

Instituto Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica



Tutorial de síntesis e instalación

Taller de Sistemas Embebidos

Integrantes:

David Medina Mayorga

Carlos Quirós Gómez

Profesor:

Dr. Ing. Johan Carvajal Godínez

6 de abril de 2024

Índice

| | |
|--|-----------|
| 1. Computador Host | 2 |
| 2. Toolkits | 2 |
| 3. Descripción de los Toolkits | 2 |
| 3.1. OpenVINO | 2 |
| 3.2. OpenCV | 3 |
| 3.3. Yocto Project | 4 |
| 3.4. VirtualBox | 4 |
| 4. Descripción del flujo de trabajo | 5 |
| 4.1. Aplicación a implementar | 5 |
| 4.1.1. Prueba de los programas en el computador host | 7 |
| 4.2. Generacion de la imagen de linux a la medida | 9 |
| 4.2.1. Instalacion de Yocto Project | 9 |
| 4.2.2. Recetas de yocto necesarias para la sintesis | 11 |
| 4.3. Cargar la imagen generada en Virtual Box | 15 |
| 5. Selecccion de aplicación | 19 |
| 6. Dependencias | 19 |
| 6.1. Lista de dependencias | 19 |
| 6.2. Mapeo de dependencias | 19 |

1. Computador Host

- Sistema Operativo: Ubuntu 22.04.4 LTS 64-bit
- Procesador: Intel Core i5-1035G1
- Memoria RAM: 8 GB
- Almacenamiento: 1.5 TB

2. Toolkits

Para este proyecto se utilizo:

- OpenVino
- OpenCV
- Yocto Project (Poky)
- Virtualbox
- Micromamba

Estos son necesarios tanto para la síntesis e instalación de la imagen como para las pruebas en la computadora host.

3. Descripción de los Toolkits

En esta sección se describen los distintos toolkits, sus características y requisitos de hardware y software.

3.1. OpenVINO

OpenVINO (Open Visual Inference and Neural Network Optimization) es un toolkit de software desarrollado por Intel que facilita el despliegue y optimización de modelos de inteligencia artificial (IA) y visión por computadora en una variedad de dispositivos, incluyendo CPUs, GPUs, FPGAs y VPUs. Entre sus características están:

1. Optimización de modelos: Ofrece herramientas para optimizar modelos de IA provenientes de diversos frameworks, garantizando un rendimiento óptimo durante la ejecución.
2. Soporte multi-dispositivo: Compatible con CPUs, GPUs, FPGAs y VPUs de Intel, brindando flexibilidad en la implementación de soluciones de IA.

3. Librerías optimizadas: Incluye librerías optimizadas que aceleran las operaciones de inferencia, mejorando la eficiencia de las aplicaciones de visión por computadora y IA.
4. Soporte para múltiples lenguajes: Proporciona interfaces de programación en C++, Python y Java, facilitando la integración de capacidades de IA en diversas aplicaciones.
5. Despliegue escalable: Permite implementar soluciones de IA desde dispositivos individuales hasta despliegues en la nube, asegurando una implementación flexible y eficaz en cualquier entorno.

3.2. OpenCV

OpenCV (Open Source Computer Vision Library) es una biblioteca de código abierto diseñada para abordar problemas relacionados con la visión por computadora y el procesamiento de imágenes. Algunas de sus características clave incluyen:

1. OpenCV ofrece una amplia gama de algoritmos y herramientas para el procesamiento de imágenes y visión por computadora, incluyendo manipulación de imágenes, detección de objetos, seguimiento de objetos, calibración de cámaras, entre otros.
2. Eficiencia y rendimiento: Está optimizado para aprovechar al máximo el hardware subyacente, lo que resulta en un rendimiento eficiente incluso en sistemas con recursos limitados.
3. Compatibilidad multiplataforma: OpenCV es compatible con varios sistemas operativos, incluyendo Windows, Linux, macOS, Android y iOS, lo que lo hace adecuado para una amplia gama de aplicaciones y entornos de desarrollo.
4. Interfaces de programación: Proporciona interfaces de programación en varios lenguajes, incluyendo C++, Python y Java, lo que facilita su integración en diferentes entornos de desarrollo y aplicaciones.
5. Licencia de código abierto: OpenCV se distribuye bajo la licencia BSD, lo que permite su uso gratuito en aplicaciones comerciales y de código cerrado, así como la modificación y distribución del código fuente.

3.3. Yocto Project

Yocto Project es una herramienta de código abierto diseñada para facilitar la creación de imágenes a medida de sistemas operativos Linux para dispositivos embebidos. Utilizando una metodología de capas y recetas, Yocto Project proporciona un entorno de construcción modular que permite seleccionar y configurar los componentes del sistema de manera precisa. Esto incluye la selección del kernel, las bibliotecas, las herramientas y las aplicaciones que formarán parte de la imagen final del sistema. Además, Yocto Project automatiza el proceso de construcción, lo que facilita el mantenimiento y la reproducción de los entornos de desarrollo. Sus características principales incluyen:

1. Construcción de sistemas embebidos: Permite la creación de sistemas operativos Linux personalizados para dispositivos embebidos, adaptados específicamente a las necesidades del proyecto.
2. Flexibilidad y modularidad: Ofrece un enfoque modular que permite seleccionar y configurar los componentes del sistema según los requisitos específicos del dispositivo, lo que garantiza una flexibilidad excepcional en el desarrollo.
3. Automatización del proceso de construcción: Proporciona herramientas y scripts que automatizan el proceso de construcción del sistema, facilitando el mantenimiento y la reproducción de los entornos de desarrollo.
4. Soporte para múltiples arquitecturas: Es compatible con una amplia variedad de arquitecturas de procesadores, lo que lo hace adecuado para una amplia gama de dispositivos embebidos, desde microcontroladores hasta sistemas de alto rendimiento.
5. Comunidad activa y documentación detallada: Cuenta con una comunidad activa de usuarios y desarrolladores que brindan soporte y comparten recursos, además de una documentación detallada que facilita su aprendizaje y uso.
6. Licencia de código abierto: Se distribuye bajo una licencia de código abierto, lo que permite su uso gratuito y la modificación del código fuente según las necesidades del proyecto.

3.4. VirtualBox

VirtualBox es un software de virtualización de código abierto que permite a los usuarios crear y ejecutar máquinas virtuales en sus sistemas informáticos. Con VirtualBox, los usuarios pueden crear entornos virtuales aislados que emulan hardware real, lo que les permite ejecutar múltiples sistemas operativos en un único equipo físico. Características principales de VirtualBox:

1. Multiplataforma: Disponible para una amplia variedad de sistemas operativos host, incluyendo Windows, macOS, Linux y Solaris.
2. Soporte para múltiples sistemas operativos invitados: Permite ejecutar una amplia variedad de sistemas operativos target, incluyendo versiones de Windows, Linux, macOS y otros sistemas operativos menos comunes.
3. Virtualización anidada: Permite la virtualización de máquinas virtuales dentro de otras máquinas virtuales, útil para entornos de desarrollo y pruebas.
4. Interfaz gráfica intuitiva: Ofrece una interfaz gráfica fácil de usar para gestionar y controlar las máquinas virtuales de manera eficiente.
5. Extensiones de Oracle VM VirtualBox: Proporciona extensiones que añaden funcionalidades adicionales, como soporte USB 2.0/3.0, aceleración 3D y más.

4. Descripción del flujo de trabajo

En esta sección se describe el uso que se le da a los toolkits en las distintas etapas del desarrollo del proyecto.

4.1. Aplicación a implementar

La aplicación fue desarrollada a partir de códigos fuente ya existentes, se realizó una combinación de un programa para la detección de las líneas del carril obtenido en [1] y para la detección de objetos se partió del código que se encuentra en [2]. Se redujeron las dependencias al mínimo y se realizaron las modificaciones necesarias para integrar ambos códigos en una sola aplicación.

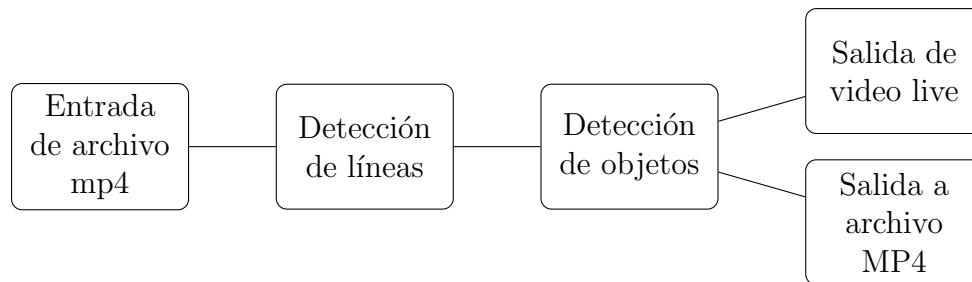
Todo el código fuente y metadata que se desarrolló se obtiene al clonar el siguiente repositorio [3]:

```
$ git clone https://github.com/DJosueMM/computer_vision_linuxImage.git
```

La aplicación se compone de dos programas escritos en Python:

- **lane_and_object_detection.py**: este programa realiza todo el procesamiento de detección de líneas de carril y objetos en la carretera.

El pipe de procesamiento se crea con OpenCV, este tiene la siguiente topología:



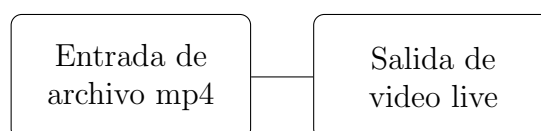
Se tiene de entrada un archivo mp4 definido por el usuario, este ingresa al pipe frame por frame. Cada frame pasa por cada bloque de procesamiento y al final es mostrado en la pantalla a tiempo real y a la vez se van almacenando en un archivo mp4 de salida.

El bloque de procesamiento ‘Detección de líneas’ calibra la cámara o el frame con el archivo ‘calibration_pickle.p’, luego por medio de numpy se realiza una transformación de la perspectiva obteniendo una imagen de vista de águila y en escala de grises. Después se generan líneas y rectángulos de colores entre las líneas obtenidas en el frame terminando así el procesamiento. La explicación detallada del funcionamiento se encuentra en [4]

El bloque de procesamiento ‘Detección de objetos’ utiliza un modelo de IA ya entrenado llamado ‘mssdlite_mobilenet_v2_fp16’, este se encuentra en los archivos fuente del repositorio y la documentación se puede revisar en [5]. En el programa se compila y se obtienen los pesos. Con el motor de inferencia de OpenVINO se procesa cada frame y cuando una detección supera el 40 % de umbral se dibuja un cuadrado de color que encierra el objeto detectado junto con el nombre del objeto. Al terminar este procesamiento se retorna el frame con los cuadros superpuestos.

Por último, se tiene dos opciones de salida:

- Se muestra el frame en la pantalla del usuario en tiempo real.
 - Se graba cada frame en un archivo de video mp4 para su posterior visualización.
- **video_player.py:** Este programa utiliza OpenCV para reproducir los videos de entrada o los que resultan del procesamiento de la detección de líneas y objetos. Esto es muy útil para la imagen creada a medida, ya que no se depende de reproductores de video en la imagen. El pipe generado es el siguiente:



Las dependencias de la aplicación son los siguientes módulos y librerías de Python3:

- cv2 (OpenCV)
- numpy
- opencvino
- pickle
- pathlib

Pathlib y Pickle vienen por defecto en Python3. Otras dependencias son los archivos de calibración del video de entrada, el modelo de IA y los videos .mp4 que se utilizarán en la aplicación, in embargo, estos están junto con el código fuente en un mismo paquete.

4.1.1. Prueba de los programas en el computador host

Para probar el funcionamiento del script en el computador host primeramente se instala micromamba, utilizando el siguiente comando en la terminal:

```
$ "${SHELL}" <(curl -L micro.mamba.pm/install.sh)
```

Se cierra la terminal y se abre otra y se crea un ambiente en micromamba:

```
$ micromamba create -n embebidos_proy1
```

Se debe revisar, estar dentro de este ambiente en la terminal:

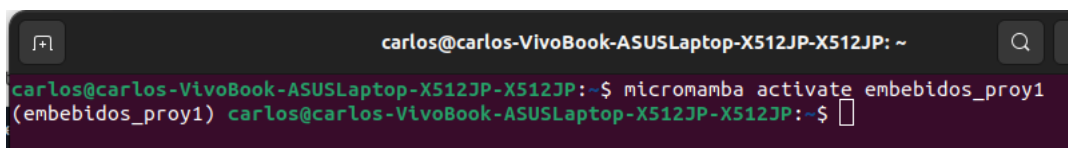


Figura 1: Terminal de ubuntu con el ambiente activo

Se instalan los paquetes necesarios:

```
$ micromamba install numpy
$ micromamba install -c conda-forge opencv
$ micromamba install -c conda-forge ocl-icd-system
$ micromamba install opencvino=2023.3
```


Con lo anterior instalado solo queda ubicarse dentro de la carpeta del proyecto y correr el script:

```
$ python3 lane_and_object_detection.py
```

Se ingresa el archivo mp4 que se desea procesar, como se muestra en la Figura. 2.

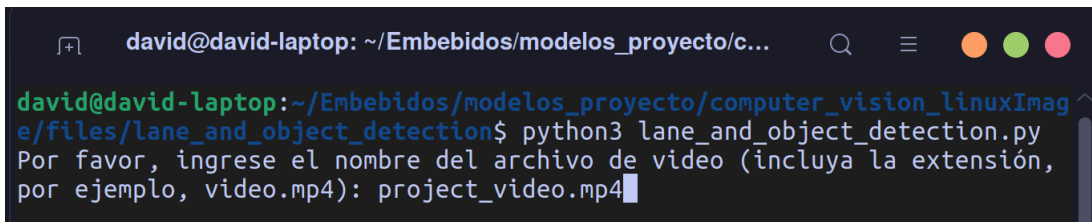


Figura 2: Terminal de ubuntu con el programa en ejecucion

Esto abrira una ventana con la deteccion de lineas de calle y detección de objetos en el video seleccionado:

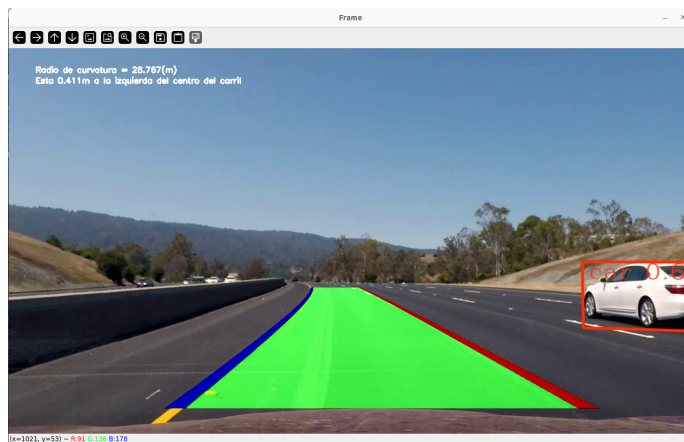


Figura 3: Programa de deteccion de lineas y objetos en ejecucion

Si se desea procesar otro video del proyecto, en el archivo 'lane_and_object_detection.py' se deben ajustar 4 parámetros en la detección de líneas estos se encuentran ya definidos en la función 'process_image':

Tabla 1: Tabla de valores ya probados

| Video (.mp4) | project_video | sol | test_video_1 | tunel | que | forest |
|--------------|---------------|-------|--------------|-------|-------|--------|
| bot_width | 0.75 | 0.552 | 0.56 | 0.392 | 0.547 | 0.432 |
| mid_width | 0.1 | 0.165 | 0.12 | 0.118 | 0.038 | 0.078 |
| height_pct | 0.62 | 0.26 | 0.68 | 0.70 | 0.53 | 0.678 |
| bottom_trim | 0.935 | 0.70 | 0.1 | 0.935 | 0.935 | 0.935 |

Para reproducir el archivo mp4 generado o cualquier otro que se encuentre en el directorio se puede ejecutar el siguiente script:

```
$ python3 video_player.py
```

Se ingresa el archivo mp4 que se desea visualizar, como se muestra en la Figura. 4.

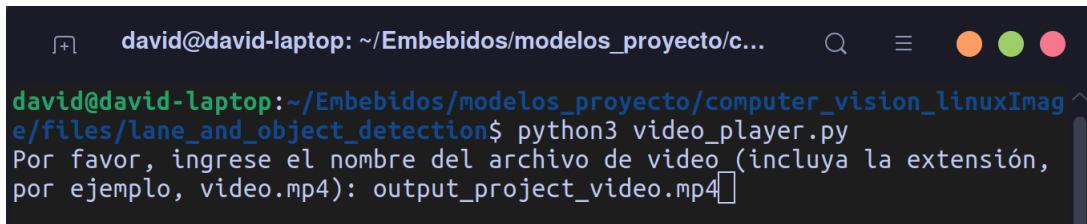


Figura 4: Terminal de ubuntu con el programa de reproduccion de video en ejecucion

Y con esto se concluye la prueba de la aplicacion en el computador host.

4.2. Generacion de la imagen de linux a la medida

Para generar la imagen de linux que contenga la aplicacion y todas sus dependencias se utilizara yocto-project.

4.2.1. Instalacion de Yocto Project

Para poder instalar Yocto se debe cumplir con los siguientes requisitos en el computador [6]:

- Mínimo 90 GB de espacio libre en el disco duro.
- Mínimo 8 GB de RAM.
- Una distribución de linux soportada para yocto. (Fedora, openSUSE, CentOS, Debian o Ubuntu).
- Git 1.8.3.1 o una versión mayor.

- tar 1.28 o una versión mayor.
- Python 3.8.0 o una versión mayor.
- gcc 8.0 o una versión mayor.
- GNU make 4.0 o una versión mayor.

Si el computador cumple con los requisitos anteriores se pueden instalar los siguientes host packages:

```
$ sudo apt install gawk wget git diffstat unzip texinfo gcc build-essential chrpath socat cpio python3 python3-pip python3-pexpect xz-utils debianutils iputils-ping python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev python3-subunit mesa-common-dev zstd liblz4-tool file locales libacl1
$ sudo locale-gen en_US.UTF-8
```

Se abre una terminal y se clona el repositorio de poky:

```
$ git clone git://git.yoctoproject.org/poky
```

Una vez clonado este repositorio, entramos en este directorio, se cambia a la rama de nanbield y se actualiza. Esto se hace con los siguientes comandos:

```
$ cd poky
$ git checkout -t origin/nanbield -b my-nanbield
$ git pull
```

Es importante fijarse estar en el branch adecuado:

```
$ git branch
```

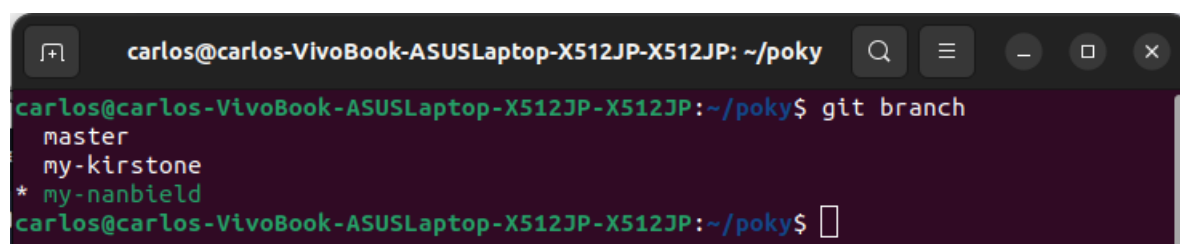


Figura 5: Verificación de branch a utilizar

Estando en el directorio de poky se realiza la comprobación de que se realizó la instalación:

```
$ source oe-init-build-env
```

Esto nos ubicará dentro de un directorio llamado **build**. Para realizar la construcción de la imagen realizaremos una modificación en el archivo **local.conf**, para lo cual nos ubicaremos dentro del directorio **conf**:

```
$ cd conf  
$ ls
```

Con un editor de preferencia se pueden descomentar las siguientes líneas en el archivo 'local.conf'. El criterio para hacer esto es que si la conexión a internet es lo suficientemente rápida para descargar algunos paquetes en un menor tiempo del que se tardaría si se contruyen localmente:

```
BB_HASHSERVE_UPSTREAM = "hashserv.yocto.io:8687"  
SSTATE_MIRRORS ?= "file://.* http://cdn.jsdelivr.net/yocto/sstate/all/  
    PATH;downloadfilename=PATH"  
BB_HASHSERVE = "auto"  
BB_SIGNATURE_HANDLER = "OEEquivHash"
```

Se guarda los cambios en este archivo y se retorna al directorio de /poky.

4.2.2. Recetas de yocto necesarias para la síntesis

En el directorio de poky, se clona el repositorio de meta-openembedded y se cambia al branch de nanbiel:

```
$ git clone https://git.openembedded.org/meta-openembedded  
$ cd meta-openembedded  
$ git checkout -t origin/nanbiel -b my-nanbiel  
$ git pull
```

Retornando al directorio de poky, se clona el repositorio de meta-intel y se cambia al branch de nanbiel:

```
$ git clone https://git.yoctoproject.org/git/meta-intel  
$ cd meta-intel  
$ git checkout -t origin/nanbiel -b my-nanbiel  
$ git pull
```

Una vez con estos repositorios clonados y en el branch correcto, se retorna al directorio de poky para poder crear una receta propia para la parte meter los archivos dentro de la imagen:

```
$ source oe-init-build-env
$ bitbake-layers create-layer ../meta-mylayer
$ cd ..
$ cd meta-mylayer
$ cd recipes-example
$ mkdir file-boost
$ cd file-boost
$ mkdir files
```

Utilizando el editor vim(se puede utilizar cualquier otro editor), creamos la receta file-boost.bb:

```
$ vim file-boost
```

Y se agregara lo siguiente, dentro de este archivo:

```
SUMMARY = "Recipe to include example.py and example.mp4"
LICENSE = "CLOSED"

SRC_URI = "file://lane_and_object_detection"

S = "${WORKDIR}"

do_install() {
    install -d ${D}${bindir}/lane_and_object_detection
    cp -r lane_and_object_detection/* ${D}${bindir}/lane_and_object_detection
}
```

La carpeta de meta-mylayer nos debe quedar de la siguiente forma:

Dentro de la carpeta **files**, se debe copiar los archivos del proyecto los cuales se encuentran en el repositorio.

Retornando al directorio de poky, en **bblayers.conf** se agrega los directores utilizados de la siguiente forma:

```
$ cd build
$ cd conf
$ vim bblayers.conf
```

Donde debe quedar de la siguiente forma:

```
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

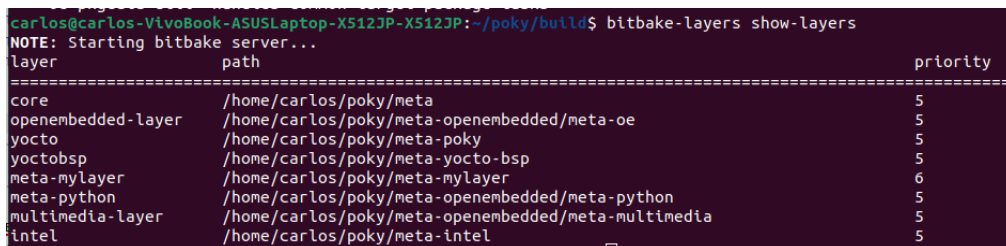
BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
/home/carlos/poky/meta \
/home/carlos/poky/meta-openembedded/meta-oe \
/home/carlos/poky/meta-poky \
/home/carlos/poky/meta-yocto-bsp \
/home/carlos/poky/meta-mylayer \
/home/carlos/poky/meta-openembedded/meta-python \
/home/carlos/poky/meta-openembedded/meta-multimedia \
/home/carlos/poky/meta-intel \
"
```

Otra forma de realizar esto es utilizando el comando de bitbake, esto realiza retornando al directorio de poky:

```
$ source oe-init-build-env
$ bitbake-layers add-layer ../meta-openembedded/meta-oe
$ bitbake-layers add-layer ../meta-openembedded/meta-python
$ bitbake-layers add-layer ../meta-mylayer
$ bitbake-layers add-layer ../meta-openembedded/meta-multimedia
$ bitbake-layers add-layer ../meta-openembedded/meta-intel
```

Y se verifica que las layers se hayan agregado correctamente:



```
carlos@carlos-VivoBook-ASUSLaptop-X512JP-X512JP:~/poky/build$ bitbake-layers show-layers
NOTE: Starting bitbake server...
layer                                path                                priority
=====
core                                /home/carlos/poky/meta              5
openembedded-layer                  /home/carlos/poky/meta-openembedded/meta-oe  5
yocto                               /home/carlos/poky/meta-poky         5
yoctobsp                           /home/carlos/poky/meta-yocto-bsp     5
meta-mylayer                        /home/carlos/poky/meta-mylayer       6
meta-python                        /home/carlos/poky/meta-openembedded/meta-python  5
multimedia-layer                    /home/carlos/poky/meta-openembedded/meta-multimedia  5
intel                              /home/carlos/poky/meta-intel        5
```

Figura 6: Layers agregados

Después de comprobar su correcta configuración se procede a editar el **local.conf**, para agregar los paquetes necesarios:

```
$ vim local.conf
```

Primeramente se agrega lo que permite realizar el archivo en formato VMDK:

```
IMAGE_FSTYPES += "wic.vmdk"
```

Se agrega la receta creada para obtener los archivos en la imagen.

```
IMAGE_INSTALL:append = " file-boost"
```

Lo relacionado a python:

```
IMAGE_INSTALL:append = " openssh \  
    python3 \  
    python3-pip \  
    python3-numpy "
```

Relacionado a OpenCV:

```
IMAGE_INSTALL:append = " opencv"
```

Lo relacionado a openvino y gstreamer:

```
IMAGE_INSTALL:append = " openvino-inference-engine"  
IMAGE_INSTALL:append = " openvino-model-optimizer"  
# Enable building inference engine python API.  
# This requires meta-python layer to be included in bblayers.conf.  
PACKAGECONFIG:append:pn-openvino-inference-engine = " python3"  
# Include OpenVINO Python API package in the target image.  
IMAGE_INSTALL:append = " openvino-inference-engine-python3"  
LICENSE_FLAGS_ACCEPTED += "commercial"  
IMAGE_INSTALL:append = " gstreamer1.0"  
IMAGE_INSTALL:append = " gstreamer1.0-plugins-base"  
IMAGE_INSTALL:append = " gstreamer1.0-plugins-good"  
IMAGE_INSTALL:append = " gstreamer1.0-plugins-bad"  
IMAGE_INSTALL:append = " gstreamer1.0-plugins-ugly"  
IMAGE_INSTALL:append = " gstreamer1.0-libav"  
IMAGE_INSTALL:append = " gstreamer1.0-rtsp-server"  
IMAGE_INSTALL:append = " gstreamer1.0-vaapi"
```

Una vez que se terminan editar el local.conf, se debe realizar un cambio en una receta para esto retornando al directorio de poky:

```
$ cd meta-openembedded/meta-oe/recipes-support/opencv/  
$ vim opencv_4.8.0.bb
```

Para activar el dnn(necesario para el python utilizado) se debe buscar **PACKAGE-CONFIG[dnn]** y activarlo:

```
PACKAGECONFIG[dnn] = "-DBUILD_opencv_dnn=ON -DPROTOBUF_UPDATE_FILES=ON  
-DBUILD_PROTOBUF=OFF -DCMAKE_CXX_STANDARD=17,  
-DBUILD_opencv_dnn=ON,protobuf protobuf-native,"
```

Se retorna al directorio de poky y se realiza el bitbake para crear la imagen:

```
$ source oe-init-build-env  
$ bitbake core-image-x11
```

Al terminar este proceso se genera la imagen que se utilizara en virtual box.

4.3. Cargar la imagen generada en Virtual Box

Para una explicación detallada se puede seguir esta guía [7]

Primeramente se realiza la instalacion de VirtualBOX:

```
$ sudo apt-get install virtualbox
```

Se ejecuta el virtual box, y se crea una nueva maquina virtual, en el boton NEW y se llena la configuracion de la siguiente manera:

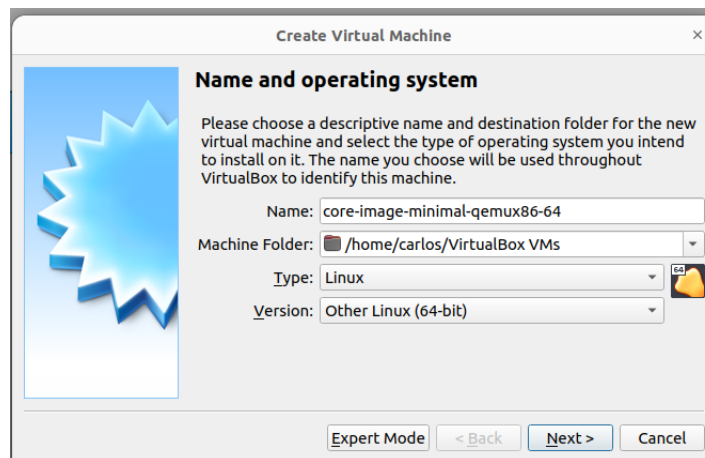


Figura 7: Creación de la maquina virtual

Seguimos con **Next** y se configura el tamaño de la memoria:

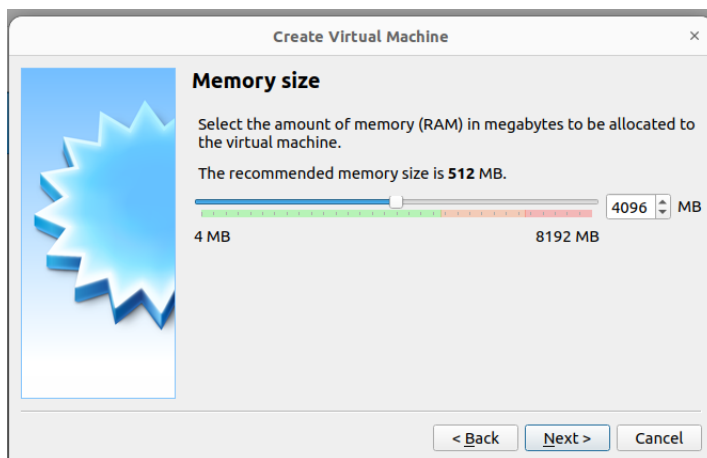


Figura 8: Configuración de memoria

Seguimos con **Next** y se selecciona la opción de no añadir disco duro:

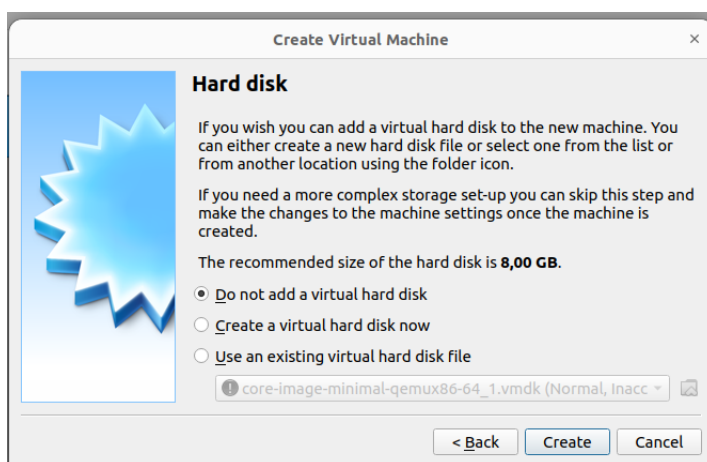


Figura 9: Configuración de disco duro

Al terminar de crear la máquina virtual esta presentará una advertencia a la cual se le debe dar **continúe** y esto nos creará la máquina virtual. Para cargar la imagen se debe de entrar en configuraciones:

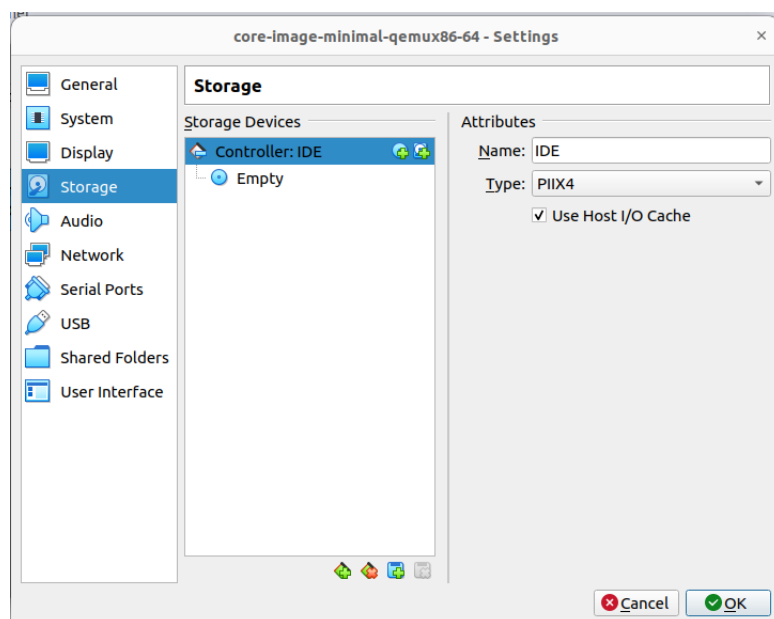


Figura 10: Configuración

En la parte de **Controller:IDE** en la segunda opción podemos agregar un disco duro:

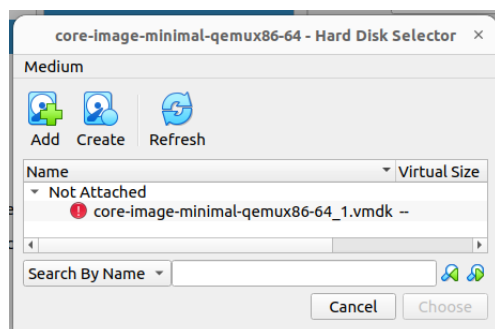


Figura 11: Añadiendo Disco Duro

Al añadir el disco este se debe buscar en la carpeta de `poky/build/tmp/images/qemux86-64` y se elige la siguiente:

`core-image-x11-qemux86-64.rootfs.wic.vmdk`

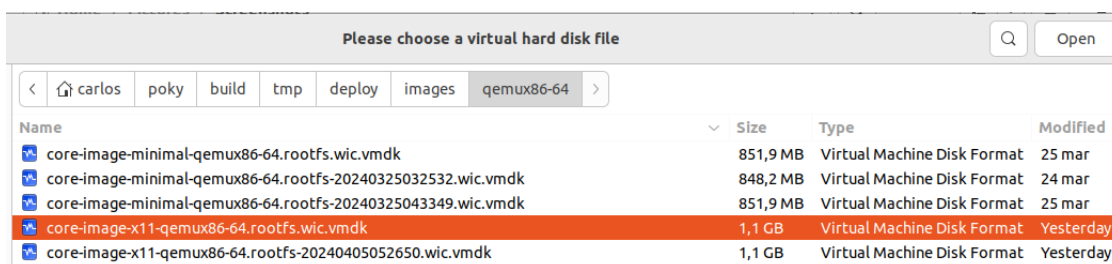


Figura 12: Añadiendo el archivo VMDK

Se termina de elegir la imagen y se puede iniciar la maquina virtual.

Para encontrar el python3 y correrlo se debe realizar lo siguiente:

```
$ cd ..
$ cd ..
$ cd usr/bin/lane_and_object_detection/
$ cd python3 lane_and_object_detection.py
```

y se elige el video **test_video_1.mp4** y se puede visualizar el resultado:

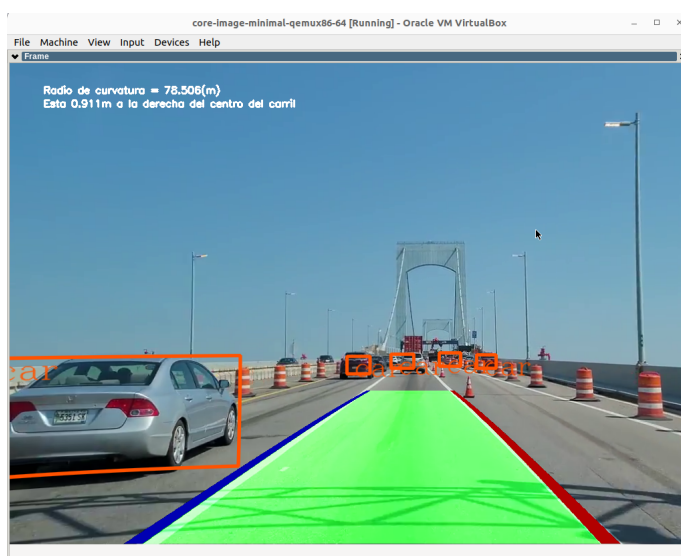


Figura 13: Añadiendo el archivo VMDK

5. Selección de aplicación

Para la parte de detección de líneas se utilizó OPENCV basado en un repositorio encontrado al cual se le realizó modificaciones para utilizar funciones de OpenCV y no alguna librería que pudiera dificultar instalarse en la imagen[8].

La detección de objetos se necesitaba algún modelo entrenado, por lo cual utilizando la herramienta OpenVino y sus ejemplos en su sitio web se utilizó, el SSDLITE MOBILENET V2, este modelo está entrenado para reconocer objetos en carreteras y carros por lo cual cumplía con lo requerido para este proyecto[5].

6. Dependencias

En esta sección se listan las dependencias de la aplicación y la imagen. También se muestra su mapeo en las capas y recetas implementadas.

6.1. Lista de dependencias

En esta lista se incluyen todas las dependencias del sistema:

- OpenCV
- Python3
- NumPy
- OpenVino
- Pickle
- Pathlib
- GStreamer
- Gestor de ventanas (X11 es el entorno gráfico para la mayoría de los sistemas Unix o similares.)
- Archivos fuente de la aplicación (Códigos en Python, modelos, archivos de calibración, videos de prueba.)

6.2. Mapeo de dependencias

Capas y recetas con las que se abordaron las dependencias:

- meta-oe: capa que contiene las recetas de OpenCV.

- meta-python: capa que contiene las recetas de Python3, NumPy, Pickle y Pathlib.
- meta-intel: capa que contiene las recetas de OpenVino.
- meta-multimedia: capa que contiene las recetas de GStreamer, que a su vez es dependencia de OpenCV.
- core-image-x11: al ejecutar el bitbake core-image-x11 se obtiene una imagen básica que incluye las recetas para el gestor de ventanas X11. Esto se puede ver en la documentación de la capa [9].
- meta-mylayer: contiene las recetas que incluyen todos los archivos fuente de la aplicación.

Referencias

- [1] S. R. Anumakonda. Advanced lane detection and object detection. [Online]. Available: <https://github.com/srianumakonda/Advanced-Lane-Detection-and-Object-Detection>
- [2] I. Corporation, “Object detection with openvino toolkit,” https://colab.research.google.com/github/openvinotoolkit/openvino_notebooks/blob/main/notebooks/401-object-detection-webcam/401-object-detection.ipynb, 2021.
- [3] D. Medina. Computer vision linux image. [Online]. Available: https://github.com/DJosueMM/computer_vision_linuxImage
- [4] S. Anumakonda, “Pairing lane detection with object detection,” <https://srianumakonda.medium.com/pairing-lane-detection-with-object-detection-665b30462952>, Mar 2021, medium, 11 min read.
- [5] Intel Corporation, “Ssd-lite mobilenet v2 model,” https://docs.openvino.ai/2022.3/omz_models_model_ssd-lite_mobilenet_v2.html, 2022.
- [6] L. Foundation and Y. Project. Yocto project quick build. [Online]. Available: <https://docs.yoctoproject.org/brief-yoctoprojectqs/index.html>
- [7] G. C. Macario, “Running a yocto image inside virtualbox,” <http://gmacario.github.io/posts/2015-11-14-running-yocto-image-inside-virtualbox>, 2015, accedido el 2 de abril de 2024.
- [8] G. Sung, “Advanced lane detection,” https://github.com/georgesung/advanced_lane_detection, 2017.
- [9] OpenEmbedded Layer Index, “core-image-x11 1.0 layer,” OpenEmbedded Layer Index, 2022, consultado el 5 de abril de 2024. [Online]. Available: <https://layers.openembedded.org/layerindex/recipe/350290/>