TensorRT Implementations of Model Quantization on Edge SoC

Yuxiao Zhou
Texas State University
y_z37@txstate.edu

Zhishan Guo
North Carolina State University
zguo32@ncsu.edu

Zheng Dong
Wayne State University
dong@wayne.edu

Kecheng Yang
Texas State University
yangk@txstate.edu

***Abstract*—Deep neural networks have shown remarkable capabilities in computer vision applications. However, their complex architectures can pose challenges for efficient real-time deployment on edge devices, as they require significant computational resources and energy costs. To overcome these challenges, TensorRT has been developed to optimize neural network models trained on major frameworks to speed up inference and minimize latency. It enables inference optimization using techniques such as model quantization which reduces computations by lowering the precision of the data type. The focus of our paper is to evaluate the effectiveness of TensorRT for model quantization. We conduct a comprehensive assessment of the accuracy, inference time, and throughput of TensorRT quantized models on an edge device. Our findings indicate that the quantization in TensorRT significantly enhances the efficiency of inference metrics while maintaining a high level of inference accuracy. Additionally, we explore various workflows for implementing quantization using TensorRT and discuss their advantages and disadvantages. Based on our analysis of these workflows, we provide recommendations for selecting an appropriate workflow for different application scenarios.**

***Index Terms*—deep neural networks, Network quantization, SoC, TensorRT, PyTorch, ONNX, edge device.**

I. Introduction

Over the last few years, deep learning (DL) has undergone significant advancements and become one of the most successful machine learning techniques. DL has enabled a wide range of applications, including computer vision, natural language processing, and autonomous control, which have been widely integrated into various software systems, including embedded ones. Unlike classical machine learning methods, deep networks can achieve high accuracy with large and over-parameterized models. The ImageNet classification leader board [1] indicates that the parameter number in state-of-the-art models for image classification have increased from 61 million to 2100 million since 2013.

Despite the high accuracy and precision achieved by large and sometimes enormous deep neural networks, their training and inference runtimes can become very slow and sluggish. Moreover, such large model architectures require significant amount of computing resources, even for inference alone. However, many applications and systems, particularly embedded ones, require real-time inference while utilizing limited hardware resources due to size, weight, power, and cost (SWaP-C) constraints. For example, autonomous vehicles must quickly process data from various sensors such as cameras and lidars to make proper control decisions in System-on-Chip (SoC). Similarly, a video surveillance system must analyze videos in real-time to detect abnormal activities. Nevertheless, for privacy and reliability reasons, this computation must often be performed on the embedded platform with limited computing resources.

The challenge posed by the inference computation barrier has resulted in a significant gap between the success of neural networks and their practical application in real-world scenarios. To address this issue, several technologies have been proposed and developed. These technologies can be broadly categorized as follows: designing low-power, highly efficient SoC chips specialized for DL inference, such as Google's Tensor Processing Unit (TPU) [6] and Intel's Vision Processing Unit (VPU) [8]; designing efficient DL model architectures by optimizing the DL model architecture in terms of its micro-architecture, designing Automated Machine Learning (AutoML) and Neural Architecture Search (NAS) [12] methods; co-designing neural network architecture and hardware together.

Despite such efforts and advances, the common, general-purpose DL framework, such as PyTorch [13], is not particularly optimized for the computing resource and time consumption of inferences. To address this issue, NVIDIA published TensorRT [2], a high-performance DL inference engine for production deployments of deep learning models.

One of the optimizations that TensorRT provides is quantization, which can reduce the precision of the weights and activations of a deep learning model. Quantization in TensorRT involves mapping the high-precision floating-point values in a model to lower-precision fixed-point or integer values. Reduced-precision inference significantly minimizes latency, which is required for many real-time services, as well as autonomous and embedded applications [2]. This quantization by TensorRT is the focus of this paper.

In this paper, we examine the effectiveness of quantization in TensorRT by comparing it to the Vanilla PyTorch (without TensorRT and Quantization) framework on edge SoC. In particular, there are three workflows that can convert the PyTorch models to quantized TensorRT engines. We evaluate the performance of three TensorRT quantization workflows under a variety of workloads and identify the performance bottlenecks in the inference using TensorRT quantization.

Contribution. Our main objective is to highlight the TensorRT

This work is supported in part by NSF grants CNS-2104181, CCF-2028481, CNS-2103604, CNS-2140346, CNS-2231523, and a REP grant from Texas State University.

quantization on edge SoC. We conducted a thorough assessment of the inference performance of quantized TensorRT engines that were converted and deployed through various workflows using different software tools. Our assessment focused on quantized model accuracy, inference time, throughput, and accuracy vs calibration batch size for each workflow. The results indicate that TensorRT quantization can significantly improve inference efficiency without compromising accuracy. There are several alternative workflows to adopt TensorRT quantization, each with its advantages and disadvantages. We analyze each workflow and suggest which one would be best suited for different application scenarios.

Organization. The rest of this paper is organized as follows: Sec. II gives a background overview of model quantization, PyTorch, ONNX, and TensorRT in these three deep learning frameworks. Sec. III presents a summary of the existing research. Sec. IV describes the methodology for our experiments, including evaluated models, workflows, and performance measuring. Sec. V provides experiment results and discussions, while Sec. VI concludes our work.

II. Background and Related Frameworks

In this section, we provide an overview of quantization, TensorRT, and other related deep-learning frameworks.

Model quantization. Model quantization is a technique used to reduce the memory and computation requirements of machine learning models by representing the model parameters in a smaller number of bits [9]. Traditionally, the weights and biases are typically represented as 32-bit floating-point numbers, which can be computationally expensive to store and process. Model quantization aims at reducing the number of bits used to represent these parameters, typically to 8-bit integers or even lower, without significantly impacting the performance of the model [5]. Model quantization can be applied to a wide range of machine learning models, including deep neural networks, convolutional neural networks, and recurrent neural networks. It has become increasingly popular in recent years due to the growing demand for deploying machine-learning models in embedded systems where the computing resources are constrained.

Quantization methodology. There are two common approaches to realize model quantization, namely *post-training quantization* (PTQ) and *quantization-aware training* (QAT) [11]. In PTQ, the models are trained using standard non-quantization techniques until it achieves the desired accuracy. Then, the weights and biases of the trained model are quantized by replacing the original 32-bit floating-point numbers with 8-bit or lower-precision fixed-point numbers. In the end, a fine-tuning step can be performed to adjust the quantized weights and biases to compensate for the loss of accuracy caused by quantization. By contrast, QAT involves training a neural network using quantization-aware optimization algorithms. During training, the model is trained to mimic the behavior of a quantized model by adjusting the weights and biases in such a way that the resulting model is better suited for hardware with limited precision.

PyTorch. PyTorch is a machine-learning framework that allows for easy transitions from research to deployment. It is primarily used as a deep learning research platform, providing speed and flexibility. It supports Tensor operations on both CPU and GPU, resulting in faster computations. It also offers various tensor routines for different scientific computation needs. Unlike other frameworks where users must repeatedly build the same neural network structure, PyTorch uses reverse-mode auto-differentiation. This technique allows users to change the network's behavior without significant overheads. PyTorch is integrated with acceleration libraries like Intel MKL and NVIDIA (cuDNN, NCCL) to maximize speed, making it fast for running networks of varying sizes. It is also memory-efficient, enabling users to train large deep-learning models [13]. Currently, PyTorch only supports running quantized operators efficiently on x86 CPUs with AVX2 support or higher and ARM CPUs. The support for NVidia GPU via TensorRT through fx2trt is still an early-stage prototype [14].

ONNX. The Open Neural Network Exchange (ONNX) [4] is an open-source artificial intelligence ecosystem that uses a common set of operators and a common file format to promote collaboration and innovation in the AI sector. The standard was created by many technology companies and research organizations to facilitate interoperability between different frameworks, tools, compilers, and runtimes. It supports multiple software frameworks such as PyTorch, TensorFlow, Caffe2, and Apache MXNet, and enables model optimization for various hardware devices. This allows users to deploy ONNX models using runtimes designed for specific hardware, which accelerates the inference execution on the device. ONNX Runtime leverages the TensorRT Execution Provider for performing quantization on GPU. TensorRT generated quantized models by taking in a full precision model and a calibration result as inputs [15].

TensorRT. TensorRT is an SDK that enables high-performance deep learning inference and is included in the NVIDIA CUDA X AI Kit. The SDK provides a deep learning inference optimizer and runtime, which ensure low latency and high throughput during deep learning inference [2]. TensorRT offers support for both PTQ and QAT techniques for creating quantized networks. PTQ involves a calibration workflow in which TensorRT measures the activation tensor distribution during network execution on representative input data and then uses that information to estimate a scale value for the tensor. Additionally, TensorRT's Quantization Toolkit is a PyTorch library that can assist in producing QAT models that are optimized by TensorRT. The toolkit also includes a recipe for PTQ that can be used to perform PTQ in PyTorch and export to ONNX [18].

III. Related Work

Xu *et al.* quantify the inference performance using TensorRT. They compared the TensorRT inference for Resnet50 with INT8 vs FP32, which shows that INT8 mode is -3.7x faster than FP32. The experiments that they did also

concluded that INT8 can also achieve the comparable accuracy with FP32 [26]. Stacker *et al.* evaluated the runtime of the deployed DNN using TensorRT and TorchScript. They chose to work with two DNN architectures: RetinaNet and PointPillars. They observed that quantization significantly reduces the runtime while having only little impact on the detection performance. [23]. Ulker *et al.* presented an evaluation of the inference performance of deep learning software tools using CNN architectures for multiple hardware platforms. They benchmarked these hardware-software pairs for a broad range of network architectures, inference batch sizes, focusing on latency and throughput. They considered both single and and half-precision floating point numbers computation in the DL frameworks. Their results reveal that TensorRT delivers minimum average execution time and highest throughput for the network models that can be translated into TensorRT engines. The performance gain from half-precision floating-point is dependent on both hardware and software tool support [24]. In Shin and Kim's recent work, they introduced a performance inference method that fuses the Jetson monitoring tool with TensorFlow and TRT source code on the Nvidia Jetson AGX Xavier platform. The CPU utilization, GPU utilization, object accuracy, latency, and power consumption of the deep learning framework were also compared and analyzed [22].

IV. Methodology

A. Neural Network Models to Evaluate

In the field of computer vision, image classification plays a crucial role in categorizing images into specific labels. Convolutional Neural Networks (CNNs) are specifically designed to handle this task. These networks employ multiple layers to detect visual patterns directly from pixel images, making them a popular choice for image classification tasks. The advantage of using CNNs lies in their ability to automatically identify significant features without any human intervention, resulting in high efficiency. This study focuses on the quantized TensorRT engine inference of the Residual Neural Networks (ResNet) [7], which was first introduced in '"Deep Residual Learning for Image Recognition." ResNet uses skip connections to improve the performance and convergence of deep neural networks. Several variants of ResNet architectures use the same concept but with varying numbers of layers. Our experiment specifically tests ResNet-50 and ResNet-152.

We also experiment with MobileNet, a small network that are well suited for platform with limited resources. It applies smart tricks in their architecture to keep the models small and efficient without sacrificing too much accuracy. MobileNet is known to be challenging to quantize [25].

B. Workflows

We designed our experiments to evaluate the performance of all possible workflows to speed up the PyTorch DL model inference by quantizing the TensorRT engine. Fig. 1 provides an overview of our experiment workflows and the software tools used in each stage. We have also highlighted the quantization tools employed in each workflow, which accelerates model inference by reducing the required precision calculations at runtime.

Pre-trained model loading. As illustrated in Fig. 1, all workflows begin with loading a pre-trained PyTorch model. During this stage, the pre-trained models are loaded using the PyTorch TorchVision library, which includes both the model architecture and pre-trained weights.

Quantization implementation. There are multiple alternative *workflows* to choose to quantize a full precision model for efficient inference. In this work, we compare the conventional, default workflow in PyTorch that does not involve quantization at all (denoted as W0) with three workflows that do integrate TensorRT quantization (tagged by W1, W2, W3, respectively). These four workflows we evaluate in this work are explained in more detail as follows.

W0: PyTorch Default

The pre-trained models are loaded on the CPU by default. To execute the inference on the GPU, we need to transfer the model from the CPU to the GPU. We also transfer the input data to the GPU to ensure that the inference executes on the GPU as well. We perform the model inference by using PyTorch Python API [13].

W1: PyTorch-Quantization

In this workflow, we quantize a PyTorch model using PyTorch-Quantization, a toolkit provided by NVIDIA for training and evaluating PyTorch models with simulated quantization. The quantized PyTorch model can be exported to ONNX and imported by TensorRT.

There are six steps in this workflow: 1) adding quantized modules, 2) PTQ, 3) QAT, 4) exporting to ONNX, 5) building the engine, and 6) engine inference.

The first step is to add quantizer modules to the neural network graph. This package provides a number of quantized layer modules, which contain quantizers for inputs and weights. These quantized layers can be substituted automatically or by manually modifying the model definition. We apply the automatic layer substitution by using `quant-modules`.

During the process of PTQ, a fixed range is selected for each quantizer. One way to achieve this is through calibration. To calibrate the activation ranges, we use a histogram-based method. To collect activation histograms, we feed sample data into the model. This is done by creating data loaders, enabling calibration in each quantizer, and feeding the calibration data into the model. A total of 1024 samples from the subset of ImageNet training data are used to estimate the distribution of activations.

Once the calibration process is complete, the quantizers will have amax set, which indicates the maximum input value that can be represented in the quantized space. The weight ranges are typically defined per channel, whereas the activation ranges are typically defined per tensor by default.

During the QAT, we fine-tune the calibrated model to improve accuracy further [21].

It should be noted that the ONNX file exported from the quantized PyTorch model cannot be directly used to build the TensorRT engine. This is because the operator "Identity_0,"

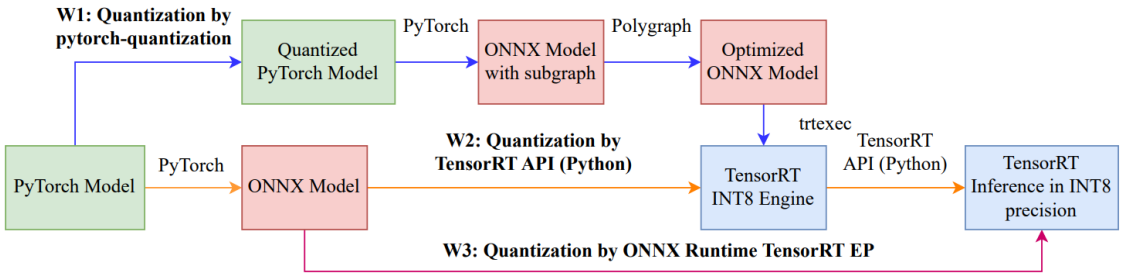

Fig. 1. An illustration of experiment workflows.

which produces an int8 zero-point, is currently not supported by TensorRT. We use Polygraphy's surgeon tool, which includes a constant folding function [20], to resolve this issue.

W2: TensorRT API Quantization

There are six steps in this workflow: 1) exporting the PyTorch model to the ONNX file, 2) defining the network, 3) setting up the calibrator, 4) configuring the builder 5) building the engine, and 6) running the engine.

In the first step, We use the `torch.onnx.export()` function in the PyTorch library exporting the PyTorch model to ONNX files. The `torch.onnx.export()` function takes an input tensor to run the model tracing its execution and then exports the traced model to an ONNX file.

Defining a network for INT8 execution is exactly the same as for any other precision. It involves operations such as creating a network definition and importing the exported ONNX model through the ONNX parser [19].

Similar to calibration in W1, we need to supply representative input data on which TensorRT runs the network to collect statistics for each activation tensor. In this project, we use 1024 images from the training set in ImageNet [3] for calibrating CNN models. Given the statistics for an activation tensor, TensorRT uses calibrators to calculate the scale values. Among four different calibrators provided by TensorRT, the IInt8EntropyCalibrator2 is recommended for CNN-based networks. It chooses the tensor's scale factor to optimize the quantized tensor's information-theoretic content and usually suppresses outliers in the distribution [21].

In the TensorRT Python API calibration is implemented by the INT8 calibrator class. Setting up the calibrator consists of the following two sub-steps: 1) create an `ImageBatchStream` object used to retrieve batch data while calibrating. `ImageBatchStream` is a helper class that takes care of file I/O, creating batch data for processing, and applying image preprocessing functions, 2) create an Int8 calibrator object with input nodes names and batch stream. The customized INT8 calibrator class must provide an implementation for `getBatchSize()` and `getBatch()` to retrieve data from the `ImageBatchStream` object.

During the TensorRT builder configuration, we set the builder precision to INT8 in addition to FP32. We also pass the calibrator object to the builder.

After we configure the builder, we can build and serialize the engine similar to the FP32 engine. Firstly, an inference execution context needs to be created. Then, memory needs to be allocated for input and output on the CUDA device. The next step is to transfer the input data from the host to the input memory that was allocated on the CUDA device. After that, TensorRT engine inference can be performed using the asynchronous execute API. The output then needs to be transferred back to the host memory. Lastly, the stream that was used for data transfers and inference execution needs to be synchronized to ensure that all operations have been completed [19].

W3: ONNX Runtime Quantization

This workflow also starts with exporting a model from PyTorch to ONNX. After obtaining the ONNX model, we perform the ONNX model quantization on GPU using the ONNX Runtime execution provider (EP).

ONNX Runtime uses its EP framework to work with various hardware acceleration libraries. This provides the flexibility to deploy ONNX models in different environments and optimize execution by taking advantage of the platform's computation capabilities. The software interacts with the EP(s) using an API to assign specific nodes or sub-graphs for execution by the EP library on supported hardware.

ONNX Runtime uses the TensorRT Execution Provider to perform quantization on the GPU. TensorRT Execution Provider requires a full precision model and a calibration result as inputs and then determines how to quantize based on its own logic [15].

We first need to configure TensorRT settings to enable model inference in INT8 precision. Then we perform quantization using TensorRT EP. This process involves implementing a `CalibrationDataReader`, computing the quantization parameters using a calibration dataset, and saving the parameters into a flatbuffer file. Finally, the model and quantization parameter files are loaded and run using the TensorRT EP [16].

C. Inference Performance Measurements

Quantized model accuracy. We assess the classification accuracy of our quantized models using the ImageNet validation dataset. This dataset comprises 50,000 photographs that have been manually labeled with the presence or absence of 1000 object categories[3]. To evaluate the performance of the quantized model, we employ two metrics: top-1 accuracy and top-5 accuracy. Top-1 accuracy measures the percentage of instances where the predicted label matches the single target label. On

the other hand, top-5 accuracy considers a classification as correct if any of the top five predictions align with the target label. These metrics provide a comprehensive assessment of the model's classification performance.

Execution time measuring. To accurately measure the inference time of the model on the GPU, we conducted several warm-up operations before taking measurements. This is because the execution speed can take some time to reach its maximum capacity. We conducted 50 hot runs of the model inference and tracked the execution time after the warm-up steps to obtain an accurate measurement of the inference time [24].

Throughput measuring. Throughput is a measure of the amount of data processed or the number of tasks completed in a specific time, usually one second. To calculate throughput, divides the number of inputs by processing time. According to [26], increasing the batch size may increase throughput. Therefore, we measure inference throughput with various batch sizes starting from one and doubling it until reaching 128 or the maximum batch size that can accommodate GPU memory.

GPU usage measuring. To monitor GPU usage, we create a separate thread alongside the inference script that is currently running. This thread is responsible for tracking GPU usage by reading the `/sys/devices/gpu.0/load` file, which provides a value representing 10 times the GPU usage. The GPU usage recording begins when the inference execution starts and ends once the inference is finished.

D. Hardware Specifications

The NVIDIA® Jetson™ is a family of embedded computing devices created by NVIDIA, designed for use in AI and machine learning applications. These devices are small, low-power, and feature-rich platforms that can run complex deep-learning models in real time. We carry out our experiments on the latest Jetson device, the NVIDIA Jetson AGX Orin 64GB. This SoC is based on the NVIDIA Ampere architecture and includes a 12-core NVIDIA Carmel ARMv8.2 CPU, a 384-core NVIDIA Volta GPU, and a 32-core NVIDIA Deep Learning Accelerator (DLA) [10].

E. Software Specifications

In our experiment, we use the latest JetPack 5.1.1 which include key components such as CUDA 11.4, TensorRT 8.5.2, cuDNN 8.6.0 and VPI 2.2 [17].

V. Experiment Results

We present our experiment results in the following five subsections: model accuracy, inference execution time, inference throughput, GPU usage and model accuracy versus calibration batch size.

A. Model Acccuracy

Quantization has many benefits but the reduction in the precision of the parameters and data can easily hurt a model's task accuracy. We verify the quality of quantized models by comparing their accuracy with the pre-trained PyTorch CNN models offered by TorchVision. Table I to Table III presents the highest top-1 accuracy and top-5 accuracies, respectively, for three evaluated models that have been quantized using different workflows.

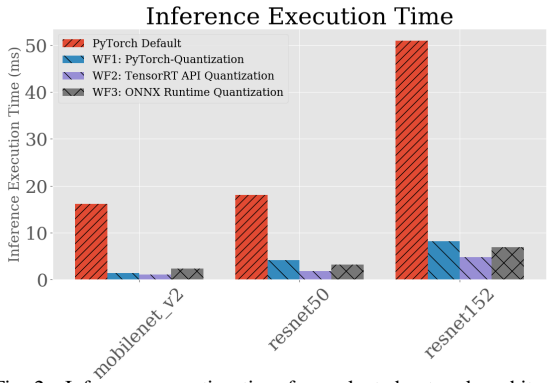

Fig. 2. Inference execution time for evaluated network architectures.

Torch-Quantization and TensorRT API Quantization achieve better accuracy than ONNX Runtime Quantization. Both of them maintain accuracy within 1% of the floating-point baseline on ResNets.

As reported in Table III, MobileNet v2, incurres a substantial loss in accuracy when quantized with PTQ of all three workflows. However, QAT in Torch-Quantization is able to maintain accuracy to within 1% of fp32 accuracy.

Observation 1. Networks with more parameters like ResNets are more robust to quantization compared to MobileNets which have fewer parameters. ONNX Runtime Quantization have higher accuracy loss compared to two other workflows. The method used to create the calibration dataset in ONNX Runtime may be the underlying cause.

B. Inference Execution Time

Time-critical applications often prioritize minimal forward execution time, where minimizing latency is more important than achieving higher throughput. In such deployments, the batch size is typically set to a minimum value.

As shown in Fig. 2, we demonstrate that quantized TensorRT engines, acquired through the TensorRT API quantization, can achieve an inference speed up about ten times faster than the PyTorch models on the NVIDIA Jetson AGX Orin platform. For MobileNet and SqueezeNet, the quantized TensorRT engine can make about 14.87 times speed up. Fig. 3 shows that the inference time of PyTorch model on GPU has higher variability and includes more outliers compared to the quantized TensorRT engines.

Observation 2. Applying TensorRT quantization can significantly improve the inference time. For example, the quantized TensorRT engine can perform up to 14.87X faster than the PyTorch model in full precision for MobileNet during inference.

Observation 3. The quantized TensorRT engines exhibit much less variation in inference time compared to the PyTorch model on ResNet architectures, indicating that they also result in more predictable and consistent execution time.

TABLE I: Best achieved top-1 accuracy and top-5 accuracy by the default and quantized ResNet-50 models

Model	PyTorch Default	Torch-Quantization	TensorRT API Quantization	ONNX Runtime Quantization
Top-1 Accuracy	76.15%	75.524%	76.092%	75.524%
Top-5 Accuracy	92.87%	92.5%	92.938%	92.478%

TABLE II: Best achieved top-1 accuracy and top-5 accuracy by the default and quantized ResNet-152 models

Model	PyTorch Default	Torch-Quantization	TensorRT API Quantization	ONNX Runtime Quantization
Top-1 Accuracy	78.312%	77.762%	78.104%	74.862%
Top-5 Accuracy	94.046%	93.802%	94.030%	92.482%

TABLE III: Best achieved top-1 accuracy and top-5 accuracy by the default and quantized MobileNet models

Model	PyTorch Default	Torch-Quantization	TensorRT API Quantization	ONNX Runtime Quantization
Top-1 Accuracy	71.878%	71.088%	70.644%	69.782%
Top-5 Accuracy	90.286%	89.942%	89.664%	89.342%

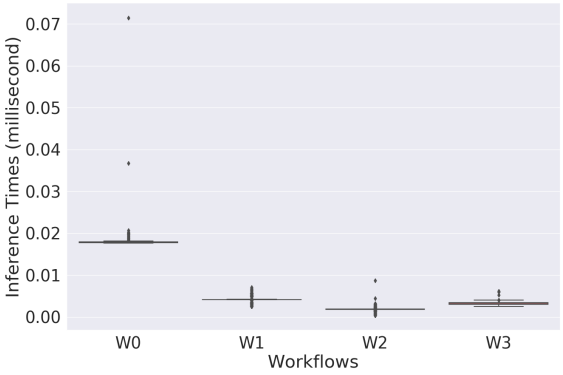

Fig. 3. Inference execution time for ResNet-50.

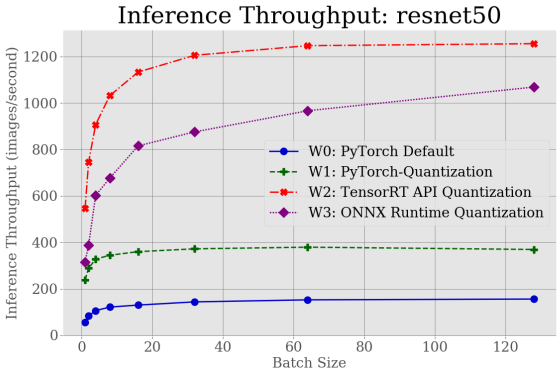

Fig. 4. Inference throughput for ResNet-50 with varying batch size.

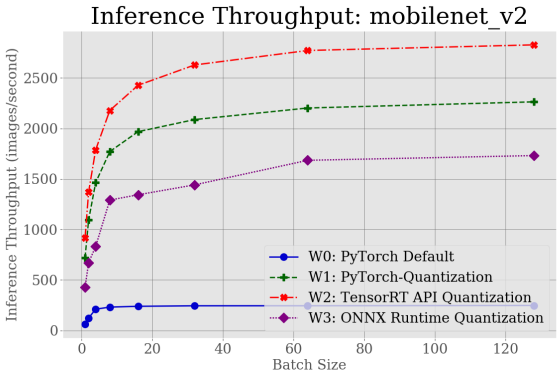

Fig. 5. Inference throughput for MobileNet-v2 with varying batch size.

C. Inference Throughput

Inference throughput is important for applications that involve multiple inference operations in a single time frame. In certain instances, delayed batch processing is acceptable. An application can benefit from increased throughput by increasing the inference batch size.

The throughput-vs-batch size curves shown in Fig. 4 and Fig. 5 demonstrate that increasing the batch size leads to a significant increase in throughput for both quantized and unquantized models until performance converges at a certain batch size. A further increase produces a limited gain in throughput because further parallelization inside the GPU is not possible.

Observation 4. Quantization can lead to a considerable improvement in inference throughput across all three workflows. Inference throughput continues to increase with an increase in batch size until it reaches the hardware limitation.

D. GPU Utilization

For inference, our goal is to minimize inference time and maximize inference throughput. Therefore, it is crucial to maximize GPU utilization during inference, particularly on embedded devices. Under-utilizing a GPU can leaves application-level performance (e.g., frames per second) on the table.

Among all TensorRT Quantization deployment workflows, both PyTorch-Quantization and TensorRT API Quantization have very high GPU utilization, close to 100%. The GPU utilization of ONNX Runtime with TensorRT Integrated ranges from 80% to 96.5% with varying input batch size and model architecture.

Observation 5. TensorRT Quantization, especially via the workflow PyTorch-Quantization and TensorRT API Quantization, maximizes the GPU utilization during inference on the edge SoC. This explains the noticable performance between TensorRT and native PyTorch specially with relative small

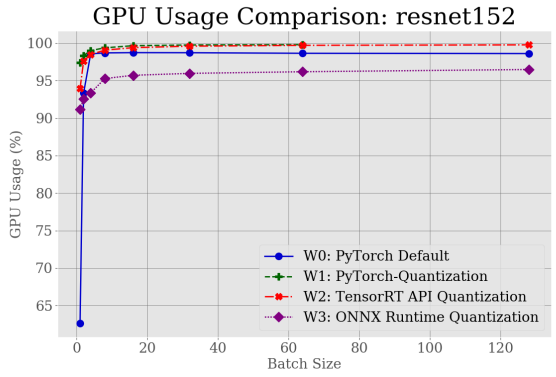

Fig. 6. GPU usage for ResNet-152 with varying batch size.

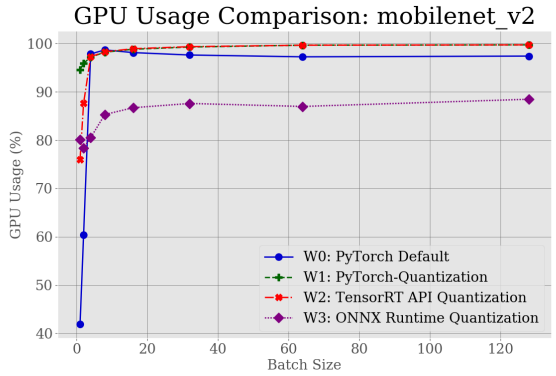

Fig. 7. GPU Usage for MobileNet-V2 with varying batch size.

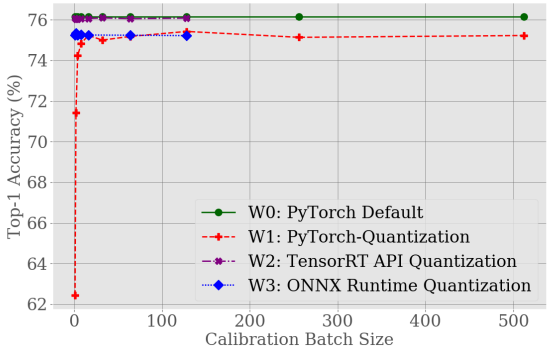

Fig. 8. Top-1 accuracy with batched calibration on ResNet-50.

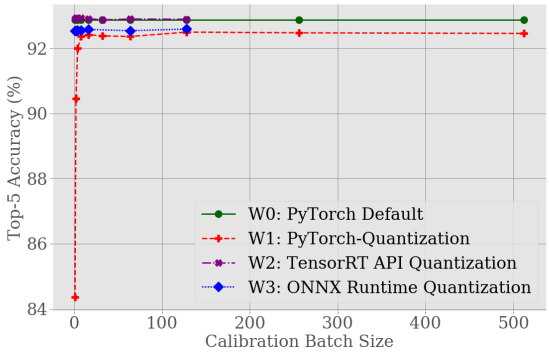

Fig. 9. Top-5 accuracy with batched calibration on ResNet-50.

input batch size.

E. Accuracy vs Calibration Batch Size

In all three workflows, we use calibration to enhance the accuracy of quantized models. During the calibration, TensorRT updates the histogram distribution for each activation tensor. If there is a new absolute max in the incoming calibration batch data, the histogram is expanded by a power of two to accommodate the new maximum value. Therefore, the size of the calibration batch can also affect the accuracy of the result TensorRT engine.

Fig. 8 to Fig. 13 display the top-1 and top-5 accuracy-vs-calibration batch size curves. From these figures, we can see that the calibration batch size does not have much impact on W2 and W3. However, calibrating with multiple calibration data of small batch size (equal to or smaller than four) can lead to poor scale value and model accuracy degradation for quantizing with the torch-quantization toolkit.

Observation 6. We need to carefully choose calibration batch size as well as other hyper parameters during model quantization especially when using the torch-quantization toolkit. Using an inappropriate batch size can result in a decrease in model accuracy.

VI. Conclusion

This paper presents an extensive comparative inference performance evaluation of a set of workflows accelerating PyTorch models with quantization using TensorRT on SoC. We focus on the local computation of CNN model inference. Based on our evaluation results, we discuss framework performance in terms of quantized model accuracy, throughput, and accuracy vs calibration batch size characteristics.

We supplemented our interpretation with an investigation of weakness and strength in each workflow. Our discussions include workflow selection for common scenarios in deep learning inference deployment for computer vision tasks. The results indicate that TensorRT API Quantization offers the most favorable overall performance for enhancing PyTorch model inference. In terms of latency and throughput, ONNX Runtime Quantization outperforms Torch-Quantization on ResNet, however, it exhibits the longest overall execution time for MobileNet. Additionally, Torch-Quantization provides QAT, which allows for fine-tuning of the calibrated model. In our experiment, the calibrated model was only fine-tuned for one epoch, but further improvement in accuracy can be achieved by employing QAT for more epochs with learning rate annealing.

We determine that no single inference workflow is optimal for all scenarios. If high inference performance is needed with limited computational resources, we recommend utilizing the TensorRT API Quantization. For applications that employ lightweight neural networks, Torch-Quantization can be employed. ONNX Runtime Quantization is suitable for accelerating inference in systems that consist of multiple frameworks like PyTorch, TensorFlow, and Apache MXNet.

Future work. In our future work, we plan to propose a new PTQ scheme that can achieve comparable accuracy to the QAT method. We also aim to conduct experiments with model

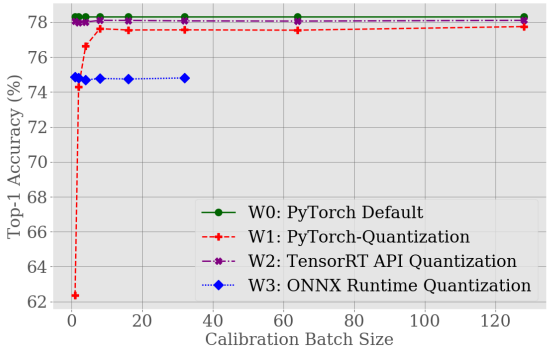

Fig. 10. Top-1 accuracy with batched calibration on ResNet-152.

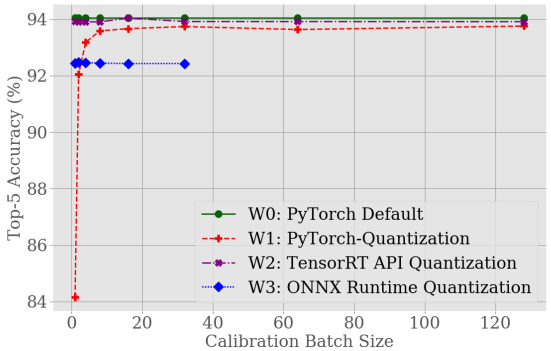

Fig. 11. Top-5 accuracy with batched calibration on ResNet-152.

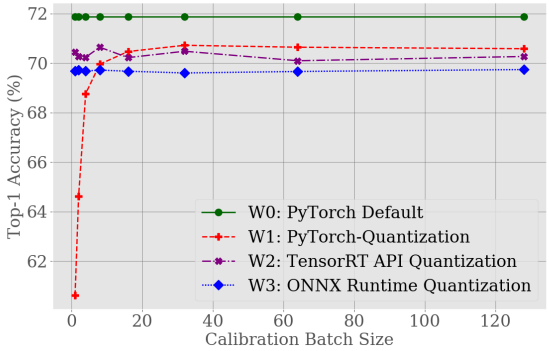

Fig. 12. Top-1 accuracy with batched calibration on MobileNet.

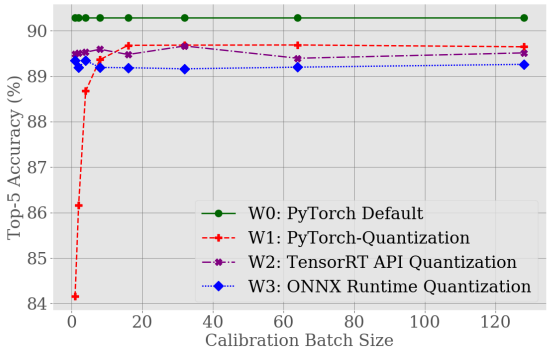

Fig. 13. Top-5 accuracy with batched calibration on MobileNet.

quantization in 4-bit. Additionally, we intend to broaden our research to include other model compression techniques such as model pruning, and to co-design the system with real-time schedulers and analysis to provide end-to-end guarantees.

References

[1] Meta AI. Image classification on imagenet. Online at https://paperswithcode.com/sota/image-classification-on-imagenet.

[2] NVIDIA Corporation. Nvidia tensorrt. Online at https://developer.nvidia.com/tensorrt.

[3] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[4] The Linux Foundation. Open neural network exchange. Online at https://onnx.ai/.

[5] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference, 2021.

[6] Google. Cloud tensor processing units (tpus). Online at https://cloud.google.com/tpu/docs/tpus.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[8] Intel. Intel® vision accelerator design with intel® movidius™ vision processing unit (vpu). Online at https://www.intel.com/content/www/us/en/developer/topic-technology/edge-5g/hardware/vision-accelerator-movidius-vpu.html.

[9] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference, 2017.

[10] Leela Karumbunathan. Nvidia jetson agx orin series technical brief. Online at https://www.nvidia.com/content/dam/en-zz/Solutions/gtcf21/jetson-orin/nvidia-jetson-agx-orin-technical-brief.pdf.

[11] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper, 2018.

[12] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European conference on computer vision (ECCV)*, pages 19–34, 2018.

[13] Meta. From research to production. Online at https://pytorch.org/.

[14] Meta. Quantization. Online at https://pytorch.org/docs/stable/quantization.html.

[15] Microsoft. Quantize onnx models. Online at https://onnxruntime.ai/docs/performance/model-optimizations/quantization.html.

[16] Microsoft. Tensorrt execution provider. Online at https://onnxruntime.ai/docs/execution-providers/TensorRT-ExecutionProvider.html.

[17] NVIDIA. Jetpack sdk. Online at https://developer.nvidia.com/embedded/jetpack.

[18] NVIDIA. Nvidia deep learning tensorrt documentation. Online at https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html.

[19] NVIDIA. Nvidia tensorrt developer guide. Online at https://docs.nvidia.com/deeplearning/tensorrt/pdf/TensorRT-Developer-Guide.pdf.

[20] NVIDIA. Onnx-tensorrt faq. Online at https://github.com/onnx/onnx-tensorrt/blob/main/docs/faq.md.

[21] NVIDIA. Quantizing resnet50. Online at https://docs.nvidia.com/deeplearning/tensorrt/pytorch-quantization-toolkit/docs/tutorials/quant_resnet50.html.

[22] Dong-Jin Shin and Jeong-Joon Kim. A deep learning framework performance evaluation to use yolo in nvidia jetson platform. *Applied Sciences*, 12(8):3734, 2022.

[23] Lukas Stäcker, Juncong Fei, Philipp Heidenreich, Frank Bonarens, Jason Rambach, Didier Stricker, and Christoph Stiller. Deployment of deep neural networks for object detection on edge ai devices with runtime optimization. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1015–1022, 2021.

[24] Berk Ulker, Sander Stuijk, Henk Corporaal, and Rob 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*, pages 48–53, 2020.

[25] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. Integer quantization for deep learning inference: Principles and empirical evaluation. *arXiv preprint arXiv:2004.09602*, 2020.

[26] Rengan Xu, Frank Han, and Quy Ta. Deep learning at scale on nvidia v100 accelerators. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 23–32. IEEE, 2018.