Resource Characterisation of Personal-Scale Sensing Models on Edge Accelerators

Mattia Antonini*
FBK CREATE-NET and University of Trento
Trento, Italy
m.antonini@fbk.eu

Tran Huy Vu*
Singapore Management University
Singapore, Singapore
hvtran.2014@smu.edu.sg

Chulhong Min
Nokia Bell Labs
Cambridge, UK
chulhong.min@nokia-bell-labs.com

Alessandro Montanari
Nokia Bell Labs
Cambridge, UK
alessandro.montanari@nokia-bell-labs.com

Akhil Mathur
Nokia Bell Labs and UCL
Cambridge, UK
akhil.mathur@nokia-bell-labs.com

Fahim Kawsar
Nokia Bell Labs
Cambridge, UK
fahim.kawsar@nokia-bell-labs.com

ABSTRACT

Edge accelerator is a class of brand-new purpose-built System On a Chip (SoC) for running deep learning models efficiently on edge devices. These accelerators offer various benefits such as ultra-low latency, sensitive data protection, and high availability due to their locality and are opening up interminable opportunities for building sensory systems in the real world. Naturally, in the context of sensory awareness systems, e.g., IoT, wearables, and other sensory devices, the emergence of edge accelerators is pushing us to rethink how we design these systems at a personal-scale. To this end, in this paper we take a closer look at the performance of a set of edge accelerators in running a collection of personal-scale sensory deep learning models. We benchmark eight different models with varying architectures and tasks (i.e., motion, audio, and vision) across seven platform configurations with three different accelerators including Google Coral, NVidia Jetson Nano, and Intel Neural Compute Stick. We report on their execution performance concerning latency, memory, and power consumption while discussing their current workflows and limitations. The results and insights lay an empirical foundation for the development of sensory systems on edge accelerators.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; **Embedded software**.

KEYWORDS

resource characterisation, edge accelerators, sensing models

*This work was done when the authors were on an internship at Nokia Bell Labs.

ACM Reference Format:
Mattia Antonini, Tran Huy Vu, Chulhong Min, Alessandro Montanari, Akhil Mathur, and Fahim Kawsar. 2019. Resource Characterisation of Personal-Scale Sensing Models on Edge Accelerators. In *First International Workshop on Challenges in Artificial Intelligence and Machine Learning (AIChallengeIoT'19), November 10–13, 2019, New York, NY, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3363347.3363363

1 INTRODUCTION

Personal-scale sensory systems are increasingly pushing the inference part of AI models to edge devices such as IoT, smartphones, wearables, etc. This transition offers attractive benefits concerning privacy, performance, and cost. In the last 18 months, this shift has resulted in the emergence of a brand-new class of neural chips aimed at inferences at the edge. The proposition is remarkable; for the first time, we can move away from software accelerators and push cloud-scale models into edge devices without compromising accuracy. Naturally, these edge accelerators are uncovering exciting opportunities for building powerful applications with complicated learning objectives and demanding computations. There have been several attempts to understand the performance characteristics of human sensing models on smart devices like smartphones and commodity devices [1, 13]. However, the characterisation of edge accelerators is at a very early stage. To this end, we take a systematic look at a set of edge accelerators, their working principles, and performance in executing a variety of human sensing models.

We benchmark seven different accelerator configurations (Google Coral Dev Board, Google Coral Accelerator with Raspberry Pi (hereinafter, RPi) 4B and 3B+, NVIDIA Jetson Nano with TensorFlow GPU and TensorRT, and Intel Neural Compute Stick with RPi 4B and 3B+) running eight deep learning models with three tasks (motion, audio, and image). We report on their execution performance concerning memory, execution time, and energy overhead and share insightful observations that lay an empirical foundation for both the evolution of these accelerators and their usage in sensory systems.

In what follows, we first present the different accelerators and their working principles. Then we discuss briefly the sensing models we use in the study, followed by the systematic report on performance metrics of the accelerators for the models. Finally, we conclude the paper by sharing key insights from this study.

	Coral Dev Board	**Coral Accelerator + RaspberryPi 3B+**	**Coral Accelerator + RaspberryPi 4B**	**NVIDIA Jetson Nano**	**Intel NCS2 + Raspberry Pi 3B+**	**Intel NCS2 + Raspberry Pi 4B**
CPU	Quad-Core Cortex A53	Quad-Core Cortex A53	Quad-Core Cortex A72	Quad-Core Cortex A57	Quad-Core Cortex A53	Quad-Core Cortex A 72
Memory	1 GB LPDDR4	1 GB LPDDR2	2 GB LPDDR4	4 GB LPDDR4	1 GB LPDDR2	2 GB LPDDR4
AI Chip	Google EdgeTPU	Google EdgeTPU	Google EdgeTPU	128 Core Maxwell GPU	Intel Movidius Myriad X VPU with 16 SHAVE cores	Intel Movidius Myriad X VPU with 16 SHAVE cores
On-Chip Memory	8 MB	8 MB	8 MB	Shared with CPU	512 MB LPDDR4 + 2.5 MB Centralized	512 MB LPDDR4 + 2.5 MB Centralized
AI Chip Interface	PCIe	USB 2.0	USB 3.0	PCIe	USB 2.0	USB 3.0
AI Chip OPs	4 TOPs	4 TOPs	4 TOPs	472 GFLOPs	1 TOPs	1 TOPs

Table 1: Specification of the hardware platforms used in the study.

2 STUDY PRELIMINARIES

2.1 Hardware Platforms

Different companies have proposed hardware solutions to accelerate the execution of deep learning algorithms at the edge of the network. In this study, we consider seven different configurations with three types of edge accelerators. Table 1 reports their hardware specifications; we consider two TensorFlow frameworks for Jetson Nano, Tensorflow GPU[1] and TensorRT[2].

Google Coral: In summer 2018, Google announced the edge version of its Tensor Processing Unit (TPU) platform known as EdgeTPU under the brand name Coral. The EdgeTPU is an application specific integrated circuit designed to deliver up to 4 Tera OPerationS (TOPS) per second using a power budget of 2 watts (2 TOPS/watt). This chip supports only signed integer operations at 8 and 16 bits and it comes with approximately 8 MB of on-chip RAM used to cache the model's parameters. Since this board has been strictly designed for optimal inference, it currently supports only TensorFlow Lite models that meet specific requirements [4] (e.g., parameter quantisation). The EdgeTPU is available in two flavours: as dev-kit called *Coral Dev Board* (See Figure 1a) and as USB dongle called *Coral Accelerator*. Coral Dev Board is a single board computer that hosts onboard RAM, storage, and other peripherals. As host device for the Coral Accelerator, we use Raspberry 4B (See Figure 1b) and 3B+. The biggest difference between Raspberry Pi (hereinafter, RPi) 4B and 3B+ regarding our benchmark study is the AI chip interface. RPi 4B supports USB 3.0 (with a maximum rate of 5 gigabits per second), whereas RPi 3B+ supports USB 2.0 (with a maximum rate of 480 megabits per second).

Jetson Nano: In March 2019, NVIDIA has announced and made available a new GPU-powered board, known as Jetson Nano, targeting the maker community (See Figure 1c). This board hosts a 64-bit quad-core Arm Cortex-A57 CPU and an NVIDIA Maxwell GPU with 128 CUDA-cores able to deliver up to 472 GFLOPs running float operations. CPU and GPU share a common bank of 4 GB of LPDDR4 RAM, which requires the tuning of the memory reservation between CPU and GPU. Since this board runs a full-fledged operating system derived from Ubuntu, the board natively supports TensorFlow 1.x compiled with GPU support and TensorRT 5.

[1]https://www.tensorflow.org/install/gpu
[2]https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.html

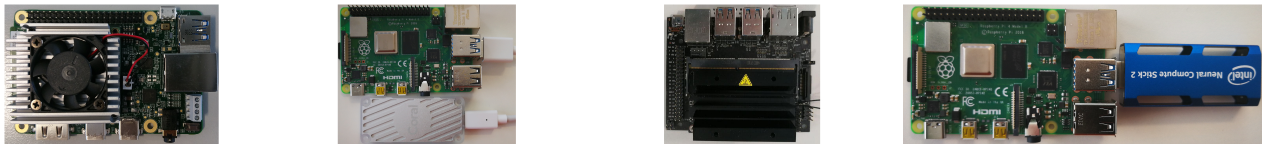
Intel NCS2: Intel has made available the new Intel Movidius Myriad X Vision Processing Unit (VPU), a low-power System-on-Chip (SoC) designed to accelerate deep-learning deployments and computer vision applications. This chip includes several processors and computing units optimised for high parallelism and DNN inference making it capable of running up to 4 TOPS with a power budget of 1.5 watts. The VPU is available in two different in-package configurations: without in-package additional RAM and with 4GBits (512 MBytes) in-package RAM. Intel has released a USB 3.0 dongle known as Intel Neural Compute Stick 2 (NCS2) that hosts the Movidius Myriad X VPU with 4GBit of RAM. This USB stick can be plugged as a co-processor to speed-up the inference of neural networks. NCS2 requires the model to be optimised using the OpenVINO framework [9]. We also consider Raspberry Pi 4B (See Figure 1d) and 3B+ as the main board for benchmarking NCS2.

2.2 Compilation Workflow

Since edge accelerators have different constraints and requirements, different optimisations need to be applied to fully exploit the hardware acceleration. Figure 2 presents the required steps for the edge accelerators we use for our benchmark study. In this paper, we consider deep learning models that have been implemented with native TensorFlow or with Keras with TensorFlow as backend.

Jetson Nano: The first step is to train the algorithm by applying full-precision training which outputs a model with parameters expressed as 32bit floating-point numbers. Then, the model needs to be frozen to convert all the inner variables to constant and make the model ready for the inference phase and further optimisation. The frozen model can natively run on the Jetson Nano using native TensorFlow with GPU support. Jetson Nano also supports TensorRT, a library that optimises the execution of neural networks by replacing the implementations of some layers with more efficient ones. TF-TRT converter needs information including input tensor name and shape, precision mode (FP16 or FP32), size of the inference batch, and size of the reserved execution memory. The output is a TensorFlow-TensorRT frozen model ready to be deployed.

Intel NCS2: Intel NCS2 also needs the full-precision frozen model to generate a model compatible with it. Then, the model is converted using the OpenVINO model optimiser [10], a cross-platform tool that runs static analysis and adjustments of the model. The optimiser needs only the shape of the input tensor and the floating number precision (*e.g.*, FP16). It returns a set of files, known



(a) Google Coral Dev Board (b) Coral Accel. with RPi 4B (c) NVIDIA Jetson Nano (d) Intel NCS2 with RPi 4B

Figure 1: Hardware platforms used in the study.

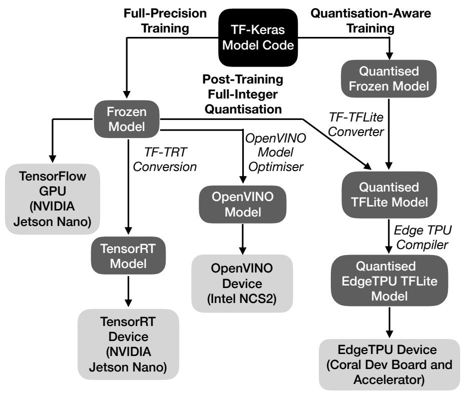

Full-Precision Training
TF-Keras Model Code
Quantisation-Aware Training
Quantised Frozen Model
Post-Training Full-Integer Quantisation
Frozen Model
TF-TFLite Converter
OpenVINO Model Optimiser
TF-TRT Conversion
TensorFlow GPU (NVIDIA Jetson Nano)
OpenVINO Model
Quantised TFLite Model
Edge TPU Compiler
TensorRT Model
OpenVINO Device (Intel NCS2)
Quantised EdgeTPU TFLite Model
TensorRT Device (NVIDIA Jetson Nano)
EdgeTPU Device (Coral Dev Board and Accelerator)

Figure 2: Compilation workflow

as Intermediate Representation (IR), that are used by the Inference Engine API to run the model over the Movidius Myriad X VPU.

Google Coral: Since EdgeTPU does not support floating-point parameters, it is essential to represent the model weights as signed-integer numbers, i.e., *quantisation.* The EdgeTPU runtime supports quantisation-aware training [5, 11] which performs parameter quantisation at training time. The model is frozen after this step and then converted to TensorFlow Lite format. As an alternative, from the v12 of the EdgeTPU runtime, it supports post-training full-integer quantisation [19]. This procedure quantises all the parameters and activations without re-training the model. It requires a small and representative dataset, which might be a part of the training set, to define the quantisation range. Note that, while quantisation-aware training requires the additional cost for re-training, higher accuracy is achievable as it is generally more tolerant to lower precision values. The last step is to feed the quantised TensorFlow Lite model to the EdgeTPU compiler [3]. The compiler verifies if the model meets the requirements [4]. It statically defines how weights are allocated in the Edge TPU on-chip memory and defines the execution of the TensorFlow Lite graph on the acceleration hardware.

Although the model meets the requirements, it is possible that some operations could not be supported by the EdgeTPU runtime. Then, the compiler tags them as *unsupported* and forces the execution of those and subsequent operations on the CPU instead of TPU. It is also possible that model's weights do not fit in the TPU on-chip memory but the operations are still executed on TPU. In this case, the weights are dynamically streamed from off-chip memory, e.g., RAM, to the on-chip memory, introducing additional latency.

2.3 Deep Learning Sensing Models

We select a broad range of deep learning sensing models tailored for motion, audio, and vision tasks which are key for personal-scale sensing. Table 2 summarises the architectures and properties of the models we benchmark. They cover diverse types of CNN architecture, e.g., with and without auxiliary branches, residual connections, depth-wise convolution, and fully connected layers.

Motion task: Motion sensors, e.g. accelerometer, gyroscope and magnetometer, are crucial components in smart devices as they provide rich information about user context [2, 20]. One of the most desired applications of motion tasks is human activity recognition (HAR). For the HAR model, we select Aroma [16]. It consists of two hierarchical classifiers. The first classifier exploits 8 convolution layers to automatically learn low-level features from the distribution of sensor data. These low-level features are then classified into different simple activities, e.g., standing and walking, using a fully connected layer and a softmax classifier. On top of this classifier, an LSTM model is applied to learn and extract meaningful complex activities, e.g., commuting, from temporal relationships in the low-level features over time. However, since the current accelerators do not support LSTM modules, we profile only the convolutional part of the model.

Audio task: Audio understanding is always on the front line of machine learning and enables a variety of sensing tasks. Using edge accelerators is promising to enable on-device audio processing, which provides clear benefits such as privacy assurance and low latency. In this paper, we consider two different audio tasks, keyword spotting and emotion recognition. Keyword Spotting is a vital component in virtual assistant applications, e.g. Google assistant. To this end, we use a deep keyword spotting (DKWS) model [15], which is a three-layer deep convolutional neural network. It is capable of detecting several spoken keywords, e.g., yes and no. The goal of emotion recognition task is to capture human psychological state unobtrusively in daily lives using the speaker's utterance. We follow the implementation proposed in [12] which comprises of 3 CNN layers, a fully connected layer, and a softmax classifier to classify four different emotions, including neutral, upset, happy, and angry. For the benchmark, we focus only on the model execution and exclude the pre-processing steps such as the extraction of Mel-Frequency Cepstral Coefficients (MFCC) features.

Image task: Image recognition is one of the most active areas of machine learning with many applications [14]. Given the popularity of these models, we profile 5 different types of neural networks, including SqueezeNet V1.0 [8], MobileNet V1 [6], EfficientNet [18], Inception V1 [17], and DenseNet121 [7].

These models cover a variety of network architectures. SqueezeNet introduced Fire modules which makes use of 1x1 convolution to squeeze the number of input channels and a 3x3 filter to reduce the total number of parameters. MobileNet V1 introduced depth-wise convolution, which applies convolutions on each channel before combining the filters to reduce the number of parameters. Recently

Task	Model	Architecture	Characteristic	Parameters (Milions)
Motion	Aroma (**A**) [16]	CNN+DNN + Residual Connections	Excluding LSTM	0.385
Audio	Audio Classification [12] (Emotion Only) (**E**)	CNN + FC	No MFCC extraction	0.249
	DKWS (**D**) [15]	CNN + FC	No MFCC extraction	0.923
Vision	SqueezeNet V1.0 (**S**) [8]	CNN + FC	Fire modules	1.235
	MobileNet V1 (**M**) [6]	CNN + FC	Traditional and depthwise convolution layers	4.232
	EfficientNet-EdgeTPU (**E2**) [18]	CNN + FC	Traditional CNN	5.440
	Inception V1 (**I**) [17]	CNN + FC	Inception Modules	6.618
	DenseNet121 (**D2**) [7]	CNN + FC + Residual Connections	Each Layer is connected to all the previous layers	7.978

Table 2: Specification of sensing models used in the study.

Google has developed EfficientNet – a new family of CNN architecture which can be optimised for different platforms; we use EfficientNet-EdgeTPU, which is optimised for the Google EdgeTPU. EfficientNet utilises architectural search (grid search on depth and width) to find a near-optimal architecture, which optimises both depth and width of a neural network. Inception V1 includes 22 convolution layers with branches of 1x1, 3x3 and 5x5 convolutions and a fully connected layer. DenseNet121 contains both convolution layers and Dense blocks which maintain residual connections from one layer to all previous layers in the same block.

2.4 Scope of the benchmark

Our goal is to investigate the resource characteristics of edge accelerators under a range of deep learning sensing models. However, there are a number of compilation and optimisation parameters that affect the resource characteristics and inference accuracy as well. For example, for the precision mode, FP16 (half-precision point) could occupy less memory and lower inference latency compared to FP32 (full-precision point), but could result in accuracy degradation. In this paper, as an initial step, we select personal-scale sensing models as a key independent variable and aim at investigating their resource characteristics. To this end, we set the compilation and optimisation parameters to the default values used in each edge accelerator. For example, for the precision mode, we used FP32 and FP16 for TensorFlow GPU and TensorRT on Jetson Nano, respectively. Intel NCS2 was also set to FP16. We leave the investigation of the compilation and optimisation parameters and their impact on the accuracy as future work.

In this aspect, we do not include other operations into the benchmark, which are required for the entire sensing pipeline such as sensing, data transmission, and data management. It is because their resource characteristics are not affected by edge accelerators.

3 PERFORMANCE BENCHMARKS

We conduct a set of benchmarks to characterise the resource usage of personal-scale sensing models on edge accelerators. We consider end-to-end model performance metrics of memory usage, execution time, and energy consumption. Given the proprietary nature of each accelerator and the limited availability of APIs, we could not include accelerator-related metrics such as TPU usage. We expect that the outcome of these experiments uncovers the feasibility of running sensing models and applications on edge accelerators.

3.1 Experimental Setup

To systematically explore the resource characteristics, we develop a benchmark script that executes the sensing models repeatedly and measures memory usage and execution time. We perform 20,000 separate inferences for every model on each platform and report the average figures. For all the experiments, we use a batch size of 1 to consider applications where the models need to process and label sensory inputs as quickly as possible, without additional latency introduced by batching several data points. For energy measurements, we use a Monsoon Power monitor.

We consider three steps in the model lifetime: *loading*, *warm-up*, and *inference*. In the loading, the model is loaded into the accelerator's on-chip memory. The warm-up refers to the first execution of the model, and the inference is for the subsequent executions. We separate the warm-up from the inference since accelerator run-time completes hardware initialisation (e.g., model caching and memory allocation) upon the first request of the model execution.

3.2 Memory Usage

We investigate the memory footprint, which is known to be a key resource bottleneck in the processing of deep learning models on embedded devices due to the large amount of parameters.

The first observation is that, when a model is executed on an accelerator, the memory is gradually allocated at three different times: (1) when the model is loaded, (2) at the first inference (warm-up) and (3) during subsequent inferences. It is important to notice that the loading and warm-up memory remains allocated for all subsequent inferences and it is deallocated only when the model is unloaded. We further discover that the way the memory is handled depends on the hardware architecture of the accelerator and also on its runtime software. For the accelerators with on-chip dedicated memory, i.e., Coral Dev (Figure 3a), Coral Accelerator, and NCS2 (Figure 3b), the compilation pipeline optimises the model to keep as much of the network as possible on the on-chip memory to ensure low latency access, thereby resulting in low utilisation of the memory on the host device. Even for large models (e.g., EfficientNet, Inception V1 and DenseNet), we observe that only 13–18 MB are allocated during the loading and the warm-up phases on the host memory of the Coral Dev Board and on the Raspberry Pi memory used with the Coral Accelerator. Also on the Intel NCS2, the host memory allocated is a bit higher than the Coral devices, but still marginal. For example, the largest amount of allocated memory is

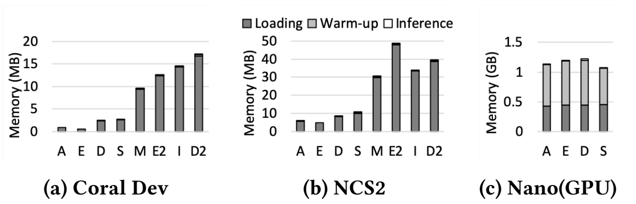


Figure 3: Host device memory footprint (notice the different scale for the y-axis).

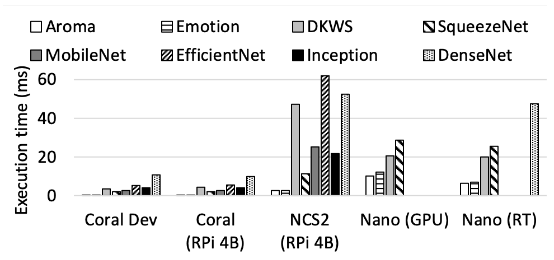


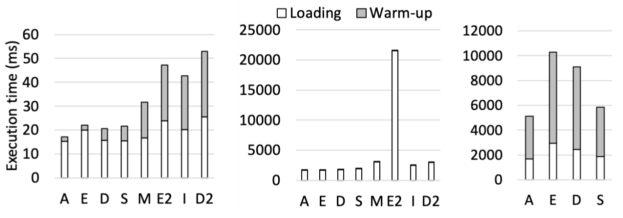
Figure 4: Execution time on different platforms.

for the EfficientNet model with about 50 MB of memory allocated for loading and warm-up (for the same model only 14 MB are allocated when using the Coral devices). For the inference phase, the host memory used is even lower, we measure less than 10 KB across all models both on Coral devices and NCS2.

On the Jetson Nano, however, we notice that significant more memory is allocated during loading and warm-up, as shown in Figure 3c for the Tensorflow GPU runtime. Only between 1 MB and 10 MB are used during inference instead. We hypothesise that this is because the TensorFlow runtime is not optimised for constrained devices with limited memory. The implication is that, since on Nano the memory is shared between the CPU and the GPU (i.e., there is no dedicated memory for the GPU), the more memory is used for a deep learning model the less is available for the operating system and other processes running on the CPU. As a consequence, we could not run the large models (MobileNet, EfficientNet, Inception, and DenseNet), because the free memory is not enough to load and perform the warm-up phase of these models. This is important because a real system would need to execute other tasks in addition to the model inference (e.g., communication, user interface and data logging) requiring memory for each of these tasks. This might become impossible if most of the free memory is consumed by model execution, as on the Jetson Nano. Therefore we observe that devices with dedicated on-chip memory and with software pipelines capable of optimising the models' memory requirements, such as Coral Dev, Coral Accelerator and NCS2, are preferable for systems which need to run processes in addition to the model execution.

3.3 Execution time

We look into the execution time of the model inference, which is a key metric for sensory systems that need to react quickly to input data. Figure 4 shows the inference time for different platforms. The results show a couple of interesting findings. First, the execution time is largely different, depending on the edge platform. In general, Coral Dev and Coral (RPi 4B) outperform other platforms. For example, one execution of the SqueezeNet model takes 2 ms both



(a) Coral Dev **(b) NCS2 (RPi 4B)** **(c) Nano (GPU)**

Figure 5: Execution time for loading and warm-up operations. X-axis reports shortened model names as in Table 2.

on Coral Dev and Coral (RPi 4B), whereas it takes 11 ms, 29 ms, and 26 ms on NCS2 (RPi 4B), Nano (GPU), and Nano (RT), respectively.

Second, as expected, the inference is faster for simpler models. For example, the execution time on Coral Dev is 0.5, 0.5, 3.5, 2.1, 2.7, 4.2, and 10.9 ms for Aroma, Emotion, DKWS, SqueezeNet, MobileNet, Inception, and DenseNet (see Table 2 for the number of parameters). DenseNet is the slowest on the Coral devices because the entire model cannot be allocated on the on-chip accelerator's memory (~8 MB) and therefore part of the parameters were allocated on the host memory (1.9 MB). This causes additional latency because parameters need to be moved between the host memory and the accelerator on-chip memory. The trend showing that simpler models run faster is observable also on different platforms. However, NCS2 (RPi 4B) is an exception to this tendency. The execution time of DKWS is 47 ms, whereas that of SqueezeNet, MobileNet, and Inception is 11ms, 25ms, and 22ms. We hypothesise that this is because DKWS has an unusual kernel size in its first convolutional layer (i.e., 8x20) which translates to heavy computation on the input data and possibly causes inefficiency because the Movidius chip is not optimised for this kernel size. We find a similar behaviour on EfficientNet and DenseNet with the Intel NCS2. While the number of parameters of EfficientNet is lower than that of DenseNet, its execution time is much higher on the Intel NCS2. We conjecture that this is because EfficientNet has been designed and optimised for the EdgeTPU architecture.

We delve deeper into the execution time of the loading and warm-up steps as shown in Figure 5. We omit the results of Coral (RPi 4B) and Nano (RT) since they show a similar trend to Coral Dev and Nano (GPU), respectively. Interestingly, edge platforms show different tendency. We notice that the loading and warm-up times for all models on Coral Dev are always below 30 ms while the NCS2 and Jetson Nano take several seconds. Knowing loading and warm-up times of these accelerators is important in reactive systems where different models need to be executed on-demand to respond to certain sensory inputs. In this context, models are dynamically loaded to perform a few inferences and then unloaded. Large loading and warm-up times will reduce the performance of the system, making it difficult to promptly respond to input data. From our benchmarks, we can conclude that Coral Dev is suitable to support reactive systems where multiple models need to be loaded and unloaded over time while NCS2 and Jetson Nano are more suitable for applications where a single model is used for long periods of time, amortising the loading and warm-up cost.

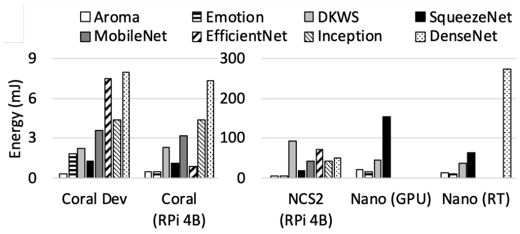


Figure 6: Energy consumption on different platforms.

Coral Dev	Coral (RPi 4B)	NCS2 (RPi 4B)	Nano
2.1	4.2	3.6	1.2

Table 3: Idle power of platforms (W).

3.4 Energy

Energy is a precious resource in battery-powered edge accelerators. Here, we define the energy overhead as the energy which is additionally consumed for the model execution. To obtain the net energy increase, we measure the difference between the average power consumed during the *model execution* and when the board is *idle*, and then multiply it by the model execution time.

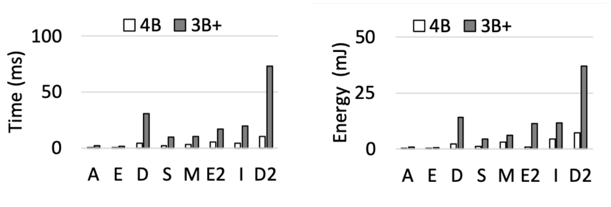
Interestingly, as shown in Figure 6, the energy overhead varies much depending on the accelerator. For example, Coral Dev and Coral (RPi 4B) mostly consume less than 10 mJ for a single execution regardless of the model; the only exception is DenseNet on Coral Dev (18.8 mJ). On the other hand, the energy overhead ranges from 5 mJ to 274 mJ on NCS2 (RPi 4B) and Jetson Nano. We can also observe that the TensorFlow framework largely impacts the energy overhead, even with the same platform. On Jetson Nano, TensorFlow GPU generally consumes more energy than TensorRT. This is probably because TensorFlow GPU is not optimised for energy efficiency and it takes longer as well for the model execution.

Table 3 shows the power draw in the idle state, i.e., when no operation is being executed. Interestingly, the power also varies much depending on the hardware specification. We notice that the devices that are self-contained (i.e., Coral Dev and Jetson Nano) draw less power when idle compared to the accelerators which require an host device to operate (i.e., NCS2 and Coral Accelerator). In fact, the power draw of the Raspberry Pi 4B alone is 2.9 W. We omit the energy overhead of the loading and warm-up operations. They have relatively little impact on the battery life as they need to be executed only once.

3.5 Performance Comparison (RPi 4B and 3B+)

According to the hardware specification, the main difference between RPi 4B and 3B+ is the interface with AI Chip as described in Table 1, i.e, USB 2.0 on RPi 3B+ and USB 3.0 on RPi 4B. In this subsection, we quantify the impact of the interface on the performance of the model execution. Here, we focus on the latency and power consumption, which are mainly affected by the interface.

Execution time: Figure 7a shows the execution time of Coral Accelerator; we omit the result on NCS2 due to the page limit. As expected, RPi 4B takes a shorter time than RPi 3B+ by virtue of its fast transmission via USB 3.0. Interestingly, the performance gap, i.e., the difference of the execution time between RPi 4B and RPi 3B+, is different depending on the type of the accelerator. With



(a) Execution time **(b) Energy overhead**

Figure 7: Performance comparison between RPi 4B and 3B+

the Coral accelerator, the ratio of RPi 3B+ to RPi 4B ranges from 3.1 (EfficientNet) to 7.3 (DenseNet). However, with Intel NCS2, the ratio mostly remains less than 1.7, except the SqueezeNet (2.4).

Energy: We also compare the energy overhead as shown in Figure 7b. The results show that, in general, RPi 3B+ consumes more energy for the model execution; the only exception is DKWS on Intel NCS2. The main reason is due to the increase in the execution time. However, the idle power of RPi 3B+ is much lower than RPi 4B. The idle power of RPi 3B+ is 2.4 W and 2.8 W with Coral Accelerator and Intel NCS2, respectively. This makes the average power of RPi 4B during the model execution (including the idle power) higher than that of RPi 3B+. For example, the average power of the Aroma model on RPi 4B is 5.0 W, whereas that on RPi 3B+ is 2.9 W. In this aspect, we estimate that the battery life of RPi 3B+ will be longer than that of RPi 4B assuming a battery with the same capacity.

4 OUTLOOK

We attempted to characterise the resource performance and suitability of personal-scale sensory models on a wide variety of edge accelerators. Beyond the numbers, our study further offers useful insights for the development of personal sensing system on top of the edge accelerators. First, as described in 2.2, the execution path of deep learning models on edge accelerator is not optimised, yet. For example, on Google Coral platforms, if an operation in the model is tagged as *unsupported*, the execution path is statically determined by putting the whole subsequent operations (including the untagged one) to CPU. It implies that the position of the untagged operation affects the performance of the model significantly. Second, the interface between CPU and AI chip is a critical bottleneck. As reported in Section 3.5, even with the same Coral accelerator, USB 3.0 accelerates the execution time by three to seven times compared to USB 2.0. Last, careful scheduling is needed to support multiple sensing models. This is because the dynamic change in sensing models incurs a significant overhead, e.g., as shown in the cost of loading and warmup on Jetson Nano (Section 3.3).

For the automated and scalable benchmark, we prototyped an end-to-end benchmarking toolkit. As a core component for the resource benchmark, it takes a sensing model and a target platform as input. Then, it converts the given model to the platform-specific model binary as described in Figure 2 and performs the benchmark. As future work, we envision this toolkit as a full-fledged, comprehensive framework for edge accelerators. If developers provide the sensing model, the test dataset, and the execution requirements, e.g., latency, accuracy, and energy budget, the toolkit can automatically test the given model on various edge accelerators in the background and recommend the most suitable platform providing an in-depth report on the expected performance.

REFERENCES

[1] Mario Almeida, Stefanos Laskaridis, Ilias Leontiadis, Stylianos I Venieris, and Nicholas D Lane. 2019. EmBench: Quantifying Performance Variations of Deep Neural Networks across Modern Commodity Devices. In *The 3rd International Workshop on Deep Learning for Mobile Systems and Applications*. ACM, 1–6.

[2] Saisakul Chernbumroong, Shuang Cang, Anthony Atkins, and Hongnian Yu. 2013. Elderly activities recognition and classification for applications in assisted living. *Expert Systems with Applications* 40, 5 (2013).

[3] Coral Team. 2019. Edge TPU Compiler. https://coral.withgoogle.com/docs/edgetpu/compiler/, Last accessed on Monday 23rd September, 2019.

[4] Google Coral Team. 2019. EdgeTPU Model Requirements. https://coral.withgoogle.com/docs/edgetpu/models-intro/#model-requirements, Last accessed on Monday 23rd September, 2019.

[5] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. 2016. Hardware-oriented Approximation of Convolutional Neural Networks. *CoRR* abs/1604.03168 (2016). arXiv:1604.03168 http://arxiv.org/abs/1604.03168

[6] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[7] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[8] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[9] Intel. 2019. Intel Distribution of OpenVINO Toolkit. https://software.intel.com/en-us/openvino-toolkit, Last accessed on Monday 23rd September, 2019.

[10] Intel. 2019. Model Optimizer Developer Guide. https://docs.openvinotoolkit.org/latest/_docs_MO_DG_Deep_Learning_Model_Optimizer_DevGuide.html, Last accessed on Monday 23rd September, 2019.

[11] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *CoRR* abs/1712.05877 (2017). arXiv:1712.05877 http://arxiv.org/abs/1712.05877

[12] Shin Katayama, Akhil Mathur, Tadashi Okoshi, Jin Nakazawa, and Fahim Kawsar. 2019. Situation-Aware Conversational Agent with Kinetic Earables. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 657–658.

[13] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, and Fahim Kawsar. 2015. An early resource characterization of deep learning on wearables, smartphones and internet-of-things devices. In *Proceedings of the 2015 international workshop on internet of things towards applications*. ACM, 7–12.

[14] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E Alsaadi. 2017. A survey of deep neural network architectures and their applications. *Neurocomputing* 234 (2017).

[15] Akhil Mathur, Anton Isopoussu, Fahim Kawsar, Nadia Berthouze, and Nicholas D Lane. 2019. Mic2Mic: using cycle-consistent generative adversarial networks to overcome microphone variability in speech systems. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*. ACM, 169–180.

[16] Liangying Peng, Ling Chen, Zhenan Ye, and Yi Zhang. 2018. AROMA: A Deep Multi-Task Learning Based Simple and Complex Human Activity Recognition Method Using Wearable Sensors. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 2 (2018), 74.

[17] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[18] Mingxing Tan and Quoc V Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. *arXiv preprint arXiv:1905.11946* (2019).

[19] TensorFlow Team. 2019. Post-Training quantization. https://www.tensorflow.org/lite/performance/post_training_quantization#full_integer_quantization_of_weights_and_activations, Last accessed on Monday 23rd September, 2019.

[20] Rui Wang, Fanglin Chen, Zhenyu Chen, Tianxing Li, Gabriella Harari, Stefanie Tignor, Xia Zhou, Dror Ben-Zeev, and Andrew T Campbell. 2014. StudentLife: assessing mental health, academic performance and behavioral trends of college students using smartphones. In *Proceedings of the 2014 ACM international joint conference on pervasive and ubiquitous computing*. ACM.