SOFE 4590U: Embedded Systems (Fall 2023)

Instructor: Dr. Akramul Azim

Assignment #2: Performance evaluation of different architectures on QEMU

Name: Alden O'Cain

Banner ID#: 100558599

Demo Videos:

1. AMD64 Architecture [Host Machine - Win10/WSL Ubuntu] - Obstacle Detection (QEMU Performance Eval.)
2. i386 Architecture [VM #1 - Raspbian OS] - Obstacle Detection (QEMU Performance Eval.)
3. x86_64 Architecture [VM #2 - Ubuntu Server OS] - Obstacle Detection (QEMU Performance Eval.)

Links:

4. Project GitHub Repository - QEMU-ObstacleDetection-PerformanceEvaluation
5. YOLO: Real-Time Object Detection (pjreddie.com)
6. Miami 4K - Night Drive - YouTube
7. Driving Downtown - New York City 4K HDR - Downtown Manhattan - YouTube

Introduction:

In the realm of embedded systems, where precision and efficiency is important, the evaluation of hardware architectures plays a crucial role in determining the success of critical applications such as object detection. This report delves into the performance assessment of object detection using the YOLO (You Only Look Once) framework, across three distinct architectures: AMD64 (Host), as well as i386 (VM #1), and x86_64 (VM #2) both emulated on QEMU - Quick Emulator.

Code for Obstacle Detection:

The YOLO (You Only Look Once) framework uses Darknet which is an open source convolutional neural network. YOLO is a cutting-edge real-time object detection system that offers impressive processing speed. YOLOv3, a variant of the system, was selected for the obstacle recognition segment of the assignment because it combines exceptional speed and accuracy. YOLO's unique approach involves processing the entire image in a single pass, dividing it into regions, and predicting bounding boxes and probabilities for each region. These bounding boxes are further weighted by their predicted probabilities, setting YOLO apart from traditional detection systems that rely on multi-location and multi-scale image processing. For more detailed information about YOLO, please refer to the website YOLO- Real-Time Object Detection.

Sources of Data:

The data utilized for my object detection experiments consisted of video footage that closely mimics real-world scenarios. These videos were selected to assess the performance of object detection across different architectures under varying environmental conditions.

The following two video sources were used in my analysis:

> "Miami 4K - Night Drive":
> This video provides a captivating nighttime drive through the streets of Miami, featuring challenging lighting conditions, dynamic traffic, and diverse objects within the urban

landscape. It serves as a representative sample of scenarios where timely object detection is critical.

"Driving Downtown - New York City 4K HDR - Downtown Manhattan":
This video showcases a high-definition, high-dynamic-range (HDR) drive through the bustling streets of downtown Manhattan, New York City. It presents a rich urban environment with a multitude of objects and challenges, ideal for evaluating object detection capabilities in an urban setting.

Both videos were downloaded at a resolution of 720p and a frame rate of 30 FPS to maintain consistent input conditions for my experiments. These video sources were chosen for their ability to simulate real-world scenarios with diverse objects and lighting conditions, providing a foundation for assessing the performance of my object detection program on different hardware architectures.

Data Preparation:

To adapt the video data for use with the obstacle detection model, a multi-step process was followed, taking into account the video sections selected for analysis. Since the obstacle detection model operates on individual images, the chosen video segments needed to be transformed into a format suitable for image processing.

1. **Video Segmentation:**
   For the "Miami 4K - Night Drive" video, a specific section from 52 minutes and 46 seconds to 54 minutes and 21 seconds was selected.
   For the "Driving Downtown - New York City 4K HDR - Downtown Manhattan" video, a portion from 46 minutes and 20 seconds to 47 minutes and 26 seconds was chosen.

   A custom Python script, located in the 'video-cropper' directory of the project's GitHub repository, was utilized to extract these segments from the source videos.

2. **Frame Extraction:**
   To convert these video sections into individual images, another Python script stored in the 'video-frame-grabber' directory of the GitHub repository was employed.

   This script processed the smaller video sections and extracted frames at a consistent frame rate, ensuring that each image represented a specific point in time within the video.

3. **Dataset Size Reduction:**
   Given the large number of frames generated (2,850 for the Miami video and 1,980 for the New York video), an additional Python script was created and stored in the 'data-set-size-reducer' directory of the project's GitHub repository.

This script allowed for the reduction of the dataset size by selecting every Nth image, where N was set to 90. The choice of 90 corresponds to saving an image approximately every 3 seconds of video (considering a 30 FPS frame rate).
This reduction significantly downsized the dataset to 32 images for the Miami video and 22 images for the New York video, making it more manageable and timely for testing.

The data preparation process transformed the selected video segments into two datasets of images suitable for object detection. This approach enabled efficient testing and evaluation of the obstacle detection program on different hardware architectures without overwhelming computational or time requirements, allowing each environment to be tested in under half an hour, facilitating a streamlined assessment process.

Running Object Detection on the Dataset:

Running the object detection model on the dataset posed a unique challenge, as the model inherently processes images individually and lacks native support for video files or batch processing without manual data entry. To address this, a tailored driver code was created, facilitating the systematic evaluation of each image within each dataset. This driver code can be found within the 'darknet-driver' directory of my GitHub repository.

Firstly, a specific dataset and image type are selected for analysis, with chosen directories such as 'new_york_data' and designated file types like '.jpg'. The driver code then automatically compiles a list of images within the chosen directory, prepared for obstacle detection. The driver code then iteratively processes each image within the list, executing the darknet object detection command for individual images in a bash shell environment. This command starts the object detection process on an individual image and saves key information about the detected objects and prediction times by stored these outputs in dedicated output files, for example, driver-output_miami_data.txt.' This approach of conducting object detection on the dataset ensured a systematic and automated assessment of each image, offering valuable insights into the model's performance across various architectures and datasets.

File Transfer Method:

Transferring the output files generated during object detection from the emulated QEMU machines back to the host machine proved to be a non-trivial task. The QEMU virtual machines had the capability to ping the host machine, ensuring connectivity in one direction, but the reverse path wasn't as straightforward. This posed a challenge in transferring data from the QEMU machines to the host machine, which was essential for data analysis and further evaluation.

To address this hurdle, a solution was devised by using a basic Flask server, a lightweight web framework written in Python. This Flask server was run on the local network and can be found within the 'file-server' directory of the project’s GitHub repository.

The Flask server was designed to accept POST requests and provided a simple yet effective means for the QEMU virtual machines to transmit their output files back to the host machine. The transfer process involved using the 'curl' command from the QEMU machine's terminal, specifying the appropriate HTTP POST request. For example, a command like:
' *curl -X POST -F 'file=@filename.txt' http://host_private_ip:port_num/upload* ' was used.

This method established a reliable method for file transfer, allowing the QEMU machines to send the output files to the host machine without the complexity of configuring network settings. The Flask server acted as an intermediary, facilitating the exchange of data between the virtual machines and the host, ensuring that the critical output files were readily accessible for subsequent analysis and assessment.

Data Parsing:

To streamline the extraction of valuable information from the output files generated during the object detection process, the 'darknet-csv-parser' script was created which can also be found in the project's GitHub repository. This script serves a critical role in systematically parsing through the output files, capturing the filename and prediction time for each image in a dataset, and organizing them into a structured CSV format for further analysis.

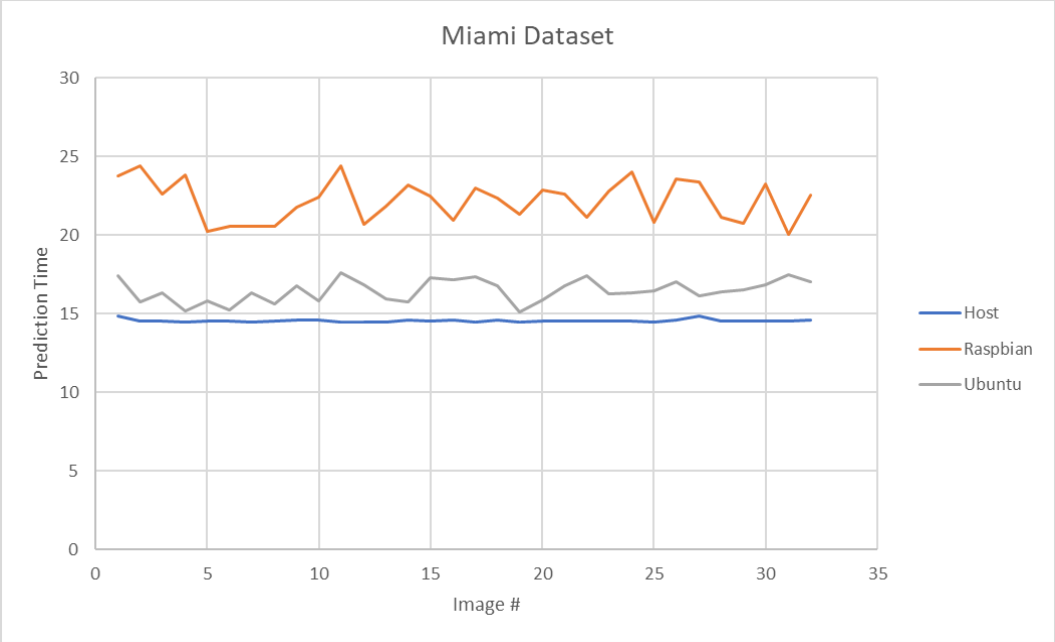
The CSV file generated by 'darknet-csv-parser' serves as a valuable resource for subsequent data analysis and visualization, allowing for a comprehensive evaluation of object detection performance across different environments.

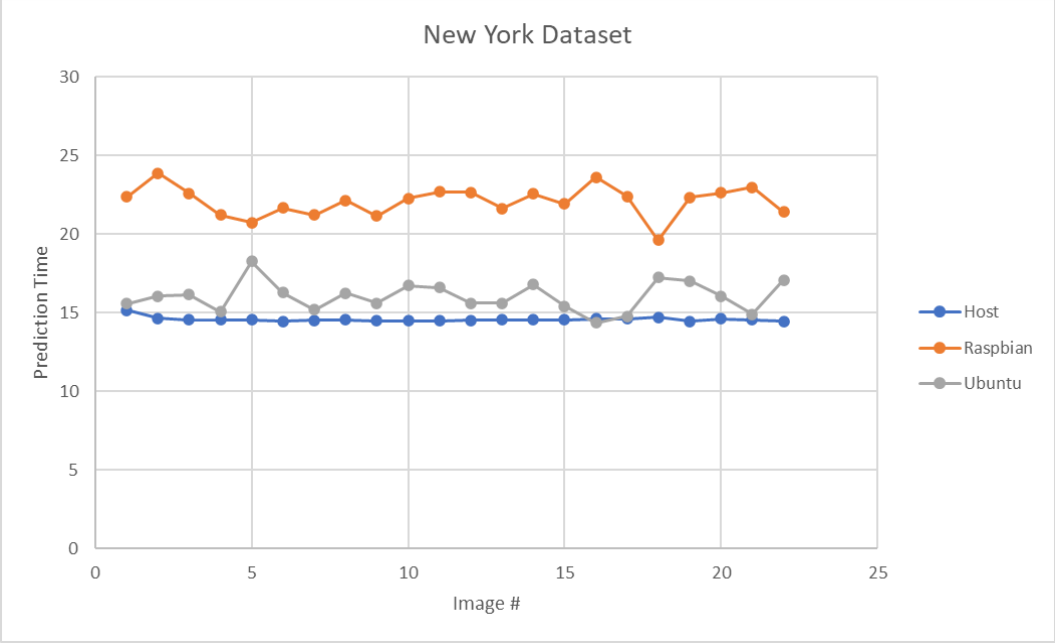
Data Analysis and Visualization:

The parsed CSV data, containing image filenames and their corresponding prediction times across various architectures and datasets, was collated and compiled into an Excel file named 'data.xlsx,' which resides in the root of the project's GitHub repository. This step was essential to centralize and structure the data for in-depth analysis and presentation.

To offer a comprehensive overview of the object detection performance, two XY scatter graphs were created, one for each dataset, 'New York' and 'Miami.' These graphs distinctly represent the prediction times for each image within the datasets and provide valuable insights into the variations observed among different hardware architectures. Specifically, the graphs feature three separate lines, each corresponding to a distinct machine configuration: 'Host,' 'Raspbian,' and 'Ubuntu.'

To further enhance the clarity of the analysis, the average prediction times for all data, 'New York,' and 'Miami' datasets were calculated. These averages, captured in the table below the graphs, offer a concise summary of the object detection performance on each machine configuration, aiding in the identification of trends and differences among the architectures.





Averages	All Data	New York	Miami
Host	14.56451	14.57635	14.55637513
Raspbian	22.15328	22.08749	22.19851291
Ubuntu	16.28641	16.02967	16.46291122

Conclusion:

In embedded systems, where precision and efficiency can be crucial, the choice of hardware architecture becomes important in shaping the outcome of critical applications, particularly in the context of object detection. This report explored the performance of object detection using the YOLO (You Only Look Once) framework, specifically the open source convolutional neural network, Darknet. Three distinct architectures were examined: AMD64 (Host), i386 (VM #1), and x86_64 (VM #2) within QEMU.

It is evident that there are distinct differences in their object detection performance. Unsurprisingly, the host machine, utilizing the AMD64 (WSL/Ubuntu running on Windows 10) architecture, emerged as top performer in this real-time object detection evaluation. Following closely behind was the x86_64 Ubuntu Server emulated in QEMU, with respectable speed and accuracy. Conversely, the i386 Raspbian OS-based VM presented a notably slower performance, likely attributed to its more constrained resource allocation, especially in terms of RAM. It is worth noting that the host machine had 32 gb of RAM available total, the Ubuntu VM, 8 gb and the Raspbian only 3 gb due to limitations imposed by either the operating system or the i386 architecture.