Canonical

Ubuntu 22.04 for NVIDIA Jetson Installation instructions

Canonical Devices and IoT, Partner Engineering

Purpose

Prerequisites

- Ubuntu image and boot firmware downloaded
- A laptop or PC (host: x86 architecture) running Ubuntu (following instructions have been tested with 22.04, but should be compatible with other versions as well)
- A monitor with DP port and an USB keyboard
- An USB-C (device side) to A cable (host side) connecting the host to the device
- A 2 pin jumper cap connector (only necessary for Nano/NX)
- A boot media to program your image on:
    - SD card: cheap but don't expect great performance
    - USB: for the same reasons than above, prefer a real disk than a stick
    - NVMe disk: must be installed to the dedicated PCIe slot

Installation instructions

All the following commands are intended to be run as a normal user (no root).

Preparation

Extract the boot firmware (might require bzip2 installation)

```Python
tar xf Jetson_Linux_R36.3.0_aarch64.tbz2 && cd Linux_for_Tegra/
```

Install the necessary packages

```Python
sudo ./tools/l4t_flash_prerequisites.sh
sudo apt install -y python3 cpp device-tree-compiler mkbootimg
```

Put the dev kit into recovery mode

While the operation is quite similar, the procedure to enable the recovery mode differs with every kit. First connect the USB cable between the host (type A) and the kit (type C). Once the recovery mode is enabled, the host should detect an USB device “0955:7023 NVIDIA Corp. APX”.

Note that once an image is up and running on the device, alternative ways could also be used:

- From the Ubuntu linux shell, run this command

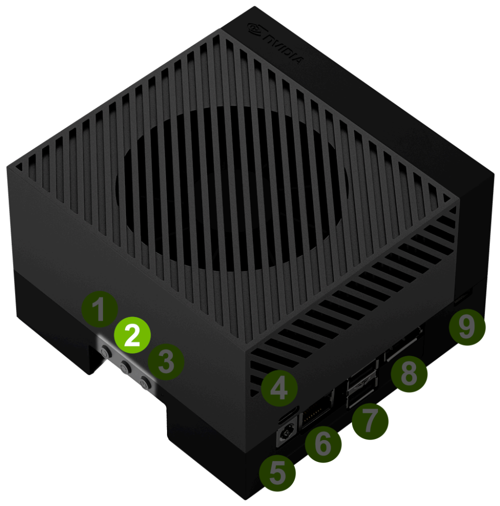
```
Python
sudo reboot --force forced-recovery
```

- From the UEFI menu, enter “Device Manager” menu, then “Boot Configuration” and check the “Boot Into Recovery” option before saving, and exit the UEFI menu with “Continue”

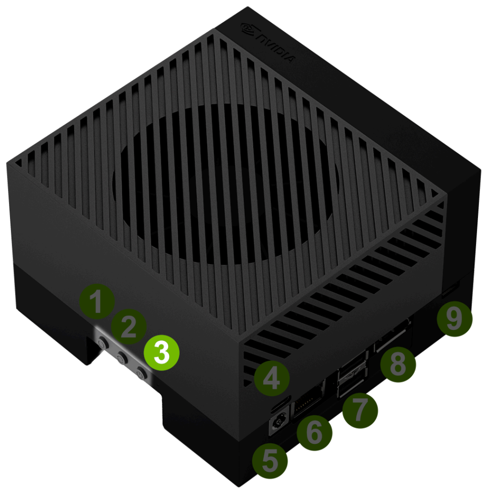
Jetson AGX Orin

The simplest way is to use the buttons on the front panel of the kit (see https://developer.nvidia.com/embedded/learn/jetson-agx-orin-devkit-user-guide/howto.html)

1. Press and hold down the Force Recovery button



2. Press and hold down the Reset button



3. Release both buttons.

Jetson Orin Nano/NX

1. With the kit powered off, connect pins 9 and 10 of the 12 pins Button Header (using a 2 pins jumper cap connector)
2. Power-on the kit
3. Program the boot firmware or Ubuntu image
4. (Optionaly) Power-off the kit and remove the jumper cap connector, otherwise the next hard reboot of the kit will enable the recovery mode once again

Program the boot firmware (QSPI upgrade)

From the Linux_for_Tegra directory, enter the following command to program the latest QSPI boot firmware, it will then reboot the kit automatically upon success.

Jetson AGX Orin

```
Python
sudo ./flash.sh p3737-0000-p3701-0000-qspi internal
```

Jetson Orin Nano/NX

```
Python
sudo ./flash.sh p3768-0000-p3767-0000-a0-qspi internal
```

Program the Ubuntu image on your external boot media

1. Insert your boot media on the host and check its device name (using “lsblk” or “dmesg”)
2. Copy the image over the boot media (assuming here it is detected as /dev/sda)

```
Python
xzcat ubuntu-22.04-preinstalled-server-arm64+tegra-igx.img.xz | sudo dd
of=/dev/sda bs=16M status=progress
sudo sync
```

3. Remove the boot media from the host, insert it on the kit

(Alternatively) Program the Ubuntu image using Nvidia's L4T restore script

This method is the easiest way to program an internal disk, such as eMMC or pre-installed NVMe disk. It also allows you to program the image with limited human interaction (such as programming an external boot media with a side computer, then plugging it to the development kit).

As a prerequisite for this method, you need to put the board into recovery mode. You can then use the backup-restore tool from Nvidia to install a raw disk image on any installed media. The tool loads an initrd flash image via the USB-C cable and boots it, enabling an IPv6 network connection over USB. It relies on nfs-kernel-server service to host the raw image, allowing a NFS mount on the development kit, connected to the host machine (which runs the script). The script will use SSH to connect to the initrd image, mount the NFS volume, and perform the raw image copy using dd. Therefore, the boot media must be connected to the development kit.

From the Linux_for_Tegra directory, enter the following command :

```
Python
sudo ./tools/backup_restore/l4t_backup_restore.sh -r --raw-image \
jammy-preinstalled-server-arm64+tegra-igx.img -e <device> <board-type>

# Full example
sudo ./tools/backup_restore/l4t_backup_restore.sh -r --raw-image \
jammy-preinstalled-server-arm64+tegra-igx.img -e nvme0n1 \
jetson-orin-nano-devkit
```

Board type:

- Jetson AGX Orin: jetson-agx-orin-devkit
- Jetson Orin Nano/NX: jetson-orin-nano-devkit

Device:

- SD card: mmcblk0 (for devices without eMMC, like Nano or NX) or mmcblk1 (for Jetson AGX)
- USB media: sda
- NVMe disk: nvme0n1

Connect a monitor

You can connect to your development kit a USB keyboard/mouse and a monitor using a Display-Port cable. You can follow the boot and kernel execution on display, then get a console prompt once Ubuntu is started (Ubuntu server).

Start Ubuntu 22.04 and install Tegra firmwares and libraries

UEFI menu

At every boot, you will get a chance to enter the UEFI menu by pressing ESC or F11 key.

```Python
Jetson System firmware version 36.2.0-gcid-34956989 date
2023-11-30T18:35:35+00:
00
ESC   to enter Setup.
F11   to enter Boot Manager Menu.
Enter to continue boot.
**  WARNING: Test Key is used.  **
..
```

This menu will eventually allow you to select a different boot option. If you don't press a key, UEFI will automatically launch the default option.

Grub

The UEFI bootloader will automatically launch GRUB, which will then launch Ubuntu.

Ubuntu, first boot

You will be required on first boot to change your password, as the pre-installed image comes with a predefined user ubuntu (password ubuntu).

Install NVIDIA proprietary software

The Ubuntu image brings anything necessary to boot Linux on a Jetson development kit. However, to unlock the features of the Orin's SOC (Wireless, bluetooth, GPU, …) you can install additional NVIDIA proprietary drivers and libraries using a Launchpad PPA :

```Python
sudo add-apt-repository ppa:ubuntu-tegra/updates
# Install Tegra firmwares and necessary Nvidia libraries
sudo apt install -y nvidia-tegra-drivers-36
# Adding user to group render allows running GPU related commands as non root
# video group is necessary to use the camera
sudo usermod -a -G render,video ubuntu
sudo reboot
```

WLAN

Once the package nvidia-tegra-drivers-36 is installed and system rebooted, you should be able to check the WLAN interface (using ip link for instance):

```
Python
ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode
DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: wlP1p1s0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT
group default qlen 1000
    link/ether 90:e8:68:bc:88:a9 brd ff:ff:ff:ff:ff:ff
```

Ubuntu server configuration

Ubuntu server comes with netplan + systemd-networkd preinstalled. The initial netplan configuration in the image should already take care of the Ethernet interface. To setup a wifi connection, you can perform the following steps to add the related configuration:

```
Python
# Install wpa supplicant package
sudo apt install wpasupplicant
# Replace <SSID> with your SSID
SSID='<SSID>'
# Replace <PWD> with your password
PWD='<PWD>'
# Create a netplan configuration for the WLAN
cat <<EOF | sudo tee /etc/netplan/51-wireless.yaml
network:
  wifis:
    wlP1p1s0:
      dhcp4: yes
      dhcp6: yes
      access-points:
        "$SSID":
          password: "$PWD"
  version: 2
EOF
sudo netplan apply
```

Once applied, you might need to reboot to see your network interface up and running, you can check this using the “ip address” command.

Install CUDA and TensorRT

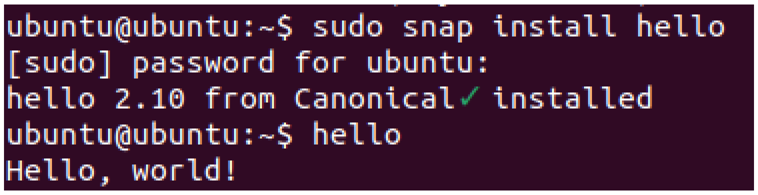
NVIDIA provides a few SDKs like CUDA Toolkit and TensorRT that allow building AI applications on Jetson devices. Those ones are maintained outside of Ubuntu but can be installed with the following commands:

```Python
# CUDA
sudo apt-key adv --fetch-keys \
"https://repo.download.nvidia.com/jetson/jetson-ota-public.asc"
sudo add-apt-repository -y \
"deb https://repo.download.nvidia.com/jetson/t234 r36.3 main"
sudo add-apt-repository -y \
"deb https://repo.download.nvidia.com/jetson/common r36.3 main"
sudo apt install -y cuda

# Tensor RT
sudo apt install -y libnvinfer-bin libnvinfer-samples

# cuda-samples dependencies
sudo apt install -y cmake

echo "export PATH=/usr/local/cuda-12.2/bin\${PATH:+:\${PATH}}" >> ~/.profile
echo "export
LD_LIBRARY_PATH=/usr/local/cuda-12.2/lib64\${LD_LIBRARY_PATH:+:\${LD_LIBRARY_PA
TH}}" >> ~/.profile
```

Test your system

Snap

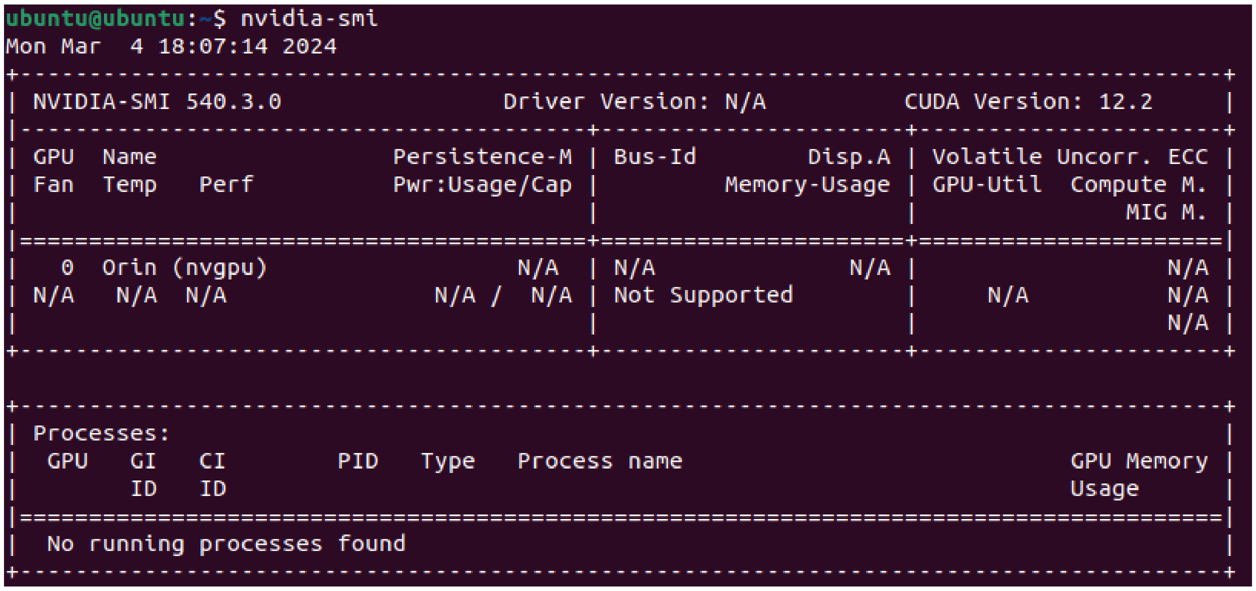
It's Ubuntu! You can install a snap!



```
ubuntu@ubuntu:~$ sudo snap install hello
[sudo] password for ubuntu:
hello 2.10 from Canonical✓ installed
ubuntu@ubuntu:~$ hello
Hello, world!
```

Nvidia-smi

Nvidia-smi can be used to display GPU related information



```
ubuntu@ubuntu:~$ nvidia-smi
Mon Mar  4 18:07:14 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 540.3.0              Driver Version: N/A          CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Orin (nvgpu)                  N/A  | N/A              N/A |                  N/A |
| N/A   N/A  N/A               N/A /  N/A | Not Supported        |     N/A          N/A |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

Run GPU's sample code application

cuda-samples

You can build and run cuda-samples test applications. You can start with "deviceQuery", but you can also build and try many others.

```
Python
git clone https://github.com/NVIDIA/cuda-samples.git -b v12.2
cd cuda-samples
cd Samples/1_Utilities/deviceQuery && make
```

Running this sample code should produce the following output

```
Python
ubuntu@ubuntu:~/cuda-samples/Samples/1_Utilities/deviceQuery$ ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Orin"
  CUDA Driver Version / Runtime Version          12.2 / 12.2
  CUDA Capability Major/Minor version number:    8.7
  Total amount of global memory:                 7619 MBytes (7989571584 bytes)
  (008) Multiprocessors, (128) CUDA Cores/MP:    1024 CUDA Cores
  GPU Max Clock rate:                            624 MHz (0.62 GHz)
  Memory Clock rate:                             624 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 4194304 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072,
65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        167936 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            Yes
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 0 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 12.2, CUDA Runtime
Version = 12.2, NumDevs = 1
Result = PASS
```

TensorRT

Python
```
mkdir ${HOME}/tensorrt-samples
ln -s /usr/src/tensorrt/data ${HOME}/tensorrt-samples/data
cp -a /usr/src/tensorrt/samples ${HOME}/tensorrt-samples/
cd ${HOME}/tensorrt-samples/samples/sampleAlgorithmSelector && make
cd ${HOME}/tensorrt-samples/bin
./sample_algorithm_selector
```

Camera

Prerequisites

The following commands were tested on an IMX219 camera module connected to a Nano and NX devkit.

Python
```
# Allow camera stack to use the right libraries
sudo update-alternatives \
      --install /etc/ld.so.conf.d/aarch64-linux-gnu_EGL.conf \
      aarch64-linux-gnu_egl_conf \
      /usr/lib/aarch64-linux-gnu/tegra-egl/ld.so.conf 1000
sudo update-alternatives \
      --install /etc/ld.so.conf.d/aarch64-linux-gnu_GL.conf \
      aarch64-linux-gnu_gl_conf \
      /usr/lib/aarch64-linux-gnu/nvidia/ld.so.conf 1000
sudo ldconfig
sudo reboot
```

Verify the camera is detected

Please also refer to this link.

Python
```
# Install v4l2-ctl
sudo apt install v4l-utils
v4l2-ctl --list-devices
v4l2-ctl --list-formats-ext
```

If your device is properly detected, the output should be close to this one:

```
Python
ubuntu@ubuntu:~$ v4l2-ctl --list-devices
NVIDIA Tegra Video Input Device (platform:tegra-camrtc-ca):
       /dev/media0

vi-output, imx219 10-0010 (platform:tegra-capture-vi:1):
       /dev/video0
```

You should then be able to detect it via the NVARGUS daemon (in this example, the 'sensor-id' is '0'):

```
Python
ubuntu@ubuntu:~$ nvargus_nvraw --sensorinfo --c 0
nvargus_nvraw version 1.15.0
Number of sensors 1, Number of modes for selected sensor 5
Selected sensor: jakku_front_RBP194 ID 0 Mode 0
Number of exposures 1
Index   Exposure time Range      Sensor Gain Range
0      0.000013 - 0.500000        1.000000 - 10.625000
Warning: Maximum value of Exposure time 0.683709 secs is more than maximum
Frame duration of 0.5 secs.
Changing
       Maximum Exposure time to 0.5 secs.
```

Capture a JPEG image with NVARGUS

Still with the same 'sensor-id'

```
Python
# Unset DISPLAY only if running the commands from SSH or a serial console
unset DISPLAY

nvargus_nvraw --c 0 --format jpg --file ${HOME}/frame-cam0.jpg
```

Gstreamer

Pre-requisites

Make sure to install the necessary gstreamer packages

```
Python
# Install gstreamer plugins
sudo apt-get install -y gstreamer1.0-tools gstreamer1.0-alsa \
   gstreamer1.0-plugins-base gstreamer1.0-plugins-good \
   gstreamer1.0-plugins-bad gstreamer1.0-plugins-ugly \
   gstreamer1.0-libav
sudo apt-get install -y libgstreamer1.0-dev \
   libgstreamer-plugins-base1.0-dev \
   libgstreamer-plugins-good1.0-dev \
   libgstreamer-plugins-bad1.0-dev
```

Camera capture using GStreamer

```
Python
# Unset DISPLAY only if running the commands from SSH or a serial console
unset DISPLAY

# Capture an image
gst-launch-1.0 nvarguscamerasrc num-buffers=1 sensor-id=0 ! \
'video/x-raw(memory:NVMM), width=(int)1920, height=(int)1080,' \
'format=(string)NV12' ! nvjpegenc ! filesink \
 location=${HOME}/gst-frame-cam0.jpg

# Capturing Video from the Camera and Record
gst-launch-1.0 nvarguscamerasrc num-buffers=300 sensor-id=0 ! \
'video/x-raw(memory:NVMM), width=(int)1920, height=(int)1080,' \
'format=(string)NV12, framerate=(fraction)30/1' ! \
nvv4l2h265enc bitrate=8000000 ! h265parse ! qtmux ! \
filesink location=test.mp4
```

Transcode using GStreamer

Using a stream from the Big Buck Bunny project, you can easily test the transcoding pipelines (note that Jetson Orin Nano don't have HW encoders and won't be able to run these pipelines):

```
Python
wget -nv
https://download.blender.org/demo/movies/BBB/bbb_sunflower_1080p_30fps_normal.m
p4.zip
unzip -qu bbb_sunflower_1080p_30fps_normal.mp4.zip
echo "H.264 Decode (NVIDIA Accelerated Decode) to H265 encode"
gst-launch-1.0 filesrc location=bbb_sunflower_1080p_30fps_normal.mp4 ! qtdemux
```

```
! queue ! \
  h264parse ! nvv4l2decoder ! nvv4l2h265enc bitrate=8000000 ! h265parse ! \
  qtmux ! filesink location=h265-reenc.mp4 -e
echo "H.265 Decode (NVIDIA Accelerated Decode) to AV1 Encode (NVIDIA
Accelerated Encode)"
gst-launch-1.0 filesrc location=h265-reenc.mp4 ! qtdemux ! queue ! h265parse !
nvv4l2decoder ! \
  nvv4l2av1enc ! matroskamux name=mux ! filesink location=av1-reenc.mkv -e
echo "AV1 Decode (NVIDIA Accelerated Decode) to H.264 encode"
gst-launch-1.0 filesrc location=av1-reenc.mkv ! matroskademux ! queue !
nvv4l2decoder ! \
  nvv4l2h264enc bitrate=20000000 ! h264parse ! queue ! qtmux name=mux !
filesink \
  location=h264-reenc.mp4 -e
echo "H.264 Decode (NVIDIA Accelerated Decode) to AV1"
gst-launch-1.0 filesrc location=h264-reenc.mp4 ! qtdemux ! \
  h264parse ! nvv4l2decoder ! nvv4l2av1enc ! matroskamux name=mux ! \
  filesink location=av1-reenc.mkv -e
```

cuDDN

Prerequisite

```
Python
sudo apt install libcudnn8-samples
```

Run cuDNN Samples

Build and run the Converted sample.

```
Python
cd /usr/src/cudnn_samples_v8
cd conv_sample
sudo make -j8

sudo chmod +x run_conv_sample.sh
sudo ./run_conv_sample.sh
```

You can also try other sample applications.

Nvidia Container runtime

You can follow these instructions to install and configure the NVIDIA Container Toolkit before running the JetPack container.
Try to run a previously built CUDA sample application:

```
Python
sudo docker run --rm -it -e DISPLAY --net=host --runtime \
nvidia -v /tmp/.X11-unix/:/tmp/.X11-unix  -v \
${HOME}/cuda-samples:/root/cuda-samples \
nvcr.io/nvidia/l4t-jetpack:r36.3.0 \
/root/cuda-samples/Samples/1_Utilities/deviceQuery/deviceQuery
```

Install the desktop environment

Some use cases might require a desktop environment. To turn your Ubuntu server image into a Desktop one, with HW accelerated rendering, run the following commands (the 1st part is only necessary once, you may skip it if camera prerequisites have already been executed):

```
Python
# Allow camera stack to use the right libraries
sudo update-alternatives \
       --install /etc/ld.so.conf.d/aarch64-linux-gnu_EGL.conf \
       aarch64-linux-gnu_egl_conf \
       /usr/lib/aarch64-linux-gnu/tegra-egl/ld.so.conf 1000

sudo update-alternatives \
       --install /etc/ld.so.conf.d/aarch64-linux-gnu_GL.conf \
       aarch64-linux-gnu_gl_conf \
       /usr/lib/aarch64-linux-gnu/nvidia/ld.so.conf 1000
sudo ldconfig

sudo apt install ubuntu-desktop-minimal
sudo sed -i 's/allowed_users.*/allowed_users=anybody/'
"/etc/X11/Xwrapper.config"
echo "needs_root_rights=yes" | sudo tee -a "/etc/X11/Xwrapper.config"
sudo sed 's/#WaylandEnable=false/WaylandEnable=false/' -i /etc/gdm3/custom.conf
sudo adduser gdm video
sudo reboot
```

VPI

Prerequisites

Install VPI and its sample applications

```Python
sudo apt install nvidia-vpi vpi3-samples libopencv cmake libpython3-dev
python3-numpy libopencv-python
```

Test

Execute steps 1 to 6 from the test plan, for each VPI sample application.

Complete Pipeline: Inferencing

Follow the instructions provided in the link above.
The example commands might need an adaptation:

```Python
sudo python3 ./imagenet.py --network=googlenet images/orange_0.jpg output_0.jpg
```