few years to handle variety of massively distributed AI applications to meet up the strict latency requirements. Meanwhile, many companies have released edge devices with smaller form factors (low power consumption and limited resources) like the popular Raspberry Pi and Nvidia's Jetson Nano for acting as compute nodes at the edge computing environments. Although the edge devices are limited in terms of computing power and hardware resources, they are powered by accelerators to enhance their performance behavior. Therefore, it is interesting to see how AI-based Deep Neural Networks perform on such devices with limited resources.**

In this work, we present and compare the performance in terms of inference time and power consumption of the four SoCs: Asus Tinker Edge R, Raspberry Pi 4, Google Coral Dev Board, Nvidia Jetson Nano, and one microcontroller: Arduino Nano 33 BLE, on different deep learning models and frameworks. We also provide a method for measuring power consumption, inference time and accuracy for the devices, which can be easily extended to other devices. Our results showcase that, for Tensorflow based quantized model, the Google Coral Dev Board delivers the best performance, both for inference time and power consumption. For a low fraction of inference computation time, i.e. less than *29.3%* of the time for MobileNetV2, the Jetson Nano performs faster than the other devices.

***Index Terms*—edge computing, deep learning, performance benchmark, edge devices, power consumption, inference time, power prediction**

I. Introduction

Advances in computing devices and high-speed mobile networking provide today's applications to be distributed globally. Such applications are thus deployed on cloud service platforms to benefit from on demand provisioning, unlimited resource pooling, and dynamic scalability. Modern deep learning techniques serve a key component in various real-time applications, like speech recognition [1], recommendation systems [2] and video classification [3]. However, deep learning-based approaches require a large volume of high-quality data to train and are very expensive in terms of computation, memory and power consumption [4]. Moreover, existing cloud computing is unable to manage these massively distributed applications and analyze their data due to: i) challenges posed on the network capacity when tasks are deployed to the cloud [5]; ii) many applications, for example, autonomous driving [6], have strict latency requirements that the cloud would have difficulty meeting since it may be far away from the users [7].

The concept of Edge Computing has been recently proposed to complement cloud computing to resolve these problems by performing certain tasks at the edge of the network [8]. The idea is to distribute parts of processing and communication to the "edge" of the network, i.e closer to the location where it is needed. As a result, the server needs less computing resources, the network is less strained and latencies is decreased.

Edge Devices can come in a variety of forms ranging from large servers to low-powered System on a chip (SoC) devices like the popular Raspberry Pi or any other ARM based devices. Deep Neural Networks (DNNs) may occupy big amounts of storage and computing resources [9], [10]. Although the edge devices are limited in terms of computing power and hardware resources, they are powered by accelerators to enhance their performance at the edge computing environments.

In the context of Edge Computing, it is rather interesting to see how devices with smaller form factors (low power consumption and limited resources) can handle DNN evaluation. There are a considerable number of articles on the benchmark of edge devices [11]–[15]. However, some of the articles are already outdated and others miss benchmarks on the major latest edge devices, thus lacking a thorough comparison between the edge devices concerning the DNN applications. Unlike other works, this article focuses on evaluating the performance of recent edge device for DNN models inferencing. The key contributions of this work are as follows:

- We present a comparative analysis on recent accelerator-based edge devices specialized for the DNN application domain. We choose following devices as our target edge devices to assess their performance behavior and capabilities for DNN applications:
  - ASUS Tinker Edge R [16]
  - Raspberry Pi 4 [17]
  - Google Coral Dev Board [18]
  - NVIDIA Jetson Nano [19]
  - Arduino Nano 33 BLE [20]
- Our evaluation perspectives include inference speed for a fixed number of images, power consumption during (accelerated) DNN inference, and models accuracies with

and without optimization for the respective device.
- We used four different deep learning inference frameworks for the evaluation: Tensorflow, TensorRT, Tensorflow Lite and RKNN-Toolkit.
- We also provide a method for measuring power consumption, inference time and accuracy for the devices, which can be easily extended to other devices. We open source the collected data and the developed method for further research[1].

The rest of this paper is organized as follows. §II gives an overview of the target edge devices, model frameworks and formats used in this work for the evaluation. In §III, the overall evaluation setup and methodology are described. §IV provides the experimental configurations details. In §V, our performance evaluation results are presented along with the discussion of those results. Finally, §VI concludes the paper and presents a future outlook.

II. Target Edge Devices, Frameworks and Model Formats

In this section, we compare different edge device architectures and present their hardware overview, and also briefly describe the characteristics of different DNN models and frameworks used in this work for benchmarking the edge devices.

A. Target Edge Devices

We have considered the following five edge devices summarized in Table I.

1) Nvidia Jetson Nano: The Jetson Nano is one of the offerings from Nvidia for edge computing [19]. With the Jetson product line, Nvidia deploys their Graphics Processing Unit (GPU) modules to the Edge with accelerated AI performance. The Jetson Nano's GPU is based on the Maxwell microarchitecture (GM20B) and comes with one streaming multiprocessor (SM) with 128 CUDA cores, which allows to run multiple neural networks in parallel [19]. The Jetson Nano is the lowest-end version of the product line with a peak performance of 472 Giga Floating Operations Per Second (GFLOPS) [21]. It comes with either 2GB or 4GB Random Access Memory (RAM) version wherefore we're testing the 4GB version. The Nano can be operated in two power modes, 5W and 10W. We used in our experiment an Nvidia Jetson Nano with 4 GB RAM and 10 W to maximize performance. Compared to the other target edge devices in this work, the Jetson Nano stands out with a fully utilizable GPU.

2) Google Coral Dev Board: Coral is a platform by Google for building AI applications on edge devices [22]. The Google Coral Dev Board is one of the offerings which features the "edge" version of the TPU (tensor processing unit) [23], an application specific integrated circuit (ASIC) designed for accelerating neural network machine learning, particularly using the TensorFlow framework [24]. The Edge TPU operates as a co-processor in an edge device and allows for an efficient aggregation of tens of thousands of ALUs (arithmetic logic units) and faster data transfer rate between the TPU and the memory. However, the Edge TPU is fine-tuned for matrix operations which are very frequent in neural network machine learning.

The Google Coral Dev Board comes with either 1GB or 4GB of RAM. The SoC integrates the Google Edge TPU with the performance of 4 trillion operations (tera-operations) per second (TOPS) or 2 Tera Operations Per Second (TOPS) per watt [25]. To make use of the dedicated unit, models in a supported format can be converted to work with the PyCoral and Tensorflow Lite framework [26].

3) Asus Tinker Edge R: Currently, ASUS offers six devices under the Tinker Board brand [27]. Two of them, the Tinker Edge R [16] and Tinker Edge T [28] are specifically made for AI applications. While the Tinker Edge T is supported by a Google Edge TPU, the Tinker Edge R uses Rockchip Neural Processing Unit (NPU) (RK3399Pro), a Machine Learning (ML) accelerator that speeds up processing efficiency, and lowers power demands. With this integrated Machine Learning (ML) accelerator, the Tinker Edge R is capable of performing 3 tera-operations per second (TOPS), using low power consumption. And it's optimized for Neural Network (NN) architecture, which means Tinker Edge R can support multiple Machine Learning (ML) frameworks and common Machine Learning (ML) models can be easily compiled and run on the Tinker Edge R. Rockchip, the manufacturer of the chip, provides the RKNN-Toolkit as a library to convert and run models from different frameworks. As operating system, Asus offers a Debian Stretch and Android 9 variant of its so called Tinker Operating System (OS) to power the board. Compared to the other devices, the board offers additional interfaces for expanding connectivity (mPCIe, M.2, serial camera, serial display), especially interesting for IoT applications.

4) Raspberry Pi 4: The devices from the Raspberry Pi series are among the most popular SoCs and represent go-to products for people who are interested in IoT [29]. In June 2019, the Raspberry Pi 4 was released, featuring a variety of RAM options and a better processor. The Raspberry Pi 4 comes with Gigabit Ethernet, along with onboard wireless networking and Bluetooth. The Rasbperry Pi is commonly used with Raspbian, a 32 Bit OS based on Debian. As the Raspberry Pi 4 is capable of running a 64 Bit OS. We're instead going to use the available 64 Bit Raspbian version (aarch64 architecture) to provide the same testing conditions as the other devices. This actually makes a difference in performance, as a blog post [30] on benchmarking the Raspberry Pi from 2020 suggests. In contrary to it's predecessors, the 4th generation is available in multiple variants with different RAM sizes (2, 4 and 8GB). For this work we're testing the 4GB variant.

While all other devices in this work (excluding the Arduino) include GPUs or co-processors for neural computation, the Raspberry Pi lacks such a dedicated unit, but stands out with its low price and availability compared to the other products.

[1]https://github.com/stephanballer/deepedgebench

TABLE I: Overview on different target edge devices specifications

	Tinker Edge R (2019) [16]	**Raspberry Pi 4 (2019)** [17]	**Google Coral Dev Board (2020)** [18]	**NVIDIA Jetson Nano (2019)** [19]	**Arduino Nano 33 BLE (2019)** [20]
CPU	RK3399Pro, Dual-core Cortex A72 @1.8GHz, Quad-core Cortex A53 @1.4GHz (ARMv8A)	BCM2711, Quad-core Cortex A72 @1.5GHz (ARMv8A)	NXP i.MX 8M, Quad-core Cortex A53 @1.5GHz (ARMv8A)	Cortex A57 @1.43GHz (ARMv8A)	nRF52840, Cortex M4 @64MHz (ARMv7E-M)
Artificial Intelligence (AI) Unit	Rockchip NPU	-	Google Edge TPU	128-core Maxwell GPU	-
Memory	LPDDR4 4GB, LPDDR3 2GB (NPU)	LPDDR4 4GB @3200MHz	LPDDR4 1GB @1600MHz	LPDDR4 4GB @1600MHz	256KB SRAM (nRF52840)
Storage	16GB eMMC, Micro SD	Micro SD	8GB eMMC, Micro SD	Micro SD	1MB Flash Memory
First Party OSs	TinkerOS, Android	Raspbian	Mendel Linux	Ubuntu	MBed OS
Optimized Frameworks	TensorFlow 1, Tensorflow Lite, Caffe, Kaldi, MXNet, ONNX	-	TensorFlow Lite	TensorFlow, TensorRT, NVCaffe, Kaldi, MXNet, DIGITS, PyTorch	TensorFlow Lite Micro

5) Arduino Nano 33 BLE: Besides the main A-Profile architecture family from Arm Ltd. company used in SoCs, the M-Profile for microcontrollers is meant to provide "low-latency, highly deterministic operation for deeply embedded systems" [31]. In their latest revision Armv8.1-M, a new M-Profile Vector Extension (MVE) is included to accelerate Machine Learning (ML) algorithms. The Cortex Armv7-M is the latest architecture used in currently available microcontrollers like Arduinos. Additionally, *TinyML* is a foundation aiming to bring ML inference to ultra-low-power devices, i.e. microcontrollers [32]. Inference frameworks like Tensorflow Lite already implement this approach by enabling Deep Learning (DL) models to be inferenced on products like the Arduino Nano 33 BLE or ESP-32 [33].

Therefore, the Arduino Nano 33 BLE is also considered as target edge device to represent this category of edge devices. It features the nRF52840, a 32-bit ARM Cortex™-M4 CPU running at 64 MHz from Nordic Semiconductors [20]. It is officially supported by the Tensorflow Lite Micro DL framework and counts to the "larger" microcontroller systems [33]. With a Flash Memory size of 1MB, it's therefore capable of running networks with up to 500KB in size [20].

B. Inference frameworks and conversion between them

All devices share Tensorflow (TF) as common framework to provide optimized model inference, though only being compatible with certain models and Tensorflow variants. The Tensor Processing Unit (TPU) of the Google Coral Dev Board is only available for use with optimized Tensorflow Lite models, RKNN-Toolkit for Tinker Edge R only supports conversion of models in Frozen Graph format and Tensorflow Lite models and TensorRT for the Nvidia Jetson Nano doesn't support Tensorflow Lite. Besides, there are further restrictions for converting models from Tensorflow to be optimized for the respective device.

To make use of the devices' NPUs, TPUs and GPUs, dedicated Python Application Programming Interfaces (APIs)s are provided to convert and inference models from one or more frameworks. In this subsection we describe four such methods.

1) Tensorflow: TensorFlow [24] is an open-source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) that flow between them. This flexible architecture deploys computation to one or more CPUs or GPUs without rewriting code. Tensorflow is the main framework we used for exporting pretrained models for inferencing on the edge devices. As a framework, it is already optimized for systems with Nvidia graphics cards in combination with their neural network library CUDNN. Nvidia's Jetson Nano is therefore capable of running networks on its integrated GPU, giving it an advantage over the other devices that perform inference solely on the Central Processing Unit (CPU).

2) TensorRT: NVIDIA TensorRT [36] is a C++ library that facilitates high performance inferencing on NVIDIA graphics processing units (GPUs). TensorRT takes a trained network, which consists of a network definition and a set of trained parameters, and produces a highly optimized runtime engine which performs inference for that network. TensorRT applies graph optimizations, layer fusion, among other optimizations, while also finding the fastest implementation of that model leveraging a diverse collection of highly optimized kernels. TensorRT supports parsing models in ONNX, Caffe and UFF format. To convert a model from Tensorflow or other frameworks, external tools like the *tf2onnx* command line tool or *uff* library have to be used [37].

Additionally, models can be optimized with TensorFlow with TensorRT (TF-TRT) optimization, which is included in the Tensorflow Python API.

3) Tensorflow Lite: TensorFlow Lite was developed to run machine learning models on microcontrollers and other Internet of Things (IoT) devices with only few kilobytes of memory [33]. Hence, runtime environments are offered for various platforms covering iOS, Android, embedded Linux and microcontrollers with API support for Java, Swift, Objective-

TABLE II: Optimization compatibility for pretrained Tensorflow models on various edge devices [34], [35]

	Asus Tinker Edge R	**Google Coral Dev Board**	**Nvidia Jetson Nano**		**Arduino Nano 33**
API	RKNN Toolkit	PyCoral/ TF Lite	Tensor-RT	TF-TRT	TF Lite micro
TF 1 (frozen graph)	👍	👎	👍	👍	-
TF 1 (saved model)	👎	👎	👍	👍	-
TF 2 (saved model)	👎	👎	👍	👍	-
TF Lite	👍	👎	👍	👎	-
TF Lite quant. (8-bit quantized model)	👍	👍	👎	👎	👍

C, C++, and Python [38]. TensorFlow Lite uses FlatBuffers as the data serialization format for network models, eschewing the Protocol Buffers format used by standard TensorFlow models. After building a model in the main Tensorflow framework, it can be converted to TFLite format with the inbuilt converter, supposing all operations used by the model are supported. Unsupported operations can be manually implemented [39].

During the conversion process, it is also possible to quantize the model, which is needed for deploying to micro-controllers or devices utilizing the Google Edge TPU. To do that a Tensorflow Lite model must be "compiled" to be compatible with the Google Edge TPU, e.g. with the Edge TPU Compiler command line tool by Coral [40]. In addition to being fully 8-bit quantized, only supported operations must be included in the model and the Tensor sizes must be constant at compile-time while only being 1, 2 or 3-dimensional (further dimensions are allowed as long as only the three innermost dimensions have a size greater than one) [26].
For the model to be used in the C++ API, which is required by the Arudino Nano 33 BLE, the TFLite model must be converted to a C array. This is done by using a hexdump tool, e.g. using the "xxd -i" Unix command.

4) RKNN-Toolkit: Rockchip offers its RKNN-Toolkit framework for running optimized inference on their AI accelerated processors [34]. It allows conversion from various frameworks (as seen in Table I) to its own ".rknn" format. The resulting models can then be run using the same framework.

C. Model formats

Pretrained Tensorflow models can come in a variety of different formats [41]. The main ones used for deploying are *SavedModel*, *TF1 Hub* format, *Frozen Graph* and *TFLite* format [41]. Additionally, TF-TRT and RKNN-Toolkit use other formats to run an accelerated inference on the dedicated unit.

1) SavedModel Format: It is the recommended format for exporting models [42]. It contains information on the complete Tensorflow program, including trained parameter values and operations. Loading and saving a model can be done via one command without further knowledge about its internal structure. By default, a saved model cannot be trained any further after saving it.

2) TF1 Hub Format: The TF1 Hub format is a custom serialization format used in by TF Hub library [42]. The TF1 Hub format is similar to the *SavedModel* format of TensorFlow 1 on a syntactic level (same file names and protocol messages) but semantically different to allow for module reuse, composition and re-training (e.g., different storage of resource initializers, different tagging conventions for metagraphs). To support loading a model in this format in the TF 2 Python API, an additional module has to be imported [42], [43].

3) Frozen Graph Format: Frozen Graph format is the deprecated format. In contrary to the *TF1 Hub* format, it saves the parameters as constants, bound to their respective operations. A *Frozen Graph* can't be trained after it's loaded, though converting it to *TF1 Hub* format is possible (yet still not trainable) [44].

4) TFLite Format: TFLite (".tflite") is used to deploy models and make them available on devices with limited resources [38]. A model must hence be converted from a TF model with one of the previous formats. Before converting, it must be considered that the TFLite runtime platform for performing the inference has only limited support for Tensorflow operations. Unsupported operations can therefore be explicitly implemented as custom operations in the runtime framework. During conversion, the model can be quantized to reduce model size and inference speed.

Table II provides a summary of the optimization compatibility for various pre-trained Tensorflow models on different devices. By looking at it, we can conclude that for a comparison using every device's full potential, a model must be available in Tensorflow Lite (and 8-bit quantized) and one of the other Tensorflow formats. The Raspberry Pi is not included in this overview as it doesn't have a dedicated unit for which a model can be optimized for. As we're focusing on IoT devices, we want to pick models that are commonly used in this category i.e. for image classification. The Tensorflow 1 Detection Zoo and the Tensorflow Model Garden therefore provide a variety of image classification models with different sizes and formats [10].

III. Methodology

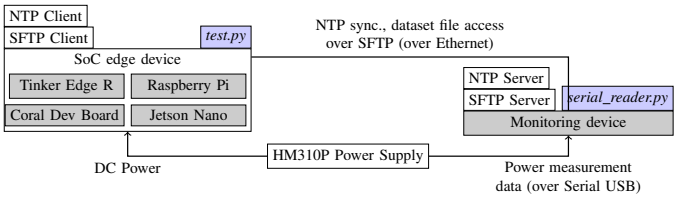


Fig. 1: High-level overview of the testing setup showcasing the workflow between each of its components.

For measuring time, power consumption and accuracy, we developed a python framework containing two main scripts: i) *test.py* for running the models on the devices and getting inference data, and ii) *serial_reader.py* for getting power measurement data to be run on the monitoring device (a system used for collecting monitoring data from all the edge devices) as depicted in Figure 1.

Power measurement and inference are performed on separate devices to avoid consumption of computing resources on the test device, therefore the clocks are coordinated. To accomplish this, we use the systemd [45] implementation of the Network Time Protocol, which is able to achieve better than 1ms deviation in local networks [46]. The monitoring device therefore runs a Network Time Protocol (NTP) server, for which the edge device is configured to request the current time from. For higher accuracy, the devices are directly connected via Ethernet as shown in Figure 1.

In the following subsections we describe the three important tasks conducted by the two developed python modules in more details.

A. Inference time calculation using test.py

The *test.py* script runs on the edge device under test, evaluates a model in a chosen framework and generates timestamps during the process. The *test.py* script allows inference of any platform-compatible model trained on the ImageNet [47] or COCO [48] dataset. On some of the devices, the ImageNet dataset occupies more memory than available. The directory containing the images are therefore saved on the monitoring device and accessed over Secure File Transfer Protocol (SFTP). Inference platforms include Tensorflow, Tensorflow Lite, RKNN-tookit, TensorRT with ONNX so the script can be run on every device up for testing. The resulting output tensors are saved into a list for each image batch. Model accuracy is calculated after the last inference is performed. The program also supports defining the batch size, input datatype, image dimensions and number of inferences that should be performed. Besides measuring the model performance for each device, which is the time a device took for each inference, we also generate inference-timestamps indicating the current workload of the device (i.e. when an inference starts and ends). This is necessary for knowing exactly what process we are monitoring while calculating performance and watts per time later on.

Labeled timestamps (e.g. *inf_start_batch_i* denoting the start time and *inf_end_batch_i* denoting the end time) are generated just before and after a loop in which the inference function is called a given amount of times. Thus, the measured time only includes the inference performed by the framework, excluding image pre-processing and evaluation of the results.

Overall inference time t_{inf}, average inference time $\bar{t}_{\text{inf}}$ and overall time spent on the testing program for N images t_{test} can then be calculated from the generated file as following:

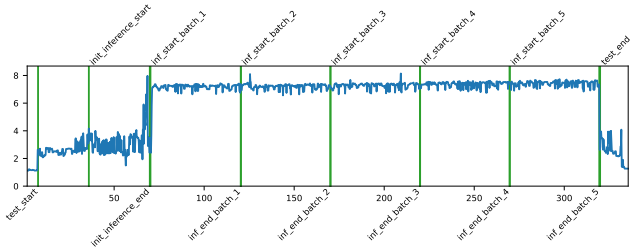


Fig. 2: Power consumption (watts) over time (seconds) and labels during 1000 x 5 images inference on the Nvidia Jetson Nano with MobilenetV2 in Tensorflow

$$
\begin{aligned}
t_{\text{inf}} &= \sum_{i=1}^{N} t_{\text{inf\_end\_batch\_}i} - t_{\text{inf\_start\_batch\_}i} \\
\bar{t}_{\text{inf}} &= \frac{t_{\text{inf}}}{N} \\
t_{\text{test}} &= t_{\text{test\_end}} - t_{\text{test\_start}}
\end{aligned}
\tag{1}
$$

B. Power consumption calculation using serial_reader.py

During the inference, the *serial_reader.py* script runs on a monitoring device to collect power consumption data. To measure power consumption, we calculate how much energy is consumed during the idle state, inference time and during the entire testing program. Knowing the exact value at any point in time during the process is mandatory to get a meaningful result. Therefore measuring with a wattmeter and reading the results by hand won't suffice. Instead, we measure voltage and current via a laboratory power supply with an integrated high precision power meter, the HM310P by Hammatek. It provides a serial Modbus USB interface over which the registers containing configuration and status of the power supply can be read and set. While no official software and documentation are available until now, a user-contributed documentation [49] shows an overview of the modbus accessible registers in the HM310P. This made it possible to implement a python library *hm310p.py* in the testing program, providing functions to control the device.

In a given interval, the *serial_reader.py* monitors the power consumption by requesting power, voltage and current from a given path to the HM310P device. Optionally, the script creates a live plot of the incoming data and/or saves the data with respective timestamps to a given path. Afterwards, the generated inference-timestamps by the *test.py* can be summarized and plotted with the same script as shown in Figure 2 for Nvidia Jetson Nano when performing inference on 1000 x 5 images with MobilnetV2 in Tensorflow.

Overall energy consumption $wm_{\textbf{inf}}$ (in wattminutes) and average power $\overline{w}_{inf}$ (in watts) during inference are evaluated for an inference-timestamps file and power-timestamps file containing a list of tuples T. Each tuple contains time in seconds, current in amps, voltage in volts and power in watts. Firstly, a list of power data can be extracted for each batch i, for which $W_i = \{w|(t, a, v, w) \in T, t_{\textbf{inf}\_\text{start\_batch\_}i} \leq t \leq$

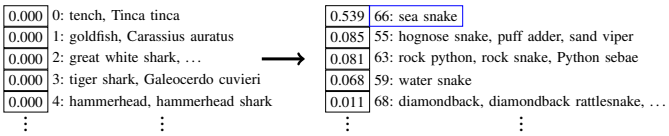


Fig. 3: Interpreting the result of an image detection model

$t_{\text{inf\_end\_batch\_}i}\}$. The respective values can then be calculated with

$$\begin{aligned} \overline{w}_i &= \frac{\sum_{w \in W_i} w}{|W_i|} \\ \overline{w}_{inf} &= \frac{\sum_{i=1}^{N} \overline{w}_i}{N} \\ wm_{\text{inf}} &= \overline{w}_{\text{inf}} \times \frac{t_{\text{inf}}}{60}. \end{aligned} \tag{2}$$

C. Model accuracy calculation using test.py

The accuracy of models can change after being converted and optimized to work on a specific platform. Therefore it is important to include it into the comparison. For common tasks like image classification and object detection, there are metrics to measure how good an algorithm performs for a given dataset.

We use two metrics to indicate the accuracy of a model, Top-1 and Top-5 accuracy. For Top-1 accuracy, only the most probable class is compared with ground truth, whereas for Top-5 accuracy, the correct class just has to be among the top five guesses. We then consider the relation of correctly detected images to all images using the following equation:

$$\text{Top} = \frac{\text{correct results}}{\text{all results}} \tag{3}$$

During evaluation in the *test.py* script, we calculate this value by labeling the resulting tensor values with their indices and then sorting them by weight as shown in Figure 3. If the ground truth value for the image equals the first value or is among the first five values, the amount of *correct results* is incremented respectively for Top-1 and Top-5 accuracy. With a significant amount of samples, we get an approximation for the accuracy of a model. The ImageNet dataset ILSVRC2012 [47] therefore provides 50,000 images for validation with ground-truth data, from which we're going to use 5,000 samples for each run.

The described methodology allows testing any device supporting the respective Python libraries of the *test.py* script. This is not the case for the Arudino Nano 33 BLE, so power consumption and inference time plus accuracy is measured separately. For measuring inference time, a C++ script is provided to be run on the arduino, implementing the TFLite C++ library. The script sets up the MobileNetV1 network and waits for the input image to be received over serial USB, which are then put into the placeholders of the input tensor. Inference time is measured on the arduino and sent back over serial USB afterwards, together with the output tensor after the inference is complete. Transmission and evaluation of the dataset data is handled by the *test.py* script, which is run on the monitoring device this time.

IV. Experimental Configurations

The previously described methodology is used with different configurations (model, edge device, framework, given amount of images from the ImageNet data) combinations for evaluations. For more precise results, one test run with 5,000 images for a decent model accuracy approximation and one with 5 images repetitively inferenced 1,000 times to further counteract inaccuracies due to possible delays when reading power measurement data from the power supply are conducted.

During all measurements, unnecessary processes like graphical user interfaces as well as internal devices like wireless Local Area Network (LAN) modules are disabled to minimize power consumption and processing power.

It may also be possible that an inference, performed solely on the CPU may be more energy-efficient than with using the dedicated unit. We therefore also test the devices using the "native" Tensorflow and Tensorflow Lite framework without optimizations for a dedicated unit.
In the results section, the main focus will be on the following collected data:

- Average time spent on inference of one of 5,000 images
- Total time spent on inference of 5,000 images
- Power during idle state (LAN on and off)
- Average power during inference of 1,000 x 5 images
- Total power consumption during inference of 1,000 x 5 images
- Accuracy for each platform-device combination

A. Models for evaluation

We pick models that can be optimized for as many devices as possible. For an extensive comparison, four model configurations are considered. The most limiting factor for finding a common model is for the network to fit on the Arudino Nano 33 BLE with only 1MB of RAM. The only image classification network from the Tensorflow Model Garden that is within that range is the 8-bit quantized "MobileNet_v1_0.25", available in *TFLite* and *TF1 Hub* format (**MobileNetV1 Quant. Lite**) [50].

Ergo, the quantized TFLite model can also be deployed to the Asus Tinker Edge R, the Google Coral Dev Board and the Raspberry Pi. Though being capable of running on the Arudino Nano 33 BLE, its small size may lead to less accurate results on the other devices. An inference for one image takes less than 2ms on the Google Coral Dev Board. With an offset of up to 1ms between power measurement timestamps and inference timestamps, up to 50% of the collected power consumption data may not be assignable to the current workload and therefore be invalid.

Selecting a bigger size model that takes longer for the evaluation does increases the accuracy of measuring inference time, since the relation of the actual inference to unintentionally included function calls (like time()) increases. Hence, we also use a larger MobileNetV2 model with 12x as many parameters, also available on the Tensorflow Model Garden.

Ascribed to the incompatibility of non-quantized models on the Google Coral Dev Board and quantized models on the Nvidia Jetson Nano, there are two main test runs for the model. The first one includes the "float_v2_1.4_224" Frozen Graph to be evaluated on the Asus Tinker Edge R, the Raspberry Pi and the Nvidia Jetson Nano (**MobileNetV2**). The second, includes the model's corresponding 8-bit quantized version to be run on the Google Coral Dev Board instead of the Nvidia Jetson Nano (**MobileNetV2 Quant. Lite**).

We also consider the non-quantized version to see the impact of quantization and use of TFLite (**MobileNetV2 Lite**).

All previously discussed image classification models are trained on the ImageNet ILSVRC2012 dataset [47].

V. Results

After performing the experiments, we get an insight into the strengths and weaknesses of each device. In the following subsections, we discuss how the devices compare in respect to the evaluation scenarios.

A. Inference performance

Figure 4 shows the time taken for performing inference on various devices when using only CPU, as well as when utilizing the dedicated AI unit across different models.

First, we will take a look at the inference performance without utilization of the dedicated AI units. Although this information may be irrelevant for actual use cases with those devices, it may be useful when comparing other devices utilizing the same CPUs that do not include an AI unit. Additionally, it gives a hint on the performance difference for Tensorflow and Tensorflow Lite framework and by how much the performance increases when using the respective dedicated units.

1) Inference performance without utilization of the dedicated AI units: For *MobileNetV2*, Nvidia Jetson Nano outperforms its competition in terms of inference performance with `685.490` seconds (∼`0.137` seconds per inference) followed by Rasphberry Pi 4 with `1037.7` seconds (∼`0.21` seconds per inference), then Coral Dev board with `1389.4` seconds (∼`0.28` seconds per inference), and lastly Asus Tinker Edge R with `1652.1` seconds (∼`0.33` seconds per inference). It shows that, Nvidia Jetson Nano is almost **2.5x** faster than the Asus Tinker Edge R.

Besides determining the best performing devices, running the same model in Tensorflow and Tensorflow Lite Runtime seemingly makes a significant difference (*MobileNetV2* vs *MobileNetV2 Lite*). From the Figure 4 we can see that, although in case of *MobileNetV2 Lite* Nvidia Jetson Nano performs best with `905.1` seconds (∼`0.181` seconds per inference) followed by Asus Tinker Edge R with `938.6` seconds (∼`0.187` seconds per inference), then Raspberry Pi 4 with `965.1` seconds (∼`0.193` seconds per inference), and lastly Coral Dev board with `1780.6` seconds (∼`0.356` seconds per inference), but when compared with *MobileNetV2* model, The Tinker Edge R (`43%` reduction in time) and Raspberry Pi (`7%` reduction in time) perform better with Tensorflow Lite, while the Coral Dev Board (`32%` increase in time) and Jetson Nano (`28%` increase in time) perform worse.

In case of the quantized TFLite models (*MobileNetV2 Quant. Lite* and *MobileNetV1 Quant. Lite*), the Asus Tinker Edge R has the best performance (`584.9` seconds for *MobileNetV2 Quant. Lite* and `22.2` seconds for *MobileNetV1 Quant. Lite*), although performing the inference for MobileNetV1 on its NPU (`24.0` seconds) is actually slower than on its CPU. The most probable reason for this behavior is the small size of the model, considering that the Tinker Edge R performed significantly better for the other models when utilizing its NPU. It is followed by Rasphberry Pi 4 with `640.0` seconds for *MobileNetV2 Quant. Lite* and `22.2` seconds for *MobileNetV1 Quant. Lite* which is the same as that of Asus Tinker Edge R, then Nvidia Jetson Nano with `657.7` seconds for *MobileNetV2 Quant. Lite* and `23.0` seconds for *MobileNetV1 Quant. Lite* and lastly, Coral Dev board with `1081.3` seconds for *MobileNetV2 Quant. Lite* and `37.6` seconds for *MobileNetV1 Quant. Lite*. One can conclude that Asus Tinker Edge R is almost **1.8x** faster for *MobileNetV2 Quant. Lite* and **1.7x** faster for *MobileNetV1 Quant. Lite* than Coral Dev board.

Additionally, when comparing *MobileNetV2 Lite* with *MobileNetV2 Quant. Lite*, shows that model quantization results in performance increment for all the devices. From the Figure 4 we can see that, for inference, there is a `27.3%` reduction in inference time for Jetson Nano, while `37.6%` for Asus Tinker Edge R, `39.3%` for Coral Dev board, and `33.7%` for Raspberry Pi 4. Coral Dev board saw the most improvement in performance by model quantization.

Lastly, the Arduino lacks behind with its microcontroller processor, taking `57534.297` seconds to inference 5000 images (∼`11.509` seconds per inference). This may seem like an enormous drawback compared to the other SoCs, but is actually quite acceptable for actual use cases when also taking power consumption during inference and idle state into account.

2) Inference performance with utilization of the dedicated AI units: Based on the optimization compatibility of AI units for various pre-trained Tensorflow models on different devices listed in the Table II, the results are shown in Figure 4. By looking at the performance, the respective devices naturally perform significantly better when optimized with their respective API (except for Tinker Edge R with MobileNetV1).

The Google Coral Dev Board delivers the best performance for MobileNetV2 (using *MobileNetV2 Quant. Lite*) with the inference time of `20.788` seconds (∼`0.004` seconds per inference). Though it must be considered that the Nvidia Jetson Nano doesn't have the advantage of running the quantized version of the model. If that is taken into account, the Jetson Nano outperforms the other devices when it comes to inference the non-quantized version in Tensorflow (shown under *MobileNetV2* in Figure 4). This is the case for both the Tensorflow-TensorRT optimized model with the inference time of `103.142` seconds (∼`0.020` seconds per inference), as well as the TensorRT framework with the inference time

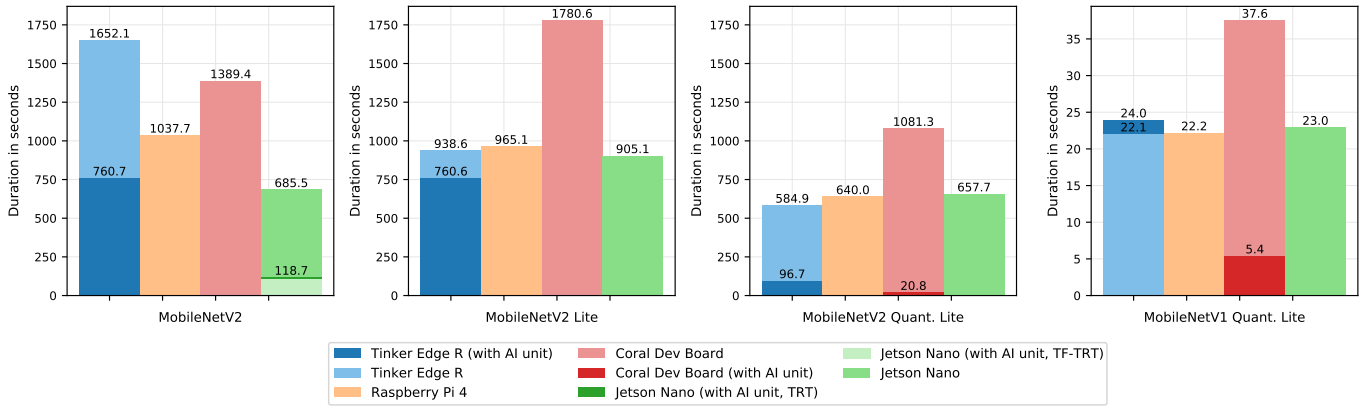


Fig. 4: Time taken for performing inference (1×5000 images test run) on various devices when using only CPU, as well as when utilizing the dedicated AI unit across different models and framework.

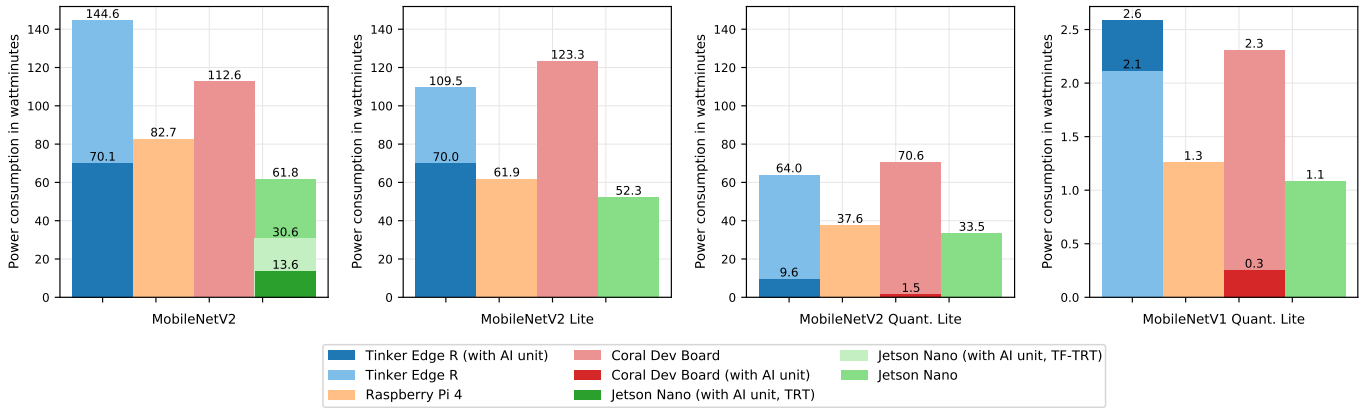


Fig. 5: Power consumption (in watt minutes) when performing inference (1×5000 images test run) by various devices when using only CPU, as well as when utilizing the dedicated AI unit across different models and frameworks.

TABLE III: Power consumption during idle state (in Watts) for various devices.

	Asus Tinker Edge R	**Raspberry Pi 4**	**Coral Dev Board**	**Nvidia Jetson Nano**	**Arduino Nano 33**
Idle (LAN)	4.932	2.643	3.081	1.391	✖
Idle (no LAN)	4.776	2.100	2.757	0.903	0.036

of `118.737` seconds (∼`0.023` seconds per inference), using the model in ONNX format.

We also see the conversion from Tensorflow to Tensorflow Lite (shown under *MobileNetV2* and *MobileNetV2 Lite* in Figure 4) keeps the structure of the model, since the MobileNetV2 model in Frozen Graph format (taking `760.7` seconds) and TFLite format (taking `760.6` seconds) perform similar to each other when ported to the RKNN-Toolkit for Asus Tinker Edge R. This indicates that the previously mentioned performance deviations between the two frameworks are due to their implementation, rather than potential changes to the model during conversion.

B. Power consumption

Next, we illustrate how energy efficient the devices are when *continuously* performing the computations. Figure 5 shows the power consumption by various devices during inference when using only CPU, and the dedicated AI unit across different models. Additionally, for many use cases, deployed edge devices will not perform explicit computations continuously. Rather, a predefined number of maximum Frames per Second (FPS) (or minute) might be anticipated to reduce energy use. Therefore it is worth looking at the power draw by the devices during their idle state i.e. while the device is not performing explicit computations. The Table III shows the power consumption during idle state (in Watts) for various devices. In idle state, the Arduino Nano 33 BLE consumes the least energy of `0.036` Watts. This is only around 0.04% of the energy consumed by the most energy-efficient device evaluated here, the Jetson Nano (with LAN disabled).

1) Power consumption without utilization of the dedicated AI units: From Figure 5, in overall, when using only CPU, the Jetson Nano consumes the least power for all models

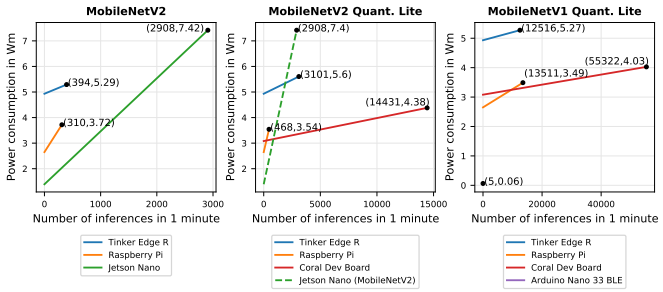


Fig. 6: Sparsely predicted power consumption at different inference rates for different models by various devices.

(61.8 wattminutes for *MobileNetV2*, 52.3 wattminutes for *MobileNetV2 Lite*, 33.5 wattminutes for *MobileNetV2 Quant. Lite* and 1.1 wattminutes for *MobileNetV1 Quant. Lite*), followed by the Raspberry Pi (82.7 wattminutes for *MobileNetV2*, 61.9 wattminutes for *MobileNet V2 Lite*, 37.6 wattminutes for *MobileNetV2 Quant. Lite* and 1.3 wattminutes for *MobileNetV1 Quant. Lite*) as shown in Figure 5. The trend we see here is very similar to the average power drawn by the devices in idle state (from Table III). However for the *MobileNetV2 Lite* and *MobileNetV2 Quant. Lite* models, the Asus Tinker Edge R consumed less power (109.5 wattminutes and 64.0 wattminutes respectively) as compared to Coral Dev Board (109.5 wattminutes and 64.0 wattminutes respectively) which is the opposite to their idle power consumption trend (Asus Tinker Edge R consuming more power than Coral Dev Board), as the Tinker Edge R's performance in Tensorflow Lite on the CPU is sufficient to compensate for its higher power draw in idle state.

Lastly, the Arduino falls behind by consuming approximately 61.879 watthours of power since it needs a longer time for doing inference.

2) Power consumption with utilization of the dedicated AI units: Analogously to inference performance, the devices are more energy efficient when using their AI unit (except for the Tinker Edge R with *MobileNetV1*). From Figure 5, the Coral Dev Board consumes the least power with just 1.543 wattminutes when considering across all *MobileNetV2* models and 0.254 wattminutes for the *MobileNetV1 Quant. Lite* model. Across all *MobileNetV2* models, the Tinker Edge R is the second best option with 9.619 wattminutes (shown under *MobileNetV2 Quant. Lite* in Figure 5), although the Jetson Nano is not far behind with 13.630 wattminutes (shown under *MobileNetV2* in Figure 5). In this regard, the Jetson Nano once again has the lowest value when looking at the non-quantized version only. For the Jetson Nano, it is also observed that, although it is faster to perform inference with the TF-TRT optimized model, but it consumes less power using the TensorRT framework (30.645 vs. 13.640 wattminutes, shown under *MobileNetV2* in Figure 5).

3) Power consumption prediction: Based on the data on energy consumption during the inference, we can now broadly predict the devices power consumption at different inference rates for different models as shown in Figure 6. The indicated values in the graphs shows the the maximum number of inferences that can be performed in one minute and power consumption (in wattminutes) during those inferences by the respective devices. Since power draw during initialization of the network, pre-processing and evaluation of the results are ignored, the actual power consumption is expected to be higher in a real-world scenario. For calculating the values, the best score in terms of power efficiency were taken for each model. As the SoCs had an Ethernet connection during testing, the respective values for idle power with LAN are used.

The Raspberry Pi is observably more energy efficient than the Coral Dev Board when performing inference using the *MobileNetV2 Quant. Lite* model (second sub-figure in Figure 6) at a inference rate of under 238 inferences per minute (50.65% of the time spent on AI computation). When also taking the Jetson Nano's results for the non-quantized Tensorflow version (*MobileNetV2* in Figure 6) into account, it outperforms the other devices for an inference rate of under 853 inferences per minute (less than 29.3% of the time spent on AI computation).

From all the results until now, the Arduino Nano 33 BLE did not hold up against the other devices. However, under different circumstances, the microcontroller performs significantly better i.e when inferences are performed in larger intervals. As seen from third sub-figure of Figure 6), at 5 frames per minute, the Arduino approximately consumes only 0.063 wattminutes which is considerably less than any other devices. This inference rate is also the maximum number of inferences that the device can perform in one minute for *MobileNetV1*.

C. Accuracy

We now review the accuracy of the models, which may differ as a result of model conversion and optimization. There are no explicit details about the accuracy of the quantized versions of the models, though expected to be similar to the non-quantized versions [50]. Table IV shows the models accuracy's (top-1 and top-5) in different frameworks for different devices.

Since only 5,000 out of 50,000 validation images were used, the claimed accuracy deviates from the test run in the native Tensorflow framework, which we can see in the Raspberry Pi's column of Table IV. No significant changes are observed when porting the models to the other frameworks, though results differ slightly when using RKNN-Toolkit, Tensorflow Lite Micro and when using the Edge TPU in Tensorflow Lite. More noticeable is the effect of quantization of the *MobileNetV2* model. Due to less precise weights, accuracy decreases by about 0.6% for Top-1 and 0.4% for Top-5 accuracy.

D. Performance on larger models

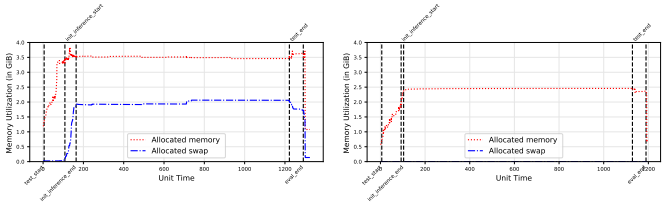
Until now, the models used for testing were comparably small. To demonstrate how model size affects the performance difference, the Raspberry Pi and Jetson Nano were additionally tested with an object detection model trained on the COCO [48] dataset from the Tensorflow Detection Zoo.

TABLE IV: Models accuracy's (top-1 and top-5) in different frameworks for different devices (1 x 5000 images).

	Claimed Accuracy	**Asus Tinker Edge R**	**Raspberry Pi 4**	**Google Coral Dev Board**	**Nvidia Jetson Nano**		**Arduino Nano 33 BLE**
		RKNN Toolkit	**None/ TF**	**PyCoral/ TFLite**	**TF-TRT**	**TensorRT**	**TFLite Micro**
MobileNetV2	0.75, 0.925	0.734, 0.907	0.733, 0.908	✖	0.733, 0.908	0.733, 0.908	✖
MobileNetV2 Lite	0.75, 0.925	0.734, 0.908	0.733, 0.908	✖	✖	✖	✖
MobileNetV2 Quant. Lite	0.75, 0.925	0.726, 0.904	0.727, 0.904	0.727, 0.904	✖	✖	✖
MobileNetV1 Quant. Lite	0.395, 0.644	0.364, 0.613	0.361, 0.611	0.359, 0.612	✖	✖	0.361, 0.612

SSD MobilenetV2 COCO has more than 17M parameters, compared to 6.06M on MobilenetV2.

Despite having no acceleration unit, the Raspberry Pi (taking `433.331` seconds for inference) now has similar performance to the Jetson Nano using the acceleration unit (taking `416.745` seconds for inference).



(a) Jetson Nano (b) Raspberry Pi

Fig. 7: Memory occupation (in GiB) over time (seconds) during 1 $\times$ 1000 images inference with SSD MobilenetV2 COCO in Tensorflow for two devices.

This can be attributed to the fact that, memory on the Jetson Nano is shared between GPU and CPU. Figure 7 shows the memory occupation over time for both the devices during 1 $\times$ 1000 images inference with SSD MobilenetV2 COCO in Tensorflow. From Figure 7a, we can observe that the internal RAM for Jetson Nano does not suffice to load the model into memory during initialization inference. As a result, the swap space on the much slower SD card is additionally used, making an inference using the GPU slower than when solely using the CPU as is the case with Raspberry Pi (shown in Figure 7b).

VI. Conclusion

In this work, we present and compare the performances in terms of inference time and power consumption of the four SoCs: Asus Tinker Edge R, Raspberry Pi 4, Google Coral Dev Board, Nvidia Jetson Nano, and one microcontroller: Arduino Nano 33 BLE for different models and frameworks. We also provide a method for measuring power consumption, inference time and accuracy for the devices, which can be easily extended to other devices.

Noticeably, the results for each model turn out to be quite different, depending on model size (In terms of operations and parameters), quantization, framework and anticipated number of inferences per time. Nevertheless we can draw some major conclusions for the following two main applications:

- **Best performing device for continuous AI computation**: The main factor here are the overall wattminutes consumed for inference a given number of images. For a Tensorflow model that can be quantized and converted to TFLite format, the Google Coral Dev Board delivers the best performance, both for inference time and power consumption. Not all models are developed to only include TFLite-compatible operations though. Therefore the Jetson Nano can run accelerated inference for Tensorflow models on its GPU. And as the entire Tensorflow framework can utilize the GPU, models can also be trained. While it might not be worth training a model on a single low-power Edge Device, distributed inference is already discussed by some recent papers [51].
- **Best performing device for sporadic AI computation**: When just sporadically performing AI computations, power efficiency is mostly dependent on power draw during idle. For small enough models, the Arduino Nano 33 BLE is by far the most power efficient option, though respectively low inference rates might affect usability. On the SoC side it depends on the model and the anticipated number of inferences per time. The Jetson Nano presumably outperforms the other devices for a low fraction of AI computation time (less than *29.3%* of the time for MobileNetV2). Otherwise, the Google Coral Dev Board is the most power efficient.

Extending the work to include other SoCs and evaluating using larger CNN models is prospective future work.

VII. Acknowledgement

This work was supported by the funding of the German Federal Ministry of Education and Research (BMBF) in the scope of the Software Campus program.

References

[1] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[2] A. M. Elkahky, Y. Song, and X. He, "A multi-view deep learning approach for cross domain user modeling in recommendation systems," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '15. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2015, p. 278–288. [Online]. Available: https://doi.org/10.1145/2736277.2741667

[3] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, ser. CVPR '14. USA: IEEE Computer Society, 2014, p. 1725–1732. [Online]. Available: https://doi.org/10.1109/CVPR.2014.223

[4] M. Zhang, F. Zhang, N. D. Lane, Y. Shu, X. Zeng, B. Fang, S. Yan, and H. Xu, "Deep learning in the era of edge computing: Challenges and opportunities," 2020.

[5] M. V. Barbera, S. Kosta, A. Mei, and J. Stefa, "To offload or not to offload? the bandwidth and energy costs of mobile cloud computing," in *2013 Proceedings IEEE INFOCOM*, 2013, pp. 1285–1293.

[6] X. Ma, T. Yao, M. Hu, Y. Dong, W. Liu, F. Wang, and J. Liu, "A survey on deep learning empowered IoT applications," *IEEE Access*, vol. 7, pp. 181 721–181 732, 2019.

[7] W. Hu, Y. Gao, K. Ha, J. Wang, B. Amos, Z. Chen, P. Pillai, and M. Satyanarayanan, "Quantifying the impact of edge computing on mobile applications," in *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2967360.2967369

[8] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[9] "OpenAI Presents GPT-3, a 175 Billion Parameters Language Model," Jul. 2020. [Online]. Available: https://news.developer.nvidia.com/openai-presents-gpt-3-a-175-billion-parameters-language-model/

[10] "Tensorflow/models," tensorflow, Feb. 2021. [Online]. Available: https://github.com/tensorflow/models

[11] M. Antonini, T. H. Vu, C. Min, A. Montanari, A. Mathur, and F. Kawsar, "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*, ser. AIChallengeIoT'19. New York, NY, USA: Association for Computing Machinery, 2019, p. 49–55. [Online]. Available: https://doi.org/10.1145/3363347.3363363

[12] B. Varghese, N. Wang, D. Bermbach, C.-H. Hong, E. de Lara, W. Shi, and C. Stewart, "A survey on edge performance benchmarking," 2020.

[13] S. Bianco, R. Cadene, L. Celona, and P. Napoletano, "Benchmark analysis of representative deep neural network architectures," *IEEE Access*, vol. 6, pp. 64 270–64 277, 2018.

[14] J. Jo, S. Jeong, and P. Kang, "Benchmarking gpu-accelerated edge devices," in *2020 IEEE International Conference on Big Data and Smart Computing (BigComp)*, 2020, pp. 117–120.

[15] A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, and P. Chen, "Function delivery network: Extending serverless computing for heterogeneous platforms," *Software: Practice and Experience*, vol. n/a, no. n/a. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2966

[16] "Tinker Edge R." [Online]. Available: https://tinker-board.asus.com/product/tinker-edge-r.html

[17] T. R. P. Foundation, "Raspberry Pi 4 Model B specifications." [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/

[18] "Dev Board." [Online]. Available: https://coral.ai/products/dev-board/

[19] "Jetson Nano Developer Kit," Mar. 2019. [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit

[20] "Arduino Nano 33 BLE." [Online]. Available: https://store.arduino.cc/arduino-nano-33-ble

[21] "Jetson Nano Brings AI Computing to Everyone," Mar. 2019. [Online]. Available: https://developer.nvidia.com/blog/jetson-nano-ai-computing/

[22] "A platform from google for local ai." [Online]. Available: https://coral.ai/about-coral/

[23] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1–12. [Online]. Available: https://doi.org/10.1145/3079856.3080246

[24] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[25] "Edge TPU performance benchmarks." [Online]. Available: https://coral.ai/docs/edgetpu/benchmarks/

[26] "TensorFlow models on the Edge TPU." [Online]. Available: https://coral.ai/docs/edgetpu/models-intro/#compatibility-overview

[27] "Asus tinker board series." [Online]. Available: https://tinker-board.asus.com/series.html

[28] "Asus tinker edge t." [Online]. Available: https://tinker-board.asus.com/product/tinker-edge-t.html

[29] "Raspberry pi products." [Online]. Available: https://www.raspberrypi.org/products/

[30] M. Croce, "Why you should run a 64 bit OS on your Raspberry Pi4," May 2020. [Online]. Available: https://matteocroce.medium.com/why-you-should-run-a-64-bit-os-on-your-raspberry-pi4-bd5290d48947

[31] A. Ltd, "M-Profile Architectures." [Online]. Available: https://developer.arm.com/architectures/cpu-architecture/m-profile

[32] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, D. Patterson, D. Pau, J.-s. Seo, J. Sieracki, U. Thakker, M. Verhelst, and P. Yadav, "Benchmarking TinyML Systems: Challenges and Direction," *arXiv:2003.04821 [cs]*, Jan. 2021, comment: 6 pages, 1 figure, 3 tables. [Online]. Available: http://arxiv.org/abs/2003.04821

[33] "TensorFlow Lite for Microcontrollers." [Online]. Available: https://www.tensorflow.org/lite/microcontrollers

[34] "Rockchip_Quick_Start_RKNN_Toolkit_V1.2.1_EN.pdf." [Online]. Available: https://repo.rock-chips.com/rk1808/rknn-toolkit_doc/Rockchip_Quick_Start_RKNN_Toolkit_V1.2.1_EN.pdf

[35] "TensorRT-Developer-Guide.pdf." [Online]. Available: https://docs.nvidia.com/deeplearning/tensorrt/pdf/TensorRT-Developer-Guide.pdf

[36] "Nvidia tensorrt." [Online]. Available: https://developer.nvidia.com/tensorrt

[37] "Onnx/tensorflow-onnx," Open Neural Network Exchange, Mar. 2021. [Online]. Available: https://github.com/onnx/tensorflow-onnx

[38] "TensorFlow Lite guide." [Online]. Available: https://www.tensorflow.org/lite/guide

[39] "TensorFlow Lite converter." [Online]. Available: https://www.tensorflow.org/lite/convert

[40] "Edge TPU Compiler." [Online]. Available: https://coral.ai/docs/edgetpu/compiler/

[41] "Model formats — TensorFlow Hub." [Online]. Available: https://www.tensorflow.org/hub/model_formats

[42] "SavedModels from TF Hub in TensorFlow 2 — TensorFlow Hub." [Online]. Available: https://www.tensorflow.org/hub/tf2_saved_model

[43] "TF1 Hub format — TensorFlow Hub." [Online]. Available: https://www.tensorflow.org/hub/tf1_hub_module

[44] "Tf.compat.v1.graph_util.convert_variables_to_constants." [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/compat/v1/graph_util/convert_variables_to_constants

[45] "Systemd." [Online]. Available: https://www.freedesktop.org/wiki/Software/systemd/

[46] "The Network Time Protocol (NTP) Distribution." [Online]. Available: https://www.eecis.udel.edu/~mills/ntp/html/index.html#info

[47] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet large scale visual recognition challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[48] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, "Microsoft COCO: Common objects in context," 2014, cite arxiv:1405.0312Comment: 1) updated annotation pipeline description and figures; 2) added new section describing datasets splits; 3) updated author list. [Online]. Available: http://arxiv.org/abs/1405.0312

[49] "Mckenm/HanmaTekPSUCmd." [Online]. Available: https://github.com/mckenm/HanmaTekPSUCmd
[50] "TensorFlow Model Garden GitHub," tensorflow, Jan. 2021. [Online]. Available: https://github.com/tensorflow/models
[51] L. Zhou, H. Wen, R. Teodorescu, and D. H. C. Du, "Distributing Deep Neural Networks with Containerized Partitions at the Edge," p. 7.