UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Development and evaluation of a lab-based smart city testbed with dynamic reconfiguration capabilities

Special Subject

Tsatrafili Eirini Eleutheria 03203

Supervisor:
Mpellas Nikolaos

UNIVERSITY OF THESSALY, 2025

Contents

Introduction

Accurate and efficient object detection is a critical component of many real-world applications, particularly in traffic monitoring and smart city research. This project focuses on developing and evaluating a lab-based testbed to assess various object detection methods. The testbed consists of an Intel RealSense camera for real-time video capture, paired with a Raspberry Pi as the primary processing unit. The system was used to test and compare the performance of different object detection models in a lab-based environment.

Several object detection models were evaluated, including Faster R-CNN ResNet-50 FPN V2, YOLO, FCOS ResNet-50, Faster R-CNN, and SSD-Lite. Among these, Faster R-CNN ResNet-50 FPN V2 demonstrated the best performance when run on GPUs, while YOLO provided the best overall balance of speed and accuracy, making it particularly suitable for real-time processing. The system also incorporates an RTSP server to stream live video, enabling easy remote monitoring and control of the testbed.

The entire system is deployed within a Docker container, allowing for simplified setup, testing, and deployment. This smart city testbed demonstrates how a flexible and cost-effective testbed can support research in object detection for applications like traffic analysis.

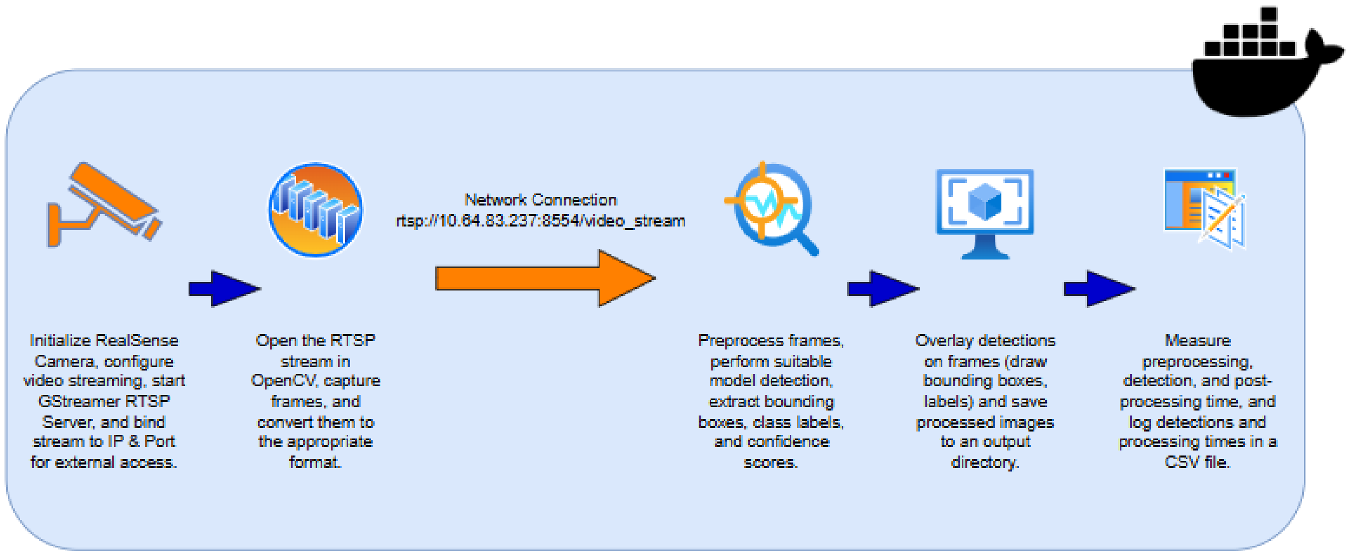
Network Connection
rtsp://10.64.83.237:8554/video_stream
Initialize RealSense Camera, configure video streaming, start GStreamer RTSP Server, and bind stream to IP & Port for external access.
Open the RTSP stream in OpenCV, capture frames, and convert them to the appropriate format.
Preprocess frames, perform suitable model detection, extract bounding boxes, class labels, and confidence scores.
Overlay detections on frames (draw bounding boxes, labels) and save processed images to an output directory.
Measure preprocessing, detection, and post-processing time, and log detections and processing times in a CSV file.

Figure 1: Subject Overview

Detection Models

In this project, several object detection models were tested, each with different advantages and disadvantages in terms of speed, accuracy, and computational requirements. The models evaluated are: YOLOv5, FCOS-ResNet50-FPN, Faster R-CNN variants, and SSD-Lite [1] [2]. More analytically for each model:

<u>YOLOv5</u>

YOLOv5 is a fast and efficient object detection model, well-suited for real-time applications. It is fast and accurate, making it ideal for tasks like traffic monitoring. YOLO models use a list of classes, such as vehicles, persons, and animals. During detection, YOLO not only identifies objects but also assigns the name of the class label.

FCOS-ResNet50-FPN

FCOS-ResNet50-FPN uses the Fully Convolutional One-Stage (FCOS) method. The model is capable of detecting objects at multiple scales using a ResNet50 backbone and Feature Pyramid Network (FPN). This model is especially effective in cluttered environments and for detecting objects of varying sizes.

Faster R-CNN

Faster R-CNN is a two-stage detection model. We tested two versions:

- **MobileNet v3 Large FPN**: It uses depthwise separable convolutions, which reduce computation while having accuracy. It also uses Feature Pyramid Networks (FPN) to help improve performance on detecting objects at multiple scales. The model is optimized for speed and efficiency, which makes it suitable for devices with limited resources.
- **ResNet50 FPN v2**: It uses a deeper ResNet50 backbone, which offers higher accuracy for detecting complex and challenging objects. The model also has a Feature Pyramid Network (FPN) to handle objects of varying sizes. While it requires more computational power compared to other models, it excels in accuracy and robustness.

SSD-Lite 320 (MobileNet v3 Large)

SSD-Lite 320 is a lightweight version of the SSD model, using a MobileNet v3 backbone. It is designed for fast inference on mobile or embedded systems with lower computational power. It is not as accurate as larger models, but it provides speed and low computation.

Dataset

Before developing the code for real-time camera-based object detection, we used a pre-existing dataset to evaluate the detection and visualization components of the system. The dataset chosen for this purpose is the Road Vehicle Dataset [3].

Description

The Road Vehicle Dataset contains various images of vehicles captured in different conditions, including a range of traffic scenes, vehicle types, and environments. This dataset is commonly used in traffic monitoring and vehicle detection tasks, making it an ideal choice for testing the initial implementation of object detection. The dataset includes:

- Diverse Vehicle Types: The dataset includes cars, trucks, buses, other vehicles and also humans, which is ideal for multi-class detection.
- Varied Lighting and Environmental Conditions: Images are captured under different lighting conditions and different angles, to challenge the detection models.

The dataset is crucial for initial testing and validation of detection models in a controlled setup, ensuring the basic functionality of the code for detecting, visualizing, and saving results before applying the system to live RTSP stream data.

Detector Code Development

Detection Process

The detection process begins after receiving a frame from the RTSP video stream. The frame, which is captured by the camera, undergoes several stages of processing to identify objects within the image. Initially, the frame is preprocessed by resizing and normalizing it to prepare it for detection. This involves transforming the image into a tensor that can be fed into the object detection model.

Once the image is in the proper format, the detection model processes it and returns a set of predicted bounding boxes. Each bounding box represents a detected object and contains the following information:

- Coordinates of the box (x1, y1, x2, y2)
- The confidence score of the detection
- The class label of the detected object

The bounding boxes and confidence scores are then used to determine which objects should be visualized. Detections with a confidence score above a set threshold are kept, while confidence score bellow it are discarded to avoid false positives.

Visualization

After deciding on the objects that should be visualized, the next step is to actually visualize their bounding boxes on the frame. This is done by converting the frame from BGR to RGB format, to ensure that they are compatible with the visualization. The bounding boxes are drawn in green to make them easily distinguishable.

Next to each bounding box, a label is added to identify the class of the detected object. The label is displayed with the confidence score, indicating the model's certainty about the detection. To make the text and box visible, the label is displayed above the bounding box.

The visualization also includes a count of the total number of objects detected, providing a quick overview of the scene.

Saving the Results

Once the bounding boxes and labels are drawn on the frame, the resulting image is saved. The output directory is created if it doesn't already exist. The images are saved in JPEG format.

In addition to saving the processed images, performance metrics are also recorded for each frame. The metrics are logged in a CSV file. Each row in the CSV contains the following information:

- **ID**: A unique identifier for the processed image
- **Image Name**: A descriptive name for the image file
- **Total Time (s)**: The overall time taken to process the frame
- **Preprocessing Time (s)**: Time taken for the preprocessing step
- **Detection Time (s)**: Time spent on running the detection model

- **Post-processing Time (s)**: Time taken for visualizing and saving the results
- **Items Detected**: The number of objects detected in the frame

Using the results in the CSV file, we will evaluate the system's performance and understand how much time each stage of the detection pipeline needs.

Models Evaluation

Evaluating the performance of the five object detection models involved testing them on two different GPUs and the Raspberry Pi 5 (8GB Model). The GPUs that where used for testing are:

- NVIDIA A2 Tensor Core GPU
- NVIDIA GeForce RTX 4090

The goal was to analyze their speed and accuracy across different hardware configurations.

Evaluation Criteria

- Detection Time: Measured in frames per second (FPS) to determine real-time performance.
- Accuracy: The amount of correctly detected objects.

Graphical Representation of the Model's Average Detection Time for Each Processor

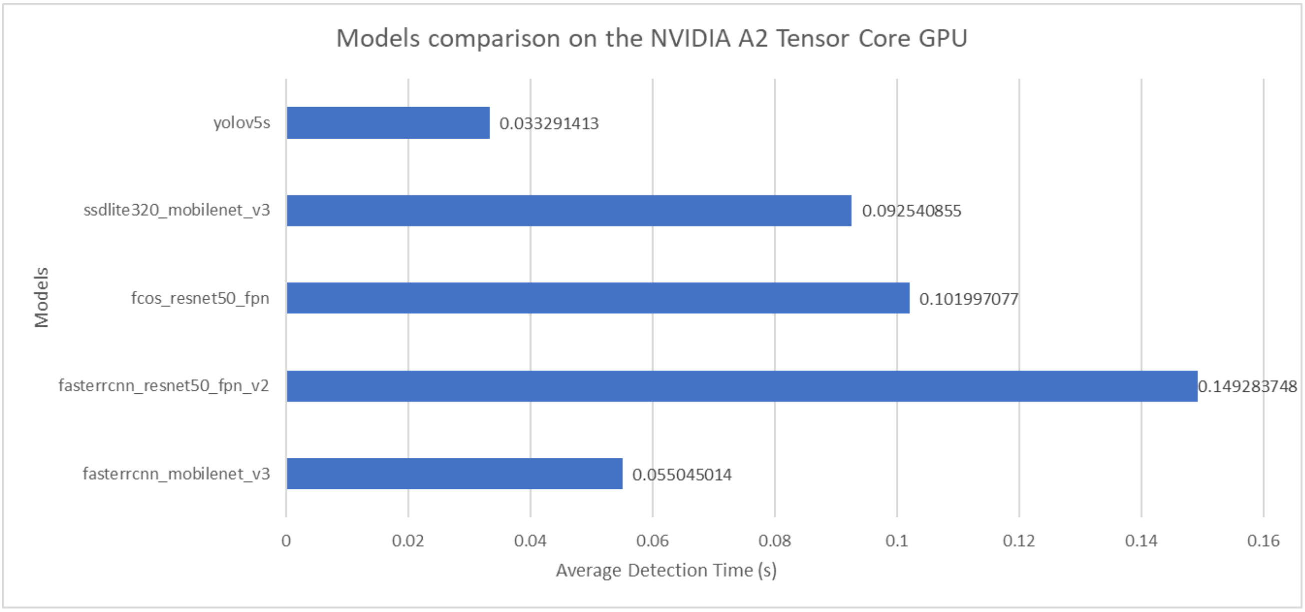

Figure 2: Models comparison on the NVIDIA A2 Tensor Core GPU on average detection time per frame

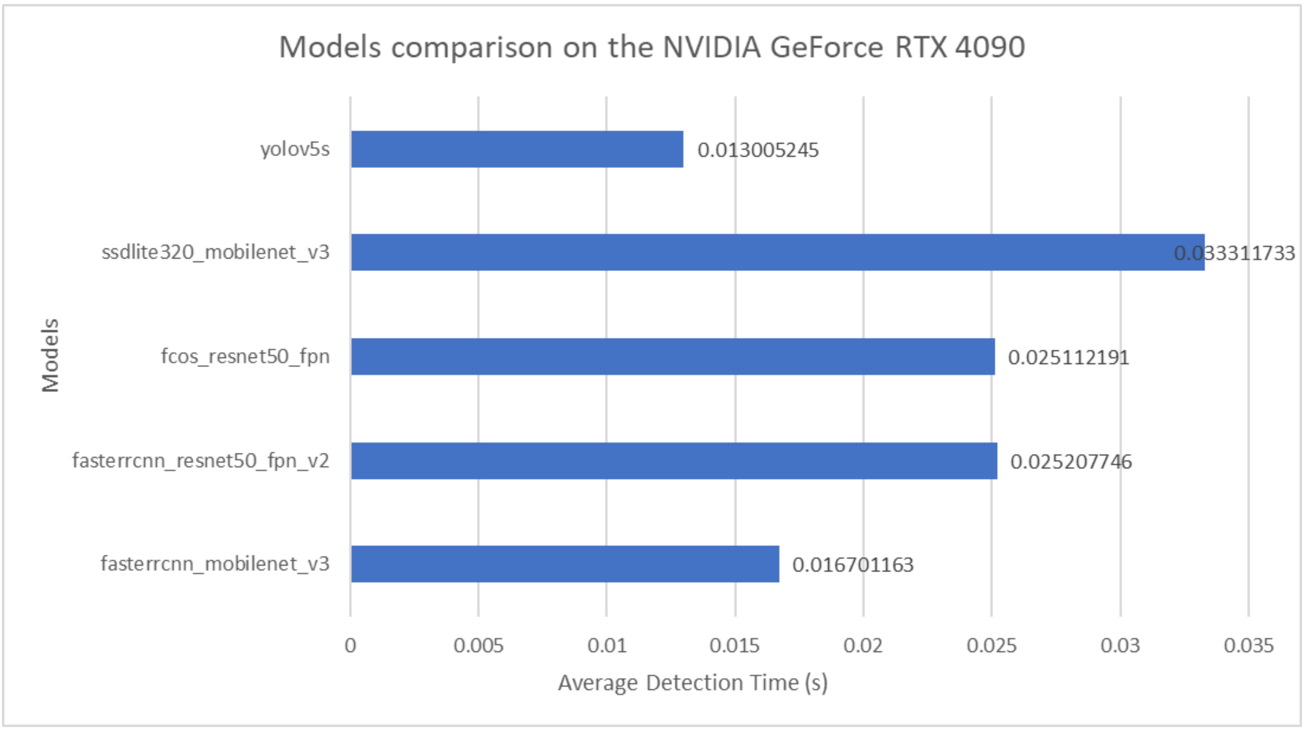

Figure 3: Models comparison on the NVIDIA GeForce RTX 4090 on average detection time per frame

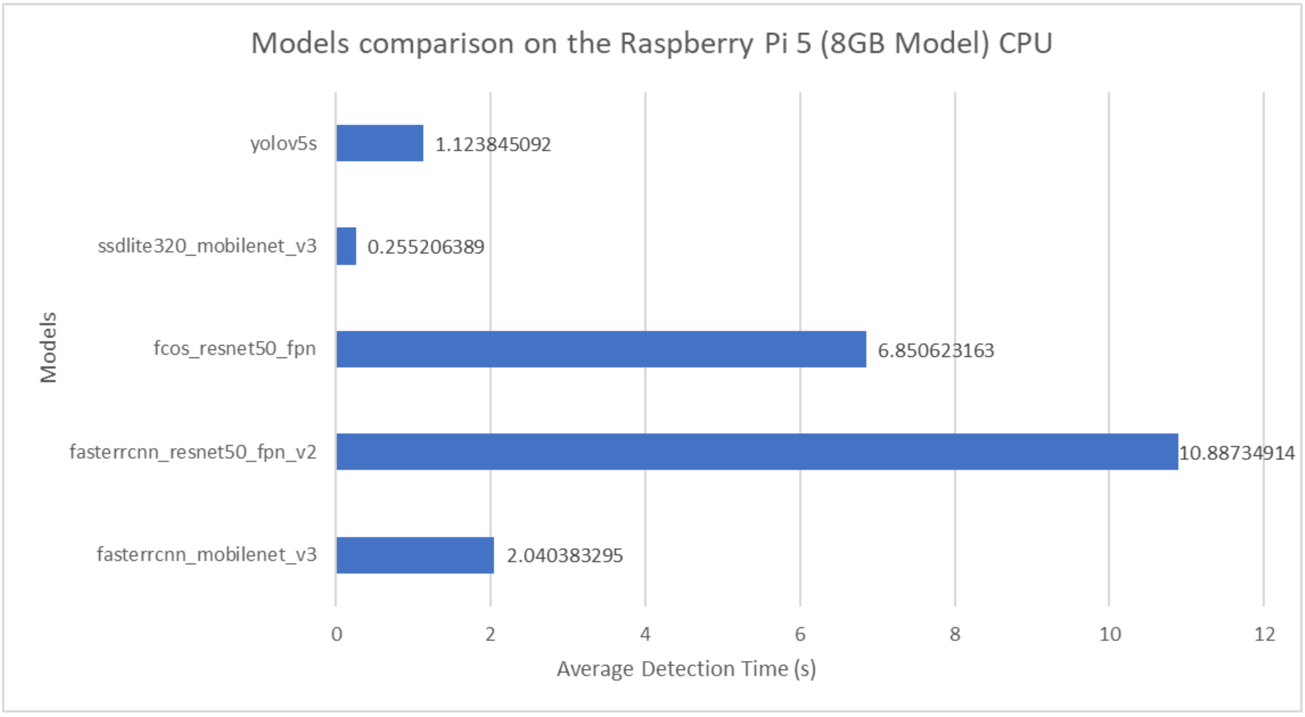

Figure 4: Models comparison on the Raspberry Pi 5 (8GB Model) on average detection time per frame

Observations

On high-performance GPUs, Faster R-CNN ResNet50 FPN v2 provided the best results. Generally, Faster R-CNN ResNet50 FPN v2 detected accurately the greatest amount of objects due to its strong feature extraction capabilities, making it ideal for complex tasks. Its execution time was much higher than other models in CPU but in the GPUs its execution time was on average the same (slightly higher as with other models. Although its execution time is slightly worse, the trade of between speed and accuracy still makes it the best candidate for object detection on GPUs.

For CPU-based inference, including testing on the Raspberry Pi 5, YOLOv5 was the best-performing model. It provided a balance of speed and accuracy, handling real-time detection efficiently despite the limitations. Other models, such as Faster R-CNN and SSD had much higher execution times on the Raspberry Pi, so that even if their detection accuracy is more efficient, it doesn't worth the extra execution time.

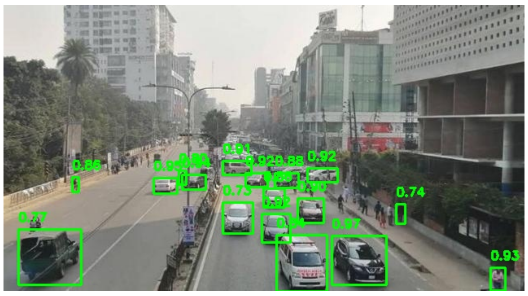

Figure 5: Excellent detection example: Faster R-CNN ResNet50 FPN v2

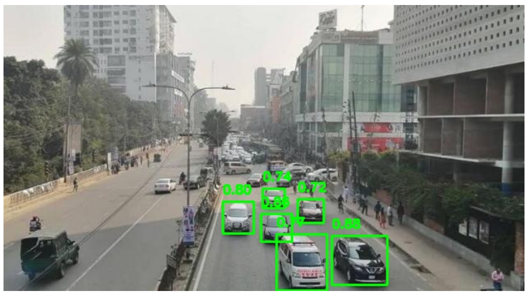

Figure 6: Good detection example: YOLOv5s

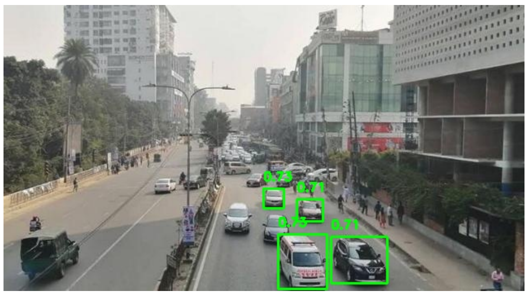

Figure 7: Bad detection example: FCOS-ResNet50-FPN

Results

The testing process revealed that the choice of model depends on the available hardware, CPU or GPU. YOLOv5 is the most efficient for CPU-based environments, while Faster R-CNN ResNet50 FPN v2 is the best choice for high-performance GPU-based environments.

RTSP RealSense Streaming and Integration

In this part, the implementation of an RTSP streaming server using a RealSense camera is described. The camera captures frames in real-time and streams them over an RTSP protocol, which can be accessed remotely using an RTSP client. This code uses GStreamer for setting up the server, allowing for efficient video streaming. **The base of the code of this section was executed by a fellow colleague.**

Dependencies

For the RTSP RealSense Streaming and Integration to function properly, several dependencies needed to be installed and configured. The setup involved installing Intel RealSense SDK from the source [4] [5] to enable communication with the RealSense camera. The GStreamer RTSP Server was also installed from source to handle video streaming over the network and also additional GStreamer plugins were installed. Compilation tools like meson, ninja-build, and cmake were required to build both libraries. Once installed, the system was configured to recognize the RealSense camera and stream its feed over RTSP.

Camera Setup

The *SensorFactory* class handles the connection and configuration of the RealSense camera. The camera pipeline is initialized using the pyrealsense2 library, which enables access to the RGB camera. The pipeline is set to stream colored images at a specific resolution (640x480) and a frame rate of 15 FPS, which might change depending on the speed of the detection model. The RTSP server is set to stream the camera feed to a given IP address and port.

Streaming Setup

The *GstServer* class is managing the RTSP server. It connects the camera feed to a URI and makes sure that the streaming is continuous. The server uses GStreamer's appsrc element to push frames from the RealSense pipeline into the RTSP stream. The frames are resized to 640x480, encoded into H.264 format, and transmitted over the network.

Frame Request Setup

Every time the RTSP server needs new data (frames), the *on_need_data* method is called. This method requests a frame from the RealSense camera, processes it, and pushes it into the GStreamer pipeline for streaming. If no frame is available, a black frame is sent as a fallback. The system makes sure that the frames are sent at the correct frame rate.

RTSP Setup

The RTSP server setup involves setting up GStreamer elements, connecting them to the appropriate signals, and launching the media pipeline. The SensorFactory creates the necessary GStreamer elements that are needed for live streaming. The video frames are continuously pushed into the stream, and the RTSP server handles the transmission over the network, making the video feed available to any clients that wants access.

Client Setup

To receive the frames from the RTSP stream, the client must establish a connection to the server. In our case, the client is the detector that needs to process frames. The client initializes an RTSP stream using OpenCV's VideoCapture function, specifying the RTSP URL. Once connected, the client continuously reads frames from the stream and processes them as needed.

Testbed

Testbed Setup

The testbed for real-time video streaming and object detection consists of a Raspberry Pi 5 (2GB) and an Intel RealSense Camera. These devices work together to capture, process, and stream real-time video over an RTSP server.

1. Raspberry Pi 5 (8GB Model)

Serves as the primary computing unit, handling video capture, processing, and RTSP streaming. Since it lacks a dedicated GPU for acceleration, all computations relies on the CPU. Despite its limited power, it supports real-time streaming at a reasonable frame rate. It has a 64 bit quad-core Arm Cortex-A76 processor and supports USB 3.0, which is where the Intel RealSense camera will be connected.

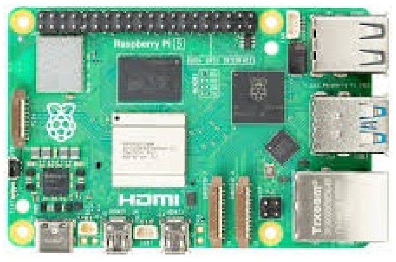

Figure 8: Raspberry Pi 5 (8GB Model)

2. Intel RealSense Depth Camera D455

The Intel RealSense camera is used as the primary vision sensor in this setup. It provides high-resolution color images with depth, which makes it ideal for real-time detection. The camera is capable of streaming video at frame rates up to 30 FPS.

Figure 9: Intel RealSense Depth Camera D455

Dockerfile

Finally, a Dockerfile was created to *"pack"* the project. A Dockerfile is a script containing a series of instructions to build a Docker image. It is a lightweight and executable package that includes everything needed to run an application, such as dependencies and system tools. It uses the Ubuntu 20.04 base image and installs important system libraries and tools needed to build and run applications with Intel RealSense SDK, OpenCV, and GStreamer. A Python virtual environment is created, and key libraries like PyTorch, torchvision, ultralytics, and pyrealsense2 are installed to support real-time video capture and image processing. The librealsense SDK is also built and installed from source, enabling the use of Intel RealSense cameras. Jupyter Notebook is included for developing and testing of the project. This setup makes the environment portable so that it can be used in any device and operating system.

Category	**Dependencies**
System Libraries	git, cmake, build-essential, libusb-1.0-0-dev, pkg-config, libssl-dev, v4l-utils, libx11-dev, libgtk-3-dev, libglfw3-dev, libx11-xcb-dev, libxcb-dri3-dev, libxcb-present-dev, libxcb-render0-dev, libxcb-shm0-dev, libxcb-xfixes0-dev, libxrandr-dev, libxinerama-dev, libxcursor-dev, libgl1-mesa-dev, libglu1-mesa-dev, libgomp1, libgl1-mesa-glx, libglib2.0-0, freeglut3-dev, mesa-common-dev, ntp, udev, usbutils
Python Libraries	opencv-python, opencv-contrib-python, torch, torchvision, ultralytics, numpy, pyrealsense2, PyGObject, notebook, python3-pip, python3-dev, python3-setuptools, python3-gi, python3-gi-cairo
GStreamer Libraries	libgstreamer1.0-dev, gstreamer1.0-plugins-base, gstreamer1.0-plugins-good, gstreamer1.0-plugins-bad, gstreamer1.0-plugins-ugly, gstreamer1.0-libav, gstreamer1.0-tools, gstreamer1.0-rtsp, gstreamer1.0-python3-plugin-loader, libgstrtspserver-1.0-dev, gstreamer-plugins-bad1.0-dev
RealSense SDK	librealsense (cloned from GitHub, built, and installed) [4]
Virtual Environment	Python3 venv, pip, setuptools
Build Tools	meson, ninja-build, flex, bison

Table 1: Chart of the dependencies needed in the dockerfile

While trying to build the RealSense RTSP streaming pipeline with GStreamer, there were a few challenges. First, setting up the RealSense camera pipeline was difficult because the stream configuration wasn't done correctly, causing problems starting the video feed. Another issue was getting frames from the camera. Sometimes no frame would be received, so we had to add a backup plan to send a black frame when this happened. We also had to make sure the video frames were sent properly to GStreamer with the right fps and format. Troubleshooting these issues required adding a lot of logs to see what was happening and checking if the camera was detected correctly.

To make the project easier to run, we used a Docker container. After writing a Dockerfile, we built the image with the command:

```
docker build -t realsense-rtsp .
```

Then, the container started with the command:

```
docker run -it --rm --privileged --device /dev/video24:/dev/video24 --device
/dev/bus/usb:/dev/bus/usb --device-cgroup-rule='c 81:* rmw' -p 8554:8554
--memory="4g" realsense-rtsp bash
```

- `docker run`: It is the main command used to start a new Docker container.
- `-it`: It stands for *"interactive terminal"*. It keeps the container's standard input open and attaches a terminal session, so you can interact with the container.
- `--rm`: It automatically removes the container once it stops, keeping things clean and preventing leftover containers.
- `--privileged`: It grants the container extended privileges, allowing it to access and manage hardware devices directly, which is necessary for the Intel RealSense camera.
- `--device /dev/video24:/dev/video24`: It passes the video device from the host machine to the container.
- `--device /dev/bus/usb:/dev/bus/usb`: It allows the container to access the USB bus on the host, so it can communicate with connected USB devices like the Intel RealSense camera.
- `--device-cgroup-rule='c 81:* rmw'`: It sets specific permissions for device groups in the container.
- `-p 8554:8554`: It maps port 8554 on the host to port 8554 in the container. The RTSP video stream is accessible through this port.
- `--memory="4g"`: It limits the container's memory usage to 4 GB, helping prevent excessive resource consumption.
- `realsense-rtsp`: It is the name of the Docker image being used to create the container.
- `bash`: It specifies that the container should start with the Bash shell.

Once inside the Docker container, we need to set up and run the RealSense camera stream via the GStreamer RTSP server. First, the virtual environment is activated using

```
source /opt/venv/bin/activate.
```

Then, the streaming script is launched with the command:

```
python camera.py --device_id 0 --fps 30 --image_width 640 --image_height 480
--ip 10.64.83.237 --port 8554 --stream_uri /video_stream.
```

Several challenges were encountered throughout this process. Ensuring the container had proper access to video and USB devices, where some of the most difficult. Also, issues with permissions occured that were addressed by adding device cgroup rules. Furthermore, configuring the RealSense camera streams with the correct resolution, format, and frame rate often required adjustments in the GStreamer pipeline settings. For certain resolutions the pipeline was not able to correctly stream the frames. Despite these difficulties, the setup was successfully completed, and the camera's video stream was made accessible over the port 8554.

Testing & Evaluation

The system was tested by running the Docker container on the two different devices:

- Raspberry Pi 5 (8GB Model) CPU
- NVIDIA GeForce RTX 4090 GPU

The goal was to see how well the system handled real-time video streaming on different devices. For each device, we measured the speed (frames per second) of the object detection models, Faster R-CNN ResNet-50 FPN V2 and YOLO, since those two models provided the best trade offs for CPUs and GPUs correspondingly. Figures 10, **??**, 11, show the result from each device.

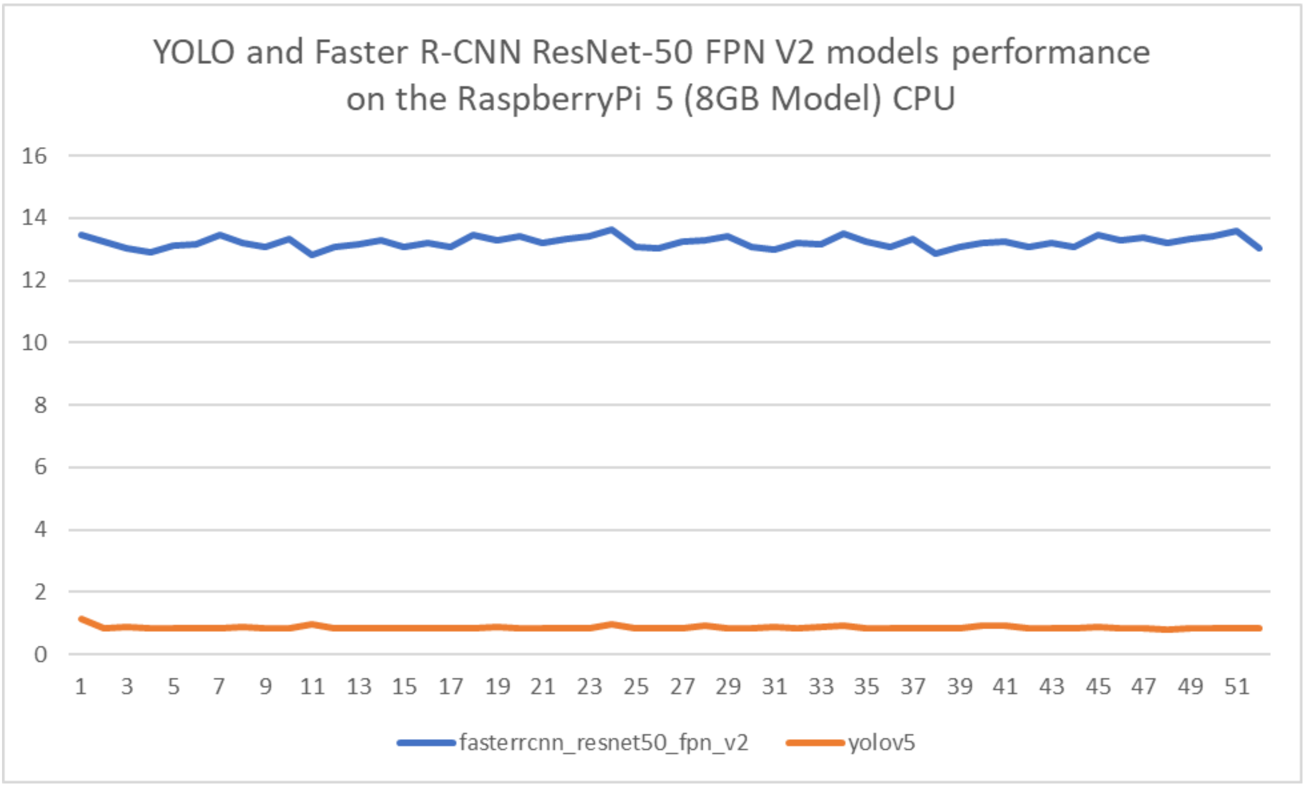

Figure 10: YOLO and Faster R-CNN ResNet-50 FPN V2 models performance on the Raspberry Pi 5 (8GB Model) CPU with real-time video streaming from the Intel RealSense camera

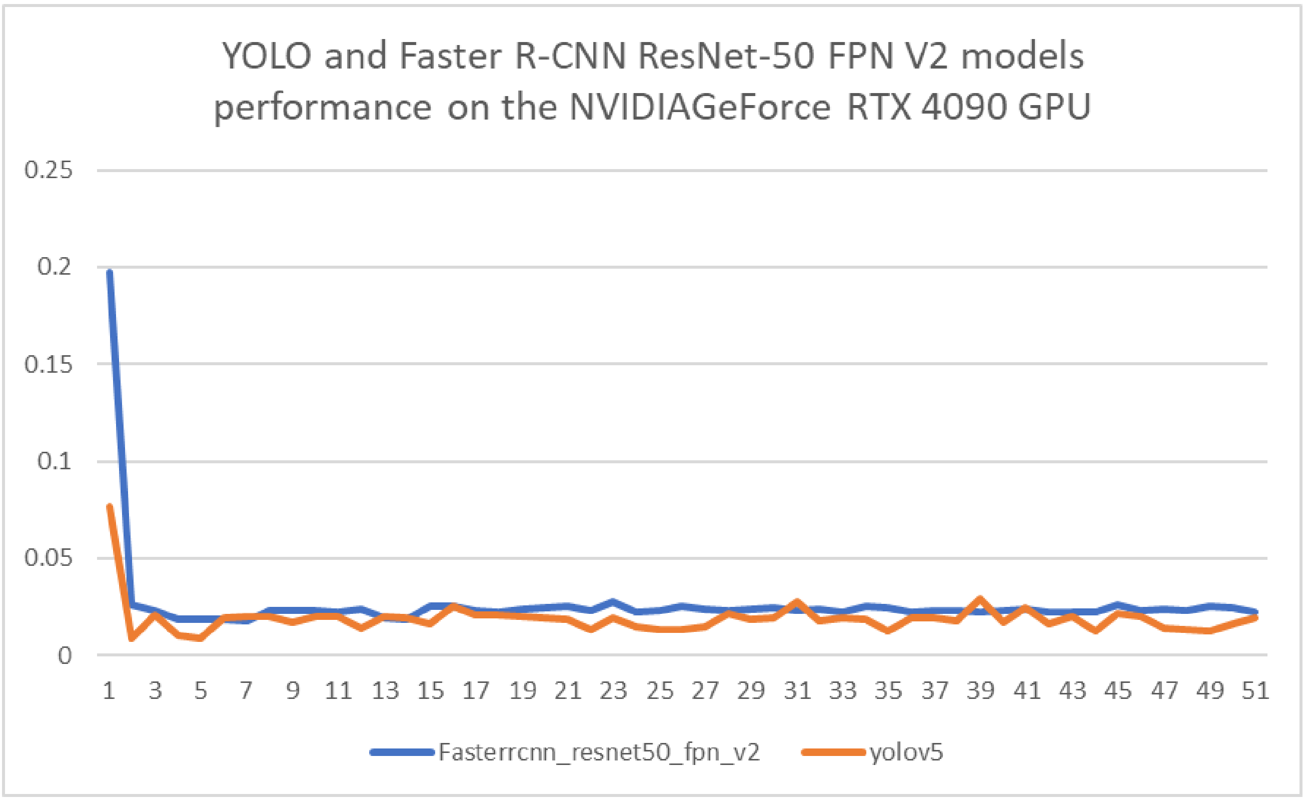

Figure 11: YOLO and Faster R-CNN ResNet-50 FPN V2 models performance on the NVIDIA GeForce RTX 4090 GPU with real-time video streaming from the Intel RealSense camera

The results above showed that the performance is different in each device for each model. Each model in each device needed a different frame rate to perform. The Faster R-CNN ResNet-50 FPN V2 perform better with 15fps while the YOLO model perform better with 45fps. The Raspberry Pi 5 (8GB Model) CPU also struggled with lower frame rates and slower detection speeds. For this device the YOLO model provided the best trade off between speed and accuracy, making it the most suitable for real-time applications on this device. The other model was extremely slow while running on the CPU. On the NVIDIA GeForce RTX 4090 GPU, Faster R-CNN ResNet-50 FPN V2 achieved the highest accuracy and great speed making it a practical choice for this device. YOLO still maintained its real-time performance as in the CPU.

Based on these evaluations, YOLO is the best overall model due to its balance of speed and accuracy across all tested devices. It was especially effective on lower-powered hardware, CPUs, while still performing well on GPUs. These results highlight the flexibility and efficiency of the testbed, demonstrating its ability to adapt to different hardware setups. Below, we can see some of the real-time detection the models used:

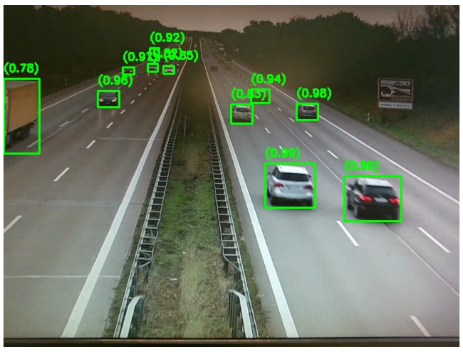

Figure 12: CPU detection using Faster R-CNN ResNet50 FPN v2

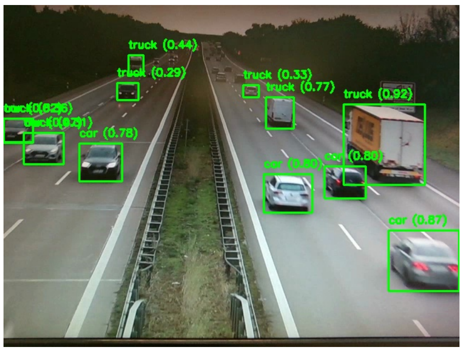

Figure 13: CPU detection using YOLOv5s

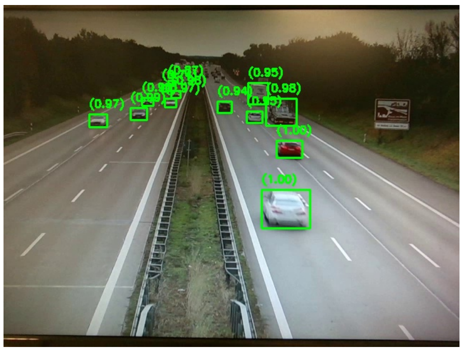

Figure 14: GPU detection using Faster R-CNN ResNet50 FPN v2

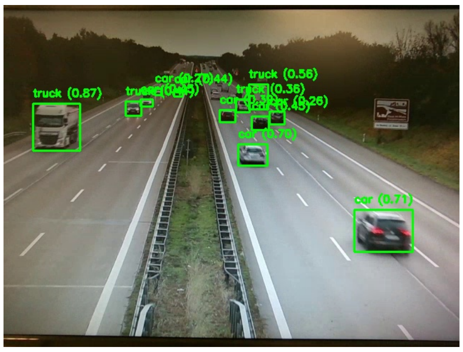

Figure 15: GPU detection using YOLOv5s

The traffic video was used from the internet, so that there is no violations of private data [6]. It was running on a big screen that the camera was pointing to. More samples of the detections can be found in the folder *detections*.

Conclusion

This project successfully created a simple and effective test bed to study different object detection methods. Using an Intel RealSense camera and a Raspberry Pi, the system was able to capture real-time video and analyze it. An RTSP server was set up to stream live video, making it easy to monitor the testbed remotely.

We tested several object detection models, including Faster R-CNN ResNet-50 FPN V2, YOLO, FCOS ResNet-50, Faster R-CNN, and SSD-Lite. From all the models, YOLO stood out because it offered the best trade off between speed and accuracy, making it perfect for real-time detections. Faster R-CNN ResNet-50 FPN V2 showed the best trade off between speed and accuracy when using GPUs. These tests helped us understand the strengths and weaknesses of different models.

The whole system was packed inside a Docker container, which made installation and running the project much easier. While setting up, we faced some challenges like giving the container access to the camera and USB devices, adjusting camera settings, and fixing issues with video streaming. Still, we managed to solve these problems and get the system working properly.

In the end, this project showed how a low-cost, flexible test setup can help with research on smart city technologies. This testbed provides a strong base for future work!

References

[1] “YOLOv5 — pytorch.org,” https://pytorch.org/hub/ultralytics_yolov5/.

[2] “Models and pre-trained weights &x2014; Torchvision 0.21 documentation — pytorch.org,” https://pytorch.org/vision/stable/models.html.

[3] https://datasetninja.com/road vehicle, “Road Vehicle - Dataset Ninja — datasetninja.com,” https://datasetninja.com/road-vehicle.

[4] “librealsense/doc/libuvc_installation.md at master · IntelRealSense/librealsense — github.com,” https://github.com/IntelRealSense/librealsense/blob/master/doc/libuvc_installation.md, [Accessed 18-02-2025].

[5] “librealsense/doc/libuvc_installation.md at master · IntelRealSense/librealsense — github.com,” https://github.com/IntelRealSense/librealsense/blob/master/doc/libuvc_installation.md, [Accessed 18-02-2025].

[6] https://pixabay.com/videos/highway-traffic-vehicles-cars-road-56310/.