2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)

2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 978-1-6654-3577-2/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPSW52791.2021.00128

Performance Evaluation of Deep Learning Compilers for Edge Inference

Gaurav Verma[1] gaurav.verma@stonybrook.edu
Yashi Gupta[1] yashi.gupta@stonybrook.edu
Abid M. Malik[2] amalik@bnl.gov
Barbara Chapman[1,2] barbara.chapman@stonybrook.edu

[1]Stony Brook University, Stony Brook, New York, USA
[2] Brookhaven National Laboratory, Upton, New York, USA

***Abstract*—Recently, edge computing has received considerable attention as a promising means to provide Deep Learning (DL) based services. However, due to the limited computation capability of the data processing units (such as CPUs, GPUs, and specialized accelerators) in edge devices, using the devices' limited resources efficiently is a challenge that affects deep learning-based analysis services. This has led to the development of several inference compilers such as TensorRT, TensorFlow Lite, Relay, and TVM, which optimize DL inference models specifically for edge devices. These compilers operate on the standard DL models available for inferencing in various frameworks, e.g., PyTorch, TensorFlow, Caffe, PaddlePaddle, and transform them into a corresponding lightweight model. TensorFlow Lite and TensorRT are considered state-of-the-art inference compilers and encompass most of the compiler optimization techniques that have been proposed for edge computing. This paper presents a detailed performance study of TensorFlow Lite (TFLite) and TensorFlow TensorRT (TF-TRT) using commonly employed DL models for edge devices on varying hardware platforms. The work compares throughput, latency performance, and power consumption. We find that the integrated TF-TRT consistently performs better at the high precision floating point on different DL architectures, especially with GPUs using tensor cores. However, it loses its edge for model compression to TFLite at low precision. TFLite which is primarily designed for mobile applications, performs better with lightweight DL models than the deep neural network-based models. It is the first detailed performance comparison of TF-TRT and TFLite inference compilers to the best of our knowledge.**

***Index Terms*—TensorFlow-TensorRT, TensorFlow Lite, Compilers for DL, Inference at Edge**

I. Introduction

With an upsurge of deep learning computing processing units, deep learning-based object detection applications have received considerable research interest in recent years. In particular, as the accuracy of deep learning-based object recognition technology surpasses human capabilities, real-world deployment of intelligent surveillance technology with CCTV is expanding. Its use includes analyzing crowds and vehicle flows, performing fire detection and localization, and detecting unauthorized garbage dumping actions in urban regions.

Edge computing is also getting a lot of attention in the scientific community, especially in High Energy Physics (HEP) [29]. For example, the next generation of HEP experiments, such as the High Luminosity Large Hadron Collider (HL-LHC) and Deep Underground Neutrino Experiment (DUNE), are investing in edge computing technology addressing real-time image analysis.

In the vast majority of deployments, pre-trained deep learning models are installed on edge devices located around sensors and provide services based on deep learning inference, such as object recognition, while simultaneously performing data acquisition. The computing resources of edge devices are usually CPUs, GPUs, or FPGAs and have limited computing and power resources compared to cloud servers. Thus, various techniques and technologies have been developed to enable efficient deep learning inference on resource-constrained edge devices. These include the development of low-power, highly efficient SoC chips specialized for deep learning inference such as Google's TPU and Intel's VPU, the development of model compression methods such as quantization and the pruning of deep learning models for resource-constrained devices, as well as the design of lightweight models with reduced weights and parameters such as Mobile-Nets and YOLO, for use in edge computing environments.

However, despite such advances, effort is still needed to optimize deep learning applications like object detection models for edge computing environments, especially to maximize resource utilization. Several deep learning compilers such as TVM, TensorFlow, TensorRT, and TensorFlow Lite have been developed to address specialized accelerators' performance issues. These solutions may increase deep learning applications' performance in processing frames per second. It is not enough to fully maximize the deep learning performance with respect to memory utilization and power consumption [20]. What are the leading performance bottlenecks in the inference compiler? How do the optimization pipelines behave under different DL workloads? These are important research questions to guide future work in this area. This work is a first effort to study them.

The main contributions of this paper are summarized as follows:

- The work presents a detailed performance analysis of TensorFlow Lite and TensorFlow-TensorRT inference compilers by comparing throughput, latency, and power consumption.
- The work describes inference compilers' performance behavior concerning DL model architecture and hardware.

- The work reveals a need for a standardized benchmark suite to analyze the performance of inference compilers' optimization pipeline for edge computing.

The remainder of the paper is organized as follows: Section II gives the requisite background to understand the problem. Section III presents the related work in this area. Section IV describes the internal architecture of the inference compilers used in the study. SectionV and VI discuss experimentation and results, respectively. Section VII provides a conclusion and potential future steps.

II. Background

Deep Neural Network (DNN) based Machine Learning applications are gaining popularity to enable Artificial Intelligence on edge. For instance, in an object detection scenario, a DNN is used to extract features from the input images and classify them from the predefined categories. The inference process in a trained DNN model relies on a forward pass. It is computation-intensive, which presents a challenge to the constrained computing resources typically available on edge devices. In a cloud data center-based approach to handling this, the input data gathered through edge devices is offloaded to the cloud for processing. The results are sent back to the edge devices after the inference has been performed. These implementations are typically limited by high latency, high memory requirements, poor real-time performance in throughput and power consumption, and poor user experience. Methods [15] have been developed to distribute the DNN's computations between the cloud and the edge. Although these approaches, e.g., an early exit [19], provide a reduction in latency, they have implementation challenges for certain types of DNN. For example, AlexNet, with over 60 million nodes is hard to partition in real-time.

The inference-on-edge paradigm has emerged as a solution to address the problems of cloud-based inferencing. It aims to bring computing close to the data source to reduce latency, bandwidth use, and power consumption. Several techniques have been proposed to enable inference on edge, including model redesign [36] and human-invented architecture [34] along with model compression solutions [6], network pruning [25], parameter quantization [10], hardware acceleration based on parallel computing [16], and software acceleration focused on optimizing resource management and pipeline design. System on a Chip (SoC) [33] designs are another notable effort to improve the efficiency of inference on edge. However, the lack of any standardized chipset [35] inhibits any general optimization.

Extreme diversity in hardware and software technologies makes the development of a framework that can optimize DNN inference models for all edge systems challenging, especially with regard to low-level optimizations. The TF-TRT integrated solution and TFLite are designed with the intention of being hardware agnostic. These frameworks take as input DNN models from DL frameworks like Tensorflow, PyTorch, Caffe2, ONNX, etc., and perform fine-grained optimizations viz., quantization, layer, and tensor fusion, along with computation graph-based optimizations, delivering a lightweight model to be deployed on the edge devices. They perform these optimizations with the goal of improving memory management, power consumption and GPU utilization. Section IV discusses these frameworks in detail.

III. Related Work

Compiler development for DNNs has been spotlighted in the modern era of Machine Learning. Apache TVM [7], Facebook's Glow [13], Intel's nGraph [9], Nvidia's TensorRT [28], Google's XLA [32] and Tensorflow Lite [14] are a few notable frameworks designed to compile deep learning models into minimum deployable modules. These frameworks accept a computation graph from deep learning frameworks, such as PyTorch, Caffe2, Tensorflow, and generate highly optimized code for machine learning accelerators.

The comprehensive survey performed by Mingzhen Li et al. [21] presents the DL compilers' unique design architecture. It emphasizes the DL-oriented multi-level IRs, front-end/back-end optimizations in TVM, nGraph, TC, Glow, and XLA. Nevertheless, it does not discuss TensorFlow-TensorRT, Tensorflow Lite, or compilers for the edge inference exhaustively. TF-TRT and TFLite provide a framework for applying fine-grained optimizations to any input DNN models employed for edge inference.

In another comprehensive review, Fang Liu et al. [23] summarizes the existing edge computing systems and introduces emblematic projects. In their work, the authors contrast edge computing systems and tools like Cloudlet [31], SpanEdge [30], and AirBox [5]; Open Source Edge Computing Projects like CORD [8], Akraino Edge [1], Apache Edgent [2], Azure IoT Edge [4]. Additionally, they review Edge Computing Systems' energy efficiency, DL optimizations and present critical design issues like multi user fairness, security, privacy, and cost model. Due to resource constraints, edge inference introduces bandwidth, throughput, power, or efficiency-related challenges. Alberto Marchisio et al. [26] in their work have examined the aforementioned challenges, current trends in hardware accelerators, hardware-level optimizations, run-time optimizations, and software-level optimizations. They discuss the related case studies and present open research challenges in hardware-software co-design, in-memory computing, hardware-aware hyper-parameter tuning, and DNN architectural exploration.

This work presents a comprehensive study of state-of-the-art frameworks, TF-TRT and TFLite, for edge inference. The work evaluates the frameworks on various hardware and DNN-based models to exhibit their effectiveness in optimizing inference on the edge devices.

IV. Overview of Deep Learning Compilers

A. *TensorFlow-TensorRT integrated solution*

TensorRT (TRT) is a CUDA-based SDK for high-performance deep learning inference. It optimizes the inference and provides a runtime that delivers low latency and high-throughput for deep learning inference applications. The tight integration of TRT with TensorFlow (TF) makes the use of high-performance inference engines possible. It facilitates TRT optimizations to the TF models by optimizing the supported graphs, leaving unsupported operations to be executed by TF. TRT scans the TF graph to identify the sub-graphs, selecting candidates for graph partitioning based on the supported operations. After identifying such candidates, it converts TF layers into TRT layers for each sub-graph. Subsequently, these sub-graphs are converted to optimized TRT engines, as explained below, replacing the existing TF sub-graph. TRT's optimizations can be ported to other Nvidia GPUs. The TRT-specific optimizations are only supported on Nvidia GPUs, restricting it from being leveraged on different back-end hardware iOS GPUs or Adreno.

The ability to take diverse DL models as input makes TRT applicable to a wide range of AI based edge applications. For example, TRT can execute a variety of computer vision models to guide an unmanned aerial system flying in dynamic environments autonomously. It can be embedded to enable high throughput execution of neural network models in autonomous vehicles, deliver video analytics at the edge, or the data center, such as NVIDIA's DeepStream [27]. TRT also provides an ONNX parser and runtime extension to frameworks like Caffe 2, MxNet, Chainer, Microsoft Cognitive Toolkit, and PyTorch.

1) Pivotal Optimizations in TensorRT: TensorRT offers `INT8` and `FP16` reduced precision calibration. Following training in 32-bit precision (F32), the inference can employ half-precision, `FP16`, or even `INT8` tensor operations because gradient back-propagation is not done during the inference phase. Lower precision helps in achieving a smaller model size, lower memory utilization and latency, and higher throughput. We can control the precision in TensorRT by specifying the Data Type in the `uff_to_trt_engine` function. Since `INT8` precision can represent only `256` values (-128 to 127), TensorRT performs calibration to represent the weights and activation as 8-bit integers minimizing the accuracy loss. The calibration is a fully automated and non-parameterized method to convert `FP32` to `INT8` using a representative input training data sample.

Additionally, it performs layer and tensor fusion by parsing the computation graph and performing graph optimizations. The graph optimizations make the execution efficient without changing the underlying computation. Ordinarily, the kernel computation is faster than the kernel launch overhead coupled with the cost of reading and writing the tensor data for each layer. TensorRT addresses the memory bandwidth bottleneck and under-utilization of available GPU resources by vertically fusing the kernels to perform the sequential operations within a single kernel launch. TensorRT identifies the layers with common input data and filter size with different weights. Further, it performs horizontal fusion to convert them into a single kernel to avoid launching more than one kernels. In short, this results in an efficient graph with fewer layers and kernel launches, reducing inference latency.

TensorRT also supports kernel auto-tuning by selecting the most suitable data layers and algorithms for each target GPU platform. For example, based on the target GPU, input data size, filter size, tensor layout, batch size, and other important parameters, the inference engine opts for the best convolution algorithm from the kernels library to perform the task. Furthermore, TRT allows dynamic tensor memory management, which allocates memory for a tensor during its life-span only, thereby reducing memory footprint and enhancing memory reuse.

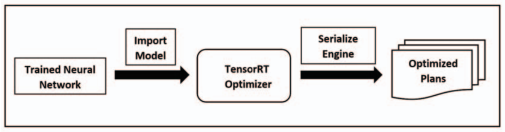


Fig. 1: Import and optimize trained models to generate inference engines

Consider Figure 1. The models trained on well-known frameworks like TensorFlow, Caffe2, MxNet, and PyTorch are fed to TRT. TRT then produces a light-weight runtime engine after optimizing the neural network computation for parameters like batch size, precision, and workspace memory for the target deployment GPU. The generated engine is an optimized inference execution engine serialized to a plan file.

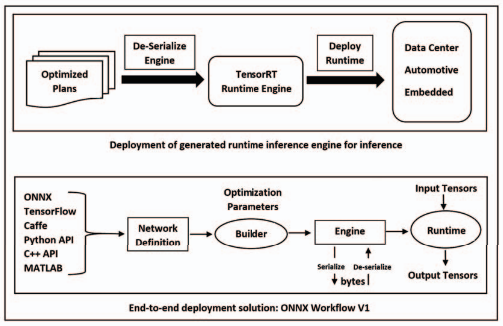


Fig. 2: Deployment Workflow

The TF-TRT workflow involves three phases as shown in Figure 2. In the first phase, a model is designed, developed, and trained in the supported frameworks. A deployment solution is then detailed and validated as part of the second phase. Factors taken into account while devising the deployment solution include: whether it is based on a single network (object detection) or multi-network (federated learning), computing device, data ingestion pipeline, data format and nature, and so on. Figure 2 shows that once the inference architecture is decided, the next step is to build the inference engine using TRT. The trained model is fed to the deployment pipeline using ONNX parser, Caffe Parser, or UFF Parser. Like ONNX, a UFF package offers utilities to convert and parse trained models from differing frameworks to a standard format. Following this, the TRT builder applies various optimization parameters, to say, batch size, mixed precision, and many more, to build an optimized inference engine specific to the infrastructure. The output inference engine is validated for accuracy as INT8 and FP16 based quantizations might lead to a slight precision loss. It is subsequently written to a plan file in a serialized format. The inference engine is initialized by deserializing this model from the plan file into an inference engine. In the last phase, the TensorRT library is linked to the deployment pipeline and is asynchronously called on-demand.

B. *TensorFlow Lite*

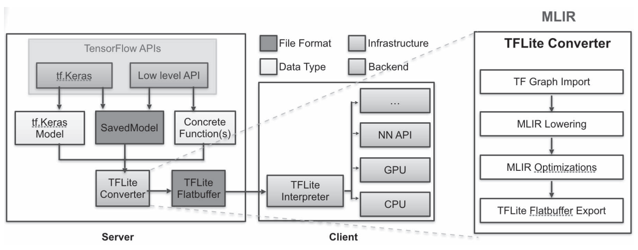


Fig. 3: TensorFlow Lite Architecture

Google's TensorFlow Lite [14] provides a set of developer tools to leverage DL models' potential on edge devices. As shown in Figure 3, TFLite consists of two main components: the server-side MLIR-based TFLite Converter and the client-side TFLite Interpreter. Multi-Level Intermediate Representation (MLIR) [18] is the reusable and extensible compiler framework for defining compiler IRs, from high-level (such as DL framework specific) to low-level LLVM-IR.

The TFLite converter tool takes a trained model in a standard format such as TensorFlow or a format interconvertible by ONNX and generates a TFLite model file (.tflite). After conversion, the model file can be deployed to a client device (e.g., a mobile or embedded system) and run locally using the TFLite interpreter. TFLite supports inference on mobile and embedded platforms, such as Android, iOS, and Linux (including Raspberry Pi), in multiple programming languages.

Edge devices are severely resource constrained, hence TFLite performs specific optimizations to generate lightweight models targeting size and latency reduction. It supports various optimization techniques, such as quantization, pruning, and clustering. By default, the TFLite interpreter utilizes CPU Kernels explicitly optimized for the ARM Neon Instruction set [3]. To fully use the back-end hardware, including accelerators like GPU, TFLite offers delegates' support. A delegate acts as a bridge between TFLite runtime and lower-level APIs associated with accelerators like OpenGL/OpenCL for Mobile GPUs. Among numerous delegates offered by TFLite, GPU Delegate is optimized for Android and iOS. It is essentially optimized to perform floating-point matrix operations, allowing an interpreter to execute GPU-supported operations on the device's GPU.

Currently, GPU delegate supports 23 TF operations, e.g., `ADD, EXP` [14]. These operations help to optimize the performance on accelerators compared to the CPUs' execution solely. It is crucial to confirm the model's supported operations before choosing a delegate. Many unsupported operations can lead to multiple hops between CPU and accelerator, impacting the latency adversely. Further, using a delegate entails added trade-offs, e.g., using a GPU Delegate induces overhead during initialization. Also, the GPU delegate does not support the quantized model.

1) Pivotal Optimizations in TensorFlow Lite: During the model conversion from a trained model to TFLite flat buffer format, TFLite Converter supports four types of quantization; dynamic range, integer, float16, and mixed precision. Dynamic range quantization supports on-the-fly quantization and de-quantization of activations so that quantized kernels can be utilized when available. Integer quantization is an optimization strategy that converts 32-bit floating-point numbers (such as weights and activation outputs) to the nearest 8-bit fixed-point numbers. Float16 quantization results in a 2x reduction in model size in exchange for minimal impacts to latency and accuracy. Mixed precision converts activations to 16-bit integer values and weights to 8-bit integer values. This mode can significantly improve the quantized model performance when activation is sensitive to the quantization while still achieving an almost 3-4x reduction in model size.

Moreover, the fully quantized model can be consumed by integer-only hardware accelerators. TFLite has exposed APIs to configure and prune either the entire model or a few selected layers to support pruning. The pruning works by removing parameters within a model that have only a minor impact on its predictions. This technique brings improvements via model compression. Another optimization that the TFLite converter performs is fusion of operations (various tensor computations). Fused operations maximize the performance of their underlying kernel implementations and provide a higher-level interface to define complex transformations like quantization. TFLite also supports clustering as an optimization. The clustering works by grouping each layer's weights in a model into a predefined number of clusters, then sharing the centered values for the weights belonging to each cluster. Hence it reduces the number of unique weight values in a model and reduces its complexity.

V. Experiments

A. *Dataset*

This section summarizes the statistical information corresponding to the datasets used by the pre-trained models to evaluate the frameworks in this work.

1) ImageNet: The ImageNet dataset is a collection of human-annotated images organized according to the WordNet hierarchy and designed for developing computer vision applications such as image classification, object detection, and object localization. WordNet is a database of English words associated with semantic relationships. Each meaningful concept in WordNet, perhaps described by multiple words or word phrases, is called a "synonym set" or "synset." ImageNet offers variations of the same object, including camera angles and lighting conditions. As per the ImageNet homepage [17], more than 14 million images are organized into over 21,000 subcategories averaging around 500 images per subcategory.

These categories are subcategories of 21 high-level categories. Additionally, slightly over 1 million images have been annotated with the bounding boxes. There are 1000 synsets and 1.2 million images with Scale-Invariant Feature Transform (SIFT) features. SIFT helps in detecting local features in an image.

2) Common Objects in Context (COCO): Microsoft's Common Objects in Context [22] is a large-scale object detection, segmentation, and captioning dataset. It consists of everyday scenes comprising common objects in their natural context. Objects are labeled using per-instance segmentations to aid in precise object localization. There are 165,482 train, 81,208 validation, and 81,434 test images encompassing 91 categories. The major portion of the dataset is non-iconic images, as they are better at generalizing.

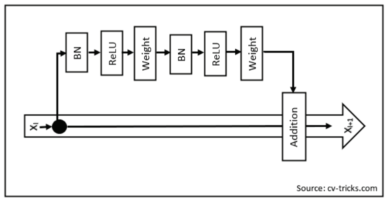
B. Evaluated Models

This section discusses the DL models used in the work for performance evaluation. We focused on Image Classification and Object Detection tasks. For the Image Classification, we used pre-trained Keras models, viz., ResNet50_v2, and MobileNet_v2, trained on the ImageNet dataset. We used the TF implementation of the SSD-MobileNet_v2 model available from the MLPerf Benchmark [12], trained on COCO Dataset to perform Object Detection.

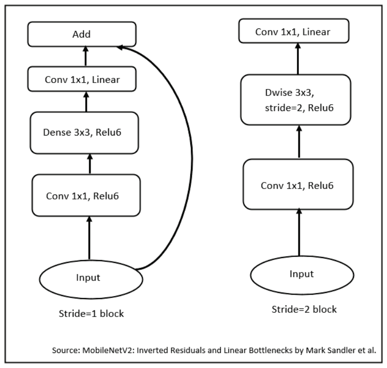
The ResNet50 is a 50-layer deep convolutional neural network (CNN). The Residual Networks (ResNet) are similar to any other deep network with convolution, pooling, activation, and fully-connected layers, except for the identity connection between the layers. The identity connection links the input to the end of the residual block. The residual block gets its name because, unlike other networks, it tries to learn the residue instead of the output. The ResNet50 inputs an image and performs convolution and max-pooling. Then it passes through four phases. In the first phase, three residual blocks contain three layers each, performing the convolution with stride 2. The stride of 2 is responsible for making the input size half and doubling the channel's height and width. As it progresses to later stages, the channel width will double, halving the input size. As shown in Figure 4a, before the multiplication with the weight matrix, it applies Batch Normalization and ReLU activation to the input tensor. Batch Normalization increases the network's performance by adjusting the input layer.

We also used MobilNet_v2. It is a lightweight CNN-based model introduced by Google, quite suitable for edge devices due to its size. It is derived from an inverted residual structure where the residual connections are between the bottleneck layers. The intermediate expansion layer uses lightweight depth-wise convolutions to filter features as a source of non-linearity. As shown in Figure 4b, generally, the first layer is a 1x1 convolution with ReLU6. The second layer is the depth-wise convolution. And lastly, the third layer is another 1×1 convolution but without any non-linearity. It contains an initially fully connected layer with 32 filters and 19 residual bottleneck layers.

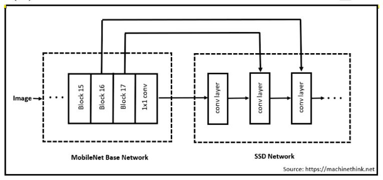
We also used the SSD_MobileNet_v2 model for the work. The SSD stands for Single Shot MultiBox Detection. The



(a) Basic architecture of ResNet50_v2



(b) Basic architecture of MobileNet_v2



(c) Basic architecture of SSD_MobileNet_v2

Fig. 4: Basic architecture of the evaluated models

Single Shot refers to the object localization and classification tasks performed in a single forward pass of the network. Multibox is the name of the technique for bounding box regression developed by Wei Liu et al. [24]. The model's input is a single image of 1x3x300x300 (BHWC) in RGB order. It is further converted to BGR format internally. The output is a typical vector containing the tracked object data. As shown in Figure 4c, the initial network consists of standard MobileNet architecture truncated before any classification layers, termed as the base network. An auxiliary network follows it; called the SSD network. The SSD network is based on a feed-forward convolutional network consisting of feature maps extraction and Object Detection using a convolution filter. It produces a fixed-size collection of bounding boxes and scores (probabilities) for the respective object classes present within those boxes.

C. Hardware Specifications

We carried out our experiments on two different Turing architecture-powered NVIDIA GPUs, GeForce RTX 2080 and Tesla T4. In our experimentation setup, where Tesla T4 GPU uses tensor cores, RTX 2080 GPU does not. The tensor cores offered by Tesla T4 help accelerate certain half-precision matrix algebra types, including General Matrix

Multiplication (GEMM). It also enables faster and easier mixed-precision computation. We can control the automatic mixed-precision by setting/unsetting the environment variable, `TF_ENABLE_AUTO_MIXED_PRECISION`. RTX 2080 has a core clock speed of 1515 MHz and a Power Consumption (TDP) of 215 Watt. On the other hand, Tesla T4 has a core clock speed of 585 MHz and power consumption (TDP) of 70 Watt. The clock speed of RTX 2080 is around 35% higher than Tesla T4, whereas typical power consumption in Tesla T4 is 3x times lower than RTX 2080. Both have PCIe 3.0 x16 interface and have GDDR6 memory types. The maximum RAM amount for RTX 2080 is 8 GB, whereas Tesla T4 offers 16 GB. Higher RAM enables Tesla T4 to store more working data and machine code currently in use, allowing quick-access and faster data processing. Additionally, the peak memory clock speed in RTX 2080 is 14000 MHz compared to 10000 MHz in Tesla T4. The better memory clock speed offers faster data read and store in the case of RTX 2080. We experimented with the `per_process_gpu_memory_fraction` flag set to 0.7. That means TensorFlow (TF) allocates a maximum of 70% of GPU memory for all its internal usage.

For the TF-Lite, we experimented with `num_threads` set to 1, 4, and 8 in the interpreter. TF-Lite by default uses the maximum number of threads available which drastically impacts the performance due to GPU resource constraints. During our experimentation, we also found that with the number of threads set to more than 8, the power consumption increased. We used the optimal number of threads based on our experiments and set `num_threads` to 4 for all our TF-Lite experiments.

D. Software Specifications

We carried out the TensorFlow Lite (TF-Lite), and TensorFlow-TensorRT (TF-TRT) experiments with the following software stacks:

For the experiments with TF-Lite, we have used TF-Lite v2.3, TensorFlow 2.4, CUDA 10.1, and CuDNN v7.5. For the experiments with TF-TRT, TensorRT's version was v5.1.5.0, TensorFlow-GPU v2.0, CUDA 10.1, and CuDNN v7.5. Since TFLite is not entirely optimized for `x86` architecture and desktop GPUs, we also carried out the experiments on a simulated device using Android Studio Emulator. We used Android Studio 4.0.1 and Pixel 3a XL simulated devices with android 10, and API 29.

VI. Results

A. Evaluation Metrics

We selected the following metrics to compare the performance of edge inference using TF-TensotRT and TFLite:

- **Throughput:** the volume of inferences within a given period, usually measured in inferences per second or samples per second (imgs/sec).
- **Latency:** the execution time to perform inference on one image, expressed in milliseconds (ms).
- **Power:** refers to the power drawn by the GPU to perform one inference. It is expressed in Watt (W),
- **Model Size:** the saved model's (.pb or .tflite) size on the disk. It is measured in Megabyte (MB).

Framework	Precision	Avg_Throughput(imgs/sec)	Avg_Latency (ms)	Avg_Power(W)	Model_Size(MB)
Native	FP32	244.50	4.089979	48	98
TF-TRT	FP32	405.73	2.464693	42	200
TFLite		5.07	197.238658	34	98
TF-TRT	FP16	485.68	2.058968	37	200
TFLite		5.10	196.078431	35	49
TF-TRT	INT8	1469.84	0.680346	34	200
TFLite		2.55	392.156862	34	25
TF-TRT	MIXED	1777.78	0.562499	34	200
TFLite		2.51	398.406374	34	25

(a) Resnet50_v2 model trained on ImageNet Dataset.

Framework	Precision	Avg_Throughput(imgs/sec)	Avg_Latency (ms)	Avg_Power(W)	Model_Size(MB)
Native	FP32	345.29	2.896116	65	14
TF-TRT	FP32	778.60	1.284356	57	32
TFLite		32.71	30.571690	35	14
TF-TRT	FP16	1326.91	0.753630	61	32
TFLite		32.91	30.385900	33	7
TRT	INT8	1803.46	0.554489	43	32
TFLite		18.53	53.966540	34	4
TF-TRT	MIXED	2320.10	0.431015	42	32
TFLite		18.68	53.533190	35	4

(b) MobileNet_v2 model trained on ImageNet Dataset.

Framework	Precision	Avg_Throughput(imgs/sec)	Avg_Latency (ms)	Avg_Power(W)	Model_Size(MB)
Native	FP32	50.72	19.715548	63	67
TF-TRT	FP32	394.31	2.536058	54	65
TFLite		8.89	112.485939	42	14
TF-TRT	FP16	399.15	2.505302	53	65
TFLite		9.28	107.758620	42	7
TF-TRT	INT8	476.57	2.098322	49	45
TFLite		4.90	204.21268	43	4
TRT	MIXED	624.43	1.601458	42	45
TFLite		4.92	204.081632	42	4

(c) SSD_MobileNet_v2 model trained on COCO Dataset.

TABLE I: Comparison between TF-Lite and TF-TensorRT on GeForce RTX 2080 GPU

B. Discussion

1) Experimentation on GeForce RTX 2080: Table I presents the results from the experiments conducted on GeForce RTX 2080 GPU. For the precision mode `FP32` and `FP16`, TF-TRT optimized ResNet50 model showed a 2x times increase in the throughput compared to the native model. In the case of `INT8` and `MIXED` precision mode, the latency has improved significantly, by a factor of 8x, boosting the throughput by nearly 7x-8x. This behavior is attributed to the TF-TRT's horizontal and vertical fusions that reduce the number of kernel launches. Across all the precision modes, TF-TRT could reduce the power consumption in a range of 10%-30%. It is noted that the saved model's size has significantly increased after the optimizations by TF-TRT, leaving the size of parameters (assets and variables) the same. We found that the induced optimized engines saved copies of the weights and variables during experiments to be later used to inspect the saved model for an efficient execution plan selection. The above finding supports the 8x times increase in the model's size post optimizations by TF-TRT. We could see a similar behavior of TF-TRT with the MobileNet too.

On the contrary, TFLite could not perform well with a deep CNN-based model like ResNet50 on `x86-64` architecture. TFLite is not optimized for desktop-based GPUs and could not utilize GPU delegate to its potential. Hence, most of the computations are getting executed on the CPUs, leading to an over 95% drop in the throughput. Irrespective of that, TFLite showed a significant decrease in the model size by 75% and an overall reduction in power consumption of roughly 35%. With the reduction in the precision mode, the space needed to save the model also reduced proportionally.

Discussing the results obtained from the experiment with the MobileNet, TF-TRT showed behavior similar to the ResNet50 model. Since MobileNet is not as deep as ResNet50, TFLite does comparatively better in performance against its performance with the ResNet50 model. Still, we can see that almost all the computation is getting executed on the CPU as it cannot employ GPU delegate on a `x86-64` architecture efficiently. Notwithstanding that we experimented with multi-threaded settings and optimized data format, NCHW, the results exhibited it did not improve the performance much.

However, SSD_MobileNet is a different architecture. The results reveal a decrease in the model size with TF-TRT. There is an 8x-11x increase in the throughput and 15%-35% reduction in the power consumption in TF-TRT. Talking about the TFLite, despite a reduction in power consumption and the model size, overall, TFLite does not perform on par with TF-TRT. TFLite displays this behavior because it is not optimized for the GPUs installed on the `x86-64` architecture. TFLite interpreter utilizes CPU Kernels explicitly optimized for the ARM Neon instruction set, not for `x86-64` instruction set.

2) Experimentation on Tesla T4: We repeated the experiments discussed in the last section on a tensor core GPU, Tesla T4. The results are summarized in Table II. As expected, overall, TF-TRT and TFLite demonstrated similar behavior for all three models. TF-TRT successfully optimized the inference latency, throughput, and power consumption. However, it suffers from an expansion in the model's size due to the reasons mentioned earlier. Being that TFLite is optimized for ARM-based GPUs. We see a drastic reduction in the throughput for all three TFLite-optimized models. Still, it can reduce the model size significantly by a factor of 75% for `INT8` precision mode.

We also observed that the throughput improved by a factor of 1.3x-1.6x on a tensor core GPU for the TF-TRT optimized models compared to a non-tensor core GPU. It was the case especially for `FP16`, `INT8`, and `MIXED` precision mode. The tensor cores are activated when the parameters of the layers are divisible by 8 or 16. Also, on a Tesla T4 machine, the GPU utilization increased extensively. Compared to the native model's utilization of a maximum of 60% of the available GPU resources, the TF-TRT optimized model utilized over 95% of the available resources on a tensor core GPU. TF-TRT's selection of hand-tuned libraries and lowering kernel launches lead to lowered GPU wait or idle time.

3) Experimentation on Pixel 3a XL, an android device: Due to the lack of any standard benchmark that can evaluate both the frameworks to provide an apple to apple comparison, we carried out experiments on an Android Device to assess the performance of TFLite. Table III shows our experimental results on Pixel 3a XL, a simulated device with android 10. The GPU delegate allows TFLite to utilize the available GPU

Framework	Precision	Avg_Throughput(imgs/sec)	Avg_Latency (ms)	Avg_Power(W)	Model_Size(MB)
Native	FP32	180.76	5.532197	64	98
TF-TRT	FP32	458.62	2.180454	38	200
TFLite		6.97	143.472022	43	98
TF-TRT	FP16	1373.41	0.728114	37	200
TFLite		4.81	207.900207	39	49
TF-TRT	INT8	2207.10	0.453083	40	200
TFLite		2.47	404.858299	33	25
TF-TRT	MIXED	2353.43	0.424911	39	200
TFLite		2.46	406.504065	32	25

(a) Resnet50_v2 model trained on ImageNet Dataset.

Framework	Precision	Avg_Throughput(imgs/sec)	Avg_Latency (ms)	Avg_Power(W)	Model_Size(MB)
Native	FP32	409.00	2.444987	43	14
TF-TRT	FP32	1584.27	0.631205	33	32
TFLite		39.56	25.278058	23	14
TF-TRT	FP16	2598.20	0.384881	34	32
TFLite		39.98	25.012506	24	7
TF-TRT	INT8	3883.14	0.257523	31	32
TFLite		18.79	53.219797	19	4
TF-TRT	MIXED	3509.76	0.284919	30	32
TFLite		18.83	53.106744	20	4

(b) MobileNet_v2 model trained on ImageNet Dataset.

Framework	Precision	Avg_Throughput(imgs/sec)	Avg_Latency (ms)	Avg_Power(W)	Model_Size(MB)
Native	FP32	32.12	31.133251	43	67
TF-TRT	FP32	402.49	2.493765	38	65
TFLite		8.57	116.686114	29	65
TF-TRT	FP16	423.93	2.375296	37	65
TFLite		8.57	116.67645	28	33
TF-TRT	INT8	610.29	1.642036	39	45
TFLite		4.75	210.526315	26	17
TF-TRT	MIXED	654.12	1.524390	37	45
TFLite		4.76	210.084033	25	17

(c) SSD_MobileNet_v2 model trained on COCO Dataset.

TABLE II: Comparison between TF-Lite and TF-TensorRT on Tesla T4 GPU

Backend	Model	Precision	Avg_Throughput(imgs/sec)	Avg_Latency (ms)	Model_Size(MB)
GPU	MobileNet_v1	Floating	745.57	10.73	17
CPU*			311.65	25.67	
GPU	MobileNet_v1	Quantized	NS	NS	NS
CPU*			571.43	14.35	4.1
GPU	SSD_MobileNet	Floating	41.67	24	27
CPU*			18.86	53	

*4 Threads; NS: Not supported

TABLE III: Execution of TF-Lite models on an android device

resources. The GPU delegate executes the GPU compatible operators, and the remaining operations get executed by the CPU. There was a 2x increase in the throughput for both the TFLite-optimized floating models compared to the native models.

Further, the GPU delegate does not support quantized models on android yet. It causes computations to be executed on the CPU. But still, TFLite performs comparatively with a 75% reduction in the model's size.

We have also estimated the loss in precision for all of the experiments conducted due to the various optimization techniques specific to individual frameworks. Conclusively, TF-TRT based optimizations lead to less than 10^{-5} precision loss for image classification tasks for the precision mode `FP32` and `FP16`. For `INT8` and `MIXED` there was a loss to an extent of 10^{-3}. Similarly, for TFLite, there was a loss in precision due to quantization by a factor of 10^{-3} for `F16` and lower precision mode. We computed the Mean Average Precision for the native and optimized model for the Object Detection task. The results for TF-TRT, TFLite, and native TF-based models were on par. To summarize, we can say that the loss in precision due to the stated compilers' various optimizations are in the acceptable range.

VII. Conclusion and Future Directions

There is a clear need to optimize DL models on edge devices for better latency and power performance. TensorFlow Lite (TFLite) and TensorFlow-TenorRT (TF-TRT) are considered state-of-the-art inference compilers for edge computing. This paper presents a detailed performance study of TFLite and TF-TRT using commonly employed DL models for edge devices on varying hardware platforms. We find that TF-TRT integration performs better at high precision mode but loses its edge for model compression to TFLite at low precision mode. TF-TRT consistently shows better performance with different DL architectures, especially with GPUs using tensor cores. However, TFLite performs better with lightweight DL models than the deep neural network-based models.

The scientific computing community is considering edge computing for its ML workload to analyze the real-time data during experiments. The performance of inference compilers such as TensorRT and TensorFlow Lite under a scientific ML workload is therefore an important question for the scientific community. As a next step, we plan to evaluate the performance of these frameworks using the DOE FAIR (Findable, Accessible, Interoperable, and Reusable) workload [11] for modern AI accelerators such as Vision Processing Unit (VPU), Field-Programmable Gate Array (FPGA), Application-Specific Integrated Circuit (ASIC), and Tensor Processing Unit (TPU).

Acknowledgment

This research work is supported in part by the U.S. Office of the Under Secretary of Defense for Research and Engineering (OUSD(R&E)) under agreement number FA8750-15-2-0119. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Office of the Under Secretary of Defense for Research and Engineering (OUSD(R&E)) or the U.S. Government.

References

[1] *Akraino*. 2018. URL: https://www.lfedge.org/projects/akraino/.

[2] *Apache Edgent*. URL: https://edgent.incubator.apache.org/.

[3] ARM. *ARM Neon Architecture*. URL: https://developer.arm.com/documentation/dht0002/a/Introducing-NEON/NEON-architecture-overview/NEON-instructions.

[4] *Azure IoT Edge*. URL: https://github.com/Azure/iotedge.

[5] Ketan Bhardwaj, Ming-Wei Shih, Pragya Agarwal, Ada Gavrilovska, Taesoo Kim, and Karsten Schwan. "Fast, scalable and secure onloading of edge functions using airbox". In: *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2016, pp. 14–27.

[6] Cristian Bucilua, Rich Caruana, and Alexandru Niculescu-Mizil. "Model compression". In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2006, pp. 535–541.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. "TVM: end-to-end optimization stack for deep learning". In: *arXiv preprint arXiv:1802.04799* 11 (2018), p. 20.

[8] *Cord*. 2018. URL: https://www.opennetworking.org/cord.

[9] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. "Intel ngraph: An intermediate representation, compiler, and executor for deep learning". In: *arXiv preprint arXiv:1801.08058* (2018).

[10] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. "Exploiting linear structure within convolutional networks for efficient evaluation". In: *Advances in neural information processing systems* 27 (2014), pp. 1269–1277.

[11] *Department of Energy Announces 8.5 Million for FAIR Data to Advance Artificial Intelligence for Science*. URL: https://www.energy.gov/articles/department-energy-announces-85-million-fair-data-advance-artificial-intelligence-science.

[12] Unai Elordi, Luis Unzueta, Jon Goenetxea, Sergio Sanchez-Carballido, Ignacio Arganda-Carreras, and Oihana Otaegui. "Benchmarking Deep Neural Network Inference Performance on Serverless Environments With MLPerf". In: *IEEE Softw.* 38.1 (2021), pp. 81–87. DOI: 10.1109/MS.2020.3030199. URL: https://doi.org/10.1109/MS.2020.3030199.

[13] *Facebook Glow*. URL: https://ai.facebook.com/tools/glow/.

[14] Google. *TensorFlow Lite*. URL: https://www.tensorflow.org/lite.

[15] Chuang Hu, Wei Bao, Dan Wang, and Fengming Liu. "Dynamic adaptive DNN surgery for inference acceleration on the edge". In: *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE. 2019, pp. 1423–1431.

[16] Loc N Huynh, Rajesh Krishna Balan, and Youngki Lee. "Deepmon: Building mobile gpu deep learning models for continuous vision applications". In: *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. 2017, pp. 186–186.

[17] *ImageNet*. URL: http://image-net.org/index.

[18] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. "MLIR: A Compiler Infrastructure for the End of Moore's Law". In: 2020. arXiv: 2002.11054 [cs.PL].

[19] En Li, Zhi Zhou, and Xu Chen. "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy". In: *Proceedings of the 2018 Workshop on Mobile Edge Communications*. 2018, pp. 31–36.

[20] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. "The Deep Learning Compiler: A Comprehensive Survey". In: *IEEE Trans. Parallel Distributed Syst.* 32.3 (2021), pp. 708–727. DOI: 10.1109/TPDS.2020.3030548. URL: https://doi.org/10.1109/TPDS.2020.3030548.

[21] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, and Depei Qian. "The Deep Learning Compiler: A Comprehensive Survey". In: *arXiv preprint arXiv:2002.03794* (2020).

[22] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. "Microsoft coco: Common objects in context". In: *European conference on computer vision*. Springer. 2014, pp. 740–755.

[23] Fang Liu, Guoming Tang, Youhuizi Li, Zhiping Cai, Xingzhou Zhang, and Tongqing Zhou. "A survey on edge computing systems and tools". In: *Proceedings of the IEEE* 107.8 (2019), pp. 1537–1562.

[24] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. "SSD: Single shot multibox detector". In: *European conference on computer vision*. Springer. 2016, pp. 21–37.

[25] Raphael Gontijo Lopes, Stefano Fenu, and Thad Starner. "Data-free knowledge distillation for deep neural networks". In: *arXiv preprint arXiv:1710.07535* (2017).

[26] Alberto Marchisio, Muhammad Abdullah Hanif, Faiq Khalid, George Plastiras, Christos Kyrkou, Theocharis Theocharides, and Muhammad Shafique. "Deep learning for edge computing: Current trends, cross-layer optimizations, and open research challenges". In: *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2019, pp. 553–559.

[27] NVIDIA. *DeepStream SDK*. URL: https://developer.nvidia.com/deepstream-sdk.

[28] NVIDIA. *TensorRT*. URL: https://developer.nvidia.com/tensorrt.

[29] Riccardo Poggi. "Digital Signal Processing in FPGA for Particle Track Reconstruction at the HL-LHC ATLAS". In: *8th International Conference on Modern Circuits and Systems Technologies, MOCAST 2019, Thessaloniki, Greece, May 13-15, 2019*. IEEE, 2019, pp. 1–4. DOI: 10.1109/MOCAST.2019.8741949. URL: https://doi.org/10.1109/MOCAST.2019.8741949.

[30] Hooman Peiro Sajjad, Ken Danniswara, Ahmad Al-Shishtawy, and Vladimir Vlassov. "Spanedge: Towards unifying stream processing over central and near-the-edge data centers". In: *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2016, pp. 168–178.

[31] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. "The case for vm-based cloudlets in mobile computing". In: *IEEE pervasive Computing* 8.4 (2009), pp. 14–23.

[32] *Tensorflow XLA*. URL: https://www.tensorflow.org/xla.

[33] Xiaying Wang, Michele Magno, Lukas Cavigelli, and Luca Benini. "Fann-on-mcu: An open-source toolkit for energy-efficient neural network inference at the edge of the internet of things". In: *IEEE Internet of Things Journal* 7.5 (2020), pp. 4403–4417.

[34] Diana Wofk, Fangchang Ma, Tien-Ju Yang, Sertac Karaman, and Vivienne Sze. "Fastdepth: Fast monocular depth estimation on embedded systems". In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 6101–6108.

[35] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. "Machine learning at facebook: Understanding inference at the edge". In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2019, pp. 331–344.

[36] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. "Learning transferable architectures for scalable image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 8697–8710.