11

Efficient Edge Deployment Demonstrated on YOLOv5 and Coral Edge TPU

Ruben Prokscha, Mathias Schneider, and Alfred Höß

Ostbayerische Technische Hochschule Amberg-Weiden, Germany

Abstract

The recent advancements towards Artificial Intelligence (AI) at the edge resonate with an impression of a dichotomy between resource intensive, highly abstracted Machine Learning (ML) research and strongly optimized, low-level embedded design. Overcoming such opposing mindsets is imperative for enabling desirable future scenarios such as autonomous driving and smart cities. edge AI must incorporate both straightforward streamlined deployments together with resource efficient execution to achieve general acceptance. This research aims to exemplify how such an endeavour could be realized, utilizing a novel low power AI accelerator together with a state-of-the-art object detection algorithm. Different considerations regarding model structure and efficient hardware acceleration are presented for deploying Deep Learning (DL) applications in resource restricted environments while maintaining the comfort of operating at a high degree of abstraction. The goal is to demonstrate what is possible in the field of edge AI once software and hardware are optimally matched.

Keywords: edge AI, object detection, deep learning, YOLO, embedded systems, tensor processing unit.

11.1 Introduction

With AI shifting from a simple research subject towards end user applications, the issue of efficient deployment moves into focus. ML workloads

are decidedly different from average computing tasks. Hence, GPUs were the common solution for such undertakings. Realizing mobile intelligent appliances, requires even more specialized, low power accelerators which can be integrated into embedded environments. Such edge solutions attracted increasing interest within the last years. The European Strategic Research and Innovation Agenda (SRIA) [1] concretizes the term even further by introducing the terms Micro-, Deep- and Meta-edge. There are several different solutions available which target this new frontier. Most prominent are the NVIDIA Jetson family, which utilizes optimized embedded GPUs, the Intel Neural Compute Stick 2 which is comprised of a specialized Vision Processing Unit (VPU) and the Google Coral edge Tensor Processing Unit (TPU), which will be the focus of this work. As such, its impact on related research is presented in the following section. The task of object detection was chosen to be part of the experimental test setup for evaluating the accelerator. You Only Look Once (YOLO) version 5 [2] serves as delegate for these class of networks in the upcoming section. It is evaluated, how models can be modified to facilitate edge TPU characteristics. Furthermore, it is shown how this optimized solution compares to models provided by Google. With a focus on deployment, a lightweight software stack is introduced which enables efficient AI solutions without sacrificing high-level development. Finally, a conclusion is provided giving a synapsis of the key findings and offering points of interest for future work.

11.2 Related Work

In recent years, the usage of decentralized AI at the edge has become a progressively relevant research topic. Thereby, besides GPU acceleration, the energy-efficient edge TPU was of special interest by research fellows. For applications with strict power or battery limitation, such as in the area of UAV, the usage of the edge TPU is evaluated in recent work. Thereby, applications comprise indoor person-following systems [3], vision-based trash and litter detection [4], and lightweight odometry estimation [5]. Using a U-Net network architecture, Roesler et al. leverage their edge AI setup combining the edge accelerator with a STM32MP157C-DK2 board for the yield estimation of grapes in an agriculture use case [6]. But also, other application domains are explored, e.g., in [7], which utilizes the edge TPU to process time-series data to determine the remaining useful life. Since at that time Recurrent Neural Networks (RNNs) were not yet supported by the accelerator, their model architecture employs a deep Convolutional Neural Network (CNN). It

is worth mentioning that their experiments included measurements for models using quantization-aware training as well as post-training quantization, which outperformed reference CPU and GPU deployments in terms of latency and accuracy. The authors in [8] examine the potential of the edge TPU for detecting network intrusion to ensure security at the edge using feed forward and CNN architectures. They elaborate their classification scores on a public benchmark dataset, and further investigate the energy efficiency of their DL algorithms in comparison to traditional CPU processing. Their studies on the effects of larger model sizes reveal a bimodal behaviour of the edge accelerator, indicating a decline of the energy efficiency ratio as soon as a certain model size is exceeded. This finding is the focus of their consecutive work and is confirmed by more refined experiments [9].

Besides this applied research of utilizing the edge accelerator for a dedicated application, more theoretical research was conducted to explore and demarcate TPU capabilities. Therefore, several benchmarks were performed to determine its performance empirically using various setups differing in the models under test, obtained metrics, or compared edge devices [10, 11, 12]. Providing micro-architectural insights, Google researcher, Yazdanbakhsh et al., elaborate an extensive evaluation covering different structures in CNNs and their effects on latency and energy consumption [13]. With a similar level of hardware details, the authors in [14] analysed the inference of 24 Google edge models, revealing major shortcomings of the edge TPU architecture which must be taken into account for efficient deployment. Furthermore, they incorporate the results into their framework for heterogeneous edge ML accelerators called Mensa, improving the edge TPU performance significantly.

11.3 Experimental Setup

Figure 11.1 depicts the setup used for this research. A Raspberry Pi 4 Model B with 4 GB memory served as base platform. The Google Coral edge TPU accelerator was connected either to a USB2 or USB3 port for performance and accuracy evaluation.

11.3.1 Google Coral Edge TPU

Google developed a custom Application Specific Integrated Circuit (ASIC) for edge inference. This specialized TPU can be connected to existing systems utilizing a USB, (m)PCIe or M.2 interface. Figure 11.1 depicts the USB

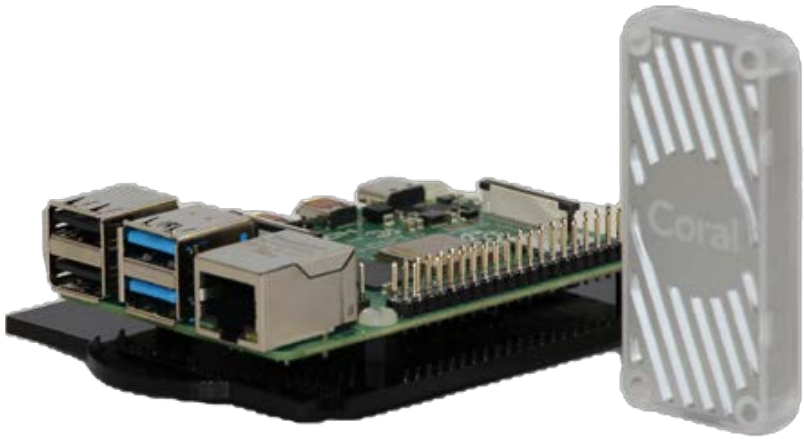


Figure 11.1 Raspberry Pi 4 with Google Coral edge TPU USB accelerator.

dongle variant of the accelerator with is advertised to perform up to four trillion operations per second. Approximately 8 MB 'scratchpad' memory is available per unit and the peak power consumption is rated at 2 W [15]. Additionally, multiple of these coprocessors can be chained together for handling bigger workloads. The TPU hardware operates on 8 Bit integer variables. Both performance and power consumption benefit from a reduced complexity in the hardware design. However, this introduces weight quantization as an additional step before deployment. The reduction in precision from floating point to 8 Bit integers variables subsequently leads to a deterioration of accuracy. Further overhead is introduced by the addition of quantization operations to the execution graph.

Deploying a model for this device entails several pitfalls due to a rather convoluted development pipeline. Google necessitates its own Tensorflow (TF) framework as starting point. Hence, models from other frameworks must be converted by means of e.g., Open Neural Network Exchange (ONNX). There, a quantization step is performed alongside a conversion to the TFLite format. The final step involves a proprietary edge TPU compiler, which translate the TFLite instructions for the edge TPU. Inference on the other hand is straight forward. The TFLite runtime provides the interfaces for loading and executing the model file, while the libedgetpu is responsible for handling the low-level communication with the accelerator. This allows for a very lightweight deployment of 10 MB to 20 MB (without model) compared to conventional GPU solutions, which can require over a gigabyte disk storage for the libraries alone.

11.3.2 YOLOv5

The original You Only Look Once (YOLO) architecture was proposed by Joseph Redmon in 2016 [16]. It performs both object detection and classification in a single model. This resulted in a significant performance increase compared to classical two stage designs (e.g., Region Based Convolutional Neural Networks (R-CNNs) [17]). Since the original design, many improvements were made. YOLOv5 [2] is based on the YOLOv3 [18] architecture. It is under constant open-source development by Ultralytics, who shifted the focus from academic research to accessible deployment. They provide an end-to-end solution which allows for training, testing and exporting models to a variety of different deployment frameworks. This includes the integration of the previously described pipeline for generating edge TPU models from version 6.1 onward.

11.4 Performance Considerations

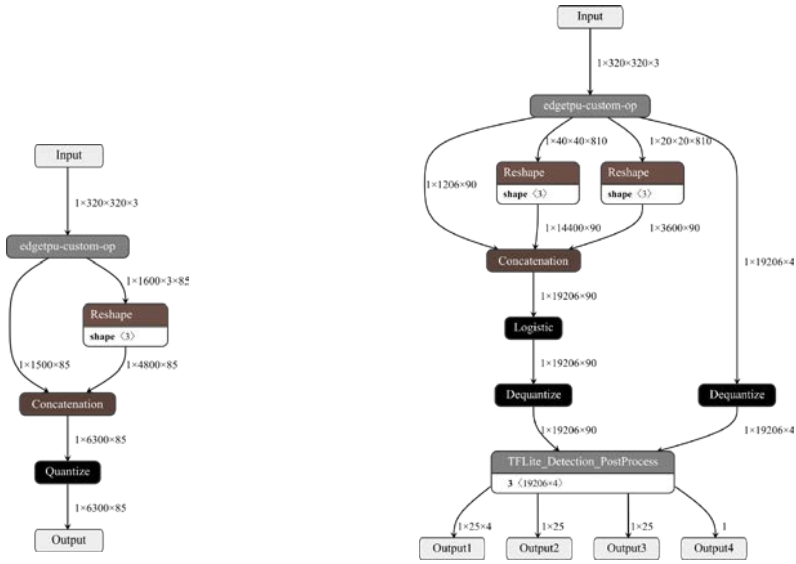
The Coral accelerator achieves its low energy footprint and high performance by sacrificing flexibility. This manifests itself in a significantly reduced instruction set [19]. The edge TPU compiler is a black box which aims to aggregate as much operations as possible and convert them into a binary which can be executed by the coprocessor. Every operation, which is not mapped accordingly, must therefore run-on CPU. This section aims to provide guidance for optimizing a model for edge TPU execution exemplified on YOLOv5 (release 6.1).

11.4.1 Graph Optimization

Figure 11.2 depicts the graphs of two edge TPU models. Figure 11.2a shows the small variant of the YOLOv5 model with additional optimizations. The EfficientDet Lite0 [20] model in Figure 11.2b was taken from the Coral model zoo [21]. Most of the graph is mapped to the edgetpu-custom-op, while some operations are still executed by the main processor. In the following, possible issues are shown when compiling a model and ways to improve the mapping are elaborated.

11.4.1.1 Incompatible Operations

The compiler only maps operations until it encounters an incompatibility. Everything after that is executed on the CPU. This is especially critical for activation functions (e.g., LeakyReLU, Hardswish), as they are distributed



(a) YOLOv5s (b) EfficientDet Lite0

Figure 11.2 Quantized edge TPU Models.

throughout the graph. While it is possible to create multiple TPU subgraphs, the overhead of transferring intermediate tensors several times between CPU and TPU usually eliminates any benefits. It is therefore advisable to use compatible activation functions (e.g., ReLU, Logistic). Furthermore, binary operations (e.g., AND, OR) are also not supported.

11.4.1.2 Tensor Transformations

The reshape and transpose operation are not mapped once their input tensor exceeds a certain soft threshold. There is no documentation on how this limit is calculated and it seems to be dependent on the general model structure. However, it could be observed, that this threshold is significantly smaller for the transpose operation. A possible explanation for this behaviour could be an inability of the accelerator to address memory in a different order. A transpose operation on CPU would imply a change in the direction (column/row wise) memory is read from the same location. If this is not supported by the TPU, memory reallocation is required.

There are several ways for addressing this issue. One approach is to reduce the size of the input tensor. In CNNs the size is proportionally

Table 11.1 Comparison of YOLOv5s model before and after optimizations.

Input Size	YOLOv5s (6.1)		YOLOv5s (6.2dev)		Speedup
	TPU/CPU	USB3 Speed	TPU/CPU	USB3 Speed	
320x320	245/40	30.10ms	253/3	24.27ms	19.37%
640x640	225/59	427.48ms	240/16	178.67ms	58.20%

propagated through the network. Hence, reducing the input size results in smaller intermediate tensors. Further reduction can be induced by limiting the number of output classes. If graph modifications are viable, a divide and conquer strategy can be used to split tensors before the operation and merging afterwards. Moving these operations to the bottom of the graph can also be an option as the instruction are fast on CPU. A last option is using mathematical transformation to change the graph beneficially.

Some of these strategies were used to optimize the YOLOv5 models which are evaluated further in this research. All changes were committed to the open-source project in a pull request [22] and are part of the next major release (6.2). Table 11.1 shows the performance impact for the demonstrator setup. Both model variants experienced a significant speedup in inference time. The variant with the larger input size improves significantly.

11.4.2 Performance Evaluation

In the following, different variants of the optimized YOLOv5 models are compared to other object detectors supplied by Google. All numerical values can be found in Table 11.2. The inference speed was evaluated utilizing the Google benchmark model tool [23]. Version 16 of libedgetpu-max was used, and each inference was repeated 100 times with a previous warm-up phase. Accuracy was determined by pycocotools and the Common Objects in Context (COCO) evaluation dataset [24]. The input images were proportionally scaled to input size with bilinear interpolation. The Google models have a postprocessing operation integrated in the model graph (ref. Figure 11.2b). It was evaluated separately for inference speed and fast Non-Maximum Suppression (NMS) [25] was used for all models as it is the default setting of this custom operation. Furthermore, the threshold for confidence was set to 0.001 and overlap to 0.65.

11.4.2.1 Speed-Accuracy Comparison

Figure 11.3 shows the mean average precision ($mAP_{50:95}$) of each tested model in relation to the inference speed. It can be observed that the edge TPU

Table 11.2 Model comparison in regards of input size, file size, operation

	Input Size	**Size in MB**	**Ops**	**Accuracy**						**Speed in ms**			
			TPU/CPU	**$mAP_{50:95}$**	**mAP_{50}**	**mAP_{75}**	**mAP_S**	**mAP_M**	**mAP_L**	**USB2**		**USB3**	
EfficientDet Lite0	320x320	5.93	260/7	24.30	39.10	25.40	5.50	27.30	42.20	164.06	+21.6	57.57	+21.79
EfficientDet Lite1	384x384	8.01	315/7	28.90	45.10	30.70	8.80	33.20	47.00	243.16	+30.57	81.43	+31.03
EfficientDet Lite2	448x448	10.67	349/8	32.30	48.90	34.80	12.20	37.10	49.60	490.06	+40.44	152.77	+39.58
SSD Mobilenetv2	300x300	7.08	99/3	15.50	27.20	15.30	1.20	10.90	34.30	31.68	+2.01	10.90	+2.06
SSDLite MobileDet	320x320	5.4	134/3	22.50	38.30	23.30	2.50	19.40	48.20	34.98	+8.77	9.56	+5.93
YOLOv5n	320x320	2.38	253/3	18.10	32.40	18.10	3.00	17.60	32.60	64.35		24.64	
	480x480	2.47	240/16	22.20	39.60	22.50	6.10	24.60	34.40	127.57		50.41	
	640x640	2.43	240/16	22.70	41.10	22.80	8.90	26.70	31.00	262.79		93.60	
YOLOv5s	320x320	7.84	253/3	26.10	43.50	27.00	6.40	27.80	45.10	67.76		24.27	
	400x400	7.96	240/16	28.50	47.40	29.40	8.90	31.60	45.10	162.15		49.42	
	480x480	8.09	240/16	29.80	49.20	31.30	11.30	33.70	44.70	238.43		77.77	
	640x640	8.78	240/16	30.20	50.50	31.60	14.10	35.00	41.10	711.97		178.67	
YOLOv5m	240x240	22.13	325/3	28.80	46.70	30.30	6.80	31.00	51.10	460.60		63.61	
	320x320	22.32	325/3	32.40	51.30	34.40	10.40	36.10	53.60	542.08		85.98	
	480x480	23.33	313/16	36.90	58.00	39.10	16.20	41.90	53.30	1268.55		238.90	
YOLOv5l	320x320	47.77	399/3	35.40	55.10	37.60	12.70	40.00	56.60	1296.39		177.60	

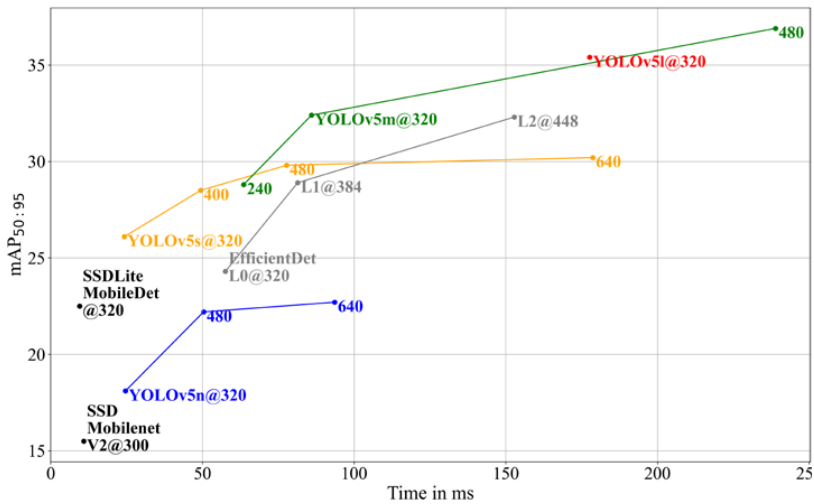


Figure 11.3 USB3 speed-accuracy comparison of different model types and configurations for edge TPU deployment.

works best lower input sizes, while larger inputs cause an unproportionate slowdown compared to the benefit in accuracy. Interesting are the nano and small models with 320 px input. They have an almost identical inference time, while the accuracy of the s-model is significantly better. They share the same vertical graph structure, while the larger one is scaled horizontally by a factor of two. Hence, the small variant has twice as many weights for each convolutional layer. This aligns with the insights from [12] that horizontal scaling is preferable. The model should be very close to a sweet spot, for which all weights are cached within the 8 MB device memory. Sacrificing some model vertical space for more width could theoretically improve the accuracy even further.

In general, YOLOv5 performs better than the other models. Only the nano model has issues, which is probably caused by its particularly small file size. If speed is the deciding factor, SSDLite MobileDet [26] [27], is still the preferable solution. The classical SSD Mobilenetv2 [26] [28] does not seem to be competitive anymore. The EfficientDet models perform reasonable, however considering the additional overhead by a particularly slow postprocessing operation, YOLOv5 should be considered the better solution. All models share a low accuracy for small objects, which could be an issue inflicted by quantization.

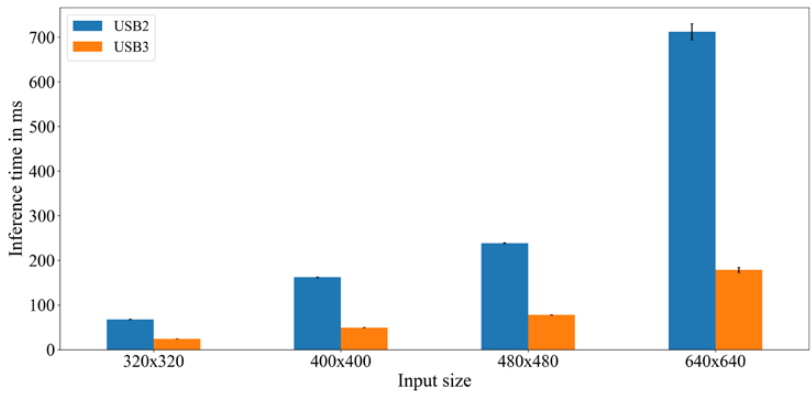


Figure 11.4 YOLOv5s inference speed comparison between USB2 and USB3

11.4.2.2 USB Speed Comparison

Considering the purpose of edge accelerators to allow for AI deployment on low power devices, USB3 might not always be an option. Hence, it should be evaluated whether a deployment utilizing USB2 is a viable option. The maximum speed for such a connection is rated at 60 MB/s, while USB3 is specified at almost ten times this value.

Figure 11.4 depicts the speed comparison the small model with varying input size. A considerable difference for the inference speed can be observed. The USB2 interface causes a slowdown by a factor of three. The model parameters should fit entirely into the device memory. Therefore, only the data transfer should impact the speed. Equation (11.1) shows how the data flowing to and from the device is calculated. $data_{\text{in}}$ only depends on the input size, while $data_{\text{out}}$ also considers the number of anchor boxes (3), strides for multi scale outputs (8, 16, 32) and class count. For the 320 px model, this results in 842.7 KB of data flow per inference, while the 640 px input increases this value to 3.37 MB. Additional data flow could arise due to intermediate tensors, which are too large to be buffered on the device. Whether this is an issue here must be determined in future research.

$$\begin{aligned} \text{data}_{\text{in}} &= 3\text{x}_{\text{in}}\text{y}_{\text{in}} \\ \text{data}_{\text{out}} &= 3\left(\frac{1}{8^2}+\frac{1}{16^2}+\frac{1}{32^2}\right)\text{x}_{\text{in}}\text{y}_{\text{in}} * (5+\text{n}_{\text{cls}}) \end{aligned} \tag{11.1}$$

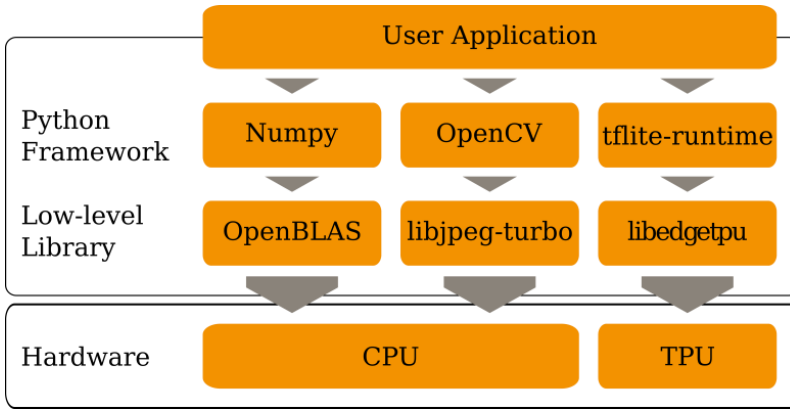


Figure 11.5 Micro software stack for fast and lightweight edge deployment.

11.4.3 Deployment Pipeline

An AI application can be considered a data pipeline of the steps. At first data must be loaded and pre-processed to comply with the model. In the context of object detection, this implies loading and scaling a jpeg image. The following steps are inference and postprocessing. The latter takes the raw model output and transforms into a usable form. This could involve thresholding, NMS and coordination transforms. The pipeline is executed for each inference, hence all steps should be highly optimized. Most efforts are usually focused towards optimizing the model while neglecting everything else. This section introduces a small deployment stack for object detection, which is both optimized and allows for the usage of well-established high-level frameworks.

The software stack depicted in Figure 11.5 shows a simple layer model for a lightweight deep vision deployment. The part concerning the TPU was previously elaborated. Loading and transforming images is often handled by OpenCV [29]. It uses shared low-level libraries to perform these operations. Providing an optimized image loader, such as libjpeg-turbo [30] can therefore accelerate the whole pipeline. Similar is true for Numpy [31], which is responsible for performing mathematical tensor operations on CPU. A dedicated math library such as OpenBLAS [32] makes use of Single Instruction Multiple Data (SIMD) which performs vector operations faster and more efficient. Such a software stack is similarly fast compared to a solution written in a compiled language, while being way more flexible. It could also be viable to package such an application into a lightweight container for easy deployment using virtualization technologies.

11.5 Conclusion and Future Work

This research demonstrated how efficient edge AI applications can be implemented in a feasible manner. It was shown that a high degree of optimization is required to make the best use of limited computing resources. Additionally, a lightweight software stack was presented, which can be used as basis for building high level ML applications. A paradigm shift towards a more deployment driven AI development, as portrait by YOLOv5, is mandatory for making ubiquitous AI possible. The Google Coral edge TPU offers high potential for enabling real-time object detection for common video stream rates on embedded systems, however there are several pitfalls associated with the device. The limited opset requires models to be designed accordingly, which must be in the interest of the developers. Another issue is the USB2 performance. Future research must evaluate, what exactly causes this drastic slowdown. If the TPU should be used in ultra-low power segments (e.g., Micro Controller Units), USB3 will not be viable. Changing the model to reduce the amount of data flowing to and from the device could alleviate this shortcoming.

Acknowledgements

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

References

[1] AENEAS, Inside Industry Association, and EPOSS. ECS – Strategic Research and Innovation Agenda 2022. en. Jan. 2022. URL: https://ecscollaborationtool.eu/publication/download/slides-ovidiu-vermesan.pdf (visited on 03/31/2022).

[2] G. Jocher et al. ultralytics/yolov5: v6.1 - TensorRT, TensorFlow Edge TPU and OpenVINO Export and Inference. Feb. 2022. URL: https://zenodo.org/record/6222936 (visited on 03/30/2022).

[3] A. Boschi et al. "A Cost-Effective Person-Following System for Assistive Unmanned Vehicles with Deep Learning at the Edge". en. In: Machines 8.3 (Aug. 2020), p. 49.

[4] M. Kraft et al. “Autonomous, Onboard Vision-Based Trash and Litter Detection in Low Altitude Aerial Images Collected by an Unmanned Aerial Vehicle”. en. In: Remote Sensing 13.5 (Mar. 2021), p. 965.

[5] N. J. Sanket et al. “PRGFlow: Benchmarking SWAP-Aware Unified Deep Visual Inertial Odometry”. en. In: arXiv:2006.06753 [cs] (June 2020).

[6] M. Roesler et al. “Deploying Deep Neural Networks on Edge Devices for Grape Segmentation”. en. In: Smart and Sustainable Agriculture. Ed. by Selma Boumerdassi, Mounir Ghogho, and Éric Renault. Vol. 1470. Cham: Springer International Publishing, 2021, pp. 30–43.

[7] C. Resende et al. “TIP4.0: Industrial Internet of Things Platform for Predictive Maintenance”. en. In: Sensors 21.14 (July 2021), p. 4676.

[8] S. Hosseininoorbin et al. “Exploring Edge TPU for Network Intrusion Detection in IoT”. en. In: arXiv:2103.16295 [cs] (Mar. 2021).

[9] S. Hosseininoorbin et al. “Exploring Deep Neural Networks on Edge TPU”. en. In: arXiv:2110.08826 [cs] (Oct. 2021).

[10] M. Alnemari and N. Bagherzadeh. “Efficient Deep Neural Networks for Edge Computing”. en. In: 2019 IEEE International Conference on Edge Computing (EDGE). Milan, Italy: IEEE, July 2019, pp. 1–7.

[11] M. Antonini et al. “Resource Characterisation of Personal-Scale Sensing Models on Edge Accelerators”. en. In: Proceedings of the First International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things. New York NY USA: ACM, Nov. 2019, pp. 49–55.

[12] A. A. Asyraaf Jainuddin et al. “Performance Analysis of Deep Neural Networks for Object Classification with Edge TPU”. In: 2020 8th International Conference on Information Technology and Multimedia (ICIMU). Aug. 2020, pp. 323–328.

[13] A. Yazdanbakhsh et al. An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks. Feb. 2021.

[14] A. Boroumand et al. “Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks”. en. In: arXiv:2109.14320 [cs] (Sept. 2021).

[15] USB Accelerator datasheet. en-us. URL : https://coral.ai/docs/accelerator/datasheet/ (visited on 03/31/2022).

[16] J. Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). June 2016, pp. 779–788.

[17] R. Girshick et al. “Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation”. In: 2014 IEEE Conference on Computer Vision and Pattern Recognition. June 2014, pp. 580–587.

[18] J. Redmon and A. Farhadi. “YOLOv3: An Incremental Improvement”. In: (Apr. 2018).

[19] TensorFlow models on the Edge TPU. en-us. URL: https://coral.ai / docs / edgetpu / models - intro / #supported – operations (visited on 03/30/2022).

[20] M. Tan, R. Pang, and Q. V. Le. “EfficientDet: Scalable and Efficient Object Detection”. In: arXiv:1911.09070 [cs, eess] (July 2020). arXiv: 1911.09070.

[21] Models - Object Detection. en-us. URL: https://coral.ai/models/object-detection/.

[22] EdgeTPU optimizations by paradigmn Pull Request #6808 ultralytics/yolov5. en. URL: https://github.com/ultralytics/yolov5/pull/6808 (visited on 03/31/2022).

[23] Performance measurement — TensorFlow Lite. en. URL : https://www.tensorflow.org/lite/performance/measurement (visited on 03/30/2022).

[24] T.-Y. Lin et al. “Microsoft COCO: Common Objects in Context”. en. In: Computer Vision – ECCV 2014. Ed. by David Fleet et al. LectureNotes in Computer Science. Cham: Springer International Publishing, 2014, pp. 740–755.

[25] J. Hosang, R. Benenson, and B. Schiele. “Learning Non-maximum Suppression”. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). ISSN: 1063-6919. July 2017, pp. 6469–6477.

[26] W. L. et al. “SSD: Single Shot MultiBox Detector”. en. In: Computer Vision – ECCV 2016. Ed. by Bastian Leibe et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 21–37.

[27] Y. Xiong et al. “MobileDets: Searching for Object Detection Architectures for Mobile Accelerators”. In: arXiv:2004.14525 [cs] (July 2020). arXiv: 2004.14525.

[28] M. Sandler et al. “MobileNetV2: Inverted Residuals and Linear Bottlenecks”. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition. June 2018, pp. 4510–4520.

[29] G. Bradski. “The OpenCV Library”. In: Dr. Dobb’s Journal of Software Tools (2000).

[30] libjpeg-turbo. original-date: 2015-07-27T07:11:54Z. Mar. 2022. URL: https://github.com/libjpeg-turbo/libjpeg-turbo (visited on 03/31/2022).

[31] C. R. Harris et al. “Array programming with NumPy”. en. In: Nature 585.7825 (Sept. 2020), pp. 357–362.

[32] Q. Wang et al. “AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs”. en. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. Denver Colorado: ACM, Nov. 2013, pp. 1–12.