Multi-Camera Object Detection on Nvidia Jetson Orin Nano using Zed 2i

1119292 Shriya Chavan
1119293 Pooja Chavan
1119235 Supriya Chilukuri

Under the guidance of Prof. Thomas Schumann

AIM OF THIS PROJECT

- ➢ Real time Object Detection System
- ➢ Using ZED Camera
- ➢ On Jetson Orin Nano
- ➢ YOLOv8 algorithm
- ➢ Do performance evaluation
- ➢ Performance Comparison

WHAT IS **OBJECT DETECTION**?

- Object detection is a computer vision task that involves identifying and locating objects in images or videos.
- Earlier approach was to use R-CNN for object detection.
- We are using YOLO, which is also a neural network.
- It is an important part of many applications, such as surveillance, self-driving cars, or robotics.

HARDWARE USED

1) NVIDIA Jetson Orin Nano

- ➢ AI Performance: 40 TOPS
- ➢ GPU max frequency: 625MHz
- ➢ CPU max frequency: 1.5GHz
- ➢ GPU : 1024-core NVIDIA Ampere architecture GPU with 32 Tensor Cores
- ➢ CPU : 6-core Arm® Cortex®-A78AE v8.2 64-bit CPU
- ➢ 1.5MB L2 + 4MB L3

HARDWARE USED

2) ZED 2i Camera

- ➢ Neural Depth Sensing
- ➢ Spatial Object Detection
- ➢ Built-in Next-Gen IMU, Barometer & Magnetometer
- ➢ 120° Wide-Angle FOV
- ➢ All-Aluminium Frame with Thermal Control
- ➢ Built-in 1.5m USB 3.0 Cable

SOFTWARE USED

Software	**Versions**
Operating System	Ubuntu 20.04
Programming language	Python 3.8
NVIDIA SDK	Jetpack 5.1.2
ZED SDK	4.0.
CUDA	11.4.19
YOLO (Neural Network)	Version 8
PyTorch	2.1.2

NVIDIA JetPack SDK 5.1.2

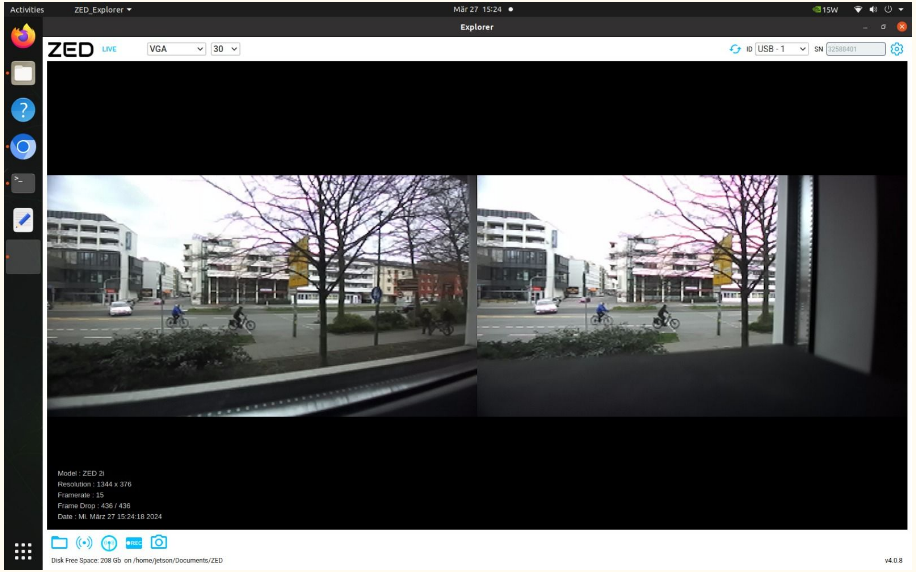
- TensorRT is a high performance deep learning inference runtime for image classification, segmentation, and object detection neural networks
  - JetPack 5.1.2 includes **TensorRT 8.5.2**
- NVIDIA DLA hardware is a fixed-function accelerator engine targeted for deep learning operations.
  - JetPack 5.1.2 includes **DLA 3.12.1**
- CUDA Deep Neural Network library provides high-performance primitives for deep learning frameworks.
  - JetPack 5.1.2 includes **cuDNN 8.6.0**
- CUDA Toolkit provides a comprehensive development environment for C and C++ developers building GPU-accelerated applications.
  - JetPack 5.1.2 includes **CUDA 11.4.19**
- OpenCV is an open source library for computer vision, image processing and machine learning.
  - JetPack 5.1.2 includes **OpenCV 4.5.4**

What is Ultralytics **YOLOv8**?

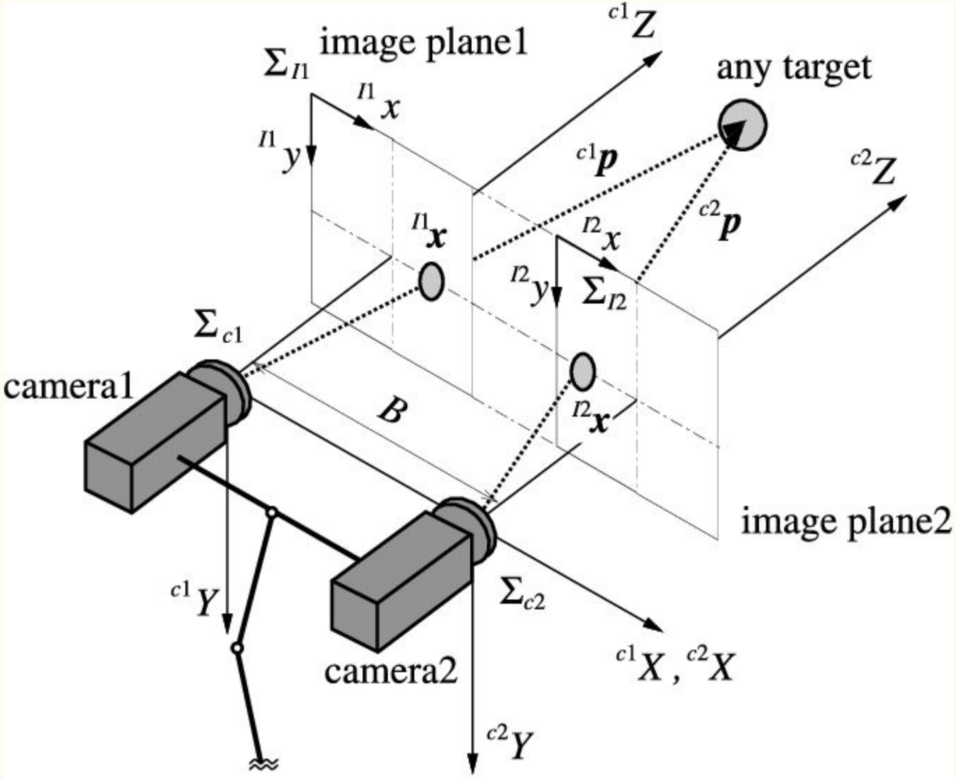
- ➢ YOLOv8 is a deep learning-based object detection model developed by UltraLytics
- ➢ It stands for "You Only Look Once version 8," which is the eighth iteration of the YOLO model series.
- ➢ It typically uses a more advanced backbone architecture, such as CSPDarknet53, to extract features from input images effectively.
- ➢ It offers compatibility with different frameworks like PyTorch or TensorFlow.

ZED SDK

- ➢ Stereo 2K cameras with dual 4MP RGB sensors.
- ➢ UVC-compliant USB 3.0 camera
- ➢ Compatible with USB 2.0
- ➢ Left and right video frames are synchronized and streamed as a single uncompressed video frame in the side-by-side format.
- ➢ Camera has a compact structure and reduced size
- ➢ It relatively simple to incorporate into robotic systems or drones.

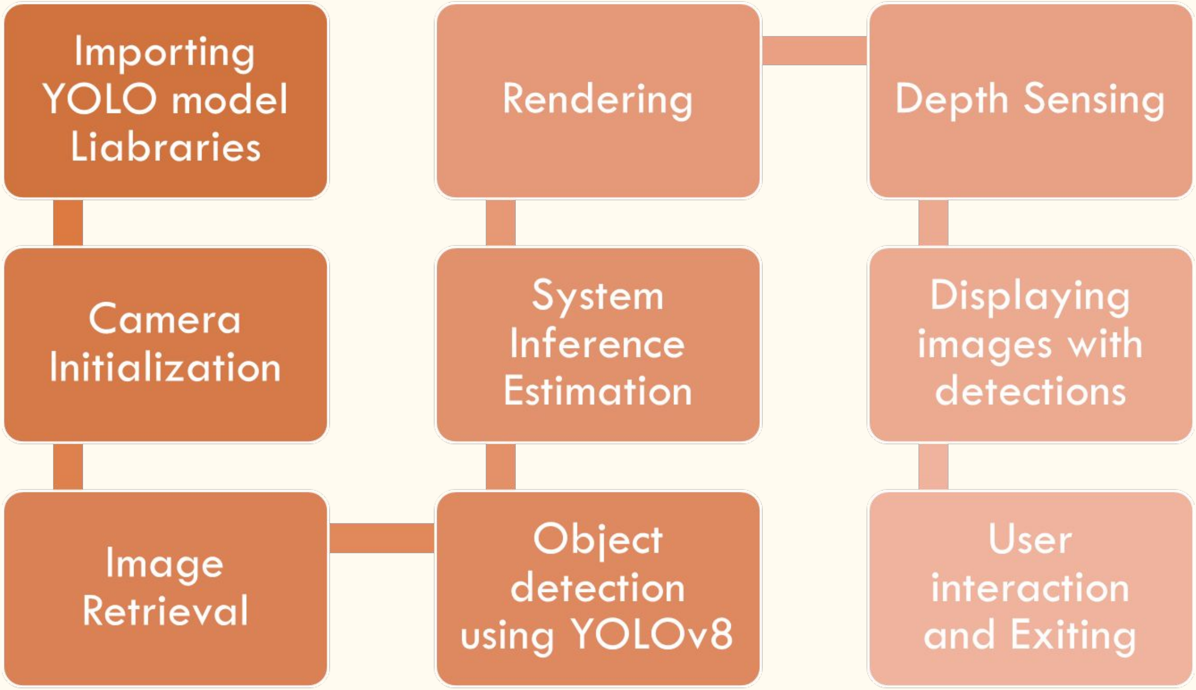


What is **depth sensing?**



- ➢ A technology that allows a device to measure the distance of an object from the sensor.
- ➢ The system in the image works by using two cameras that are slightly offset from each other.
- ➢ Each camera captures a slightly different image of the scene. By comparing the two images, the disparity is calculated.
- ➢ This disparity is used to calculate the distance of the object from the camera.

Process Flow Diagram



IMPLEMENTATION

Camera Instantiation and Initialising Parameters

```
zed = [[] ,[]]

for i in range(2):
    zed[i] = sl.Camera()

init_params = sl.InitParameters()
init_params.camera_resolution = sl.RESOLUTION.VGA
init_params.camera_fps = 100
init_params.coordinate_units = sl.UNIT.METER
init_params.depth_mode = sl.DEPTH_MODE.ULTRA
init_params.coordinate_system = sl.COORDINATE_SYSTEM.RIGHT_HANDED_Y_UP
init_params.depth_maximum_distance = 50
```

Argument Parser and Loading the YOLO models

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--weights', nargs='+', type=str, default=['yolov8n.pt', 'yolov8l.pt'], help='model.pt paths for each camera')
    parser.add_argument('--svo', type=str, default=None, help='optional svo file')
    parser.add_argument('--img_size', type=int, default=416, help='inference size (pixels)')
    parser.add_argument('--conf_thres', type=float, default=0.4, help='object confidence threshold')
    opt = parser.parse_args()
```

Threaded model inference initialization

```
capture_threads = [
    Thread(target=torch_thread, args=(opt.weights[i], opt.img_size, i + 1, opt.conf_thres)) for i in range(2)
]
for thread in capture_threads:
    thread.start()
```

What happens in the **torch_thread** function?

```
def torch_thread(weights, img_size, camera_id, conf_thres=0.2, iou_thres=0.45):
    global image_nets, exit_signal, run_signals, detections

    print("Intializing Network for Camera", camera_id)

    model = YOLO(weights)

    while not exit_signal:
        start_time = time.time()  # Start time for measuring inference time
        if run_signals[camera_id - 1]:
            lock[camera_id - 1].acquire()
            img = cv2.cvtColor(image_nets[camera_id - 1], cv2.COLOR_BGRA2RGB)
            det = model.predict(img, save=False, imgsz=img_size, conf=conf_thres, iou=iou_thres)[0].cpu().numpy().boxes
            detections[camera_id - 1] = detections_to_custom_box(det, image_nets[camera_id - 1])
            lock[camera_id - 1].release()
            end_time = time.time()  # End time for measuring inference time
            inference_time = end_time - start_time  # Calculate inference time for the current frame
            fps = 1.0 / inference_time  # Calculate frames per second
            print(f"Inference FPS for Camera {camera_id}: {fps:.2f}")
            run_signals[camera_id - 1] = False

        sleep(0.01)

```

Converting detections to Custom Box Objects

```python
def detections_to_custom_box(detections, im0):
    output = []
    for i, det in enumerate(detections):
        xywh = det.xywh[0]

        # Creating ingestable objects for the ZED SDK
        obj = sl.CustomBoxObjectData()
        obj.bounding_box_2d = xywh2abcd(xywh, im0.shape)
        obj.label = det.cls
        obj.probability = det.conf
        obj.is_grounded = False
        output.append(obj)
    return output
```

Image Acquisition and Object Retrieval

```
lock[i].acquire()
zed[i].retrieve_image(image_left_tmp[i], sl.VIEW.LEFT, sl.MEM.CPU, display_resolutions[i])
image_nets[i] = image_left_tmp[i].get_data()

lock[i].release()
run_signals[i] = True

while run_signals[i]:
    sleep(0.001)

lock[i].acquire()
# Ingest detections from respective camera image net
zed[i].ingest_custom_box_objects(detections[i])
det_list = detections[i]

lock[i].release()
zed[i].retrieve_objects(objects[i], obj_runtime_param)
```

Depth Sensing

```
# Retrieve display data
zed[i].retrieve_measure(depth_map[i], sl.MEASURE.DEPTH)
```

```
bbox = det.bounding_box_2d
```

```
center = np.mean(bbox, axis=0)
x = round(center[0])
y = round(center[1])


err, depth_value = depth_map[i].get_value(x,y)
```

The code is scalable. Implementing the project with more than two cameras is also possible.

```python
# Create OpenGL viewer
viewer = gl.GLViewer()

point_cloud_res1 = sl.Resolution(min(camera_res1.width, 720), min(camera_res1.height, 404))
point_cloud_res2 = sl.Resolution(min(camera_res2.width, 720), min(camera_res2.height, 404))

point_cloud_render1 = sl.Mat()
point_cloud_render2 = sl.Mat()
viewer.init(camera_infos1.camera_model, point_cloud_res1, obj_param.enable_tracking)
viewer.init(camera_infos2.camera_model, point_cloud_res2, obj_param.enable_tracking)

point_cloud1 = sl.Mat(point_cloud_res1.width, point_cloud_res1.height, sl.MAT_TYPE.F32_C4, sl.MEM.CPU)
point_cloud2 = sl.Mat(point_cloud_res2.width, point_cloud_res2.height, sl.MAT_TYPE.F32_C4, sl.MEM.CPU)

display_resolution1 = sl.Resolution(min(camera_res1.width, 1280), min(camera_res1.height, 720))
display_resolution2 = sl.Resolution(min(camera_res2.width, 1280), min(camera_res2.height, 720))

image_scale1 = [display_resolution1.width / camera_res1.width, display_resolution1.height / camera_res1.height]
image_scale2 = [display_resolution2.width / camera_res2.width, display_resolution2.height / camera_res2.height]

image_left_ocv1 = np.full((display_resolution1.height, display_resolution1.width, 4), [245, 239, 239, 255], np.uint8)
image_left_ocv2 = np.full((display_resolution2.height, display_resolution2.width, 4), [245, 239, 239, 255], np.uint8)

camera_config1 = camera_infos1.camera_configuration
camera_config2 = camera_infos2.camera_configuration

tracks_resolution1 = sl.Resolution(400, display_resolution1.height)
tracks_resolution2 = sl.Resolution(400, display_resolution2.height)

track_view_generator1 = cv_viewer.TrackingViewer(tracks_resolution1, camera_config1.fps, init_params.depth_maximum_distance)
track_view_generator2 = cv_viewer.TrackingViewer(tracks_resolution2, camera_config2.fps, init_params.depth_maximum_distance)

track_view_generator1.set_camera_calibration(camera_config1.calibration_parameters)
track_view_generator2.set_camera_calibration(camera_config2.calibration_parameters)

image_track_ocv1 = np.zeros((tracks_resolution1.height, tracks_resolution1.width, 4), np.uint8)
image_track_ocv2 = np.zeros((tracks_resolution2.height, tracks_resolution2.width, 4), np.uint8)
```

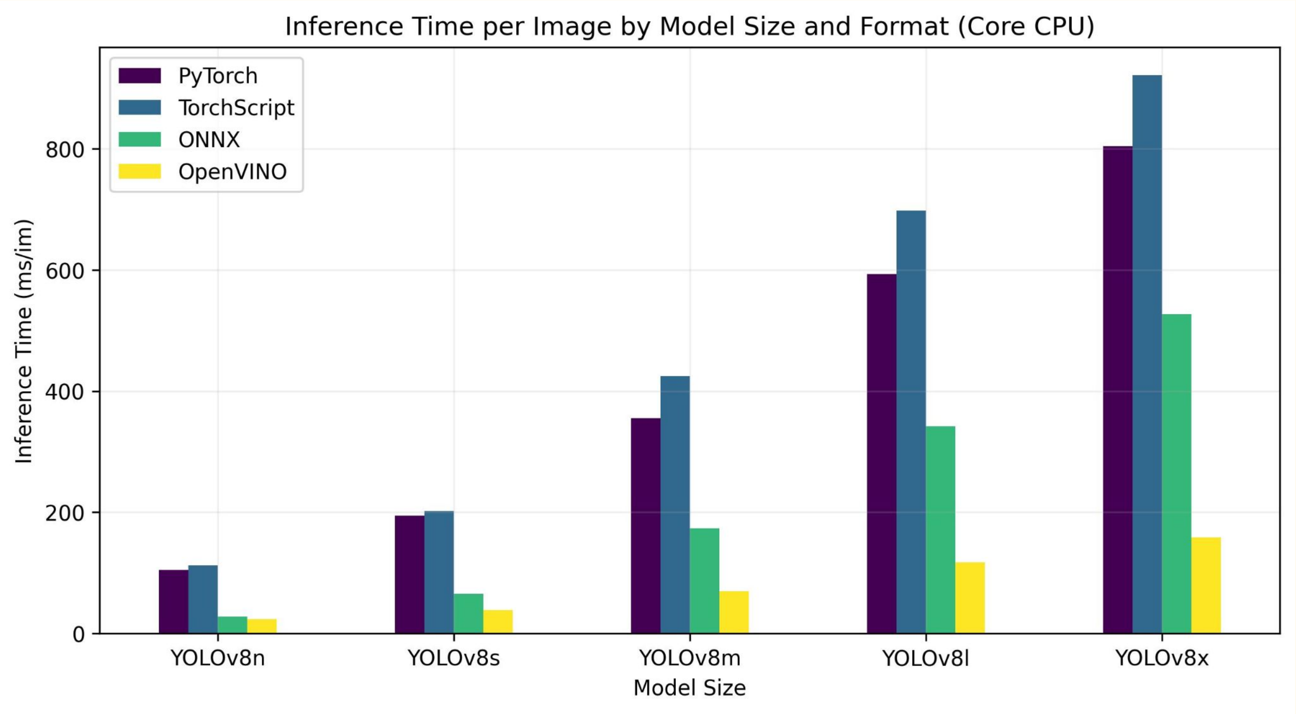
```python
for i in range(2):

    point_cloud_resolutions.append(sl.Resolution(min(camera_resolutions[i].width, 720), min(camera_resolutions[i].height, 404)))
    point_clouds.append(sl.Mat(point_cloud_resolutions[i].width, point_cloud_resolutions[i].height, sl.MAT_TYPE.F32_C4, sl.MEM.CPU))
    viewer.init(camera_infos[i].camera_model, point_cloud_resolutions[i], obj_param.enable_tracking)
    display_resolutions.append(sl.Resolution(min(camera_resolutions[i].width, 1280), min(camera_resolutions[i].height, 720)))
    image_scales.append([display_resolutions[i].width / camera_resolutions[i].width, display_resolutions[i].height / camera_resolutions[i].height])
    image_left_ocvs.append(np.full((display_resolutions[i].height, display_resolutions[i].width, 4), [245, 239, 239, 255], np.uint8))
    camera_configs.append(camera_infos[i].camera_configuration)
    tracks_resolutions.append(sl.Resolution(400, display_resolutions[i].height))
    track_view_generators.append(cv_viewer.TrackingViewer(tracks_resolutions[i], camera_configs[i].fps, init_params.depth_maximum_distance))
    track_view_generators[i].set_camera_calibration(camera_configs[i].calibration_parameters)
    image_track_ocvs.append(np.zeros((tracks_resolutions[i].height, tracks_resolutions[i].width, 4), np.uint8))
```

Concept of **FPS** and **Inference**

- Frames per Second (FPS) in object detection refers to the number of frames or images processed by the detection system in one second.
- In object detection using neural networks like YOLO, "inference" refers to applying a trained model to input data to make predictions or detections.
- Higher inference time, more computational resources.

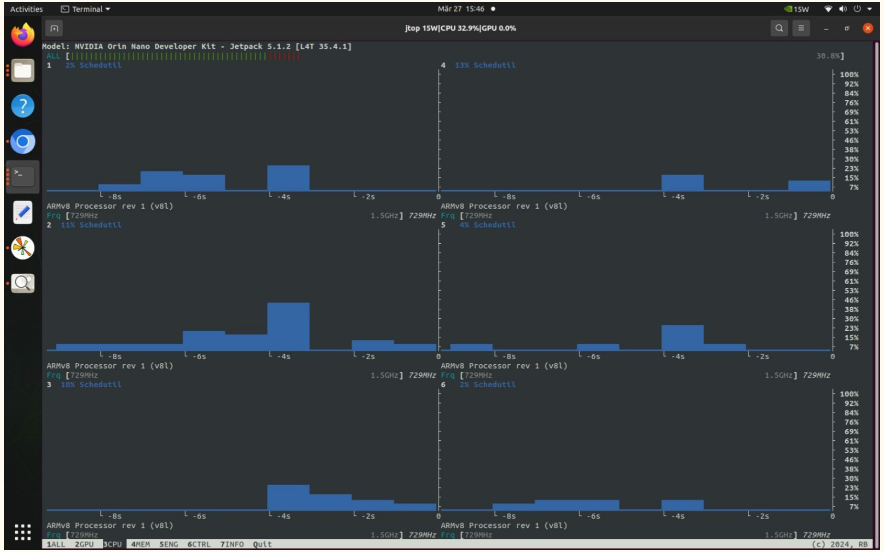
PERFORMANCE COMPARISON

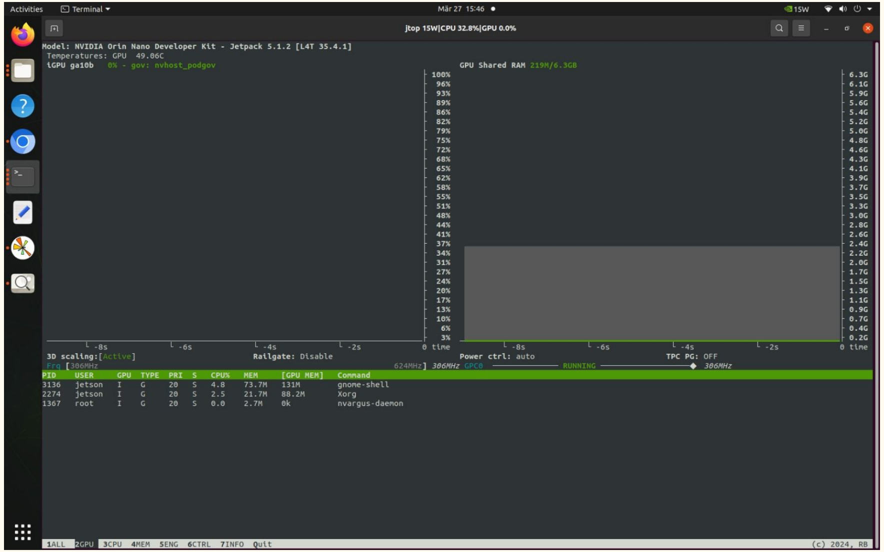


YOLOv8 models vs inference time bar graph

https://rb.gy/wtq56e

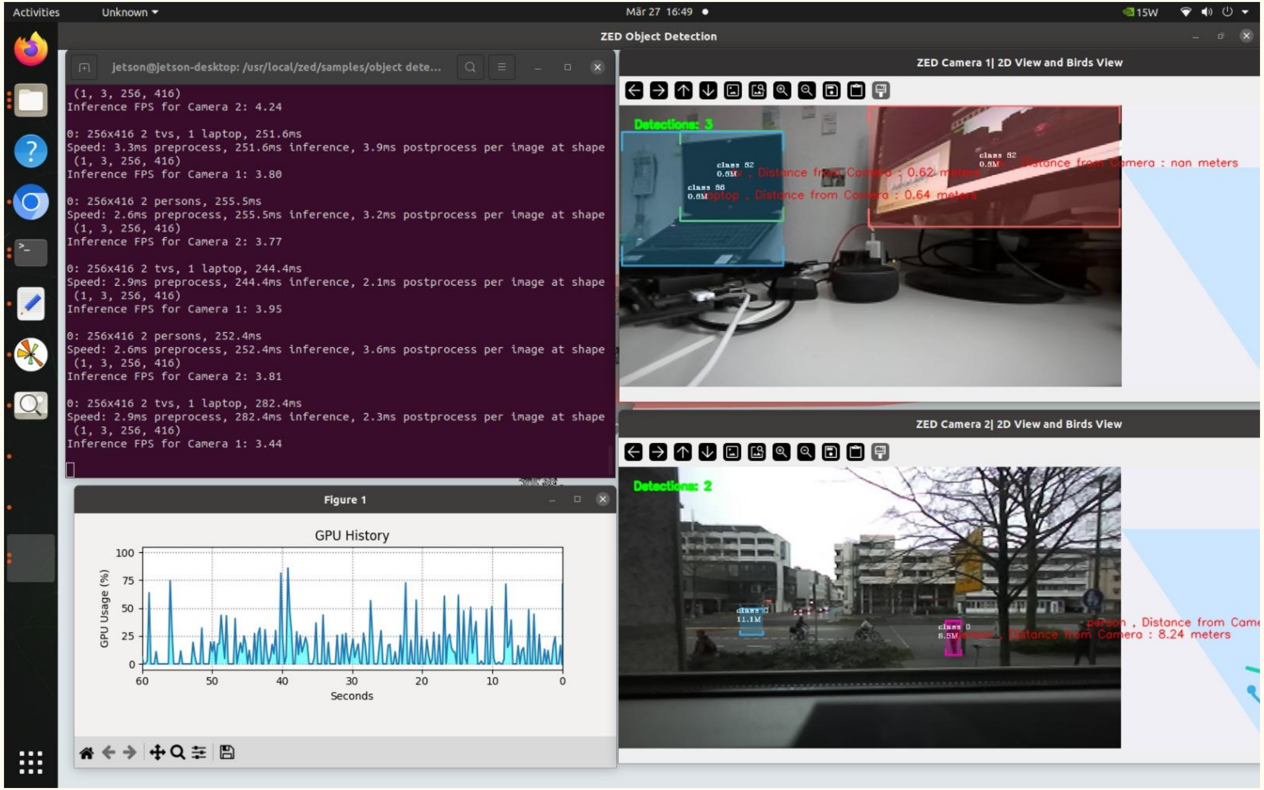
CPU and GPU status without object detection



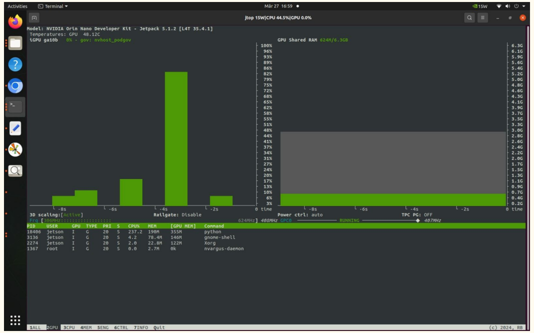


RESULTS

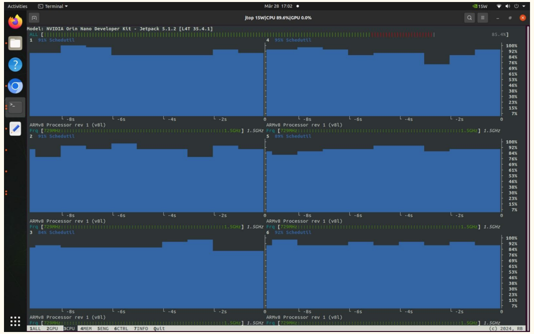
- **Using same models on both the cameras (YOLOv8n)**



GPU status

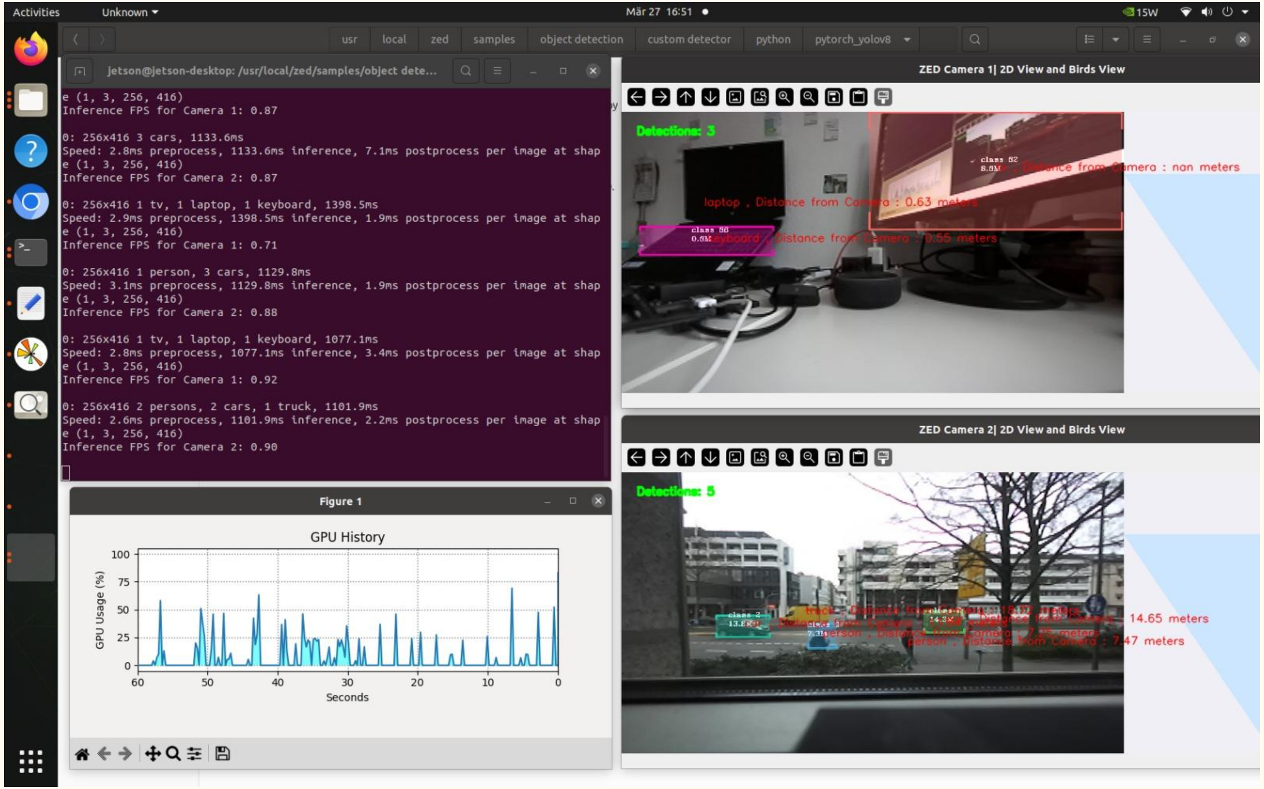


CPU status

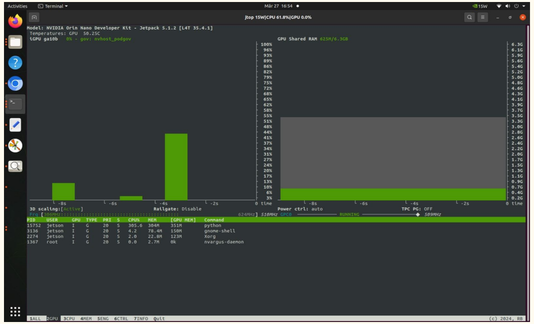


RESULTS

- Using same models on both the cameras (YOLOv8l)



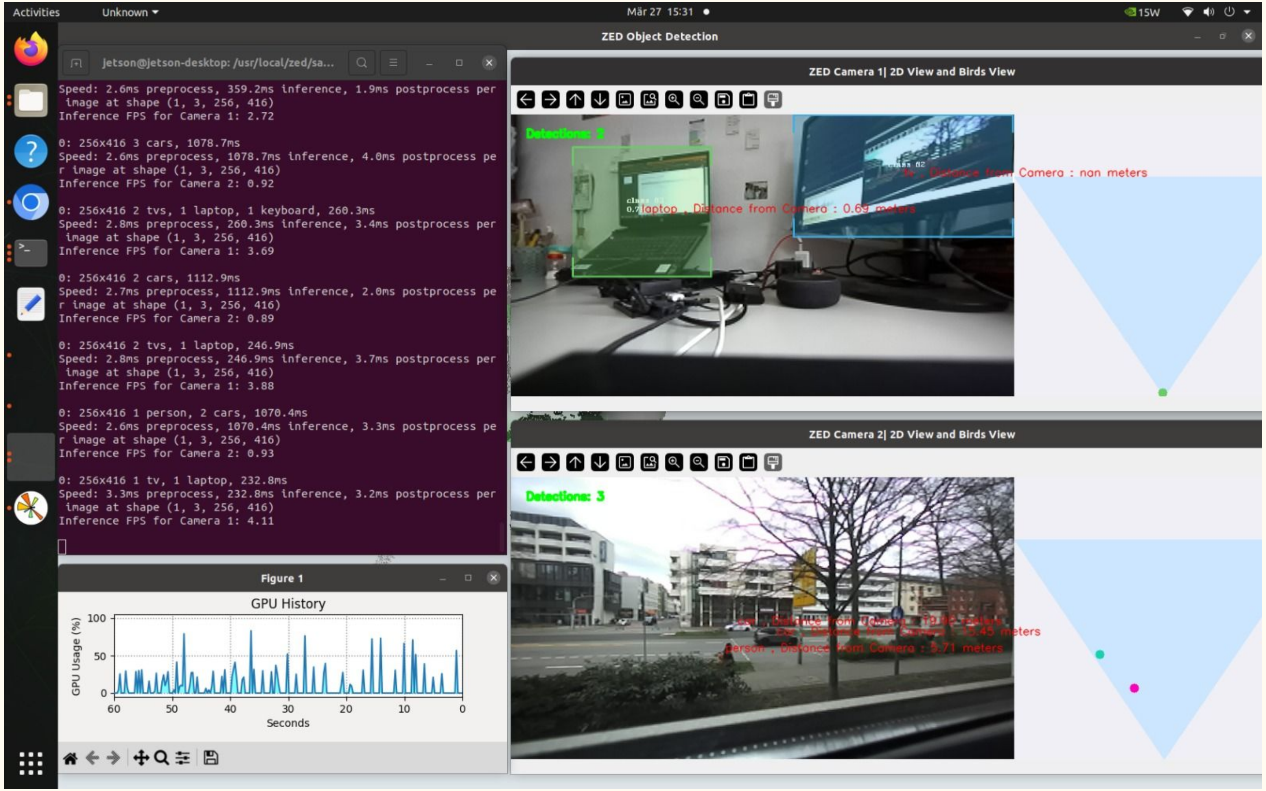
GPU status



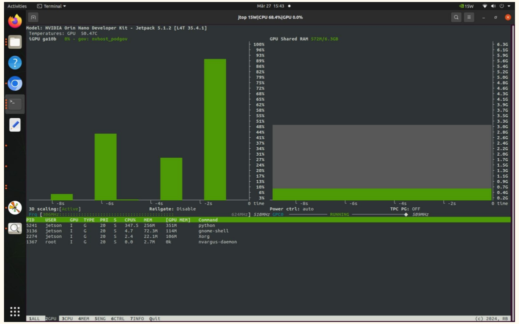
CPU status

RESULTS

- Using different models on both the cameras (Yolov8n and YOLOv8l)

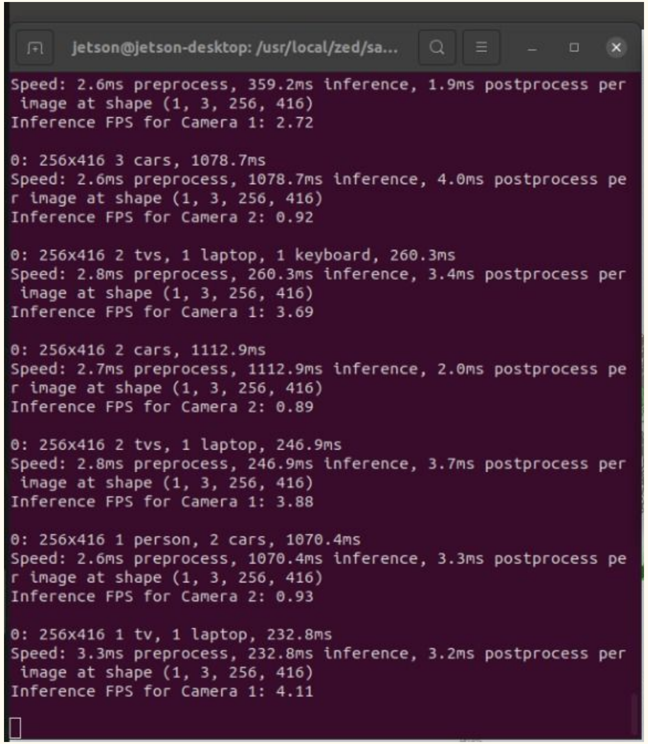


GPU status



CPU status

Observations based on the previous output




```
Speed: 2.6ms preprocess, 359.2ms inference, 1.9ms postprocess per
 image at shape (1, 3, 256, 416)
Inference FPS for Camera 1: 2.72

0: 256x416 3 cars, 1078.7ms
Speed: 2.6ms preprocess, 1078.7ms inference, 4.0ms postprocess pe
r image at shape (1, 3, 256, 416)
Inference FPS for Camera 2: 0.92

0: 256x416 2 tvs, 1 laptop, 1 keyboard, 260.3ms
Speed: 2.8ms preprocess, 260.3ms inference, 3.4ms postprocess per
 image at shape (1, 3, 256, 416)
Inference FPS for Camera 1: 3.69

0: 256x416 2 cars, 1112.9ms
Speed: 2.7ms preprocess, 1112.9ms inference, 2.0ms postprocess pe
r image at shape (1, 3, 256, 416)
Inference FPS for Camera 2: 0.89

0: 256x416 2 tvs, 1 laptop, 246.9ms
Speed: 2.8ms preprocess, 246.9ms inference, 3.7ms postprocess per
 image at shape (1, 3, 256, 416)
Inference FPS for Camera 1: 3.88

0: 256x416 1 person, 2 cars, 1070.4ms
Speed: 2.6ms preprocess, 1070.4ms inference, 3.3ms postprocess pe
r image at shape (1, 3, 256, 416)
Inference FPS for Camera 2: 0.93

0: 256x416 1 tv, 1 laptop, 232.8ms
Speed: 3.3ms preprocess, 232.8ms inference, 3.2ms postprocess per
 image at shape (1, 3, 256, 416)
Inference FPS for Camera 1: 4.11
```

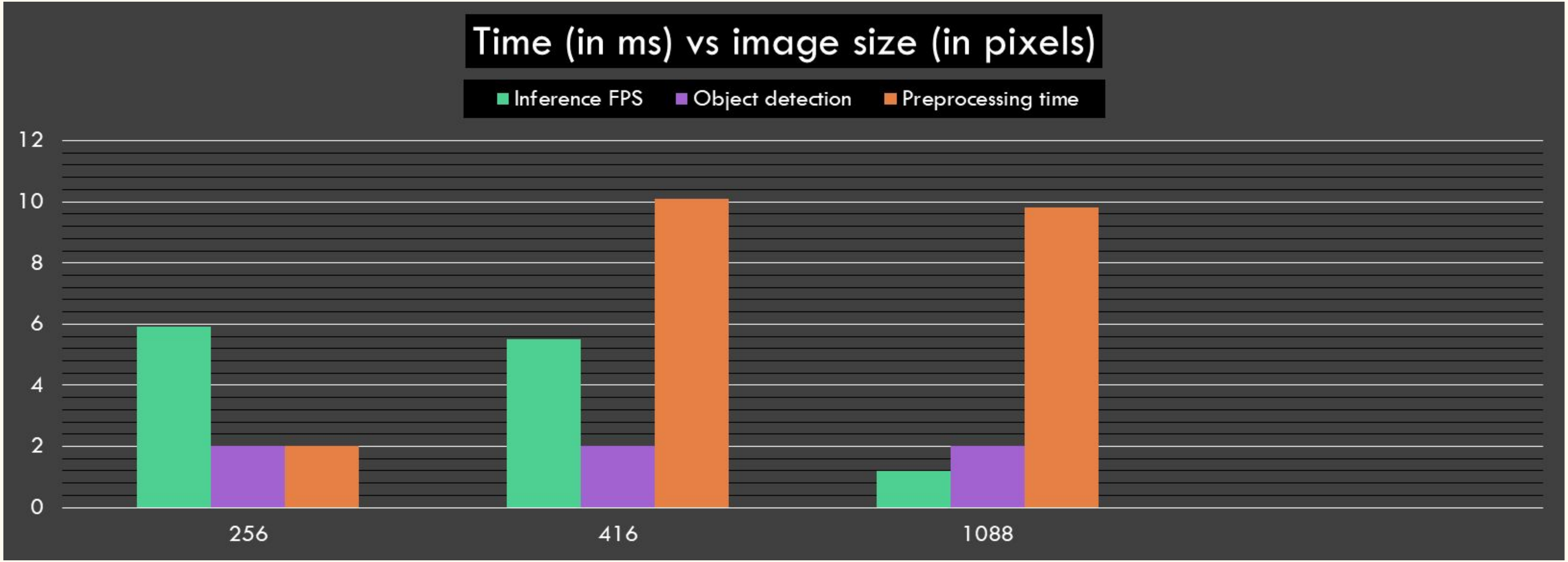
➢ Inference Time: YOLOv8l model requires more computational resources and time for inference due to its larger size and complexity.

➢ Detected Objects: the models are detecting different objects based on their respective features and capabilities.

➢ Inference FPS: the YOLOv8n model performs faster on Camera 1. This could be due to the smaller size and complexity of the YOLOv8n model compared to YOLOv8l.

➢ Model Size: The difference in model size and complexity can affect the performance and accuracy of object detection, as well as the computational resources required for inference.

Observations based on **variation of image size**



CONCLUSION

- Successfully implemented Object Detecting using two ZED 2i cameras.
- With the help of version 8 of YOLO algorithm, optimal performance is ensured.
- Flexible code.
- Performance comparison for various scenarios.
- Numerous applications such as smoke detection, autonomous driving, surveillance, and beyond.

GITHUB REPOSITORY: https://github.com/supriyac282/Jetson_Orin_Nano/tree/main

FUTURE WORK / PROJECT PROPOSAL

REAL TIME MULTI-OBJECT TRACKING

- Objective: Develop a real-time object tracking system.
- Application: To track a single object across the screen, e.g. person/car/object on a live feed.
- Object Tracking using Open cv’s built in trackers.
- Multi-Object Tracking using SORT algorithm.
- Explore whether to use Kalman filtering for improved accuracy if implementing in a noisy environment.

THANK YOU