Characterizing the Deployment of Deep Neural Networks on Commercial Edge Devices

Ramyad Hadidi, Jiashen Cao, Yilun Xie, Bahar Asgari, Tushar Krishna, Hyesoon Kim
Georgia Institute of Technology
Email: rhadidi@gatech.edu

***Abstract*—The great success of deep neural networks (DNNs) has significantly assisted humans in numerous applications such as computer vision. DNNs are widely used in today's applications and systems. However, in-the-edge inference of DNNs is still a severe challenge mainly because of the contradiction between the inherent intensive resource requirements of DNNs and the tight resource availability of edge devices. Nevertheless, in-the-edge inferencing preserves privacy in several user-centric domains and applies in several scenarios with limited Internet connectivity (e.g., drones, robots, autonomous vehicles). That is why several companies have released specialized edge devices for accelerating the execution performance of DNNs in the edge. Although preliminary studies have characterized such edge devices separately, a unified comparison with the same set of assumptions has not been fully performed. In this paper, we endeavor to address this knowledge gap by characterizing several commercial edge devices on popular frameworks using well-known convolution neural networks (CNNs), a type of DNN. We analyze the impact of frameworks, their software stack, and their implemented optimizations on the final performance. Moreover, we measure energy consumption and temperature behavior of these edge devices.**

I. Introduction & Motivation

Deep neural networks (DNNs) and machine learning techniques have achieved great success in solving several traditionally challenging problems [1]–[3] such as computer vision [4], [5] and video recognition [6], [7]. DNNs are widely used today in numerous applications, from recommender systems [8] to autonomous vehicles [9], [10]. However, the execution platform of several of these applications lacks the resources for the efficient execution of DNNs, such as inexpensive robots [11]–[14], unmanned aerial vehicles (UAVs) [15], [16], and Internet-of-things (IoT) devices [17]. This is because the fast prediction of DNNs (i.e., inferencing) is a resource-intensive task [18] that requires energy, large memories, and capable processors. The traditional solution to this problem is to offload all the computations to the cloud. Nevertheless, such offloading is not possible in several situations because of privacy concerns [19]–[22], limited Internet connectivity, or tight-timing constraints (e.g., home video recordings, drones, and robots surveying a disaster area).

To amortize the cost of DNN, researchers have studied several machine learning techniques, such as weight pruning [18], [23]–[25], quantization [26]–[29], and mixed precision inferencing [30]. Furthermore, in-the-edge inferencing enforces the computation of single-batch inference because of the limited number of available requests in a given time. Compared to multi-batch inferencing (the common practice in cloud servers), single-batch inferencing increases memory footprint, which is an added challenge for in-the-edge devices. Therefore, edge specific DNN frameworks, such as TensorFlow Lite [31] and TensorRT [32], integrate and adapt aforementioned techniques for single-batch inference to achieve high performance. Moreover, companies have started to build device-specific frameworks for efficient DNN execution, such as Microsoft ELL [33] for Raspberry Pis [34] with several compiler- and software-level optimizations. However, using only software techniques cannot guarantee the fast execution of DNNs. This is because current hardware platforms are not specifically designed for DNNs, the execution of which has unique characteristics. This inefficiency of general-purpose hardware platforms for DNN execution has led to specialized hardware designs and ASIC chips targeting high-performance computing (HPC). Additionally, companies have also released specialized accelerator devices for performing fast in-the-edge inferencing, such as Google's EdgeTPU [35], Nvidia's Jetson Nano [36], and Intel's Movidius Neural Computer Stick [37].

This paper presents a unified comparison between commercial edge devices (Table III) with the same set of assumptions among several frameworks (Table II) with the same DNN models (Table I). While focusing on edge-specific single-batch inferencing, we analyze and compare the performance of several widely used frameworks and their supported optimizations for edge devices. To gain better insights, we profile the software stack of two widely used frameworks (TensrorFlow and PyTorch) on two CPU- and GPU-based edge devices (Raspberry Pi 3B and Jetson TX2). Additionally, we study accelerator-oriented frameworks for edge devices, such as Movidius toolkit. Besides the characterization of several edge devices, to the best of our knowledge, this is the first characterization of EdgeTPU and Jetson Nano[1]. In addition to the performance characterization of edge devices, we investigate whether HPC-level devices (Xeon and HPC GPUs) are a good candidate for single-batch inferencing. Finally, by using a power analyzer and thermal camera, we measure the energy per inference and temperature behavior of edge devices. Our experiments are reproducible and extendable to new platforms by utilizing our open source project on GitHub[2].
The following are the key contributions of this paper:

[1] Besides the marketing news blogs by companies [35], [38].
[2] More info at comparch.gatech.edu/hparch/edgeBench.

TABLE I

An overview of DNN[†] models used in the paper.

Model Name	Input Size	FLOP (giga)	Number of Parameters	FLOP/Param.
ResNet-18 [44]	224x224	1.83	11.69 m	156.54
ResNet-50 [44]	224x224	4.14	25.56 m	161.97
ResNet-101 [44]	224x224	7.87	44.55 m	176.66
Xception [45]	224x224	4.65	22.91 m	202.97
MobileNet-v2 [46]	224x224	0.32	3.53 m	90.65
Inception-v4 [47]	224x224	12.27	42.71 m	287.29
AlexNet [48]	224x224	0.72	102.14 m	7.05
VGG16 [5]	224x224	15.47	138.36 m	111.81
VGG19 [5]	224x224	19.63	143.66 m	136.64
VGG-S [5]	32x32	0.11	32.11 m	3.42
VGG-S [5]	224x224	3.27	102.91 m	31.77
CifarNet [49]	32x32	0.01	0.79 m	12.65
SSD [39] with MobileNet-v1 [40]	300x300	0.98	4.23 m	236.07
YOLOv3 [41], [42]	224x224	38.97	62.00 m	628.54
TinyYolo [42]	224x224	5.56	15.87 m	350.35
C3D [43]	12x112x112	57.99	89.00 m	734.05

[†] In this paper, we focus on convolution neural network (CNN) models, a type of DNN, because of their popularity and support in several studied frameworks. Nevertheless, most of the computations in CNNs, similar to other DNNs, are dominated by matrix-matrix and matrix-vector multiplications.

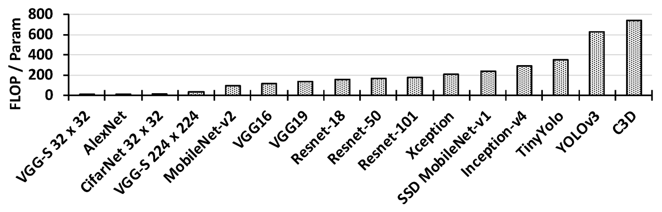

Fig. 1. DNN models of this paper, sorted by FLOP/Param for one inference.

- Analyzing commonly used frameworks, their optimizations for in-the-edge inferencing, and their software stack.
- Studying the impact of edge-specific frameworks on the performance of single-batch inference.
- Measuring energy consumption per inference and temperature behavior of edge devices.
- Comparing performance of single-batch inference of HPC-level and edge devices.
- Characterizing EdgeTPU and Jetson Nano for the first time, to the best of our knowledge.

II. DNN Models Overview

In the Table I, we present a brief overview of the DNN models used in this paper. We choose several computer vision models for object recognition, one for object detection (SSD [39] with feature extractor based on MobileNet-v1 [40]) and three for video recognition (YOLO [41], TinyYolo [42], and C3D [43]). All the models are based on convolution neural network (CNN), which mainly utilize convolution and fully-connected layers. The C3D model uses 3D convolution layers to process time series. We select computer vision DNN models since most frameworks support CNNs and their underlying implementations are optimized. We plan to extend our models to include more varieties of DNN models, such as RNNs and LSTMs, in the future work. We ensure that all the implementations are in fact identical for performing single-batch inferences across all framework implementations.

Table I lists the DNN models used in this paper, along with their number of parameters and their total floating operations (FLOP) per one inference, a proxy for memory, and a proxy for computation footprints, respectively. Additionally, in the table, we calculate FLOP per parameter (for one inference), which shows the degree of the compute intensity of the model. Figure 1 illustrates models sorted by their FLOP/Param. From purely an execution performance perspective, a model with higher FLOP/Param is more compute-intensive than a model with lower FLOP/Param, which is more memory-intensive. From a machine learning performance perspective, one can argue that a relatively high number of parameters could indicate a poorly designed model that does not efficiently uses the full potential of its parameters. For instance, a model that has a higher top-1 accuracy density (%/parameter) [50] could be more efficient. However, there is no clear consensus on this topic and such discussion is out of the scope of this paper. Therefore, we chose DNN models that are popular and are actively referenced and studied in the research community.

III. DNN Frameworks

This paper seeks to cover the most popular off-the-shelf DNN frameworks actively used in the industry or academia. Yet, setting up all the possible combinations of frameworks, devices, models, and weights is a highly time-consuming process, given that each framework usually requires its own model description format. Recent endeavors such as ONNX ecosystem [51] try to address this issue, but they are still in the introduction stage. Although some frameworks provide a predefined set of models, ensuring a unified model across all of frameworks is also time-consuming. Even after putting aside FPGA-based frameworks, which create custom hardware designs that are highly dependent on the model parameters, device-specific frameworks, such as TensorFlow Lite [31] used for EdgeTPU and NCSDK toolkit [52] used for Intel Movidius Stick, require extra steps (e.g., quantization-aware training, recompiling the model) for deployment. This section briefly describes the frameworks used in this paper.

A. Description of Frameworks

(1) TensorFlow: TensorFlow [53] is a widely used framework developed by Google. The majority of the software is available as open source with an Apache 2.0 license. The TensorFlow engine is written in C/C++ (CPU) and CUDA (GPU), and has several interfacing languages, including Python. Additionally, the TensorFlow ecosystem provides several visualizing (e.g., TensorBoard) and cross-platform compilation tools (e.g., EdgeTPU and TPU). For execution, TensorFlow generates a *static* computational graph (after the acceptance of this paper, a beta version of TensorFlow 2.0 [54] has been announced that supports dynamic graphs besides Eager execution mode [55]). Similar to all frameworks that train a network, TensorFlow adds automatic differentiation [56] to this graph for training. The automatic differentiation of TensorFlow eases the design of new models and introducing new learnable parameters since backpropagation operations for computing gradients are automatically defined in the computational graph.

(2) TensorFlow-Lite: TensorFlow-Lite [31] (TFLite) is the wrapper of the TensorFlow engine for mobile and IoT devices. To reduce the memory and computation footprints, TFLite performs various optimizations on the computation graph and weights of the model. For instance, TFLite offers pruning, structured pruning, and quantization (both post-training and during the training). For deployment, TFLite *freezes* the computation graph by removing several redundant and unnecessary operations (e.g., converting variable to constant operations).

(3) Keras: Keras [57] is a high-level deep learning API that is built on top of TensorFlow. It is written in Python and is released under the MIT license. The interface of Keras has been developed to provide easy and fast prototyping and to minimize the idea-to-result time. Keras is now integrated into TensorFlow. Therefore, for some models for which Keras has an implementation, we use Keras and TensorFlow implementations interchangeably.

(4) Caffe/2: Caffe2 [58] is the successor and a lightweight version of Caffe [59], an academic endeavor, now supported by Facebook. While Caffe supports deploying CNN models on clusters or mobile devices, it is more useful for large-scale use cases rather than mobile, distributed computation, and reduced precision computation use cases. To this end, Caffe2 supports large-scale distributed training, mobile deployment, new hardware, and quantized computation. Moreover, Caffe2 is intended to be modular to enable the fast prototyping of ideas and scalability. Caffe and Caffe2 are written in C++ and CUDA and are open sourced under the Apache 2.0 license. Caffe2 offers Python APIs for its engine.

(5) Movidius NCSDK Toolkit: The neural computing device of Intel, known as Movidius Neural Compute Stick (see Section IV), requires its compatible toolkit, Movidius Neural Compute SDK (NCSDK) [52], to run DNNs. Since the optimizations supported by the NCSDK toolkit are hand-tuned, importing and compiling a new model is a strenuous process [60] (NCS2 [61], announced after this paper acceptance, claims supporting popular frameworks).

(6) PyTorch: PyTorch [62], the Python version of Torch, a computational framework written in Lua, was open-sourced by Facebook in 2017 under the BSD license. Caffe2 (in C/C++) was merged into PyTorch in Facebook in 2018. Since Torch was originally developed as an academic project, it features a large number of community-driven packages and tools. PyTorch, in contrast with TensorFlow, closely represents the Python scientific computing library (i.e., numpy). Internally, PyTorch constructs dynamic computation graphs, which means that during each inference, the computation graph is defined, utilized, and freed. Thus, constructing the entire graph is not necessary for execution. The advantage of such dynamic graphs is the efficient use of memory in cases where the input may vary. On the other hand, the disadvantage of dynamic graphs is the limited opportunities for global optimizations because the entire graph is not known during the execution.

(7) TensorRT: NVidia TensorRT [32] is built on CUDA and includes several optimizations to deliver high throughputs and low latencies for DNN applications. The focus of TenorRT is end-user performance. Therefore, TensorRT can optimize models that are trained on other frameworks after they are imported. Besides mixed-precision computation and quantization in INT8 and FP16, TensorRT tries to minimize the memory footprint by reusing memory and fusion operations. Although all other major frameworks support some part of these optimizations, we find TensorRT to be more user-friendly and easier to deploy.

(8) DarkNet: DarkNet [63] is a standalone open-source framework written in C and CUDA. The project code base is small and thus is understandable and modifiable. Since DarkNet is written in C without using high-level languages, it is a good candidate for low-level hardware and software optimizations and creating micro-benchmarks.

(9) FPGA Frameworks: To utilize an FPGA-based edge device, PYNQ board [64], we use a set of frameworks, FINN [65] and TVM VTA [66], [67] stack. FINN uses binarized weights for its implementations. TVM uses a customized RISC-based instruction set but on tensor registers. VTA deploys a custom-hardware design on the FPGA by utilizing a PYNQ overlay [68], which enables the offloading of predefined functions to the FPGA. Then, by utilizing the TVM just-in-time compiler, a user can offload the computations of a DNN model to the FPGA without interacting with any hardware-level code.

TABLE II
THE SPECIFICATIONS OF FRAMEWORKS USED IN THIS PAPER.

		TensorFlow	TFLite	Caffe1/2	Movidius	PyTorch	TensorRT	DarkNet
Language†		Python						C
Industry Backed		✓						✗
Training Framework		✓	✗	✓				
Usability		***	*	**	*	***	**	**
Adding New Models		**	*	***	*	***	**	***
Pre-Defined Models		***	*	**	*	***	**	**
Documentation		**	*	*	*	***	*	*
No Extra Steps		✓	✗	✓	✗	✓	✓	✓
Mobile Device Deployment		✗	✓		✗			
Low-Level Modifications		**	*	**	*	*	*	***
Compatibility with Others		*	*	*	*	*	**	*
Optimizations	Quantization	✓	✓	✓	✓	✓	✓	✗
	Mixed-Precision‡	✗	✗	✗	✗	✗	✓	✗
	Dynamic Graph	✗§	✗§	✗	✗	✓	✓	✗
	Pruning‡‡	✓††	✓	✗	✗	✗	✓	✗
	Fusion	✓††	✓	✗	✓	✗	✓	✗
	Auto Tuning	✗	✗	✗	✗	✗	✓	✗
	Half-Precision	✓	✓	✓	✓	✓	✓	✗

† Main interfacing language. *,**,*** More stars represents higher/easier/better.
‡ Support for mixed-precision inferencing, and not training.
§ TensorFlow Eager execution [55] provides dynamic graphs, but for debugging purposes. After the acceptance of this paper, a beta version of TensorFlow 2.0 [54] has been announced that supports dynamic graphs. ‡‡ Every framework supports pruning by zeroing out weights. Here, we show if a framework can automatically benefit from such fragmented weights. †† Experimental implementation.

TABLE III

THE SPECIFICATIONS OF HARDWARE PLATFORMS USED IN THIS PAPER.

Category	**IoT/Edge Devices**	**GPU-Based Edge Devices**		**Custom-ASIC Edge Accelerators**		**FPGA Based**	**HPC Platforms**			
							CPU	**GPU**		
Platform	Raspberry Pi 3B [34]*	Jetson TX2 [69]	Jetson Nano [36]	EdgeTPU [35]	Movidius NCS [37]$^\blacklozenge$	PYNQ-Z1 [64]	Xeon	RTX 2080	GTX Titan X	Titan Xp
CPU	4-core Ctx.A53 @1.2 GHz*	4-core Ctx.A57 2-core Denver2 @2 GHz	4-core Ctx.A57 @1.43 GHz	4-core Ctx.A53 & Ctx.-M4 @1.5 GHz	N/Ap	4-core Ctx.A9 @650 MHz	2x 22-core E5-2696 v4 @2.20GHz	N/Ap*	N/Ap	N/Ap
GPU	No GPGPU	256-core Pascal μA	128-core Maxwell μA	N/Ap	N/Ap	N/Ap	N/Ap	2944-core Turing μA	3072-core Maxwell μA	3840-core Pascal μA
Accelerator	N/Ap	N/Ap	N/Ap	EdgeTPU	Myriad 2 VPU	ZYNQ XC7Z020	N/Ap	N/Ap	N/Ap	N/Ap
Memory†	1 GB LPDDR2	8 GB LPDDR4	4 GB LPDDR4	N/Av*	N/Av	630 KB BRAM 512 MB DDR3	264 GB DDR4	8 GB GDDR6	12 GB GDDR5	12 GB GDDR5X
Idle Power‡	1.33	1.90	1.25	3.24	0.36	2.65	≈70	≈39	≈15	≈55
Average Power‡	2.73	9.65	4.58	4.14	1.52	5.24	300 TDP	≈	≈100	≈
Platform	All	All	All	TFLite	NCSDK	TVM/FINN	All	All	All	All

† Effective memory size used for acceleration/execution of DNNs, e.g., GPU/CPU/Accelerator memory size. * Ctx.: Arm Cortex. N/Ap: Not applicable. N/Av: Not available.
‡ : Measured idle and average power while executing DNNs, in Watts. * : Raspberry Pi 4B [70], with 4-core Ctx.A72 and maximum of 4 GB LPDDR4, was released after this paper acceptance. With better memory technology and out-of-order execution, Raspberry Pi 4B is expected to perform better. $^\blacklozenge$ Intel Neural Compute Stick 2 [61] with a new VPU chip and support for several frameworks was announced during paper submission, but the product was not released.

B. Framework Comparisons and Optimizations

This section compares the preceding frameworks from various points of view. In addition to some general aspects such as their programming language, development support from industry, training capability, usability, supported computations for models, and available documentations, Table II summarizes some relative technical advantages and disadvantages of the aforementioned frameworks[3]. For instance, TFLite and Movidius (i.e., NCSDK) require extra steps to configure a model for execution. As a result of such steps, both frameworks provide more optimizations for edge devices. For example, quantization-aware training and freezing a graph in TFLite considerably reduce the memory footprint of the DNN models implemented by TensorFlow. TensorRT also provides the same set of optimizations with less programming effort (and thus less aggressive benefits) with auto-tuning. On the other hand, the *full support* for mobile-device deployment is available only on TFLite, and *partially* on other frameworks such as Caffe2.

Low-level modifications, such as changing data ordering, optimizing for cache hits, and performing kernel fusion, are cumbersome tasks. For all DNN frameworks, such modifications are even harder because of their implementation in a high-level language (except DarkNet, which is written in C). Another aspect is the compatibility of frameworks with each other (e.g., the same model description format as input and similar output format). For this paper, we work with several frameworks. Unfortunately, we find limited compatibility among frameworks. Additionally, each framework assumes a set of hyperparameters that might be different in other frameworks. TensorRT provides better compatibly in importing models from other frameworks (including ONNX format); however, this is mainly because TensorRT is an inference-oriented framework, the main purpose of which is efficient inferencing. Similarly, TFLite is an inference-oriented framework backed by TensorFlow training procedures and its model description.

[3] Please note that these frameworks are actively being developed. As a result of that, although the technical statements about each framework might change, the discussions about the mentioned optimizations still hold true.

We also study the implemented state-of-the-art optimizations in each framework in their code base. Particularly, we study weight quantization, mixed-precision inferencing, dynamic construction/deconstruction/allocation of computation graph, ability to leverage pruned wights, kernel fusion for efficient caching and use of computing units, auto-tuning to hardware platforms, and support for half-precision numbers (FP16). Each optimization can be implemented in various methods with numerous tradeoffs. Additionally, several of such optimizations can be implemented by users in their application. In Table II, we address only the existence of such optimizations that are officially implemented by the authors of the each framework. Quantization to common datatypes is implemented for all frameworks that are supported by the industry. Quantization is popular because, as several studies show [18], [27], [28], using smaller datatypes can significantly reduce pressure on memory and storage in all systems. Furthermore, implementing quantization to a fixed data type is a fairly easy task for industry. On the other hand, mixed-precision inferencing requires several software-hardware co-designs, and the benefit is limited to specific hardware platforms (e.g., Nvidia Turing μArchitecture [71]) that can exploit such an imbalance in their computational units.

The dynamic manipulation of computation graphs enables efficient memory reuse and minimizes memory footprint. Therefore, increasing the performance of inference. TensorRT supports dynamic graphs, and PyTorch supports them by design choice. The static computational graph could also be beneficial, for example, by enabling offline optimizations (e.g.,TFLite). Thus, both options are viable choices for constructing a computational graph. Another optimization, weight

TABLE IV
THE SUMMARY OF EXPERIMENTS DONE IN THIS PAPER.

Experiments	**Execution Time**	**Framework Analysis)**						**Edge vs. HPC**		**Virtualization Overhead**	**Energy Measurments**		**Temperature**
Section/Figure	VI-A/2	VI-B/3	VI-B/4	VI-B/6	VI-B/7	VI-B/8	VI-B/5	VI-C/9	VI-C/10	VI-D/13	VI-E/11	VI-E/12	VI-F/14
Metric	Inference Time (ms or s)						Latency Breakdown	Inference Time (ms)	Speedup Over TX2	Inference Time (s)	Energy per Inference (mJ)	Inf. Time (ms) vs. Power (w)	Temp-erature (°C)
FW/Devices	RPi/TFLite,TF Nano/T-RT TX2/PT EdgeTPU/TFLite Mavidus/NCSDK PYNQ/TVM	RPi/DarkNet RPi/Caffe RPi/TF RPi/PT	TX2/DarkNet TX2/Caffe TX2/TF TX2/PT	GTX/TF GTX/PT	Nano/T-RT Nano/PT	RPi/TF RPi/T-Lite	RPi/PT RPi/TF TX2/PT TX2/TF	TX2/PT Xeon/PT GTX/PT T-XP/PT 2080/PT	TX2/PT Xeon/PT GTX/PT T-XP/PT 2080/PT	Bare Metal RPi/TF Docker RPi/TF	RPi/TFLite Nano/T-RT TX2/PT EdgeTPU/T-Lite Mavidus/NCSDK GTX/PT	RPi/TFLite Nano/T-RT TX2/PT EdgeTPU/T-Lite Mavidus/NCSDK GTX/PT	RPi/TFLite Nano/T-RT TX2/PT EdgeTPU/T-Lite Mavidus/NCSDK GTX/PT

FW: Framework, **TX2**: Jetson TX2, **Nano**: Jetson Nano, **PT**: PyTorch, **TF**: TensorFlow, **TFLite**: TensorFlow Lite, **T-RT**: Tensor RT, **GTX**: GTX Titan X, **T-XP**: Titan Xp, **2080**: RTX 2080

pruning [18], [23]–[25], [72], [73], reduces the storage and computation footprints by removing some of the network connections. However, a framework needs to take further steps to exploit these benefits (e.g., change data representation format to match underlying hardware [73]). This is because data and computations are now in sparse format. Although all frameworks benefit from pruning in reducing their storage footprint, only TensorFlow, TFLite, and TensorRT take further steps in exploiting it for sparse computations.

Another essential optimization is kernel fusions [74], which reduces memory traffic by reusing data that are already populated and eliminating redundant loads and stores. The effectiveness of kernel fusions depends on several factors, such as hardware resources and applied software techniques. Therefore, kernel fusions are mostly supported on platform-specific frameworks (e.g., TFLite, Movidius SDK, and TensorRT). Usually, each framework requires careful tuning to particular hardware by an experienced engineer. After tuning, the models and weights are released for use by users. However, in this approach, generally, users cannot create new efficient models since they have limited knowledge of both hardware and framework. Auto-tuning support in TensorRT tries to solve this issue so that users create *relatively* tuned models. Finally, inferencing using half-precision floating numbers (FP16) is supported by almost all frameworks, similar to quantization.

IV. Edge Devices

This section provides an overview of our hardware platforms, including edge devices and edge accelerators, as well as FPGA-based and CPU/GPU high-performance computing (HPC) platforms. Table III summarizes their internal organization and their measured idle and average power usage.

(1) Raspberry Pi 3B: Raspberry Pi 3B [34] is a small and affordable single-board computer. On a Broadcom SoC, Raspberry Pi has a quad-core ARM Cortex-A53 processor clocking at 1.2 GHz connected to a 1 GB LPDDR2 RAM. Raspberry Pi has no GPGPU capability and no specialized hardware accelerator. Raspberry Pi and Arduino [75], with similar specifications, are good representations of IoT/Edge devices and are the main platforms for portable robots (See Table III footnotes about next generation of Raspberry Pi).

(2) Jetson TX2: Jetson TX2 [69] is a high-performance embedded platform with a 256-core Pascal architecture GPU. Jetson TX2 has two sets of CPUs, quad-core ARM Cortex-A57 processor at 2.0 GHz and dual-core Nvidia Denver 2 at 2.0 GHz. The total available memory is 8 GB LPDDR4 with a 128-bit interface. The memory is hard-wired to the memory controller and is shared between the ARM processor and GPU. Thus, the memory transfer speed of the GPU is not limited by conventional PCIe bus speeds. Jetson TX2 does not utilize hardware accelerators.

(3) Jetson Nano: Nvidia Jetson Nano [36] board is a smaller version of Jetson TX2 for edge applications. Jetson Nano is a GPU-based single-board computer with a 128-core Maxwell architecture GPU and a quad-core ARM Cortex-A57 clocking at 1.43 GHz. The total available memory is 4 GB LPDDR4 with a 64-bit interface. Similar to Jetson TX2, Jetson Nano's memory is shared between the processor and GPU.

(4) EdgeTPU: Edge TPU [35] is a small ASIC designed by Google that provides high-performance DNN acceleration for low-power and edge devices. The ASIC design is hosted on a single-board computer, similar to Raspberry Pi. With an NXP SoC, the host computer has a quad-core ARM Cortex-A53 processor and one Cortex-M4 processor. The available memory for the processors is 1 GB LPDDR4. The design of EdgeTPU and its specifications have not been released.

(5) Movidius Neural Compute Stick: Intel's Movidius Neural Compute Stick (NCS) [37] is a plug-and-play USB device that connects to any platform for accelerating DNNs. It is based on the Intel Movidius Myriad 2 vision processing unit (VPU), which has 12 Streaming Hybrid Architecture Vector Engine (SHAVE) Cores. The internal core architecture of SHAVE is VLIW, with single instruction multiple data (SIMD) functional units. VPU natively supports mixed precisions, 32-, 16-, and some 8-bit datatypes. NCS was among the first special hardware designs for edge devices. (See Table III footnotes about next NCS).

(6) PYNQ: PYNQ [64] board is an open-source endeavor from Xilinx that provides SoCs integrated with FPGA fabric (Zynq series). PYNQ is designed for embedded applications. We use the PYQN-Z1 board that has a dual-core Cortex-A9 processor at 650 MHz and a 512 MB DDR3 memory with a 16-bit interface. The FPGA is from the Artix-7 family with 13,300 logic slices, 220 DSP slices, and 630 KB BRAM.

(7) HPC Platforms: As a point of comparison with the lightweight edge devices and accelerators, Table III also lists the characteristics and power usage of a Xeon CPU and three GPUs that are known platforms for running DNN computations in the cloud and on servers.

V. Experimental Setups

Execution Time: For each experiment, the execution time is measured by running several single-batch inferences in a loop.

To accurately measure time per inference as an end user, we do not include any initialization time (e.g., library loading, live input setup, and model weight loading) because this is a one-time cost that occurs during device setup. For frameworks that permit us to bypass initialization time in their code, we time their inferences by only accounting for inference time. For other frameworks, we run single-batch inferences several times (200–1000) to reduce the impact of initialization.

Power Measurements: For devices, the power of which is supplied through USB ports, we measure power using a USB digital multimeter [76] that records voltage and current every second. The accuracy of the multimeter for voltage and current are $\pm(0.05\% + 2\text{digits})$ and $\pm(0.1\% + 4\text{digits})$, respectively. For devices, the power of which is supplied with an outlet, we use a power analyzer, with an accuracy of ± 0.005 W.

Thermal Measurements: For thermal evaluations, each experiment runs until the temperature reaches steady-state in the room temperature. We measure the processor surface temperature of each device using a thermal camera, Flir One [77]. For devices with heatsink (see Table VI), the measured temperature is the surface temperature of the heatsink. Since the thermal resistance of a transistor chip is smaller than that of the heatsink, the temperature of the heatsink surface is 5–10 degrees Celsius lower than that of the in-package junction [78].

VI. Characterization

This section studies the characteristics of edge devices and frameworks for DNN inference. Table IV summarizes all experiments and refers to the related sub-sections and figures.

A. Execution Time Analysis

To choose a well-performing framework for each device (see Section VI-B for cross-framework analysis), this section evaluates them by measuring the inference time of several models. Before analyzing the results, Table V summarizes the compatibility of models and platforms. For instance, on RPi, we deployed all models, but larger models required a dynamic computation graph because of the small memory size of RPi. Such models experience an order of magnitude higher inference time, marked with ◇ (i.e., AlexNet, VGG16, and C3D). In these scenarios, PyTorch uses its dynamic graph to manage limited memory availability, whereas TensorFlow fails to run such models. For smaller models, however, TensorFlow achieves better performance than PyTorch.

TABLE V
Models and platforms compatibility matrix.

Model \ Platform	RPi3	Jetson TX2	Jetson Nano	EdgeTPU	Movidius	PYNQ
ResNet-18	✓	✓	✓	△	✓	✓
ResNet-50	✓	✓	✓	✓	✓	◇◇
MobileNet-v2	✓	✓	✓	✓	✓	◇◇
Inception-v4	✓	✓	✓	✓	✓	◇◇
AlexNet	◇	✓	✓	△	✓	◇◇
VGG16	◇	✓	✓	✓	✓	◇◇
SSD MobileNet-v1	▽	✓	✓	✓	✓	◇◇
TinyYolo	✓	✓	✓	△	✓	◇◇
C3D	◇	✓	✓	△	✓	◇◇

◇ Large memory usage, uses dynamic graph.
▽ Code incompatibility. ◇◇ Large BRAM usage. Requires accessing host DDR3, considerably slowdowns execution.
△ Barriers in converting models to TFLite. Check §VI-A.

Moreover, as Table V lists, we were unable to run a few models on some platforms. As an example, for the SSD model on RPi and C3D on Movidius, marked with ▽, we face several code incompatibility issues in the base code implementation – SSD uses an extra image processing library on top of DNN frameworks. On EdgeTPU, several reasons prevented us from converting models to TFLite, marked with △. This is because (i) only the quantized model is supported by the edgeTPU compiler; (ii) in some cases, the quantized model can only be obtained by a quantization-aware training, not post-training quantization; (iii) quantization-aware training or freezing the graph in TensorFlow does not necessarily create compatible quantized models for the EdgeTPU compilation tool; and (iv) obtaining the compatible TFLite model is possible with careful fine-tuning, but we were unable to find such parameters[4]. For PYQN board experiments, although the frameworks we use (FINN and TVM) have implemented small models (CifarNet, and ResNet-18), we face challenges in extending their framework to larger models. This is because of limited resources on the FPGA for larger models (for TVM VTA) and retraining requirements (for FINN). Additionally, not every model currently complies to VTA compatible code, since some parameters of the model must match the hardware specification [79]. Besides, large models require runtime data transfer between the host memory and the FPGA, which causes severe slowdowns.[5]

Figure 2 illustrates time per inference in milliseconds (ms) for several models on edge devices. Time per inference per model varies widely across devices (Figure 2). In most cases, either GPU-based devices or EdgeTPU provides the best performance. However, as shown in Table V, EdgeTPU faces several barriers in compiling new models to TFLite. Although this is an engineering problem, it limits EdgeTPU end-user usability, especially for new models. On the other hand, the models in Jetson Nano/Jetson TX work out of the shelf with automatic tuning. For Jetson Nano, TensorRT yields the best performance numbers by applying several optimizations (see Table II), whereas, for Jetson TX2, most of the results are with PyTorch with no optimization. As shown, the Jetson TX2

[4]Although it is a common practice to use randomized weights for the performance evaluation of a model, such evaluations are not close to reality. This is because TFLite uses quantized weights. So, the final performance is directly proportional to the sparsity of weights and their sparsity pattern. So, although using randomized weights is a proxy/shortcut to the final performance number, it would not be realistic data.

[5]After the paper acceptance, based on reviewers' comments, we tried to add some data points for additional small models for the PYNQ board. However, first, we could not find a suitable frontend that could compile the models without tight hardware/software constraints. Regularly, we had to change a model to be able to execute it on the board, which required training. Second, even after changing the model, we found that a non-optimized hardware implementation could be slower than its CPU-based implementations. Therefore, for optimizing the hardware implementation, one must profile several implementations to tune hardware parameters, a time-consuming task. Nevertheless, we believe the endeavors to bridge the gap between software and hardware for DNNs are instrumental.

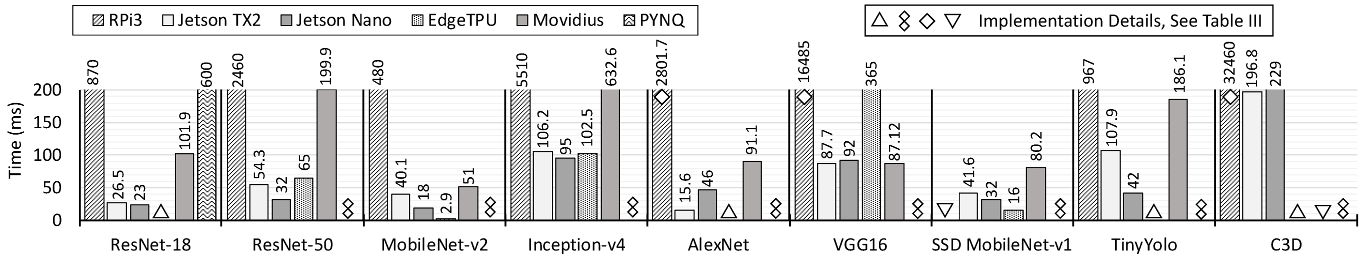

Fig. 2. Time per inference on all the edge devices with best performing framework. See V for implementation details.

results are as close as possible to the best inference time. The Movidius Stick performance results in some cases are close to the best case (i.e., MobileNet-v2 and C3D), but in others, they are much higher (i.e., ResNet-50 and Inception-v4) compared to the best cases. This is because (i) Movidius models require careful fine-tuning by experts, which in the case of new models has not been fully done; and (ii) the design of Movidius Stick is older than that of other devices. In fact, at the time of writing this paper, Intel released news about the second generation of these sticks, claiming an 8x speedup [80].

B. Frameworks Analysis

In this subsection, we aim to study how the choice of DNN frameworks affects execution performance. Section III-B provided a high-level overview of each framework implementation. This section first presents a framework comparison and then analyzes the benefit of using edge-specific frameworks (i.e., TFLite and TensorRT), and finally analyzes the software stack of two popular frameworks (i.e., TensroFlow and PyTorch) on edge GPU and CPU platforms.

1) Frameworks Comparisons: Figures 3 and 4 depict cross-platform time per inference on RPi and Jetson TX2, respectively. Since DarkNet is not industry-backed, we were not able to find/implement some complex models. The results on RPi show that TensorFlow is the fastest among the frameworks (we compare edge-specific frameworks, TFLite and TensorRT, in Section VI-B2). For instance, MobileNet-v2 on TensorFlow achieves 1.40 seconds per inference, while Caffe and PyTorch achieve 2.27 and 8.25 seconds per inference, respectively. However, as discussed, PyTorch can execute large models, such as VGG16, that TensorFlow cannot run because of limited memory errors. On our GPU platform, Jetson TX2, PyTorch performs faster than TensorFlow. As we discuss in Section VI-B3, in TensorFlow, the overhead of using a static computation graph on GPU exceeds its performance gains. Interestingly, the performance of Caffe is always better than that of TensorFlow, except for MobileNet-v2 (Figure 4). Including the fact that Caffe was released in late 2013 and not updated actively after 2017, the performance of TensorFlow is significantly low on small GPUs. In fact, we performed another experiment on an HPC GPU, GTX Titan X, and observed similar behavior for TensorFlow versus another framework, PyTorch, shown in Figure 6. Since all the frameworks use similar CUDA libraries in the backend (which differ in library versions due to compatibility issues), we believe the low performance of TensorFlow is not mostly caused by its implementation but by its hard usability. First, TensorFlow, due to its huge codebase, has a mix of several APIs without good documentation on their differences. The introduction of new APIs with new parameters with every update also confuse users on the best API to use. Second, several optimization flags, such as fusing operations, are hidden or not easily accessible. On the other hand TensorFlow is highly customizable and supports several type of models.

2) Edge-Specific Frameworks: Edge-specific frameworks (Section III-B) heavily optimize DNN inference on edge devices. To understand the benefit of using these frameworks, Figures 7 and 8 compare the execution time of single-batch inferences with several models on Jetson Nano (with PyTorch and TensroRT) and RPi (with PyTorch, TensorFlow, and TFLite), respectively. Seeking a fair comparison, we use the same hardware platform, whose special capabilities can be utilized by all the target frameworks. To this end, we select RPi over EdgeTPU becasue the accelerator on EdgeTPU is only accessible by TFLite and not TensorFlow. Figure 7 shows an average of 4.1x speedup using TensorRT on Jetson Nano compared to PyTorch. TensorRT has several optimizations, such as mixed- and low-precision (INT8) inferencing, that PyTorch does not support. Additionally, fusion and auto tunings enable TensorRT to efficiently use the underlying hardware. Generally, models with large memory footprints (AlexNet and VGG16) and large inputs (C3D and TinyYolo) achieve smaller

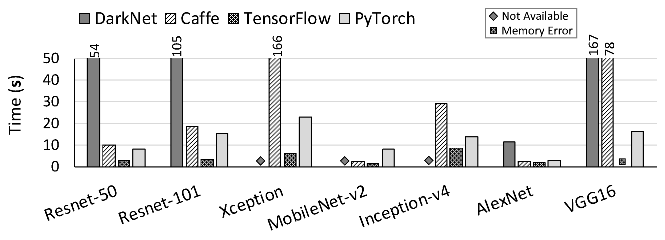

Fig. 3. Time per inference on RPi across different frameworks.

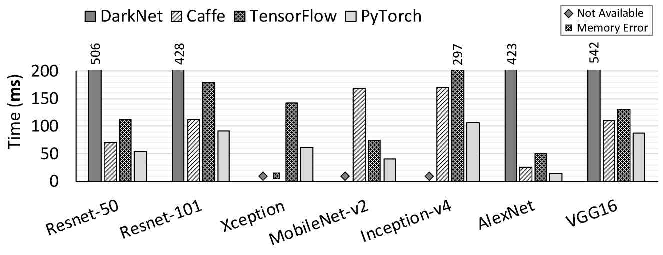

Fig. 4. Time per inference on Jetson TX2 across different frameworks.

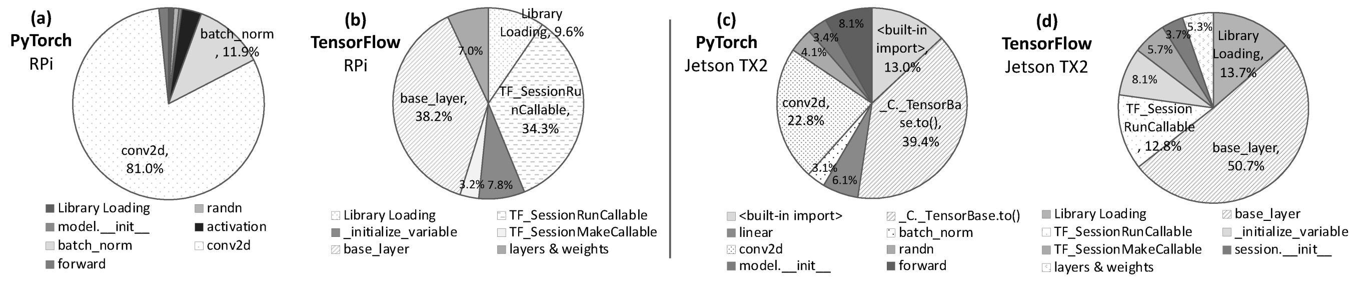

Fig. 5. Profiling of two popular frameworks software stacks on Raspberry Pi and Jetson TX2.

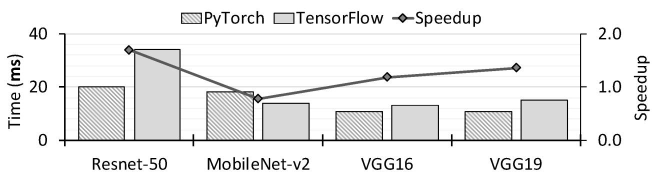

Fig. 6. Time per inference on GTX Titan X (TensorFlow and PyTorch).

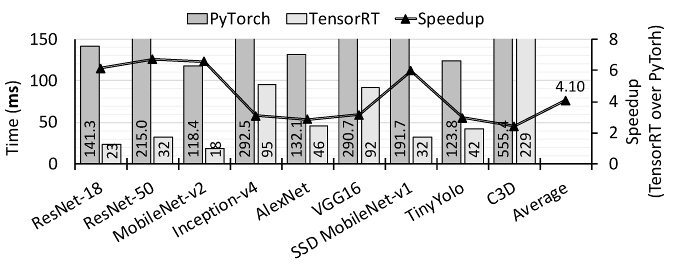

Fig. 7. Time per inference on Jetson Nano with PyTorch and TensorRT.

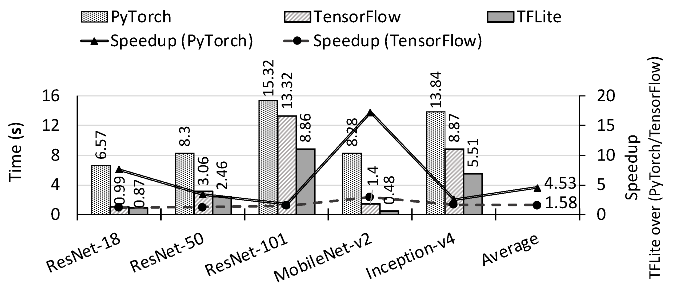

Fig. 8. Time per inference on RPi with TensorFlow, PyTorch, and TFLite.

speedups compared to other models. For TFLite, Figure 8 depicts an average speedup of 1.58x on RPi with TensorFlow and a 4.53x speedup with PyTorch. Although TFLite supports low-precision inferencing, the RPi hardware does not support it. In fact, TFLite implements several optimizations, but the optimizations are not fine-tuned to RPi. For clarification, the achieved gain for TFLite is smaller than that for TenorRT since TensorFlow already does several optimizations on its static graph (compare PyTorch performance with TensorFlow). In summary, although edge-specific frameworks/models require extra preparations, their achieved performance is better compared with non-optimized implementations.

3) Software-Stack Analysis: To better understand the internal behavior of frameworks, we profile the duration of low-level functions by using the built-in profiler of Python (cProfile [81]) for combinations of two frameworks and two platforms. To understand major functions/tasks, from the profiling results, we grouped functions with similar tasks. We first profile PyTorch and TensorFlow frameworks on RPi, with relatively low computing capability (Table III) by running 30 inferences. Figures 5a and b show that RPi spends most of its time on low-level arithmetic primitives. More specifically, PyTorch spends 96.15% on compute-related functions (i.e., `conv2d`, `batch_norm`, and `activation`), while Tensorflow spends 44.84% of its total time within a computing session (i.e., `TF.RunCallableSession`). Among the compute-related functions in PyTorch, the `conv2d` low-level primitive accounts for 80.95% of the entire program runtime. The dynamic graph implementation of PyTorch (see Section IIIA) helps accomplish faster graph setup, which results in negligible graph setup time (Figure 5a). On the other hand, the graph construction time in TensorFlow (i.e., `base_layer` function) accounts for 38.22% of the total time. This is a one-time cost for all inferences on TensorFlow (we could not run as many inferences with profiler to amortize this cost further as with our other experiments). Since the static graph implementation is easier to optimize before the actual computation, the overall runtime of TensorFlow on RPi (Figure 3) is less than that of PyTorch on the same device.

We repeat the same profiling of PyTorch and Tensorflow on Jetson TX2 by profiling 1000 inferences. Compared to RPi, Jetson TX2 is equipped with a relatively high-performance CPU and an easily-accessible GPU. All of our frameworks on TX2 are able to utilize the GPU to speed up the DNN computations. Our results show that the computation graph setup and the actual computation profiles on TX2 differ from those on RPi, mainly because adding a GPU to the system significantly drops the time spent on the actual computation on both frameworks. As a result, as Figures 5c and d show, PyTorch and TensorFlow spend a notable portion of the total time on computation graph setup in the GPU version (i.e., `_C._Tensorbase.to()`, `model.__init__` in PyTorch; and `base_layer` in TensorFlow). Although TensorFlow substantially outperforms PyTorch on RPi, PyTorch is faster on platforms with a GPU. The reason could be that the overhead of using a static computation graph exceeds the performance gained from its optimizations for TensroFlow on the GPU.

C. Edge Versus HPC Platforms

HPC platforms are known to be the best options for performing DNN inferencing. This is because HPC platforms, such as HPC GPUs, are designed to exploit massive data parallelism available at data centers, where large companies batch sev-

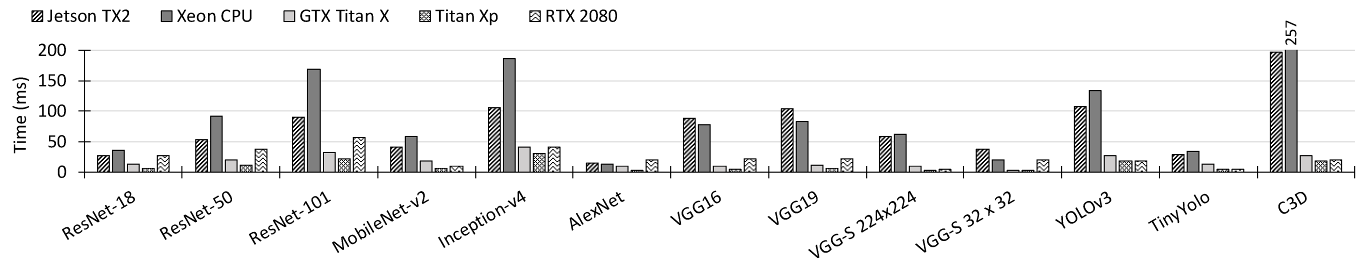

Fig. 9. Time per inference comparison between edge and HPC platforms with PyTorch framework.

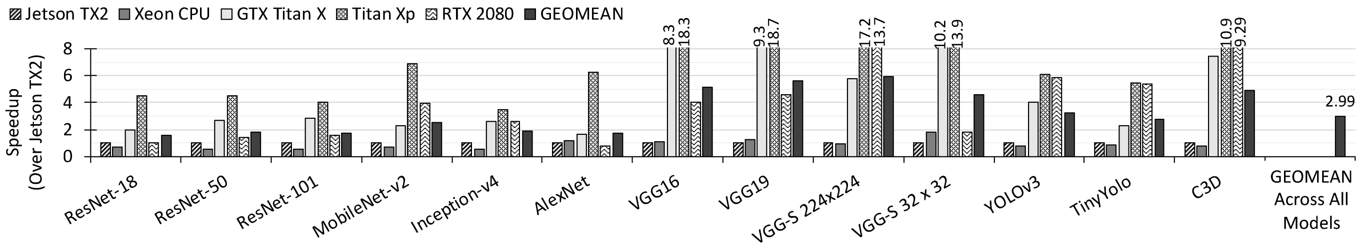

Fig. 10. Speedup (over Jetson TX2) of time per inference between edge and HPC platforms with PyTorch. For hardware specifications, see Table III.

eral requests together and perform multi-batch computations. Multi-batch inferencing helps to amortize the cost of data movement and eliminates redundant load and stores. On the other hand, for edge devices, the number of requests is limited and real-time performance is crucial. Thus, special edge devices are designed for efficient single-batch inferencing. Here, we want to see if these edge-specific designs are efficient with respect to HPC platforms in single-batch inferencing.

To make such a comparison between edge and HPC devices for single-batch inferencing, we use a common framework (i.e., PyTorch) for deployment. Then, we measure time per inference on several HPC platforms and Jetson TX2, shown in Figure 9. To make a fair comparison of single-batch inferencing across the platforms, we do not use any edge-specific techniques to increase the performance. We choose Jetson TX2 as our edge device because its hardware is not heavily optimized (see Figure 2). As we see in Figures 9 and 10, the average speedup over Jetson TX2 on all benchmarks is only 3x. Most of these platforms are designed to be throughput-oriented for multi-batch DNN computations (both training and inferencing). However, single-batch inferencing is a latency-sensitive task, which requires a new design philosophy, both in hardware and frameworks. Although CPUs are known to be designed for latency-sensitive tasks, our experiments show that CPUs are not beneficial for single-batch inferencing.

More specifically, on several benchmarks, the Xeon CPU performance is lower than that of all platforms. This is because most benchmarks are compute-bounded and benefit from more available cores. In fact, only for memory-bounded benchmarks (e.g., VGG16 and VGG19), does Xeon CPU perform similarly to TX2 because of its large memory hierarchy. On HPC GPUs, the benchmarks with large memory footprint such as VGG models and C3D generally achieve higher speedups. This is because HPC GPUs have larger memories and caches. On the other hand, benchmarks with higher compute per memory such as ResNet models benefit less from HPC GPUs. In summary, single-batch inferencing requires a different hardware design perspective, which specially designed edge devices and frameworks aim to reach.

D. Virtualization Overhead Study

With a diverse set of hardware platforms and frameworks, virtualization could provide several benefits by decoupling hardware/software setup and reducing the programmer's effort. Nevertheless, the virtualization environment should support auto-tuning to each specific hardware platform to maximize performance (e.g., using INT8 on architectures that supports it). Endeavors to design such virtualization tools are underway, but virtualization itself has overhead. This overhead is caused by several translations for system calls and environment isolations. In this section, we evaluate the overhead of virtualized environments by executing DNN models inside and outside such an environment. We use Docker [82], a widely used virtualization tool in both academia and industry. Figure 13 shows the results of executing DNNs on RPi with/without Docker. As seen, the overhead is almost negligible, within 5%, in all cases. Contrary to popular belief about virtualization overhead, we do not observe a significant slowdown with virtualization.

E. Energy Measurements

Inferencing in the edge dictates processing one input most efficiently. Specifically, for edge inferencing, an efficient device is fast and power efficient. Thus, we measure the energy per

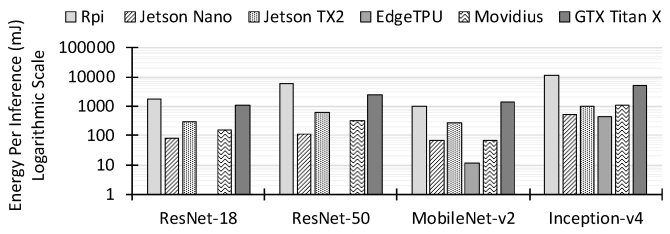

Fig. 11. Energy per inference across platforms.

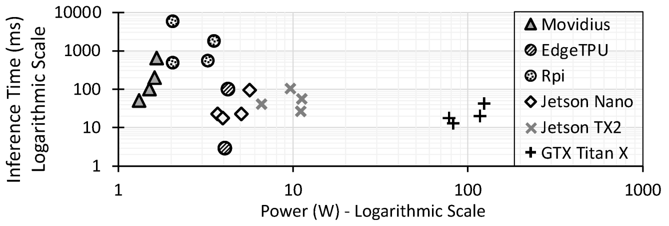

Fig. 12. Inference time and active power usage graph across platforms.

inference across all our platforms and include one HPC GPU as a comparison point with high-end devices. Figure 11 shows energy per inference for our platforms with four models. As expected, RPi has the highest energy per inference value because (i) it is designed as a cheap and general-purpose single-board computer without accounting for energy efficiency, and (ii) it has the longest inferencing time among our devices. After RPi, GTX Titan X has the highest energy per inference, between 1 J to 5 J per inference for ResNet-18 and Inception-v4, respectively. Although Jetson TX2 is a GPU-based design, its energy consumption is lower, between 0.3 J to 1 J per inference for ResNet-18 and Inception-v4, respectively. This is an average of a 5x energy savings with respect to GTX Titan X. Nevertheless, edge-specific devices lower the energy consumption to as low as 11 mJ per inference (MobileNet-v2 on EdgeTPU). Jetson Nano consumes 84 mJ to 0.5 J energy per inference for ResNet-18 and Inception-v4, respectively. Movidius Stick also has a similar profile, which is between 66 mJ to 1 J for MobileNet-v2 and Inception-v4, respectively.

For a better perspective, we also compare platforms within inference time versus active power graph. Figure 12 shows such a graph, in which the left corner illustrates the most energy efficient and fastest device. Each dot represents a model, and dots are grouped based on their platform. As seen, GTX Titan X resides far in the left side of the graph, with an average of 100 W active power usage. Several platforms have similar inference time but much lower active power: Jetson TX2, Jetson Nano, Edge TPU, and Movidius Stick. In fact, Movidius Stick is the platform with the lowest active power usage. On the other hand, EdgeTPU is the platform with the lowest inference time. However, both devices make a tradeoff to be at such extremes. Jetson Nano resides in the middle by balancing inference time and power usage.

F. Temperature Measurements

This section evaluates the correlation between temperature and power usage when running the inference of a heavy DNN (i.e., Inception-v4) on various edge devices using the most efficient

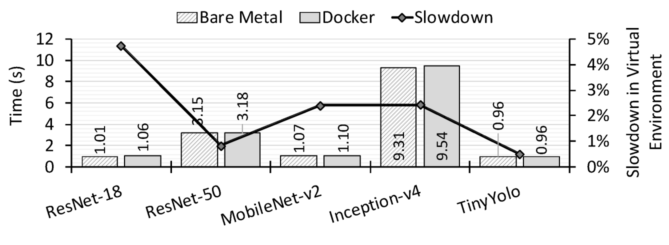

Fig. 13. Time per inference on Bare Metal RPi and Docker-based RPi.

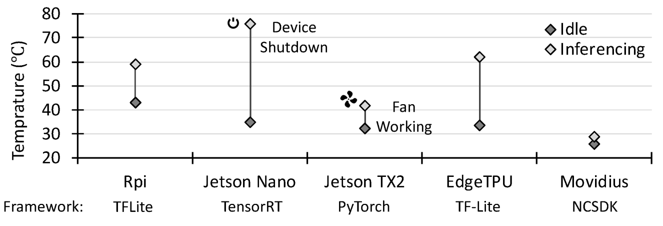

Fig. 14. Temperature behavior of edge devices while executing DNNs.

TABLE VI
Device specifications for temperature experiments.

Device	Heatsink	Cooling Fan	Idle Temperature	Fan Activated?
Raspberry Pi	✗ 14x14 mm	✗	43.3 °C	✗
Jetson TX2	✓ 80x55x20 mm	✓	32.4 °C	✓
Jetson Nano	✓ 59x39x17 mm	✗	35.2 °C	✗
Edge TPU	✓ 44x40x9 mm	✓	33.9 °C	✗
Movidius	✓† 60x27x14 mm	✗	25.8 °C	✗

† USB stick is designed as a heatsink.

framework for each device. Table VI lists the availability and characteristics of the cooling instruments (i.e., the heatsink and fan) for the edge devices and their idle temperatures. As Figure 14 illustrates, the temperature variation of Movidus is the lowest even though it is not equipped with a fan. Although the temperature and the power usage of Movidus are the lowest among the peers, the trend is not always valid. For instance, the power usage of Jetson TX2 is higher than that of Jetson Nano, while their temperatures are opposite. In fact, the figure and table suggest that the temperature of Jetson TX2 is perhaps controlled by the activated fan, and so should be the temperature of Jetson Nano. Further, from Figure 12 and Figure 14 we infer that the temperatures of RPi and Edge TPU are similar; however, the power usage of the latter is approximately 5x lower as that of the former.

VII. Discussions

In this paper, we tried to analyze popular frameworks and implementation optimizations for DNN deployment on the edge devices. Since these frameworks are actively being improved, we tried not to include a tighter conclusion about each framework. Additionally, there are several other frameworks that we did get a chance to cover, so concluding about a framework would not be entirely fair. Moreover, as seen, there is no single best framework for all cases. Depending on the model, the computation type, and, in several cases, the final weight sparsity, each framework delivers different results. In fact, these tradeoffs, as any other generalized frameworks, occur because each framework tries to offer a simple interface to a wide range of users while optimizing execution performance. Therefore, our goal is to give the reader the knowledge to reach to their own conclusion about what is the best framework for their use case based on the provided data.

VIII. Related Work

Edge computing with advantages such as high performance, security, and scalability has introduced a new paradigm for processing DNNs. To enable the inference of DNNs at the edge, a group of studies has proposed techniques that trade accuracy (sometimes) with performance to create small models. Pruning [18], [23]–[25], [72], [73] is one example of common practices that remove the close-to-zero weights. Other examples of the first group of efforts are quantization and low-precision inference [26]–[29], [83], which modify the representations of numbers to achieve simpler calculations. The second group of studies develops mobile-specific models [40], [84]–[87]. They aim to handcraft efficient operations or models to reduce the number of parameters [84] to create efficient operation to minimize computation density [40], or use resource-efficient connections [87]. The third group of studies distributes the computations of models among available resource-constrained edge devices. For instance, Neurosurgeon [88] dynamically partitions a DNN model between a *single* edge device and the cloud. MoDNN [89] creates a local distributed mobile computing system and accelerates DNN computations with model parallelism. Hadidi et al. [11], [90]–[92] investigate the distribution of DNN models for single-batch inferences with model-parallelism methods, while deploying distributed systems in robots [14], [93], IoT devices [94], and FPGAs [79].

The importance of in-the-edge inference of DNNs that has been shown in the preceding work has encouraged researchers and industry to propose several lightweight frameworks and specialized edge devices. Selecting an appropriate combination of framework and hardware platform for a specific application requires characterization. The following are the few studies that have evaluated some of the edge frameworks and devices. In the first characterization study of edge devices [95], the inference time and energy consumption of five well-known CNN models are measured on NCS, Intel Joule 570X, and Raspberry Pi 3B, in which TensorFlow, Caffe, NCSDK, and OpenBlas are used as frameworks. The paper suggests that Raspberry Pi with Caffe requires the least amount of power.

Another study [96] evaluates the latency, memory footprint, and energy of using TensorFlow, Caffe2, MXNet, PyTorch, and TensorFlow Lite for the inference of two CNNs on MacBook Pro, Intel's Fog Reference Design, Jetson TX2, Raspberry Pi 3B+, and Huawei Nexus 6P (Nexus 6P). This research concludes that TensorFlow runs larger models faster than Caffe2 does, and vice versa for smaller models. Moreover, they suggest that PyTorch is more memory efficient than other frameworks. The latency and throughput of CNN inference are studied in another work [97], in which TensorFlow and TensorRT are frameworks to target Jetson TX2 as a mobile device in comparison with CPU and GPU platforms. Besides the preceding real-world characterization studies, SyNERGY [98] extends the performance evaluation of DNNs by proposing a fine-grained energy prediction framework on edge platforms. SyNERGY integrates ARM Streamline Performance Analyzer with the Caffe and CuDNNv5 frameworks to quantify the energy consumption of DNNs on the Nvidia Jetson TX1. MLModelScope [99], an ongoing endeavor, tries to explore the accuracy and performance of the models across frameworks, models, and systems.

In addition to the frameworks studied in this work and prior characterization papers, Apache MXNet [100] and TVM [66] used in Amazon Web Services, and Microsoft Cognitive Toolkit (CNTK) [101], utilized in Microsoft Azure, are other three open-sourced DNN frameworks backed by industry. MXNet is scalable to support distribution on the dynamic cloud infrastructure and is flexible to support imperative and symbolic programming. TVM implemented extensive compiler-level optimizations. CNTK, which implements DNNs as a series of computational steps via a directed graph, provides advantages such as combining well-known neural network types (e.g., feed-forward DNNs, CNNs, RNNs, and LSTMs). Our experiments can be extended to these frameworks by running our provided source code in our GitHub.

Finally, several recent academic efforts have proposed custom accelerators [73], [102], [102]–[111], which improve inference by utilizing sparsity, reducing memory accesses, or employing efficient dataflows. In addition, many of the custom designs have targeted FPGA/ASIC implementations for inference acceleration [79], [112]–[117].

IX. Conclusion

This paper investigated the in-the-edge inference of DNNs from the perspectives of execution time, energy consumption, and temperature. We hope that the following insights from our research lead users to knowingly choose their required package (i.e., a combination of framework and platform) for a specific edge application. Additionally, we observed that even though DNN computations are heavily dominated by matrix-matrix multiplications, custom hardware designs for edge devices matters in final performance. We sought to give a perspective into the performance of edge devices with our experiments that covered several technologies (hardware with different microarchitecture/circuit designs), software frameworks, and optimization techniques. We noted hardware and software co-designs and their support on edge devices to be one of the main reasons behind achieving low inference time. Our energy measurements showed a tradeoff between energy consumption and inference time on edge devices (e.g., Movidius vs. Jetson Nano). We believe that such a tradeoff could be utilized to design efficient and application-specific devices. Finally, we analyzed the crucial impact of DNN frameworks by studying their software stack, optimizations, and virtualization.

Acknowledgements

We thank anonymous reviewers for their constructive feedbacks for improving the paper. We are grateful to Younmin Bae for sharing his experience regarding FPGA-based implementations. This study was supported in part by National Science Foundation under grant number CSR-1815047.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *26th Annual Conference on Neural Information Processing Systems (NIPS)*. ACM, 2012, pp. 1097–1105.

[5] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations*. ACM, 2015.

[6] M. S. Ryoo, K. Kim, and H. J. Yang, "Extreme low resolution activity recognition with multi-siamese embedding learning," in *AAAI'18*. IEEE, Feb. 2018.

[7] K. Simonyan and A. Zisserman, "Two-stream convolutional networks for action recognition in videos," in *NIPS'14*. ACM, 2014, pp. 568–576.

[8] S. Zhang, L. Yao, A. Sun, and Y. Tay, "Deep learning based recommender system: A survey and new perspectives," *ACM Computing Surveys (CSUR)*, vol. 52, no. 1, p. 5, 2019.

[9] E. Ohn-Bar and M. M. Trivedi, "Looking at humans in the age of self-driving and highly automated vehicles," *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 1, pp. 90–104, 2016.

[10] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[11] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Distributed perception by collaborative robots," *IEEE Robotics and Automation Letters (RA-L), Invited to IEEE/RSJ International Conference on Intelligent Robots and Systems 2018 (IROS)*, vol. 3, no. 4, pp. 3709–3716, Oct 2018.

[12] A. Giusti, J. Guzzi, and n. . . p. . . p. . I. t. . A. v. . . y. . . Cireşan, Dan C and He, Fang-Lin and Rodríguez, Juan P and Fontana, Flavio and Faessler, Matthias and Forster, Christian and Schmidhuber, Jürgen and Di Caro, Gianni and others, journal = IEEE Robotics and Automation Letters.

[13] M. Pfeiffer, M. Schaeuble, J. Nieto, R. Siegwart, and C. Cadena, "From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots," in *2017 ieee international conference on robotics and automation (icra)*. IEEE, 2017, pp. 1527–1533.

[14] M. L. Merck, B. Wang, L. Liu, C. Jia, A. Siqueira, Q. Huang, A. Saraha, D. Lim, J. Cao, R. Hadidi *et al.*, "Characterizing the execution of deep neural networks on collaborative robots and edge devices," in *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*. ACM, 2019, p. 65.

[15] A. Singh, B. Ganapathysubramanian, A. K. Singh, and S. Sarkar, "Machine learning for high-throughput stress phenotyping in plants," *Trends in plant science*, vol. 21, no. 2, pp. 110–124, 2016.

[16] H. Lu, Y. Li, S. Mu, D. Wang, H. Kim, and S. Serikawa, "Motor anomaly detection for unmanned aerial vehicles using reinforcement learning," *IEEE internet of things journal*, vol. 5, no. 4, pp. 2315–2322, 2018.

[17] O. B. Sezer, E. Dogdu, and A. M. Ozbayoglu, "Context-aware computing, learning, and big data in internet of things: a survey," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 1–27, 2018.

[18] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *4th International Conference on Learning Representations*. ACM, 2016.

[19] S. Li, L. Da Xu, and S. Zhao, "The internet of things: a survey," *Information Systems Frontiers*, vol. 17, no. 2, pp. 243–259, 2015.

[20] F. Biscotti, J. Skorupa, R. Contu *et al.*, "The impact of the internet of things on data centers," *Gartner Research*, vol. 18, 2014.

[21] I. Lee and K. Lee, "The internet of things (iot): Applications, investments, and challenges for enterprises," *Business Horizons*, vol. 58, no. 4, pp. 431–440, 2015.

[22] R. Khan, S. U. Khan, R. Zaheer, and S. Khan, "Future internet: The internet of things architecture, possible applications and key challenges," in *FIT'12*. IEEE, 2012, pp. 257–260.

[23] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *44th International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 548–560.

[24] J. Lin, Y. Rao, J. Lu, and J. Zhou, "Runtime neural pruning," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 2181–2191.

[25] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in neural information processing systems*, 2016, pp. 2074–2082.

[26] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplication," *arXiv preprint arXiv:1412.7024*, 2014.

[27] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.

[28] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," in *Proceeding Deep Learning and Unsupervised Feature Learning NIPS Workshop*, vol. 1. ACM, 2011, p. 4.

[29] U. Köster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Gray, S. Hall, L. Hornof *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 1742–1752.

[30] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, 2015, pp. 1737–1746.

[31] Google, "Introduction to tensorflow lite," tensorflow.org/mobile/tflite, 2017, [Online; accessed 09/27/19].

[32] N. Corp., "Nvidia tensorrt," developer.nvidia.com/tensorrt, [Online; accessed 09/27/19].

[33] M. Corp., "Embedded learning library (ell)," microsoft.github.io/ELL, 2017, [Online; accessed 09/27/19].

[34] R. P. Foundation, "Raspberry pi 3b," raspberrypi.org/products/raspberry-pi-3-model-b, 2017, [Online; accessed 09/27/19].

[35] G. LLC., "Edge tpu," cloud.google.com/edge-tpu, 2019, [Online; accessed 09/27/19].

[36] N. Corp., "Jetson nano," developer.nvidia.com/embedded/buy/jetson-nano-devkit, 2019, [Online; accessed 09/27/19].

[37] I. Corp., "Intel movidius neural compute stick," software.intel.com/en-us/movidius-ncs, 2019, [Online; accessed 09/27/19].

[38] Nvidia, "Jetson nano brings ai computing to everyone," devblogs.nvidia.com/jetson-nano-ai-computing, [Online; accessed 09/27/19].

[39] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision*. Springer, 2016, pp. 21–37.

[40] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[41] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[42] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[43] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning spatiotemporal features with 3d convolutional networks," in *Computer Vision (ICCV), 2015 IEEE International Conference on*. IEEE, 2015, pp. 4489–4497.

[44] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[45] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," pp. 1251–1258, 2017.

[46] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[47] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[48] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[49] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

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

[51] F. Inc., "Open neural network exchange format," https://onnx.ai/, 2019, [Online; accessed 09/27/19].

[52] I. Corp., "Intel movidius neural compute sdk," movidius.github.io/ncsdk/tools/tools_overview.html, 2019, [Online; accessed 09/27/19].

[53] M. Abadi *et al.*, "Tensorflow," software available from tensorflow.org.

[54] Google LLC., "Tensorflow 2.0 rc," tensorflow.org/alpha/guide/eager, 2019, [Online; accessed 09/27/19].

[55] Google LLC., "Tensorflow eager execution essentials," tensorflow.org/beta, 2019, [Online; accessed 09/27/19].

[56] L. B. Rall, "Automatic differentiation: Techniques and applications," 1981.

[57] F. Chollet *et al.*, "Keras," github.com/fchollet/keras, 2015.

[58] Facebook Inc., "Caffe2," caffe2.ai/docs/getting-started, [Online; accessed 09/27/19].

[59] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[60] Intel Corp., software.intel.com/en-us/articles/prize-winning-ai-development-for-the-intel-movidius-neural-compute-stick, note = [Online; accessed 09/27/19], title = Prize-Winning AI Development for the Intel Movidius Neural Compute Stick, year = 2019,.

[61] I. Corp., "Intel neural compute stick 2," software.intel.com/en-us/neural-compute-stick, 2019, [Online; accessed 09/27/19].

[62] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[63] J. Redmon, pjreddie.com/darknet, title = Darknet: Open Source Neural Networks in C, year = 2013–2016,.

[64] Xilinx Inc., "Pynq: Python productivity for zynq," pynq.io, 2019, [Online; accessed 09/27/19].

[65] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 65–74.

[66] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated end-to-end optimizing compiler for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 578–594.

[67] TVM developers, "Vta: Deep learning accelerator stack," docs.tvm.ai/vta, [Online; accessed 09/27/19].

[68] Xilinx Inc., "Pynq overlays," pynq.readthedocs.io/en/v2.4/pynq_overlays, [Online; accessed 09/27/19].

[69] Nvidia Corp., "Nvidia jetson tx2," nvidia.com/object/embedded-systems-dev-kits-modules.html, 2017, [Online; accessed 09/27/19].

[70] Raspberry PI Foundation, "Raspberry pi 4b," raspberrypi.org/products/raspberry-pi-4-model-b, 2019, [Online; accessed 09/27/19].

[71] Nvidia Corp., "Tuning cuda applications for turing," docs.nvidia.com/cuda/turing-tuning-guide, 2019, [Online; accessed 09/27/19].

[72] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, p. 32, 2017.

[73] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, "Eridanus: Efficiently running inference of dnns using systolic arrays," *IEEE Micro*, 2019.

[74] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar, "Optimizing data warehousing applications for gpus using kernel fusion/fission," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 2012, pp. 2433–2442.

[75] M. Banzi and M. Shiloh, *Getting started with Arduino: the open source electronics prototyping platform*. Maker Media, Inc., 2014.

[76] Makerhawk, "Um25c usb power meter," makerhawk.com, 2019, [Online; accessed 09/27/19].

[77] FLIR, "Flir one thermal camera," flir.com/flirone/ios-android, note = [Online; accessed 09/27/19], 2019.

[78] L. P. Eric Bogatin, Dick Potter, *Roadmaps of Packaging Technology*. Integrated Circuit Engineering Corporation, 1997.

[79] Y. Bae, R. Hadidi, B. Asgari, J. Cao, and H. Kim, "Capella: Customizing perception for edge devices by efficiently allocating fpgas to dnns," in *2019 International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2019.

[80] Intel Inc., "Intel neural compute stick," software.intel.com/en-us/neural-compute-stick, [Online; accessed 09/27/19].

[81] Python Software Foundation, "The python profilers," docs.python.org/2/library/profile.html, [Online; accessed 09/27/19].

[82] Docker Inc., "Docker: Enterprise application container platform," docker.com, [Online; accessed 09/27/19].

[83] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning*, 2016, pp. 2849–2858.

[84] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[85] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 6848–6856.

[86] G. Huang, S. Liu, L. Van der Maaten, and K. Q. Weinberger, "Condensenet: An efficient densenet using learned group convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2752–2761.

[87] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[88] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 615–629.

[89] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "Modnn: Local distributed mobile computing system for deep neural network," in *2017 Design, automation and Test in eurpe (Date)*. IEEE, 2017, pp. 1396–1401.

[90] R. Hadidi, J. Cao, M. Woodward, M. S. Ryoo, and H. Kim, "Real-time image recognition using collaborative iot devices," in *Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-designing Pareto-efficient Deep Learning*. ACM, 2018, p. 4.

[91] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Collaborative execution of deep neural networks on internet of things devices," *arXiv preprint arXiv:1901.02537*, 2019.

[92] R. Hadidi, J. Cao, M. Woodward, M. S. Ryoo, and H. Kim, "Musical chair: Efficient real-time recognition using collaborative iot devices," *arXiv preprint arXiv:1802.02138*, 2018.

[93] R. Hadidi, J. Cao, M. L. Merck, A. Siqueira, Q. Huang, A. Saraha, C. Jia, B. Wang, D. Lim, L. Liu, and H. Kim, "Understanding the power consumption of executing deep neural networks on a distributed robot system," *Algorithms and Architectures for Learning in-the-Loop Systems in Autonomous Flight, International Conference on Robotics and Automation (ICRA) 2019*, 2019.

[94] R. Hadidi, J. Cao, T. Kirshna, M. S. Ryoo, and H. Kim, "An edge-centric scalable intelligent framework to collaboratively execute dnn," *Demo for SysML Conference, Palo Alto, CA*, 2019.

[95] D. Pena, A. Forembski, X. Xu, and D. Moloney, "Benchmarking of cnns for low-cost, low-power robotics applications," in *RSS 2017 Workshop: New Frontier for Deep Learning in Robotics*, 2017.

[96] X. Zhang, Y. Wang, and W. Shi, "pcamp: Performance comparison of machine learning packages on the edges," in {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[97] J. Hanhirova, T. Kämäräinen, S. Seppälä, M. Siekkinen, V. Hirvisalo, and A. Ylä-Jääski, "Latency and throughput characterization of convolutional neural networks for mobile computer vision," in *Proceedings of the 9th ACM Multimedia Systems Conference*. ACM, 2018, pp. 204–215.

[98] C. F. Rodrigues, G. Riley, and M. Luján, "Synergy: An energy measurement and prediction framework for convolutional neural networks on jetson tx1," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. The Steering Committee of The World Congress in Computer Science, 2018, pp. 375–382.

[99] IBM-Illinois Center for Cognitive Computing Systems Research (C3SR), "Mlmodelscope," docs.mlmodelscope.org, [Online; accessed 09/27/19].

[100] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[101] Microsoft Research, "The microsoft cognitive toolkit (cnkt)," docs.microsoft.com/en-us/cognitive-toolkit, [Online; accessed 09/27/19].

[102] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[103] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, "Redeye: Analog convnet image sensor architecture for continuous mobile vision," in *ISCA'16*. ACM, 2016, pp. 255–266.

[104] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 75–84.

[105] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 92–104.

[106] S. Wang, D. Zhou, X. Han, and T. Yoshimura, "Chain-nn: An energy-efficient 1d chain architecture for accelerating deep convolutional neural networks," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1032–1037.

[107] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, vol. 49, no. 4. ACM, 2014, pp. 269–284.

[108] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[109] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 1–12.

[110] J. Dean, "Machine learning for systems and systems for machine learning," 2017.

[111] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, "Lodestar: Creating locally-dense cnns for efficient inference on systolic arrays," in *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 2019, p. 233.

[112] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 17.

[113] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[114] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 16–25.

[115] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 26–35.

[116] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 243–254.

[117] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 267–278.