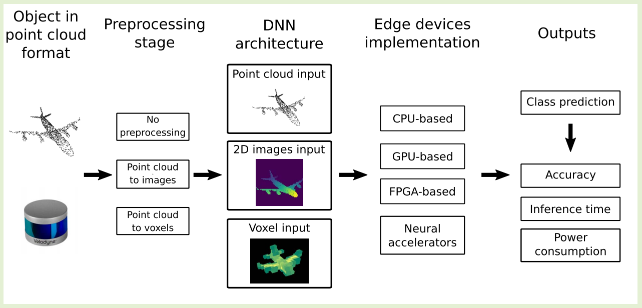
Characterizing Deep Neural Networks on Edge Computing Systems for Object Classification in 3D Point Clouds

Cristian Wisultschew, Alejandro Pérez, Andrés Otero, Gabriel Mujica, *Member, IEEE*, and Jorge Portilla, *Senior Member, IEEE*

***Abstract*—The current trend of shifting computing from the cloud to the edge of the Internet of Things is influencing deep learning applications. Moving intelligence closer to the point of need entails advantages in terms of performance, power consumption, security, and privacy. The problem arises with data sources that generate a massive amount of information, making data processing challenging for edge devices. This is the case of point clouds generated by LIDAR sensors. Implementations at the edge become even more challenging when heavy processing algorithms such as deep neural networks are selected. However, deep neural networks are the state-of-the-art solution to carry out object classification tasks as they provide the best results in terms of accuracy when working with high data volumes. This work demonstrates that the processing of point cloud-based sensors using deep neural networks at the edge is becoming feasible with the emergence of new devices with high computing capacity combined with reduced power consumption. In this regard, a characterization of first-in-class deep learning classification algorithms working with point cloud data as inputs and running over different state-of-the-art edge processing architectures is provided. A broad range of devices, including CPUs, GPU-based, SoC FPGA-based, and deep learning neural accelerators, have been evaluated in terms of inference time, classification accuracy, and power consumption. As a result, it demonstrates that neural accelerators with integrated host CPUs represent the best trade-off between power consumption and performance, making them a perfect solution for IoT applications at the edge level.**



***Index Terms*—Deep neural networks, edge computing, LIDAR, neural accelerators, object classification, point cloud.**

I. Introduction

A GROWING number of cyber-physical systems require perceiving the physical environment where they operate. To this end, visible imaging sensors have been traditionally used, enabling high-quality object detection and classification. However, there are specific applications, such as robotics, autonomous vehicles, or surveillance systems, in which it is also essential to locate objects in space. Determining the position of the detected objects using 2D data produced by visible cameras is highly challenging due to their depth ambiguity [1]. Alternatively, 3D sensors generating spatial information inherently, such as LIght Detection And Ranging (LIDAR) sensors, RGB-D cameras, and 3D scanners, can be employed. These sensors have been experiencing significant technical progress in recent times.

Among all the sensors generating 3D information, LIDARs stand out since they provide rich, dense, and precise spatial data in the form of point clouds. Additionally, this technology provides an increase in sensor robustness and scanning rate [2]. Since they are based on laser technology, LIDARs are not affected by ambient lighting conditions, unlike 2D cameras, whose precision decreases at night. For the same reason, LIDAR technology is robust in most weather conditions, including rainy environments, as was demonstrated in [3]. Despite these advantages, the main drawback of the LIDAR technology is the cost. However, their price has decreased in recent years, making it feasible to integrate them in Internet of Things (IoT) scenarios. Moreover, a further significant reduction is expected soon with novel solid-state LIDAR technology [2]. LIDARs' benefits make them a key

Manuscript received 13 June 2022; accepted 18 July 2022. Date of publication 27 July 2022; date of current version 1 September 2022. This work was supported in part by the InSecTT Project (www.insectt.eu) through the Electronic Component Systems for European Leadership Joint Undertaking (JU) under Grant 876038; and in part by JU through the European Union's Horizon 2020 Research and Innovation Program and Austria, Sweden, Spain, Italy, France, Portugal, Ireland, Finland, Slovenia, Poland, The Netherlands, and Turkey. The associate editor coordinating the review of this article and approving it for publication was Dr. Fabrizio Santi. *(Corresponding author: Cristian Wisultschew.)*

The authors are with the Centro de Electrónica Industrial, Universidad Politécnica de Madrid, 28006 Madrid, Spain (e-mail: cristian.wpuigdellivol@upm.es; a.pvicente@upm.es; andres.otero@upm.es; gabriel.mujica@upm.es; jorge.portilla@upm.es).

Digital Object Identifier 10.1109/JSEN.2022.3193060

technology in critical systems that need accurate artificial vision.

Among the systems where LIDARs are becoming indispensable are autonomous cars [4]. Autonomous systems are highly latency-sensitive, so sensor data processing can not be offloaded to the cloud. For these time-sensitive applications, edge computing is preferred over cloud computing, as processing times can be guaranteed by avoiding the round-trip time to the cloud, which also benefits power consumption. There are also advantages when computing at the network edge in terms of scalability since the network bandwidth is not increased when adding new devices. Besides, the amount of data flowing through the network is reduced, resulting in a reduction of issues related to privacy and security, and failures associated with loss of information during communications [5].

Deep Neural Networks (DNNs) are the algorithmic solution primarily adopted in state-of-the-art to process the enormous amount of data produced by LIDARs in real-time. DNNs are part of the Artificial Intelligence (AI) field, and they have achieved breakthroughs in a wide range of areas such as computer vision, speech recognition, and natural language processing, showing a high level of abstraction when working with complex data [6]. In the computer vision field, DNN algorithms outstand in applications such as object detection and classification, motion tracking, action recognition, human pose estimation, and semantic segmentation [7]. The downside of DNNs is their high computing performance demands.

The complexity of DNNs is boosted when applied to 3D LIDAR sensor data, much denser than 2D visible images. Therefore, it is highly challenging to achieve real-time requirements while maintaining low-power consumption when implemented at the edge. In this regard, the use of specific Deep Learning (DL) neural accelerators for the edge is opening new opportunities [8], [9].

As far as the authors know, there is no in-depth comparative study in the state-of-the-art focused on algorithms and edge processing architectures for object classification on 3D point clouds. In this work, representative DNN algorithms are studied and combined with state-of-the-art edge processing elements to provide a complete characterization, including performance, accuracy, and power consumption metrics, as is shown in the flowchart of the graphical abstract. The processing solutions considered focus on edge devices, including CPUs, GPU-based, SoC FPGA-based, and neural accelerators. To carry out this analysis, some of the models and edge devices have been adapted, resulting in the following original contributions:

- A novel approach for transforming a 3D voxel representation into concatenated 2D images to implement 3D DNN architectures into edge devices that only support 2D data.
- An adaptation of the original PointNet for devices that use quantized data, avoiding the operations not supported by the quantization operation.
- Development, manufacturing, and deployment of a custom IoT node including a low-power processor and a neural accelerator integrated into the same board.
- Design of Multi-View Depth Map projection Network (MVDMNet), a novel DNN to provide object classification results over point clouds. MVDMNet takes 2D images generated using a point cloud depth map projection from different views as input.

The rest of this paper is structured as follows. In Section II, related works which use different edge processing platforms to process DNN algorithms are described. State-of-the-art DNN architectures that use point clouds as input to provide object classification results are explained in Section III. These DNNs are the ones implemented and evaluated in this work. In section IV, the dataset used to train the DNN is presented along with the frameworks used to design, train, and make inferences with DNNs. A detailed description of the different platforms used for the implementation, along with the description of the point cloud transformations performed to adapt the data for each DNN architecture, is provided in Section V. The point clouds transformation to the format supported by each DNN is provided in Section VI. Experimental results are discussed in Section VII. Finally, conclusions and future lines of work are provided in Section VIII.

II. Related Work

This section presents the most relevant works focusing on DNN algorithms for object classification using point clouds at the edge. Since no works address these issues together, DNN edge implementations for object classification tasks using 2D images as input are studied first. Afterward, the state-of-the-art DNN algorithms that use point clouds as input to perform object classification are described.

A characterization of DNNs that perform object detection with 2D images by using edge processing platforms with different architectures such as CPUs, GPU-based, SoC FPGA-based and DL neural accelerators is presented in [10]. They analyze the impact of DNN design frameworks, their software stack, and their implemented optimizations on the final performance. Power consumption and temperature behavior of the different processing elements are measured. However, there is no information about the accuracy provided by the processing elements along with each DNN. Accuracy is a critical factor when dealing with these platforms because, in some architectures, the accuracy decreases significantly, as is explained in this work. Regarding DL neural accelerators, in [10] authors implement two of the most used nowadays. Differently, in this paper, five of the most used state-of-the-art DL neural accelerators for the edge are characterized together with CPU and GPU-based architectures. Besides, the DNNs implemented by the authors in [10] use 2D images as input, in contrast to the 3D point cloud data used in the DNNs implemented in the presented work, with the advantages that it entails. In [10] they only execute one of the nine evaluated DNNs on the SoC FPGA-based device in contrast with the proposed work in which all the architectures of DNNs are implemented on an FPGA.

Multiple DNN architectures exist that use point cloud data as an input to provide object detection and classification results [11]. Detailed descriptions of the most used DNN architectures which use point clouds are provided in [12]. There are few works in the state of the art that address the use of DNNs with 3D objects. As far as the authors know, these works always use one of the following 3D object representations: image-based, voxel-based, and point-based [13], [14]. No other architectures that provide object classification capabilities have been found by the authors in the current state of the art. However, there are also hybrid architectures that combine two of the three existing ones, such as VoxMVCNN [13], which combines image-based with voxel-based representations. However, in the presented work, the authors aim to present a comparative study between the base architectures to provide an analysis between their behaviors when running using different processing architectures such as CPUs, GPUs, FPGAs, and neural accelerators. They also gather the most commonly used point cloud datasets for 3D shape classification, 3D object detection and tracking, and 3D point cloud segmentation tasks. In turn, a comparative in terms of accuracy of the most relevant DNN works which use point cloud data is presented by the authors in [12]. However, all these implementations are performed using a desktop computer, so the provided inference times and power consumption metrics cannot be directly extrapolated to implementations on edge IoT devices.

In summary, there are works in state of the art focused on object classification algorithms in the edge, using 2D images [10], [15]. Some other works perform object classification using point clouds as input [12], [16]. However, no publications combine both approaches as proposed here. A range of edge-specific IoT low-power devices with different processing architectures such as CPUs, GPU-based, SoC FPGA-based, and DL neural accelerators are used for running DNNs in the presented work. Three DNN architectures that use point cloud as inputs were implemented to provide alternatives in terms of inference times and classification accuracy.

III. Deep Neural Network Architectures for Point Cloud Processing

In this section, the most relevant DNN architectures that use point clouds as inputs to perform object classification are explained. Emphasis is put on the inference stage, which is the part that can be performed at the edge.

A. PointNet

PointNet [17] is a DNN architecture proposed by Qi *et al.* in 2017 that takes as input raw point clouds without requiring any pre-processing transformation, which facilitates real-time processing. The data precision offered by PointNet is relatively high but at a higher computational cost. The PointNet architecture consists of two main modules, one for global point cloud classification and another dedicated to 3D local segmentation. Only the global classification module was put into practice to perform the comparison with the other implemented DNNs under similar conditions. PointNet has essential properties related to point cloud treatment, such as permutation and transformation invariance [18]. The resulting output obtained from the DNN is invariant to the possible spatial transformations on the input point cloud, such as rotation, translation, or denoising. This property allows the DNN to be useful for real classification situations where the point cloud representation has imperfections. All the results provided in this work for the PointNet have been achieved by using the same network structure hyper-parameters reported in [17].

B. Multi-View Depth Map Projection Network

Unlike PointNet, multi-view projection networks use concatenated one-channel 2D images as inputs. These images are generated by converting the original point cloud, for which two different methods have been proposed in state of the art: statistical projection and multi-view. The statistical projection method is based on the work presented in [19] for a road detection application using LIDAR sensors in autonomous vehicles. It relies on top-view 2D images that encode different statistic metrics related to a specific region of interest, such as the point density, mean reflectivity, and maximum elevation. Each of these metrics is encoded as one unique image channel. In this regard, images can be generated from the point cloud and used to feed the DNN. The original method only uses the top-view 2D images, which is a limitation. For this reason, authors in [20] proposed merging the information from multiple views trying to increase the obtained accuracy. In this regard, the one-channel 2D image of each view are concatenated before being provided as input to the DNN. However, experimental results show that adding the knowledge of different views may imply an increase or decrease in the DNN accuracy, depending on the information extracted by each view. A reduction in the accuracy of the network may result if the views are similar since DNN learn better the more varied the data is. However, if the views are significantly different, it will increase the accuracy. Authors in [20] show DNN accuracy results running object classification algorithms with a different number of views.

A complete characterization of different images generated using statistical projection along with multi-view methods is originally explored in this work. The detailed description of the applied transformations is presented in subsection VI-A.1. From the statistical projection method, the best results in terms of accuracy are obtained with the maximum elevation projection. Then, multiple views of this projection are generated as proposed by authors in [20]. A new architecture that combines these two transformations is originally proposed in this work. This DNN architecture is called MVDMNet. The hyper-parameters of the MVDMNet architecture are detailed in Table I. For each MVDMNet layer, information about the layer type, the input size, and the number of trainable parameters are presented in Table I. The selection of the number of views and their location is detailed in section VI-A.1.

C. VoxNet

The VoxNet architecture was presented in [21] by Maturana *et al.* aiming at classifying 3D objects with low resolution. This low resolution may affect the information

TABLE I
MVDMNet Architecture Hyper-Parameters

Layer type	Output shape	Parameters number
Input image	(64, 64, 3)	0
Convolution 2D	(64, 64, 32)	896
Activation	(64, 64, 32)	0
Convolution 2D	(64, 64, 32)	9248
Activation	(64, 64, 32)	0
Max pooling 2D	(32, 32, 32)	0
Dropout	(32, 32, 32)	0
Convolution 2D	(32, 32, 64)	18496
Activation	(32, 32, 64)	0
Convolution 2D	(32, 32, 64)	36928
Activation	(32, 32, 64)	0
Max pooling 2D	(16, 16, 64)	0
Dropout	(16, 16, 64)	0
Convolution 2D	(16, 16, 64)	36928
Activation	(16, 16, 64)	0
Convolution 2D	(16, 16, 64)	36928
Activation	(16, 16, 64)	0
Max pooling 2D	(8, 8, 64)	0
Dropout	(8, 8, 64)	0
Flatten	(4096)	0
Dense	(512)	2097664
Activation	(512)	0
Dropout	(512)	0
Dense	(40)	2097664
Activation	(40)	0

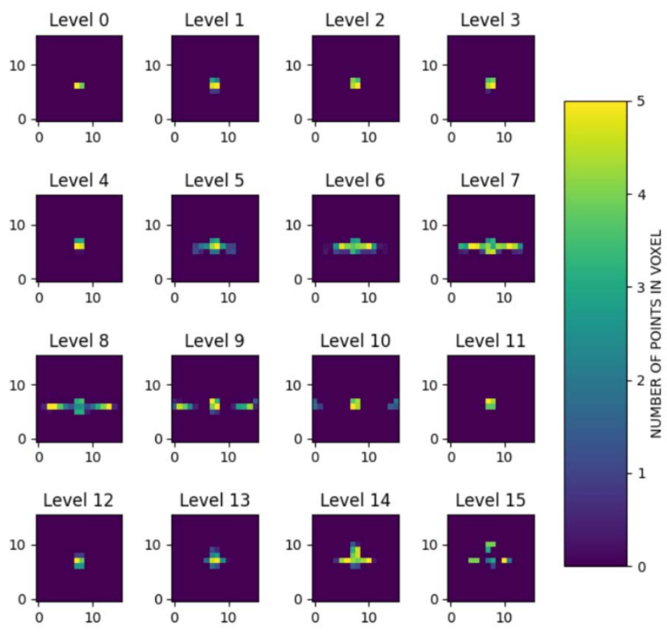


Fig. 1. Voxel transformation composed of 16 images (channels) of 16 × 16 pixels depth images of an airplane object.

that the DNN can extract from the point cloud. However, when using VoxNet, the accuracy is almost unaffected, as shown in the section VII-B. In VoxNet, a voxel grid type transformation is applied to the input data as a preprocessing step. The voxel grid representation [22] is a 3D matrix space composed of a fixed number of voxels. A voxel consists of a representation similar to a pixel, but instead of representing a two-dimensional plane, it represents a volume feature on a three-dimensional grid.

It is assumed that inference processing starts from the data collection in a point cloud format. Afterward, the input object is compressed as a 16 × 16 × 16 density voxel grid as explained in [21]. This process is described in section VI-A.2. In DNNs fed by 3D data, it is a common practice to apply 3D convolutions for extracting feature maps from a volumetric representation [21]. However, the problem arises from the impossibility of implementing these operations in some DL neural accelerators. To solve this issue, sectioning the voxel grid in as many sections as layers of voxels along an axis is originally proposed in this work, thus obtaining the depth levels of the voxelized object as a batch of images, which are admitted by these devices.

For the proposed architecture, starting from a 16 × 16 × 16 voxel grid, the neural network will interpret the input as a 16 × 16 matrix with 16 depth channels and perform a 2D convolution filter over each depth channel, extracting the features of each one as if it were a 2D image that instead of having three RGB depth channels has 16 associated channels. Fig. 1 shows an example of transformation from point cloud to a 16 × 16 × 16 voxel grid, showing the 16 depth levels along the z-axis, being in the XY plane where 2D convolutions apply. The resulting DNN has the advantage of computing a high number of volumetric data at a very low inference time while maintaining competitive precision numbers along with optimal power draw. One challenge facing volumetric classification is the feature loss resulting from the voxel grid transformation, preventing achieving superior precision compared to other DNN architectures. The VoxNet network structure implemented in this work follows the architecture described in [21].

TABLE II
Hyper-Parameters Used During the Training Stage

Hyper-parameter	Value
Optimizer	Adam
Activation function	ReLU
Learning rate	0.001
Loss function	Cross-entropy
Batch size	64
Epochs	100

The hyper-parameters used for training are shown in Table II. In this case, the same parameters have been applied to the three DNN models.

IV. Dataset and Framework Selection

The point cloud dataset selected for the comparative analysis performed in this work is presented in this section. Then, the deep learning frameworks used at the training and inference stages are shown.

A. Dataset

The object classification dataset selected for the analysis performed in this work is the Princeton ModelNet40 [23]. This is the most widely used database in state of the art targeting the evaluation of classification algorithms on point clouds. ModelNet40 consists of 12311 3D CAD objects from 40 different classes divided into 9843 items for training and 2468 items for validation. Each CAD object is converted to a point cloud of 2048 points. The MVDMNet and VoxNet architectures use 2048 points per object. However, in this work, the point clouds were reduced from 2048 to 1024 points for the PointNet architecture since this is the maximum size supported by Movidius NCS, one of the DL neural accelerators used in this work. A random procedure is proposed to select the points in each training epoch. This strategy provides greater diversity in the training dataset allowing the network to improve its learning and obtain better accuracy results. Each point cloud object is defined by its spatial coordinates on the X, Y, and Z axes. Luminosity data is not available in the ModelNet40 database, unlike the LIDAR sensors data.

B. Deep Learning Framework

DL frameworks are libraries that facilitate DL algorithms design, training, and validation through a high-level programming interface. They allow implementing neural models without getting into the details of the underlying algorithms. Therefore, it is essential to use a framework to obtain the fastest processing times and the shortest development time when targeting edge devices. Each edge device is compatible with a different set of frameworks. Since the framework may also affect the accuracy, it has been decided to use TensorFlow [24], which is compatible with all the devices used in this work. This way results only depend on the selected topology or the device, not the framework.

TensorFlow supports Python, C++, and Java programming languages. However, its Python API is much richer than the others. For this reason, in this work, both training and inference are performed using Python. Besides, as part of the TensorFlow ecosystem, a Lite version of the library is provided for mobile and IoT devices. This light version includes support for integer data types, not included in the full version.

V. Implementation in IoT Edge Devices

This section describes the platforms selected for the comparisons provided in this work. All of them are specifically designed for the edge as they have very low power consumption, including CPUs, GPUs, FPGAs, and edge-specific neural accelerators. Besides, a desktop PC has been included in the analysis for the sake of completeness. The section also describes how the DNN algorithms have been adapted to each platform in this work. The frameworks admitted in each case are specified along with the requirements for the implementation, as each processing device has specific particularities that must be taken into account.

A. Desktop PC

A desktop PC has been included in the comparison to offer a performance reference of a high-performance CPU, although it is clear that this solution is not compatible with the edge requirements. Personal computers are compatible with all the DL frameworks available at the moment, but TensorFlow has been chosen to provide a fair comparison with the rest of the solutions. The desktop PC used for this work is equipped with an Intel i7-8700 CPU with six cores operating at 3.2 GHz, 32 GB DDR4 RAM, and running an Ubuntu 18.04 operating system. GPU support has not been used in this implementation.

B. CPU-Based Devices for the Edge

CPUs are ubiquitous due to their flexibility. However, featured by a single or a low number of cores, CPU-based platforms are expected to be less suitable for executing DNN algorithms that are highly parallelizable. Nevertheless, they have been included in this work since CPU-based platforms are prevalent for edge applications. In particular, two different CPU-based families are tackled. First, a very low-power custom edge device working as an IoT node, then a Raspberry Pi 3 (RPi3) together with a Raspberry Pi 4 (RPi4) acting as Single-Board Computes (SBCs) solution. These devices provide a traditional edge solution that serves as a benchmark to compare with the other presented architectures.

1) Custom IoT Edge Node: The Cookie platform [25], [26] has been selected as a representative example of an IoT node. It was designed at *Centro de Electrónica Industrial* (CEI) of Universidad Politécnica de Madrid and it is composed of four modular layers compatible among them. Each layer accomplishes a specific purpose: processing, communication, power supply, and sensing/actuation. This structure allows including different exchangeable layers according to the specifications of each application. Only the processing and power supply layers have been used in this work. The processing layer was designed to perform edge computing with low power consumption and includes an Atmel SAMA5D3 processor with an external RAM of 256 MB and an SD card reader. The SAMA5D3 processor is a 32-bit medium-performance, low-power ARM Cortex-A5 core working at a clock speed of 536 MHz and consuming less than 150 mW. The approximate cost of this board for a 1000-unit production run is about 30 $.

An embedded Debian 9 Stretch Linux operating system installed in a flash SD card runs in the processing layer to ensure compatibility with the TensorFlow Lite library. TensorFlow Lite is the TensorFlow framework adaptation for IoT devices as it allows DNNs to be optimized for devices with reduced computational resources. The data type of the DNNs trained with TensorFlow framework is Floating-Point 32-bits (FP32), and it was trained on the desktop PC. Then, it was converted to Unsigned 8-bit INTeger (UINT8) data type which is the TensorFlow Lite admitted format, and then, the DNN was deployed into the Cookie node. However, the TensorFlow Lite library is not directly compatible with the Cookie CPU as these libraries require the microprocessor to have an integrated NEON vector instruction set [27] and the ARMv7 version of the SAMA5D3 processor does not incorporate this instruction set. To solve this, it was necessary to recompile the TensorFlow Lite libraries. The library was cross-compiled for ARMv7 architectures from the desktop PC, and the neon optimization flags were removed.

2) RPi3 and RPi4: Raspberry Pi is a family of SBCs developed by the Raspberry Pi Foundation. In this work, an RPi3 (RPi3B+ model) and the 4 GB RAM version of the RPi4 (RPi4 model) were selected for comparison. These devices handle single-precision floating-point formats (FP32), so DNNs were trained using this format. As for the rest of the implementation, the training stage has been carried out on a desktop PC. Both in training and inference, the TensorFlow framework was used.

C. GPU-Based Devices for the Edge

GPUs have traditionally been used to perform graphics rendering. However, their usage has been extended in the last years to general-purpose computation, exploiting the inherent parallelism offered by the vast amount of cores their include. Consequently, GPUs are widely used in practice to process DNNs [28]. Apart from inference, their benefits are also exploited for training, being the most popular option nowadays and leveraging their compatibility with the most popular DL frameworks.

For edge inference, specific low-power consumption GPU-based devices are available in the market, integrating a GPU and a CPU in the same system-on-chip. The best known are the Jetson family boards commercialized by NVIDIA. In this work, two models of the Jetson family were selected: the Jetson Xavier NX, a high-performance, high-cost platform, and the Jetson Nano, the version with the lowest cost, power consumption, and performance of the family. Both devices were used in the maximum performance mode in the presented work, considering that GPU-based architectures are expected to offer lower performance than neural accelerators. This way, the comparative shows how far the GPU-based architectures are from the neural accelerator architectures. Experimental results validate this assumption.

The data type selected for the edge GPU DNN trainings was FP32, and the TensorFlow framework was utilized. Then, the DNN model was converted to TensorRT with FP16 data type. TensorRT is a set of NVIDIA libraries optimized for adapting and running DNNs using GPUs. These libraries are also used to deploy the DNN in the GPU-based edge devices.

D. SoC FPGA-Based Device for the Edge

Field Programmable Gate Arrays (FPGAs) are well-known configurable digital circuits widely deployed in the industry. Their programmable logic resources can be customized to each application, allowing further optimization compared to GPUs and CPUs and higher power efficiency. The main FPGAs disadvantage is that the implementation of the designs is highly time-consuming. However, there are solutions for directly porting DNN models to an SoC FPGA-based circuit. Among them is the Apache TVM [29] used in this work.

The Xilinx PYNQ-Z1 is a 165 $ development board that comes with a Zynq-7000 All Programmable System on Chip (APSoC). It integrates an FPGA and a multi-core processor into a single chip. For this reason, this approach it is refer as SoC FPGA-based devices. The board works on the PYNQ platform by running an image based on the Ubuntu distribution for Linux.

In this work, a state-of-the-art neural network compiler [30] was used for running all three DNN architectures on the reconfigurable hardware. The compiler used in this work is Apache TVM, which receives as input the DNN model from a DL framework and transforms it into an Intermediate Representation (IR) that takes the form of a computational graph. This low-level optimized graph can target different hardware back-ends, including custom hardware accelerators on FPGAs such as the Versatile Tensor Accelerator (VTA) [31], a programmable DNN accelerator fully integrated into the TVM compiler stack. Its architecture consists of fetch, load, store, and compute modules that communicate via FIFO queues and SRAM blocks. Among the main advantages offered by VTA are its fully customizable GEneral Matrix Multiply (GEMM) tensor unit, task-level pipeline parallelism, and a just-in-time runtime that enables heterogeneous execution between the PL and PS subsystems.

The inference pipeline was performed in the provided implementation by first converting the DNN from the TensorFlow framework into the IR, followed by an 8-bit integer quantization for GEMM operations support. Afterward, the quantized compute graph undergoes a packing transformation process so that the GEMM core performs tensorization on matrix-multiplication operations such as 2D convolutions. Once the computing graph is packed, the next step is to generate customized executable libraries for optimizing DNN operation kernels for a specific hardware target. This DNN hardware optimization is made by using AutoTVM [32], an automated tuning optimizer that uses an ML-based cost model to search for the best possible configuration for achieving maximum performance on the selected hardware device. It is necessary to consider that some DNN operations used in TensorFlow are not supported by the TVM compiler. When this occurs, the conflicting operations are sent to the CPU to be processed. Thus, the DNN is split by executing one part in the FPGA and the conflictive part in the CPU, which causes a considerable increase in the processing time. Note that it is also possible to implement the DNN architectures manually in the PL without using the VTA compilation tools. Additionally, it is also possible to fully implement in the PL the preprocessing of the point cloud to the formats supported by VoxNet and MVDMNet, which will provide higher acceleration. However, these PL-based solutions have not been implemented, since the development time would increase considerably and it is out of the scope of this work.

E. Deep Learning Neural Accelerators for the Edge

Neural accelerators are hardware architectures specialized for deep learning applications. In particular, for performing matrix multiplications, the ubiquitous operation in DNN algorithms. From this perspective, it can be said that GPUs, or FPGAs, are more flexible as they can be adapted to different types of algorithms. However, the high specialization of the neural accelerators makes them potentially much more energy-efficient, a critical parameter in embedded systems. These processors can be found under different names such as Tensor Processing Unit (TPU) [33], Neural Processing Unit (NPU) [34], or Vision Processing Unit (VPU) [35].

1) Coral EdgeTPU-Based Devices: EdgeTPU is a commercial neural accelerator designed by Google to execute DNN inferences with a performance of up to 4 TOPS and 2 W of power consumption. In this work, three different EdgeTPU-based devices have been used. The first one is the Cookie Coral, which is a custom board for the Cookie IoT

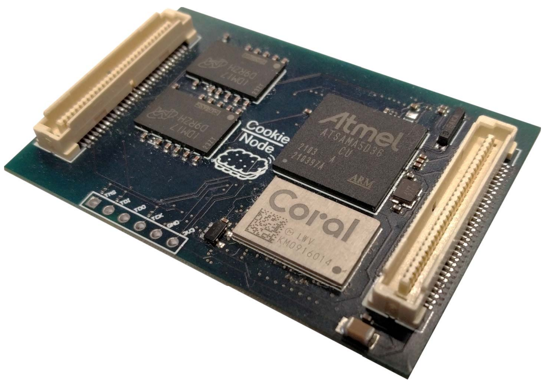


Fig. 2. Cookie IoT node with an EdgeTPU accelerator chip integrated.

platform (explained in Section V-B.1), in which an EdgeTPU chip has been integrated into the same board, as it is shown in Fig. 2. The approximate cost of this board for a 1000-unit production run is about 50 $. It must be highlighted that this new node has been specially created for this work. In this regard, it is possible to take advantage of the reduced power consumption provided by the Cookie IoT node combined with the high performance offered by the Coral EdgeTPU accelerator. The second EdgeTPU-based device is the Coral Dev Board Mini, an SBC with an EdgeTPU chip integrated into the board, wireless connectivity, and running a variation of Debian Linux. Finally, the third EdgeTPU-based device is the Coral USB accelerator, a USB device that provides an EdgeTPU as a co-processor for any device that can handle Debian Linux, macOS, and Windows 10.

The EdgeTPU architecture handles UINT8. For this reason, it is necessary to quantize the DNN weights. There are two different approaches for quantization: post-training quantization and quantization aware training. Post-training quantization trains the DNN using float data type, and after the training stage, the quantization is performed. This is the most straightforward alternative. However, there is almost always a loss of accuracy that, in some cases, can be critical. Differently, quantization-aware training quantizes all the DNN operations before the training stage. This alternative is the most complex one since not all the operations inside the DNN are supported for 8-bit integers in the DL frameworks. However, with this solution, the accuracy decreases slightly or, in some cases, is maintained as shown in section VII-B.

Quantization-aware training has been performed in this work using TensorFlow Lite. Nevertheless, it must be considered that the PointNet architecture is not compatible with this technique due to some operations not supported by the quantization, such as batch normalization, one-dimensional convolutions, and one-dimensional max pool. For this reason, an adaptation of the original PointNet is proposed in this work for devices that use quantized data. This custom PointNet uses two-dimensional convolutions and max pool stages. The convolutional step is the one that requires the most computational resources to be processed, and adding one dimension implies an increase in size. In this regard, the processing time is considerably affected, as shown in subsection VII-C. Custom PointNet also removes the batch normalization layers, resulting in slightly lower accuracy as illustrated in subsection X.

2) Movidius Neural Compute Stick 1 and 2: Intel Movidius Neural Compute Stick (NCS) is designed to run DNN at the edge with high performance. Two versions are available: the Movidius NCS1, powered by the Intel Movidius Myriad 2 VPU, and the NCS2, featured by a Myriad X VPU. These processors are used to accelerate DNNs by running parts of the DNNs in parallel. They act as a co-processor and must be connected to a host machine using the USB 3.0 interface. The NCS1 is now discontinued, and NCS2 costs 114 $.

The Movidius NCS architecture handles FP16 data types. In this regard, it is necessary to convert the weights of the DNN since they are trained in FP32. The conversion is achieved starting with the TensorFlow trained files using a desktop PC and performing post-training quantization with OpenVINO toolkit libraries [36] which are specific for Intel devices. These libraries convert the data of the DNN to FP16 data type and adapt the DNN operations to the NCS architecture. The precision is almost not affected during the conversion, as shown in section VII-B.

In the ModelNet40 database, each object is composed of 2048 points. NCS2 has a memory of 320 MB to store the entire DNN. The model trained with objects of 2048 points exceeds this size, so it was necessary to reduce the number of points of each object to 1024. There are two ways to select the points: the first is random, and the second is to take specific fixed points. The first approach was implemented since 1024 points were randomly selected for each object at each training epoch. This strategy allows a greater diversity in the training dataset, which provides better results in terms of accuracy.

3) Rockchip AI Stick: The Rockchip AI Stick is a USB device that integrates Rockchip's RK1808 chip as a co-processor for any host that can handle Windows, Linux, or macOS. The chip has a Neural Processing Unit (NPU) integrated. The stick connects to a host machine via USB3.0. It requires a host processor with an x86 architecture.

The Rockchip AI stick NPU architecture handles UINT8 as EdgeTPU architectures. For this reason, the TensorFlow Lite framework is used to quantize the DNN algorithm weights. Then, the RocKchip Neural Net (RKNN) libraries are used to adapt the TensorFlow Lite quantized weights to a format compatible with the AI stick architecture. RKNN is a DNN tool developed for the use of NPU on embedded platforms that allow the deployment of DNNs.

The most relevant specifications to compare each of the devices detailed in this section are summarized in Table III. These metrics are the ones provided by the manufacturers. Regarding performance metrics, two measures are provided depending on the data type used by each architecture, Tera FLoating Points Operations per Second (TFLOPS) for floating point-based architectures and TOPS for integer-based architectures. The power consumption metrics are provided for maximum performance modes. In Table III the cookie specifications are not included since it is a custom board. The TFLOPS parameter provided by the Pynq is hardware-dependent, so it is not included either.

TABLE III
DEVICES SPECIFICATIONS PROVIDED BY EACH MANUFACTURER

Device	Price	TFLOPS/ TOPS	Power consumption	Need host?
Desktop PC	1200 $	0.423 TFLOPS	65 W	no
Cookie	30 $	–	–	no
RPi3	35 $	0.003 TFLOPS	3.7 W	no
RPi4	55 $	0.013 TFLOPS	6 W	no
Jetson Nano	100 $	2 TOPS	10 W	no
Jetson NX	400 $	21 TOPS	15 W	no
Pynq	165 $	–	2.6 W	no
Movidius NCS1	96 $	1 TOPS	1.5 W	yes
Movidius NCS2	114 $	4 TOPS	1.5 W	yes
Cookie Coral	50 $	4 TOPS	–	no
Coral DevBoard	100 $	4 TOPS	2 W	no
Coral USB	60 $	4 TOPS	2 W	yes
Rockchip AI stick	86 $	3 TOPS	1.5 W	yes

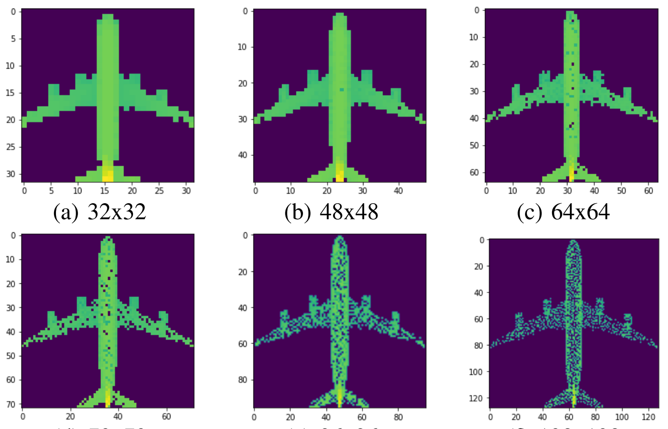

(a) 32x32 (b) 48x48 (c) 64x64 (d) 72x72 (e) 96x96 (f) 128x128

Fig. 3. Different images sizes of depth map projection.

TABLE IV
ACCURACY FOR DIFFERNET IMAGE SIZES

Image size	Accuracy
32x32	81.72%
48x48	84.40%
64x64	**85.90%**
72x72	83.87%
96x96	84.23%
128x128	83.79%

TABLE V
ACCURACY FOR DIFFERNET STATISTICAL PROJECTION IMAGES

Statistical projection	Accuracy
Depth max	**85.90%**
Depth min	83.55%
Depth mean	84.48%
Points density	85.17%
Mean STD	83.95%

VI. POINT CLOUD PREPROCESSING

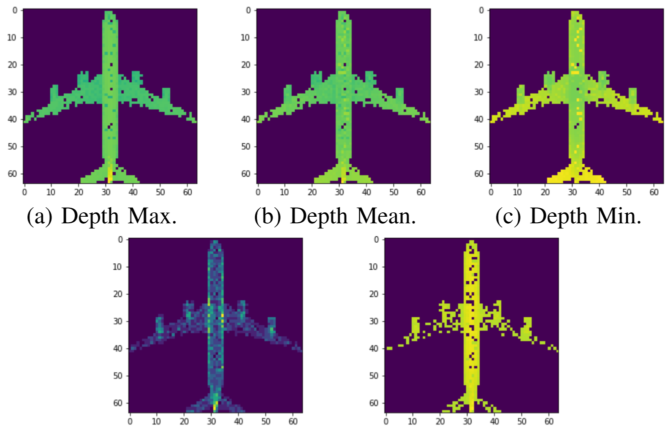
A. Point Cloud Transformation

The explanation of how to convert the point cloud to one-channel 2D depth map images and voxel formats that are used as inputs to the MVDMNet and VoxNet architectures is detailed in this section.

1) Point Cloud to One-Channel 2D Depth Map Images: As mentioned before, the database used in this work is in the form of a point cloud to simulate what would be produced with a LIDAR sensor. However, MVDMNet uses 2D images as input. The point cloud must be converted using statistical projection and multi-view methods to obtain these images. In this regard, one-channel 2D depth map images from different views are used as the input for the MVDMNet, as was explained in section III-B. First, it is necessary to select the image size that provides the best results in terms of accuracy. Different images sizes generated using the maximum elevation projection with a top-view of the 3D point cloud were evaluated and presented in Table IV. These generated images of an airplane object class are shown in Fig. 3. As it is a depth map, the pixels colored in yellow represent the closest points, and those colored in green represent the farthest points. The image size of 64×64 provides the best results in terms of accuracy. For this reason, this is the size used in the rest of the experiments presented in this section.

Authors in [19] proposed six statistical projection methods to convert the 3D point cloud into 2D images. These projections are used to calculate the value of each pixel of the 2D image. The six statistics computed for generating each projection are the following: points density, mean reflectivity, mean STandard Deviation (STD), minimum, maximum, and mean depth. The statistical projection that provides the best results in terms of accuracy was selected for MVDMNet. As the ModelNet40 dataset does not provide reflectivity value, mean reflectivity statistical projection can not be implemented. The implementation presented in this section was performed with a top-view of the 3D point cloud as proposed by authors in [19]. Accuracy results when using each statistical projection on the ModelNet40 dataset are presented in Table V. The generated images for each statistical projection of an airplane object class are shown in Fig. 4. Note that the pixels colored in yellow represent the low-value points, and those colored in green represent high-value points. Maximum depth statistical projection provides the best results in terms of accuracy. For this reason, this is the projection method used in the rest of the experiments presented in this section.

With the image size and the projection that provides the best results in terms of accuracy selected, an exploration was carried out to define the optimal views of the 3D objects before the projecting stage. These views were evaluated by setting up viewpoints (as virtual cameras that show a specific view of the 3D object) which are necessary to create a multi-view shape representation. Each viewpoint generates a one-channel image. It is assumed that the objects are upright oriented along the vertical axis. The selection of the views which maximize the accuracy was carried out searching among 96 options. The selection procedure is as follows. First, each of the 96 images was used to train the DNN, providing different accuracy results. The 10 views with the best results in terms of accuracy are presented in Table VI. Then, these views were concatenated and used to train a



(a) Depth Max. (b) Depth Mean. (c) Depth Min. (d) Points density. (e) Mean STD.

Fig. 4. Statistical projection images evaluated.

TABLE VI
ACCURACY FOR DIFFERNET VIEWS

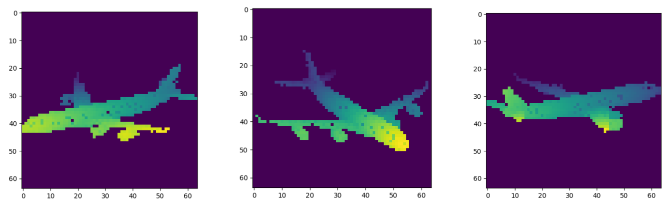
Rotation around the vertical axis	Elevation from the ground plane	Accuracy
230º	20º	**87.64%**
150º	20º	87.60%
60º	20º	87.56%
150º	10º	87.46%
140º	20º	87.23%
120º	10º	87.15%
220º	20º	87.15%
240º	20º	87.15%
90º	10º	87.15%
310º	10º	87.07%

TABLE VII
ACCURACY FOR DIFFERNET CONCATENATED VIEWS

Number of concatenated images	Accuracy
1	87.64%
2	87.71%
3	**88.53%**
4	87.64%
5	86.62%
6	87.35%
7	87.07%
8	87.03%
9	86.87%
10	85.86%

DNN. It is necessary to consider that the higher the number of concatenated images, the longer the inference time of the DNN to process this information. Experiments to select the number of images that maximize the accuracy were carried out using up to 10 concatenated images of different views as it is presented in Table VI. The option with the highest accuracy was obtained with three concatenated images. Using more than three images provides lower accuracy results due to the overfitting problem [37].

The angles selected for the three views correspond to the best views in terms of accuracy are 230°, 150°, and 60° rotating around the vertical axis and 20° elevated from the ground plane when pointing towards the centroid of the object.



(a) 60º rotation. (b) 150º rotation. (c) 230º rotation.

Fig. 5. Final input image of an airplane object class composed by three one-dimensional depth map projection images with the three best views showing the rotation of each view.

These values are shown in Table VI. An example of these three views for an object in the airplane class of the ModelNet40 database is shown in Fig. 5. Also, this figure represents an example of the final input image, which is composed of three concatenated one-channel 2D depth map images of the three best views. For this reason, the input image size is (64, 64, 3). The experiments for the presented exploration to select the best image sizes, statistical projections, and views were carried out by training and validating the results using the ModelNet40 dataset. Only this three views option is considered for the experimental results in the next section since it has been proved to provide the best results in terms of accuracy.

2) Point Cloud to Voxels: Before using VoxNet, the point cloud must be transformed into a voxel-based grid representation, as it is explained in [21]. This transformation reduces the point cloud resolution without significantly altering the global features.

Voxel-based surfaces can be formed by varying-size voxels, which requires more processing power or equal-size voxels. Concerning equal-size voxels, the voxel grid will be a 3D matrix with dimensions [X, Y, Z] where X, Y, and Z are the number of voxels dividing the length of each coordinate axis. Authors in [21] convert the point cloud into a binary voxel grid, in which the voxels take the value depending on whether or not a point of the cloud occupies the voxel. However, the frameworks used in this work do not support DNN architectures with binary data. Thus, in the VoxNet implemented in this work, each voxel will have a specific value, consisting of the number of inside points. Each voxel value is calculated with a density or concentration ratio of points.

This transformation was used to convert the point cloud objects before being processed by VoxNet. The point cloud was compressed as a specific voxel grid size, where each voxel was measured by a point cloud density ratio so that the resulting representation was independent of the number of points fed to the DNN. Different voxel grid dimensions were considered and evaluated as shown in Table VIII. On the one side, a high precision loss was obtained using the size of [4 × 4 × 4]. On the other side, the [32 × 32 × 32] size produced a considerable increase in processing time without providing an increment in the accuracy. Note that these inference times are calculated using the desktop PC described in Section V-A. Therefore, the [16 × 16 × 16] size option was the most efficient in terms of inference time and accuracy since the accuracy obtained was the highest with a

TABLE VIII
ACCURACY AND INFERENCE TIMES FOR DIFFERNET VOXEL GRID SIZES FOR MODELNET40 DATABASE

Voxel grid size	Inference time	Accuracy
4x4x4	4.39 ms	72.93%
8x8x8	5.34 ms	82.41%
12x12x12	6.60 ms	86.58%
16x16x16	8.16 ms	**87.84%**
24x24x24	11.58 ms	86.95%
32x32x32	15.04 ms	87.11%

TABLE IX
DNN MODEL SIZES AND TIME TO TRAIN ONE EPOCH

DNN model	data type	Size	Time to train one epoch
PointNet	FP32	40.2 MB	420 s
Custom PointNet	UINT8	50.9 MB	151 s
MVDMNet	FP32	27.2 MB	109 s
MVDMNet	UINT8	2.3 MB	277 s
VoxNet	FP32	7.4 MB	3 s
VoxNet	UINT8	0.685 MB	3 s

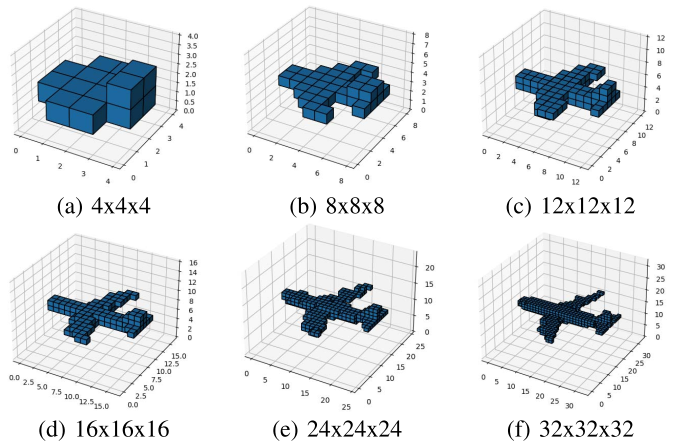


Fig. 6. Voxel representation with different voxel grid sizes.

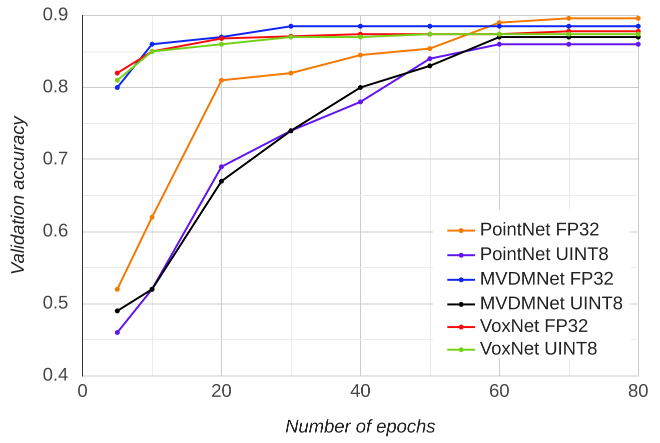


Fig. 7. Validation accuracy obtained on each epoch during the training stage.

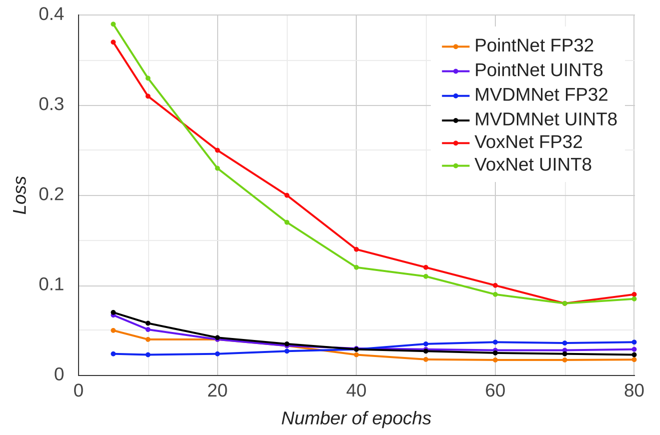


Fig. 8. Loss obtained on each epoch during the training stage.

considerable reduction in the processing time regarding the [32 × 32 × 32] voxel grid size alternative. Fig. 6 shows the resulting transformation of an airplane class point cloud object from ModelNet40 to the voxel grid sizes compared in Table VIII.

VII. RESULTS

The experimental results provided by the implementations of the three DNN architectures running on the different edge technologies are presented in this section. They include results related with the training stage. Also, information about the accuracy of the validation dataset and the inference times is presented. Besides, the average power consumption of the different platforms executing each of the DNNs is shown. Finally, the preprocessing times of converting the data from the point cloud format to the formats supported by VoxNet and MVDMNet are provided.

A. Training Stage

This section shows the results obtained during the training stage of each architecture when performing the training using the desktop PC. Table IX shows information about the size of each DNN model implemented. Note that each DNN provides different results depending on the data type used during the training stage. The size of the model is an important parameter for evaluating the computational effort needed to process the DNN. This factor is essential in edge devices as they have computational constraints. Table IX also provides information about the time needed to train each epoch. Note that this parameter is related to the DNN size.

The evolution of the validation accuracy and the loss parameters obtained on each epoch during the training stage is presented in Fig. 7 and Fig. 8.

B. Validation Accuracy

Table X shows the validation accuracy results obtained during the training stage provided by different devices when using ModelNet40 database. The training and validation data were split randomly, representing 75% and 25% of the total dataset, respectively. The splitting was performed randomly only the first time, and then it was conserved for the rest of the experiments. The comparison of the different devices is fairer this way since there are slight variations in the accuracy

TABLE X
VALIDATION ACCURACY

	DNN architecture		
Processing platform	**PointNet**	**MVDMNet**	**VoxNet**
Desktop PC	88.98%	88.53%	87.84%
Cookie	–	87.88%	87.72%
RPi3	88.98%	88.53%	87.84%
RPi4	88.98%	88.53%	87.84%
Jetson Nano	88.98 %	88.53%	87.84%
Jetson NX	88.98%	88.53%	87.84%
Pynq	79.52%	86.76%	83.57%
Movidius NCS1	88.21%	88.53%	87.84%
Movidius NCS2	88.21%	88.53%	87.84%
Cookie Coral	–	85.74%	86.71%
Coral DevBoard	85.82%*	87.93%	87.48%
Coral USB	85.82%*	87.93%	87.48%
Rockchip AI stick	37.12%*	84.85%	81.81%

when splitting the data randomly for each experiment. These variations may lead to incorrect conclusions in devices with similar accuracy.

As mentioned in section V-E.1, the DNNs that rely on quantized weights use the custom PointNet adaptation proposed in this work, which is marked with a * symbol in Table X, to highlight that it is different from the original version. Note that the validation accuracy results of this custom PointNet are slightly lower than the original version.

It is necessary to consider some cases that could not be implemented, such as the processing of PointNet and PointNet custom over Cookie and Cookie Coral platforms due to a memory error provided by the vast size of the DNN. PointNet architectures requires more RAM than the 256 MB available in these devices. Besides, the validation accuracy of the Rockchip AI stick in PointNet custom architecture obtained an inadmissible value of 37.12%. This loss of precision is produced in the conversion stage of the DNN TensorFlow Lite model to a Rockchip AI stick compatible model using RKNN API. This is caused by existing limitations in the API.

C. Inference Times

This section reports the inference times for each architecture without including the preprocessing time. The inference time was measured staring in the precise moment when the data of the object in each of the 3 formats analyzed in this work is provided to the DNN as input until the moment when the output of the DNN is obtained. This output contains the information of the last layer of the DNN, in which the information regarding the prediction of the input objects class is contained. Table XI shows the average inference time when running 1000 inferences using random objects. The average value is calculated as there are slight differences in the measurements when processing each object individually due to the the experiments are running on a Linux OS which performs other processes at the same time.

The Cookie and Cookie Coral inference times using PointNet were not obtained due to the issue mentioned in subsection VII-B. Regarding the PYNQ device, some PointNet operations are not supported by the GEMM core.

TABLE XI
INFERENCE TIMES

	DNN architecture		
Processing platform	**PointNet**	**MVDMNet**	**VoxNet**
Desktop PC	10.45 ms 15.75 ms*	2.71 ms	1.48 ms
Cookie	–	98.91 s	6.77 s
RPi3	446.21 ms	76.96 ms	23.33 ms
RPi4	236.65 ms	35.62 ms	9.51 ms
Jetson Nano	67.12 ms	7.71 ms	5.92 ms
Jetson NX	28.63 ms	2.56 ms	2.12 ms
Pynq	258.78 ms	16.67 ms	3.28 ms
Movidius NCS1	192.12 ms	9.68 ms	3.59 ms
Movidius NCS2	59.40 ms	3.04 ms	1.62 ms
Cookie Coral	–	223.33 ms	9.11 ms
Coral DevBoard	219.86 ms*	4.61 ms	3.12 ms
Coral USB	23.30 ms*	0.70 ms	**0.21 ms**
Rockchip AI stick	14.04 ms*	3.74 ms	3.35 ms

TABLE XII
INFERENCE THROUGHPUT IN FPS

	DNN architecture		
Processing platform	**PointNet**	**MVDMNet**	**VoxNet**
Desktop PC	95.69 FPS 63.49 FPS*	369.0 FPS	675.67 FPS
Cookie	–	0.01 FPS	0.15 FPS
RPi3	2.24 FPS	12.99 FPS	42.86 FPS
RPi4	4.23 FPS	28.07 FPS	105.15 FPS
Jetson Nano	14.90 FPS	129.70 FPS	168.92 FPS
Jetson NX	34.92 FPS	393.70 FPS	471.70 FPS
Pynq	3.86 FPS	59.99 FPS	304.88 FPS
Movidius NCS1	5.21 FPS	103.31 FPS	278.55 FPS
Movidius NCS2	16.83 FPS	328.95 FPS	617.28 FPS
Cookie Coral	–	4.48 FPS	109.80 FPS
Coral DevBoard	4.55 FPS*	216.92 FPS	320.51 FPS
Coral USB	42.91 FPS*	1428.57 FPS	**4761.90 FPS**
Rockchip AI stick	71.25 FPS*	267.38 FPS	298.51 FPS

As a result, these conflictive operations were executed in the embedded ARM processor, causing a significant increase in the processing time.

It is necessary to take into account that the inference time between PointNet and custom PointNet is significantly different, as explained in subsection V-E.1. For reference, custom PointNet average inference time running on the Desktop PC is 15.75 ms compared to 10.75 ms for the original PointNet. Both networks were tested using FP32 data type. As it is shown in tables XI and XII, custom PointNet inference times are marked with a * symbol. Inference time results are provided using ms and Frames Per Second (FPS) metrics in Table XI and Table XII respectively.

D. Preprocessing Time

This section shows the results of the preprocessing time required to transform the point cloud into voxel and image formats. The preprocessing has been implemented using sequential software in the host processors of each platform. No specific hardware optimizations have been addressed at

TABLE XIII
PREPROCESING TIMES

	Data transformation	
Processing platform	**MVDMNet**	**VoxNet**
Desktop PC	88.12 ms	0.393 ms
Cookie	5.287 s	13.28 ms
RPi3	1.603 s	3.02 ms
RPi4	856.91 ms	1.59 ms
Jetson Nano	661.771 ms	1.64 ms
Jetson NX	434.47 ms	**1.36 ms**
Cookie Coral	9.59 s	20.04 ms
Coral DevBoard	1.464 s	3.81 ms

TABLE XIV
PREPROCESSING TIMES COMBINED WITH INFERENCE TIMES

	Data transformation		
Processing platform	**PointNet**	**MVDMNet**	**VoxNet**
Desktop PC	10.45 15.75*	90.83 ms	1.87 ms
Cookie	-	104.20 s	20.05 s
RPi3	446.21 ms	1.68 s	26.35 ms
RPi4	236.65 ms	892.53 ms	11.10 ms
Jetson Nano	67.12 ms	669.48 ms	7.56 ms
Jetson NX	28.63 ms	437.03 ms	**3.48 ms**
Cookie Coral	-	9.81 s	29.15 ms
Coral DevBoard	219.86 ms*	1.47 s	6.93 ms

this point. This could be a time-consuming step to be carried out manually, which would imply, for instance, custom CUDA developments for GPUs or VHDL descriptions for FPGAs. This is out of the scope of the paper. For this reason, this work has focused on obtaining the CPU preprocessing times to compare each architecture. In the case of Jetson NX and Nano, the preprocessing time is calculated using only the CPU. These times are presented in Table XIII. Besides, the preprocessing times combined with the inference times, which constitute the real total time to process an object, are shown in Table XIV. Note that neural accelerators can not perform preprocessing since they only operate as co-processors dedicated exclusively for running DNN.

E. Power Consumption

This section presents power consumption data for the different processing devices. As all the devices analyzed in this work are connected via USB (except the desktop PC and the Jetson NX), a USB power consumption meter has been used for the experiments, which performs both real-time and average power consumption measurements. For the desktop PC and the Jetson NX, the power consumption reported by their manufacturers is provided. Results correspond to the average power consumption when running 10000 inferences of random objects are presented in tables XV, XVI, and XVII. Information about the power consumption of each architecture on Idle mode is also presented. The processing of PointNet over the Cookie and Cookie Coral platforms could not be carried out as mentioned in Subsection VII-B.

TABLE XV
POWER CONSUMPTION

		DNN architecture		
Processing platform	**Idle**	**PointNet**	**MVDMNet**	**VoxNet**
Desktop PC	35 W	65 W	65 W	65 W
Cookie	0.42 W	–	**0.53 W**	**0.53 W**
RPi3	2.45 W	4.62 W	3.39 W	4.31 W
RPi4	2.67 W	4.82 W	4.91 W	3.76 W
Jetson Nano	3.14 W	6.74 W	7.75 W	4.65 W
Jetson NX	7.3 W	15 W	15 W	15 W
Pynq	1.90 W	2.32 W	2.19 W	2.06 W
Movidius NCS1	0.35 W	1.19 W	1.58 W	1.26 W
Movidius NCS2	0.59 W	1.46 W	1.39 W	1.27 W
Cookie Coral	1.23 W	–	2.28 W	2.18 W
Coral DevBoard	0.79 W	1.72 W	1.25 W	1.20 W
Coral USB	0.36 W	0.99 W	1.41 W	1.04 W
Rockchip AI stick	0.69 W	1.34 W	1.07 W	0.99 W

TABLE XVI
NUMBER OF FPS CONSUMING 1 W

	DNN architecture		
Processing platform	**PointNet**	**MVDMNet**	**VoxNet**
Desktop PC	1.47 FPS/W	5.68 FPS/W	10.39 FPS/W
Cookie	–	0.02 FPS/W	0.03 FPS/W
RPi3	0.48 FPS/W	3.83 FPS/W	9.94 FPS/W
RPi4	0.88 FPS/W	5.72 FPS/W	27.97 FPS/W
Jetson Nano	2.21 FPS/W	16.74 FPS/W	36.33 FPS/W
Jetson NX	2.32 FPS/W	26.25 FPS/W	31.45 FPS/W
Pynq	1.66 FPS/W	27.39 FPS/W	148.0 FPS/W
Movidius NCS1	4.38 FPS/W	65.39 FPS/W	221.07 FPS/W
Movidius NCS2	11.53 FPS/W	236.65 FPS/W	486.05 FPS/W
Cookie Coral	–	1.97 FPS/W	50.37 FPS/W
Coral DevBoard	2.65 FPS/W	173.54 FPS/W	267.10 FPS/W
Coral USB	43.34 FPS/W	1013.17 FPS/W	**4578.75 FPS/W**
Rockchip AI stick	53.17 FPS/W	249.89 FPS/W	301.52 FPS/W

By relating the classification throughput in FPS with the power consumption, the FPS/W metric is obtained. These results are shown in Table XVI. This metric better represents the energetic cost when performing inference using different architectures, resulting in a critical parameter when dealing with edge systems.

Table XVII presents the Energy spent per inference, parameter that may become a interesting metric of comparison for certain tasks. As it is explained in [38], this metric measures the Energy needed to perform one inference. The Energy per inference parameter is calculated multiplying the average total power per inference, parameter presented in Table XV, by the inference time, which is provided in Table XI.

F. Analysis of the Results

Focusing on the classification accuracy, it mainly depends on the data type handled by the implemented DNN. In this regard, the use of quantized data may produce slight accuracy reductions, as shown in Table X. Moreover, some DL frameworks such as TensorFlow are optimized for using float data types during training. Therefore, quantization

TABLE XVII
ENERGY PER INFERENCE

Processing platform	DNN architecture		
	PointNet	MVDMNet	VoxNet
Desktop PC	679.25 mJ	176.15 mJ	96.20 mJ
Cookie	–	52.42 J	52.42 J
RPi3	2060.52 mJ	261.03 mJ	99.13 mJ
RPi4	1140.50 mJ	174.89 mJ	35.72 mJ
Jetson Nano	452.38 mJ	59.75 mJ	27.52 mJ
Jetson NX	429.45 mJ	38.4 mJ	31.80 mJ
Pynq	445.72 mJ	21.20 mJ	6.76 mJ
Movidius NCS1	228.62 mJ	15.29 mJ	4.52 mJ
Movidius NCS2	86.72 mJ	4.22 mJ	2.06 mJ
Cookie Coral	–	509.19 mJ	19.86 mJ
Coral DevBoard	378.15 mJ	5.76 mJ	3.74 mJ
Coral USB	23.07 mJ	0.99 mJ	**0.22 mJ**
Rockchip AI stick	18.81 mJ	4.00 mJ	3.32 mJ

complicates DNN designs, making them unfeasible in some cases, as happens with the PointNet architecture.

The accuracy is also reduced when the device requires an adaptation step from the trained models to a device-specific format, as happens with the neural accelerators and SoC FPGA-based devices. This adaptation is carried out by applying a device-specific API. In these cases, it must be considered if the API supports various data types and is well documented. In addition, APIs must be up-to-date as the DNN landscape is in constant evolution. RKNN is the Rockchip AI stick API, and does not release updates, is under-documented, and produces an accuracy loss on certain types of architectures such as PointNet, as shown in table X. OpenVINO, the API provided by Intel, is one of the best regarding these issues, is constantly updated, and has rich documentation. The EdgeTPU-based device implementation has the advantage of working directly with TensorFlow Lite, so it does not require an API for the adaptation. In turn, Jetson devices benefit from running DNNs in their native format. They also rely on the NVIDIA TensorRT API to increase performance. This API is constantly updated and offers a low development time.

In terms of inference times, it is shown in Tables XI and XII that the neural accelerators architectures provide the best results. In particular, the highest performance is provided by Coral EdgeTPU USB, being more than four times faster than the desktop PC running VoxNet and MVDMNet. The cause is that Coral is working with an 8-bit integer data type. Movidius NCS2 also stands out in inference time using an FP16 data type.

When comparing the three platforms that integrate Coral EdgeTPU, it can be seen in Tables XI and XII that the performance obtained by EdgeTPU USB device is much higher compared with Cookie Coral or Coral DevBoard. The reason is that some computations, such as data adaptation to EdgeTPU format, must be performed in the CPU. Thus, apart from the deep learning throughput, the computational capacity of the integrated CPUs is highly relevant for the overall performance of the application. In the case of Coral USB, a desktop PC with a powerful CPU is used as a host, surpassing significantly the capacity of the CPUs that can be found in the Cookie Coral or the Coral DevBoard. Moreover, it can also be seen that the performance provided by the Coral DevBoard is higher than the one obtained by the Cookie Coral. This is also related to the difference in computing resources of both CPUs. The Cookie Coral has an ultra-low power CPU, resulting in a limited computational capacity.

Regarding the Pynq device, DNN operations optimized for the PL reached an $\times 40$ improvement compared to the ones running on the ARM processor. However, because of the complexity of the PointNet architecture, some operations are not supported by the TVM compiler, which implies that they cannot be implemented in the FPGA. For this reason, the only manner to implement PointNet into the Pynq device when using the TVM compiler is to process these conflicting operations by using the ARM processor, leveraging the heterogeneous features offered by the JIT compiler. However, the inference performance was significantly affected because of the bottleneck generated by the workloads executed on the CPU.

Regarding the preprocessing time, PointNet benefits from using point clouds directly, not requiring preprocessing. However, it has the drawback of providing the highest inference times along with the largest model sizes. On the other hand, MVDMNet architecture requires extremely high preprocessing times, making the inference and preprocessing times the most elevated. However, the VoxNet architecture provides low preprocessing times and, together with the reduced inference times, makes it the fastest alternative.

In terms of power consumption, as shown in Table XV, neural accelerators offer a low power consumption, and this makes these devices suitable for IoT edge applications. From Table XVI it can also be seen that the devices with the best results are those based on neural accelerators since their architectures are optimized to minimize the power consumption. However, most of the solutions with neural accelerators integrated are co-processors and need a host to work. The extra power consumption required by the host, and the increase of the cost, must be considered in the overall system budget. The Cookie Coral platform and the Coral DevBoard devices integrate a CPU with a neural accelerator on the same board, and therefore they do not require a host. This makes them the most suitable solutions for IoT edge scenarios.

VIII. Conclusion and Future Lines of Work

In this work, three different DNN architectures implemented on edge devices are characterized when processing point clouds to perform object classification tasks. Results in terms of accuracy, inference time, and power consumption are presented. In this regard, it facilitates developers to select the alternative that best suits the required specifications.

It can be concluded that among all the edge processing devices studied in this work, the ones that provide the best results in terms of performance and power consumption when performing object classification tasks with point clouds as input are the neural accelerators. However, most of them need a host to work, making these devices inappropriate for edge implementations. For this reason, edge devices that incorporate a neural accelerator in the same board as Cookie Coral and Coral DevBoard are the most suitable for edge applications. In this regard, there is an optimal trade-off between power consumption and performance. The Cookie Coral is designed for IoT application prototyping since its modular architecture offers greater flexibility compared with Coral DevBoard. This modularity allows the implementation of different processing, communication, power, and sensing layers according to the requirements of each specific application. Besides, the cost is reduced by half compared to the Coral DevBoard as it is a custom platform.

Among the architectures studied, the best accuracy-inference time ratio when processing point clouds is achieved by VoxNet since it has the lowest inference time by far, and the accuracy is slightly lower than that of PointNet. Besides, it is also the fastest alternative when considering the processing time and the preprocessing time. However, it should be noted that some DNN architectures that are not image-based do not provide accurate results when running into some neural accelerator edge devices. Therefore, for some edge devices the most suitable alternative is to use an image-based architecture such as MVDMNet.

As future work, an edge system that detects and classifies each object that enters a critical region will be developed and deployed into a real scenario. A LIDAR will be used as an input sensor to provide an accurate point cloud map of the area of interest. Then, this point cloud will feed a VoxNet DNN architecture and perform this processing using a Cookie Coral.

Acknowledgment

The document reflects only the author's view and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] S. Vora, A. H. Lang, B. Helou, and O. Beijbom, "PointPainting: Sequential fusion for 3D object detection," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2020, pp. 4603–4611.

[2] T. Raj, F. H. Hashim, A. B. Huddin, M. F. Ibrahim, and A. Hussain, "A survey on LiDAR scanning mechanisms," *Electronics*, vol. 9, no. 5, p. 741, Apr. 2020. [Online]. Available: https://www.mdpi.com/2079-9292/9/5/741

[3] A. Filgueira, H. González-Jorge, S. Lagüela, L. Díaz-Vilariño, and P. Arias, "Quantifying the influence of rain in LiDAR performance," *Measurement*, vol. 95, pp. 143–148, Jan. 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0263224116305577

[4] M. Elhousni and X. Huang, "A survey on 3D LiDAR localization for autonomous vehicles," in *Proc. IEEE Intell. Vehicles Symp. (IV)*, Oct. 2020, pp. 1879–1884.

[5] S. Parikh, D. Dave, R. Patel, and N. Doshi, "Security and privacy issues in cloud, fog and edge computing," *Proc. Comput. Sci.*, vol. 160, pp. 734–739, Jan. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050919317181

[6] K. Ota, M. S. Dao, V. Mezaris, and F. G. B. D. Natale, "Deep learning for mobile multimedia: A survey," *ACM Trans. Multimedia Comput., Commun., Appl.*, vol. 13, no. 3, pp. 1–22, Aug. 2017, doi: 10.1145/3092831.

[7] A. Voulodimos, N. Doulamis, A. Doulamis, and E. Protopapadakis, "Deep learning for computer vision: A brief review," *Comput. Intell. Neurosci.*, vol. 2018, pp. 1–13, Feb. 2018.

[8] B. Barry *et al.*, "Always-on vision processing unit for mobile applications," *IEEE Micro*, vol. 35, no. 2, pp. 56–66, Mar./Apr. 2015.

[9] L. Kljucaric, A. Johnson, and A. D. George, "Architectural analysis of deep learning on edge accelerators," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2020, pp. 1–7.

[10] R. Hadidi, J. Cao, Y. Xie, B. Asgari, T. Krishna, and H. Kim, "Characterizing the deployment of deep neural networks on commercial edge devices," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Nov. 2019, pp. 35–48.

[11] W. Liu, J. Sun, W. Li, T. Hu, and P. Wang, "Deep learning on point clouds and its application: A survey," *Sensors*, vol. 19, no. 19, p. 4188, Sep. 2019.

[12] Y. Guo, H. Wang, Q. Hu, H. Liu, L. Liu, and M. Bennamoun, "Deep learning for 3D point clouds: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 43, no. 12, pp. 4338–4364, Dec. 2021.

[13] J.-C. Su, M. Gadelha, R. Wang, and S. Maji, "A deeper look at 3D shape classifiers," in *Proc. Eur. Conf. Comput. Vis. (ECCV) Workshops*, 2018, pp. 1–16.

[14] J.-B. Weibel, R. Rohrböck, and M. Vincze, "Measuring the Sim2Real gap in 3D object classification for different 3D data representation," in *Proc. Int. Conf. Comput. Vis. Syst.* Cham, Switzerland: Springer, 2021, pp. 107–116.

[15] A. A. Süzen, B. Duman, and B. Şen, "Benchmark analysis of Jetson TX2, Jetson nano and raspberry PI using deep-CNN," in *Proc. Int. Congr. Hum.-Comput. Interact., Optim. Robot. Appl. (HORA)*, Jun. 2020, pp. 1–5.

[16] S. Qi *et al.*, "Review of multi-view 3D object recognition methods based on deep learning," *Displays*, vol. 69, Sep. 2021, Art. no. 102053.

[17] R. Q. Charles, H. Su, M. Kaichun, and L. J. Guibas, "PointNet: Deep learning on point sets for 3D classification and segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 652–660.

[18] D. Rempe, T. Birdal, Y. Zhao, Z. Gojcic, S. Sridhar, and L. J. Guibas, "CaSPR: Learning canonical spatiotemporal point cloud representations," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2020, pp. 13688–13701.

[19] L. Caltagirone, S. Scheidegger, L. Svensson, and M. Wahde, "Fast LiDAR-based road detection using fully convolutional neural networks," in *Proc. IEEE Intell. Vehicles Symp. (IV)*, Jun. 2017, pp. 1019–1024.

[20] H. Su, S. Maji, E. Kalogerakis, and E. G. Learned-Miller, "Multi-view convolutional neural networks for 3D shape recognition," in *Proc. ICCV*, 2015, pp. 945–953.

[21] D. Maturana and S. Scherer, "VoxNet: A 3D convolutional neural network for real-time object recognition," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Sep. 2015, pp. 922–928.

[22] E. Ahmed *et al.*, "A survey on deep learning advances on different 3D data representations," 2018, *arXiv:1808.01462*.

[23] Z. Wu *et al.*, "3D ShapeNets: A deep representation for volumetric shapes," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2015, pp. 1912–1920.

[24] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*. Savannah, GA, USA: USENIX Association, Nov. 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[25] P. Merino, G. Mujica, J. Señor, and J. Portilla, "A modular IoT hardware platform for distributed and secured extreme edge computing," *Electronics*, vol. 9, no. 3, p. 538, Mar. 2020. [Online]. Available: https://www.mdpi.com/2079-9292/9/3/538

[26] J. Portilla, A. de Castro, E. de la Torre, and T. Riesgo, "A modular architecture for nodes in wireless sensor networks," *J. Universal Comput. Sci.*, vol. 12, no. 3, pp. 328–339, 2006.

[27] V. G. Reddy, "Neon technology introduction," *ARM Corp.*, vol. 4, no. 1, pp. 1–33, 2008.

[28] S. Mittal and S. Vaishay, "A survey of techniques for optimizing deep learning on GPUs," *J. Syst. Archit.*, vol. 99, Oct. 2019, Art. no. 101635. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762119302656

[29] T. Chen *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. 13th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*. Carlsbad, CA, USA: USENIX Association, Oct. 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[30] M. Li *et al.*, "The deep learning compiler: A comprehensive survey," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 708–727, Mar. 2021.

[31] T. Moreau *et al.*, "A hardware–software blueprint for flexible deep learning specialization," *IEEE Micro*, vol. 39, no. 5, pp. 8–16, Sep./Oct. 2019.

[32] T. Chen *et al.*, "Learning to optimize tensor programs," in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst. (NIPS)*. Red Hook, NY, USA: Curran Associates, 2018, pp. 3393–3404.
[33] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and evaluation of the first tensor processing unit," *IEEE Micro*, vol. 38, no. 3, pp. 10–19, May/Jun. 2018.
[34] F. Daghero, D. J. Pagliari, and M. Poncino, "Energy-efficient deep learning inference on edge devices," in *Hardware Accelerator Systems for Artificial Intelligence and Machine Learning* (Advances in Computers), S. Kim and G. C. Deka, Eds. Amsterdam, The Netherlands: Elsevier, 2021, vol. 122, ch. 8, pp. 247–301. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0065245820300553
[35] S. Rivas-Gomez, A. J. Pena, D. Moloney, E. Laure, and S. Markidis, "Exploring the vision processing unit as co-processor for inference," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2018, pp. 589–598.
[36] A. Demidovskij *et al.*, "OpenVINO deep learning workbench: A platform for model optimization, analysis and deployment," in *Proc. IEEE 32nd Int. Conf. Tools Artif. Intell. (ICTAI)*, Nov. 2020, pp. 661–668.
[37] I. Bilbao and J. Bilbao, "Overfitting problem and the over-training in the era of data: Particularly for artificial neural networks," in *Proc. 8th Int. Conf. Intell. Comput. Inf. Syst. (ICICIS)*, 2017, pp. 173–177.
[38] E. Rapuano *et al.*, "An FPGA-based hardware accelerator for CNNs inference on board satellites: Benchmarking with myriad 2-based solution for the CloudScout case study," *Remote Sens.*, vol. 13, no. 8, p. 1518, Apr. 2021.

Cristian Wisultschew received the B.S. and M.Sc. degrees in industrial electronics from the Universidad Politécnica de Madrid, Madrid, Spain, in 2017 and 2018, respectively, where he is currently pursuing the Ph.D. degree.

He carries out his research activity with the Centro de Electrónica Industrial, UPM. He is the author of three papers published in international conferences and journals. His research interests are focused on sensor system integration, digital embedded systems, embedded machine learning, deep learning HW accelerators, and the Internet of Things. He is participating in two European H2020 research projects, SCOTT and InSecTT, and related to real-time object detection and classification systems deployed at the edge of IoT for railway level crossing applications. He is also participating in a Spain Government funded project, PLATINO, related to accelerating the processing of deep learning algorithms in embedded systems using specific deep learning neural accelerators.

Alejandro Pérez received the B.Sc. degree in industrial technologies engineering from the Universidad Politécnica de Madrid (UPM), Madrid, Spain, in 2020, where he is currently pursuing the M.Sc. degree in industrial engineering with emphasis in electronics and control automation. During the last year, he was awarded with a scholarship to collaborate with the Centro de Electrónica Industrial (CEI) on deep learning HW accelerators for three-dimensional representations. His research interests are focused on computer vision algorithms, deep learning HW accelerators, reconfigurable computing on FPGAs, heterogeneous processing, and cyber-physical systems.

Andrés Otero received the M.Sc. (Hons.) degree in telecommunication engineering from the University of Vigo in 2007 and the Master of Research and Ph.D. degrees in industrial electronics from the Universidad Politécnica de Madrid (UPM), in 2009 and 2014, respectively.

He is currently an Assistant Professor of Electronics with UPM and a Researcher with the Centro de Electrónica Industrial (CEI). His current research interests are focused on embedded system design, reconfigurable systems on FPGAs, evolvable hardware, and embedded machine learning. During the last years, he has been involved in different research projects in these areas. He is the author of more than 30 papers published in international conferences and journals. He has served as the Program Committee Member of different international conferences in the field of reconfigurable systems, such as SPL, ERSA, ReConFig, DASIP, and ReCoSoC.

Gabriel Mujica (Member, IEEE) received the Ph.D. degree in industrial electronics engineering from the Universidad Politécnica de Madrid.

He is an Assistant Professor and a Research Member with the Center of Industrial Electronics, Universidad Politécnica de Madrid, where he is mainly involved in the area of networked embedded systems and wireless sensor networks (WSN). He has participated in different national and European research projects (including Horizon 2020 projects), related to the development and optimization of WSN, and the integration of heterogeneous hardware, software, and communication technologies for wireless distributed systems, with a particular focus on the performance evaluation of sensor platforms under real deployment and commissioning. In this way, he has authored several contributions in high-impact conferences and journals. He has collaborated in the organization of research tutorials and seminars, and as a reviewer for several international conferences and journals (IEEE and Springer). Moreover, his visiting research stay at the Trinity College Dublin strengthened the vision and applicability of IoT technologies for smart and sustainable cities, leveraging collaborations in the area of distributed systems within such contexts. Currently, his main research interests are related to multi-hop distributed networks and hardware-software co-design and protocols for embedded systems in smart urban application scenarios.

Jorge Portilla (Senior Member, IEEE) received the M.Sc. degree in physics from the Universidad Complutense de Madrid, Madrid, Spain, in 2003, and the Ph.D. degree in electronic engineering from the Universidad Politécnica de Madrid (UPM), Madrid, in 2010.

He was a Visiting Researcher with the Industrial Technology Research Institute, Hsinchu, Taiwan, in 2008, and also with the National Taipei University of Technology (Taipei Tech), Taipei, Taiwan, in 2018, working on wireless sensor networks hardware platforms and network clustering techniques. He is currently a tenure Assistant Professor with the Universidad Politécnica de Madrid. He carries out his research activity with the Centro de Electrónica Industrial, belonging to UPM. He has participated in more than 30 funded research projects, including European Union FP7 and H2020 projects, and Spain Government funded projects, and private industry funded projects, mainly related to wireless sensor networks and the Internet of Things. He has numerous publications in prestigious international conferences and in journals with impact factor. His research interests are focused on wireless sensor networks, the Internet of Things, digital embedded systems, and reconfigurable FPGA-based embedded systems.