2024 IEEE International Conference on Mobility, Operations, Services and Technologies (MOST) | 979-8-3503-0773-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/MOST60774.2024.00019

Enhancing Real-time Inference Performance for Time-Critical Software-Defined Vehicles

Sumaiya†, Reza Jafarpourmarzouni†, Sidi Lu‡ and Zheng Dong†
Department of Computer Science, Wayne State University, Detroit, Michigan, USA†
Department of Computer Science, William & Mary, Williamsburg, VA, USA‡
Email: †hl8746@wayne.edu, †hl7377@wayne.edu, ‡sidi@wm.edu, †dong@wayne.edu

***Abstract*—In the rapidly evolving landscape of vehicle computing, the efficiency and reliability of real-time responses are paramount. The primary rationale lies in the dynamic and unpredictable nature of road environments, where swift and accurate recognition of obstacles, pedestrians, and other vehicles is essential for safe navigation. Faster inference time ensures minimal latency in decision-making, allowing for immediate responses to sudden changes in the driving scenario, such as unexpected pedestrian movements or the rapid approach of other vehicles. This rapid processing capability is indispensable for preventing accidents and enhancing passenger safety. In response to these challenges, our research presents an experimental investigation aimed at accelerating inference time and maximizing throughput. We conducted a comparative analysis of four different workflows using the mainstream object detection models on TensorRT for Full Precision (FP32), Half Precision (FP16), and Integer Precision (INT8). Our results showcase the inference performance of each workflow and observations with their respective accuracy levels. This paper provides a detailed guide for selecting an appropriate workflow based on specific requirements for inference performance and accuracy, offering valuable insights for advancements in the domain of software-defined vehicles and other real-time systems.**

I. Introduction

Vehicles today are increasingly reliant on software, with high-end models containing up to 150 million lines of code. Leading automotive companies are advancing towards software-defined vehicles (SDVs) [1], which offer a suite of intelligent applications. These include oil life prediction [2], brake pad diagnostics [3], trajectory planning [4], smart remote assistance [5], dynamic travel time estimation [6], adaptive cruise control [7], battery failure forecasting [8], real-time object detection [9], in-vehicle air quality control [10], etc. All these functionalities are extensively supported by large and over-parameterized deep learning (DL) models, to attain a high degree of accuracy in perception and decision-making in real-world scenarios. This trend is evident in the number of parameters of areas such as image classification – from 61 million to 2100 million [11]; and in object detection – around 62 million to 138 million [12].

Particularly, in the domain of SDVs, the task of object detection stands out as particularly crucial, given its direct impact on the safety of these vehicles. Presently, the methodologies employed for object detection in SDVs can be broadly categorized into one-stage and two-stage object detection algorithms [13], [14]. The two-stage object detection networks are known for their exceptional accuracy. On the other hand, one-stage object detection networks are designed for faster speed. This difference between the one-stage and two-stage networks in terms of speed and accuracy represents another key consideration in the development of real-world SDVs [15].

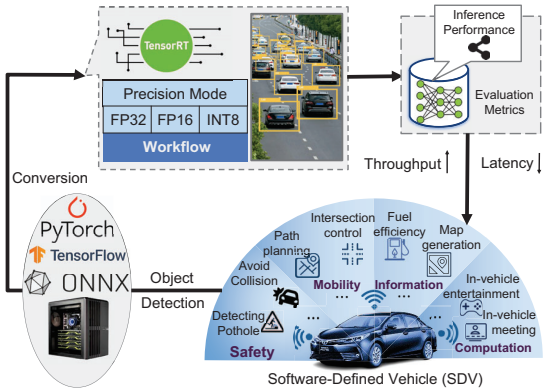


Fig. 1: An overview of the software-defined vehicles (SDVs), which integrates the mainstream object detection model using frameworks (PyTorch, TensorFlow, ONNX) and NVIDIA GPU into TensorRT precision modes, for the optimized DL inference performance.

However, as to the real-world applications of DL-based object detection models, the enhancement in model size and complexity is accompanied by a decrease in inference efficiency, evident in both throughput and execution time, which significantly impacts the safety of SDVs. Furthermore, the operational demands of these larger models necessitate significant computational resources. This presents notable challenges in the context of SDVs, where real-world applications are often constrained by limited computational capabilities. Therefore, while the accuracy and precision of large object detection models are advantageous, their practical deployment in resource-constrained environments, such as in SDVs, highlights a critical area for further optimization and development.

The significant computation barrier for deep learning (DL) model inference reveals a substantial research gap between the success of DL models and their practical application in SDVs. To mitigate this research gap, many technologies have been proposed and developed such as NVIDIA Graphics Processing Unit (GPU) for parallel processing and CUDA Optimization. As to the hardware acceleration, Intel's OpenVINO [16], model quantization [17], fine-tuning [18], hyper-parameter optimization [19], and model pruning [20] have been proposed for better performance under operational constraints. They can optimize inference pipeline [21] to reduce bottlenecks in data loading. Besides, edge computing technologies [15] have been

979-8-3503-0773-3/24/$31.00 ©2024 IEEE
DOI 10.1109/MOST60774.2024.00019

widely used to process data near the data source to reduce latency and speed up response times.

While DL frameworks like PyTorch [22] and TensorFlow [23] have made significant progress and offer robust platforms for developing complex models, they are not inherently optimized for resource efficiency and time management during the inference phase. To bridge this gap, NVIDIA has developed TensorRT [24], a specialized DL inference engine optimizer renowned for its ability to enhance the performance and efficiency of deep learning models through a series of advanced optimizations. TensorRT is designed and developed to enhance the overall inference performance, enabling developers to fine-tune neural network models trained in popular frameworks such as PyTorch and TensorFlow. This optimization plays a significant role in deploying these models across a variety of platforms, including SDVs, embedded systems, and automotive products, ensuring that they run efficiently in real-world applications where resource management and inference speed are critical.

In this work, we conducted a comprehensive analysis to evaluate the optimized performance of TensorRT, specifically in comparison to the default TensorFlow and PyTorch frameworks within the context of the perception module (object detection model) for SDVs. Our approach involved generating two distinct Road Maps, based on TensorFlow and PyTorch, respectively. For each of the Road Maps, we have two workflows to integrate the TensorRT engines with TensorFlow and PyTorch-compatible models. The brief schematic of our proposed methods (Road Maps) is shown in Fig. 1. Furthermore, We explored the precision calibration of TensorRT, which showcases a bottleneck between inference performance and accuracy using the mainstream object detection models.

Contribution. Our aim is to optimize the real-time inference speed of SDVs while upholding accuracy. To achieve this, we harnessed the vehicle's camera-captured video streams to conduct a comparative analysis. This involved generating two distinct Road Maps using TensorFlow and PyTorch frameworks, respectively. Subsequently, each Road Map underwent two workflows utilizing the YOLO object detection model on the TensorRT platform. Through meticulous evaluation across various TensorRT precision formats - Full Precision Floating Point 32, Half Precision Floating Point 16, and Integer Precision (INT8) - we sought to enhance overall inference performance. Our findings revealed compelling trends: the Half Precision Floating Point 16 format notably accelerated inference speed while maintaining acceptable accuracy levels. However, the INT8 format displayed the fastest inference speeds, albeit with a marginal compromise on accuracy. Our analysis delved into comprehensive inference performance metrics, emphasizing throughput and execution time, shedding light on the distinct strengths and limitations of each workflow.

Organization. Our research is organized throughout the paper as follows. Sec. II provides the background of popular frameworks (TensorFlow, PyTorch, ONNX) that we used throughout our research including TensorRT. Sec. III describes the experimental design of our paper which includes Road Maps, Workflows, Dataset Selection, and Hardware-Software specifications. In Sec. IV, we describe our workflows in detail. Experimental Results Analysis and Conclusion are provided in Sec.V and Sec.VI.

II. Background and Related Work

In this section, we offer a detailed outline of the frameworks employed in our research, with a particular focus on the essential building blocks. Additionally, we provide an insightful overview of the latest advancements in TensorRT in time-critical real-world scenarios.

A. TensorFlow

TensorFlow is an open-source machine learning framework developed by Google Brain known for its versatility and robustness [25]. One of its most notable features is the ability to create large-scale neural networks with multiple layers, making it suitable for complex tasks in areas of Deep Learning.

TensorFlow operates on the principle of dataflow graphs, where nodes represent mathematical operations, and edges represent the multidimensional data arrays (tensors) communicated between them. This graph-based structure enables efficient computation and parallel processing, which is essential for handling large datasets and complex algorithms. TensorFlow also excels in scalability, capable of running on a variety of platforms, from individual computers to large-scale cloud servers. It supports various languages like Python, C++, and JavaScript. TensorFlow's extensive library includes a wide range of tools and functionalities, such as TensorFlow Lite for mobile and IoT devices, TensorFlow.js for machine learning in JavaScript environments, and TensorFlow Extended (TFX) for end-to-end machine learning pipelines [26]. These tools have enabled TensorFlow to be applied in real-world scenarios such as healthcare for patient diagnosis, finance for risk analysis, and in the automotive industry for self-driving technologies.

B. PyTorch

PyTorch is an open-source machine learning library developed by the Facebook AI Research lab which is known for its flexibility, ease of use, and dynamic computational graph, which allows for more intuitive coding of deep learning models [27]. It supports fast tensor operations on both CPUs and GPUs, significantly boosting computational speed, essential for training Deep Learning models. Operations on these tensors, like matrix multiplications, convolutions, and other mathematical computations, are critical for building and training neural network models. By leveraging GPU acceleration, PyTorch can perform these tensor operations much faster compared to CPU-only processing.

Unlike many other frameworks where the structure of a neural network must be predefined and used repeatedly, PyTorch employs reverse-mode auto-differentiation. This technique permits users to alter the behavior of their network as needed, without incurring significant overheads. This feature greatly enhances the flexibility of model design, allowing for more dynamic and creative approaches to deep learning. PyTorch is also designed to be memory-efficient compared to other frameworks, enabling the training of huge deep-learning models. This efficiency in memory usage and computational speed positions PyTorch as a highly effective tool for modern machine-learning challenges.

C. ONNX

ONNX stands for Open Neural Network Exchange, is an open-source format designed to represent machine learning models [28]. It was created with a collaborative effort by Microsoft, Amazon, Facebook, and others to establish an industry standard for machine learning interoperability. The primary goal of ONNX is to enable models trained in one framework, such as PyTorch or TensorFlow, to be transferred and deployed in another for inference, thus addressing the common challenge of framework lock-in.

The core advantage of ONNX lies in its ability to decouple models from the frameworks in which they were trained. This is achieved through a shared model representation that can be understood by various software tools, making it easier to move models between different frameworks or deploy them across various platforms and devices. ONNX supports a wide range of established machine learning operations, allowing for complex models to be represented within its standard.

When it comes to real-world applications, ONNX's flexibility and framework-agnostic nature make it highly relevant, especially in the field of Autonomous Vehicles (AV). In AV, machine learning models must often be transferred between various simulation environments, testing frameworks, and the final embedded systems used in vehicles. ONNX facilitates this by ensuring that models remain consistent and performant across different software environments.

D. TensorRT

TensorRT is a high-performance deep learning inference optimizer and runtime library developed by NVIDIA [29]. It is specifically designed for production environments, offering a powerful toolset for optimizing, deploying, and running deep learning models on NVIDIA GPUs. The primary focus of TensorRT is to enhance the performance and efficiency of deep learning applications in real-time scenarios, such as in autonomous vehicles, healthcare, and robotics.

TensorRT has five types of optimization techniques to perform to enhance the performance and efficiency of deep learning models. One key technique is layer fusion, which combines multiple layers of a neural network into a single, more efficient layer. This process reduces the overhead of passing data between layers, thereby lowering latency and improving throughput. Another technique involves precision calibration, where TensorRT fine-tunes the model to use mixed precision computations. By deciding which operations can use lower-precision arithmetic (such as FP16 or INT8), it strikes a balance between inference performance and model accuracy.

FP32 (Full Precision Floating Point 32-bit). FP32 is the highest level of precision offered in TensorRT. In this mode, each number in the model's computations is represented using 32 bits. This high level of precision is ideal for maintaining the accuracy of the model; however, FP32 requires more computational resources, which can lead to slower inference speeds compared to lower-precision formats.

FP16 (Half Precision Floating Point 16-bit). FP16 reduces the numerical precision of the model by representing each number with 16 bits. This reduction in precision allows for faster processing and lower memory usage, making it a suitable option for real-world scenarios where speed is more critical compared to accuracy. FP16 is often used in applications where the model is robust enough to tolerate a slight decrease in precision without significant loss in overall performance.

INT8 (Integer 8-bit). The INT8 mode represents numbers using just 8 bits, which is the lowest level of precision in TensorRT. This mode significantly speeds up the inference process and reduces the model's memory usage almost large level. The challenge with INT8 is managing the potential loss in accuracy due to the drastic reduction in precision. TensorRT addresses this through advanced calibration techniques that attempt to maintain as much accuracy as possible.

Kernel auto-tuning is another significant optimization. Here, TensorRT tests various configurations and selects the most efficient kernels (the core computational functions) for the specific GPU architecture. Dynamic tensor memory is another technique, which enhances memory reuse by allocating memory to tensors only for the duration of their usage, thus reducing overall memory consumption and avoiding allocation overhead. Moreover, TensorRT introduces multi-stream execution, allowing the processing of multiple input streams in parallel, further boosting the model's throughput.

These five types of optimizations collectively make TensorRT a powerful tool for deploying deep learning models, especially in scenarios requiring high-speed, efficient processing. By streamlining models for better performance on NVIDIA GPUs, TensorRT plays a significant role in various real-time and high-performance computing applications, especially in the field of SDVs. Several approaches using TensorRT have been made to make inference faster for real-world scenarios - Zhou *et al.* improved real-time inference performance for DL model using TensorRT [11], Jocher *et al.* explored TensorFlow, OpenVINO and TensorRT on Ultralytics models [30]–[32] Shafi *et al.* uses TensorRT to characterize NN inference for Edge Devices [33], Hong *et al.* optimized overall performance of Multi-object tracking with TensorrRT [34], In the same year, Wang *et al.* deployed their model with TensorRT on real-time to increase inference on edge devices [35], Jiang *et al.* visits semantic segmentation for real-world Autonomous Driving using TensorRT TF16 mode [36], Huang *et al.* uses parallel execution mode using CUDA context for better inference throughput in real-world [37]. Although they were able to make the inference performance better in real-world scenarios, there was a significant drop in accuracy.

E. The Gap in Previous Work

Shin and Kim introduced a performance inference approach on the Nvidia Jetson AGX Xavier, integrating the Jetson monitoring tool with TensorFlow and TRT, and analyzed various performance metrics of the deep learning framework [38]. Jeong *et al.* developed a method using TensorRT that leverages both GPU and NPUs, enhancing a single DNN application's throughput significantly [39]. Ulker *et al.* assessed the inference efficacy of deep learning tools on multiple platforms using CNNs, focusing on latency and throughput, and found that TensorRT provides the lowest average execution time and highest throughput for compatible network models [40]. Zhou *et al.* presented an evaluation of the inference performance integrated with popular DL frameworks and TensorRT [11].

In the domain of SDVs, the current state-of-the-art demon-

strates impressive achievements in accelerating inference performance using TensorRT. However, it is noteworthy that existing research lacks a comprehensive exploration of all TensorRT precision modes. While several studies have employed a singular precision mode within TensorRT to boost inference performance, our approach is distinctively more in-depth. We are advancing our methodologies by thoroughly investigating the effects of multiple precision modes, specifically FP32, FP16, and INT8, across several workflows incorporated with various frameworks. Our complete examination aims to optimize inference performance more effectively, thereby contributing to a more nuanced understanding of the field in software-defined vehicles.

III. Experimental System Design

A. Basic Deep Neural Network Models of SDV

Plenty of deep neural network (DNN) models are deployed in the computing system of SDV for sensing, perception, localization, prediction, control, and entertainment. Object detection is a fundamental deep learning application for SDVs [1]. In this work, we choose the fundamental application, object detection, as the case study. As to the deep learning-based object detection approaches, the state-of-the-art methods include the Regions with CNN features (RCNN) series [41]–[44], Single Shot MultiBox Detector (SSD) series [45], [46], and You Only Look Once (YOLO) series [**?**], [47], [48]. In this paper, we focus on the TensorRT inference of two latest DNN models: YOLOv4 [49] and YOLOv5s [50]. YOLOv4 emphasizes performance optimization on NVIDIA hardware, while YOLOv5s, a variant of YOLOv5, is recognized for its implementation and performance enhancements within the PyTorch framework, diverging from the original Darknet-based YOLO models.

B. Road Maps and Workflows

To implement the inference in an already trained model, multiple mainstream machine learning frameworks can be used. The typical examples are TensorFlow and PyTorch. In this work, we introduce two comprehensive Road Maps designed to enhance the inference performance of all possible workflows for SDV based on TensorFlow and PyTorch, respectively (tagged by RM_T and RM_P).

For RM_T, we compare the conventional, default workflow in TensorFlow that does not involve TensorRT at all (denoted as W_0) with two workflows that integrate TensorRT (denoted as W_1 and W_2, respectively) on three open-source automotive datasets. Besides, for each workflow, we consider all three precision levels, including Full Precision (FP32), Half Precision (FP16), and Integer Precision (INT8). As to RM_P, we also have the similar design, considering W_0, W_1, and W_2 with FP32, FP16, and INT8. Throughout these workflows, we consistently maintained an image size of 416 $\times$ 416 and varied the batch size between 1 to 8. These workflows that we evaluate are summarized below. Detailed information on workflows will be discussed in Section IV.

RM_T/RM_P-W_0: TensorFlow/ PyTorch Default: By default for both PyTorch and TensorFlow framework, we loaded our already trained model and ensures that the model's inference executes on GPU. Then, we run the inference of the model through TensorFlow and PyTorch using Python API.

RM_T-W_1: TensorFlow-TensorRT: In this workflow of TensorFlow, we accelerate inference performance using TensorFlow-TensorRT Graph Converter. At a high level, there are three steps in this workflow, including i) Conversion of the TensorFlow Model to Frozen (.pb) Format, ii) Conversion of the Frozen Model to the TensorRT-compatible version, and iii) deploying model for inference.

RM_T-W_2: TensorFlow-ONNX-TensorRT: This workflow is systematically divided into four key stages. Each stage plays a vital role in transforming the model from its original state into an optimized format ready for deployment and inference. The stages are as follows: i) Conversion of TensorFlow to Frozen Model, the same way as W_1 does. ii) Frozen model to ONNX Conversion, iii) TensorRT Engine Building, and iv) Deploy for Inference.

RM_P-W_1: Torch-TensorRT: In this workflow of PyTorch, we employed the model using Torch-TensorRT (a collaborative effort combining PyTorch with NVIDIA's TensorRT). The Torch-TensorRT workflow has three unique phases: i) Simplifying TorchScript module, ii) Transformation, and iii) Execution of optimized graph.

RM_P-W_2: Torch-ONNX-TensorRT: In this workflow, the tasks were methodically divided into three discrete components: i) the exportation of the PyTorch model to an ONNX file format, ii) the Building of the TensorRT engine, iii) the deployment phase focused on enhancing inference performance.

C. Dataset Selection

Among all publicly available datasets, we selected COCO and KITTI & BDD100K for testing the inference performance. These datasets are widely used in the evaluation of computer vision models, particularly for object detection tasks like those performed by the YOLO model. Here's an elaborate description of each dataset and why they are well-suited for testing YOLO's performance, especially in real-world object detection scenarios:

Fig. 2: An example of two public datasets: a) BDD100K video dataset; and b) KITTI dataset.

COCO. The COCO dataset contains over 200,000 labeled images with more than 1.5 million object instances across 80 different object categories. It includes various types of annotations, such as object segmentation, keypoint detection, and captioning, which are useful for a range of computer vision tasks. COCO's diverse and complex scenes make it an excellent benchmark for testing the robustness and accuracy of YOLO in real-world conditions.

KITTI. The KITTI dataset is specialized for autonomous driving scenarios. It includes various images collected from vehicle-mounted cameras and annotations for tasks like 2D and 3D object detection, object tracking, and optical flow. The dataset offers real-world driving scenes with various environmental conditions, providing valuable insights into the model's

performance in outdoor, dynamic conditions. Testing YOLO on KITTI allows for the evaluation of its effectiveness in real-time, dynamic environments, which is critical for applications like autonomous driving.

BDD100K. The BDD100K dataset contains over 100,000 video sequences, each lasting 40 seconds featuring diverse driving conditions including various times of the day, weather, and urban/rural settings. The dataset is richly annotated with labels for objects like cars, pedestrians, and traffic signs, necessary for training detection models. These annotations are essential for training and validating object detection models. The dataset includes different types of annotations like image-level tags, object bounding boxes, drivable areas, lane markings, and full-frame instance segmentation.

D. Hardware Setup

Our hardware setup includes an NVIDIA GPU Workstation, made for high-demand tasks, for which we have listed detailed information in the following Fig. 3. It has an Intel Xeon CPU, which is great for handling complex tasks efficiently. This CPU is good at processing multiple tasks at once. The workstation is also equipped with an impressive memory, which helps in doing several things at once and managing big datasets. This is important for data analysis, machine learning, and running many programs or simulations together. A key part of this setup is its four NVIDIA GeForce RTX 2080 Ti graphics cards, each with 11 GB of memory. This setup offers a formidable parallel processing capability, crucial for demanding tasks like deep learning, rendering, and advanced graphical computations. The RTX 2080 Ti is renowned for its exceptional performance in professional applications, particularly in AI and machine learning workloads.

	NVIDIA GPU Workstation
CPU	Intel Xeon E5-2690 v4
GPU	4 x 11 GB GeForce RTX 2080 Ti
Frequency	2.6 GHZ
Core	14
Memory	64 GB
OS	Ubuntu 18.04 LTS

Fig. 3: The configuration of NVIDIA GPU workstation.

E. Software Description

Our project involved the establishment of two distinct software environments, meticulously designed to support specific Road Maps. The initial environment, Designed for RM_T, is created using a robust Python virtual environment framework. This foundational environment integrates several critical components, including TensorFlow, TensorRT, CUDA, CuDNN, ONNX, and tf2onnx, ensuring a balanced and efficient workflow. Subsequently, we developed a second work environment, specifically architectured for RM_P. This environment is built within a Docker image, named Torch-TensorRT, which offers a streamlined and isolated setup. Utilizing this Docker image, we created a Docker container, within which we configured ONNX and an array of essential Python libraries. The following two tables show the software specifications for setting up these two environments respectively.

TABLE I: Software configuration for RM_T.

CUDA	TensorFlow	CuDNN	TensorRT	ONNX
12.1	2.11	8.9	8.6.5	1.15.0

TABLE II: Software configuration for RM_P.

CUDA	PyTorch	ONNX	TensorRT	Torch-TensorRT
12.1	2.2.0	1.15.0	8.6.5	2.2.0

IV. Detailed Description of Workflows

In this section, we provide a more detailed description of workflows that we evaluate in this work. Figure 4 effectively illustrates the procedural workflows for RM_T, which integrate the TensorRT engines with TensorFlow-compatible models. Figure 5 adeptly details the strategies for RM_P that integrate the TensorRT engines with PyTorch-compatible models.

A. RM_T-W_1: TensorFlow-TensorRT

Optimizing object detection models (e.g., YOLOv4) for enhanced performance necessitates a systematic approach, particularly when utilizing TensorFlow-TensorRT. This process bound several pivotal steps designed to align the model with the intended precision modes, including FP32, FP16, and INT8. As shown in Figure 4, the initial step involves training YOLOv4 using TensorFlow. After the training process, the model undergoes conversion into the frozen protocol buffer (PB) format, a critical step for model deployment. The following delineates this multi-stage optimization framework:

1) **Transformation of TensorFlow Model to Frozen (.pb) Format:** Figure 4 illustrates the beginning of our process with a YOLO model, which is trained via TensorFlow. This model is then converted into a 'frozen' version, stored in a .pb file. This initial conversion is a simple yet essential step, setting the foundation for future enhancements with TensorRT. This step is crucial as it lays the groundwork for more complex upgrades and optimizations of the model.
2) **Frozen Model to TensorRT Conversion:** After getting the frozen model, the focus shifts to exploiting TensorRT's capabilities for boosting model performance. This phase entails transitioning the TensorFlow model into a TensorRT-compatible operational format.
   - Conversion Parameters Setup: We set the maximum workspace size to 4GB for TensorRT optimizations and specified the precision mode for conversion.
   - Conversion Process: For each precision mode (FP32, FP16, or INT8), we adjust the *use_calibration* parameter accordingly. Here, Calibration is crucial for INT8 conversion for accuracy retention. Then, we initialize a TensorRT graph converter with the specified input model directory and conversion parameters. In scenarios employing INT8 precision, the *use_calibration* parameter becomes essential. When enabling it to (True), it instigates the creation of a calibration graph, which is instrumental in establishing the quantization range. This stage necessitates a calibration dataset to facilitate an accurate conversion of the TensorFlow model into the TensorRT framework. Conversely, setting this parameter to False signals the expectation of existing quantization nodes for each tensor (excluding those

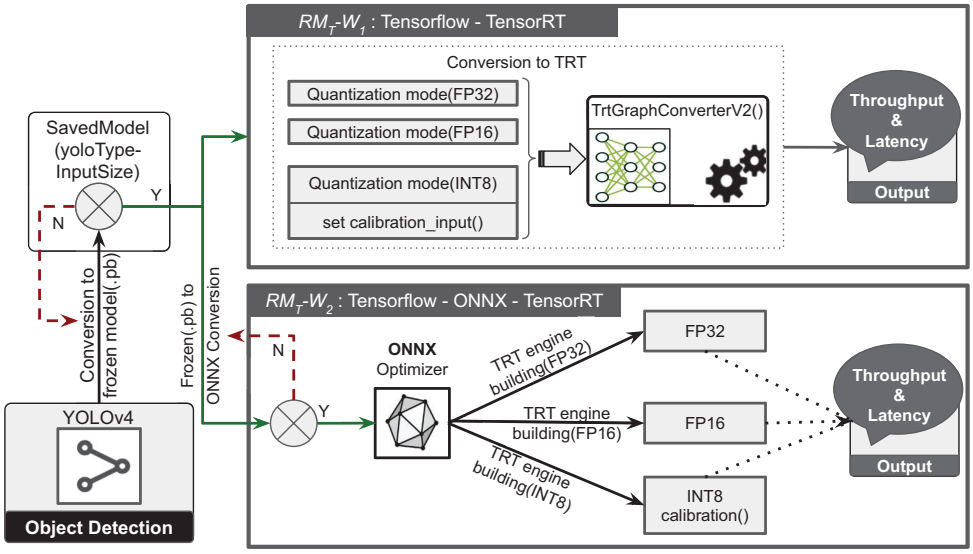


Fig. 4: An overview of Road Maps 1, RM_T-W_1 and RM_T-W_2 as well as the software tools used in each stage.

subjected to fusion). A lack of requisite range in this context is likely to result in errors.

- Model Conversion: Finally, we execute the conversion process using *TrtGraphConverterV2*. For INT8, it utilizes the calibration_input function to calibrate the model. The converted model is saved in a specified directory, indicating the YOLO type, TensorRT precision mode, and input size.

3) **Inference Performance:** For performing the inference we load the TRT saved model from the directory to evaluate the inference. To be noted, for each of the precision modes we selected specific files to evaluate the overall performance.
Among three conversions, FP32 is the Highest among the three and uses 32-bit floating-point numbers. It offers a balance between accuracy and performance but may not provide significant speedup compared to native TensorFlow models. Precision-FP16, offers faster computation and reduced model size, with a slight compromise in accuracy, compared to FP32. INT8 (8-bit Integer) provides the highest speedup and efficiency, significantly reducing model size and computational overhead.

B. RM_T-W_2*: TensorFlow-ONNX-TensorRT*

In this workflow, once the training phase is completed, two key parameters are established and locked in the batch size, which dictates the number of samples processed together and the precision level, which can be FP32, FP16, or INT8. Following this, the trained model undergoes optimization through the TensorRT optimizer. This process results in the creation of an optimized runtime environment, commonly referred to as a plan. The plan is encapsulated in a .plan file, which represents a serialized format of the TensorRT engine, essentially a compact, efficient representation of the model optimized for deployment. To utilize this model for inference tasks, the plan file must be deserialized, into a format that the TensorRT runtime can execute. This deserialization is a crucial step for the model to be operational and perform inference using the capabilities of TensorRT. The comprehensive overview of Figure 4, which details the TensorFlow-ONNX-TensorRT pipeline, is presented now in detail.

1) **Transformation of TensorFlow Model to Frozen(.pb) Format:** As presented in Figure 4, the process begins with the Yolo model that has already been trained using TensorFlow. This model is then turned into a 'frozen' version, which is saved in a .pb file format. This first change is a basic step that prepares the model for further upgrades using TensorRT.
2) **File (.pb) to ONNX conversion:** The next step in the process is the conversion of the .pb model into the ONNX format, which requires the initial installation of tf2onnx. Once tf2onnx is set up, the model conversion from .pb to ONNX can be achieved in two distinct ways. The first approach is to utilize the command line - a more direct and script-based method. Alternatively, the second approach involves using the Python API, which offers a more programmable and flexible interface for this task. In our case, we opted for the Python API method to carry out this conversion to ONNX format. Upon the successful generation of the ONNX file, we conducted a thorough verification of our(.onnx) file before creating the TensorRT engine using it. This verification process serves as a preliminary assurance, confirming that the model is adequately prepared and suitable for the subsequent phase of engine building.
3) **TensorRT Engine Creation:** To create the TRT engine from the onnx file we followed NVIDIA's document thoroughly. We start the engine-building process by instantiating A *TRT_LOGGER* which is significant for monitoring the behavior of TensorRT operations and debugging. Then, we create a *trt_runtime* object using the logger, which is necessary for creating and running the TensorRT engine. Sequentially, we create a function to build the engine; where, the function initializes a TensorRT builder, network, and configuration, along with an ONNX parser,

and sets the maximum workspace size for the builder. This phase also involved configuring the network's input shape and meticulously parsing the ONNX file, with a robust error-reporting mechanism for any parsing failures. The culmination of this process was the conversion of the optimized model into a serialized *(.plan)* format, signifying the successful creation of the TensorRT engine. We diligently saved the engine across various configurations, such as FP32, FP16, and INT8, with a special focus on implementing a calibration process for the INT8 engine to preserve accuracy.

4) **Deploy for Inference:** Before inference, we deserialized the .plan file to load the optimized model into the TensorRT runtime. Then we use the deserialized model to run inference on new data. This step is significantly faster and more efficient, especially on NVIDIA GPUs. After successfully creating the TRT engine for FP32, FP16, and INT8 we run inference to observe the performance. For each of the precision modes, we then deserialize the .plan file to load the optimized model into the TensorRT runtime. To run inference, from our saved TRT engine, we follow some important steps, including i) The inference process starts with pre-processing images from datasets. ii) Then, we allocate Input and output buffers on the GPU. iii) After allocating I/O buffers, we transfer them from the host to these input buffers on the GPU. iv) Promptly, the GPU performs the inference process. v) Finally, inference results are then transferred back from the GPU to the host, and results are reshaped as required.

C. RM$_P$-W_1: Torch-TensorRT

In RM$_P$, we enhanced the inference of the object detection model (YOLO) by employing Torch-TensorRT, which is a collaborative effort of Meta AI combining PyTorch with NVIDIA's TensorRT [51]. Torch-TensorRT is designed to optimize and run compatible network segments while allowing PyTorch to manage the execution of the rest of the network graph. Figure 5 presents this workflow's detailed work.

1) **Model loading & Dataset Selection:** We start the RM$_P$-W_1 by loading a pre-trained YOLOv5 small model, from Ultralytics using the torchhub.load function. This function is a part of PyTorch's hub module, designed to facilitate the easy loading and use of pre-trained models. The selection of the YOLOv5 small model is likely driven by its balance of speed and accuracy, making it an ideal choice for various real-time object detection applications. At this stage, we also specify the dataset with specific transformations (resizing images to 416×416 and converting them to tensors).
2) **Model Tracing:** Upon resizing the images to the specified dimensions, we proceed to transform our model into TorchScript modules utilizing PyTorch's Just-In-Time (JIT) compiler. This crucial step converts the PyTorch model into a TorchScript format, a prerequisite for subsequent optimization using TensorRT. Consequently, the TorchScript model is stored in the *traced_model* (referenced in Figure 5) and is preserved on the disk. For optimization with 32-bit floating point precision, we compiled the *traced_model* using TensorRT and specified the FP32 precision. This process entails defining both the input shape and the data type as torch.float32. In a parallel approach, we also developed an alternative version of the model employing FP16 precision. This was achieved by altering the data type to *torch.half*. While Torch-TensorRT does not directly provide INT8 precision mode without adjusting the model compilation to use INT8 precision we limit our exploration for Torch-TensorRT's precision modes within full-precision FP32 and half precision FP16.
3) **Inference Performance:** Afterward, we rerun the benchmark function for both the FP32 and FP16 optimized models to measure and compare their performance against the original model. We start the benchmarking procedure by synchronizing the CUDA device to ensure it starts from a consistent state. For each image, we record the start time using *torch.cuda.Event* and execute model to evaluate the performance of inference on each image from the dataset. After that, we again record the end time and synchronize the CUDA device to ensure accurate timing measurement. To ensure comprehensive evaluation, we repeat the benchmark function twice: once for the original model and subsequently for each FP32 and FP16 optimized model. This rigorous approach allows us to methodically measure and compare their performance metrics against the baseline model.
Precision. FP32 uses 32-bit floating-point numbers, offering higher precision. FP16 uses 16-bit, which is less precise but requires less memory and computational resources.
Performance. FP16 often allows for faster computation than FP32 because it uses smaller data types. This can be particularly beneficial in environments in edge devices where memory bandwidth is a bottleneck.
Memory Usage. FP16 models generally use less memory compared to FP32, which can be advantageous for deploying models on memory-constrained devices.

D. RM$_P$-W_2: Torch-ONNX-TensorRT

In this workflow, the journey also begins by loading a pre-trained YOLOv5 small model, from Ultralytics using the torchhub.load function, as delineated in our previous workflow RM$_P$-W_1. Figure 5 provides a schematic representation of RM$_P$, which is dedicated to enhancing the inference performance of the Yolo model. We now delve into a detailed illustration of W_2, as illustrated in the comprehensive overview of Figure 5.

1) **ONNX Conversion:** After loading our model the subsequent step involves the transformation of the YOLOv5s model into the ONNX format. This conversion is essential for ensuring the model's compatibility with a broader range of platforms and optimizing frameworks. The *torch.onnx.export* function is employed for this purpose, effectively translating the PyTorch model into an ONNX model while preserving its architecture and learned weights. Following the conversion, the model undergoes a thorough verification process using ONNX's checker tool, invoked via *onnx.checker.check_model*. Once the model has passed the verification stage, the final step involves

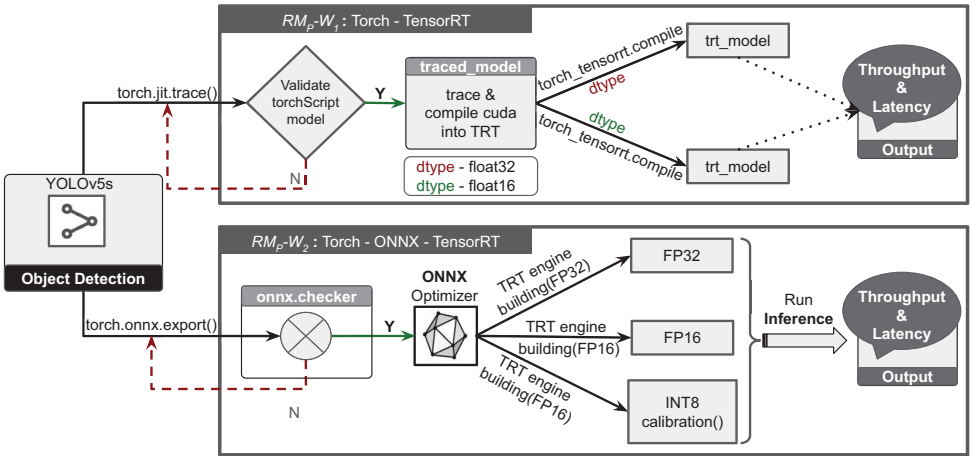


Fig. 5: An overview of Road Maps 2, RM$_P$-W_1 and RM$_P$-W_2 as well as the software tools used in each stage.

saving the now-validated yolo.onnx model and ready-to-use in the environments that support the ONNX format.

2) **TensorRT Engine Building:** To build the engine for specific precision mode (e.g., FP32, FP16, or INT8). First, we import the necessary libraries and set up the TensorRT logger. The Engine Building procedure is listed below. We divide the procedure into two parts: i) Generic Engine building and ii) what we did differently to build the engine for each specific Precision Mode.

Generic Engine building. We successfully optimize our object detection model by executing a series of strategic steps. Initially, we establish the groundwork by initializing the TensorRT builder and configuring its settings. Following this, we optimize our computational resource allocation by setting the maximum workspace size. Our efforts then focus on defining the network, where we utilize the explicit batch flag to enhance batch processing capabilities. Further, we advance to parse the ONNX model, during which we meticulously identify and mark the crucial output layer, ensuring the precision of our model's outputs. Finally, we achieved a significant milestone in building a serialized network. This is accomplished through the integration of the TensorRT builder, our defined network, and the builder configuration, culminating in an optimized and efficient model ready for deployment. We define a function to serialize the engine and save it to a file. In the main execution block where the ONNX model path and engine path are defined, the engine is built and then deserialized for later use.

Specific Differences for Each Precision Mode. FP32 (default precision): The FP32 code block is the default setting and doesn't explicitly set a precision flag. On the other hand, FP16 (Half Precision) is enabled by setting the FP16 flag in the builder configuration: *builder_config.set_flag(trt.BuilderFlag.FP16)*. To be noted, the FP16 mode offers a balance between performance and accuracy. INT8 (Integer Precision) is enabled by setting the INT8 flag: *builder_config.set_flag(trt.BuilderFlag.INT8)*. We did an additional calibration process using the dataset to maintain accuracy, with the ImageBatchStream and Calibrator classes. It offers the highest performance, especially in terms of throughput and latency, but it requires careful calibration to maintain the model's accuracy.

3) **Deployment of the Inference Performance:** We measured the inference performance of both inference throughput and inference execution time by creating three benchmark functions. The first benchmark function *(benchmark_trt_FP32 (engine, dummy_input)* we implemented to get the inference of engine for floating point 32. In this benchmark function, we passed the engine instead of the model and changed the benchmark function accordingly. i) The implementation of the first step involves creating an execution context using NVIDIA's CUDA for running TensorRT inference. This context is essential for managing resources and controlling the inference pipeline. ii) Then, A dummy input tensor, which is the input image, assumed to be a PyTorch tensor, is converted to a NumPy array *(dummy_input_np)*, and its shape is used to determine the input and output sizes for memory allocation. The input *(input_shape)* and expected output *(output_shape)* dimensions are defined, with an output shape of (1, 25200, 85). iii) We allocate Memory space on the GPU for both the input *(d_input)* and output *(d_output)* using CUDA's mem_alloc function. The sizes of these allocations are computed based on the product of the respective shapes and the size of a float32 data type. iv) Later, A CUDA stream is initialized to manage the sequence of operations asynchronously, allowing for concurrent execution of operations and data transfers. v) The input data *(h_input)* is flattened to np.float32. A warm-up loop executes to ensure the GPU is ready for benchmarking. The actual benchmarking involves repeatedly transferring the input data to the GPU, executing the model inference, and transferring the output data back to the CPU. This process is timed for each run within a loop of *n_runs* iterations. vi) After each inference run, we record the time. These times we use to calculate the average inference time and frames per second (FPS), which are crucial metrics for understanding the model's performance. vii) Finally, the CUDA stream was synchronized to ensure all operations were

completed. The active optimization profile was set to 0, likely to switch back to a default or specific configuration. Similarly, we implemented benchmark functions *(benchmark_trt_FP16)* and *(benchmark_trt_INT8*), which are designed to evaluate the performance of a deep learning model using NVIDIA's TensorRT with FP16 and INT8 precision. The key steps to implement these functions are pretty similar. In *(benchmark_trt_FP16)*, the primary difference is the precision mode used for inference. This function benchmarks the model in FP16, which can offer faster performance and reduced memory usage compared to FP32, especially on GPUs with Tensor Cores optimized for FP16 calculations. In *(benchmark_trt_INT8)*, this function benchmarks the engine in INT8 mode, which is faster and more memory-efficient than FP32 and FP16, especially on GPUs optimized for INT8 calculations.

V. Experimental Results and Analysis

In this section, we present our inference performance results in three subsections, discussing the computation metrics for the inference performance, experimental results, and analysis for RM_T (W_1-W_2) and RM_P (W_1-W_2).

A. Computation Metrics for Inference Performance

In this subsection, we present how we computed the inference performance (Execution Time and Throughput) of the model and TensorRT Engines. For YOLO and similar real-time object detection systems, both these metrics are critical: A lower inference execution time ensures that each frame is processed quickly, which is vital for making immediate decisions based on the latest available data. A higher inference throughput means the system can handle more data at any given time, which is crucial for maintaining real-time performance even under heavy data loads. Therefore, balancing these two metrics is a key aspect of designing and tuning real-time object detection systems.

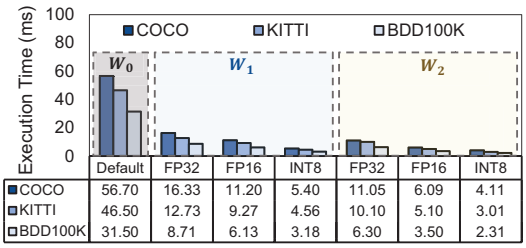
Inference Execution Time. It refers to the amount of time it takes for the object detection system to process an input (like a frame from a video) and produce an output (in this case, the detection of objects within the frame). A shorter inference execution time means the system can analyze and respond to its environment more rapidly. This metric is usually measured in milliseconds (ms). In our experiments, we measured the end-to-end execution time of our model's forward inference cycle, explicitly focusing on the time taken for the model to process input and produce output. This measurement deliberately excluded time spent on data retrieval, model initialization, and input pre-processing to accurately measure the raw inference performance of the model and engines. For RM_T, the TensorFlow model (YOLOv4) first subjects to a series of 'warm-up' runs, a critical step in ensuring that the TensorFlow execution graph is fully optimized and any JIT (Just-In-Time) compilation or GPU initialization processes are completed. This approach mirrors the methodology applied in the PyTorch analysis, aiming to neutralize any start-up anomalies that might skew the performance data. Following this, the model was run for a set number of iterations, with the execution time for each iteration being precisely captured using the time module. These recorded times, marking the period from the initiation to the completion of the inference process, provided the raw data needed to calculate the average inference time and FPS. For the PyTorch model (RM_P), we underwent a similar evaluation. The experiment commenced with a 'warm-up' phase, a common practice in deep learning benchmarks to stabilize performance metrics. Here, we executed the model inference repetitively for a predefined number of times (50 in this case) before proceeding to the actual measurements. This warm-up procedure ensures that any initial latency associated with model loading or CUDA operations is mitigated, thus providing a more consistent and reliable measurement during the benchmarking phase. Following this, we conducted 100 inference runs. In each run, precise time measurements were captured using Python's time function, marking the start and end of the model's inference process. For every iteration, we recorded the time immediately before initiating the model inference *(start_time)* and immediately after its completion *(end_time)*, ensuring synchronization with the GPU using *torch.cuda.synchronize()* to obtain accurate timing. The difference between these two timestamps provided the duration of each inference, which we added to our timings list.

Inference Throughput. This refers to the number of inputs (e.g., video frames) the object detection system can process in a given amount of time. It's a measure of the overall processing capacity of the system. Throughput is often measured in frames per second (FPS). For both TensorFlow and PyTorch models, the inference throughput was evaluated by processing multiple data inputs in batches. This methodology closely resembles real-world scenarios where models often handle several inputs simultaneously, making it a pertinent measure of performance. We initialized the model in an inference mode and prepared a batch of inputs, calibrated to the model's input specifications. The experiment ran these batches through the model repeatedly, ensuring the system's computational resources were maximally utilized. The total number of inferences (the number of batches multiplied by the batch size) processed in a given time frame was then recorded. By dividing this number by the total time taken to process these inferences, we arrived at the throughput rate, measured in inferences per second. This rate provided a quantifiable measure of the model's capacity to process data under load.

B. Experimental Results of RM_T (W_1-W_2)

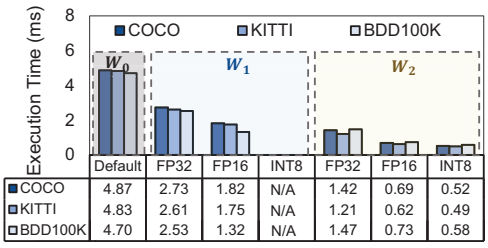
In this subsection, we present the results for both of the workflows and our observations based on the results.

Time-critical systems, such as autonomous vehicles, having a minimum execution time is often more important than achieving high throughput due to the urgent nature and immediacy. For such scenarios, for real-time applications, a minimum batch size is often used to minimize latency. During inference, batch size determines how many images the model processes at once. Larger batch size can increase throughput (number of images processed per second) but also increase execution time (time to process each individual image). In our Experiment, we varied the size of the batch between 1 to 8 and the image size of 416 $\times$ 416. In Fig. 6 (a), we analyze and compare the inference performance of two distinct configurations: W_1-TensorFlow with TensorRT integration and W_2-TensorFlow with ONNX-TensorRT conversion. We also



	Default	FP32	FP16	INT8	FP32	FP16	INT8
COCO	56.70	16.33	11.20	5.40	11.05	6.09	4.11
KITTI	46.50	12.73	9.27	4.56	10.10	5.10	3.01
BDD100K	31.50	8.71	6.13	3.18	6.30	3.50	2.31

(a) The inference execution time of RM_T.



	Default	FP32	FP16	INT8	FP32	FP16	INT8
COCO	4.87	2.73	1.82	N/A	1.42	0.69	0.52
KITTI	4.83	2.61	1.75	N/A	1.21	0.62	0.49
BDD100K	4.70	2.53	1.32	N/A	1.47	0.73	0.58

(b) The inference execution time of RM_P.

Fig. 6: The inference execution time for RM_T and RM_P.

compare them with the default workflow without TensorRT on three datasets, i.e., COCO, KITTI, and BDD100K. In our first workflow W_1, which involves TensorFlow integrated directly with TensorRT, we observed a substantial reduction in execution time, ranging between 3.5 to 7 times faster than the baseline. Concurrently, from Fig. 7 (a), there was an appreciable increase in throughput, achieving an improvement in the range of 2 to 3.5 times compared to W_0. This underscores the efficiency gains that can be achieved by directly integrating TensorFlow with TensorRT. The second workflow, RM_T-W_2 (TensorFlow-ONNX-TensorRT), employs a different approach by incorporating an ONNX intermediary between TensorFlow and TensorRT. This method yielded even more pronounced improvements. We recorded a decrease in execution time by approximately 5 to 9 times faster than the default workflow W_0(shown in Fig. 6 (a)). As shown in Fig. 7 (a), the throughput experienced an uplift, ranging from 2.5 to 4 times. These findings highlight the effectiveness of the RM_T-W_2 workflow in optimizing both the speed and processing capacity of the models. Our results demonstrate that both configurations offer significant advantages in terms of execution time efficiency and throughput enhancement. However, RM_T-W_2 stands out as an effective approach for improving model performance, underscoring the potential of this method in enhancing computational efficiency in real-world model inference tasks.

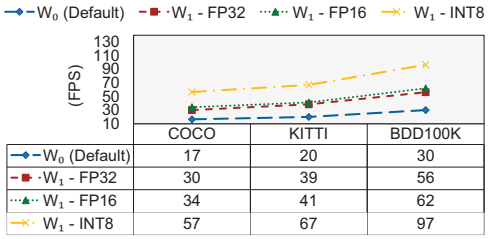
Observation 1. In our comprehensive analysis, it becomes evident that the workflow RM_T-W_1, TensorFlow-TensorRT emerges as the fine choice for executing inference on YOLO models. This workflow perfectly balances execution speed and efficiency, outperforming others regarding inference execution. However, it's important to note that while the throughput in RM_T-W_1, shows a marginal improvement, it does not match the significant enhancements observed in inference execution times. This distinction is particularly crucial in time-sensitive environments, such as real-world applications, where expedited execution time holds more value than throughput efficiency. Therefore, in evaluating the overall inference performance, we conclude that the results are satisfactory. An intriguing observation is made with the INT8 precision. While it significantly reduces the inference time and increases throughput, we notice a discernible decline in the accuracy of the object detection model.

Observation 2. Our analysis reveals that RM_T-W_2, which employs the Tensorflow-ONNX-TensorRT pipeline, outperforms RM_T-W_1 in terms of both inference execution time and throughput. This advancement is notable as it achieves a reduction in latency for FP32 that is comparable to what was attained for FP16 in RM_T-W_1, and this is accomplished without any significant compromise in accuracy. Furthermore, RM_T-W_2 also surpasses the first workflow in throughput efficacy. However, it is important to mention that a slight decrease in accuracy was observed for the INT8 precision, though this reduction is not as pronounced as that seen in RM_T-W_1. This observation shows that while RM_T-W_2 offers enhanced performance in key areas, the choice of precision plays a critical role in balancing accuracy with efficiency.

C. Experimental Results of RM_P (W_1-W_2)

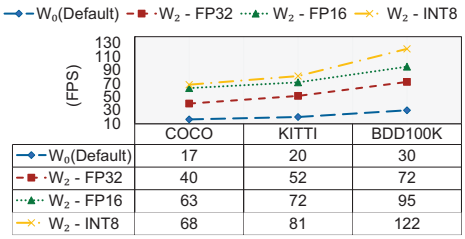
As shown in Fig. 6 (b) and Fig. 7 (b), we analyze and compare the inference performance of Torch-TensorRT integration and Torch with ONNX-TensorRT conversion.

We find that RM_P-W_1 (Torch-TensorRT) reveals a noteworthy improvement in the model's performance following its compilation with Torch-TensorRT. Initially, the model exhibited an inference execution time of 4.87 milliseconds (ms), shown in Fig. 6 (b), and an inference throughput of 202.32 frames per second (FPS) which is shown in Fig. 7 (b). This performance enhances significantly post-compilation. For W_1, in the context of Floating Point 32 (FP32) precision, there was a doubling increment in the inference throughput, along with a roughly 50% reduction in inference execution time. This indicates a marked improvement in processing efficiency. Furthermore, when the model operated under Floating Point 16 (FP16, torch.half) precision in W_1, the enhancements were even more pronounced. The inference execution time decreased by approximately 67% relative to the default model W_0, while the inference throughput saw an increase of 2.8 times. These improvements highlight the effectiveness of Torch-TensorRT in optimizing the model's performance, particularly in reducing execution times and increasing throughput, which are critical factors in real-time applications. In our comprehensive evaluation of RM_P-W_2(Torch-ONNX-TensorRT), we have documented substantial advancements in inference performance. This improvement is particularly pronounced when utilizing FP16 precision, achieved without compromising the model's accuracy. Our initial default inference execution time is 4.87 milliseconds in W_0 Fig. 6 (b) and throughput of 205.35 frames per second (FPS) from Fig. 7 (b). Upon transitioning to FP32 precision for W_2, we observed a marked improvement. the inference execution time was reduced by 71%, and the inference throughput increased by 3.5 times compared to our default model W_0. This enhancement in performance indicates the efficiency gains achieved through precision optimization. However, the most striking improvements were evident when we switched to FP16 precision for W_2. In this scenario, we recorded an 86%



	COCO	KITTI	BDD100K
W_0 (Default)	17	20	30
W_1 - FP32	30	39	56
W_1 - FP16	34	41	62
W_1 - INT8	57	67	97

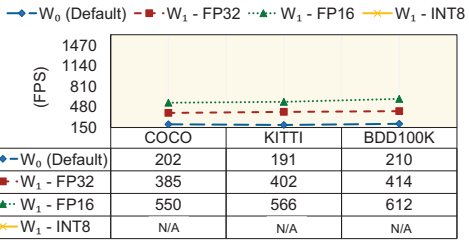
(i) The throughput comparison between W_0 and W_1



	COCO	KITTI	BDD100K
W_0(Default)	17	20	30
W_2 - FP32	40	52	72
W_2 - FP16	63	72	95
W_2 - INT8	68	81	122

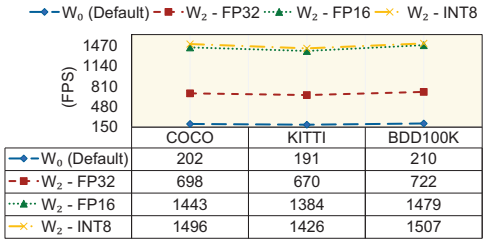
(ii) The throughput comparison between W_0 and W_2

(a) The inference throughput of RM_T.



	COCO	KITTI	BDD100K
W_0 (Default)	202	191	210
W_1 - FP32	385	402	414
W_1 - FP16	550	566	612
W_1 - INT8	N/A	N/A	N/A

(i) The throughput comparison between W_0 and W_1



	COCO	KITTI	BDD100K
W_0 (Default)	202	191	210
W_2 - FP32	698	670	722
W_2 - FP16	1443	1384	1479
W_2 - INT8	1496	1426	1507

(ii) The throughput comparison between W_0 and W_2

(b) The inference throughput of RM_P.

Fig. 7: The inference throughput for RM_T and RM_P.

reduction in execution time alongside a 7 times increase in throughput. This significant boost in performance underscores the effectiveness of FP16 precision in enhancing our model's speed and efficiency. Further experiments were conducted with 8-bit INT precision calibration in W_2. Here, we noted that the performance outcomes were not substantially different from those observed with FP16 precision. This finding suggests that while 8-bit INT precision offers benefits, FP16 remains the more impactful choice in terms of improving inference performance. Our analysis conclusively demonstrates that to optimize the inference performance in the context of RM_P-W_2, FP16 precision emerges as the most effective option. It strikes an optimal balance between enhancing performance efficiency and maintaining accuracy, making it a preferred choice for models prioritizing both aspects.

Observation 3. In examining the inference performance of RM_P-W_1, it has been observed that the workflow benefits from a notable enhancement in efficiency. There is a considerable reduction in inference latency, exhibiting a promising decrease of approximately 70 percent. Concurrently, this improvement is accompanied by a threefold increase in inference throughput. When evaluating the system across both precision modes - FP32 and FP16 - a commendable equilibrium between inference performance and accuracy is evident. This balance is crucial for optimizing system efficiency without compromising on the accuracy of outcomes.

Observation 4. It is important to highlight that RM_P-W_2, operating in the FP16 precision mode, demonstrates superior performance relative to all other evaluated workflows, particularly in terms of inference throughput and execution time, when contrasted with accuracy metrics. This workflow achieves a sevenfold increase in inference throughput, coupled with an almost sixfold reduction in execution time. Additionally, when considering the FP16 precision mode, the observed decrease in accuracy, as compared to the original model, is very minimal.

VI. Conclusion

In this paper, we conduct a thorough comparative analysis of the inference performance of an object detection model using TensorFlow and PyTorch-compatible frameworks accelerated by TensorRT for resource-limited SDVs. Based on the outcomes of our evaluation, we observe TensorRT's precision mode (e.g., FP32, FP16, and INT8) performance in terms of latency and throughput after optimization.

Our research includes a detailed examination of each workflow's strengths and weaknesses based on precision modes to enhance our understanding of time-critical systems. Our results show that for both our TensorFlow and PyTorch Framework, the workflow of ONNX to TensorRT conversion has the best result for improving model inference. Upon evaluating different precision modes, we observed that The FP16 precision mode has the best inference performance result in terms of throughput, and latency. Although the INT8 precision mode demonstrated commendable inference results, our observation indicates that the FP16 mode maintains an optimal balance between inference performance and accuracy. This equilibrium is vital in real-world applications of SDVs, where precision and reliability are supreme concerns. To conclude, for applications demanding high inference performance with limited computational resources, we recommend the ONNX to TensorRT with the FP16 precision mode conversion approach as an optimal solution.

Acknowledgement

This work is supported in part by the National Science Foundation (NSF) grant, CNS-2103604, CNS-2140346, CNS-2231523, and CNS-2348151, as well as Commonwealth Cyber Initiative grant HC-3Q24-048.

REFERENCES

[1] S. Lu and W. Shi, "Vehicle computing: Vision and challenges," *Journal of Information and Intelligence*, vol. 1, no. 1, pp. 23–35, 2023.

[2] H. Raposo, J. T. Farinha, I. Fonseca, and L. A. Ferreira, "Condition monitoring with prediction based on diesel engine oil analysis: A case study for urban buses," in *Actuators*, vol. 8, no. 1. MDPI, 2019, p. 14.

[3] S. C. Subramanian, S. Darbha, and K. Rajagopal, "A diagnostic system for air brakes in commercial vehicles," *IEEE transactions on intelligent transportation systems*, vol. 7, no. 3, pp. 360–376, 2006.

[4] I. A. Ntousakis, I. K. Nikolos, and M. Papageorgiou, "Optimal vehicle trajectory planning in the context of cooperative merging on highways," *Transportation research part C: emerging technologies*, vol. 71, pp. 464–488, 2016.

[5] S. Lu, R. Zhong, and W. Shi, "Teleoperation technologies for enhancing connected and autonomous vehicles," in *2022 IEEE 19th International Conference on Mobile Ad Hoc and Smart Systems (MASS)*. IEEE, 2022, pp. 435–443.

[6] Y. Wang, Y. Zheng, and Y. Xue, "Travel time estimation of a path using sparse trajectories," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 25–34.

[7] I. Karafyllis, D. Theodosis, and M. Papageorgiou, "Nonlinear adaptive cruise control of vehicular platoons," *International Journal of Control*, vol. 96, no. 1, pp. 147–169, 2023.

[8] S. Lu, Y. Yao, and W. Shi, "CLONE: Collaborative learning on the edges," *IEEE Internet of Things Journal*, vol. 8, no. 13, pp. 10 222–10 236, 2020.

[9] D. Du, Y. Qi, H. Yu, Y. Yang, K. Duan, G. Li, W. Zhang, Q. Huang, and Q. Tian, "The unmanned aerial vehicle benchmark: Object detection and tracking," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 370–386.

[10] K. V. T. Agullo, J. P. A. Sasis, and J. T. Sese, "Air purification system for air quality monitoring in-vehicle," in *2022 International Electronics Symposium (Ies)*. IEEE, 2022, pp. 136–141.

[11] Y. Zhou and K. Yang, "Exploring tensorrt to improve real-time inference for deep learning," in *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. IEEE, 2022, pp. 2011–2018.

[12] G. Cheng, X. Yuan, X. Yao, K. Yan, Q. Zeng, X. Xie, and J. Han, "Towards large-scale small object detection: Survey and benchmarks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.

[13] J. Zhou, K. Feng, W. Li, J. Han, and F. Pan, "TS4Net: Two-stage sample selective strategy for rotating object detection," *Neurocomputing*, vol. 501, pp. 753–764, 2022.

[14] S. Lu, X. Yuan, and W. Shi, "Edge compression: An integrated framework for compressive imaging processing on cavs," in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2020, pp. 125–138.

[15] S. Liang, H. Wu, L. Zhen, Q. Hua, S. Garg, G. Kaddoum, M. M. Hassan, and K. Yu, "Edge yolo: Real-time intelligent object detection system based on edge-cloud cooperation in autonomous vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 12, pp. 25 345–25 360, 2022.

[16] A. Kozlov and D. Osokin, "Development of real-time adas object detector for deployment on cpu," in *Intelligent Systems and Applications: Proceedings of the 2019 Intelligent Systems Conference (IntelliSys) Volume 1*. Springer, 2020, pp. 740–750.

[17] C. Ding, S. Wang, N. Liu, K. Xu, Y. Wang, and Y. Liang, "Req-yolo: A resource-aware, efficient quantization framework for object detection on fpgas," in *proceedings of the 2019 ACM/SIGDA international symposium on field-programmable gate arrays*, 2019, pp. 33–42.

[18] M. Subramanian, K. Shanmugavadivel, and P. Nandhini, "On fine-tuning deep learning models using transfer learning and hyper-parameters optimization for disease identification in maize leaves," *Neural Computing and Applications*, vol. 34, no. 16, pp. 13 951–13 968, 2022.

[19] L. Xu, W. Yan, and J. Ji, "The research of a novel wog-yolo algorithm forautonomous driving object detection," 2022.

[20] J. Zhang, P. Wang, Z. Zhao, and F. Su, "Pruned-yolo: Learning efficient object detector using model pruning," in *International Conference on Artificial Neural Networks*. Springer, 2021, pp. 34–45.

[21] S. Dulepet, P. Maji, M. Harsh, and K. Washabaugh, "Deploying a scalable object detection inference pipeline part, 2020. erişim tarihi: 21 ralık 2020."

[22] S. Imambi, K. B. Prakash, and G. Kanagachidambaresan, "PyTorch," *Programming with TensorFlow: Solution for Edge Computing Applications*, pp. 87–104, 2021.

[23] B. Pang, E. Nijkamp, and Y. N. Wu, "Deep learning with TensorFlow: A review," *Journal of Educational and Behavioral Statistics*, vol. 45, no. 2, pp. 227–248, 2020.

[24] X. Xia, J. Li, J. Wu, X. Wang, X. Xiao, M. Zheng, and R. Wang, "TRT-ViT: TensorRT-oriented vision transformer," *arXiv preprint arXiv:2205.09579*, 2022.

[25] "TensorFlow," https://www.tensorflow.org/, 2015.

[26] "TensorFlow-Extended," https://www.tensorflow.org/tfx, 2017.

[27] "Pytorch," https://pytorch.org/, 2016.

[28] "Onnx," https://onnx.ai/, 2017.

[29] "NVIDIA TensorRT," https://developer.nvidia.com/tensorrt, 2016.

[30] V. Zunin, "Intel openvino toolkit for computer vision: Object detection and semantic segmentation," in *2021 International Russian Automation Conference (RusAutoCon)*. IEEE, 2021, pp. 847–851.

[31] G. Jocher, A. Chaurasia, A. Stoken, J. Borovec, Y. Kwon, J. Fang, K. Michael, D. Montes, J. Nadar, P. Skalski *et al.*, "ultralytics/yolov5: v6. 1-tensorrt, tensorflow edge tpu and openvino export and inference," *Zenodo*, 2022.

[32] L. Shen, H. Tao, Y. Ni, Y. Wang, and V. Stojanovic, "Improved yolov3 model with feature map cropping for multi-scale road object detection," *Measurement Science and Technology*, vol. 34, no. 4, p. 045406, 2023.

[33] O. Shafi, C. Rai, R. Sen, and G. Ananthanarayanan, "Demystifying tensorrt: Characterizing neural network inference engine on nvidia edge devices," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2021, pp. 226–237.

[34] H.-K. Hong and J.-W. Jeon, "An optimized multi-object tracking with tensorrt," in *2023 International Technical Conference on Circuits/Systems, Computers, and Communications (ITC-CSCC)*. IEEE, 2023, pp. 1–4.

[35] H. Wang, C. Shi, S. Shi, M. Lei, S. Wang, D. He, B. Schiele, and L. Wang, "Dsvt: Dynamic sparse voxel transformer with rotated sets," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 13 520–13 529.

[36] F. Jiang, C. Tu, G. Zhang, J. Li, H. Huang, J. Lin, D. Feng, and J. Pu, "Revisiting multi-modal 3d semantic segmentation in real-world autonomous driving," *arXiv preprint arXiv:2310.08826*, 2023.

[37] Y. Huang, Y. Zhang, B. Feng, X. Guo, Y. Zhang, and Y. Ding, "A close look at multi-tenant parallel cnn inference for autonomous driving," in *IFIP International Conference on Network and Parallel Computing*. Springer, 2020, pp. 92–104.

[38] D.-J. Shin and J.-J. Kim, "A deep learning framework performance evaluation to use yolo in nvidia jetson platform," *Applied Sciences*, vol. 12, no. 8, p. 3734, 2022.

[39] E. Jeong, J. Kim, S. Tan, J. Lee, and S. Ha, "Deep learning inference parallelization on heterogeneous processors with tensorrt," *IEEE Embedded Systems Letters*, vol. 14, no. 1, pp. 15–18, 2021.

[40] 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*, 2020, pp. 48–53.

[41] R. Girshick, "Fast R-CNN," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.

[42] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.

[43] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[44] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2961–2969.

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

[46] C.-Y. Fu, W. Liu, A. Ranga, A. Tyagi, and A. C. Berg, "DSSD: deconvolutional single shot detector," *arXiv preprint arXiv:1701.06659*, 2017.

[47] J. Redmon and A. Farhadi, "YOLO9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7263–7271.

[48] Redmon, Joseph and Farhadi, Ali, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[49] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "YOLOv4: Optimal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934*, 2020.

[50] D. Wang and D. He, "Channel pruned YOLO V5s-based deep learning approach for rapid and accurate apple fruitlet detection before fruit thinning," *Biosystems Engineering*, vol. 210, pp. 271–281, 2021.
[51] "Torch-TensorRT," https://github.com/pytorch/TensorRT, 2023.