Performance Characterization of using Quantization for DNN Inference on Edge Devices: Extended Version

Hyunho Ahn*, Tian Chen*, Nawras Alnaasan, Aamir Shafi, Mustafa Abduljabbar, Hari Subramoni, and Dhabaleswar K. (DK) Panda
The Ohio State University
{ahn.377, chen.9891, alnaasan.1, shafi.16, abduljabbar.1, subramoni.1, panda.2}@osu.edu

Abstract—**Quantization is a popular technique used in Deep Neural Networks (DNN) inference to reduce the size of models and improve the overall numerical performance by exploiting native hardware. This paper attempts to conduct an elaborate performance characterization of the benefits of using quantization techniques—mainly FP16/INT8 variants with static and dynamic schemes—using the MLPerf Edge Inference benchmarking methodology. The study is conducted on Intel x86 processors and Raspberry Pi device with ARM processor. The paper uses a number of DNN inference frameworks, including OpenVINO (for Intel CPUs only), TensorFlow Lite (TFLite), ONNX, and PyTorch with MobileNetV2, VGG-19, and DenseNet-121. The single-stream, multi-stream, and offline scenarios of the MLPerf Edge Inference benchmarks are used for measuring latency and throughput in our experiments. Our evaluation reveals that OpenVINO and TFLite are the most optimized frameworks for Intel CPUs and Raspberry Pi device, respectively. We observe no loss in accuracy except for the static quantization techniques. We also observed the benefits of using quantization for these optimized frameworks. For example, INT8-based quantized models deliver $3.3\times$ and $4\times$ better performance over FP32 using OpenVINO on Intel CPU and TFLite on Raspberry Pi device, respectively, for the MLPerf offline scenario. To the best of our knowledge, this paper is the first one that presents a unique characterization study characterizing the impact of quantization for a range of DNN inference frameworks—including OpenVINO, TFLite, PyTorch, and ONNX—on Intel x86 processors and Raspberry Pi device with ARM processor using the MLPerf Edge Inference benchmark methodology.**

Index Terms—**Quantization, Edge, Inference, MLPerf**

I. Introduction

The last decade has seen the emergence of Deep Neural Network (DNN) training as an important workload on parallel systems, including High-Performance Computing and Cloud hardware. DNNs have been found to be very useful in many applications, including Computer Vision and Natural Language Processing, due to their high accuracy that is mainly due to the large number of training parameters. While significant successes [1]–[3] have been realized in training such large networks, there is relatively less focus on deploying them for inference on edge devices. The deployment of these large models for inference on commodity servers, as well as resource-constrained environments, is vital for successful *democratization* of Artificial Intelligence (AI) models.

A common challenge in deploying large models for inference is the sheer size of these models due to the large number of parameters. One technique to address this is quantization which allows using the lower-precision number format for storing weights and activations during DNN training and inference [4]. This means using formats like INT8, FP16, etc., instead of the default FP32. While quantization has been very successful in DNN training, this paper focuses on inference only.

A. Motivation

The main motivation of this paper is to conduct performance characterization using quantization for DNN inference on edge systems, including Intel x86 systems and Raspberry Pi 4B device equipped with ARM processor. We are interested in quantifying the reduction in sizes of quantized models while also measuring the accuracy of these models—the goal is to reduce the size while not affecting the accuracy. We are also motivated to explore and use the commonly used quantization techniques, including FP16 and INT8 variations. This study is done using a variety of DNN inference frameworks, including OpenVINO [5] (for Intel CPUs only), TensorFlow Lite (TFLite) [6], ONNX [7], and PyTorch [8] using specialized backends and libraries for the corresponding x86 and ARM processors. The overall goal of using quantization is to: 1) reduce the memory/energy footprint of AI models without losing accuracy and 2) improve numerical performance by exploiting native hardware support for faster arithmetic. We use the benchmarking methodology adopted by the MLPerf Edge Inference benchmarks [9].

This paper makes the following key contributions:

- Explore the use of various quantization techniques—based on INT8/FP16 and static/dynamic strategies—on a range of DNN inference frameworks, including OpenVINO, PyTorch, TFLite, and ONNX.
- The performance evaluation is done on Intel CPUs (Cascade Lake and Skylake) and Raspberry Pi 4B equipped with ARM processor.

*These authors contributed equally to this work

- The performance characterization reveals that the size of original models is reduced by a quarter for INT8-based models without losing accuracy. The only exception is when static quantization is utilized, where we witnessed a slight accuracy reduction.
- The characterization study uses a range of popular AI models—including MobileNetV2 [3], VGG-19 [2], and DenseNet-121 [1]. We found that OpenVINO and TFLite are the most optimized frameworks for Intel CPUs and Raspberry Pi 4B device, respectively. For the MLPerf offline scenario, INT8-based quantized models deliver 3.3× and 4× better performance over FP32 using OpenVINO on Intel CPU and TFLite on Raspberry Pi device, respectively.
- The evaluation is done using the MLPerf Edge Inference benchmark and uses the single-stream, multi-stream, and offline scenarios. We also studied the impact of using optimized numerical instructions like Vector Neural Network Instruction (VNNI) [10] provided by the Cascade Lake processors.

To the best of our knowledge, this paper presents a unique characterization study that studies the impact of quantization for a range of DNN inference frameworks—including OpenVINO, TFLite, PyTorch, and ONNX—on Intel x86 processors and Raspberry Pi device with ARM processor using the MLPerf Edge Inference benchmark methodology.

Rest of the paper is organized as follows. Section II presents background on DNN inference frameworks the MLPerf Edge Inference benchmark. Section III reviews important concepts related to quantization and provides an overview of our approach to quantizing models for OpenVINO, PyTorch, TFLite, and ONNX. The experimental setup for our characterization study is provided in Section IV that is followed by the detailed evaluation and analysis in Section V. Section VI presents related work and the paper is concluded in Section VII.

II. Background

A. Deep Learning Frameworks on Edge Devices

Deep Learning (DL) frameworks provide a high-level interface and building blocks for designing, training, and validating Deep Neural Networks (DNNs) on a wide range of devices. There is a plethora of ML/DL frameworks such as TensorFlow [11], PyTorch [8], CoreML [12], ONNX [7], OpenVINO [5]. Each of these frameworks differs in terms of purpose, performance, model API, and hardware compatibility. Some frameworks are designed for a specific hardware architecture, like CoreML, which is exclusively used for Apple devices. Other frameworks like OpenVINO and TensorFlow Lite (TFLite) [6] are more focused on providing an efficient and portable solution for model inference on devices that have limited memory and computing resources.

One solution to address the limitations of edge devices is the quantization of DL models to reduce the size and compute requirements for performing inference tasks. Furthermore, several low-level libraries can be used to accelerate the performance of edge devices. For instance, the ArmNN library [13] bridges the gap between the DL framework and underlying architectures by increasing the efficiency of the Arm Cortex-A CPUs and Arm Mali GPUs. ONNX supports similar libraries like NVIDIA TensorRT [14] and Intel oneDNN [15]. Intel also provides its optimized TensorFlow version for Intel CPUs [16], which uses oneDNN to fully utilize the Advanced Vector eXtensions (AVX) instruction set.

For our experiments, we select four representative frameworks that support model quantization: 1) PyTorch, which allows the training, quantization, and deployment of models within the same framework, 2) TFLite, which is the optimized TensorFlow runtime for edge devices, 3) ONNX, which offers great flexibility in translating models from/to other DL frameworks, and 4) OpenVINO, an Intel developed framework which is integrated with several Intel acceleration libraries.

B. MLPerf Inference Benchmark

The MLPerf Inference Benchmark Suite [9] is a standard machine learning (ML) benchmark suite that prescribes a set of rules and best practices to fairly evaluate the inference performance of ML hardware. It spans multiple ML models and tasks in the Computer Vision and Natural Language Processing domains, including image classification, object detection, medical imaging, speech-to-text, translation, etc. Each task and model are well-defined to ensure the reproducibility and accessibility of the benchmarks. An MLPerf Inference submission system consists of System Under Test (SUT), Load Generator (LoadGen), Accuracy Script, and Data Set unit. SUT includes the hardware, architecture, and software used in the inference. SUT should follow Model-equivalence rules, which provide a complete list of disallowed and allowed techniques in benchmarking. These rules are in place to help submitters efficiently reimplement models on various architectures. The LoadGen is a traffic generator that loads the SUT and measures performance. It produces the query traffic according to the rules of each scenario. MLPerf identifies four inference scenarios that represent many critical inference applications in real-life use cases: the single-stream, multi-stream, server and offline scenarios. Among the server senario is not required in edge benchmark. We conduct our experience in the remaining three scenarios. In each scenario, the LoadGen process generates inference requests in a particular pattern. In the single-stream and multi-stream scenarios, the LoadGen sends the next query as soon as SUT completes the previous query. In offline scenario, LoadGen sends one query with all samples to the SUT at the beginning of the execution. According to Model-equivalence rules, dynamically switching between one or more batch sizes within the scenario's limits is allowed. Following this rule, we tweak offline batch size for a given SUT in order to prevent device out of memory, as well as maximize inference throughput. Table I shows specific metrics measured in each scenario to evaluate SUT performance. For single-stream scenario, 90%-ile measured latency are measured so that 90% of total queries would be done in a given time. Similar to multi-stream scenario, but 99% of total queries would be done in a given time. Offline

TABLE I: Criteria of MLPerf Testing Scenario

Senario	Duration	Samples/Query	Performance Metric
Single-stream	1024 queries and 60 seconds	1	90%-ile measured latency (millisecond)
Multi-stream	270,336 queries and 600 seconds	8	99%-ile measured latency (millisecond)
Offline	1 query and 60 seconds	At least 24,576	Measured throughput (samples/sec)

TABLE II: The Analyzed Quantization Methods.

Quantization method	Dynamic/ Static	Bits	Data Type	Symmetric/Asymmetric
Default	N/A	32	FP32	N/A
INT8-DQ	Dynamic	8	INT8	Asymmetric
INT8-SQ	Static	8	INT8	Asymmetric
FP16	Static	16	FP16	Symmetric
INT8-OM	Static	8	INT8	Symmetric on weights, Asymmetric on activations

TABLE III: Combination of Quantization Methods and DNN Frameworks Used for Performance Evaluation.

Quantization method	PyTorch	TFLite	ONNX	OpenVINO
Default	✓	✓	✓	✓
INT8-DQ		✓	✓	
INT8-SQ	✓	✓	✓	
FP16		✓		
INT8-OM				✓

measures average throughput during inferencing in terms of samples per second.

III. Proposed Approaches and Guidelines for Deep Neural Network Quantization

This section provides an overview of relevant quantization concepts and how we used them to generate quantized models for different DL frameworks, including PyTorch, TFLite, ONNX, and OpenVINO.

A. *Quantization Methodology*

Most DNN training and inference frameworks use FP32 datatypes by default. However, the weights and activations of DNNs may not require the full range and accuracy of FP32. This provides an opportunity to exploit leaner number formats like FP16, INT16, and INT8 via model quantization. Using smaller datatypes to represent a model can lead to reduced memory footprint, smaller latency, and improved throughput. This approach is especially beneficial for edge devices with limited memory and compute resources. There are several technicalities involved when it comes to mapping the full range of FP32 values into a smaller representation:

1) Scaling Factor: In order to convert FP32 values to smaller representations, the scaling factor is used to divide the floating-point values and round them to the nearest integer. We then multiply the output by the scaling factor again. The scaling factor is critical for minimizing the difference between the original and quantized values, which in turn minimizes the quantization error.

2) Clipping Range: The clipping range determines the range of values that will be retained after quantization. All other values that fall outside this range will be clipped to the minimum or maximum bounds of this range. Clipping is performed to avoid overflow errors in the new representation and to reduce the impact of outliers that can cause issues during the quantization process. The process of choosing the clipping range is called *calibration*.

3) Quantization Symmetry: Quantization can be either symmetric or asymmetric depending on how we select the clipping range. If the minimum and maximum bounds are set to have the same distance from the central value (usually zero), then the quantized values will be symmetrically distributed. For example, in 8-bit quantization, the clipping range can be between -128 to +127 for symmetric quantization. On the other hand, in asymmetric quantization, the minimum and maximum bounds of the clipping range may have different distances from the center. This results in asymmetric distribution of quantized values. An example for 8-bit asymmetric quantization is to select the clipping range between 0 to 255.

4) Static vs. Dynamic Quantization: Another important aspect of quantization is the timing of when the scaling factor and clipping range are determined. In static quantization, the quantization parameters are determined, pre-calculated, and fixed during the inference process. Static quantization is often only applied to the weights. In dynamic quantization, on the other hand, the quantization parameters adapt to the input data while the inference is being performed. Dynamic quantization is applied on both the activations and weights and is useful when the data fed to the network varies greatly between different samples. Dynamic quantization is generally considered to be more accurate than static quantization, but it is relatively more compute-heavy compared to static quantization.

5) Post-Training Quantization (PTQ) vs. Quantization-Aware Training (QAT): In post-training quantization (PTQ), we perform quantization on a pre-trained DNN. Weights and activations are determined without retraining the DNN model. PTQ is useful when the data is limited or unlabeled. In contrast, quantization-aware training (QAT) is incorporated into the training process, which requires dataset access. During QAT, the network is trained with quantized weights and activations, which usually results in better accuracy at the cost of being a slower process compared to PTQ [4]. Also, this method additively processes pruning to optimize the network. In this paper, we focus on the quantization method only. Thus, we narrowed our experiments to the PTQ approach alone.

The quantization methods that we use in this work are detailed in Table II, which include 1) INT8 dynamic asymmetric quantization (INT8-DQ), 2) INT8 static asymmetric quantization (INT8-SQ), 3) half-precision static symmetric quantization (FP16), and 4) 8-bit static symmetric quantization on weights and asymmetric quantization on activations (INT8-OM). Table III shows the quantization method support offered by different DNN frameworks for Convolutional Neural Networks (CNNs) quantization.

B. Quantization Approachs Based on DL Frameworks

To evaluate the aforementioned quantization methods and inference scenarios, we select three representative Convolutional Neural Networks (CNNs): DenseNet-12, MobileNetV2, and VGG-19. Depending on the DL framework and quantization technique detailed Tables II and III, we quantize these three models using different configurations to evaluate their performance in terms of latency, throughput, and accuracy. Below are the proposed approaches and guidelines to quantize these models using PyTorch, TFLite, ONNX, and OpenVINO.

1) PyTorch Models: The PyTorch versions of DenseNet-121, MobileNetV2, and VGG-19 models and their weights are obtained from TorchVision [17]. PyTorch's API provides two different quantization methods called Eager Mode Quantization and FX Graph Mode Quantization. Eager Mode Quantization is an experimental feature. The user needs to perform manual operator fusion for quantization and dequantization. FX Graph Mode Quantization is a newly offered feature that automates quantization. In this work, we use the FX Graph Mode Quantization method to perform static quantization over the default FP32 model to obtain the INT8-SQ models. We only perform static quantization using PyTorch due to the framework's lack of support for dynamic quantization over convolution layers.

2) TFLite Models: The TFLite versions of DenseNet-121, MobileNetV2, and VGG-19 models and their weights are obtained from Keras Applications [18]. The quantized models were generated using TFLite default quantization converter. There are four quantization variants of TFLite models: 1) Dynamic Quantization (DQ), 2) Static Quantization (SQ), 3) FP16 quantization (FP16), 4) 16-bit activations with 8-bit weights (Mixed). Dynamic quantization is the default setting of the TFLite converter. The "dynamic-range" operators dynamically quantize activations based on their range to 8-bits and perform computations with 8-bit weights and activations. Compared to full fixed-point static quantization, the outputs of the dynamic-range operators are stored in floating-points, resulting in lesser speedups for the dynamic quantized method when compared to the full fixed-point one. Static quantization, as known as full integer quantization in TFLite, offers additional latency enhancements, decreases in peak memory usage, and improved compatibility with hardware devices that only support integers. We implemented a representative dataset feeder using the ImageNet 2012 calibration dataset, which is provided by the MLPerf Inference Benchmarks. By using this representative dataset, calibration was performed on the SQ models.

3) ONNX Models: The ONNX versions of MobileNetV2 and VGG19 model were obtained from ONNX Model Zoo [19]. The ONNX version of DenseNet-121 model was obtained through the export of the TorchVision version of DenseNet-121 from PyTorch. Quantized ONNX models can be represented in either operator-oriented (QOperator) or tensor-oriented (QDQ; Quantize and DeQuantize) methods. In the operator-oriented representation, all quantized operators have their own ONNX definitions. In contrast, in the tensor-oriented representation, quantization and dequantization functions are inserted between the original operators. The operator-oriented representation can be converted to its equivalent QDQ format [7]. In our evaluation, the ONNX Runtime APIs were used to perform dynamic and static quantization over the original ONNX format model.

4) OpenVINO Models: The OpenVINO framework supports both the ONNX format and the OpenVINO Intermediate Representation (IR) format. However, the IR format is recommended as it allows for more optimizations when using the OpenVINO Model Optimizer (MO), which only supports the IR format. To obtain quantized IR models, we first convert the original DenseNet-121, MobileNetV2, and VGG-19 ONNX models to FP32 IR models using the MO with default settings. Then, using OpenVINO Post-training Optimization Tool (POT), we perform uniform integer quantization on the obtained IR models. We implement a calibration dataset feeder using the same ImageNet 2012 calibration dataset provided by MLPerf, which provides samples needed for calibration.

The OpenVINO Post-training Optimization Tool (POT) offers a range of hyperparameters to fine-tune the quantization algorithms, giving users flexibility in choosing the number of quantized bits, number of calibration samples, symmetric/asymmetric quantization, granularity, range estimators, etc. To further improve quantization quality, we tune POT hyperparameters in five separate ways and pick one hyperparameter set with the best balance between accuracy, performance, and model size. Using this hyperparameter set, we conduct quantization on the non-quantized IR models.

IV. Experimental setup

This section details the hardware platform used for conducting this study. We also enumerate the state-of-the-art models and DL frameworks used along with models and datasets. Details on the selected quantization methods are also presented.

A. Hardware Configurations

The hardware configurations used in this paper are presented in Table IV. We rely on two HPC platforms—TACC Frontera and an internal system at The Ohio State University called RI2—as well as an edge device—Raspberry Pi 4B—to conduct our characterization study.

B. Software Packages and Versions

MLPerf Edge Inference benchmark suite v2.1 has been used in this study. This suite contains the LoadGen python module—responsible for generating input traffic—-that is built with the default setting.

The 2.9.1 version of TensorFlow Lite module in Intel-Tensorflow [16] package is used on Frontera and RI2 systems. Also, the 2.9.1 version of tflite-runtime is utilized on the

TABLE IV: Hardware specification of the Raspberry Pi 4B, in-house RI2 System, and the TACC Frontera System.

Specification	**Raspberry Pi 4B**	**Frontera**	**Ri2**
Processor Family	Cortex-A72 (ARMv8)	Xeon Cascade Lake	Xeon Skylake
Processor Model	Broadcom BCM2711	Platinum 8280	Gold 6132
Clock Speed	1.5 GHz	2.7 GHz	2.6 GHz
Sockets	1	2	2
Cores Per socket	4	28	14
RAM	8 GB	192 GB	192 GB

Raspberry Pi 4B device. The 1.12.1 version is used with PyTorch and ONNX runtime on all platforms. OpenVINO version 2022.2.0 is employed for Frontera and RI2 systems. OpenVINO is built from source code for Frontera and RI2 systems following the official build guide for CentOS. We did not use the model optimization features on frameworks not to impact the quantization characteristic.

C. Models and Datasets

In this study, we used three representative popular image classification DNN models, DenseNet-121, MobileNetV2, and VGG-19, are used:

- Dense Convolutional Network (DenseNet) has a feed-forward fashion between layer-to-layer connections. It embraced the observation that convolutional networks can be substantially deeper, more accurate, and more efficient to train if they contain shorter connections between layers close to the input and those close to the output.
- MobileNet is a class of efficient models for mobile and embedded vision applications. It is based on a streamlined architecture that uses depth-wise separable convolutions to build light weight deep neural networks.
- VGG is a class of deep convolutional networks which use architecture with tiny (3x3) convolution filters. By increasing the depth to 16-19 weight layers. It significantly improved accuracy in the large-scale image recognition setting compared with its prior state-of-the-art results.

The validation dataset of ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012) [20] is used to input data for all models under test. Input images are resized for the size that is suggested on each model. Final values are rescaled to $[0.0, 1.0]$ and then normalized using the mean value of $[0.485, 0.456, 0.406]$ and standard deviation of $[0.229, 0.224, 0.225]$.

V. Evaluation and analysis

In this section, we show the results of performance characteristics on quantization and analysis the results. The following quantization techniques were used: FP32 = Default, INT8-SQ = Static Quantization with INT8 Format, INT8-DQ = Dynamic Quantization with INT8 Format, FP16 = Half-Precision Format, INT8-OM = 8-bit symmetric quantization on weights and asymmetric quantization on activations.

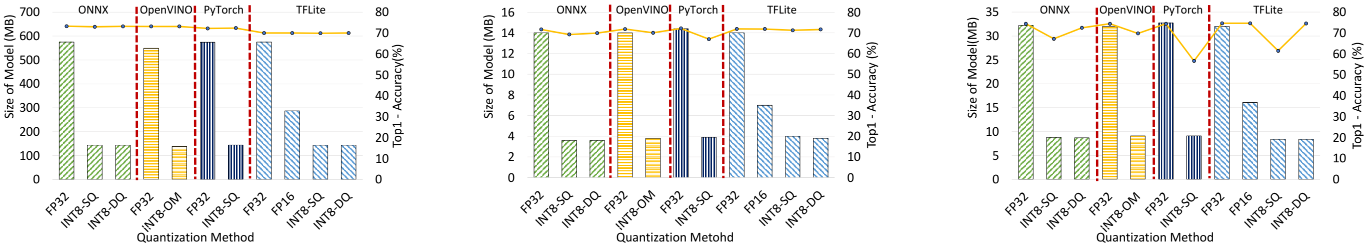
A. Model Accuracy and Size of Quantized Models

Figure 1 shows the overall experimental results of the model accuracy and size, including VGG-19, MobileNetV2, and DenseNet-121 on all four frameworks (ONNX, PyTorch, TFLite, and OpenVINO) using the ImageNet validation dataset. Since INT8-SQ, INT8-DQ, and INT8-OM utilize the 8-bit integer representation, we witness the model size reduction by a quarter to the original FP32 model. The model size of the FP16 variant is reduced by half. We note that while the model sizes are reduced substantially, the accuracy of quantized models is as good as the original FP32 models. The only exception is the INT8-SQ variants because of the use of static clipping range during the model calibration. The drop in accuracy with the INT8-SQ quantized model is the most visible for PyTorch and TFLite for the DenseNet-121 model.

B. Evaluating Inference Latency and Throughput using MLPerf Benchmark

MobileNetV2. Figures 2 and 3 present the MLPerf Edge inference benchmarks performance numbers—single-stream, multi-stream, and offline scenarios—for the MobileNetV2 model on the TACC Frontera system and the Raspberry Pi 4B device, respectively. The reason for choosing MobileNetV2 is the small model size and high accuracy (as discussed in Section V-A). Figure 2 shows that ONNX and OpenVINO are the most optimized frameworks for Intel CPUs on the Frontera system for the default FP32 format. The OM performance of OpenVINO shows performance benefits over FP32. The DQ method—for both Frontera and Raspberry—is always slower than FP32 because DQ exhibits overhead due to scale factor calculation at runtime. PyTorch is slower than ONNX and OpenVINO, but SQ improves the performance as it employs the FBGEMM library, which is optimized for low-precision calculation on the x86 architecture. TFLite shows the lowest performance among the frameworks, and quantization does not enhance the latency and throughput. The main reason is TFLite primarily targets ARM and embedded devices and is not optimized for Intel CPUs. Also, we observe that SQ only shows performance benefits in the offline scenario for the ONNX framework. This is because ONNX provides better quantization inferences with mini-match on Intel CPUs. Figure 3 presents the performance evaluation on the Raspberry Pi 4B device. Here, we exclude the OpenVINO framework since it mainly targets Intel CPU. Like Frontera, we note that DQ does not improve performance. Contrary to Frontera, where TFLite exhibited the worst performance, TFLite here shows the best performance compared to other frameworks, including ONNX and PyTorch. We do not observe any benefits of using quantization with the ONNX framework. On the other hand, the SQ performance for PyTorch is significantly faster than FP32. The main reason is that PyTorch uses the optimized QNNPACK as the backend compute library with quantized solutions.

DenseNet-121 and VGG-19. Figure 4 plots the single-stream, multi-stream, and offline scenario results—with Open-

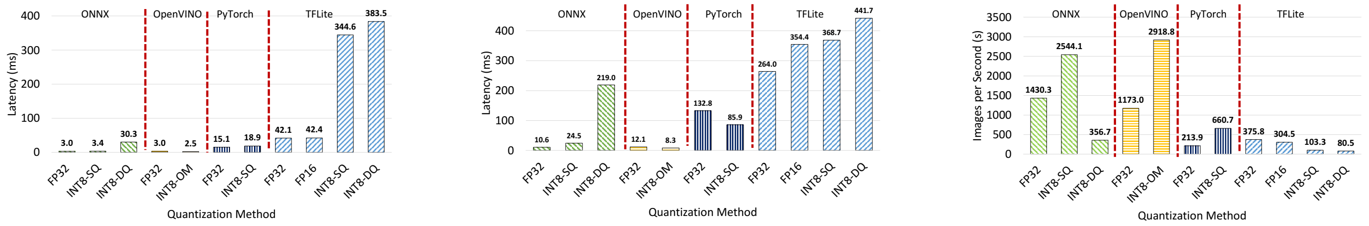


(a) VGG-19

(b) MobileNetV2

(c) DenseNet-121

Fig. 1: Model accuracy and size for VGG-19, MobileNetV2, and DenseNet-121 on all four frameworks (ONNX, PyTorch, TFLite, and OpenVINO) using the ImageNet validation dataset. Accuracy is plotted with the line on the y2 axis

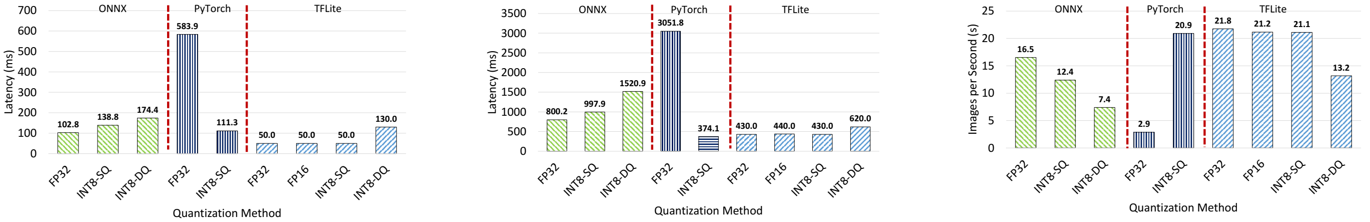


(a) Single-stream

(b) Multi-stream

(c) Offline

Fig. 2: Inference performance of ONNX, OpenVINO. PyTorch, and TFLite using MLPerf Edge benchmarks with single-stream, multi-stream, and offline scenarios on the TACC Frontera System. The model is MobileNetV2.



(a) Single-stream

(b) Multi-stream

(c) Offline

Fig. 3: Inference performance of ONNX, PyTorch, and TFLite using MLPerf Edge benchmarks with single-stream, multi-stream, and offline scenarios on the Raspberry Pi 4B device. The model is MobileNetV2.

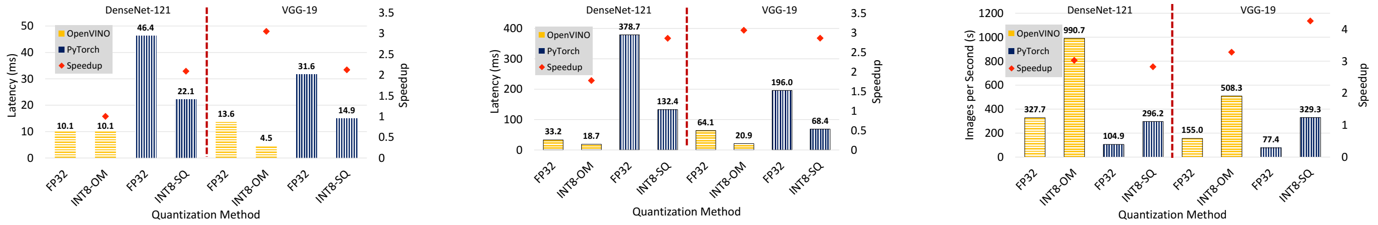
VINO and PyTorch—for DenseNet-121 and VGG-19 on the Frontera system. Results here follow the same trend as discussed earlier for Figure 2. In addition, we plot the obtained speedup on the y2 axis that is calculated by the following formula: $\frac{quantized\_performance}{FP32\_performance}$ for offline scenario and $\frac{FP32\_performance}{quantized\_performance}$ for single/multi-stream scenarios. Figure 5 plots the same scenarios/models with TFLite and PyTorch on the Raspberry Pi 4B device.

C. Impact of the Batch Size on the MLPerf Offline Scenario

The batch size hyperparameter controls the number of input images that DNN frameworks can process simultaneously during inference. This sub-section analyzes the impact of batch size using quantized weights/activations. Figure 6 shows the inference performance—for the offline scenario—of ONNX and OpenVINO (on Frontera) and TFLite (on Raspberry Pi 4B) by varying the batch size from 1 to 32. This study is done using the MobileNetV2 model. The speedup is also plotted with a line on the y2 axis using the formula: $\frac{quantized\_performance}{FP32\_performance}$. On the Frontera system (Figures 6a and 6b), we observe that the speedup improves by increasing the batch size. The best speedups of 1.8 and 2.5 are witnessed for ONNX and OpenVINO, respectively, with 32 batch size. Also, Figure 6c shows that we only witness modest benefits of increasing batch size for the TFLite framework on the Raspberry Pi 4B device. This is because the ARM processor on the device is not able to efficiently process batches of input compared to scalar input.

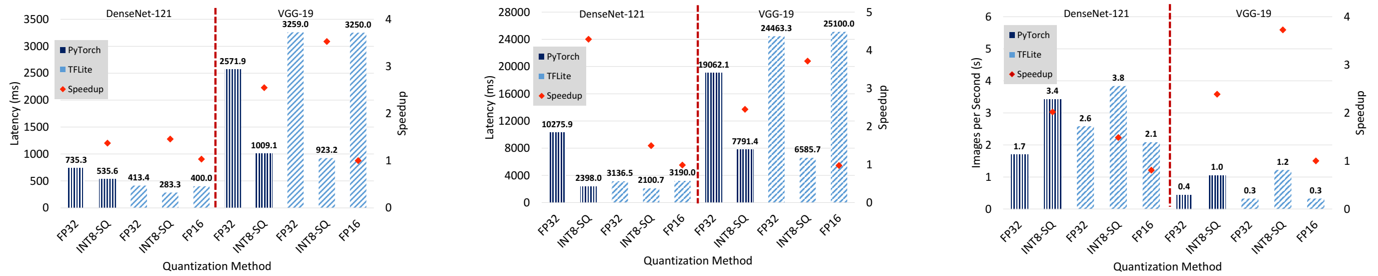
D. Benefits of Hardware Support for Inference Tasks

Many vendors are now providing hardware support for accelerating inference tasks involving quantized weights/activations. In this sub-section, we demonstrate the benefits of using a newer generation of Intel CPU (Cascade Lake vs. Skylake) for the inference performance evaluation—single-stream latency, multi-stream latency, and



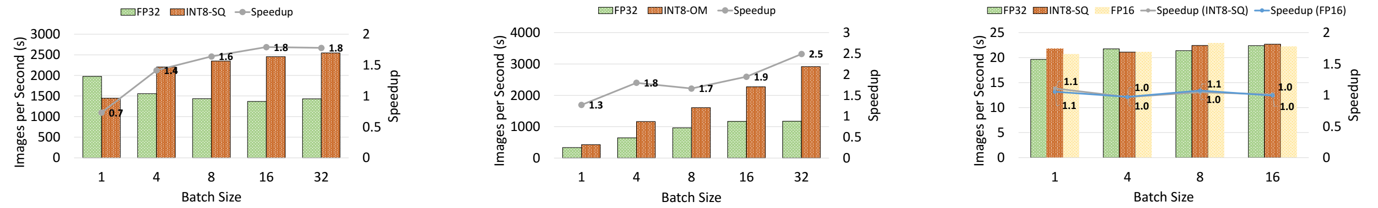
(a) Single-stream (b) Multi-stream (c) Offline

Fig. 4: Inference performance of OpenVINO and PyTorch using MLPerf Edge benchmarks with single-stream, multi-stream, and offline scenarios on the TACC Frontera System. Models are VGG-19 and DenseNet-121. Speedup is also plotted with diamonds on the y2 axis using the formula: $\frac{quantized\_performance}{FP32\_performance}$ for offline, $\frac{FP32\_performance}{quantized\_performance}$ for single/multi-stream.



(a) Single-stream (b) Multi-stream (c) Offline

Fig. 5: Inference performance of TFLite and PyTorch using MLPerf Edge benchmarks with single-stream, multi-stream, and offline scenarios on the Raspberry Pi 4B device. Models are VGG-19 and DenseNet-121. Speedup is also plotted with diamonds on the y2 axis using the formula: $\frac{quantized\_performance}{FP32\_performance}$ for offline, $\frac{FP32\_performance}{quantized\_performance}$ for single/multi-stream.



(a) ONNX (Frontera) (b) OpenVINO (Frontera) (c) TFlite (Raspberry)

Fig. 6: Inference performance of ONNX and OpenVINO (on Frontera) and TFLite (on Raspberry Pi 4B) using MLPerf Edge benchmarks with the offline scenario. The model is MobileNetV2. Speedup is also plotted with a line on the y2 axis using the formula: $\frac{quantized\_performance}{FP32\_performance}$.

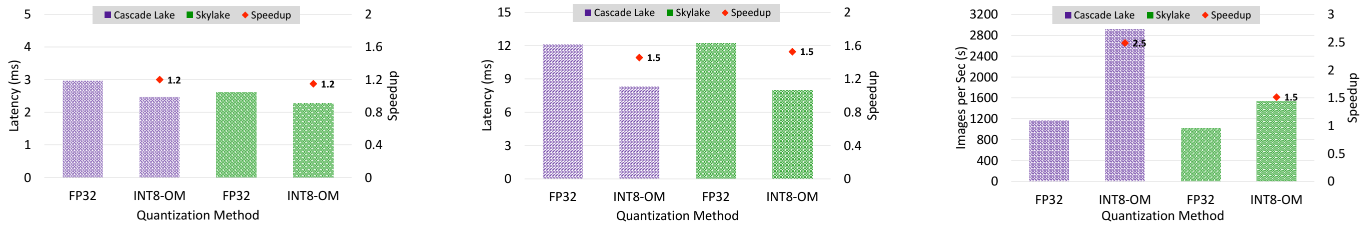
offline scenarios—with the OpenVINO framework. This is depicted in Figure 7, where Frontera and RI2 systems are equipped with Cascade Lake and Skylake processors, respectively. The main reason for better performance—especially for the offline scenario shown in Figure 7c (see 2.5× vs. 1.5× speedup)—is that the Cascade Lake processors are equipped with AVX-512 Vector Neural Network Instructions (VNNI) [10] boosting INT8 operations.

VI. Related Work

Quantization is a widely adopted method for edge device deep learning model inference. In [4], Gholami et al. conduct a survey of quantization methods. They state quantization could give benefits over multiple hardware devices like NVIDIA GPUs and ARM CPUs. Quantization results in higher power efficiency and performance, especially for edge devices. However, the study is a survey of existing quantization methods, hence, there are no numerical results in the survey.

In [21], Ulker et al. benchmark half-precision quantization on different devices with various state-of-the-art Deep Learning Frameworks. They provide detail on framework compatibility and indicate the best frameworks for each device model combination. They report the throughput and benefits of using half precision. However, no other quantization methods are further introduced, and the study has not covered frameworks targeting edge devices like TFLite.

Efforts have been made in [22] to use compiler-based approaches to generate quantized models optimized for various platforms with different device types. However, the authors use



(a) Single-stream

(b) Multi-stream

(c) Offline

Fig. 7: Inference performance of OpenVINO on Frontera (Cascade Lake processors) and RI2 (Skylake processors) using MLPerf Edge benchmarks. The model is MobileNetV2. Speedup is also plotted with diamonds on the y2 axis using the formula: $\frac{quantized\_performance}{FP32\_performance}$ for offline, $\frac{FP32\_performance}{quantized\_performance}$ for single/multi-stream.

quantized models as sanity checks for their compiler approach with limited quantization methods adopted.

In our work, we conduct a thorough analysis of multiple quantization methods in conjunction with popular deep-learning frameworks and hardware platforms from both the edge and high-end servers' worlds.

VII. Conclusions

Quantization is a useful technique in DNN inference since it reduces memory footprint of AI models and improves performance without incurring accuracy loss. However, the diversity of edge devices and DNN frameworks makes it hard to adopt this technique and get the desired performance gains. In this paper, we evaluated several quantization methods of TFLite, Pytorch, ONNX, and OpenVINO on Intel Skylake, Intel Cascade Lake, and ARMv8 processors with MobileNetV2, DenseNet-121, and VGG-19. We utilized the methodology of the MLPerf Edge Inference benchmark with three scenarios—single-stream, multi-stream, and offline—to thoroughly understand the characteristic of quantization. The paper studied important quantization features including number format (like FP16 and INT8), symmetric vs. asymmetric, and static vs. dynamic approaches. We showed quantization can achieve up to $4.3\times$ times speedup compared to FP32. However, in the absence of instruction set support and/or algorithmic optimizations such as those adopted by FBGEMM, quantization can adversely impact the inference performance. In addition to the edge platform studied herein, we compared two generations of Intel processors (Skylake vs. Cascade Lake) to emphasize the effect of hardware and library support on quantization. Overall, we highlighted the characteristics of quantization to help developers and researchers effectively adopt it in their particular configuration. In the future, we plan to study, evaluation, and characterize the impact of quantization on NVIDIA edge devices including AGX Orin using TensorRT inference framework.

VIII. Acknowledgments

This research is supported in part by NSF grants #1818253, #1854828, #1931537, #2007991, #2018627, #2112606, and XRAC grant #NCR-130002.

References

[1] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," 2016.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014.

[3] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," 2018.

[4] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," 2021.

[5] Intel, "Intel® Distribution of OpenVINO™ Toolkit." https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html. [Online; Accessed 21-January-2023].

[6] Google, "TensorFlow Lite." https://www.tensorflow.org/lite. [Online; Accessed 21-January-2023].

[7] ONNX Runtime developers, "ONNX Runtime." https://onnxruntime.ai/. [Online; Accessed 21-January-2023].

[8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, Curran Associates, Inc., 2019.

[9] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 446–459, 2020.

[10] A. Rodriguez, E. Segal, E. Meiri, E. Fomenko, Y. J. Kim, H. Shen, and B. Ziv, "Lower Numerical Precision Deep Learning Inference and Training," Jan. 2018.

[11] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015," *Software available from tensorflow. org*, 2016.

[12] Apple, "Core ML." https://developer.apple.com/documentation/coreml. [Online; Accessed 21-January-2023].

[13] Arm Software, "Arm NN ML Software." https://github.com/ARM-software/armnn. [Online; Accessed 21-January-2023].

[14] NVIDIA, "NVIDIA TensorRT." https://developer.nvidia.com/tensorrt. [Online; Accessed 21-January-2023].

[15] Intel, "oneDNN: Intel oneAPI Deep Neural Network Library." https://www.intel.com/content/www/us/en/developer/tools/oneapi/onednn.html. [Online; Accessed 21-January-2023].

[16] Intel, "TensorFlow Optimizations on Modern Intel® Architecture." https://www.intel.com/content/www/us/en/developer/articles/technical/tensorflow-optimizations-on-modern-intel-architecture.html. [Online; Accessed 21-January-2023].

[17] S. Marcel and Y. Rodriguez, "Torchvision the machine-vision package of torch," in *Proceedings of the 18th ACM International Conference on Multimedia*, MM '10, (New York, NY, USA), p. 1485–1488, Association for Computing Machinery, 2010.

[18] F. Chollet *et al.*, "Keras Applications." https://keras.io/api/applications/. [Online; Accessed 21-January-2023].

[19] ONNX, "ONNX Model Zoo." https://github.com/onnx/models. [Online; Accessed 21-January-2023].

[20] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[21] B. Ulker, S. Stuijk, H. Corporaal, and R. Wijnhoven, "Reviewing inference performance of state-of-the-art deep learning frameworks," in *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '20, (New York, NY, USA), p. 48–53, Association for Computing Machinery, 2020.

[22] A. Jain, S. Bhattacharya, M. Masuda, V. Sharma, and Y. Wang, "Efficient execution of quantized deep learning models: A compiler approach," 2020.