



Hailo TAPPAS User Guide

Release 3.16.0
March 2022



1 Disclaimer and Proprietary Information Notice

1.1 Copyright

© 2022 Hailo Technologies Ltd (“Hailo”). All Rights Reserved. No part of this document may be reproduced or transmitted in any form without the expressed, written permission of Hailo. Nothing contained in this document should be construed as granting any license or right to use proprietary information for that matter, without the written permission of Hailo. This version of the document supersedes all previous versions.

1.2 General Notice

Hailo, to the fullest extent permitted by law, provides this document “as-is” and disclaims all warranties, either express or implied, statutory or otherwise, including but not limited to the implied warranties of merchantability, non-infringement of third parties’ rights, and fitness for particular purpose. Although Hailo used reasonable efforts to ensure the accuracy of the content of this document, it is possible that this document may contain technical inaccuracies or other errors. Hailo assumes no liability for any error in this document, and for damages, whether direct, indirect, incidental, consequential or otherwise, that may result from such error, including, but not limited to loss of data or profits. The content in this document is subject to change without prior notice and Hailo reserves the right to make changes to content of this document without providing a notification to its users.

Hailo TAPPAS Applications



Overview

Hailo's TAPPAS (Template APPLications And Solutions) is an infrastructure designed for easy development and deployment of high-performance edge applications based on the industry-leading Hailo-8™ AI processor. Hailo TAPPAS is pre-packaged with a rich set of applications built on top of state-of-the-art deep neural networks, demonstrating Hailo's best-in-class throughput and power efficiency. For users seeking to quickly deploy their own networks on the Hailo-8, the TAPPAS provides an easy-to-use, [GStreamer](#)-based template for application development.

TAPPAS pre-trained applications and development environment reduce development time and effort, allowing customers to accelerate time to market. The open applications infrastructure offers different operational modes, as well as bring-your-own-model and DIY support functionalities that help improve development flexibility.

Changelog

TAPPAS v3.16.0 (March 2022)

- New Apps:
 - Hailo [Century](#) app - Demonstrates detection on one video file source over 6 different Hailo-8 devices
 - Python app - A classification app using a post-process written in Python
- New Elements:
 - Tracking element "HailoTracker" - Add tracking capabilities
 - Python element "HailoPyFilter" - Enables to write post-processes using Python
- Yocto Hardknott is now supported
- Raspberrypi 4 Ubuntu dedicated apps
- HailoCropper cropping bug fixes
- HailoCropper now accepts cropping method as a shared object (.so)

TAPPAS v3.14.1 (March 2022)

- Fix Yocto Gatesgarth compilation issue
- Added support for hosts without X-Video adapter

TAPPAS v3.15.0 (February 2022)

- New Apps:
 - Detection and depth estimation - Networks switch app
 - Detection (MobilenetSSD) - Single scale tilling app

TAPPAS v3.14.0 (January 2022)

- New Apps:
 - Cascading apps - Face detection and then facial landmarking
- New Yocto layer - Meta-hailo-tappas
- Window enlargement is now supported
- Added the ability to run on multiple devices
- Improved latency on Multi-device RTSP app

TAPPAS v3.13.0 (November 2021)

- Context switch networks in multi-stream apps are now supported
- New Apps:
 - Yolact - Instance segmentation
 - FastDepth - Depth estimation
 - Two networks in parallel on the same device - FastDepth + Mobilenet SSD
 - Retinaface
- Control Element Integration - Displaying device stats inside a GStreamer pipeline (Power, Temperature)
- New Yocto recipes - Compiling our GStreamer plugins is now available as a Yocto recipe
- Added a C++ detection example (native C++ example for writing an app, without GStreamer)

TAPPAS v3.12.0 (October 2021)

- Detection app - MobilenetSSD added
- NVR multi-stream multi device app (detection and pose estimation)
- Facial Landmarks app
- Segmentation app
- Classification app
- Face detection app
- Hailomuxer gstreamer element
- Postprocess implementations for various networks
- GStreamer infrastructure improvements
- Added ARM architecture support and documentation

TAPPAS v3.11.0 (September 2021)

- GStreamer based initial release
- NVR multi-stream detection app
- Detection app
- Hailofilter gstreamer element
- Pose Estimation app

Table of Contents

1. **Getting Started**
 1. **Prerequisites**
 2. **Getting Started**
 3. **Verify Hailo Installation**
 4. **GStreamer**
 5. **Hailo GStreamer Concepts**
 6. **Where to Go From Here?**
 7. **Terminology**
 8. **Useful Links**
2. **GST-launch based X86 applications**
 1. **Sanity pipeline**
 2. **Detection Pipeline**
 3. **Instance Segmentation Pipeline**
 4. **Depth Estimation Pipeline**
 5. **Detection and Depth Estimation Pipelines**
 6. **Multi-Stream Pipeline**
 7. **Pose Estimation Pipeline**
 8. **Segmentation Pipeline**
 9. **Facial Landmarks Pipeline**
 10. **Face Detection Pipeline**
 11. **Face Detection and Facial Landmarking Pipeline**
 12. **Tiling Pipeline**
 13. **Classification Pipeline**
 14. **Multi-stream Multi-device Pipeline**
 15. **Detection and Depth Estimation - networks switch App**
 16. **Python Classification Pipeline**
 17. **Century Pipeline**
3. **GST-launch based ARM applications**
 1. **Detection Pipeline**
4. **GST-launch based Raspberry Pi applications**
 1. **Sanity Pipeline**
 2. **Detection**
 3. **Depth Estimation**
 4. **Multinetworks parallel**
 5. **Pose Estimation**
 6. **Face Detection**
 7. **Classification**
5. **Native C++ applications**
 1. **Detection**
6. **Hailo GStreamer Elements**
7. **HailoNet**
 1. **HailoFilter**
 2. **HailoFilter2**
 3. **HailoPython**

4. **HailoOverlay**
5. **HailoMuxer**
6. **HailoDeviceStats**
7. **HailoAggregator**
8. **HailoCropper**
9. **HailoTileAggregator**
10. **HailoTileCropper**
8. **Installation**
 1. **Docker Install**
 2. **Manual Install**
 3. **Yocto**
 4. **Cross Compile**
9. **Further Reading**
 1. **GStreamer Framework**
 2. **Debugging with GstShark**
 3. **Debugging with Gst-Instruments**
10. **Writing Your Own Postprocess**
 1. **Getting Started**
 2. **Compiling and Running**
 3. **Filter Basics**
 4. **Next Steps (Drawing)**
11. **Writing Your Own Python Postprocess**
 1. **Overview**
 2. **Getting Started**
 3. **Next Steps (Drawing)**

Prerequisites

- Ubuntu 18.04 or 20.04 - You can check it with the following command:
`lsb_release -r`
- Hailo-8 device - Check that your board is recognized by opening a terminal and running: `lspci -d 1e60:.` You should get in response: `bb:dd:ff Co-processor: Hailo Technologies Ltd. Hailo-8 AI Processor (rev 01)`
- At least 6GB's of free disk space

Getting started

X86

We provide three installation methods:

- The simple and recommended installation method is detailed in the [Docker install guide](#)
- You can follow our [Manual install guide](#)
- If you already have a pre built docker image follow our instructions for [Running TAPPAS container from pre-built Docker image](#)

Arm

We provide two installation methods:

- Yocto - integration of Hailo layers in embedded BSP [Read more about Yocto installation](#)
- Cross compilation - cross compile Hailo GStreamer plugins and post processes [Read more about the cross compilation](#)

Verify Hailo Installation

Make sure that hailo is identified correctly by running this command: `hailortcli fw-control identify`, The expected output should look similar to the one below:

```
root@hailo-nvr:/hailo# hailortcli fw-control identify
Identifying board
Control Protocol Version: 2
Firmware Version: X.X.X (develop,app)
Logger Version: 0
Board Name: Hailo-8
Device Architecture: HAILO8_B0
Serial Number: 0000000000000000
Part Number: HM218B1C2FA
Product Name: HAILO-8 AI ACCELERATOR M.2 M KEY MODULE
```

GStreamer

GStreamer is a framework for creating streaming media applications.

GStreamer's development framework makes it possible to write any type of streaming multimedia application. The GStreamer framework is designed to make it easy to write

applications that handle audio or video or both. It isn't restricted to audio and video and can process any kind of data flow. The framework is based on plugins that will provide various codecs and other functionality. The plugins can be linked and arranged in a pipeline. This pipeline defines the flow of the data. The GStreamer core function is to provide a framework for plugins, data flow, and media type handling/negotiation. It also provides an API to write applications using the various plugins.

For additional details check [GStreamer overview](#)

Hailo GStreamer Concepts

Into the GStreamer framework, Hailo brings its functionality so we can infer video frames easily and intuitively without compromising on performance and flexibility.

Hailo Concepts

- **Network encapsulation** - Since in a configured network group, there are only input and output layers a GstHailoNet will be associated to a "Network" by its configured input and output pads
- **Network independent elements** - The GStreamer elements will be network independent, so the same infrastructure elements can be used for different applicative pipelines that use different NN functionality, configuration, activation, and pipelines. Using the new API we can better decouple network configuration and activation stages and thus better support network switching
- **GStreamer Hailo decoupling** - Applicative code will use Hailo API and as such will be GStreamer independent. This will help us build and develop the NN and postprocessing functionality in a controlled environment (with all modern IDE and debugging capabilities).
- **Context control** - Our elements will be contextless and thus leave the context (thread) control to the pipeline builder
- **GStreamer reuse** - our pipeline will use as many off the shelf GStreamer elements

Hailo Elements

- [hailonet](#) - Element for sending and receiving data from Hailo-8 chip
- [hailofilter](#) - Element which enables the user to apply a postprocess or drawing operation to a frame and its tensors
- [hailomuxer](#) - Muxer element used for Multi-Hailo-8 setups
- [hailodevicestats](#) - Hailodevicestats is an element that samples power and temperature
- [hailopython](#) - Element which enables the user to apply a postprocess or drawing operation to a frame and its tensors via python.
- [hailoaggregator](#) - HailoAggregator is an element designed for application with cascading networks. It has 2 sink pads and 1 source
- [hailocropper](#) - HailoCropper is an element designed for application with cascading networks. It has 1 sink and 2 sources
- [hailotileaggregator](#) - HailoTileAggregator is an element designed for application with tiles. It has 2 sink pads and 1 source
- [hailotilecropper](#) - HailoTileCropper is an element designed for application with tiles. It has 1 sink and 2 sources

Where to Go From Here?

That's a great question! Hailo provides a sanity application that helps you verify that the installation phase went well. This is a good starting point. Apps can be found under: [Apps path](#)

Terminology

NVR (Network Video Recorder)

NVR is a specialized hardware and software solution used in IP (Internet Protocol) video surveillance systems. In most cases, the NVR is intended for obtaining video streams from the IP cameras (via the IP network) for the purpose of storage and subsequent playback.

Real Time Streaming Protocol (RTSP)

Is a network control protocol designed for use in entertainment and communications systems to control streaming media servers. This protocol is used for establishing and controlling media sessions between endpoints.

Video Acceleration API (VA-API)

VA-API is an open source API made by Intel that allows applications to use hardware video acceleration capabilities, usually provided by the GPU. It is implemented by the [libva](#) library and utilizes hardware-specific drivers.

Useful Links

Some useful GStreamer debugging tools:

- [Writhing your own postprocess](#) - A detailed guide about how to write your own postprocess
- [Debugging tips](#) - Debugging tips from our experience
- [Cross compile](#) - A cross-compilation guide
- [GstShark](#) - Profiling tool for GStreamer
- [GstInstruments](#) - Basic debugging tool for GStreamer

GST-launch based X86 applications

GST-Launch based applications

1. [Sanity Pipeline](#) - Helps you verify that all the required components are installed correctly
2. [Detection](#) - single-stream object detection pipeline on top of GStreamer using the Hailo-8 device.
3. [Depth Estimation](#) - single-stream depth estimation pipeline on top of GStreamer using the Hailo-8 device.
4. [Multinetworks parallel](#) - single-stream multi-networks pipeline on top of GStreamer using the Hailo-8 device.
5. [Instance segmentation](#) - single-stream instance segmentation on top of GStreamer using the Hailo-8 device.
6. [Multi-stream detection](#) - Multi stream object detection (up to 8 RTSP cameras into one Hailo-8 chip).
7. [Pose Estimation](#) - Human pose estimation using [centerpose](#) network.
8. [Segmentation](#) - Semantic segmentation using [resnet18_fcn8](#) network on top of GStreamer using the Hailo-8 device.
9. [Facial Landmarks](#) - Facial landmarking application.
10. [Face Detection](#) - Face detection application.
11. [Face Detection and Facial Landmarking Pipeline](#) - Face detection and then facial landmarking.
12. [Tiling](#) - Single scale tiling detection application.
13. [Classification](#) - Classification app using [resnet_v1_50](#) network.
14. [Multi-stream Multi-device](#) - Demonstrates Hailo's capabilities using multiple-chips and multiple-streams.
15. [Detection and Depth Estimation - networks switch App](#) - Demonstrates Hailonet network-switch capability.
16. [Python Classification Pipeline](#) - Classification app using [resnet_v1_50](#) with python post-processing.
17. [Century Pipeline](#) - demonstrates detection on one video file source over 6 different Hailo-8 devices.

Sanity pipeline

Overview

Sanity apps purpose is to help you verify that all the required components have been installed successfully.

First of all, you would need to run `sanity_gstreamer.sh` and make sure that the image presented looks like the one that would be presented later.

Sanity GStreamer

This app should launch first.

NOTE: Open the source code in your preferred editor to see how simple this app is.

In order to run the app just `cd` to the `sanity_pipeline` directory and launch the app

```
cd $TAPPAS_WORKSPACE/apps/gstreamer/x86/sanity_pipeline
./sanity_gstreamer.sh
```

The output should look like:



If the output is similar to the image shown above, you are good to go to the next verification phase!

Detection Pipeline

Overview:

`detection.sh` demonstrates detection on one video file source and verifies Hailo's configuration. This is done by running a **single-stream object detection pipeline** on top of GStreamer using the Hailo-8 device.

Options

```
./detection.sh [--input FILL-ME]
```

- `--input` is an optional flag, a path to the video displayed (default is `detection.mp4`).
- `--network` is a flag that sets which network to use. Choose from [YoloV3, YoloV4, YoloV5, Mobilenet_ssd], default is YoloV5. this will set the HEF file to use, the `hailofilter` function to use and the scales of the frame to match the width and height input dimensions of the network.
- `--show-fps` is an optional flag that enables printing FPS on screen.
- `--print-gst-launch` is a flag that prints the ready `gst-launch` command without running it

Supported Networks:

- 'YoloV5' - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/yolov5m.yaml
- 'YoloV4' - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/yolov4_leaky.yaml
- 'YoloV3' - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/yolov3_gluon.yaml
- 'Mobilenet_ssd' - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/ssd_mobilenet_v1.yaml

Run

```
cd $TAPPAS_WORKSPACE/apps/gstreamer/x86/detection
./detection.sh
```

The output should look like:



How it works

This section is optional and provides a drill-down into the implementation of the **detection** app with a focus on explaining the **GStreamer** pipeline. This section uses **yolov5** as an example network so network input width, height, and hef name are set accordingly.

```
gst-launch-1.0 \
  filesrc location=$video_device ! decodebin ! videoconvert ! \
  videoscale ! video/x-raw,width=640,height=640,pixel-aspect-
  ratio=1/1 ! \
  queue ! \
  hailonet hef-path=$hef_path debug=False is-active=true qos=false
  batch-size=8 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
  time=0 ! \
  hailofilter function-name=yolov5 so-path=$POSTPROCESS_SO
  qos=false debug=False ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
  time=0 ! \
  hailofilter so-path=$DRAW_POSTPROCESS_SO qos=false debug=False !
  \
  videoconvert ! \
  fpsdisplaysink video-sink=xvimagesink name=hailo_display
  sync=true text-overlay=false ${additional_parameters}
```

Let's explain this pipeline section by section:

1. `filesrc location=$video_device ! decodebin ! videoconvert !`

Specifies the location of the video used, then decodes and converts to the required format.

2. `videoscale ! video/x-raw,pixel-aspect-ratio=1/1 ! \`

Re-scale the video dimensions to fit the input of the network. In this case it is rescaling the video to 640x640 with the caps negotiation of **hailonet**.

```
3. queue ! \
```

Before sending the frames into the `hailonet` element, set a queue so no frames are lost (Read more about queues [here](#))

```
4. hailonet hef-path=$hef_path debug=False is-active=true
qos=false batch-size=8 ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
```

Performs the inference on the Hailo-8 device.

```
5. hailofilter function-name=yolov5 so-path=$POSTPROCESS_S0
qos=false debug=False ! \
queue name=hailo_draw0 leaky=no max-size-buffers=30 max-size-
bytes=0 max-size-time=0 ! \
hailofilter so-path=$DRAW_POSTPROCESS_S0 qos=false debug=False
! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
```

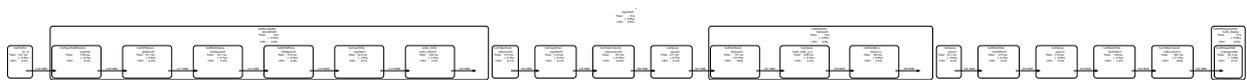
Each `hailofilter` performs a given post-process. In this case the first performs the `Yolov5m` post-process and the second performs box drawing.

```
6. videoconvert ! \
fpsdisplaysink video-sink=xvimagesink name=hailo_display
sync=true text-overlay=false ${additonal_parameters}
```

Apply the final convert to let GStreamer utilize the format required by the `fpsdisplaysink` element

NOTE: Additional details about the pipeline provided in further examples

Structure of the pipeline



Depth Estimation Pipelines

Depth Estimation

`depth_estimation.sh` demonstrates depth estimation on one video file source. This is done by running a `single-stream object depth estimation pipeline` on top of GStreamer using the Hailo-8 device.

Options

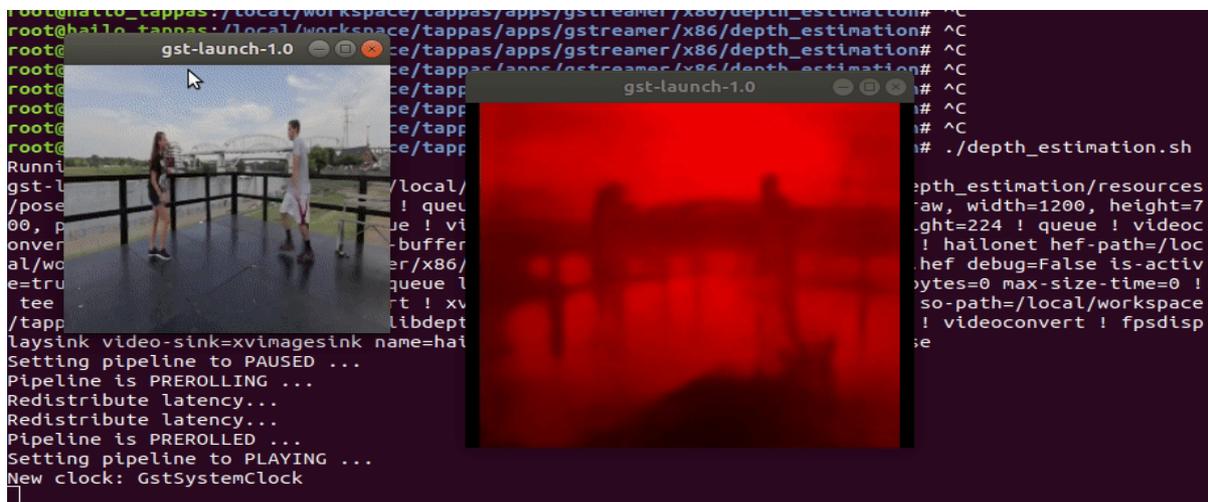
```
./depth_estimation.sh [--video-src FILL-ME]
```

- `-i --input` is an optional flag, a path to the video displayed.
- `--print-gst-launch` is a flag that prints the ready `gst-launch` command without running it
- `--show-fps` is an optional flag that enables printing FPS on screen

Run

```
cd /local/workspace/tappas/apps/gstreamer/x86/depth_estimation
./depth_estimation.sh
```

The output should look like:



Model

- `fast_depth` in resolution of 224X224X3.

How it works

This section is optional and provides a drill-down into the implementation of the `depth estimation` app with a focus on explaining the `GStreamer` pipeline. This section uses `fast_depth` as an example network so network input width, height, hef name, are set accordingly.

```
gst-launch-1.0 \
  $source_element ! queue ! \
  videobox autocrop=true ! video/x-raw, width=1200, height=700,
pixel-aspect-ratio=1/1 ! \
  queue ! \
  videoscale ! video/x-raw, width=224, height=224 ! queue !
videoconvert ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailonet hef-path=$hef_path debug=False is-active=true qos=false
batch-size=1 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  tee name=t ! queue ! videoconvert ! xvimagesink sync=false t. ! \
  hailofilter so-path=$draw_so qos=false debug=False ! \
  videoconvert ! \
  fpsdisplaysink video-sink=xvimagesink name=hailo_display
sync=false text-overlay=false ${additional_parameters}"
```

Let's explain this pipeline section by section:

1. `filesrc location=$video_device ! decodebin ! videoconvert !`

Specifies the location of the video used, then decodes and converts to the required format.

2. `videobox autocrop=true ! video/x-raw, width=1200, height=700,
pixel-aspect-ratio=1/1 ! \
videoscale ! video/x-raw, width=224, height=224`

Re-scales the video dimensions to fit the input of the network. In this case it is cropping the video and rescaling the video to 224x224 with the caps negotiation of `hailonet`.

3. `queue leaky=no max-size-buffers=13 max-size-bytes=0 max-size-
time=0 ! \`

Before sending the frames into `hailonet` element, set a queue so no frames are lost (Read more about queues [here](#))

4. `hailonet hef-path=$hef_path debug=False is-active=true
qos=false batch-size=1 ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \`

Performs the inference on the Hailo-8 device.

```
5. hailofilter so-path=$DRAW_POSTPROCESS_S0 qos=false debug=False
! \
```

Performs a given draw-process, in this case, performs `fast_depth` depth estimation drawing per pixel.

```
6. videoconvert ! \
fpsdisplaysink video-sink=ximagesink name=hailo_display
sync=true text-overlay=false ${additional_parameters}
```

Apply the final convert to let GStreamer utilize the format required by the `fpsdisplaysink` element

NOTE: Additional details about the pipeline provided in further examples

Instance Segmentation Pipeline

Overview:

`instance_segmentation.sh` demonstrates instance segmentation on one video file source and verifies Hailo's configuration. This is done by running a `single-stream instance segmentation pipeline` on top of GStreamer using the Hailo-8 device.

Options

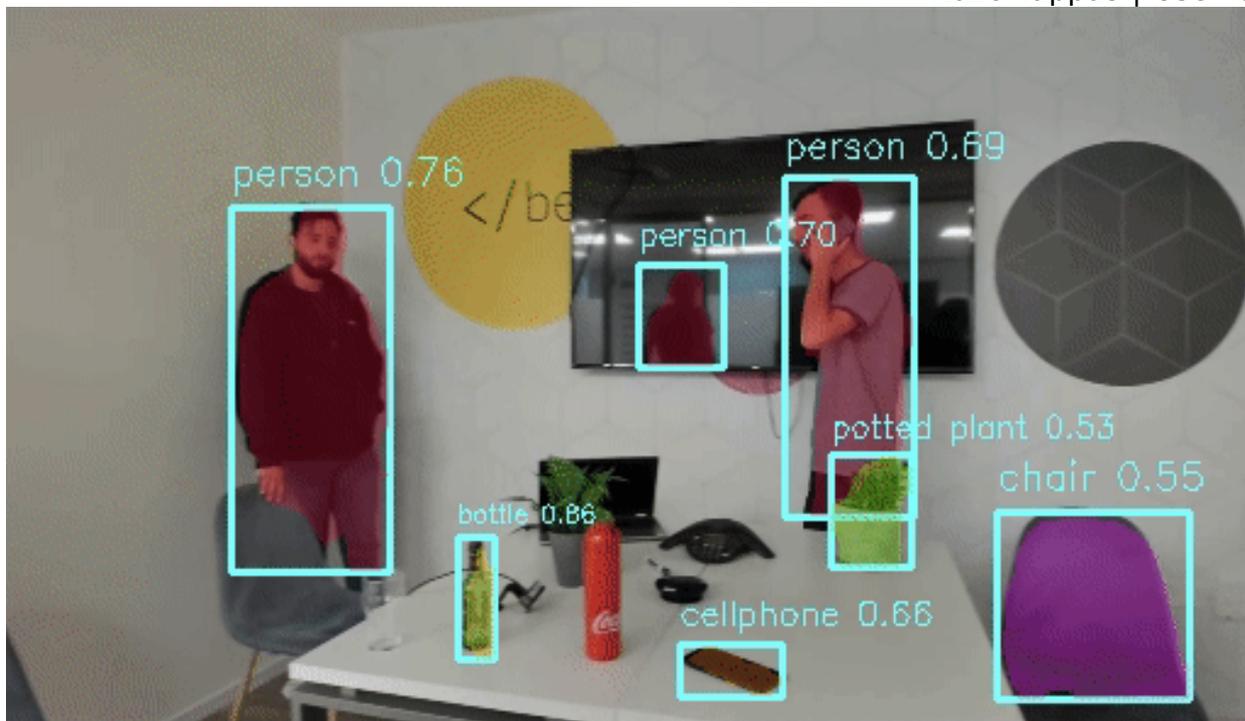
```
./instance_segmentation.sh [--input FILL-ME]
```

- `--input` is an optional flag, a path to the video displayed (default is `detection.mp4`).
- `--show-fps` is an optional flag that enables printing FPS on screen.
- `--print-gst-launch` is a flag that prints the ready `gst-launch` command without running it"

Run

```
cd $TAPPAS_WORKSPACE/apps/gstreamer/x86/instance_segmentation
./instance_segmentation.sh
```

The output should look like:



How it works

This section is optional and provides a drill-down into the implementation of the `instance_segmentation` app with a focus on explaining the `GStreamer` pipeline. This section uses `yolact_regnetx_800mf_fpn_20classes` as an example network so network input width, height, and hef name are set accordingly.

```
gst-launch-1.0 \
  filesrc location=$video_device ! decodebin ! videoconvert ! \
  videoscale ! video/x-raw,width=512,height=512,pixel-aspect-
  ratio=1/1 ! \
  queue queue leaky=no max-size-buffers=30 max-size-bytes=0 max-
  size-time=0 ! \
  hailonet hef-path=$hef_path debug=False is-active=true qos=false
  batch-size=8 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
  time=0 ! \
  hailofilter so-path=$POSTPROCESS_S0 qos=false debug=False ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
  time=0 ! \
  hailofilter so-path=$DRAW_POSTPROCESS_S0 qos=false debug=False !
  \
  videoconvert ! \
  fpsdisplaysink video-sink=ximagesink name=hailo_display sync=true
  text-overlay=false ${additional_parameters}
```

Let's explain this pipeline section by section:

1. `filesrc location=$video_device ! decodebin ! videoconvert !`

Specifies the location of the video used, then decodes and converts to the required format.

```
2. videoscale ! video/x-raw,pixel-aspect-ratio=1/1 ! \
```

Re-scale the video dimensions to fit the input of the network. In this case it is rescaling the video to 512x512 with the caps negotiation of `hailonet`.

```
3. queue queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \
```

Before sending the frames into the `hailonet` element, set a queue so no frames are lost (Read more about queues [here](#))

```
4. hailonet hef-path=$hef_path debug=False is-active=true
   qos=false batch-size=8 ! \
   queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \
```

Performs the inference on the Hailo-8 device.

```
5. hailofilter so-path=$POSTPROCESS_S0 qos=false debug=False ! \
   queue name=hailo_draw0 leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \
   hailofilter so-path=$DRAW_POSTPROCESS_S0 qos=false debug=False ! \
   queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \
```

Each `hailofilter` performs a given post-process. In this case the first performs the `yolact` post-process and the second performs box and segmentation mask drawing.

```
6. videoconvert ! \
   fpsdisplaysink video-sink=ximagesink name=hailo_display
   sync=true text-overlay=false ${additional_parameters}
```

Apply the final convert to let GStreamer utilize the format required by the `fpsdisplaysink` element

NOTE: Additional details about the pipeline provided in further examples

Detection and Depth Estimation Pipelines

`detection_and_depth_estimation.sh` demonstrates depth estimation and detection on one video file source. This is done by running two streams on top of GStreamer using one Hailo-8 device with using two `hailonet` elements.

Options

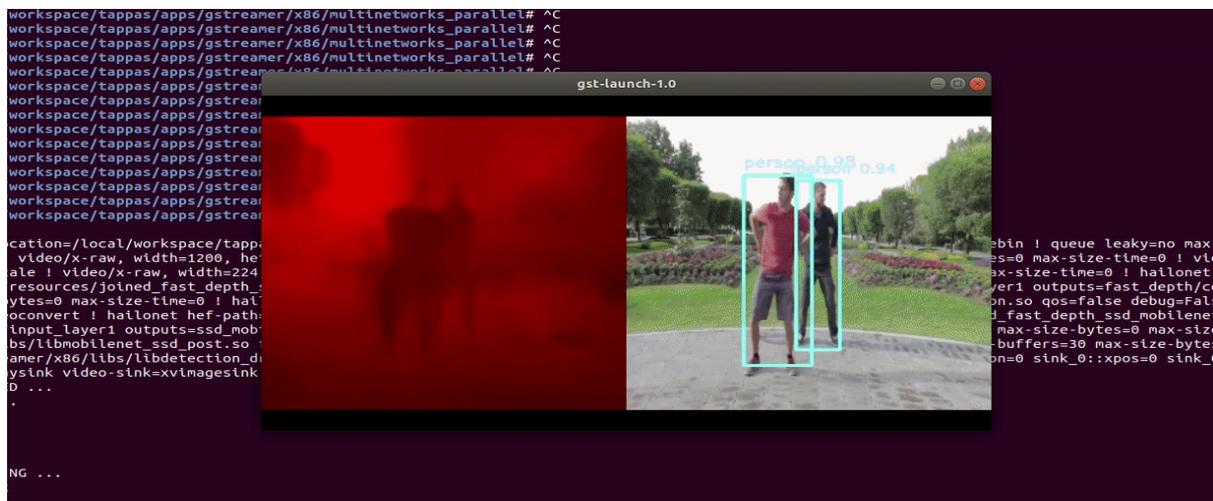
```
./detection_and_depth_estimation.sh [--video-src FILL-ME]
```

- **-i --input** is an optional flag, a path to the video displayed.
- **--print-gst-launch** is a flag that prints the ready gst-launch command without running it
- **--show-fps** is an optional flag that enables printing FPS on screen

Run

```
cd /local/workspace/tappas/apps/gstreamer/x86/multinetworks_parallel
./detection_and_depth_estimation.sh
```

The output should look like:



Model

- **fast_depth** in resolution of 224X224X3.
- **mobilenet_ssd** in resolution of 300X300X3.

How it works

This section is optional and provides a drill-down into the implementation of the app with a focus on explaining the **GStreamer** pipeline. This section uses **fast_depth** as an example network so network input width, height, hef name, are set accordingly.

```
gst-launch-1.0 \
  $source_element ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \
  videobox autocrop=true ! video/x-raw, width=1200, height=700, pixel-aspect-ratio=1/1 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \
  videoscale ! video/x-raw, width=300, height=300 ! queue ! \
  tee name=t ! \
  videoscale ! video/x-raw, width=224, height=224! videoconvert ! \
```

```

queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailonet hef-path=$hef_path debug=False is-active=true
inputs=$depth_estimation_inputs outputs=$depth_estimation_outputs
qos=false batch-size=1 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailofilter so-path=$depth_estimation_draw_so qos=false
debug=False ! \
  videoscale ! video/x-raw, width=300, height=300 ! \
  comp.sink_0 \
  t. ! \
  videoconvert ! \
  hailonet hef-path=$hef_path debug=False is-active=true
inputs=$detection_inputs outputs=$detection_outputs qos=false batch-
size=1 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailofilter so-path=$detection_post_so function-
name=mobilenet_ssd_merged qos=false debug=False ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailofilter so-path=$detection_draw_so qos=false debug=False ! \
  comp.sink_1 \
  compositor name=comp start-time-selection=0 $compositor_locations
! queue ! videoconvert ! \
  fpsdisplaysink video-sink=xvimagesink name=hailo_display
sync=false text-overlay=false ${additional_parameters}

```

Let's explain this pipeline section by section:

1. `filesrc location=$video_device ! decodebin ! videoconvert !`

Specifies the location of the video used, then decodes and converts to the required format.

2. `videobox autocrop=true ! video/x-raw, width=1200, height=700, pixel-aspect-ratio=1/1 ! \
 videoscale ! video/x-raw, width=300, height=300`

Re-scales the video dimensions to fit the input of the network. In this case it is cropping the video and rescaling the video to 224x224 with the caps negotiation of `hailonet`.

3. `tee name=t !`

Split into two threads - one for `mobilenet_ssd` and the other for `fast_depth`.

4. `queue leaky=no max-size-buffers=13 max-size-bytes=0 max-size-
time=0 ! \`

Before sending the frames into `hailonet` element, set a queue so no frames are lost (Read more about queues [here](#))

5.

```
hailonet hef-path=$hef_path debug=False is-active=true
inputs=$depth_estimation_inputs
outputs=$depth_estimation_outputs qos=false batch-size=1
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
```

Performs the inference on the Hailo-8 device.

NOTE: We pre define the input and the output layers of each network, giving the net name argument.

6.

```
hailofilter so-path=$DRAW_POSTPROCESS_S0 qos=false debug=False
! \
```

Performs a given draw-process, in this case, performs `fast_depth` depth estimation drawing per pixel.

7.

```
compositor ! \
fpsdisplaysink video-sink=ximagesink name=hailo_display
sync=true text-overlay=false ${additional_parameters}
```

Apply the final convert to let GStreamer utilize the format required by the `fpsdisplaysink` element

NOTE: Additional details about the pipeline provided in further examples

Multi-Stream RTSP object detection Pipeline

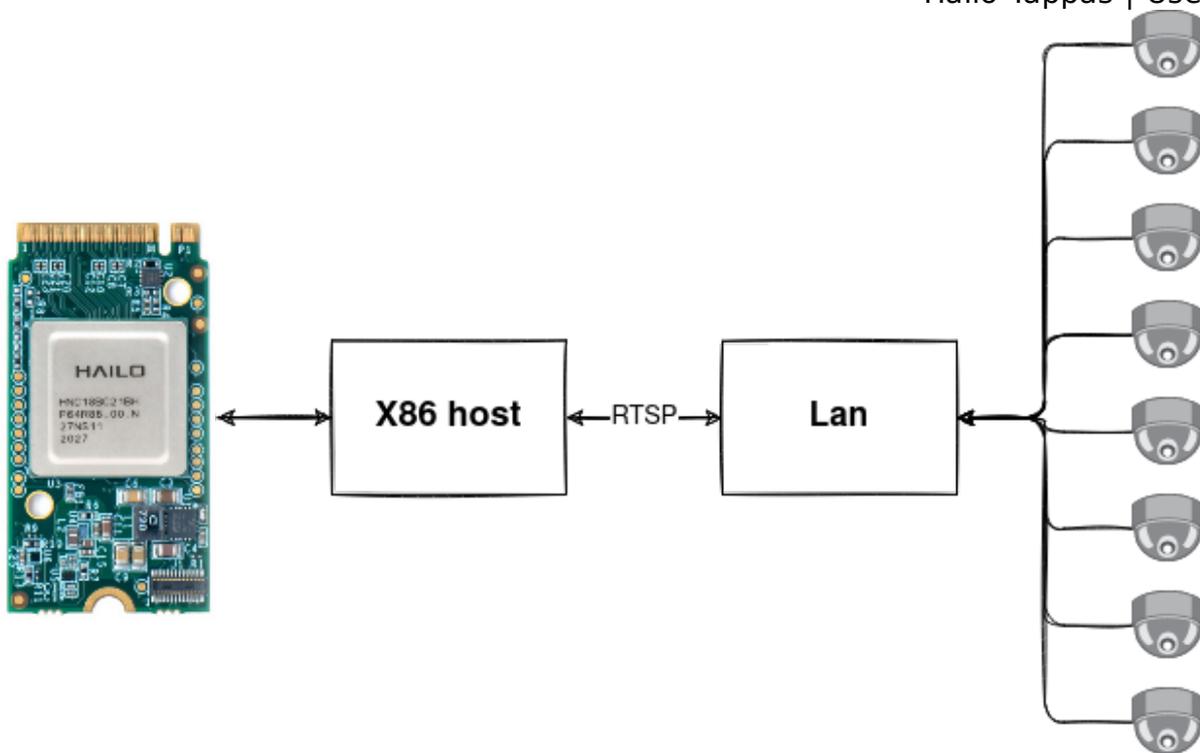
Overview

This GStreamer pipeline demonstrates object detection on 8 camera streams over RTSP protocol.

All the streams are processed in parallel through the decode and scale phases, and enter the Hailo device frame by frame.

Afterwards postprocess and drawing phases add the classified object and bounding boxes to each frame.

The last step is to match each frame back to its respective stream and output all of them to the display.



Real Time Streaming Protocol (RTSP) is a network control protocol designed for use in entertainment and communications systems to control streaming media servers. The protocol is used for establishing and controlling media sessions between endpoint.

Prerequisites

- TensorPC
- Ubuntu 18.04
- [RTSP Cameras](#), We recommend using: [AXIS M10 Network Cameras](#)
- Hailo-8 device connected via PCIe

Preparations

1. Before running, configuration of the RTSP camera sources is required. open the `multistream_pipeline.sh` in edit mode with your preferred editor. Configure the eight sources to match your own cameras.

```
readonly SRC_0="rtsp://<ip address>/?h264x=4 user-id=<username> user-pw=<password>"
readonly SRC_1="rtsp://<ip address>/?h264x=4 user-id=<username> user-pw=<password>"
etc..
```

Run the pipeline

```
./multistream_pipeline.sh
```

1. `--show-fps` prints the fps to the output.

```
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 513, dropped: 0, current: 29.85, average: 26.68
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 528, dropped: 0, current: 29.11, average: 26.74
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 543, dropped: 0, current: 28.66, average: 26.79
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 558, dropped: 0, current: 28.28, average: 26.83
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 570, dropped: 0, current: 23.48, average: 26.75
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 584, dropped: 0, current: 26.98, average: 26.76
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 598, dropped: 0, current: 27.85, average: 26.78
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 613, dropped: 0, current: 28.46, average: 26.82
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 626, dropped: 0, current: 25.70, average: 26.80
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 641, dropped: 0, current: 28.17, average: 26.83
```

2. `--disable-vaapi` disables the vaapi accelerator usage. This replaces decoder elements and videosink elements from vaapi to decodebin, videoscale and autovideosink.
3. `--num-of-sources` sets the number of rtsp sources to use by given input. the default and recommended value in this pipeline is 8 sources"
4. `--debug` uses `gst-top` to print time and memory consuming elements, saves the results as text and graph.

```
Execution ended after 0:00:23.599335345
Setting pipeline to PAUSED ...
Setting pipeline to READY ...
Setting pipeline to NULL ...
Freeing pipeline ...

** (gst-top-1.0:545889): WARNING **: 15:54:52.438: gst-launch-1.0 (545892) exited with code 256
Most time and memory consuming elements:
ELEMENT      %CPU  %TIME  TIME
hailosend0   23.3  21.4   5.51 s
hailo_pre_infer_q_0  9.4   8.6    2.22 s
comp        8.8   8.1    2.08 s
Full pipeline report saved to pipeline_report.txt and pipeline_graph.svg
```

Open the `pipeline_report.txt` to view the full report showing all elements, your report should be similar to this:

```
ELEMENT      %CPU  %TIME  TIME
hailosend0   23.4  22.7   18.4 s
hailorecv0   8.7   8.4    6.84 s
hailo_pre_infer_q_0  8.2   8.0    6.48 s
comp        8.2   8.0    6.47 s
hailofilter0  5.5   5.3    4.32 s
videoconvert5  2.6   2.5    2.02 s
videoconvert4  2.6   2.5    2.01 s
videoconvert1  2.6   2.5    2.01 s
videoconvert3  2.5   2.4    1.96 s
videoconvert0  2.4   2.4    1.92 s
videoconvert2  2.4   2.3    1.89 s
vaapisink0   2.2   2.1    1.73 s
hailofilter1  2.1   2.0    1.65 s
videoconvert6  1.7   1.7    1.37 s
videoconvert7  1.7   1.7    1.35 s
fun         1.5   1.4    1.17 s
hailo_infer_q_0  1.0   0.9    752 ms
vaapidcode4  0.6   0.6    503 ms
vaapidcode0  0.6   0.6    496 ms
vaapipostproc4  0.6   0.6    493 ms
vaapidcode1  0.6   0.6    489 ms
vaapipostproc1  0.6   0.6    489 ms
vaapipostproc5  0.6   0.6    488 ms
vaapidcode5  0.6   0.6    485 ms
vaapipostproc2  0.6   0.6    485 ms
vaapipostproc0  0.6   0.6    474 ms
vaapipostproc3  0.6   0.6    470 ms
vaapidcode3  0.6   0.5    442 ms
hailo_postprocess0  0.6   0.5    441 ms
vaapidcode2  0.6   0.5    439 ms
vaapipostproc6  0.4   0.4    344 ms
```

NOTE: When the debug flag is used and the app is running inside of a docker, exit the app by typing `Ctrl+C` in order to save the results. (Due to docker X11 display communication issues)

Model

YOLOv5 is a modern object detection architecture that is based on the **YOLOv3** meta-architecture with **CSPNet** backbone. The **YOLOv5** was released on 05/2020 with a very efficient design and SoTA accuracy results on the **COCO benchmark**.

in this pipeline, we're using a specific variant of the YOLOv5 architecture - `yo1ov5m` that stands for medium sized networks.

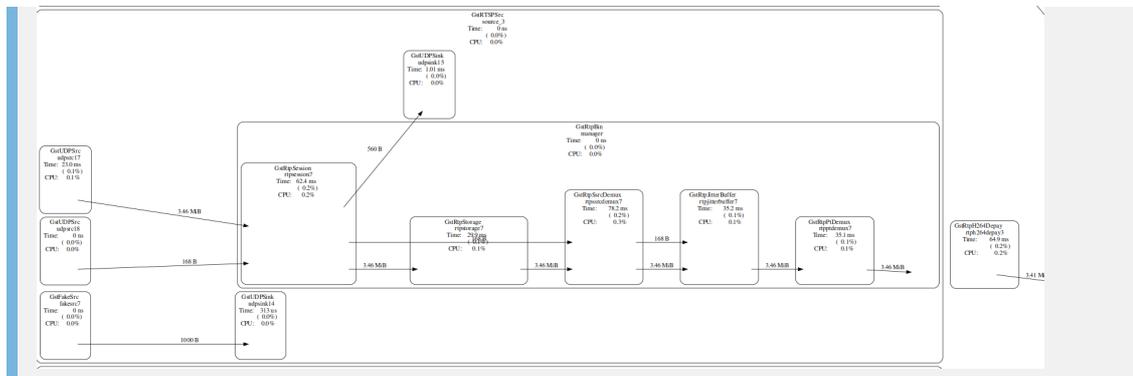
- Pre trained and compiled `yo1ov5m` model stored as `.hef` file.
- Resolution: 640x640x3
- Full precision accuracy: 41.7mAP
- Dataset: COCO val2017 <https://cocodataset.org/#home>

Enter the git project to read further: <https://github.com/ultralytics/yolov5> Link to the network yaml in Hailo Model Zoo - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/yolov5m.yaml

Overview of the pipeline

The following elements are the structure of the pipeline:

- `rtspsrc` makes a connection to an rtsp server and read the data. used as a src to get the video stream from rtsp-cameras.
- `rtpH264depay` extracts h264 video from rtp packets.

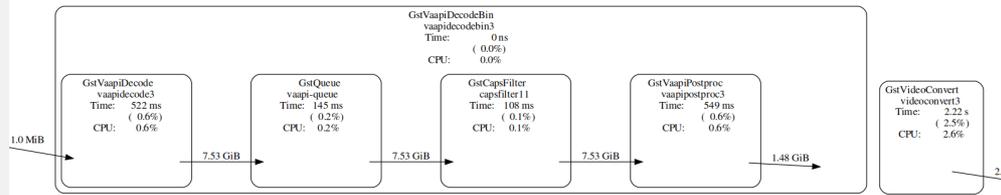


- `vaapi<CODEC>dec` video decoding and scaling - this element uses vaapi hardware acceleration to improve the pipeline performance. Video Acceleration API (VA-API) is an open source API made by Intel that allows applications to use hardware video acceleration capabilities, usually provided by the GPU. It is implemented by libva library and combined with a hardware-specific driver.

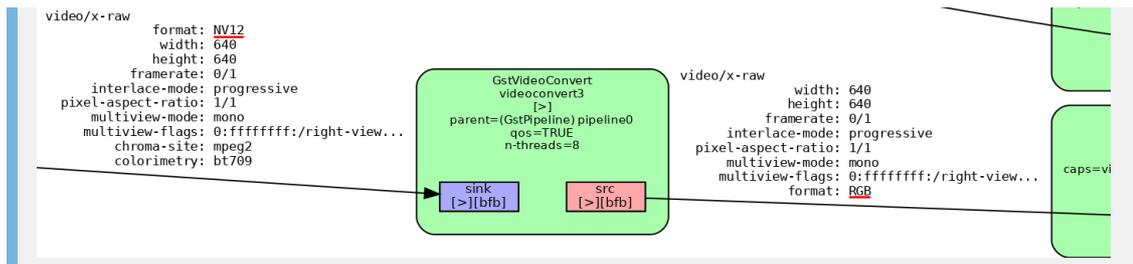
In this pipeline, the bin is responsible for decoding h264 format and scaling the frame to 640X640. It contains the following elements:

- `vaapi<CODEC>dec` is used to decode JPEG, MPEG-2, MPEG-4:2, H.264 AVC, H.264 MVC, VP8, VP9, VC-1, WMV3, HEVC videos to VA surfaces (vaapi's memory format), depending on the actual value of 'CODEC' and the underlying hardware capabilities. This plugin is also able to implicitly download the decoded surface to raw YUV buffers.
- `vaapi<CODEC>dec` is used to filter VA surfaces, for e.g. scaling, deinterlacing, noise reduction or sharpening. This plugin is also used to upload raw YUV pixels into VA surfaces.
- `vaapi<CODEC>dec` responsible for rendering VA surfaces to an X11 or Wayland display (used in this pipeline by the `fpsdisplaysink`).

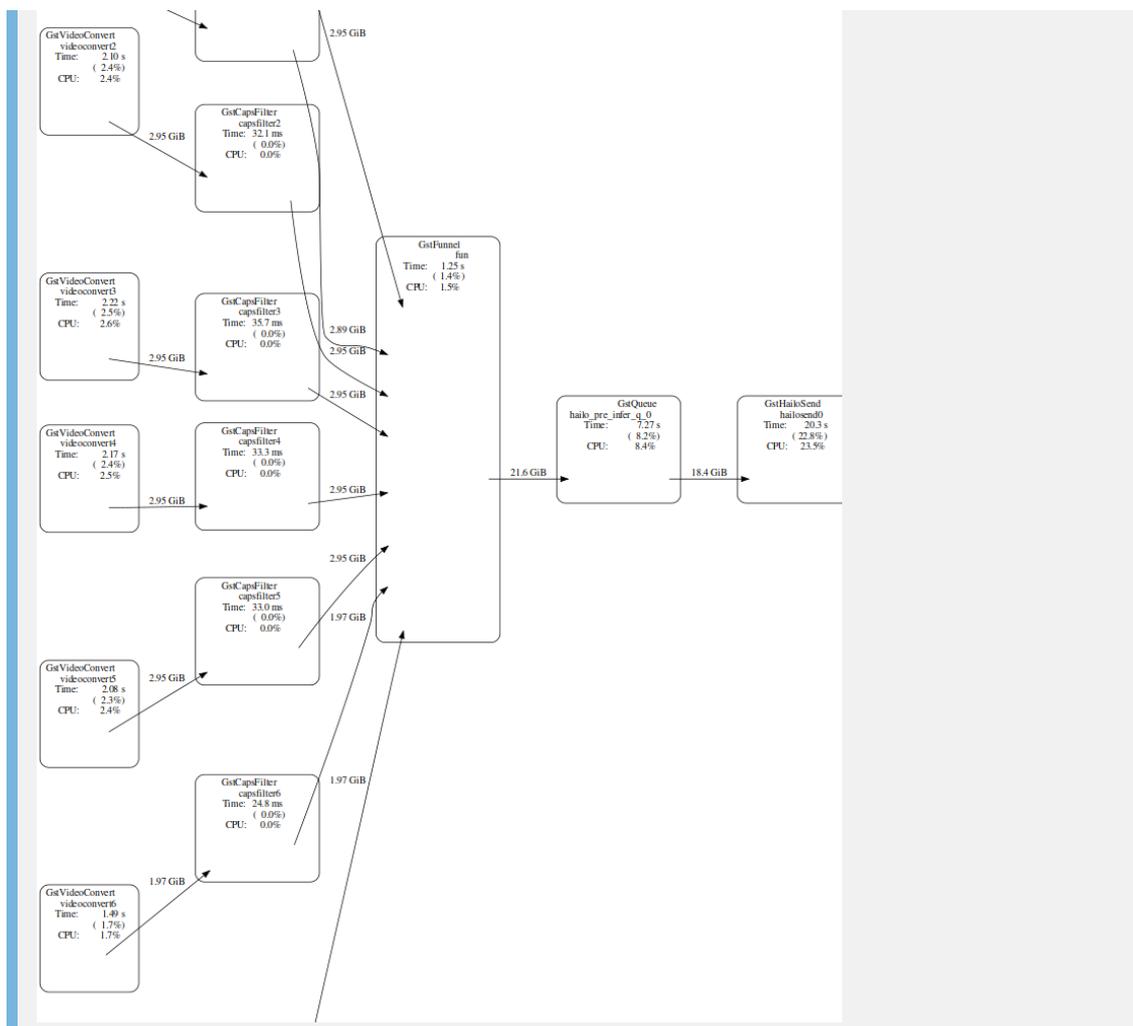
NOTE: In case your device does not support vaapi acceleration - you should replace the vaapi elements in the pipeline with `--disable-vaapi` argument. this includes swapping decoder elements and videosink elements with regular decodebin, videoscale (instead of vappi's postproc) and autovideosink.



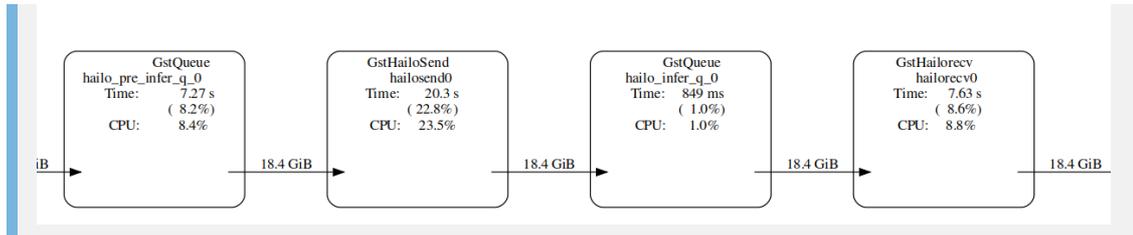
- **videoconvert** converting the frame into RGB format



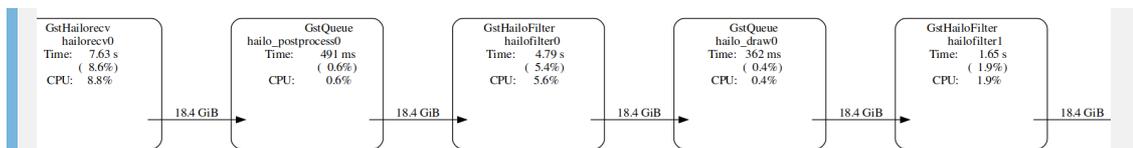
- **funnel** takes multiple input sinks and outputs one source. an N-to-1 funnel that attaches a streamid to each stream, can later be used to demux back into separate streams. this lets you queue frames from multiple streams to send to the hailo device one at a time.



- **hailonet** Performs the inference on the Hailo-8 device - configures the chip with the hef and starts hailo's inference process - sets streaming mode and sends the buffers into the chip. requires the following properties: **hef-path** - points to the compiled yolov5m hef, **qos** that must be set to false - to disable frame drops, and **batch-size**. [read more about hailonet](#)



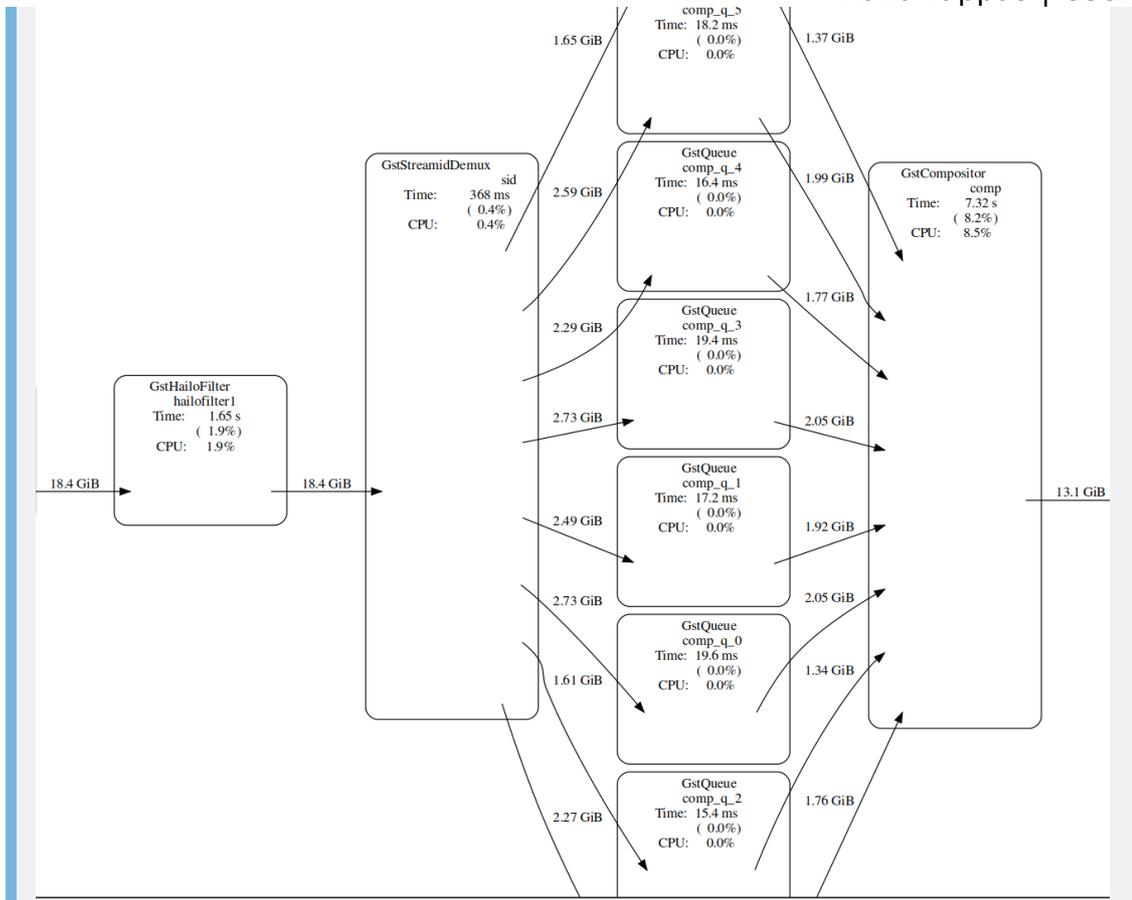
- **hailofilter** performs given postprocess, chosen with the **so-path** property. in this pipeline, two are incorporated to performe yolov5m postprocess and box drawing.



****NOTE****: If multiple hailofilters are present and dependent on each other, then `qos` must be disabled for each.

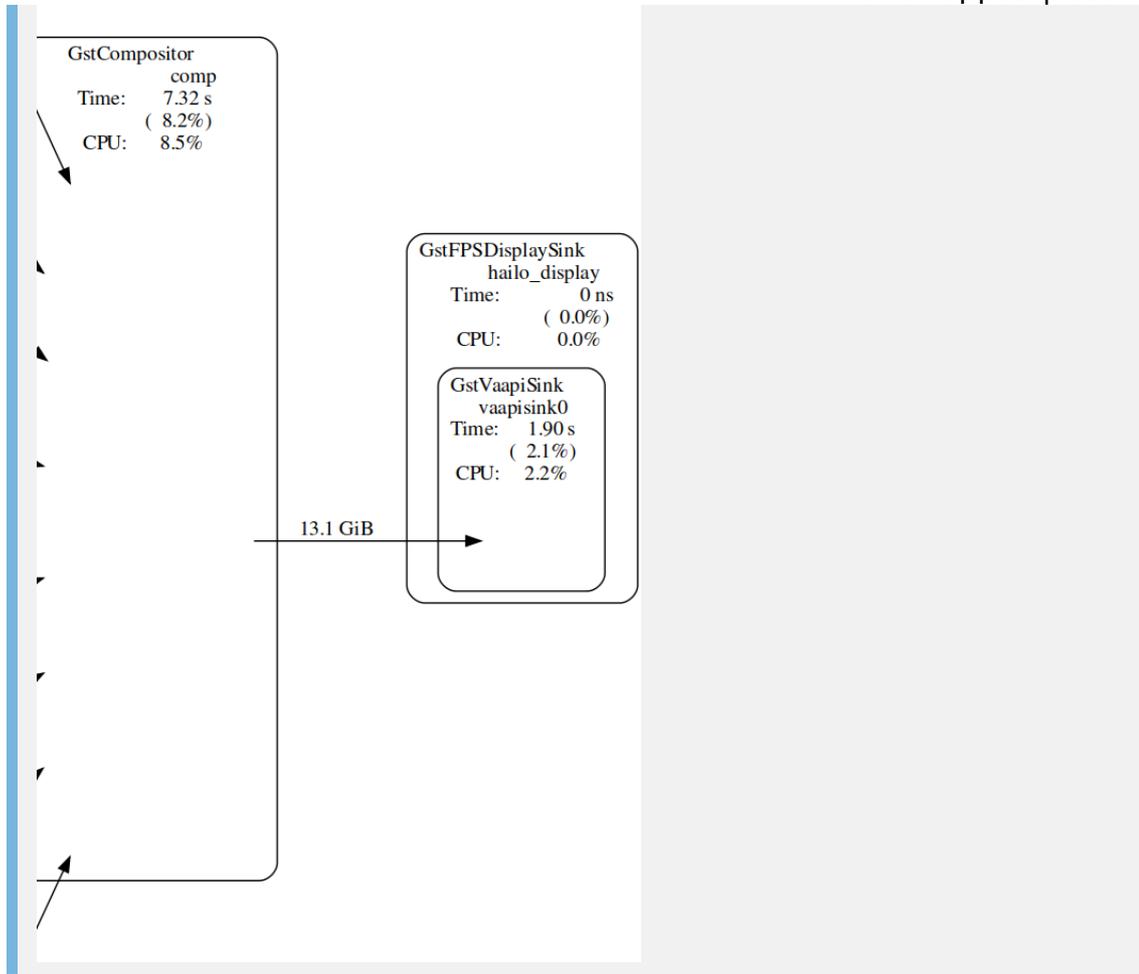
If there is only one hailofilter, then qos may be enabled (although it is still recommended to disable).

- **streamiddemux** a reverse to the funnel. It is a 1-to-N demuxer that splits a serialized stream based on stream id to multiple outputs.
- **compositor** composites pictures from multiple sources. handy for multi-stream/tiling like applications, as it lets you input many streams and draw them all together as a grid.

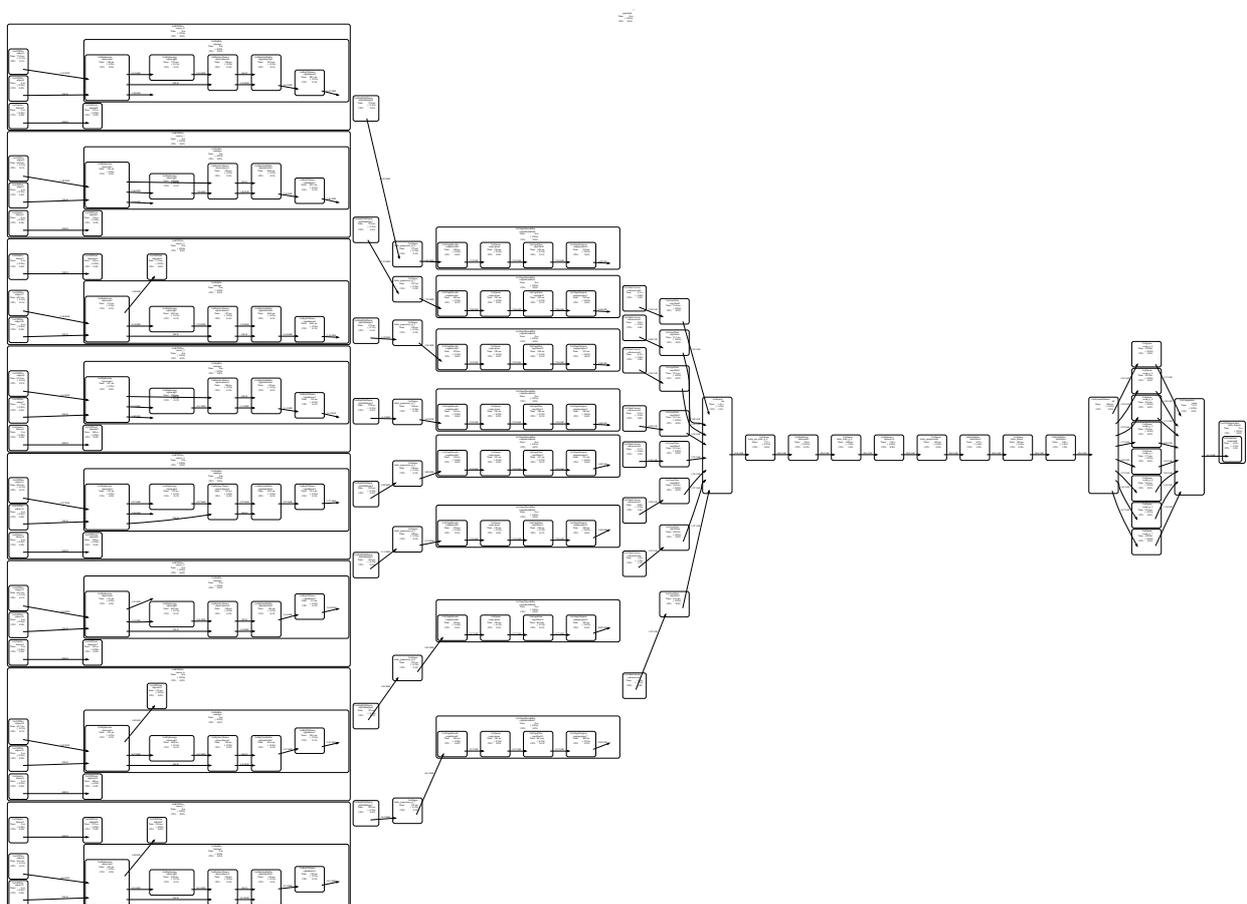


- `fpsdisplaysink` outputs video into the screen, and displays the current and average framerate.

NOTE: `sync=false` property in `fpsdisplaysink` element disables real-time synchronization with the pipeline - it is mandatory on this case to reach the best performance.



Entire pipeline



Pose Estimation Pipeline

Overview:

`hailo_pose_estimation.sh` demonstrates human pose estimation on one video file source and verifies Hailo's configuration. This is done by running a `single-stream pose estimation pipeline` on top of GStreamer using the Hailo-8 device.

Options

```
./hailo_pose_estimation.sh [--input FILL-ME]
```

- `--input` is an optional flag, a path to the video displayed (default is `detection.mp4`).
- `--show-fps` is an optional flag that enables printing FPS on screen.
- `--network` Set network to use. choose from [`centerpose`, `centerpose_faster`, `centerpose_416`], default is `centerpose`
- `--print-gst-launch` is a flag that prints the ready `gst-launch` command without running it"

Run

```
cd $TAPPAS_WORKSPACE/apps/gstreamer/x86/pose_estimation
./hailo_pose_estimation.sh
```

The output should look like:



How it works

This section is optional and provides a drill-down into the implementation of the `pose_estimation` app with a focus on explaining the `GStreamer` pipeline. This section

uses `centerpose_regnetx_1.6gf_fpn` as an example network so network input width, height, and hef name are set accordingly.

```
gst-launch-1.0 \
  filesrc location=$video_device ! decodebin ! videoconvert ! \
  videoscale ! video/x-raw,width=640,height=640,pixel-aspect-
  ratio=1/1 ! \
  queue queue leaky=no max-size-buffers=30 max-size-bytes=0 max-
  size-time=0 ! \
  hailonet hef-path=$hef_path debug=False is-active=true qos=false
  batch-size=8 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
  time=0 ! \
  hailofilter function-name=yolov5 so-path=$POSTPROCESS_SO
  qos=false debug=False ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
  time=0 ! \
  hailofilter so-path=$DRAW_POSTPROCESS_SO qos=false debug=False !
  \
  videoconvert ! \
  fpsdisplaysink video-sink=xvimagesink name=hailo_display
  sync=true text-overlay=false ${additional_parameters}
```

Let's explain this pipeline section by section:

1. `filesrc location=$video_device ! decodebin ! videoconvert !`

Specifies the location of the video used, then decodes and converts to the required format.

2. `videoscale ! video/x-raw,pixel-aspect-ratio=1/1 ! \`

Re-scale the video dimensions to fit the input of the network. In this case it is rescaling the video to 640x640 with the caps negotiation of `hailonet`.

3. `queue queue leaky=no max-size-buffers=30 max-size-bytes=0 max-
 size-time=0 ! \`

Before sending the frames into the `hailonet` element, set a queue so no frames are lost (Read more about queues [here](#))

4. `hailonet hef-path=$hef_path debug=False is-active=true
 qos=false batch-size=8 ! \
 queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
 time=0 ! \`

Performs the inference on the Hailo-8 device.

5.

```
hailofilter function-name=yolov5 so-path=$POSTPROCESS_S0
qos=false debug=False ! \
  queue name=hailo_draw0 leaky=no max-size-buffers=30 max-size-
bytes=0 max-size-time=0 ! \
  hailofilter so-path=$DRAW_POSTPROCESS_S0 qos=false debug=False
! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
```

Each `hailofilter` performs a given post-process. In this case the first performs the `centerpose` post-process and the second performs box and skeleton drawing.

6.

```
videoconvert ! \
fpsdisplaysink video-sink=xvimagesink name=hailo_display
sync=true text-overlay=false ${additional_parameters}
```

Apply the final convert to let GStreamer utilize the format required by the `fpsdisplaysink` element

NOTE: Additional details about the pipeline provided in further examples

Segmentation Pipelines

Overview

`semantic_segmentation.sh` demonstrates semantic segmentation on one video file source. This is done by running a `single-stream object semantic segmentation pipeline` on top of GStreamer using the Hailo-8 device.

Options

`semantic_segmentation.sh` demonstrates semantic segmentation on one video file source. This is done by running a `single-stream object semantic segmentation pipeline` on top of GStreamer using the Hailo-8 device.

Options

```
./semantic_segmentation.sh [--input FILL-ME]
```

- `--input` is an optional flag, a path to the video displayed (default is `full_mov_slow.mp4`).
- `--print-gst-launch` is a flag that prints the ready `gst-launch` command without running it
- `--show-fps` is an optional flag that enables printing FPS on screen

Supported Network

- 'fcn8_resnet_v1_18' - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/fcn8_resnet_v1_18.yaml

Run

```
cd $TAPPAS_WORKSPACE/apps/gstreamer/x86/segmentation
./semantic_segmentation.sh
```

The output should look like:



How it works

This section is optional and provides a drill-down into the implementation of the **semantic segmentation** app with a focus on explaining the **GStreamer** pipeline. This section uses **resnet18_fcn8_fhd** as an example network so network input width, height, hef name, are set accordingly.

Model

- **fcn8_resnet_v1_18** in resolution of 1920x1024x3.
- Numeric accuracy 65.18mIOU.
- Pre trained on cityscapes using GlounCV and a resnet-18- FCN8 architecture.

```
gst-launch-1.0 \
  filesrc location=$video_device ! decodebin ! \
  videoscale ! video/x-raw,pixel-aspect-ratio=1/1 ! videoconvert !
  \
  queue leaky=no max-size-buffers=13 max-size-bytes=0 max-size-
time=0 ! \
  hailonet hef-path=$hef_path debug=False is-active=true qos=false
batch-size=8 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailofilter so-path=$DRAW_POSTPROCESS_S0 qos=false debug=False !
  \
  videoconvert ! \
  fpsdisplaysink video-sink=xvimagesink name=hailo_display
sync=true text-overlay=false ${additional_parameters}
```

Let's explain this pipeline section by section:

1. `filesrc location=$video_device ! decodebin ! videoconvert !`

Specifies the location of the video used, then decodes and converts to the required format.

```
2. videoscale ! video/x-raw,pixel-aspect-ratio=1/1 ! \
```

Re-scales the video dimensions to fit the input of the network. In this case it is rescaling the video to 1920x1024 with the caps negotiation of `hailonet`.

```
3. queue leaky=no max-size-buffers=13 max-size-bytes=0 max-size-time=0 ! \
```

Before sending the frames into `hailonet` element, set a queue so no frames are lost (Read more about queues [here](#))

```
4. hailonet hef-path=$hef_path debug=False is-active=true
qos=false batch-size=8 ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \
```

Performs the inference on the Hailo-8 device.

```
5. hailofilter so-path=$DRAW_POSTPROCESS_S0 qos=false debug=False
! \
```

Performs a given draw-process, in this case, performs `resnet18_fcn8_fhd` semantic segmentation drawing per pixel.

```
6. videoconvert ! \
fpsdisplaysink video-sink=xvimagesink name=hailo_display
sync=true text-overlay=false ${additonal_parameters}
```

Apply the final convert to let GStreamer utilize the format required by the `fpsdisplaysink` element

NOTE: Additional details about the pipeline provided in further examples

Facial Landmarks Pipeline

Overview:

`facial_landmarks.sh` demonstrates facial landmarking on one video file source and verifies Hailo's configuration. This is done by running a **single-stream facial landmarking pipeline** on top of GStreamer using the Hailo-8 device.

Options

```
./facial_landmarks.sh
```

- `--input` is an optional flag, a path to the video displayed (default is `faces_120_120.mp4`).
- `--show-fps` is an optional flag that enables printing FPS on screen.
- `--print-gst-launch` is a flag that prints the ready `gst-launch` command without running it"

Run

```
cd $TAPPAS_WORKSPACE/apps/gstreamer/x86/facial_landmarks/
./facial_landmarks.sh
```

The output should look like:

```
root@hailo_tappas:/hailo/apps/x86/face_landmarks# ./face_landmarks.sh
gst-launch-1.0 filesrc location=/hailo/apps/x86/face_landmarks/faces_120_120.mp4 name=src_0 ! decodebin ! videoscale ! v
ideo/x-raw,pixel-aspect-ratio=1/1 ! videoconvert ! queue ! hailonet hef-path=/hailo/apps/x86/face_landmarks/tddfa_mobile
net_v1_app/tddfa_mobilenet_v1.hef debug=False is-active=true qos=false ! queue leaky=no max_size_buffers=30 max-size-byt
es=0 max-size-time=0 ! hailofilter so-path=/hailo/apps/x86/gstreamer/libface_landmarks_post.so qos=false debug=False !
queue leaky=no max_size_buffers=30 max-size-bytes=0 max-size-time=0 ! hailofilter so-path=/hailo/apps/x86/gstreamer/lib
face_landmarks_draw.so qos=false debug=False ! videoconvert ! fpsdisplaysink video-sink=ximagesink name=hailo_display sy
nc=true text-overlay=false
Setting pipeline to PAUSED ...
Pipeline is PREROLLING ...
Redistribute latency...
Redistribute latency...
Pipeline is PREROLLED ...
Setting pipeline to PLAYING ...
New clock: GstSystemClock

```



How it works

This section is optional and provides a drill-down into the implementation of the **face landmarks** app with a focus on explaining the **GStreamer** pipeline. This section uses `tddfa_mobilenet_v1` as an example network so network input width, height, hef name, are set accordingly.

```
gst-launch-1.0 \
  $source_element ! decodebin ! \
  videoscale ! video/x-raw,pixel-aspect-ratio=1/1 ! videoconvert ! \
  \
  queue leaky=no max_size_buffers=30 max-size-bytes=0 max-size-
```

```

time=0 ! \
  hailonet hef-path=$hef_path debug=False is-active=true qos=false
! \
  queue leaky=no max_size_buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailofilter so-path=$POSTPROCESS_SO qos=false debug=False ! \
  queue leaky=no max_size_buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailofilter so-path=$DRAW_POSTPROCESS_SO qos=false debug=False !
\
  videoconvert ! \
  fpsdisplaysink video-sink=xvimagesink name=hailo_display
sync=true

```

Let's explain this pipeline section by section:

1. `filesrc location=$video_device ! decodebin ! videoconvert !`

Specifies the location of the video used, then decodes and converts to the required format.

2. `videoscale ! video/x-raw,pixel-aspect-ratio=1/1 ! \`

Re-scales the video dimensions to fit the input of the network. In this case it is rescaling the video to 120x120 with the caps negotiation of `hailonet`.

3. `queue leaky=no max_size_buffers=30 max-size-bytes=0 max-size-time=0 ! \`

)Before sending the frames into `hailonet` element, set a queue so no frame are lost (Read more about queues [here](#))

4. `hailonet hef-path=$hef_path debug=False is-active=true qos=false ! \
queue leaky=no max_size_buffers=30 max-size-bytes=0 max-size-time=0 ! \`

Performs the inference on the Hailo-8 device.

5. `hailofilter so-path=$POSTPROCESS_SO qos=false debug=False ! \
queue leaky=no max_size_buffers=30 max-size-bytes=0 max-size-time=0 ! \
hailofilter so-path=$DRAW_POSTPROCESS_SO qos=false debug=False ! \`

Performs a given post-process, in that case, performs `tddfa_mobilenet_v1` post-process and then landmarks drawing.

```
6. videoconvert ! \  
   fpsdisplaysink video-sink=xvimagesink name=hailo_display  
   sync=true
```

Apply the final convert to let GStreamer utilize the format required by the `fpsdisplaysink` element.

NOTE: Additional details about the pipeline provided in further examples

Face Detection Pipeline

Overview:

The purpose of `face_detection.sh` is to demonstrate face detection on one video file source and to verify Hailo's configuration. This is done by running a `single-stream face detection pipeline` on top of GStreamer using the Hailo-8 device.

Options

```
/face_detection.sh
```

- `--network` is a flag that sets which network to use. choose from [lightface, retinaface], default is lightface. this will set the hef file to use, the `hailofilter` function to use, and the scales of the frame to match the width/height input dimensions of the network.
- `--input` is an optional flag, a path to the video displayed (default is `face_detection.mp4`).
- `--show-fps` is an optional flag that enables printing FPS on screen.
- `--print-gst-launch` is a flag that prints the ready `gst-launch` command without running it"

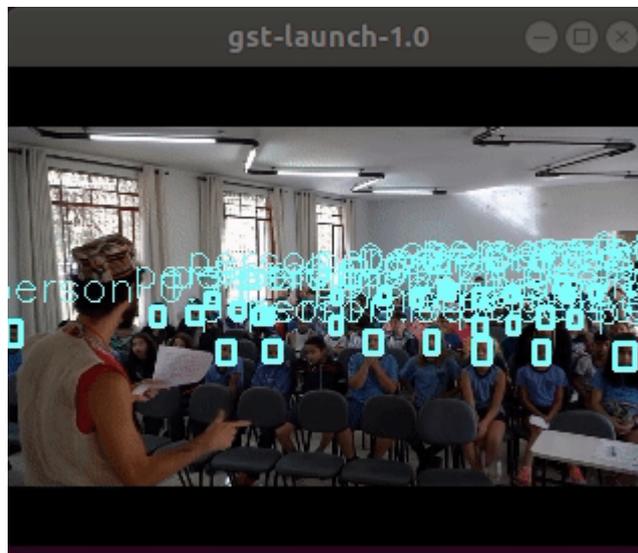
Supported Networks

- 'retinaface' - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/retinaface_mobile_net_v1.yaml
- 'lightface' - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/lightface_slim.yaml

Run

```
cd $TAPPAS_WORKSPACE/apps/gstreamer/x86/face_detection/  
./face_detection.sh
```

The output should look like:



How it works

This section is optional and provides a drill-down into the implementation of the **face detection** app with a focus on explaining the **GStreamer** pipeline. This section uses **lightface_slim** as an example network so network input width, height, hef name, are set accordingly.

```
gst-launch-1.0 \
  filesrc location=$input_source ! decodebin ! \
  videoscale ! video/x-raw, pixel-aspect-ratio=1/1 ! videoconvert !
\
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailonet hef-path=$hef_path debug=False is-active=true qos=false
! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailofilter function-name=$network_name so-path=$POSTPROCESS_S0
qos=false debug=False ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailofilter so-path=$DRAW_POSTPROCESS_S0 qos=false debug=False !
\
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  videoconvert ! \
  fpsdisplaysink video-sink=xvimagesink name=hailo_display
sync=false text-overlay=false ${additional_parameters}
```

Let's explain this pipeline section by section:

1. `filesrc location=$video_device ! decodebin ! videoconvert !`

Specifying the location of the video used, then decode and convert to the required format.

```
2. videoscale ! video/x-raw,pixel-aspect-ratio=1/1 ! \
```

Re-scale the video dimensions to fit the input of the network. In this case it is rescaling the video to 320x240 with the caps negotiation of `hailonet`. #

```
3. queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \
```

Before sending the frames into `hailonet` element set a queue so no frame would be lost (Read more about queue [here](#))

```
4. hailonet hef-path=$hef_path debug=False is-active=true
qos=false ! \
queue leaky=no max_size_buffers=30 max-size-bytes=0 max-size-time=0 ! \
```

Sending and receiving the data, separated by a non-leaky queue.

NOTE: `qos` must be disabled for `hailonet` since dropping frames may cause these elements to run out of alignment.

```
5. hailofilter so-path=$POSTPROCESS_S0 qos=false debug=False ! \
queue leaky=no max_size_buffers=30 max-size-bytes=0 max-size-time=0 ! \
hailofilter so-path=$DRAW_POSTPROCESS_S0 qos=false debug=False
! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \
```

Performs given post-process, in that case, performers `lightface_slim` post-process and detection box drawing

```
6. videoconvert ! \
fpsdisplaysink video-sink=xvimagesink name=hailo_display
sync=true
```

Apply the final convert to let GStreamer find out the format required by the `fpsdisplaysink` element

NOTE: Additional details about the pipeline provided in further examples

Face Detection and Facial Landmarking Pipeline

`face_detection_and_landmarks.sh` demonstrates face detection and facial landmarking on one video file source. This is done by running a face detection pipeline (infer + postprocessing), cropping and scaling all detected faces, and sending them into a 2nd network of facial landmarking. All resulting detections and landmarks are then aggregated and drawn on the original frame. The two networks are running using one Hailo-8 device with two `hailonet` elements.

Options

```
./face_detection_and_landmarks.sh [OPTIONS] [-i INPUT_PATH]
```

- `-i --input` is an optional flag, a path to the video/camera displayed.
- `--print-gst-launch` prints the ready `gst-launch` command without running it
- `--show-fps` optional - enables printing FPS on screen

Run

```
cd $TAPPAS_WORKSPACE/apps/gstreamer/x86/cascading_networks
./face_detection_and_landmarks.sh
```

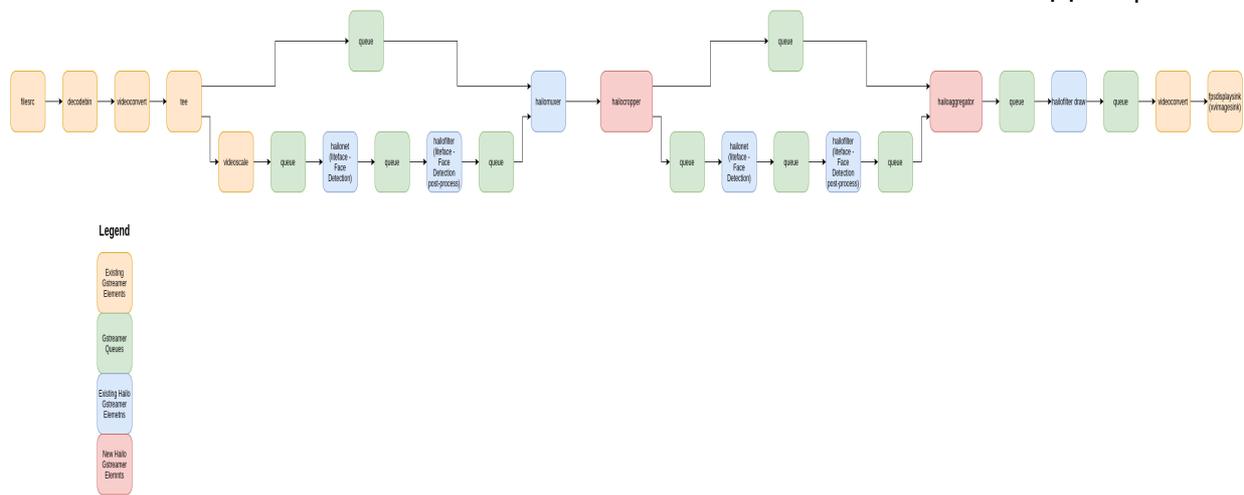
The output should look like:



Model

- `lightface_slim` in resolution of 320X240X3 - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/lightface_slim.yaml.
- `tddfa_mobilenet_v1` in resolution of 120X120X3 - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/tddfa_mobilenet_v1.yaml.

How it works



This section is optional and provides a drill-down into the implementation of the app with a focus on explaining the `GStreamer` pipeline. This section uses `lightface_slim` as an example network so network input width, height, hef name, are set accordingly.

```
FACE_DETECTION_PIPELINE="videoscale qos=false ! \
    queue leaky=no max-size-buffers=3 max-size-bytes=0 max-size-
time=0 ! \
    hailonet net-
name=joined_lightface_slim_tddfa_mobilenet_v1/lightface_slim \
    hef-path=$hef_path is-active=true qos=false ! \
    queue leaky=no max-size-buffers=3 max-size-bytes=0 max-size-
time=0 ! \
    hailofilter so-path=$detection_postprocess_so function-
name=lightface qos=false ! \
    queue leaky=no max-size-buffers=3 max-size-bytes=0 max-size-
time=0"

FACIAL_LANDMARKS_PIPELINE="queue leaky=no max-size-buffers=3 max-
size-bytes=0 max-size-time=0 ! \
    hailonet net-
name=joined_lightface_slim_tddfa_mobilenet_v1/tddfa_mobilenet_v1 \
    hef-path=$hef_path is-active=true qos=false ! \
    queue leaky=no max-size-buffers=3 max-size-bytes=0 max-size-
time=0 ! \
    hailofilter function-name=facial_landmarks_merged so-
path=$landmarks_postprocess_so qos=false ! \
    queue leaky=no max-size-buffers=3 max-size-bytes=0 max-size-
time=0"

gst-launch-1.0 \
    $source_element ! \
    tee name=t hailomuxer name=hmux \
    t. ! queue leaky=no max-size-buffers=3 max-size-bytes=0 max-size-
time=0 ! hmux. \
    t. ! $FACE_DETECTION_PIPELINE ! hmux. \
    hmux. ! queue leaky=no max-size-buffers=3 max-size-bytes=0 max-
size-time=0 ! \
    hailocropper internal-offset=$internal_offset name=cropper
hailoaggregator name=agg \
    cropper. ! queue leaky=no max-size-buffers=3 max-size-bytes=0
max-size-time=0 ! agg. \
```

```
cropper. ! $FACIAL_LANDMARKS_PIPELINE ! agg. \
agg. ! queue leaky=no max-size-buffers=3 max-size-bytes=0 max-
size-time=0 ! \
hailofilter so-path=$landmarks_draw_so qos=false ! \
queue leaky=no max-size-buffers=3 max-size-bytes=0 max-size-
time=0 ! videoconvert ! \
fpsdisplaysink video-sink=xvimagesink name=hailo_display
sync=false text-overlay=false ${additional_parameters}
```

Let's explain this pipeline section by section:

1. `filesrc location=$video_device ! decodebin ! videoconvert !`

Specifies the location of the video used, then decodes and converts to the required format.

2. `tee name=t hailomuxer name=hmux`

Split into two threads - one for doing face detection, the other one for getting the original frame. We merge those 2 threads back by using hailomuxer, which takes the frame from it's first sink and adds the metadata from the other sink

3. `t. ! queue leaky=no max-size-buffers=3 max-size-bytes=0 max-
size-time=0 ! hmux. \`

The first thread, only passes the original frame.

4. `t. ! $FACE_DETECTION_PIPELINE ! hmux. \`

The second thread performs the face detection pipeline where `FACE_DETECTION_PIPELINE` is:

- `videoscale qos=false ! \`

Scales the picture to a resolution negotiated with the hailonet down the pipeline, according to the needed resolution by the hef file.

- `queue leaky=no max-size-buffers=3 max-size-bytes=0 max-
size-time=0 ! \`

Before sending the frames into `hailonet` element, set a queue so no frames are lost (Read more about queues [here](#))

- `hailonet hef-path=$hef_path debug=False is-active=true net-
name=joined_lightface_slim_tddfa_mobilenet_v1/lightface_sli
m qos=false batch-size=1`

```
queue leaky=no max-size-buffers=3 max-size-bytes=0 max-
size-time=0 ! \
```

Performs the inference on the Hailo-8 device.

- ```
hailofilter so-path=$detection_postprocess_so qos=false
debug=False ! \
queue leaky=no max-size-buffers=3 max-size-bytes=0 max-
size-time=0 ! \
```

Performs a given post-process, in this case, performs `lightface_slim` face detection post-processing.

5. 

```
hmux. ! queue leaky=no max-size-buffers=3 max-size-bytes=0 max-
size-time=0 ! \
hailocropper internal-offset=$internal_offset name=cropper
hailoaggregator name=agg \
```

links the hailomuxer to a queue and defines the cascading network elements `hailocropper` and `hailoaggregator`. `hailocropper` splits the pipeline into 2 threads, the first thread passes the original frame, the other thread passes the cropped of the original frame created by `hailocropper` according to the detections added to the buffer by prior `hailofilter` post-processing, the buffers are also scaled to the following `hailonet`, done by caps negotiation. The `hailoaggregator` gets the original frame and then knows to wait for all related cropped buffers and add all related metadata on the original frame, and send it forward.

6. 

```
cropper. ! queue leaky=no max-size-buffers=3 max-size-bytes=0
max-size-time=0 ! agg. \
```

The first part of the cascading network pipeline, passes the original frame on the bypass pads to `hailoaggregator`.

7. 

```
cropper. ! $FACIAL_LANDMARKS_PIPELINE ! agg. \
```

The second part of the cascading network pipeline, performs a second network on all detections, which are cropped and scaled to the needed resolution by the HEF in the `hailonet`. `FACIAL_LANDMARKS_PIPELINE` consists of:

- ```
queue leaky=no max-size-buffers=3 max-size-bytes=0 max-
size-time=0 ! \
```

Before sending the frames into the `hailonet` element, set a queue so no frames are lost (Read more about queues [here](#))

- `hailonet net-name=joined_lightface_slim_tddfa_mobilenet_v1/tddfa_mobilenet_v1 \`
`hef-path=$hef_path is-active=true qos=false ! \`

Performs inference on the Hailo-8 device.

- `hailofilter function-name=facial_landmarks_merged so-path=$landmarks_postprocess_so qos=false ! \`
`queue leaky=no max-size-buffers=3 max-size-bytes=0 max-size-time=0`

Performs a given post-process, in this case, performs `tddfa_mobilenet_v1` facial landmarks post-processing.

8. `agg. ! queue leaky=no max-size-buffers=3 max-size-bytes=0 max-size-time=0 ! \`
`hailofilter so-path=$landmarks_draw_so qos=false ! \`

Aggregates all detected faces with their landmarks on the original frame, and draws them over the frame using the `hailofilter` with specific drawing function.

9. `queue leaky=no max-size-buffers=3 max-size-bytes=0 max-size-time=0 ! videoconvert ! \`
`fpsdisplaysink video-sink=xvimagesink name=hailo_display sync=false text-overlay=false`

Display the final image using `fpsdisplaysink`.

NOTE: Additional details about the pipeline provided in further examples

Tiling Pipeline

Single Scale Tiling

Single scale tiling FHD Gstreamer pipeline demonstrates splitting each frame into several tiles which are processed independently by `hailonet` element. This method is especially effective for detecting small objects in high-resolution frames. This process is separated into 4 elements -

- `hailotilecropper` which splits the frame into tiles, by separating the frame into rows and columns (given as parameters to the element).
- `hailonet` which performs an inference on each frame on the Hailo8 device.
- `hailofilter` which performs the postprocess - parses the tensor output to detections.
- `hailotileaggregator` which aggregates the cropped tiles and stitches them back to the original resolution.

Model

- `ssd_mobilenet_v1_visdrone` in resolution of 300X300 - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/ssd_mobilenet_v1_visdrone.yaml.

The VisDrone dataset consists of only small objects which we can assume are always confined within an single tile. As such it is better suited for running single-scale tiling with little overlap and without additional filtering.

Options

```
./tiling.sh [OPTIONS] [-i INPUT_PATH]
```

- `-i --input` is an optional flag, a path to the video file displayed.
- `--print-gst-launch` prints the ready `gst-launch` command without running it
- `--show-fps` optional - enables printing FPS on screen
- `--tiles-x-axis` optional - set number of tiles along x axis (columns)
- `--tiles-y-axis` optional - set number of tiles along y axis (rows)
- `--overlap-x-axis` optional - set overlap in percentage between tiles along x axis (columns)
- `--overlap-y-axis` optional - set overlap in percentage between tiles along y axis (rows)
- `--iou-threshold` optional - set iou threshold for NMS.
- `--sync-pipeline` optional - set pipeline to sync to video file timing.

Run

```
cd $TAPPAS_WORKSPACE/apps/gstreamer/x86/tiling
./tiling.sh
```

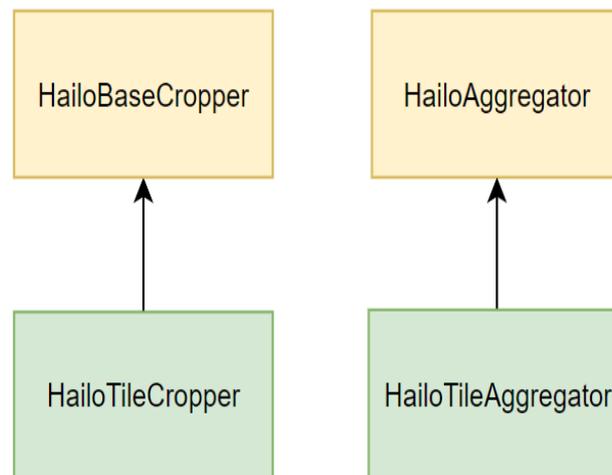
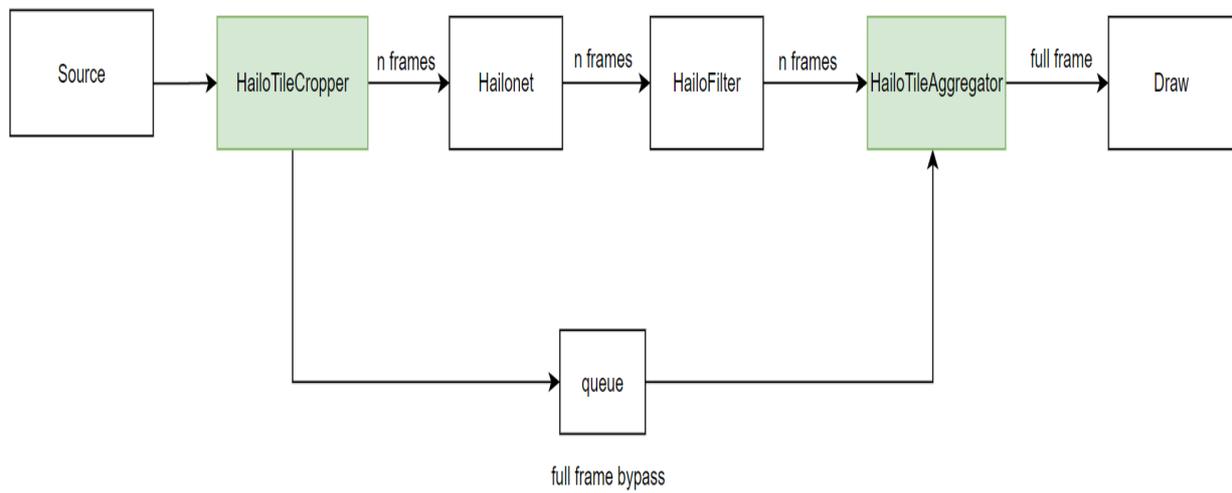
The output should look like:

```

e so-path=/local/workspace/tappas/apps/gstreamer/x86/lib/libnoblent_ssd_post.so qos=false debug=false | queue leaky=no max-size-buffers=3 max-size-bytes=0 max-size-time=0 | agg. agg. | queue leaky=no
ax-size-buffers=3 max-size-bytes=0 max-size-time=0 | hallooverlay so-path=/local/workspace/tappas/apps/gstreamer/x86/lib/libdetection_draw.so qos=false | queue leaky=no max-size-buffers=3 max-size-bytes
0 max-size-time=0 | videoconvert | fpsdisplaysink video-sink=xvimagesink name=hailo_display sync=false text-overlay=false -v | grep hailo_display
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display/GstXvImageSink:xvimagesink8: sync = false
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display.GstGhostPad:stnk.GstProxyPad:proxypad3: caps = video/x-raw, width=(int)1280, height=(int)720, interlace-mode=(string)progressive, multiview-mode=(str
ng)mono, multiview-flags=(GstVideoMultiviewFlagsSet)0:ffffffff:/right-view-first/left-flipped/left-flipped/right-flipped/right-flipped/half-aspect/mixed-mono, pixel-aspect-ratio=(fraction)1/1, framerate=
(fraction)24000/1081, format=(string)YV12
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display.GstGhostPad:stnk.GstPad:sink: caps = video/x-raw, width=(int)1280, height=(int)720, interlace-mode=(string)progressive, multiview-mode=(st
tag)mono, multiview-flags=(GstVideoMultiviewFlagsSet)0:ffffffff:/right-view-first/left-flipped/left-flipped/right-flipped/right-flipped/half-aspect/mixed-mono, pixel-aspect-ratio=(fraction)1/1, framerate
(fraction)24000/1081, format=(string)YV12
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display.GstGhostPad:stnk: caps = video/x-raw, width=(int)1280, height=(int)720, interlace-mode=(string)progressive, multiview-mode=(string)mono, multiview-fl
gs=(GstVideoMultiviewFlagsSet)0:ffffffff:/right-view-first/left-flipped/left-flipped/right-flipped/right-flipped/half-aspect/mixed-mono, pixel-aspect-ratio=(fraction)1/1, framerate=(fraction)24000/1081,
ormat=(string)YV12
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display/GstXvImageSink:xvimagesink8: sync = false
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 27, dropped: 0, current: 53.15, average: 53.15
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 53, dropped: 0, current: 50.74, average: 51.94
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 79, dropped: 0, current: 50.83, average: 51.57
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 105, dropped: 0, current: 50.83, average: 51.38
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 131, dropped: 0, current: 50.64, average: 51.23
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 157, dropped: 0, current: 50.98, average: 51.19
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 183, dropped: 0, current: 50.64, average: 51.11
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 209, dropped: 0, current: 50.77, average: 51.07
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 235, dropped: 0, current: 50.55, average: 51.01
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 261, dropped: 0, current: 51.23, average: 51.03
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 287, dropped: 0, current: 50.63, average: 51.08
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 313, dropped: 0, current: 50.73, average: 50.97
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 339, dropped: 0, current: 51.06, average: 50.98
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 365, dropped: 0, current: 50.73, average: 50.96
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 391, dropped: 0, current: 50.45, average: 50.93
oob|hailo_tappas:/local/workspace/tappas# ./app/gstreamer/x86/tiling/tiling.sh -show fps -tiles x-axis 4 -tiles y-axis 3
Printing fps
Running
st-launch-1.0 filesrc location=/local/workspace/tappas/apps/gstreamer/x86/tiling/resources/17er_tiber.mp4 name=src_0 | decodebin | videoconvert qos=false | video/x-raw, pixel-aspect-ratio=1/1 | queue leak
y=no max-size-buffers=3 max-size-bytes=0 max-size-time=0 | hailolettercropper internal-offset=true name=cropper tiles-along-x-axis=4 tiles-along-y-axis=3 overlap-x-axis=0.815 overlap-y-axis=0.617 hailoaggr
gatar name=agg cropper. | queue leaky=no max-size-buffers=3 max-size-bytes=0 max-size-time=0 | agg. cropper. | hailonet hef-path=/local/workspace/tappas/apps/gstreamer/x86/tiling/resources/ssd_noblent_
_lvidrone.hef device-id=8080:07:00.0 debug=false is-active=true qos=false batch-size=1 | queue leaky=no max-size-buffers=3 max-size-bytes=0 max-size-time=0 | hailopost-function name=noblent_ssd_vidsdro
ne so-path=/local/workspace/tappas/apps/gstreamer/x86/lib/libnoblent_ssd_post.so qos=false debug=false | queue leaky=no max-size-buffers=3 max-size-bytes=0 max-size-time=0 | agg. agg. | queue leaky=no
max-size-buffers=3 max-size-bytes=0 max-size-time=0 | hallooverlay so-path=/local/workspace/tappas/apps/gstreamer/x86/lib/libdetection_draw.so qos=false | queue leaky=no max-size-buffers=3 max-size-bytes
0 max-size-time=0 | videoconvert | fpsdisplaysink video-sink=xvimagesink name=hailo_display sync=false text-overlay=false -v | grep hailo_display
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display/GstXvImageSink:xvimagesink8: sync = false
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display.GstGhostPad:stnk.GstProxyPad:proxypad3: caps = video/x-raw, width=(int)1280, height=(int)720, interlace-mode=(string)progressive, multiview-mode=(str
ng)mono, multiview-flags=(GstVideoMultiviewFlagsSet)0:ffffffff:/right-view-first/left-flipped/left-flipped/right-flipped/right-flipped/half-aspect/mixed-mono, pixel-aspect-ratio=(fraction)1/1, framerate=
(fraction)24000/1081, format=(string)YV12
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display.GstGhostPad:stnk.GstPad:sink: caps = video/x-raw, width=(int)1280, height=(int)720, interlace-mode=(string)progressive, multiview-mode=(st
tag)mono, multiview-flags=(GstVideoMultiviewFlagsSet)0:ffffffff:/right-view-first/left-flipped/left-flipped/right-flipped/right-flipped/half-aspect/mixed-mono, pixel-aspect-ratio=(fraction)1/1, framerate
(fraction)24000/1081, format=(string)YV12
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display.GstGhostPad:stnk: caps = video/x-raw, width=(int)1280, height=(int)720, interlace-mode=(string)progressive, multiview-mode=(string)mono, multiview-fl
gs=(GstVideoMultiviewFlagsSet)0:ffffffff:/right-view-first/left-flipped/left-flipped/right-flipped/right-flipped/half-aspect/mixed-mono, pixel-aspect-ratio=(fraction)1/1, framerate=(fraction)24000/1081,
ormat=(string)YV12
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display/GstXvImageSink:xvimagesink8: sync = false
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 27, dropped: 0, current: 53.31, average: 53.31
GstPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 53, dropped: 0, current: 50.78, average: 52.04
Croot|hailo_tappas:/local/workspace/tappas# ^C
oob|hailo_tappas:/local/workspace/tappas#
0# 8:hailo@hailo-79521:2022-01-16 07:59:48.552*2 root|hailo_tappas:/l/ 16:20 26-Jan-22

```

How it works



```

filesrc location=$input_source name=src_0 ! decodebin ! videoconvert
qos=false

```

`filesrc` - source of the pipeline reads the video file and decodes it.

```

TILE_CROPPER_ELEMENT="hailotilecropper internal-
offset=$internal_offset name=cropper \
tiles-along-x-axis=$tiles_along_x_axis tiles-along-y-
axis=$tiles_along_y_axis overlap-x-axis=$overlap_x_axis overlap-y-
axis=$overlap_y_axis"

```

```

$TILE_CROPPER_ELEMENT hailotileaggregator iou-
threshold=$iou_threshold name=agg \
cropper. ! queue leaky=no max-size-buffers=3 max-size-bytes=0 max-
size-time=0 ! agg. \
cropper. ! $DETECTION_PIPELINE ! agg. \

```

`hailotilecropper` splits the pipeline into 2 threads, the first thread passes the original frame, the other thread passes the crops of the original frame created by `hailotilecropper` according to given tiles number per x/y axis and overlap parameters. The buffers are also scaled to the following `hailonet`, done by caps negotiation. The `hailotileaggregator` gets the original frame and then knows to wait

for all related cropped buffers and add all related metadata on the original frame, sending everything together once aggregated. It also performs NMS process to merge detections on overlap tiles.

```
DETECTION_PIPELINE="\
hailonet hef-path=$hef_path device-id=$hailo_bus_id is-active=true
qos=false batch-size=1 ! \
queue leaky=no max-size-buffers=3 max-size-bytes=0 max-size-time=0 !
\
hailofilter2 function-name=$postprocess_func_name so-
path=$detection_postprocess_so qos=false ! \
queue leaky=no max-size-buffers=3 max-size-bytes=0 max-size-time=0"
```

focusing on the detection part: `hailonet` performs inference on the Hailo-8 device running `mobilenet_v1_visdrone.hef` for each tile crop. `hailofilter` performs the mobilenet postprocess and creates the detection objects to pass through the pipeline.

```
agg. ! queue leaky=no max-size-buffers=3 max-size-bytes=0 max-size-
time=0 ! \
hailooverlay qos=false ! \
```

`hailotileaggregator` sends the frame forward into the `hailooverlay` which draws the detections over the frame.

```
queue leaky=no max-size-buffers=3 max-size-bytes=0 max-size-time=0 !
videoconvert ! \

fpsdisplaysink video-sink=xvimagesink name=hailo_display
```

Pipeline ends in the sink to the display.

Multi Scale Tiling

Multi-scale tiling FHD Gstreamer pipeline demonstrates a case where the video and the training dataset includes objects in different sizes. Dividing the frame to small tiles might miss large objects or "cut" them to small objects. The solution is to split each frame into number of scales (layers) each includes several tiles.

Multi-scale tiling strategy also allows us to filter the correct detection over several scales. For example we use 3 sets of tiles at 3 different scales:

- Large scale, one tile to cover the entire frame (1x1)
- Medium scale dividing the frame to 2x2 tiles.
- Small scale dividing the frame to 3x3 tiles.

In this mode we use $1 + 4 + 9 = 14$ tiles for each frame. We can simplify the process by highlighting the main tasks: crop -> inference -> post-process -> aggregate -> remove exceeded boxes -> remove large landscape -> perform NMS

Model

- `mobilenet_ssd` in resolution of 300X300X3. https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/ssd_mobilenet_v1.yaml

Options

```
./multi_scale_tiling.sh [OPTIONS] [-i INPUT_PATH]
```

- `-i --input` is an optional flag, a path to the video file displayed.
- `--print-gst-launch` prints the ready `gst-launch` command without running it
- `--show-fps` optional - enables printing FPS on screen
- `--tiles-x-axis` optional - set number of tiles along x axis (columns)
- `--tiles-y-axis` optional - set number of tiles along y axis (rows)
- `--overlap-x-axis` optional - set overlap in percentage between tiles along x axis (columns)
- `--overlap-y-axis` optional - set overlap in percentage between tiles along y axis (rows)
- `--iou-threshold` optional - set iou threshold for NMS.
- `--border-threshold` optional - set border threshold to Remove tile's exceeded objects.
- `--scale-level` optional - set scales (layers of tiles) in addition to the main layer.
1: [(1 X 1)] 2: [(1 X 1), (2 X 2)] 3: [(1 X 1), (2 X 2), (3 X 3)]'

Run

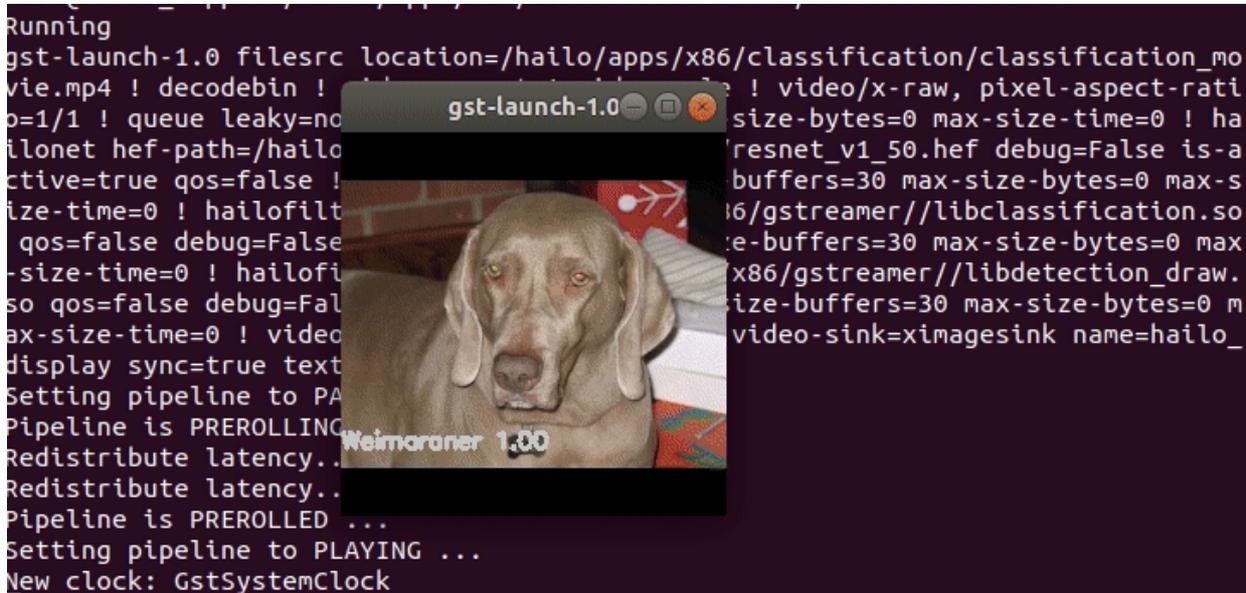
```
cd $TAPPAS_WORKSPACE/apps/gstreamer/x86/tiling
./multi_scale_tiling.sh
```

The output should look like:

Classification Pipeline

Overview

The purpose of `classification.sh` is to demonstrate classification on one video file source. This is done by running a `single-stream object classification pipeline` on top of GStreamer using the Hailo-8 device.



Options

```
./classification.sh [--input FILL-ME]
```

- `--input` is an optional flag, a path to the video displayed (default is `classification_movie.mp4`).
- `--show-fps` is a flag that prints the pipeline's fps to the screen.
- `--print-gst-launch` is a flag that prints the ready `gst-launch` command without running it.

Supported Networks:

- 'resnet_v1_50' - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/resnet_v1_50.yaml

Run

```
cd $TAPPAS_WORKSPACE/apps/gstreamer/x86/classification
./classification.sh
```

How it works

This section is optional and provides a drill-down into the implementation of the `classification` app with a focus on explaining the `GStreamer` pipeline. This section uses `resnet_v1_50` as an example network so network input width, height, and hef name are set accordingly.

```
gst-launch-1.0 \
  filesrc location=$input_source ! decodebin ! videoconvert ! \
  videoscale ! video/x-raw, pixel-aspect-ratio=1/1 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailonet hef-path=$hef_path debug=False is-active=true qos=false
! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailofilter so-path=$POSTPROCESS_S0 qos=false debug=False ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailooverlay qos=false ! \
  videoconvert ! \
  fpsdisplaysink video-sink=xvimagesink name=hailo_display
sync=true text-overlay=false ${additional_parameters}"
```

Let's explain this pipeline section by section:

1. `filesrc location=$video_device ! decodebin ! videoconvert !`

Specifies the location of the video used, then decode and convert to the required format.

2. `videoscale ! video/x-raw, pixel-aspect-ratio=1/1 ! \`

Re-scale the video dimensions to fit the input of the network. In this case it is rescaling the video to 112X112 with the caps negotiation of `hailonet`. `hailonet` Extracts the needed resolution from the HEF file during the caps negotiation, and makes sure that the needed resolution is passed from previous elements.

3. `queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \`

Before sending the frames into `hailonet` element set a queue so no frames are lost (Read more about queues [here](#))

4. `hailonet hef-path=$hef_path debug=False is-active=true
qos=false ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \`

Performs the inference on the Hailo-8 device.

5. `hailofilter2 so-path=$POSTPROCESS_S0 qos=false debug=False ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \`

Performs a given post-process, in this case, performs `resnet_v1_50` classification post-process, which is mainly doing top1 on the inference output.

6.

```
hailooverlay qos=false ! \  
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-  
time=0 ! \  

```

Performs given draw-process, in that case, performs drawing the top1 class name over the image.

7.

```
videoconvert ! \  
fpsdisplaysink video-sink=xvimagesink name=hailo_display  
sync=true text-overlay=false "${additional_parameters}"  

```

Apply the final convert to let GStreamer utilize the format required by the `fpsdisplaysink` element.

Multi-Stream and Multi-Device Pipeline

Overview

This GStreamer pipeline demonstrates object detection on 8 camera streams over RTSP protocol. This pipeline also demonstrates using two hailo8 devices in parallel.

All the streams are processed in parallel through the decode and scale phases, and enter the Hailo devices frame by frame. **Each** hailo device is in charge of one inference task (one for yolov5 and the other for centerpose)

Afterwards postprocess and drawing phases add the classified object and bounding boxes to each frame.

The last step is to match each frame back to its respective stream and output all of them to the display.

Real Time Streaming Protocol (RTSP) is a network control protocol designed for use in entertainment and communications systems to control streaming media servers. The protocol is used for establishing and controlling media sessions between endpoint.

Prerequisites

- TensorPC
- Ubuntu 18.04
- [RTSP Cameras](#), We recommend using: [AXIS M10 Network Cameras](#)
- Two Hailo-8 devices connected via PCIe

Preparations

1. Before running, configuration of the RTSP camera sources is required. Open the `rtsp_detection_and_pose_estimation.sh` in edit mode with your preferred editor. Configure the eight sources to match your own cameras.

```
readonly SRC_0="rtsp://<ip address>/?h264x=4 user-id=<username> user-pw=<password>"
readonly SRC_1="rtsp://<ip address>/?h264x=4 user-id=<username> user-pw=<password>"
etc..
```

Run the pipeline

```
./rtsp_detection_and_pose_estimation.sh
```

1. `--show-fps` prints the fps to the output.

```
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 513, dropped: 0, current: 29.85, average: 26.68
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 528, dropped: 0, current: 29.11, average: 26.74
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 543, dropped: 0, current: 28.66, average: 26.79
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 558, dropped: 0, current: 28.28, average: 26.83
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 570, dropped: 0, current: 23.48, average: 26.75
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 584, dropped: 0, current: 26.98, average: 26.76
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 598, dropped: 0, current: 27.85, average: 26.78
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 613, dropped: 0, current: 28.46, average: 26.82
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 626, dropped: 0, current: 25.70, average: 26.80
tPipeline:pipeline0/GstFPSDisplaySink:hailo_display: last-message = rendered: 641, dropped: 0, current: 28.17, average: 26.83
```

2. `--disable-vaapi` disables the vaapi accelerator usage. This replaces decoder elements and videosink elements from vaapi to decodebin, videoscale and autovideosink.
3. `--num-of-sources` sets the number of rtsp sources to use by given input. The default and recommended value in this pipeline is 8 sources.
4. `--debug` uses `gst-top` to print time and memory consuming elements, saves the results as text and graph.

```
Execution ended after 0:00:23.599335345
Setting pipeline to PAUSED ...
Setting pipeline to READY ...
Setting pipeline to NULL ...
Freeing pipeline ...

** (gst-top-1.0:545889): WARNING **: 15:54:52.438: gst-launch-1.0 (545892) exited with code 256
Most time and memory consuming elements:
ELEMENT          %CPU  %TIME  TIME
hailosend0       23.3   21.4   5.51 s
hailo_pre_infer_q_0  9.4    8.6   2.22 s
comp             8.8    8.1   2.08 s
Full pipeline report saved to pipeline_report.txt and pipeline_graph.svg
```

Open the `pipeline_report.txt` to view the full report showing all elements. Your report should be similar to this:

```
ELEMENT          %CPU  %TIME  TIME
hailosend        78.5   24.4   11.8 s
hailofilter2     69.4   21.6   10.4 s
queue6          19.4    6.0   2.92 s
hailosend        13.4    4.2   2.02 s
hailo_draw0      11.5    3.6   1.73 s
hailo_pre_infer_q_1  9.6    3.0   1.44 s
hailo_pre_infer_q_0  9.3    2.9   1.40 s
hailorecv        8.9    2.8   1.34 s
hailofilter1     7.9    2.5   1.19 s
hailorecv        5.8    1.8   867 ms
videoscale0     5.0    1.6   749 ms
videoconvert4    3.6    1.1   537 ms
videoconvert3    3.5    1.1   527 ms
videoconvert1    3.5    1.1   523 ms
videoconvert0    3.5    1.1   522 ms
videoconvert5    3.5    1.1   520 ms
videoconvert2    3.4    1.1   513 ms
hailofilter0     2.6    0.8   389 ms
videoconvert6    2.5    0.8   369 ms
splitter         2.4    0.8   367 ms
videoconvert7    2.4    0.8   364 ms
fun              2.3    0.7   353 ms
hailomuxer       2.0    0.6   306 ms
hailo_pre_split  1.8    0.6   266 ms
vaapisink0       1.3    0.4   203 ms
vaapipostproc4  1.0    0.3   155 ms
vaapipostproc1  1.0    0.3   155 ms
vaapi decode2    1.0    0.3   150 ms
vaapi postproc3  1.0    0.3   148 ms
vaapi postproc0  1.0    0.3   145 ms
vaapi decode0    1.0    0.3   144 ms
vaapi postproc2  1.0    0.3   143 ms
vaapi postproc5  1.0    0.3   143 ms
vaapi decode1    0.9    0.3   133 ms
vaapi decode4    0.9    0.3   131 ms
vaapi decode5    0.9    0.3   129 ms
vaapi decode3    0.8    0.3   125 ms
vaapi postproc6  0.8    0.2   120 ms
vaapi decode7    0.8    0.2   119 ms
vaapi postproc7  0.7    0.2   110 ms
vaapi decode6    0.7    0.2   106 ms
hailo_preprocess_q_0  0.7    0.2   104 ms
hailo_postprocess0  0.7    0.2   100 ms
sid              0.7    0.2   99.8 ms
```

NOTE: When the debug flag is used and the app is running inside of a docker, exit the app by typing **Ctrl+C** in order to save the results. (Due to docker X11 display communication issues)

Detection and Depth Estimation - networks switch App

Overview:

`detection_and_depth_estimation_networks_switch` demonstrates network switch between two networks: Detection network and Depth estimation network on one video source using one Hailo-8 device. The switch is done every frame, so all frames are inferred by both networks. This is a C++ executable that runs a GStreamer application with extra logic applied through probes

Options

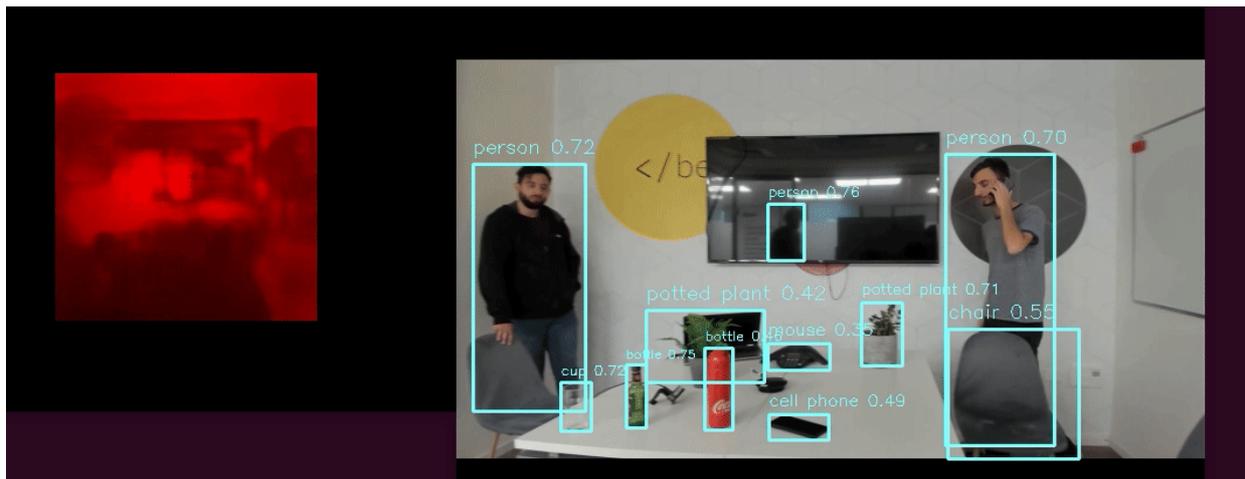
```
./detection_and_depth_estimation_networks_switch [--input FILL-ME --show-fps]
```

- `--input` is an optional flag, a path to the video displayed (default is `instance_segmentation.mp4`).
- `--show-fps` is an optional flag that enables printing FPS on screen.

Run

```
cd
$TAPPAS_WORKSPACE/apps/gstreamer/x86/network_switch/detection_and_depth_estimation_networks_switch
```

The output should look like:



How the application works

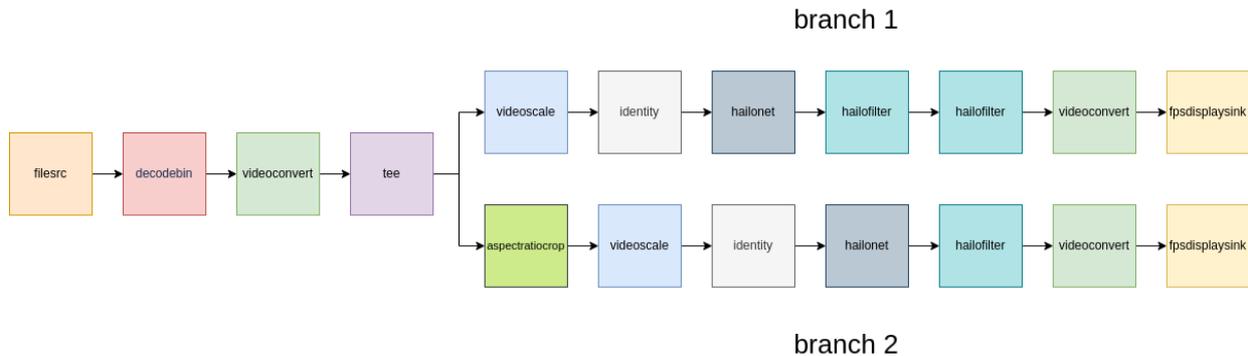
This section explains the network switch. The app builds a gstreamer pipeline (that is explained below) and modifies the `is-active` property of its hailonet elements. This is done by applying buffer-probe callbacks on the input pad (sink pad) of each hailonet element. The callbacks perform network switching by blocking a hailonet element when it is time to switch: turning off one hailonet and turning on the other. Before

turning a hailonet element on, it has to flush the buffers out of the element, this is done by sending the `flush` signal. [read more about hailonet](#)

How the pipeline works

This section is optional and provides a drill-down into the implementation of the `Detection and Depth Estimation networks switch` app with a focus on explaining the `GStreamer` pipeline.

Pipeline diagram



The following elements are the structure of the pipeline:

- `filesrc` reads data from a file in the local file system.
- `decodebin` constructs a decoding sub-pipeline using available decoders and demuxers
- `videoconvert` converts the frame into RGB format.
- `tee` splits data to multiple pads. After this, the pipeline splits into two branches.
 - `branch 1` detection
 - `videoscale` resizes a video frame to the input size of hailonet.
 - `identity` dummy element that passes incoming data through unmodified. In this pipeline it is used for catching EOS events before hailonet 1.
 - `hailonet` Performs the inference on the Hailo-8 device. Requires the `is-active` property that controls whether this element should be active. In case there are two hailonets in a pipeline and each one uses a different hef-file (like in this case) they can't be active at the same time, so when initializing the pipeline this instance of hailonet is set to `is-active=false` and the other one is set to true. This instance of hailonet performs yolov5s network inference for detection. [read more about hailonet](#)
 - `hailofilter` performs the given postprocess, chosen with the `so-path` property. This instance is in charge of yolo post processing.
 - `hailofilter` this instance is in charge of the yolo drawing process.
 - `videoconvert` converts the frame into negotiated format.
 - `fpsdisplaysink` outputs video onto the screen, and displays the current and average framerate.

NOTE: `sync=false` property in `fpsdisplaysink` element disables real-time synchronization within the pipeline - it is mandatory in this case to reach the best performance.

- **branch 2** depth estimation
 - **aspectratio** crops video frames to specified ratio. If it's not included in the pipeline then padding is added to the frames and this behavior is unwanted in case of depth estimation.
 - **videoscale** same as in branch 1
 - **identity** dummy element that passes incoming data through unmodified. In this pipeline it is used for catching EOS events before hailonet 2.
 - **hailonet** this instance of hailonet performs fast-depth network inference for depth estimation. When initializing the pipeline this instance of hailonet is set to is-active=true.
 - **hailofilter** this instance of hailofilter is in charge of depth-estimation post processing and drawing.
 - **videoconvert** same as in branch 1
 - **fpsdisplaysink** same as in branch 1

NOTE: **queue** elements were not presented for clearness. Queue positions can be observed here:



Models

YOLOv5 is a modern object detection architecture that is based on the **YOLOv3** meta-architecture with **CSPNet** backbone. The **YOLOv5** was released on 05/2020 with a very efficient design and State of the art accuracy results on the **COCO benchmark**.

In this pipeline, using a specific variant of the **YOLOv5** architecture - **yo1ov5m** that stands for medium sized networks.

- Pre trained and compiled **yo1ov5m** model stored as .hef file.
- Resolution: 640x640x3
- Full precision accuracy: 41.7mAP
- Dataset: COCO val2017 <https://cocodataset.org/#home>

Enter the git project to read further: <https://github.com/ultralytics/yolov5> Link to the network yaml in Hailo Model Zoo - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/yolov5m.yaml

Centerpose

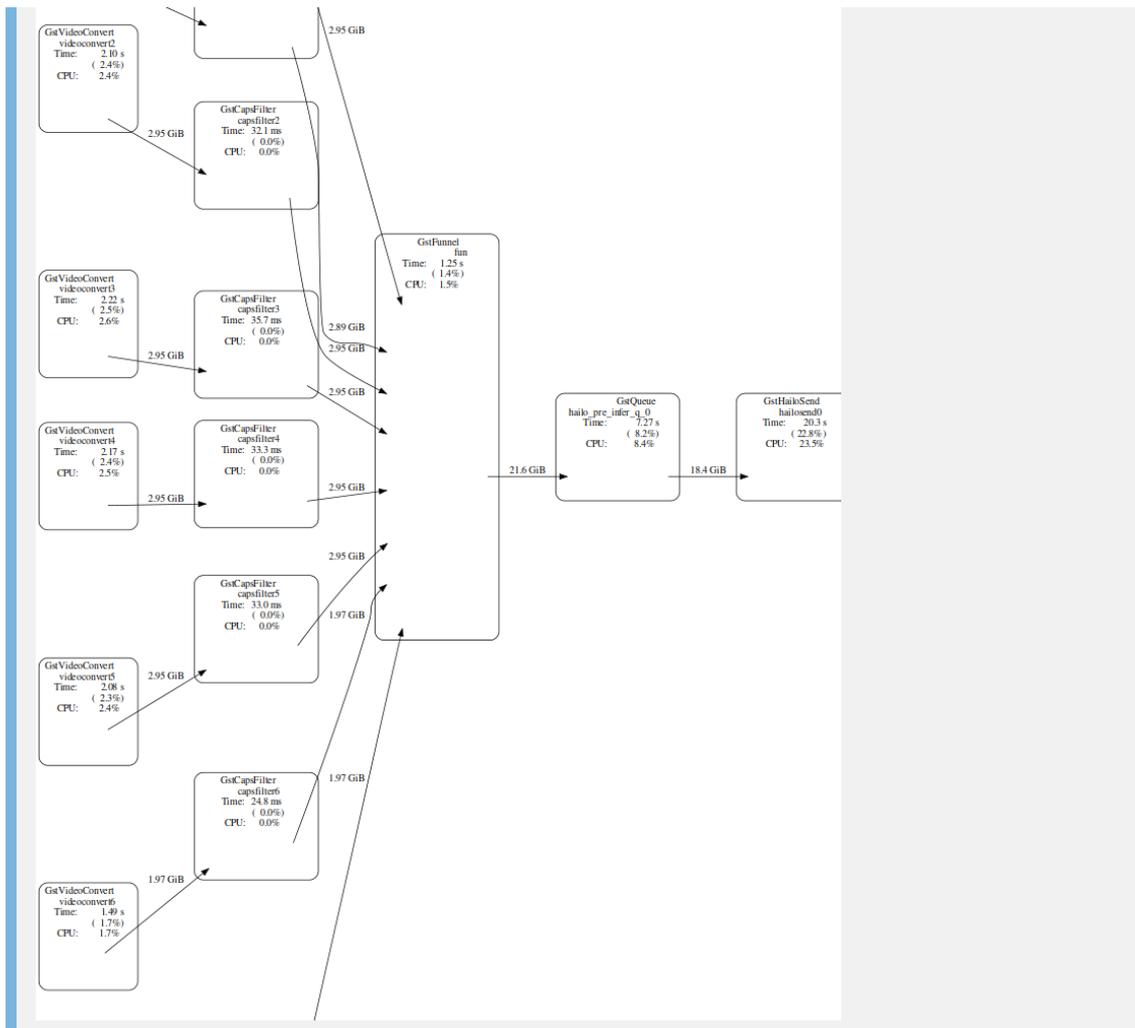
- Hailo trained. Based on centerpose architecture with RegnetX_1.6FG backbone
- Resolution: 640x640x3
- Dataset COCO-Pose

Link to the network yaml in Hailo Model Zoo - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/centerpose_repvgg_a0.yaml

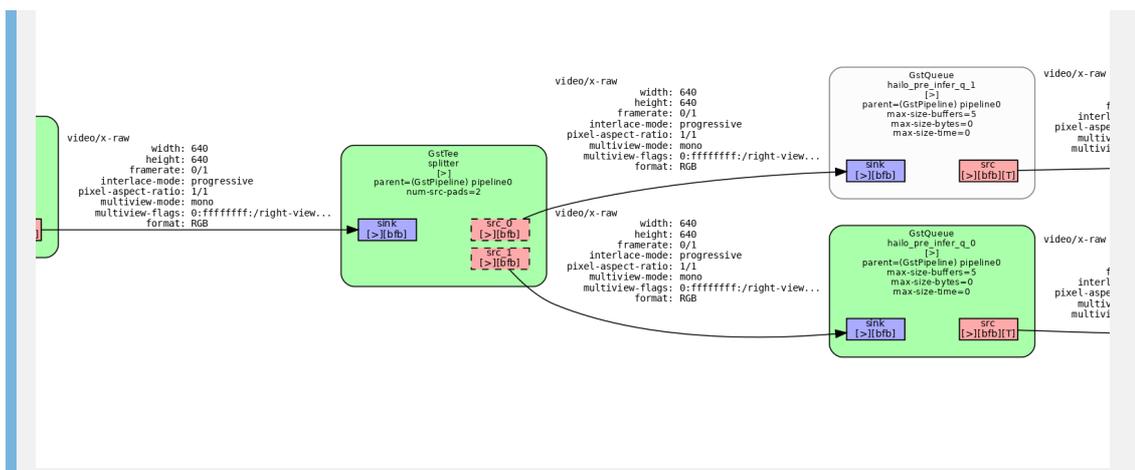
Overview of the pipeline

The following elements are the structure of the pipeline:

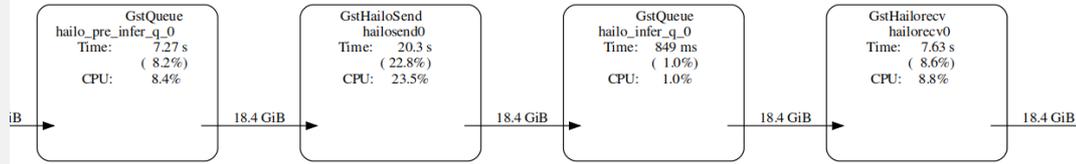
- **funnel** takes multiple input sinks and outputs one source. N-to-1 funnel that attaches a streamid to each stream and can later be used to demux back into separate streams. This lets you queue frames from multiple streams to send to the Hailo device one at a time.



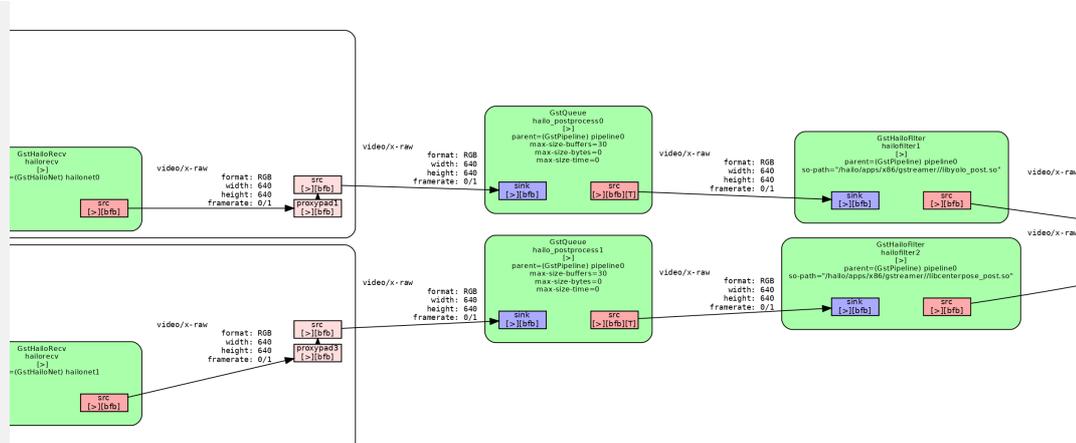
- **tee** duplicates the incoming frame and passes it into two different streams - to perform two different inferences on different chips.



- **hailonet** Performs the inference on the Hailo-8 device - configures the chip with the hef and starts Hailo's inference process - sets streaming mode and sends the buffers into the chip. Requires the following properties: **hef-path** - points to the compiled yolov5m hef, **qos** must be set to false to disable frame drops.



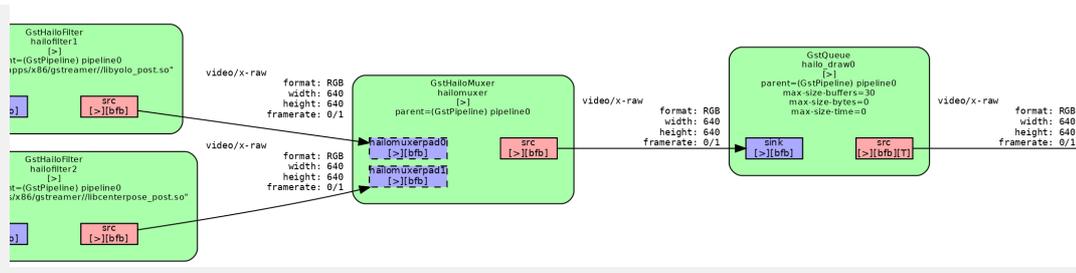
- **hailofilter** performs given postprocess, chosen with the **so-path** property. in this pipeline, two filters performs yolov5m and centerpose in parallel.



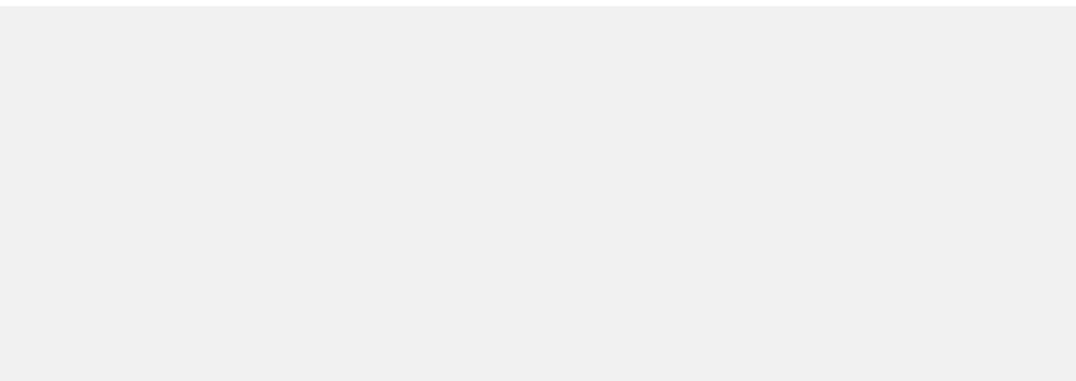
****NOTE****: If multiple hailofilters are present and dependent on each other, then `qos` must be disabled for each.

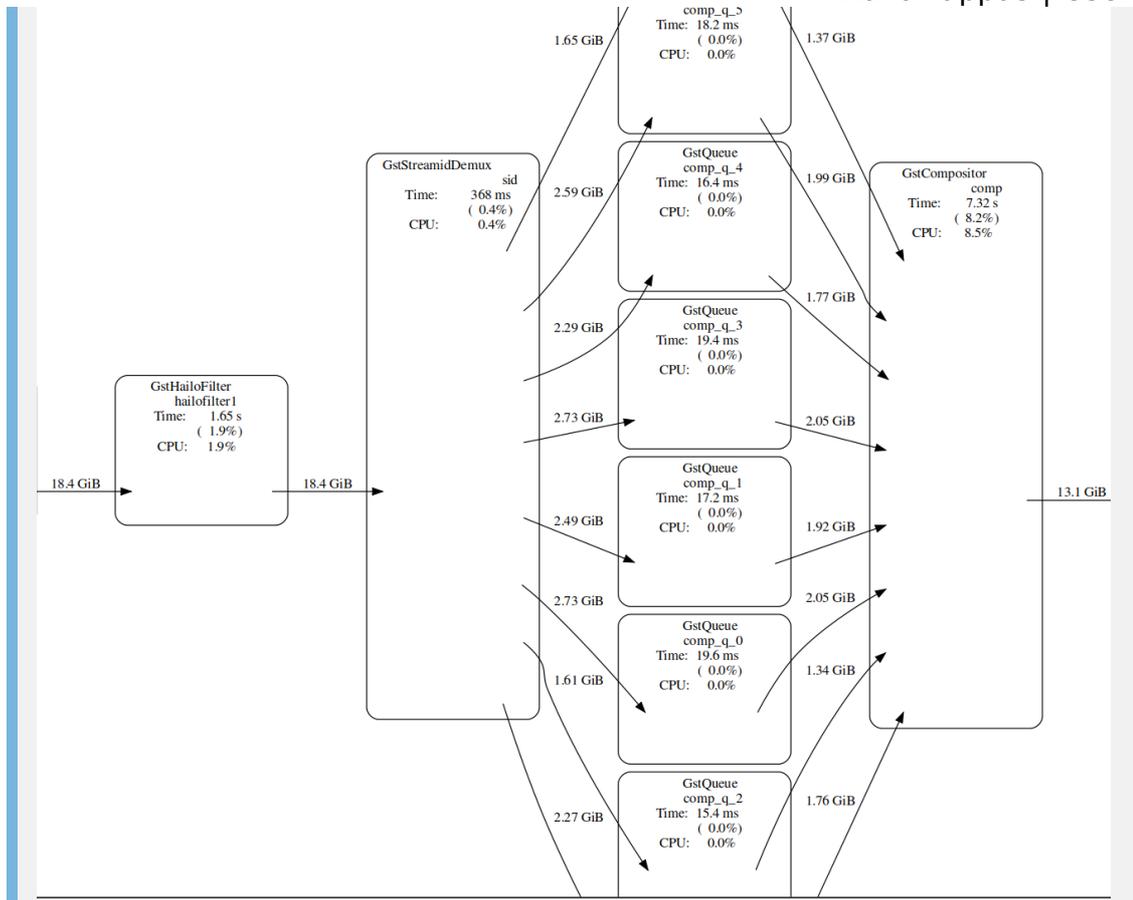
If there is only one hailofilter, then qos may be enabled (although it is still recommended to disable).

- **hailomuxer** muxes 2 similar streams into 1 stream, holding both stream's metadata.



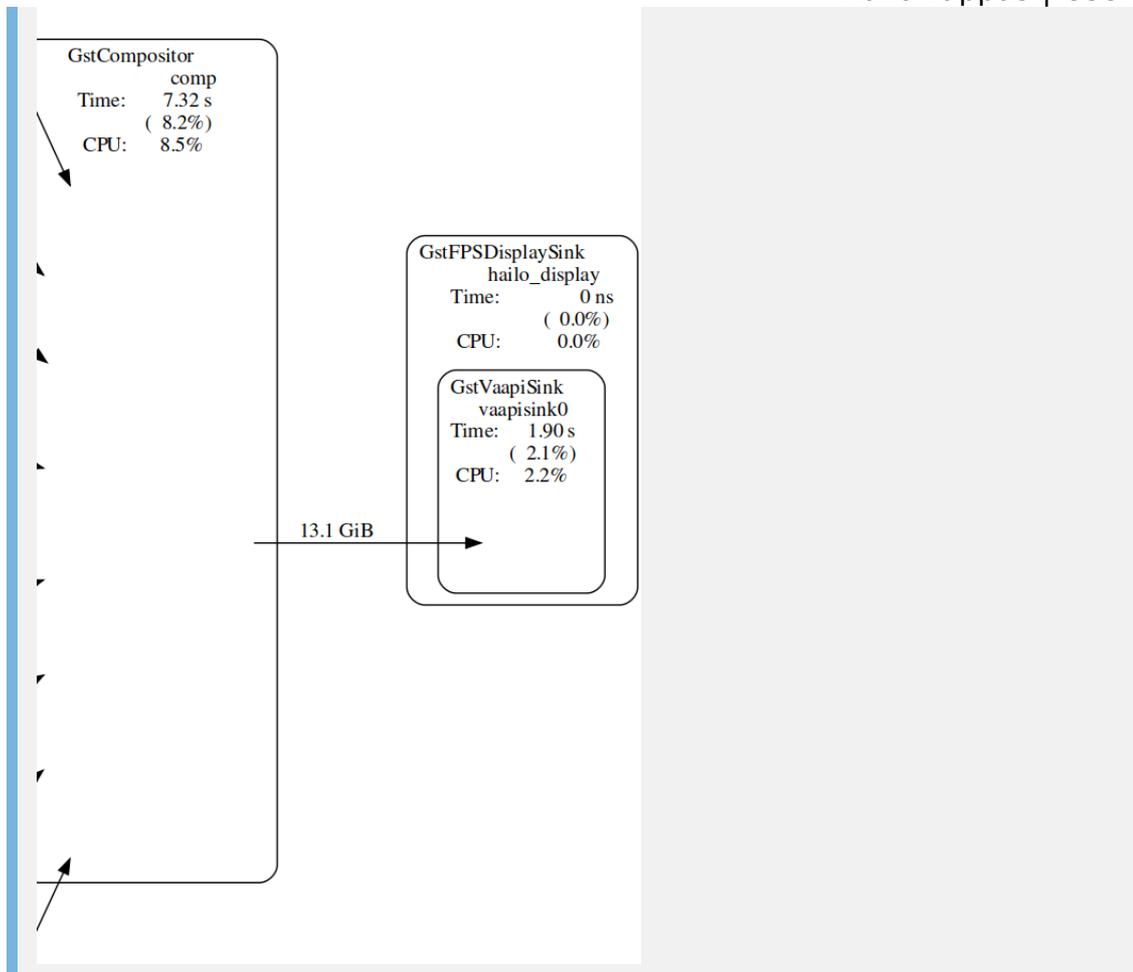
- **streamidmux** a reverse to the funnel. It is a 1-to-N demuxer that splits a serialized stream based on stream id to multiple outputs.
- **compositor** composes pictures from multiple sources. Handy for multi-stream/tiling like applications, as it lets you input many streams and draw them all together as a grid.





- `fpsdisplaysink` outputs video into the screen, and displays the current and average framerate.

NOTE: `sync=false` property in `fpsdisplaysink` element disables real-time synchronization with the pipeline - it is mandatory in this case to reach the best performance.



Python Classification Pipeline

Classification

The purpose of `classification.sh` is to demonstrate classification on one video file source with python post-processing. This is done by running a `single-stream object classification pipeline` on top of GStreamer using the Hailo-8 device.

```
-size-time=0 ! hailofilter so-path=/hailo/apps/x86/gstreamer/libdetection_draw.
so qos=false debug=False ! queue leaky=no max-size-buffers=30 max-size-bytes=0 m
ax-size-time=0 ! videoconvert ! fpsdisplaysink video-sink=ximagesink name=hailo_
display sync=true text-overlay=false
Setting pipeline to PAUSED ...
Pipeline is PREROLLING ...
Redistribute latency...
Redistribute latency...
Pipeline is PREROLLED ...
Setting pipeline to PLAYING ...
New clock: GstSystemClock
^Chandling interrupt.
Interrupt: Stopping pipeline ...
Execution ended after 0:00:17.053081956
Setting pipeline to PAUSED ...
Setting pipeline to READY ...
^C
root@hailo_tappas:/hailo/apps/x86/classification# ^C
root@hailo_tappas:/hailo/apps/x86/classification# ^C
root@hailo_tappas:/hailo/apps/x86/classification# cd ..
root@hailo_tappas:/hailo/apps/x86# cd
```

Options

```
./classification.sh [--input FILL-ME]
```

- `--input` is an optional flag, a path to the video displayed (default is `classification_movie.mp4`).
- `--show-fps` is a flag that prints the pipeline's fps to the screen.
- `--print-gst-launch` is a flag that prints the ready `gst-launch` command without running it.

Supported Networks:

- 'resnet_v1_50' - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/resnet_v1_50.yaml

Run

```
cd $TAPPAS_WORKSPACE/apps/gstreamer/x86/classification
./classification.sh
```

How it works

This section is optional and provides a drill-down into the implementation of the `classification` app with a focus on explaining the `GStreamer` pipeline. This section

uses `resnet_v1_50` as an example network so network input width, height, and hef name are set accordingly.

```
gst-launch-1.0 \
  filesrc location=$input_source ! decodebin ! videoconvert ! \
  videoscale ! video/x-raw, pixel-aspect-ratio=1/1 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailonet hef-path=$hef_path debug=False is-active=true qos=false
! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailopython module=$POSTPROCESS_MODULE qos=false ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailooverlay qos=false ! \
  videoconvert ! \
  fpsdisplaysink video-sink=xvimagesink name=hailo_display
sync=true text-overlay=false ${additonal_parameters}"
```

Let's explain this pipeline section by section:

1. `filesrc location=$video_device ! decodebin ! videoconvert !`

Specifies the location of the video used, then decode and convert to the required format.

2. `videoscale ! video/x-raw,pixel-aspect-ratio=1/1 ! \`

Re-scale the video dimensions to fit the input of the network. In this case it is rescaling the video to 112X112 with the caps negotiation of `hailonet`. `hailonet` Extracts the needed resolution from the HEF file during the caps negotiation, and makes sure that the needed resolution is passed from previous elements.

3. `queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \`

Before sending the frames into `hailonet` element set a queue so no frames are lost (Read more about queues [here](#))

4. `hailonet hef-path=$hef_path debug=False is-active=true qos=false
! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \`

Performs the inference on the Hailo-8 device.

5.

```
hailopython so-path=$POSTPROCESS_MODULE qos=false debug=False !
\
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
```

Performs a given post-process, in this case, performs `resnet_v1_50` classification post-process, which is mainly doing top1 on the inference output.

6.

```
hailooverlay qos=False ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
```

Performs drawing on the original image, in that case, performs drawing the top1 class name over the image.

7.

```
videoconvert ! \
fpsdisplaysink video-sink=xvimagesink name=hailo_display
sync=true text-overlay=false ${additional_parameters}"
```

Apply the final convert to let GStreamer utilize the format required by the `fpsdisplaysink` element.

Century Pipeline

Overview:

`century.sh` demonstrates detection on one video file source over 6 different Hailo-8 devices. This pipeline run's the detection network YoloX.

Options

```
./century.sh [--input FILL-ME]
```

- `--input` is an optional flag, a path to the video displayed (default is `detection.mp4`).
- `--show-fps` is an optional flag that enables printing FPS on screen.
- `--print-gst-launch` is a flag that prints the ready `gst-launch` command without running it

Supported Networks:

- 'yolox_l' - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/yolox_l_leaky.yaml

Run

```
cd $TAPPAS_WORKSPACE/apps/gstreamer/x86/century
./century.sh
```

The output should look like:



How it works

This section is optional and provides a drill-down into the implementation of the **century** app with a focus on explaining the **GStreamer** pipeline. This section uses **yoloX** as an example network so network input width, height, and hef name are set accordingly.

```
gst-launch-1.0 \
  filesrc location=$video_device ! decodebin ! videoconvert ! \
  videoscale ! video/x-raw, pixel-aspect-ratio=1/1 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailonet hef-path=$hef_path device-count=$device_count is-
active=true ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailofilter function-name=yoloX so-path=$POSTPROCESS_S0 qos=false
! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailooverlay qos=false !
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  videoconvert ! \
  fpsdisplaysink video-sink=xvimagesink name=hailo_display
sync=false text-overlay=false ${additional_parameters}
```

Let's explain this pipeline section by section:

1. `filesrc location=$video_device ! decodebin ! videoconvert !`

Specifies the location of the video used, then decodes and converts to the required format.

2. `videoscale ! video/x-raw,pixel-aspect-ratio=1/1 ! \`

Re-scale the video dimensions to fit the input of the network. In this case it is rescaling the video to 640x640 with the caps negotiation of `hailonet`.

3. `queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \`

Before sending the frames into the `hailonet` element, set a queue so no frames are lost (Read more about queues [here](#))

4. `hailonet hef-path=$hef_path device-count=$device_count is-active=true ! \`
`queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \`

Performs the inference on the Hailo-8 device via `$device_count` devices, which is set to 4 in this app.

5. `hailofilter2 function-name=yolox so-path=$POSTPROCESS_S0 qos=false ! \`
`queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \`

`hailofilter` performs a given post-process. In this case the performs the `Yolox` post-process.

6. `hailooverlay qos=false ! \`
`queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \`

`hailooverlay` draws the post-processed boxes on the frame.

7. `videoconvert ! \`
`queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \`
`fpsdisplaysink video-sink=xvimagesink name=hailo_display sync=false text-overlay=false ${additional_parameters}`

Apply the final convert to let GStreamer utilize the format required by the `fpsdisplaysink` element

GST-launch based Arm applications

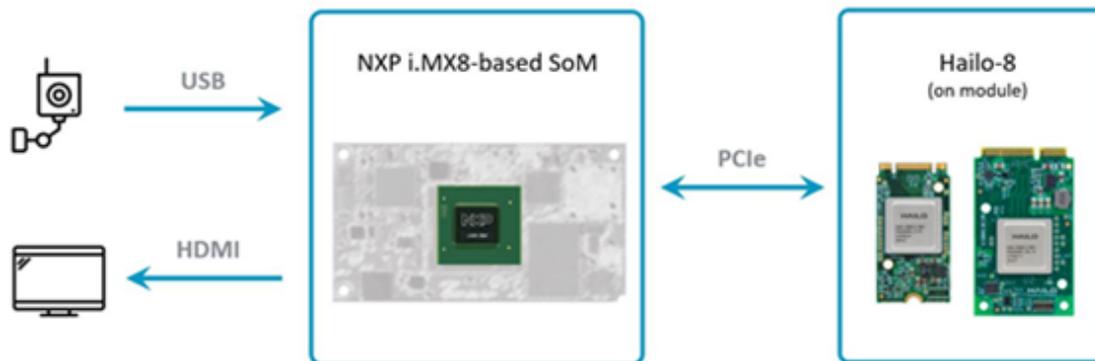
1. [Detection](#) - single-stream object detection pipeline on top of GStreamer using the Hailo-8 device.

Detection Pipeline Arm

Overview

Our requirement from this pipeline is a real-time high-accuracy object detection to run on a single video stream using an embedded host. The required input video resolution was HD (high definition, 720p).

The chosen platform for this project is based on NXP's i.MX8M Arm processor. The Hailo-8™ AI processor is connected to it as an AI accelerator.



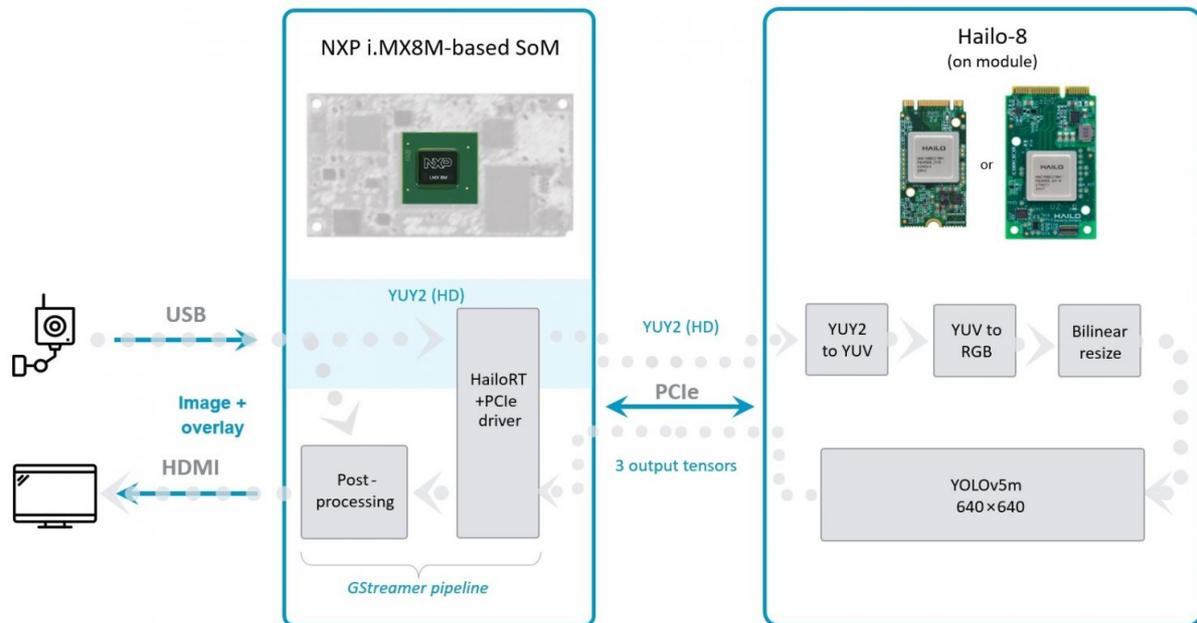
Drill Down

Although the i.MX8M is a capable host, processing and decoding real-time HD video is bound to utilize a lot of the CPU's resources, which may eventually reduce performance. To solve this problem, most of the vision pre-processing pipeline has been offloaded to the Hailo-8 device in our application.

The camera sends the raw video stream, encoded in YUV color format using the YUY2 layout. The data passes through Hailo's runtime software library, called HailoRT, and through Hailo's PCIe driver. The data's format is kept unmodified, and it is sent to the Hailo-8 device as is.

Hailo-8's NN core handles the data pre-processing, which includes decoding the YUY2 scheme, converting from the YUV color space to RGB and, finally, resizing the frames into the resolution expected by the deep learning detection model.

The Hailo Dataflow Compiler supports adding these pre-processing stages to any model when compiling it. In this case, they are added before the YOLOv5m detection model.



Options

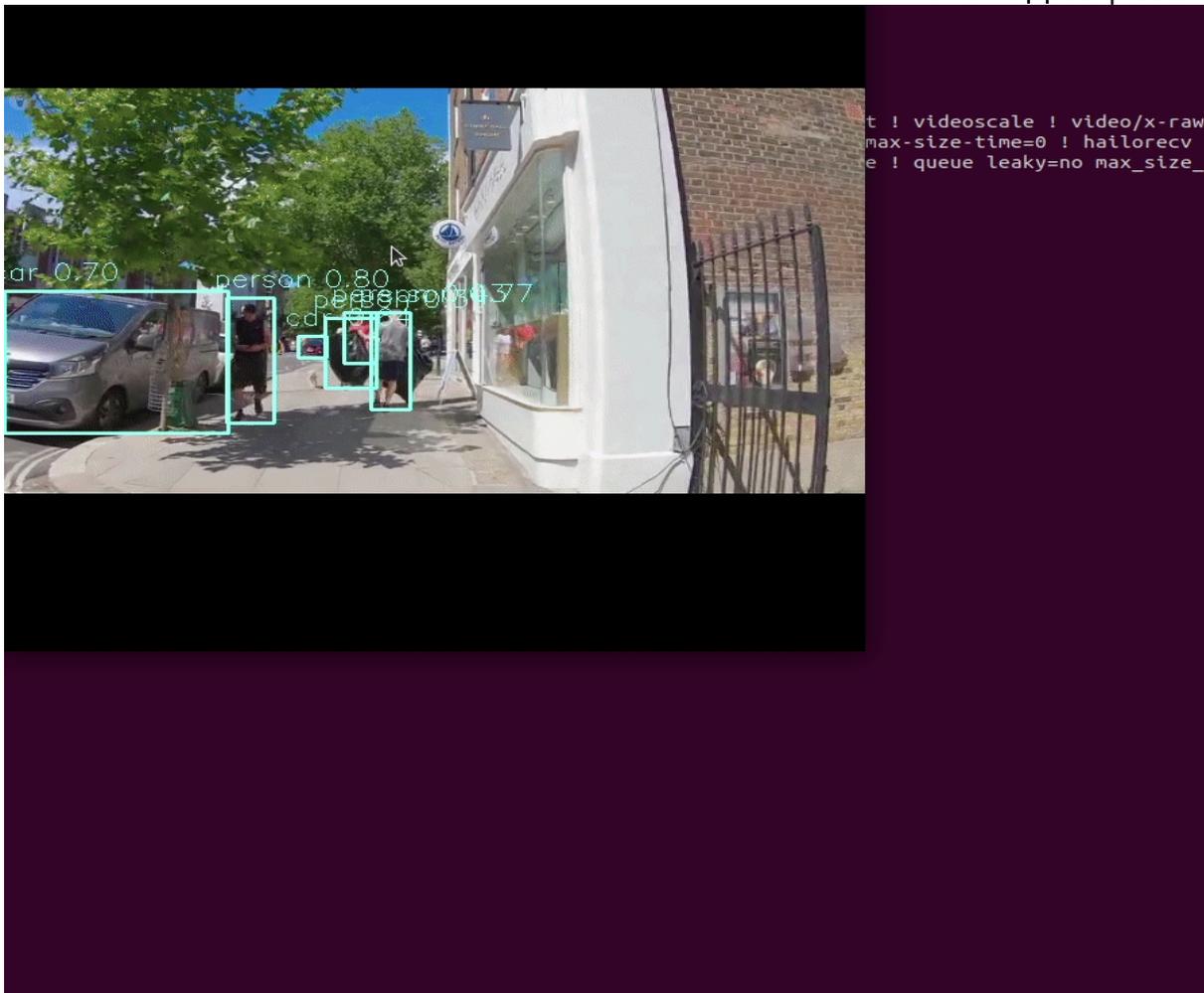
```
./detection.sh [--input FILL-ME]
```

- `--input` is an optional flag, path to the video camera used (default is `/dev/video2`).
- `--show-fps` is an optional flag that enables printing FPS on screen.
- `--print-gst-launch` is a flag that prints the ready gst-launch command without running it"

Run

```
cd $TAPPAS_WORKSPACE/arm/apps/detection
./detection.sh
```

The output should look like:



How it works

This section is optional and provides a drill-down into the implementation of the **detection** app with a focus on explaining the **GStreamer** pipeline. This section uses **yolov5** as an example network so network input width, height, and hef name are set accordingly.

```
gst-launch-1.0 \
  v4l2src device=$input_source ! video/x-
raw,format=YUY2,width=1280,height=720,framerate=30/1 ! \
  queue leaky=downstream max-size-buffers=5 max-size-bytes=0 max-
size-time=0 ! \
  hailonet hef-path=$hef_path debug=False is-active=true qos=false
batch-size=1 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailofilter function-name=$network_name so-path=$postprocess_so
qos=false debug=False ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailofilter so-path=$draw_so qos=false debug=False ! \
  queue leaky=downstream max-size-buffers=5 max-size-bytes=0 max-
size-time=0 ! \
  videoconvert ! \
  fpsdisplaysink video-sink=xvimagesink name=hailo_display
sync=false text-overlay=false ${additional_parameters}
```

Let's explain this pipeline section by section:

1.

```
v4l2src device=$input_source ! video/x-raw,format=YUY2,width=1280,height=720,framerate=30/1
```

Specifies the path of the camera, specify the required format and resolution.

2.

```
queue leaky=downstream max-size-buffers=5 max-size-bytes=0 max-size-time=0 ! \
```

Before sending the frames into the `hailonet` element, set a queue to leaky (Read more about queues [here](#))

3.

```
hailonet hef-path=$hef_path debug=False is-active=true qos=false batch-size=1 ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \
```

Performs the inference on the Hailo-8 device.

4.

```
hailofilter function-name=yolov5 so-path=$POSTPROCESS_S0 qos=false debug=False ! \
queue name=hailo_draw0 leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \
hailofilter so-path=$DRAW_POSTPROCESS_S0 qos=false debug=False ! \
queue leaky=downstream max-size-buffers=5 max-size-bytes=0 max-size-time=0 ! \
```

Each `hailofilter` performs a given post-process. In this case the first performs the `Yolov5m` post-process and the second performs box drawing. Then set a leaky queue to let the sink drop frames.

5.

```
videoconvert ! \
fpsdisplaysink video-sink=xvimagesink name=hailo_display sync=true text-overlay=false ${additional_parameters}
```

Apply the final convert to let GStreamer utilize the format required by the `fpsdisplaysink` element

Links

- [hailofilter](#)
- [Blog post about this setup](#)

GST-launch based Raspberry Pi applications

GST-Launch based applications on Raspberry Pi

1. [Sanity Pipeline](#) - Helps you verify that all the required components are installed correctly
2. [Detection](#) - single-stream object detection pipeline on top of GStreamer using the Hailo-8 device.
3. [Depth Estimation](#) - single-stream depth estimation pipeline on top of GStreamer using the Hailo-8 device.
4. [Multinetworks parallel](#) - single-stream multi-networks pipeline on top of GStreamer using the Hailo-8 device.
5. [Pose Estimation](#) - Human pose estimation using [centerpose](#) network.
6. [Face Detection](#) - Face detection application.
7. [Classification](#) - Classification app using [resnet_v1_50](#) network.

Sanity pipeline Raspberry Pi

Overview

Sanity apps purpose is to help you verify that all the required components have been installed successfully.

First of all, you would need to run `sanity_gstreamer.sh` and make sure that the image presented looks like the one that would be presented later.

Sanity GStreamer

This app should launch first.

NOTE: Open the source code in your preferred editor to see how simple this app is.

In order to run the app just `cd` to the `sanity_pipeline` directory and launch the app

```
cd $TAPPAS_WORKSPACE/apps/gstreamer/raspberrypi/sanity_pipeline
./sanity_gstreamer.sh
```

The output should look like:



If the output is similar to the image shown above, you are good to go to the next verification phase!

Detection Pipeline Raspberry Pi

Overview:

`detection.sh` demonstrates detection on one video file source and verifies Hailo's configuration. This is done by running a `single-stream object detection pipeline` on top of GStreamer using the Hailo-8 device.

Options

```
./detection.sh [--input FILL-ME]
```

- `--input` is an optional flag, a path to the video file displayed (default is `detection.mp4`).
- `--network` is a flag that sets which network to use. choose from [`yolov5`, `mobilenet_ssd`], default is `yolov5`. this will set the hef file to use, the `hailofilter` function to use and the scales of the frame to match the width and height input dimensions of the network.
- `--show-fps` is an optional flag that enables printing FPS on screen.
- `--print-gst-launch` is a flag that prints the ready `gst-launch` command without running it
- `--print-device-stats` Print the power and temperature measured

Supported Networks:

- 'yolov5' - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/yolov5m.yaml
- 'mobilenet_ssd' - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/ssd_mobilenet_v1.yaml

Run

```
cd $TAPPAS_WORKSPACE/apps/gstreamer/raspberrypi/detection
./detection.sh
```

The output should look like:

```

sanity_pipeline $
sanity_pipeline $
sanity_pipeline $
sanity_pipeline $
sanity_pipeline $

```

How it works

This section is optional and provides a drill-down into the implementation of the **detection** app with a focus on explaining the **GStreamer** pipeline. This section uses **yolov5** as an example network so network input width, height, and hef name are set accordingly.

```

gst-launch-1.0 \
  gst-launch-1.0 ${stats_element} \
  filesrc location=$input_source name=src_0 ! qtdemux ! h264parse !
  avdec_h264 ! \
  videoscale n-threads=8 ! video/x-raw, pixel-aspect-ratio=1/1 !
  videoconvert n-threads=8 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
  time=0 ! \
  hailonet hef-path=$hef_path device-id=$hailo_bus_id debug=False
  is-active=true qos=false batch-size=$batch_size ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
  time=0 ! \
  hailofilter2 function-name=$network_name so-path=$postprocess_so
  qos=false ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
  time=0 ! \
  hailooverlay ! \
  videoconvert n-threads=8 ! \
  fpsdisplaysink video-sink=ximagesink name=hailo_display
  sync=$sync_pipeline text-overlay=false ${additional_parameters}"

```

Let's explain this pipeline section by section:

1.

```
filesrc location=$input_source name=src_0 ! qtdemux ! h264parse
! avdec_h264 ! \
```

Specifies the location of the video used, then decodes it.

2.

```
filesrc location=$input_source name=src_0 ! qtdemux ! h264parse
! avdec_h264 ! \
```

Specifies the location of the video used, then decodes it.

3.

```
videoscale n-threads=8 ! video/x-raw, pixel-aspect-ratio=1/1 !
videoconvert n-threads=8 ! \
```

Re-scale the video dimensions to fit the input of the network. In this case it is rescaling the video to 640x640 with the caps negotiation of `hailonet`. Then convert it to the required format.

4.

```
queue ! \
```

Before sending the frames into the `hailonet` element, set a queue so no frames are lost (Read more about queues [here](#))

5.

```
hailonet hef-path=$hef_path device-id=$hailo_bus_id debug=False
is-active=true qos=false batch-size=$batch_size ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
```

Performs the inference on the Hailo-8 device.

6.

```
hailofilter2 function-name=$network_name so-path=$postprocess_so
qos=false ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
```

Each `hailofilter` performs a given post-process. In this case performs the `Yolov5m` post-process.

7.

```
hailooverlay ! \
```

Performs the drawing.

8.

```
videoconvert n-threads=8 ! \
fpsdisplaysink video-sink=ximagesink name=hailo_display
```

```
sync=$sync_pipeline text-overlay=false ${additional_parameters}"
```

Apply the final convert to let GStreamer utilize the format required by the `fpsdisplaysink` element

NOTE: Additional details about the pipeline provided in further examples


```

gst-launch-1.0 \
  filesrc location=$input_source name=src_0 ! qtdemux ! h264parse !
  avdec_h264 ! queue ! videoconvert n-threads=8 ! queue ! \
  tee name=t ! queue leaky=no max-size-buffers=30 max-size-bytes=0
  max-size-time=0 ! \
  aspectratiocrop aspect-ratio=1/1 ! queue ! videoscale ! queue ! \
  hailonet hef-path=$hef_path device-id=$hailo_bus_id debug=False
  is-active=true qos=false batch-size=1 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
  time=0 ! \
  hailofilter so-path=$draw_so qos=false debug=False ! videoconvert
  n-threads=8 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
  time=0 ! \
  videoconvert ! fpsdisplaysink video-sink=ximagesink
  name=hailo_display sync=false text-overlay=false \
  t. ! queue leaky=no max-size-buffers=30 max-size-bytes=0 max-
  size-time=0 ! \
  videoscale ! video/x-raw, width=300, height=300 ! queue !
  videoconvert n-threads=8 ! \
  ximagesink sync=false ${additional_parameters}

```

Let's explain this pipeline section by section:

1. `filesrc location=$input_source name=src_0 ! qtdemux ! h264parse ! avdec_h264 ! queue ! videoconvert n-threads=8 ! queue ! \`

Specifies the location of the video used, then decodes and converts to the required format using 8 threads for acceleration.

2. `tee name=t`

declare a tee that splits the pipeline into two branches.

3. `queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \ aspectratiocrop aspect-ratio=1/1 ! queue ! videoscale ! queue ! \`

The beginning of the first split of the tee. The network used expects no borders, so a crop mechanism is needed.

Re-scales the video dimensions to fit the input of the network. In this case it is cropping the video and rescaling the video to 224x224 with the caps negotiation of `hailonet`.

4. `hailonet hef-path=$hef_path device-id=$hailo_bus_id debug=False is-active=true qos=false batch-size=1 ! \ queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \`

Performs the inference on the Hailo-8 device.

5.

```
hailofilter so-path=$draw_so qos=false debug=False !
videoconvert n-threads=8 ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
```

Performs a given draw-process, in this case, performs `fast_depth` depth estimation drawing per pixel.

6.

```
videoconvert n-threads=8 ! fpsdisplaysink video-sink=ximagesink
name=hailo_display sync=false text-overlay=false \
```

Apply the final convert to let GStreamer utilize the format required by the `fpsdisplaysink` element

7.

```
t. ! queue leaky=no max-size-buffers=30 max-size-bytes=0 max-
size-time=0 ! \
videoscale ! video/x-raw, width=300, height=300 ! queue
```

The beginning of the second split of the tee. Re-scales the video dimensions.

8.

```
videoconvert n-threads=8 ! \
ximagesink sync=false ${additional_parameters}
```

Apply the final convert to let GStreamer utilize the format required by the `ximagesink` element

NOTE: Additional details about the pipeline provided in further examples

Detection and Depth Estimation Pipeline Raspberry Pi

Detection and Depth Estimation

`detection_and_depth_estimation.sh` demonstrates depth estimation and detection on one video file source. This is done by running two streams on top of GStreamer using one Hailo-8 device with using two `hailonet` elements.

Options

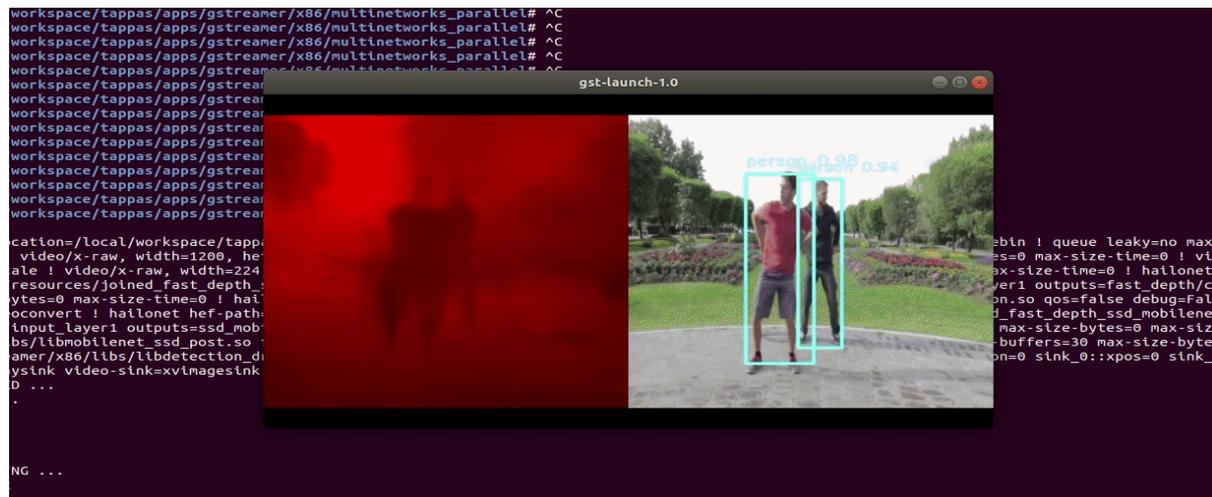
```
./detection_and_depth_estimation.sh [--video-src FILL-ME]
```

- `-i --input` is an optional flag, a path to the video displayed.
- `--print-gst-launch` is a flag that prints the ready gst-launch command without running it
- `--show-fps` is an optional flag that enables printing FPS on screen

Run

```
cd
$TAPPAS_WORKSPACE/apps/gstreamer/raspberrypi/multinetworks_parallel/
./detection_and_depth_estimation.sh
```

The output should look like:



Model

- `fast_depth` in resolution of 224X224X3.
- `mobilenet_ssd` in resolution of 300X300X3.

How it works

This section is optional and provides a drill-down into the implementation of the app with a focus on explaining the GStreamer pipeline. This section uses `fast_depth` as an example network so network input width, height, hef name, are set accordingly.

```

gst-launch-1.0 \
  $source_element ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  videoconvert n-threads=8 !
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  tee name=t ! \
  aspectratiocrop aspect-ratio=1/1 ! \
  queue ! videoscale n-threads=8 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailonet hef-path=$hef_path device-id=$hailo_bus_id debug=False
is-active=true net-name=$depth_estimation_net_name qos=false batch-
size=1 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailofilter so-path=$depth_estimation_draw_so qos=false
debug=False ! videoconvert n-threads=8 ! \
  fpsdisplaysink video-sink=ximagesink name=hailo_display
sync=false text-overlay=false \
  t. ! \
  videoscale n-threads=8 ! queue ! \
  hailonet hef-path=$hef_path device-id=$hailo_bus_id debug=False
is-active=true net-name=$detection_net_name qos=false batch-size=1 !
\
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailofilter2 so-path=$detection_post_so function-
name=mobilenet_ssd_merged qos=false ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailooverlay ! videoconvert n-threads=8 ! \
  fpsdisplaysink video-sink=ximagesink name=hailo_display2
sync=false text-overlay=false ${additional_parameters} "

```

Let's explain this pipeline section by section:

1. `filesrc location=$video_source name=src_0 ! qtdemux ! h264parse ! avdec_h264 \`

Specifies the location of the video used and then decodes

2. `queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \`

Before sending the frames into `hailonet` element, set a queue so no frames are lost (Read more about queues [here]

(<https://gstreamer.freedesktop.org/documentation/coreelements/queue.html?gi-language=c>)

```
3. videoconvert n-threads=8 !
```

converts to the required format.

```
4. tee name=t !
```

Split into two threads - one for mobilenet_ssd and the other for fast_depth.

```
5. aspectratiocrop aspect-ratio=1/1 ! videoscale n-threads=8 ! \
```

Re-scales the video dimensions to fit the input of the network using 8 threads for acceleration. In this case it is cropping the video and rescaling the video to 224x224 with the caps negotiation of **hailonet**.

```
6. hailonet hef-path=$hef_path device-id=$hailo_bus_id debug=False
  is-active=true net-name=$depth_estimation_net_name qos=false
  batch-size=1 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
  time=0 ! \
```

Performs the inference on the Hailo-8 device.

NOTE: We pre define the input and the output layers of each network, giving the net-name argument.

```
6. hailofilter so-path=$DRAW_POSTPROCESS_S0 qos=false debug=False !
  \
```

Performs a given draw-process, in this case, performs **fast_depth** depth estimation drawing per pixel.

```
7. videoconvert n-threads=8 ! \
  fpsdisplaysink video-sink=ximagesink name=hailo_display
  sync=false text-overlay=false \
```

Apply the final convert to let GStreamer utilize the format required by the **fpsdisplaysink** element

```
8. t. ! \
```

beginning of another split of the tee

```
9. videoscale n-threads=8 !
```

Re-scales the video dimensions to fit the input of the network using 8 threads for acceleration.

10.

```
hailonet hef-path=$hef_path device-id=$hailo_bus_id debug=False
is-active=true net-name=$detection_net_name qos=false batch-
size=1 ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
```

Performs the inference on the Hailo-8 device.

11.

```
hailofilter2 so-path=$detection_post_so function-
name=mobilenet_ssd_merged qos=false ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
```

Performs a given post-process, in this case - detection post process.

12.

```
hailooverlay ! \
```

Performs a draw process, based on the meta data of the buffers. this is a newer api (comparing to using hailofilter for drawing).

13.

```
videoconvert n-threads=8 ! \
fpsdisplaysink video-sink=ximagesink name=hailo_display2
sync=false text-overlay=false ${additonal_parameters} ! \
```

Apply the final convert to let GStreamer utilize the format required by the `fpsdisplaysink` element

NOTE: Additional details about the pipeline provided in further examples

Pose Estimation Pipeline Raspberry Pi

Overview:

`hailo_pose_estimation.sh` demonstrates human pose estimation on one video file source and verifies Hailo's configuration. This is done by running a `single-stream pose estimation pipeline` on top of GStreamer using the Hailo-8 device.

Options

```
./hailo_pose_estimation.sh [--input FILL-ME]
```

- `--input` is an optional flag, a path to the video displayed (default is `detection.mp4`).
- `--show-fps` is an optional flag that enables printing FPS on screen.
- `--network` Set network to use. choose from [`centerpose`, `centerpose_416`], default is `centerpose`
- `--print-gst-launch` is a flag that prints the ready `gst-launch` command without running it"

Run

```
cd $TAPPAS_WORKSPACE/apps/gstreamer/raspberrypi/pose_estimation  
./hailo_pose_estimation.sh
```

The output should look like:



How it works

This section is optional and provides a drill-down into the implementation of the `pose_estimation` app with a focus on explaining the `GStreamer` pipeline. This section

uses `centerpose_regnetx_1.6gf_fpn` as an example network so network input width, height, and hef name are set accordingly.

```
gst-launch-1.0 \
  filesrc location=$input_source name=src_0 ! qtdemux ! h264parse !
  avdec_h264 ! \
  videoscale n-threads=8 ! video/x-raw, pixel-aspect-ratio=1/1 !
  videoconvert n-threads=8 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
  time=0 ! \
  hailonet hef-path=$hef_path device-id=$hailo_bus_id debug=False
  is-active=true qos=false batch-size=1 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
  time=0 ! \
  hailofilter so-path=$postprocess_so qos=false debug=False
  function-name=$network_name ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
  time=0 ! \
  hailofilter so-path=$draw_so qos=false debug=False ! \
  videoconvert n-threads=8 ! \
  fpsdisplaysink video-sink=ximagesink name=hailo_display
  sync=$sync_pipeline text-overlay=false ${additional_parameters}
```

Let's explain this pipeline section by section:

1. `filesrc location=$input_source name=src_0 ! qtdemux ! h264parse ! avdec_h264 ! \ videoscale n-threads=8 ! video/x-raw, pixel-aspect-ratio=1/1 ! videoconvert n-threads=8 ! \`

Specifies the location of the video used, then decodes the data. Re-scale the video dimensions to fit the input of the network, In this case it is rescaling the video to 640x640 with the caps negotiation of `hailonet`. Converts to the required format using 8 threads for acceleration.

2. `queue leaky=no max-size-buffers=30 max-size-bytes=0max-size-time=0 ! \`

Before sending the frames into the `hailonet` element, set a queue so no frames are lost (Read more about queues [here](#))

3. `hailonet hef-path=$hef_path device-id=$hailo_bus_id debug=False is-active=true qos=false batch-size=1 ! \ queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! \`

Performs the inference on the Hailo-8 device.

4.

```
hailofilter so-path=$postprocess_so qos=false debug=False
function-name=$network_name ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
hailofilter so-path=$draw_so qos=false debug=False ! \
```

Each `hailofilter` performs a given post-process. In this case the first performs the `centerpose` post-process and the second performs box and skeleton drawing.

5.

```
videoconvert n-threads=8 ! \
fpsdisplaysink video-sink=ximagesink name=hailo_display
sync=$sync_pipeline text-overlay=false ${additional_parameters}
```

Apply the final convert to let GStreamer utilize the format required by the `fpsdisplaysink` element

NOTE: Additional details about the pipeline provided in further examples

Face Detection Pipeline Raspberry Pi

Overview:

The purpose of `face_detection.sh` is to demonstrate face detection on one video file source and to verify Hailo's configuration. This is done by running a `single-stream face detection pipeline` on top of GStreamer using the Hailo-8 device.

Options

```
/face_detection.sh
```

- `--input` is an optional flag, a path to the video displayed (default is `face_detection.mp4`).
- `--show-fps` is an optional flag that enables printing FPS on screen.
- `--print-gst-launch` is a flag that prints the ready `gst-launch` command without running it"

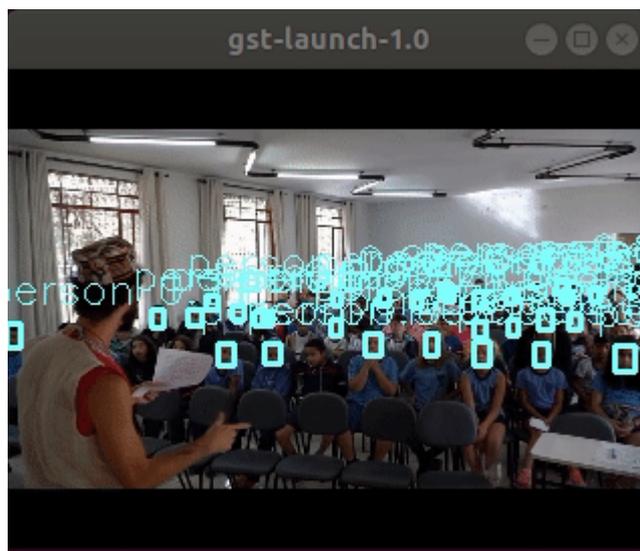
Supported Networks

- 'liteface' - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/lightface_slim.yaml

Run

```
cd $TAPPAS_WORKSPACE/apps/gstreamer/raspberrypi/face_detection/  
./face_detection.sh
```

The output should look like:



How it works

This section is optional and provides a drill-down into the implementation of the `face detection` app with a focus on explaining the `GStreamer` pipeline. This section uses

`lightface_slim` as an example network so network input width, height, hef name, are set accordingly.

```
gst-launch-1.0 \
  filesrc location=$input_source name=src_0 ! qtdemux ! h264parse !
  avdec_h264 ! videoconvert n-threads=8 ! tee name=t hailomuxer
  name=mux \
  t. ! queue leaky=no max-size-buffers=30 max-size-bytes=0 max-
  size-time=0 ! mux. \
  t. ! videoscale n-threads=8 ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
  time=0 ! \
  hailonet hef-path=$hef_path device-id=$hailo_bus_id debug=False
  is-active=true qos=false ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
  time=0 ! \
  hailofilter2 function-name=$network_name so-path=$postprocess_so
  qos=false ! mux. \
  mux. ! queue leaky=no max-size-buffers=30 max-size-bytes=0 max-
  size-time=0 ! \
  hailooverlay ! queue leaky=no max-size-buffers=30 max-size-
  bytes=0 max-size-time=0 ! \
  videoconvert n-threads=8 ! \
  fpsdisplaysink video-sink=ximagesink name=hailo_display
  sync=$sync_pipeline text-overlay=false ${additional_parameters}
```

Let's explain this pipeline section by section:

1. `filesrc location=$input_source name=src_0 ! qtdemux ! h264parse ! avdec_h264 ! videoconvert n-threads=8`

Specifying the location of the video used, then decode and convert to the required format using 8 threads for acceleration.

2. `tee name=t`

splitting to two branches of the pipeline

3. `hailomuxer name=mux`

declaration of hailomuxer element

4. `t. ! queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! mux. \`

a branch of the tee, passing the original frame to the muxer without re-scale

5.

```
t. ! videoscale n-threads=8 ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
```

Another branch of the tee that will perform the inference. Re-scale the video dimensions to fit the input of the network. In this case it is rescaling the video to 320x240 with the caps negotiation of `hailonet`.

6.

```
hailonet hef-path=$hef_path device-id=$hailo_bus_id
debug=False is-active=true qos=false ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-
size-time=0 ! \
```

Performs the inference on the Hailo-8 device.

NOTE: `qos` must be disabled for `hailonet` since dropping frames may cause these elements to run out of alignment.

7.

```
hailofilter2 function-name=$network_name so-path=$postprocess_so
qos=false ! mux. \
mux.
```

Each `hailofilter` performs a given post-process. In this case the first performs the `face detection` post-process. Enters the mux.

8.

```
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
hailooverlay ! queue leaky=no max-size-buffers=30 max-size-
bytes=0 max-size-time=0 ! \
```

Performs the Drawing.

9.

```
videoconvert n-threads=8 ! \
fpsdisplaysink video-sink=ximagesink name=hailo_display
sync=$sync_pipeline text-overlay=false ${additional_parameters}
```

Apply the final convert to let GStreamer utilize the format required by the `fpsdisplaysink` element

NOTE: Additional details about the pipeline provided in further examples

Classification Pipeline Raspberry Pi

Classification

The purpose of `classification.sh` is to demonstrate classification on one video file source. This is done by running a `single-stream object classification pipeline` on top of GStreamer using the Hailo-8 device.

```
-size-time=0 ! hailofilter so-path=/hailo/apps/x86/gstreamer/libdetection_draw.
so qos=false debug=False ! queue leaky=no max-size-buffers=30 max-size-bytes=0 m
ax-size-time=0 ! videoconvert ! fpsdisplaysink video-sink=ximagesink name=hailo_
display sync=true text-overlay=false
Setting pipeline to PAUSED ...
Pipeline is PREROLLING ...
Redistribute latency...
Redistribute latency...
Pipeline is PREROLLED ...
Setting pipeline to PLAYING ...
New clock: GstSystemClock
^Chandling interrupt.
Interrupt: Stopping pipeline ...
Execution ended after 0:00:17.053081956
Setting pipeline to PAUSED ...
Setting pipeline to READY ...
^C
root@hailo_tappas:/hailo/apps/x86/classification# ^C
root@hailo_tappas:/hailo/apps/x86/classification# ^C
root@hailo_tappas:/hailo/apps/x86/classification# cd ..
root@hailo_tappas:/hailo/apps/x86# cd
```

Options

```
./classification.sh [--input FILL-ME]
```

- `--input` is an optional flag, a path to the video displayed (default is `classification_movie.mp4`).
- `--show-fps` is a flag that prints the pipeline's fps to the screen.
- `--print-gst-launch` is a flag that prints the ready `gst-launch` command without running it.

Supported Networks:

- 'resnet_v1_50' - https://github.com/hailo-ai/hailo_model_zoo/blob/master/hailo_model_zoo/cfg/networks/resnet_v1_50.yaml

Run

```
cd $TAPPAS_WORKSPACE/apps/gstreamer/raspberrypi/classification
./classification.sh
```

How it works

This section is optional and provides a drill-down into the implementation of the `classification` app with a focus on explaining the `GStreamer` pipeline. This section

uses `resnet_v1_50` as an example network so network input width, height, and hef name are set accordingly.

```
gst-launch-1.0 \
  filesrc location=$input_source ! qtdemux ! h264parse ! avdec_h264
! videoconvert n-threads=8 ! \
  tee name=t hailomuxer name=hmux \
  t. ! queue leaky=no max-size-buffers=30 max-size-bytes=0 max-
size-time=0 ! hmux. \
  t. ! videoscale n-threads=8 ! video/x-raw, pixel-aspect-ratio=1/1
! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailonet hef-path=$hef_path device-id=$hailo_bus_id debug=False
is-active=true qos=false ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  hailofilter2 so-path=$postprocess_so qos=false ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! hmux. \
  hmux. ! hailooverlay ! \
  queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
  videoconvert n-threads=8 ! \
  fpsdisplaysink video-sink=ximagesink name=hailo_display
sync=false text-overlay=false ${additional_parameters}
```

Let's explain this pipeline section by section:

1. `filesrc location=$input_source ! qtdemux ! h264parse ! avdec_h264 ! videoconvert n-threads=8 ! \`

Specifies the location of the video used, then decode and convert to the required format using 8 threads for acceleration.

2. `tee name=t`

Declare a tee that splits the pipeline into two branches in order to keep the original resolution.

3. `hailomuxer name=hmux`

Declare a hailomuxer.

4. `t. ! queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! hmux. \`

A connection between the first split of the tee to the hailomuxer.

5.

```
t. ! videoscale n-threads=8 ! video/x-raw, pixel-aspect-
ratio=1/1 ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
```

The first split of the tee. Re-scale the video dimensions to fit the input of the network. In this case it is rescaling the video to 112X112 with the caps negotiation of `hailonet`. `hailonet` Extracts the needed resolution from the HEF file during the caps negotiation, and makes sure that the needed resolution is passed from previous elements. Before sending the frames into `hailonet` element set a queue so no frames are lost (Read more about queues [here](#))

6.

```
hailonet hef-path=$hef_path device-id=$hailo_bus_id debug=False
is-active=true qos=false ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
```

Performs the inference on the Hailo-8 device.

7.

```
hailofilter2 so-path=$postprocess_so qos=false ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! hmux. \
```

Performs a given post-process, in this case, performs `resnet_v1_50` classification post-process, which is mainly doing top1 on the inference output. Connected to the hailomuxer.

8.

```
hmux. ! hailooverlay ! \
queue leaky=no max-size-buffers=30 max-size-bytes=0 max-size-
time=0 ! \
```

A connection between the hailomuxer output to the hailooverlay element.
Performs classification draw-process.

9.

```
videoconvert n-threads=8 ! \
fpsdisplaysink video-sink=ximagesink name=hailo_display
sync=false text-overlay=false ${additonal_parameters}
```

Apply the final convert to let GStreamer utilize the format required by the `fpsdisplaysink` element.

Native Applications

Native detection application

Overview

This example demonstrate the use of libhailort's C API as part of a detection application. The example uses the Yolov5m model, on top of the Hailo-8 device.

Compiling with CMake

Run the following commands from the application's directory.

```
cmake -H. -Bbuild
cmake --build build
```

Running the example

```
./build/detection_app
```

Example details

The example demonstrates the use of libhailort's C API, all functions calls are based on the header provided in `hailort/include/hailo/hailort.h`. The input images are located in `input_images/` directory and the output images are written to `output_images/` directory. The application works on bitmap images with the following properties:

- 24bits per pixel
- Image size: 640x640

Code structure

- main function: The main function gets the input images and passes them to `infer` function.
- infer function: First, the function is preparing the device for inference:
 - Device initialization Open the Hailo PCIe device.
Used APIs: `hailo_create_pcie_device`
 - Configure the device from an HEF The next step is to create an `hailo_hef` object, and use it to configure the device for inference. Then, init an `hailo_configure_params_t` object with default values, configure the device and receive an `hailo_configured_network_group` object.
Used APIs: `hailo_create_hef_file()`, `hailo_init_configure_params`, `hailo_configure_device()`
 - Build VStreams
 - Initialize VStream parameters (both input and output).

- Create VStreams.

Used APIs: `hailo_make_input_vstream_params`,
`hailo_make_output_vstream_params`, `hailo_create_input_vstreams`,
`hailo_create_output_vstreams`

- Activating the network group before starting inference **Used APIs:**
`hailo_activate_network_group()`

Afterwards, the `infer` function starts the inference threads:

- One thread for writing the data to the device using the `write_all` function.
Used APIs: `hailo_vstream_write_raw_buffer`
- Three threads for receiving data from the device using the `read_all` function. **Used APIs:** `hailo_vstream_read_raw_buffer`
- One thread for post-processing the data received from the device, drawing the detected objects and writing the output files to the output directory. `FeatureData` is an object used for gathering the information needed for the post-processing and is created for each feature in the model.

Hailo GStreamer Elements

1. **HailoNet** - A bin element which contains a **hailosend** element, a **hailorecv** element and a queue between them. Responsible for configuring and running inference on the Hailo-8 device.
2. **HailoFilter** - An element which enables the user to apply a postprocess or drawing operation to a frame and its tensors.
3. **HailoMuxer** - An element designed for our new multi-device application. It muxes 2 similar streams into 1 stream, holding both stream's metadata.
4. **HailoDeviceStats** - Hailodevicestats is an element that samples power and temperature
5. **HailoAggregator** - HailoAggregator is an element designed for application with cascading networks. It has 2 sink pads and 1 source
6. **HailoCropper** - HailoCropper is an element designed for application with cascading networks. It has 1 sink and 2 sources

HailoNet

Overview

Hailonet is a bin element which contains a `hailosend` element, a `hailorecv` element and a queue between them. The `hailosend` element is responsible for sending the data received from the hailonet's sink to the `Hailo-8 device` for inference. Inference is done via the `VStreams API`. The `hailorecv` element will read the output buffers from the device and attach them as metadata to the source frame that inferred them. That is why the `hailonet` has only one source, even in cases where the `HEF` has more than one output layer.

Parameters

Configuration and activation of the Hailo network is done when the pipeline is started.

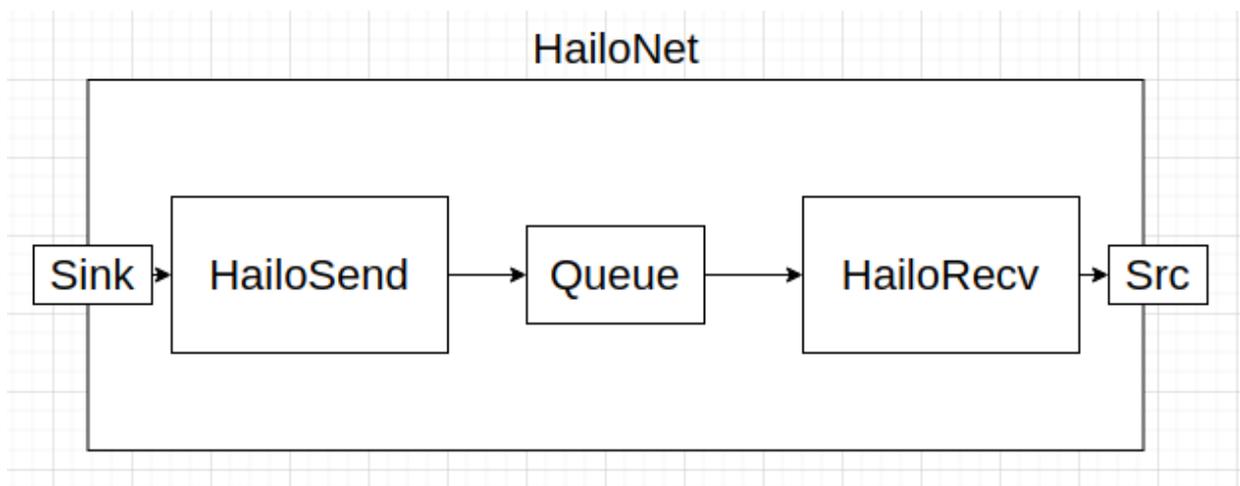
Infers data using the Hailo-8 chip. The data is inferred according to the selected HEF (`hef` property). Currently, only HEFs with one input layer are supported!

Selecting a specific PCIe device (when there are more than one) can be done with the `device-id` property.

Networks switching can be done with the `is-active` property (this can't be done in a CLI application since this property needs to be turned on and off during runtime).

For multi-context networks the `batch-size` property can be used to specify the batch size.

Using the inputs and outputs properties, specific VStreams can be selected for input and output inference.



Hierarchy

```
GObject
+----GInitiallyUnowned
  +----GstObject
    +----GstElement
      +----GstBin
        +----GstHailoNet
```

Implemented Interfaces:

GstChildProxy

Pad Templates:

SINK template: 'sink'

Availability: Always

Capabilities:

ANY

SRC template: 'src'

Availability: Always

Capabilities:

ANY

Element has no clocking capabilities.

Element has no URI handling capabilities.

Pads:

SINK: 'sink'

Pad Template: 'sink'

SRC: 'src'

Pad Template: 'src'

Element Properties:

name	: The name of the object flags: readable, writable String. Default: "hailonet0"
parent	: The parent of the object flags: readable, writable Object of type "GstObject"
async-handling changes	: The bin will handle Asynchronous state changes flags: readable, writable Boolean. Default: false
message-forward	: Forwards all children messages flags: readable, writable Boolean. Default: false
debug	: Should print debug information flags: readable, writable Boolean. Default: false
device-id	: Device ID ([<domain>]:<bus>:<device>.<func>, same as in <code>lspci</code> command) flags: readable, writable String. Default: null
hef-path	: Location of the HEF file to read flags: readable, writable String. Default: null
net-name	: Configure and run this specific network. If not passed, configure and run the default network - ONLY if there is one network in the HEF! flags: readable, writable String. Default: null
batch-size	: How many frame to send in one batch flags: readable, writable Unsigned Integer. Range: 1 - 16 Default: 1

outputs-min-pool-size: The minimum amount of buffers to allocate for each output layer

flags: readable, writable

Unsigned Integer. Range: 0 - 4294967295

Default: 16

outputs-max-pool-size: The maximum amount of buffers to allocate for each output layer or 0 for unlimited

flags: readable, writable

Unsigned Integer. Range: 0 - 4294967295

Default: 0

is-active : Controls whether this element should be active. By default, the hailonet element will not be active unless there is only one hailonet in the pipeline

flags: readable, writable

Boolean. Default: **false**

Children:

hailorecv

hailo_infer_q_0

hailosend

HailoFilter

Overview

Hailofilter is an element which enables the user to apply a postprocess or drawing operation to a frame and its tensors. It provides an entry point for a compiled .so file that the user writes, inside of which they will have access to the original image frame, the tensors output by the network for that frame, and any metadata attached. At first the hailofilter will read the buffer from the sink pad, then apply the filter defined in the provided .so, until finally sending the filtered buffer along the source pad to continue down the pipeline.

Parameters

The most important parameter here is the `so-path`. Here you provide the path to your compiled .so (shared object file) that applies your wanted filter.

By default, the hailofilter will call on a filter() function within the .so as the entry point. If your .so has multiple entry points, for example in the case of slightly different network flavors, then you can chose which specific filter function to apply via the `function-name` parameter.

As a member of the GstVideoFilter hierarchy, the hailofilter element supports qos (Quality of Service). Although qos typically tries to garuantee some level of performance, it can lead to frames dropping. For this reason it is **advised to always set `qos=false`** to avoid either tensors being dropped or not drawn.

Hierarchy

```

GObject
+----GInitiallyUnowned
  +----GstObject
    +----GstElement
      +----GstBaseTransform
        +----GstVideoFilter
          +----GstHailoFilter

Pad Templates:
SINK template: 'sink'
Availability: Always
Capabilities:
  video/x-raw
    format: { (string)RGB, (string)YUY2 }
    width: [ 1, 2147483647 ]
    height: [ 1, 2147483647 ]
    framerate: [ 0/1, 2147483647/1 ]

SRC template: 'src'
Availability: Always
Capabilities:
  video/x-raw
    format: { (string)RGB, (string)YUY2 }
    width: [ 1, 2147483647 ]

```

```

height: [ 1, 2147483647 ]
framerate: [ 0/1, 2147483647/1 ]

```

Element has no clocking capabilities.
 Element has no URI handling capabilities.

Pads:

```

SINK: 'sink'
  Pad Template: 'sink'
SRC: 'src'
  Pad Template: 'src'

```

Element Properties:

```

name          : The name of the object
               flags: readable, writable
               String. Default: "hailofilter0"
parent        : The parent of the object
               flags: readable, writable
               Object of type "GstObject"
qos           : Handle Quality-of-Service events
               flags: readable, writable
               Boolean. Default: true
debug         : debug
               flags: readable, writable, controllable
               Boolean. Default: false
so-path       : Location of the so file to load
               flags: readable, writable, changeable only in
NULL or READY state
               String. Default: null
function-name : function-name
               flags: readable, writable, changeable only in
NULL or READY state
               String. Default: "filter"

```

HailoFilter2

Overview

HailoFilter2 is an element which enables the user to apply a postprocess operation on hailonet's output tensors. It provides an entry point for a compiled .so file that the user writes, inside of which they will have access to the original image frame, the tensors output by the network for that frame, and any metadata attached. At first the hailoFilter2 will read the buffer from the sink pad, then apply the filter defined in the provided .so, until finally sending the filtered buffer along the source pad to continue down the pipeline.

Parameters

The most important parameter here is the `so-path`. Here you provide the path to your compiled .so that applies your wanted filter.

By default, the hailoFilter2 will call on a filter() function within the .so as the entry point. If your .so has multiple entry points, for example in the case of slightly different network flavors, then you can choose which specific filter function to apply via the `function-name` parameter.

As a member of the GstVideoFilter hierarchy, the hailoFilter2 element supports qos (Quality of Service). Although qos typically tries to guarantee some level of performance, it can lead to frames dropping. For this reason it is **advised to always set `qos=false`** to avoid either tensors being dropped or not drawn.

Hierarchy

```
GObject
+----GInitiallyUnowned
  +----GstObject
    +----GstElement
      +----GstBaseTransform
        +----GstHailoFilter2
```

Pad Templates:

```
SRC template: 'src'
Availability: Always
Capabilities:
  ANY
```

```
SINK template: 'sink'
Availability: Always
Capabilities:
  ANY
```

Element has no clocking capabilities.
Element has no URI handling capabilities.

Pads:

```
SINK: 'sink'
  Pad Template: 'sink'
```

SRC: 'src'

Pad Template: 'src'

Element Properties:

name	: The name of the object flags: readable, writable String. Default: "hailofilter2-0"
parent	: The parent of the object flags: readable, writable Object of type "GstObject"
qos	: Handle Quality-of-Service events flags: readable, writable Boolean. Default: false
so-path	: Location of the so file to load flags: readable, writable, changeable only in
NULL or READY state	String. Default: null
function-name	: function-name flags: readable, writable, changeable only in
NULL or READY state	String. Default: "filter"

HailoOverlay

Overview

HailoOverlay is a drawing element that can draw postprocessed results on an incoming video frame. This element supports the following results:

- Detection - Draws the rectangle over the frame, with the label and confidence (rounded).
- Classification - Draws a classification over the frame, at the top left corner of the frame.
- Landmarks - Draws a set of points on the given frame at the wanted coordinates.
- Tiles - Can draw tiles as a thin rectangle.

Parameters

As a member of the GstBaseTransform hierarchy, the hailooverlay element supports qos ([Quality of Service](#)). Although qos typically tries to guarantee some level of performance, it can lead to frames dropping. For this reason it is **advised to always set `qos=false`** to avoid either tensors being dropped or not drawn.

Hierarchy

```
GObject
+----GInitiallyUnowned
  +----GstObject
    +----GstElement
      +----GstBaseTransform
        +----GstHailoOverlay
```

Pad Templates:

SINK template: **'sink'**

Availability: Always

Capabilities:

video/x-raw

format: { (string)RGB }

width: [1, 2147483647]

height: [1, 2147483647]

framerate: [0/1, 2147483647/1]

SRC template: **'src'**

Availability: Always

Capabilities:

video/x-raw

format: { (string)RGB }

width: [1, 2147483647]

height: [1, 2147483647]

framerate: [0/1, 2147483647/1]

Element has no clocking capabilities.

Element has no URI handling capabilities.

Pads:

```
SINK: 'sink'  
  Pad Template: 'sink'  
SRC: 'src'  
  Pad Template: 'src'
```

Element Properties:

```
name          : The name of the object  
               flags: readable, writable  
               String. Default: "hailooverlay0"  
parent        : The parent of the object  
               flags: readable, writable  
               Object of type "GstObject"  
qos           : Handle Quality-of-Service events  
               flags: readable, writable  
               Boolean. Default: false
```

HailoDeviceStats

- [HailoDeviceStats](#)
 - [Overview](#)
 - [Parameters](#)
 - [Hierarchy](#)

Overview

Hailodevicestats is an element that samples power and temperature. It doesn't have any pads, it just has to be part of the pipeline. An example for using this element could be found under the [detection / multistream_multidevice](#) app.

Parameters

Determine the time period between samples with the [interval](#) property.

Choose device with the [device-id](#) property.

Hierarchy

```
GObject
+----GInitiallyUnowned
+----GstObject
+----GstElement
+----GstHailoDeviceStats
```

Pad Templates:

none

Element has no clocking capabilities.

Element has no URI handling capabilities.

Pads:

none

Properties:

name	: The name of the object flags: readable, writable String. Default: "hailodevicestats0"
parent	: The parent of the object flags: readable, writable Object of type "GstObject"

```
interval          : Time period between samples, in seconds
                  flags: readable, writable
                  Unsigned Integer. Range: 0 - 4294967295
Default: 1

device-id         : Device ID ([<domain>]:<bus>:<device>.<func>,
same as in lspci command)
                  flags: readable, writable
                  String. Default: null

silent           : Should print statistics
                  flags: readable, writable
                  Boolean. Default: false

power-measurement : Current power measurement of device
                  flags: readable
                  Float. Range:                0 -
3.402823e+38 Default:                0

temperature      : Current temperature of device
                  flags: readable
                  Float. Range:                0 -
3.402823e+38 Default:                0
```

HailoMuxer

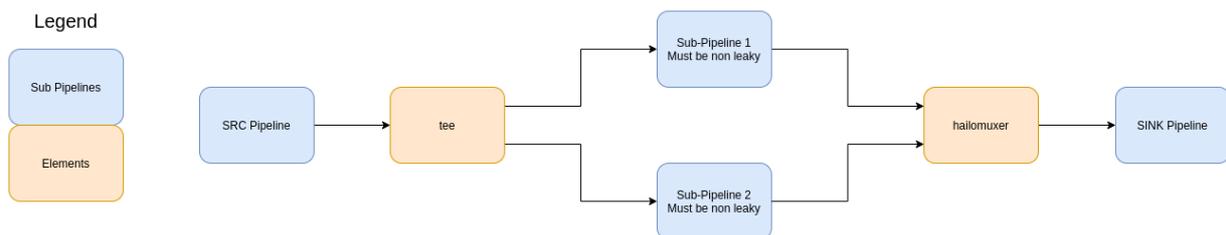
Overview

HailoMuxer is an element designed for our new multi-device application. It muxes 2 similar streams into 1 stream, holding both stream's metadata. It has 2 src elements and 1 sink, and whenever there are buffers on both src pads, it takes only 1 of the buffers and passes it on, with both buffer's metadata.

Parameters

There are no unique properties to hailomuxer. The only parameters are the baseclass parameters, which are 'name' and 'parent'.

Example



Hierarchy

```

GObject
+----GInitiallyUnowned
+----GstObject
+----GstElement
+----GstHailoMuxer
  
```

Pad Templates:

```

SRC template: 'src'
Availability: Always
Capabilities:
  ANY
  
```

```

SINK template: 'sink_%u'
Availability: On request
Capabilities:
  ANY
  
```

Element has no clocking capabilities.
Element has no URI handling capabilities.

Pads:

```

SRC: 'src'
  Pad Template: 'src'
  
```

Element Properties:

```

name          : The name of the object
flags: readable, writable
String. Default: "hailomuxer0"
  
```

```
parent          : The parent of the object  
                 flags: readable, writable  
                 Object of type "GstObject"
```

HailoPython

- [HailoPython](#)
 - [Overview](#)
 - [Parameters](#)
 - [Hierarchy](#)

Overview

HailoPython is an element which enables the user to apply processing operations to an image via python. It provides an entry point for a python module that the user writes, inside of which they will have access to the Hailo raw-output (output tensors) and postprocessed-outputs (detections, classifications etc..) as well as the gstreamer buffer. The python function will be called for each buffer going through the hailopython element.

Parameters

The two parameters that define the function to call are `module` and `function` for the module path and function name respectively. In addition, as a member of the `GstVideoFilter` hierarchy, the `hailofilter` element supports `qos` ([Quality of Service](#)). Although `qos` typically tries to guarantee some level of performance, it can lead to frames dropping. For this reason it is **advised to always set `qos=false`** to avoid either tensors being dropped or not drawn.

Hierarchy

```

GObject
+----GInitiallyUnowned
  +----GstObject
    +----GstElement
      +----GstBaseTransform
        +----GstVideoFilter
          +----GstHailoPython

Pad Templates:
SRC template: 'src'
Availability: Always
Capabilities:
  video/x-raw
    format: { (string)RGB, (string)YUY2 }
    width: [ 1, 2147483647 ]
    height: [ 1, 2147483647 ]
    framerate: [ 0/1, 2147483647/1 ]

SINK template: 'sink'
Availability: Always
Capabilities:
  video/x-raw
    format: { (string)RGB, (string)YUY2 }
    width: [ 1, 2147483647 ]
    height: [ 1, 2147483647 ]

```

```
framerate: [ 0/1, 2147483647/1 ]
```

```
Element has no clocking capabilities.
Element has no URI handling capabilities.
```

```
Pads:
```

```
  SINK: 'sink'
    Pad Template: 'sink'
  SRC: 'src'
    Pad Template: 'src'
```

```
Element Properties:
```

```
  name          : The name of the object
                  flags: readable, writable
                  String. Default: "hailopython0"
  parent        : The parent of the object
                  flags: readable, writable
                  Object of type "GstObject"
  qos          : Handle Quality-of-Service events
                  flags: readable, writable
                  Boolean. Default: true
  module       : Python module name
                  flags: readable, writable
                  String. Default:
"/local/workspace/tappas/processor.py"
  function      : Python function name
                  flags: readable, writable
                  String. Default: "run"
```

HailoCropper

Overview

HailoCropper is an element providing cropping functionality, designed for application with cascading networks, meaning doing one task based on a previous task. It has 1 sink and 2 sources. **HailoCropper** receives a frame on its sink pad, then invokes its **prepare_crops** method that returns the vector of crop regions of interest (crop_roi), For each crop_roi it creates a cropped image (representing its x, y, width, height in the full frame). The cropped images are then sent to the second src. From the first src we push the original frame that the detections were cropped from.

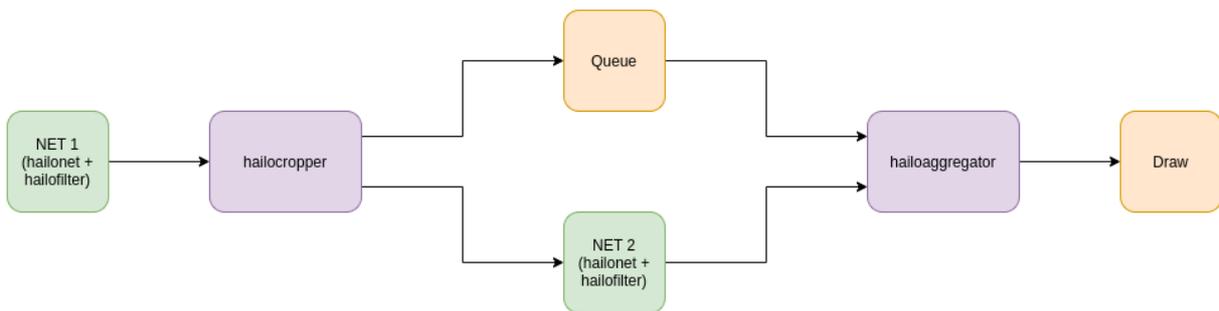
By default, **HailoCropper** receives a video frame that has detections (means a previous **HailoNet** + **HailoFilter** ran) on its sinkpad. For each detection it creates a cropped image (using a specific algorithm to create a scaled image with the same aspect ratio). This is used by the cascading networks app [Face Landmarks based on Face Detection](#).

Derived classes can override the default **prepare_crops** behaviour and decide where to crop and how many times. **hailotilecropper** element does this exact thing when splitting the frame into tiles by rows and columns.

Parameters

There is only one property for this element other than the common 'name' and 'parent'. The name of this boolean property is 'internal-offset' and it is used to determine whether we use the original offset* of the buffer or overwrite it with our own offset. The offset of the buffer is given to the original buffer and all the crops, and used by the hailoaggregator, to make sure the cropped detections we are 'muxing' with the original buffer are actually from the same buffer. *Offset is an attribute of buffer that determines on what offset this buffer is since the start of the pipeline run, represented by number of buffers. It's similar to frame-id in video. On some videos the offset attribute is not created by the filesrc element and it is set to -1 (casted to uint64), therefore if we want to use it to determine what the current frame is, we should somehow track the number of buffers and set this offset accordingly.

Example



Hierarchy

```

GObject
+----GInitiallyUnowned
+----GstObject
+----GstElement
+----GstHailoCropper
  
```

Pad Templates:

```

SRC template: 'src'
Availability: Always
Capabilities:
  ANY
  
```

```

SINK template: 'sink'
Availability: Always
Capabilities:
  ANY
  
```

Element has no clocking capabilities.
Element has no URI handling capabilities.

Pads:

```

SINK: 'sink'
  Pad Template: 'sink'
SRC: 'src_0'
  Pad Template: 'src'
SRC: 'src_1'
  
```

Pad Template: 'src'

Element Properties:

name	: The name of the object flags: readable, writable String. Default: "hailocropper0"
parent	: The parent of the object flags: readable, writable Object of type "GstObject"
internal-offset offset.	: Whether to use Gstreamer offset of internal offset. flags: readable, writable, controllable Boolean. Default: false

HailoTileCropper

Overview

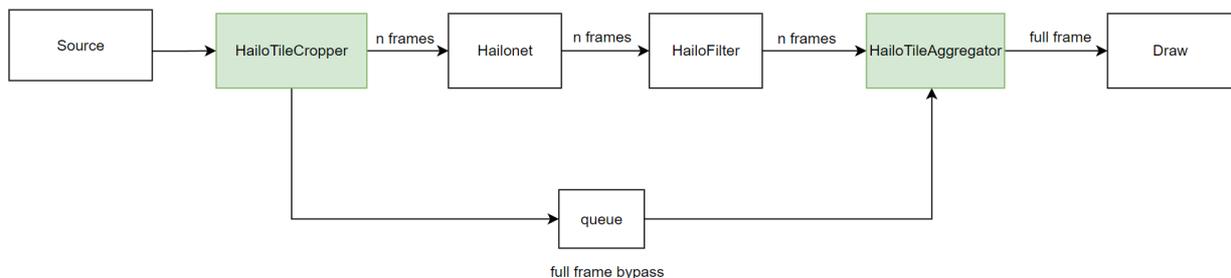
HailoTileCropper is a derived element of **hailoCropper** and it is used in the **Tiling** app. It overrides the default **prepare_crops** behaviour to return a vector of tile regions of interest, and allows splitting the incoming frame into tiles by rows and columns. Each tile stores their x, y, width, and height (with overlap between tiles included) in the full frame. Just like the base HailoCropper, the full original frame is sent to the first src pad while all the cropped images are sent to the second.

hailoaggregator will aggregate the cropped tiles and stitch them back to the original resolution.

Parameters

- **tiles-along-x-axis** : Number of tiles along x axis (columns) - default 2
- **tiles-along-y-axis** : Number of tiles along x axis (rows) - default 2
- **overlap-x-axis** : Overlap in percentage between tiles along x axis (columns) - default 0
- **overlap-y-axis** : Overlap in percentage between tiles along y axis (rows) - default 0
- **tiling-mode** : Tiling mode (0 - single-scale, 1 - multi-scale) - default 0
- **scale-level** : Scales (layers of tiles) in addition to the main layer 1: [(1 X 1)] 2: [(1 X 1), (2 X 2)] 3: [(1 X 1), (2 X 2), (3 X 3)] - default 2

Example



Hierarchy

```

GObject
+----GInitiallyUnowned
+----GstObject
+----GstElement
+----GstHailoBaseCropper
+----GstHailoTileCropper

Pad Templates:
SRC template: 'src'
Availability: Always
Capabilities:
video/x-raw
format: { (string)RGB, (string)YUY2 }
  
```

```

width: [ 1, 2147483647 ]
height: [ 1, 2147483647 ]
framerate: [ 0/1, 2147483647/1 ]

```

SINK template: 'sink'

Availability: Always

Capabilities:

video/x-raw

```

format: { (string)RGB, (string)YUY2 }
width: [ 1, 2147483647 ]
height: [ 1, 2147483647 ]
framerate: [ 0/1, 2147483647/1 ]

```

Element has no clocking capabilities.

Element has no URI handling capabilities.

Pads:

SINK: 'sink'

Pad Template: 'sink'

SRC: 'src_0'

Pad Template: 'src'

SRC: 'src_1'

Pad Template: 'src'

Element Properties:

name	: The name of the object flags: readable, writable String. Default: "hailotilecropper0"
parent	: The parent of the object flags: readable, writable Object of type "GstObject"
internal-offset	: Whether to use Gstreamer offset of internal offset. NOTE: If using file sources, Gstreamer does not generate offsets for buffers, so this property should be set to true in such cases. flags: readable, writable, controllable Boolean. Default: false
tiles-along-x-axis	: Number of tiles along x axis (columns) flags: readable, writable, changeable only in
NULL or READY state	Unsigned Integer. Range: 1 - 20 Default: 2
tiles-along-y-axis	: Number of tiles along x axis (rows) flags: readable, writable, changeable only in
NULL or READY state	Unsigned Integer. Range: 1 - 20 Default: 2
overlap-x-axis	: Overlap in percentage between tiles along x axis (columns) flags: readable, writable, changeable only in
NULL or READY state	Float. Range: 0 - 1 Default: 0
overlap-y-axis	: Overlap in percentage between tiles along y axis (rows)

```

NULL or READY state      flags: readable, writable, changeable only in
1 Default:
  tiling-mode             Float. Range:          0 -
                          0
                          : Tiling mode
                          flags: readable, writable
                          Enum "GstHailoTileCropperTilingMode" Default:
0, "single-scale"
                          (0): single-scale    - Single Scale
                          (1): multi-scale     - Multi Scale
  scale-level             : 1: [(1 X 1)] 2: [(1 X 1), (2 X 2)] 3: [(1 X
1), (2 X 2), (3 X 3)]
NULL or READY state      flags: readable, writable, changeable only in
                          Unsigned Integer. Range: 1 - 3 Default: 2

```

HailoAggregator

Overview

HailoAggregator is an element designed for applications with cascading networks or cropping functionality, meaning doing one task based on a previous task. A complement to the [HailoCropper](#), the two elements work together to form versatile apps. It has 2 sink pads and 1 source: the first sinkpad receives the original frame from an upstream hailocropper, while the other receives cropped buffers from that hailocropper. The HailoAggregator waits for all crops of a given original frame to arrive, then sends the original buffer with the combined metadata of all collected crops. HailoAggregator also performs a 'flattening' functionality on the detection metadata when receiving each frame: detections are taken from the cropped frame, copied to the main frame and re-scaled/moved to their corresponding location in the main frame (x,y,width,height). As an example:

- [Face Landmarks based on Face Detection](#) - HailoCropper crops each face detection -> HailoNet + FaceLandmarks post for each face -> HailoAggregator aggregates the frames back.
- [Tiling - hailotilecropper](#) crops the image to tiles -> HailoNet + Detection post for each tile -> HailoAggregator aggregates the frames back and 'flatten` the detection objects in the metadata.

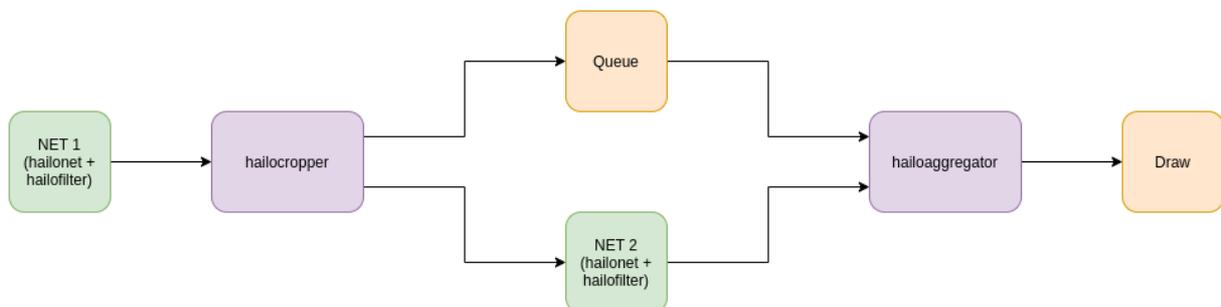
HailoAggregator exports two methods to extend or override in derived elements:

- `handle_sub_frame_roi`: Functionality to perform for each incoming sub frame. Calls `flattening` method.
- `post_aggregation`: Functionality to perform after all frames are aggregated successfully. Base implementation does nothing.

Parameters

There are no unique properties to hailoaggregator. The only parameters are the baseclass parameters, which are 'name' and 'parent'.

Example



Hierarchy

```

GObject
+----GInitiallyUnowned
+----GstObject
  
```

```
+-----GstElement
+-----GstHailoAggregator
```

Pad Templates:

```
SRC template: 'src'
  Availability: Always
  Capabilities:
    ANY
```

```
SINK template: 'sink'
  Availability: Always
  Capabilities:
    ANY
```

Element has no clocking capabilities.
Element has no URI handling capabilities.

Pads:

```
SINK: 'sink_0'
  Pad Template: 'sink'
SINK: 'sink_1'
  Pad Template: 'sink'
SRC: 'src'
  Pad Template: 'src'
```

Element Properties:

```
name          : The name of the object
                flags: readable, writable
                String. Default: "hailoaggregator0"
parent        : The parent of the object
                flags: readable, writable
                Object of type "GstObject"
flatten-detections : perform a 'flattening' functionality on the
detection metadata
                when receiving each frame.
                flags: readable, writable, changeable only in
NULL or READY state
                Boolean. Default: true
```

HailoTileAggregator

Overview

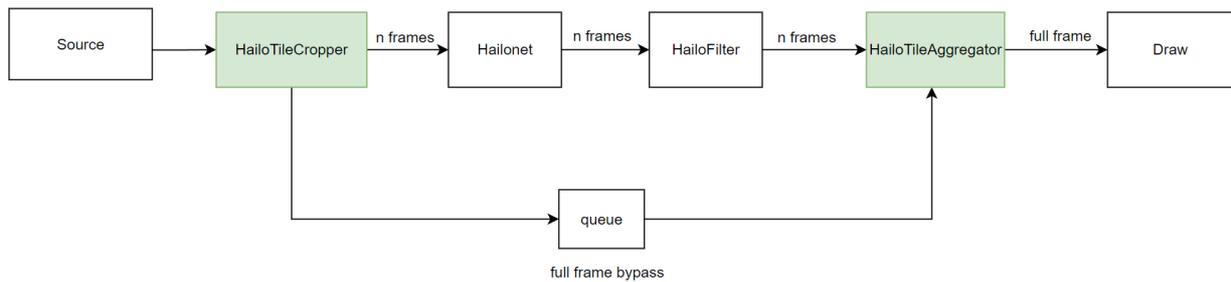
HailoTileAggregator is a derived element of **hailoAggregator** and it is used in the **Tiling** app. A complement to the **HailoTileCropper**, the two elements work together to form a versatile tiling apps.

The element extends two methods of the parent element:

- **handle_sub_frame_roi**: Functionality to perform for each incoming sub frame. Performs **remove_exceeded_bboxes** (remove boxes close to boundary - using given **border_threshold**) and then parent element performs flatten detections.

- **post_aggregation**: Functionality to perform after all frames are aggregated successfully. Performs **remove_large_landscape** and **NMS**.

Example



Hierarchy

GObject

```

+----GInitiallyUnowned
  +----GstObject
    +----GstElement
      +----GstHailoAggregator
        +----GstHailoTileAggregator
  
```

Pad Templates:

```

SRC template: 'src'
Availability: Always
Capabilities:
  ANY
  
```

```

SINK template: 'sink'
Availability: Always
Capabilities:
  ANY
  
```

Element has no clocking capabilities.

Element has no URI handling capabilities.

Pads:

```

SINK: 'sink_0'
  Pad Template: 'sink'
SINK: 'sink_1'
  Pad Template: 'sink'
SRC: 'src'
  Pad Template: 'src'
  
```

Element Properties:

```

name          : The name of the object
                flags: readable, writable
                String. Default: "hailotileaggregator0"
parent        : The parent of the object
                flags: readable, writable
                Object of type "GstObject"
flatten-detections : perform a 'flattening' functionality on the
detection metadata when receiving each frame
                flags: readable, writable, changeable only in
  
```

```
NULL or READY state
    iou-threshold      Boolean. Default: true
                      : threshold
                      flags: readable, writable, changeable only in
NULL or READY state
    border-threshold  Float. Range:          0 -
1 Default:           0.3
    border-threshold : border threshold
                      flags: readable, writable, changeable only in
NULL or READY state
    remove-large-landscape: remove large landscape objects when running
1 Default:           0.1
    remove-large-landscape: remove large landscape objects when running
in multi-scale mode
                      flags: readable, writable, changeable only in
NULL or READY state
```

Installation

Using Dockers

Install Docker

The section below would help you with the installation of Docker.

```
# Install curl
sudo apt-get install -y curl

# Get and install docker
curl -fsSL https://get.docker.com -o get-docker.sh
sh get-docker.sh

# Add your user (who has root privileges) to the Docker group
sudo usermod -aG docker $USER

# Reboot/log out in order to apply the changes to the group
sudo reboot
```

Note: Consider reading [Running out of disk space](#) if your system is space limited

Running TAPPAS container from pre-built Docker image

Preparations

HailoRT PCIe driver is required - install instructions are provided in HailoRT documentations. make sure that the driver is installed correctly by: [Verify Hailo installation](#).

Unzip `tappas_VERSION_docker.zip`, it should contain the following files:

- **hailo-docker-tappas-VERSION.tar**: the pre-built docker image
- **run_tappas_docker.sh**: Script that loads and runs the docker image
- **dockerfile.tappas_run**: Dockerfile used within the first load

Running for the first time

In order to use TAPPAS release Docker image, you should run the following script:

```
./run_tappas_docker.sh --tappas-image TAPPAS_IMAGE_PATH
```

NOTE: TAPPAS_IMAGE_PATH is the path to the **hailo-docker-tappas-VERSION.tar**

The script would load the docker image, and start a new container. The script might take a couple of minutes, and after that, you are ready to go.

Resuming (Second time and on)

From now on you should run the script with the `--resume` flag

```
./run_tappas_docker.sh --resume
```

NOTE: The reason that you want to use the `--resume` flag is that the container already exists, so only attaching to the container is required.

Flags and advanced use-cases

```
./run_tappas_docker.sh [options]
Options:
  --help           Show this help
  --tappas-image   Path to tappas image
  --resume        Resume an old container
  --container-name Start a container with a specific name,
defaults to hailo_tappas_container
```

Use-cases

For building a new container with the default name:

```
./run_tappas_docker.sh --tappas-image TAPPAS_IMAGE_PATH
```

For resuming an old container:

```
./run_tappas_docker.sh --resume
```

Both of these methods can receive a container name:

```
./run_tappas_docker.sh --tappas-image TAPPAS_IMAGE_PATH --container-
name CONTAINER_NAME
./run_tappas_docker.sh --resume --container-name CONTAINER_NAME
```

for example:

```
./run_hailort_docker.sh hailo-docker-tappas-VERSION.tar --container-
name hailo_tappas_container
```

Build Docker image by your own

Preparations

HailoRT PCIe driver is required - install instructions are provided in HailoRT documentations. make sure that the driver is installed correctly by: [Verify Hailo installation](#).

Steps

Enter the TAPPAS release directory:

```

├─ tappas_VERSION_user_guide.pdf
├─ tappas_VERSION_linux_installer.zip
├─ tappas_VERSION_docker.zip

```

Firstly, unpack the `tappas_VERSION_linux_installer.zip`. Before we start the build process, we must copy the `HailoRT` release to the repo under the name `release`.

Note: This version runs and tested with HailoRT version 4.5.0.

The tree should look like this:

```

├─ build_docker.sh
├─ core
├─ docker
├─ docs
├─ downloader
├─ manual-install.md
├─ README.md
├─ release --> copied `HailoRT` release
├─ resources
├─ tools
├─ apps
├─── gstreamer
├─── x86
├─── arm
├─── native

```

After that

```

# enter the repo
cd tappas

# Build the Docker
./build_docker.sh

# Run the Docker
./docker/run_docker.sh

```

Details

This section describes Hailo-Docker files hierarchy and purpose.

Lets take a look inside the `docker` folder, you can see: `Dockerfile.base`, `Dockerfile.gstreamer`, `run_docker.sh`, `scripts`

run_docker.sh - An easy-to-use run script that handles all the arguments required by the docker.

```

$ ./run_docker.sh --help
Run Hailo Docker:
The default mode is trying to create a new container

```

Options:

```

--help          Show this help
--resume       Resume an old container
--resume-command Resume command (used only when --resume flag
is used)
--override     Start a new container, if exists already,
delete the previous one

```

- If no flags specified, the script would try to create a new container (and could potentially fail if one already exists)
- **--override** - When used the script would create a new container and delete the previous one if exists
- **--resume** - The script would try to resume the last container created
- **--resume-command** - In case where **--resume** flags is used, the default command is `/bin/bash`, this could be changed using this flags

Dockerfile.base - Our base dockerfile. Installs required packages, Hailo's software, and drivers. **Dockerfile.gstreamer** - Based on top of **Dockerfile.base**. Installs and copies GStreamer requirements.

scripts - Directory of scripts used by the Docker files.

Compile hailonet and hailotools inside the docker

install_hailo_gstreamer.sh - compiles Hailo's gstreamer plugin including hailonet, hailofilter and the posprocess files. This script is called inside the docker build process.

Sometimes you will prefer to change the sources inside the docker for development or debug purposes and compile them inside the docker. Running this script from `$TAPPAS_WORKSPACE/scripts` directory will build and deploy the sources.

Troubleshooting

Hailo containers are taking to much space

Creating new docker containers with **--override** does not assure that the directory of cached images and containers is cleaned. to prevent your system to ran out of memory and clean `/var/lib/docker` run `docker system prune` from time to time.

Running out of disk space

Change Docker root directory - By default, Docker stores most of its data inside the `/var/lib/docker` directory on Linux systems. There may come a time when you want to move this storage space to a new location. For example, the most obvious reason might be that you're running out of disk space.

Firstly, stop the Docker from running

```

$ sudo systemctl stop docker.service
$ sudo systemctl stop docker.socket

```

Next, we need to edit the `/lib/systemd/system/docker.service` file

```
$ sudo vim /lib/systemd/system/docker.service
```

The line we need to edit looks like this:

```
ExecStart=/usr/bin/dockerd -H fd://
```

Edit the line by putting a `-g` and the new desired location of your Docker directory. When you're done making this change, you can save and exit the file.

```
ExecStart=/usr/bin/dockerd -g /new/path/docker -H fd://
```

```
[Unit]
Description=Docker Application Container Engine
Documentation=https://docs.docker.com
After=network-online.target firewalld.service containerd.service
Wants=network-online.target
Requires=docker.socket containerd.service

[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
ExecStart=/usr/bin/dockerd -g /new/path/docker -H fd:// --containerd=/run/containerd/containerd.sock
ExecReload=/bin/kill -s HUP $MAINPID
TimeoutSec=0
RestartSec=2
Restart=always

# Note that StartLimit* options were moved from "Service" to "Unit" in systemd 229.
```

If you haven't already, create the new directory where you plan to move your Docker files to.

```
$ sudo mkdir -p /new/path/docker
```

Next, reload the systemd configuration for Docker, since we made changes earlier. Then, we can start Docker.

```
$ sudo systemctl daemon-reload
$ sudo systemctl start docker
```

Just to make sure that it worked, run the `ps` command to make sure that the Docker service is utilizing the new directory location.

```
$ ps aux | grep -i docker | grep -v grep
```

```
rotomb@hat-138-lap: /run$ ps aux | grep -i docker | grep -v grep
root    12849  6.2  0.4 1728252 75648 ?        Ssl   17:50   0:00 /usr/bin/dockerd -g /new/path/docker -H fd:// --containerd=/run/containerd/containerd.sock
```

Manual Install

A guide about how to install our required components manually.

NOTE: Only ubuntu 18.04 with GStreamer 1.14 is tested.

Download required files

Download all the required files (models, videos, compiled .so's) by running:

```
cd downloader
pip install -r requirements.txt
python main.py
```

NOTE: This could take up to a couple of minutes **NOTE:** python 3.6 or above is required

Hailo install

First you would need to install Hailo's platform, follow our install guide for that. After Hailo is installed: And then [Make sure that Hailo works](#)

GStreamer install

Install the required packages from apt

```
add-apt-repository ppa:oibaf/graphics-drivers
apt-get update

apt-get update && apt-get -y --no-install-recommends install \
wget \
build-essential \
pkg-config \
software-properties-common pciutils \
lshw \
lsb-release \
va-driver-all \
vainfo \
autoconf \
automake \
libtool \
bison \
flex \
gstreamer-1.0 \
gstreamer1.0-dev \
libgstreamer1.0-0 \
gstreamer1.0-plugins-base \
gstreamer1.0-plugins-bad \
gstreamer1.0-plugins-ugly \
gstreamer1.0-libav \
gstreamer1.0-vaapi \
gstreamer1.0-doc \
gstreamer1.0-tools \
```

```

gststreamer1.0-x \
gststreamer1.0-alsa \
gststreamer1.0-g1 \
gststreamer1.0-gtk3 \
gststreamer1.0-qt5 \
gststreamer1.0-pulseaudio \
python-gst-1.0 \
libgirepository1.0-dev \
libgststreamer-plugins-base1.0-dev \
libcairo2-dev \
gir1.2-gstreamer-1.0 \
python3-gi \
python-gi-dev

```

Then verify that GStreamer is installed in the right version by using this follow command:

```
gst-launch-1.0 --gst-version | awk '{print $NF}' | cut -d. -f1,2
```

The expected output should be **1.14**

RTSP

If you are planning to use **RTSP** source, a patch to fix an issue in RTSP plugin is required within **gst-plugins-good** and therefore you can't install **gst-plugins-good** directly from apt. If you have no plans to use RTSP source just run:

```
apt-get install gstreamer1.0-plugins-good
```

If you do plan to use **RTSP** source

Compile gst-plugins-good

```

git clone -b 1.14 https://github.com/GStreamer/gst-plugins-good.git
cd gst-plugins-good
git apply <your parent
dir>/hailo/core/patches/rtsp/rtspsrc_stream_id_path.patch

```

This section provided above would clone and apply the patch, you can verify that the patch applied successfully by running **git status** and verify that **gststrtsprc.c** is modified.

```

root@hailo-nvr:/hailo/sources/gst-plugins-good# git status
On branch 1.14
Your branch is up to date with 'origin/1.14'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
  directory)

```

```
modified:   gst/rtsp/gstrtspsrc.c
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

After that compile `gst-plugins-good`

```
meson build --prefix /usr/
ninja -C build
sudo ninja -C build install
```

You can verify that the install works by running the follow command for example:

```
root@hailo-nvr:/hailo/sources/gst-plugins-good# gst-inspect-1.0 rtsp
Plugin Details:
  Name                rtsp
  Description          transfer data via RTSP
  Filename             /usr/lib/x86_64-linux-gnu/gstreamer-
1.0/libgstrtsps.so
  Version              1.14.5
  License              LGPL
  Source module        gst-plugins-good
  Binary package       GStreamer Good Plug-ins source release
  Origin URL           Unknown package origin

  rtpdec: RTP Decoder
  rtspsrc: RTSP packet receiver

  2 features:
  +- 2 elements
```

Install Opencv

Hailo GStreamer plugins requires opencv in version (4.5.2). You can get the required modules pre compiled from our sources and copy them to your file system with:

```
cp -r <base_dir>/hailo/core/opencv/* /usr/lib/ && \
```

Or compile it from source:

1. Install core packages:

```
sudo apt update
sudo apt install -y cmake g++ make
```

2. Download the latest opencv release source code via git or zip file (4.5.2 as for 1.6.2021):

```
sudo apt install wget unzip
wget -O opencv.zip
https://github.com/opencv/opencv/archive/4.5.2.zip
unzip opencv-4.5.2.zip
```

Or

```
sudo apt install git
git clone https://github.com/opencv/opencv.git
git checkout tags/4.5.2
```

3. Create a build directory

```
mkdir -p build && cd build
```

4. Build and install using cmake with flags (each flag should start with -D).

```
cmake -DOPENCV_GENERATE_PKGCONFIG=ON -DBUILD_LIST=core,imgproc -
DINSTALL_C_EXAMPLES=ON -DINSTALL_PYTHON_EXAMPLES=ON -
DCMAKE_INSTALL_PREFIX=/usr/local -DCMAKE_BUILD_TYPE=RELEASE
../opencv
make -j4
make install
```

- `OPENCV_GENERATE_CONFIG=ON` : Create pkg-config file for the install
- `BUILD_LIST=core,imgproc` : Build only the relevant modules
- `INSTALL_C_EXAMPLES=ON, INSTALL_PYTHON_EXAMPLES=ON` : Install C and Python examples
- `CMAKE_INSTALL_PREFIX=/usr` : choose the installation folder
- `DCMAKE_BUILD_TYPE=RELEASE` : choose the build type

NOTE: `BUILD_LIST` argument will build and install only a small section of opencv library, we use it to get only what the application requires, and to faster the build process. If you want all of opencv skip it.

Hailo plugins

Copy Hailo GStreamer plugins:

```
cp <base_dir>/hailo/x86/gstreamer/libgsthailometa.so /usr/lib/x86_64-
linux-gnu/gstreamer-1.0/ && \
cp <base_dir>/hailo/x86/gstreamer/libgsthailo.so /usr/lib/x86_64-
linux-gnu/gstreamer-1.0/ && \
cp <base_dir>/hailo/x86/gstreamer/libhrt.so /usr/lib/x86_64-linux-
gnu/
```

And that's it, you are ready to go. Check our [Getting started](#) section.

Install GStreamer **VAAPI** plugins

1. Check if your platform matches VAAPI's requirements

- Hardware requirements
 - Hardware supported by i965 driver or iHD, such as
 - Intel Ironlake, Sandybridge, Ivybridge, Haswell, Broadwell, Skylake, etc. (HD Graphics)
 - Intel BayTrail, Braswell
 - Intel Poulsbo (US15W)
 - Intel Medfield or Cedar Trail
 - Hardware supported by AMD Radeonsi driver, such as the list below
 - AMD Carrizo, Bristol Ridge, Raven Ridge, Picasso, Renoir
 - AMD Tonga, Fiji, Polaris XX, Vega XX, Navi 1X
 - Other hardware supported by Mesa VA gallium state-tracker

(taken from <https://github.com/GStreamer/gstreamer-vaapi/blob/master/README>)

- Get hardware information about Intel Graphics Card with:

```
lshw -c video
```

Or use

```
lspci | grep VGA
```

Make sure that VGA compatible device with Intel drivers present.

In addition, you can list available devices in the following location:

```
ls /dev/dri
```

Output should look like :

```
card0 card1 renderD128 renderD129
```

2. Install Drivers

```
add-apt-repository ppa:oibaf/graphics-drivers
apt update
apt dist-upgrade

reboot
```

install VA-API drivers.

```
apt install va-driver-all
```

3. List Available video drivers

```
ls /usr/lib/x86_64-linux-gnu/dri | grep drv_video.so
```

Use vainfo (diagnostic tool for VA-API) to check that everything is loaded correctly without any warning or mistakes.

```
apt install vainfo
```

```
vainfo
```

4. Install gstreamer-vaapi

```
apt install gstreamer1.0-vaapi
```

Yocto

This section will guide through the integration of Hailo's Yocto layer's into your own Yocto environment.

Two layers are provided by Hailo, the first one is `meta-hailo` which is packed within the `HailoRT` release and the second one is `meta-hailo-tappas` which is packed within the `TAPPAS` release.

`meta-hailo-tappas` is a layer that based un-top of `meta-hailo` that adds `TAPPAS` recipes.

The layers were built and validated with the following Yocto releases:

- Warrior (kernel 4.19.35)
- Zeus (kernel 5.4.24)
- Dunfell (kernel 5.4.85)
- Gatesgarth (kernel 5.10.9)

Extraction

HailoRT

From the `HailoRT` release, untar `platform.tar.gz` without installing HailoRT locally. In this case the Yocto files are located under `platform/`. (you can read more in the `HailoRT` documentation)

set a `HAILORT_EXTERNALSRC` variable in your `conf/local.conf` file to point to the root directory of the `HailoRT` release you have extracted:

```
HAILORT_EXTERNALSRC = "<insert full HailoRT path here>"
IMAGE_INSTALL_append = "hailo-firmware libhailort hailo-pci
libgsthailo"
```

Tappas

From the `TAPPAS` release, untar `tappas_VERSION_linux_installer.zip`, the Yocto files are located under `yocto/`. the layer uses the unpacked release directory as an external source. In order for the build to work you will have to set a `TAPPAS_EXTERNALSRC` variable in your `conf/local.conf` file to point to the root directory of the `TAPPAS` release you have extracted:

Add the following to your image in your `conf/local.conf`:

```
TAPPAS_EXTERNALSRC = "<insert full TAPPAS path here>"
IMAGE_INSTALL_append = "libgsthailotools"
```

Build your image

Run bitbake and build your image. After the build successfully finished, burn the Image to your embedded device.

Copy the ARM apps

From within the x86 container, copy the `arm` apps into the embedded device using your preferred way of copying

Validating the integration's success

Make sure that the following conditions have been met on the target device:

- Running `hailortcli fw-control identify` prints the right configurations
- Running `gst-inspect-1.0 | grep hailo` returns hailo elements:

```
hailo: hailonet: hailonet element
hailodevicestats: hailodevicestats element
```

- Running `gst-inspect-1.0 | grep hailotools` returns hailotools elements:

```
hailotools: hailomuxer: Muxer pipe fitting
hailotools: hailofilter: Hailo postprocessing and drawing
element
```

- post-processes shared object files exists at `/usr/lib/hailo-post-processes`

Recipes

libgsthailo

Hailo's GStreamer plugin for running inference on the hailo8 chip. Depends on `libhailort` and GStreamer, the source files located under the HailoRT release. `platform/hailort/gstreamer`.

The recipe compiles and copies the `libgsthailo.so` file to `/usr/lib/gstreamer-1.0` on the target device's root file system, make it loadable by GStreamer as a plugin.

Note - this recipe requires a definition of `HAILO_EXTERNALSRC` in the `local.conf` - point to the HailoRT release.

libgsthailotools

Hailo's TAPPAS gstreamer elements and post-processes. Depends on `libgsthailo` and GStreamer. the source files located in the TAPPAS release under `core/hailo/gstreamer`. The recipe compiles with meson and copies the `libgsthailotools.so` file to `/usr/lib/gstreamer-1.0` and the post processes to `/usr/lib/hailo-post-processes` on the target device's root file system.

Note - this recipe requires a definition of `TAPPAS_EXTERNALSRC` in the `local.conf` - point to the TAPPAS release.

Cross-compile Hailo's GStreamer plugins

Overview

Hailo recommended method at the moment for cross-compilation is using Yocto SDK (aka Toolchain). We provide wrapper scripts whose only requirement is a Yocto toolchain to make this as easy as possible.

Preparations

In order to cross-compile you need to run **TAPPAS** container on a X86 machine and copy the Yocto toolchain to the container.

Toolchain

What is Toolchain?

A standard Toolchain consists of the following:

- Cross-Development Toolchain: This toolchain contains a compiler, debugger, and various miscellaneous tools.
- Libraries, Headers, and Symbols: The libraries, headers, and symbols are specific to the image (i.e. they match the image).
- Environment Setup Script: This *.sh file, once run, sets up the cross-development environment by defining variables and preparing for Toolchain use.

You can use the standard Toolchain to independently develop and test code that is destined to run on some target machine.

What should I add to my image?

For this example we would add the recipes to a NXP-IMX based image

Must Add

```
# GStreamer plugins
IMAGE_INSTALL_append += "
    imx-gst1.0-plugin
    gstreamer1.0-plugins-bad-videoparsersbad
    gstreamer1.0-plugins-good-video4linux2
    gstreamer1.0-plugins-base
"

# Opencv requirements for the hailo gstreamer plugin's postprocess
CORE_IMAGE_EXTRA_INSTALL += "
    libopencv-core-dev
    libopencv-highgui-dev
"
```

Nice to add

```
# GStreamer plugins
IMAGE_INSTALL_append += "
    gstreamer1.0-python
    gst-shark
    gst-instruments
"

# Enable trace hooks for GStreamer
PACKAGECONFIG_append_pn-gstreamer1.0 = "gst-tracer-hooks"
```

Prepare the Toolchain

In order to generate a Yocto toolchain, use this following command

```
# Generate the Toolchain
bitbake -c do_populate_sdk <image name>
```

Compress the toolchain to tar.gz

```
cd <BUILD_DIR>/tmp/deploy/sdk
touch toolchain.tar.gz
tar -czf toolchain.tar.gz --exclude=toolchain.tar.gz .
```

Copy the `toolchain.tar.gz` to the container using `docker cp`

```
docker cp toolchain.tar.gz
hailo_tappas_container:/local/workspace/tappas
```

Components

GstHailo

Compiling the `gst-hailo` component. This script, firstly unpack and installs the toolchain (if not installed already), and only after that, cross-compile.

Flags

```
$ ./cross_compile_gsthailo.py --help
usage: cross_compile_gsthailo.py [-h]
                                {aarch64,armv7l} {debug,release}
                                toolchain_tar_path
```

Cross-compile `gst-hailo`.

positional arguments:

```
{aarch64,armv7l}  Arch to compile to
{debug,release}   Build and compilation type
toolchain_tar_path Toolchain TAR path
```

optional arguments:

```
-h, --help          show this help message and exit
```

Example

An example for executing the script:

NOTE: In this example we assume that the toolchain is located under `toolchain-raw/hailo-dartmx8m-zeus-aarch64-toolchain.tar.gz`

```
$ ./cross_compile_gsthailo.py aarch64 debug toolchain-raw/hailo-
dartmx8m-zeus-aarch64-toolchain.tar.gz
INFO:cross_compile_gsthailo.py:Compiling gstreamer
INFO:cross_compile_gsthailo.py:extracting toolchain
INFO:cross_compile_gsthailo.py:installing toolchain
INFO:cross_compile_gsthailo.py:installing
/$TAPPAS_WORKSPACE/tools/cross-compiler/toolchain-raw/fsl-imx-
xwayland-glibc-x86_64-fsl-image-gui-aarch64-imx8mq-var-dart-
toolchain-5.4-zeus.sh
INFO:cross_compile_gsthailo.py:toolchain ready to use
(/local/workspace/tappas/tools/cross-compiler/toolchain)
INFO:cross_compile_gsthailo.py:using environment setup found in
/$TAPPAS_WORKSPACE/tools/cross-compiler/toolchain/environment-setup-
aarch64-poky-linux
INFO:cross_compile_gsthailo.py:Starting compilation...
INFO:cross_compile_gsthailo.py:Compilation done
```

A good practice is to check the output using `file` command

```
$ ls aarch64-gsthailo-build/
CMakeCache.txt  CMakeFiles  Makefile  cmake_install.cmake
libgsthailo.so
$ file aarch64-gsthailo-build/libgsthailo.so
aarch64-gsthailo-build/libgsthailo.so: ELF 64-bit LSB shared object,
ARM aarch64, version 1 (SYSV), dynamically linked,
BuildID[sha1]=e55c1655c113e99bb649dbb03c15b844142503ee, with
debug_info, not stripped
```

As you can see, the file is compatible to `aarch64` like we wanted to

GstHailoTools

This script cross-compiles `gst-hailo-tools`. This script, firstly unpack and installs the toolchain (if not installed already), and only after that, cross-compiles.

Flags

```
$ ./cross_compile_gsthailotools.py --help
usage: cross_compile_gsthailotools.py [-h]
                                     [--yocto-distribution
YOCTO_DISTRIBUTION]
                                     {aarch64,armv7l}
{debug,release}
```

```
toolchain_tar_path
```

```
Cross-compile gst-hailo.
```

```
positional arguments:
```

```
{aarch64,armv7l}      Arch to compile to
{debug,release}       Build and compilation type
toolchain_tar_path     Toolchain TAR path
```

```
optional arguments:
```

```
-h, --help            show this help message and exit
--yocto-distribution YOCTO_DISTRIBUTION
                        The name of the Yocto distribution to use
```

Example

Run the compilation script

NOTE: In this example we assume that the toolchain is located under `toolchain-raw/hailo-dartmx8m-zeus-aarch64-toolchain.tar.gz`

```
$ ./cross_compile_gsthailotools.py aarch64 debug toolchain
INFO:./cross_compile_gsthailotools.py:Building hailofilter plugin and
post processes
INFO:./cross_compile_gsthailotools.py:Running Meson build.
INFO:./cross_compile_gsthailotools.py:Running Ninja command.
```

Check the output directory

```
$ ls aarch64-gsthailotools-build/
build.ninja  compile_commands.json  config.h  libs  meson-info
meson-logs  meson-private  plugins
```

`libgsthailotools.so` is stored under `libs`

```
$ ls aarch64-gsthailotools-build/plugins/*.so
libgsthailotools.so
```

And the post-processes are stored under `plugins`

```
$ ls aarch64-gsthailotools-build/libs/*.so
libcenterpose_post.so  libmobilenet_ssd_post.so
libclassification.so  libsegmentation_draw.so
libdebug.so           libyolo_post.so
libdetection_draw.so
```

Copy the cross-compiled files

Find out where the **GStreamer** plugins are stored in your embedded device by running the following command:

```
gst-inspect-1.0 filesrc | grep Filename | awk '{print $2}' | xargs  
dirname
```

Copy **libgsthailo.so** + **libgsthailotools.so** to the path found out above. Copy the post-processes **so** files under **libs** to the embedded device under **/usr/lib/hailo-post-processes** (create the directory if it does not exist)

Run **gst-inspect-1.0 hailo** and **gst-inspect-1.0 hailotools** and make sure that no error raises

Further Reading

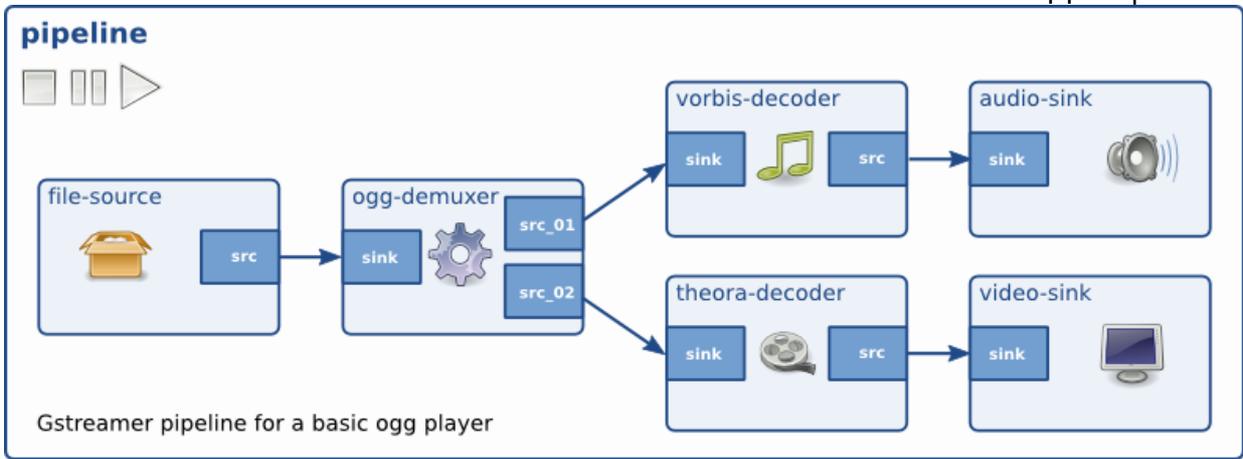
GStreamer Framework

GStreamer Principles

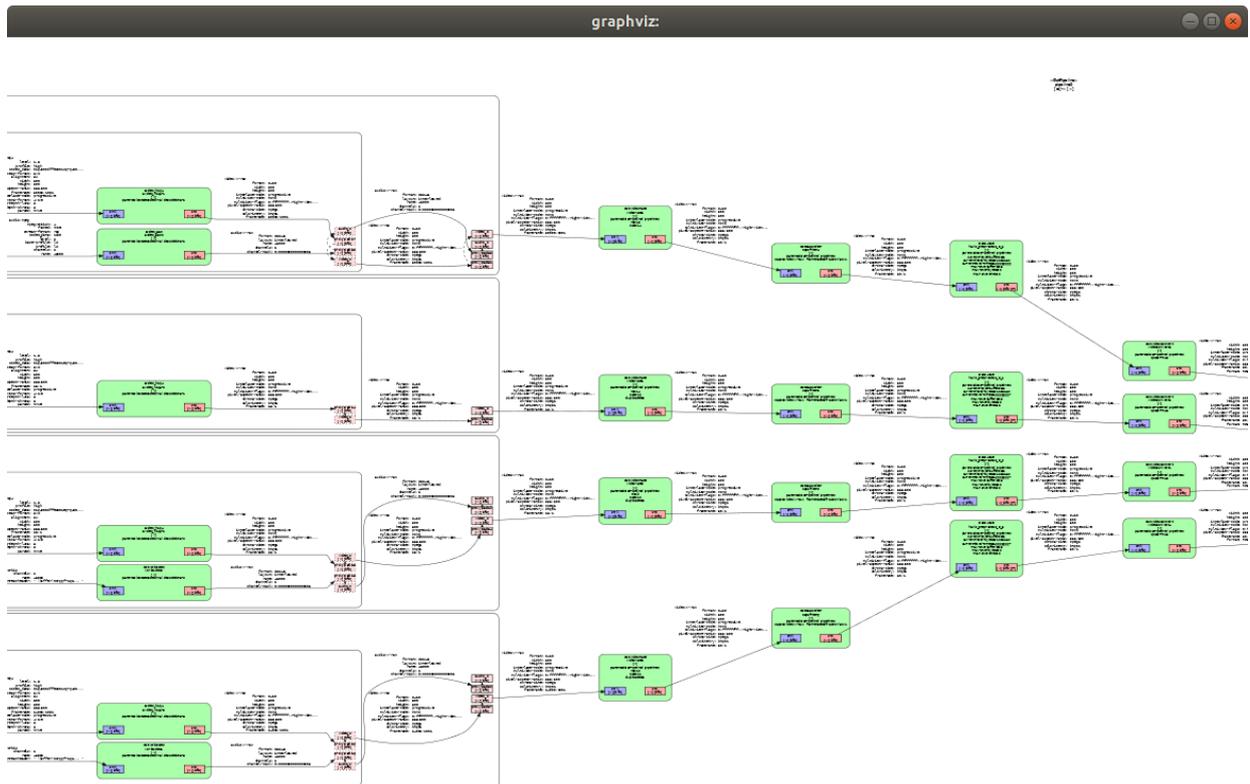
- Object-oriented - All GStreamer Objects can be extended using the GObject inheritance methods. All plugins are loaded dynamically and can be extended and upgraded independently.
- GStreamer adheres to GObject, the GLib 2.0 object model. A programmer familiar with GLib 2.0 or GTK+ will be comfortable with GStreamer.
- Extensible core plugins to encapsulate all common media streaming functionalities.
- Allow binary-only plugins - Plugins are shared libraries that are loaded at runtime.
- High performance
 - using GLib's GSlice allocator
 - ref-counting and copy on write minimize the usage of memcpy.
 - allowing hardware acceleration by using specialized plugins.

GStreamer Elements

- **Elements** - have one specific function for processing/ generating / consuming data. By chaining together several such elements, you create a pipeline that can do a specific task.
- **Pads** - are an element's input and output, where you can connect other elements. A pad can be viewed as a “plug” or “port” on an element where links may be made with other elements, and through which data can flow to or from those elements. Data types are negotiated between pads using a process called caps negotiation. Data types are described by GstCaps.
- **Bin** - A bin is a container for a collection of elements. Since bins are subclasses of elements themselves, you can mostly control a bin as if it were an element, thereby abstracting away a lot of complexity for your application. A pipeline is a top-level bin. It provides a bus for the application and manages the synchronization for its children.



Debugging with GstShark



GstShark is an open-source project from RidgeRun that provides benchmarks and profiling tools for GStreamer 1.7.1 (and above). It includes tracers for generating debug information plus some tools to analyze the debug information. GstShark provides easy to use and useful tracers, paired with analysis tools to enable straightforward optimizations.

GstShark leverages GStreamer's tracing hooks and open-source and standard tracing and plotting tools to simplify the process of understanding the bottlenecks in your pipeline.

The profiling tool provides 3 general features that can be used to debug the pipeline:

- **Console printouts** - At the most basic level, you should get printouts from the traces about the different measurements made. If you know what you are looking for, you may see it here at runtime.
- **Graphic visualization** - Shown above, gst-shark can generate a pipeline graph that shows how elements are connected and what caps were negotiated between them. This is a very convenient feature to look at the pipeline in a more comfortable way. The graph is generated at runtime so it is a great way to see and debug how elements were actually connected and what formats the data ended up in.
- **Gst-plot** - A suite of graph generating scripts are included in gst-shark that will plot different graphs for each tracer metric enabled. This is a powerful tool to visualize each metric that can be used for deeper debugging.

Install

Our docker image already contains GstShark! If you decide to not use our Docker image, our suggestion is to follow RidgeRun tutorial: [GstShark](#)

Bash shortcuts

As part of our creation of the Docker image, we copy some convenient shortcuts to GstShark:

```
vim ~/.bashrc
```

```
# set gstreamer debug
gst_set_debug() {
    export GST_SHARK_LOCATION=/tmp/profile
    export GST_DEBUG_DUMP_DOT_DIR=<PATH YOU WANT DUMP FILES>
    export GST_DEBUG="GST_TRACER:7"
    export
    GST_TRACERS="cpuusage;proctime;interlatency;scheduletime;buffer;bitrate;framerate;queuelevel;graphic"
    echo 'export
    GST_TRACERS="cpuusage;proctime;interlatency;scheduletime;buffer;bitrate;framerate;queuelevel;graphic"'
}

# set gstreamer to only show graphic
gst_set_graphic() {
    export GST_SHARK_LOCATION=/tmp/profile
    export GST_DEBUG_DUMP_DOT_DIR=<PATH YOU WANT DUMP FILES>
    export GST_DEBUG="GST_TRACER:7"
    export GST_TRACERS="graphic"
    echo 'export
    GST_TRACERS="cpuusage;proctime;interlatency;scheduletime;buffer;bitrate;framerate;queuelevel;graphic"'
}

# unset gstreamer debug
gst_unset_debug() {
    unset GST_TRACERS
}

# run gst-plot
gst_plot_debug() {
    cd <PATH TO GST-SHARK REPO FOLDER: gst-shark/scripts/graphics>
    ./gstshark-plot $GST_SHARK_LOCATION -p
    cd -
}
```

Note that we added 4 functions: two sets, an unset, and a plot function. The set functions enable gst-shark by setting environment variables, the chief of which is GST_TRACERS. This enables the different trace hooks in the pipeline. The available

tracers are listed in the echo command at the end of each set. You can enable any combination of the available tracers, just chain them together with a ';' (notice that the difference between `gst_set_debug` and `gst_set_graphic` is that `gst_set_debug` enables all tracers whereas `gst_set_graphic` only enables the graphic tracer that draws the pipeline graph). `GST_SHARK_LOCATION` and `GST_DEBUG_DUMP_DOT_DIR` set locations where the dump files are stored, the first sets where the tracer dumps are (used for `gst-plot`), and the latter where the dot file is saved (the graphic pipeline graph). `Unset` disables all tracers, and `gst_plot_debug` runs `gst-plot`.

Using GstShark

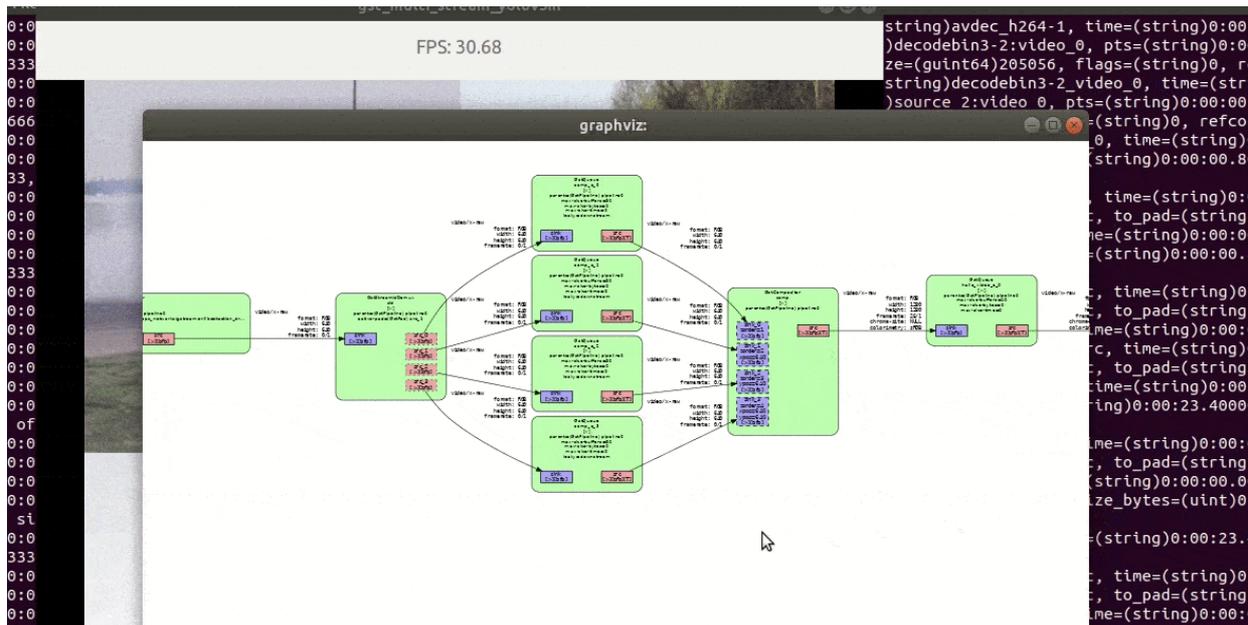
Let's say you have a gstreamer app you want to profile. Start by enabling `gst-shark`:

```

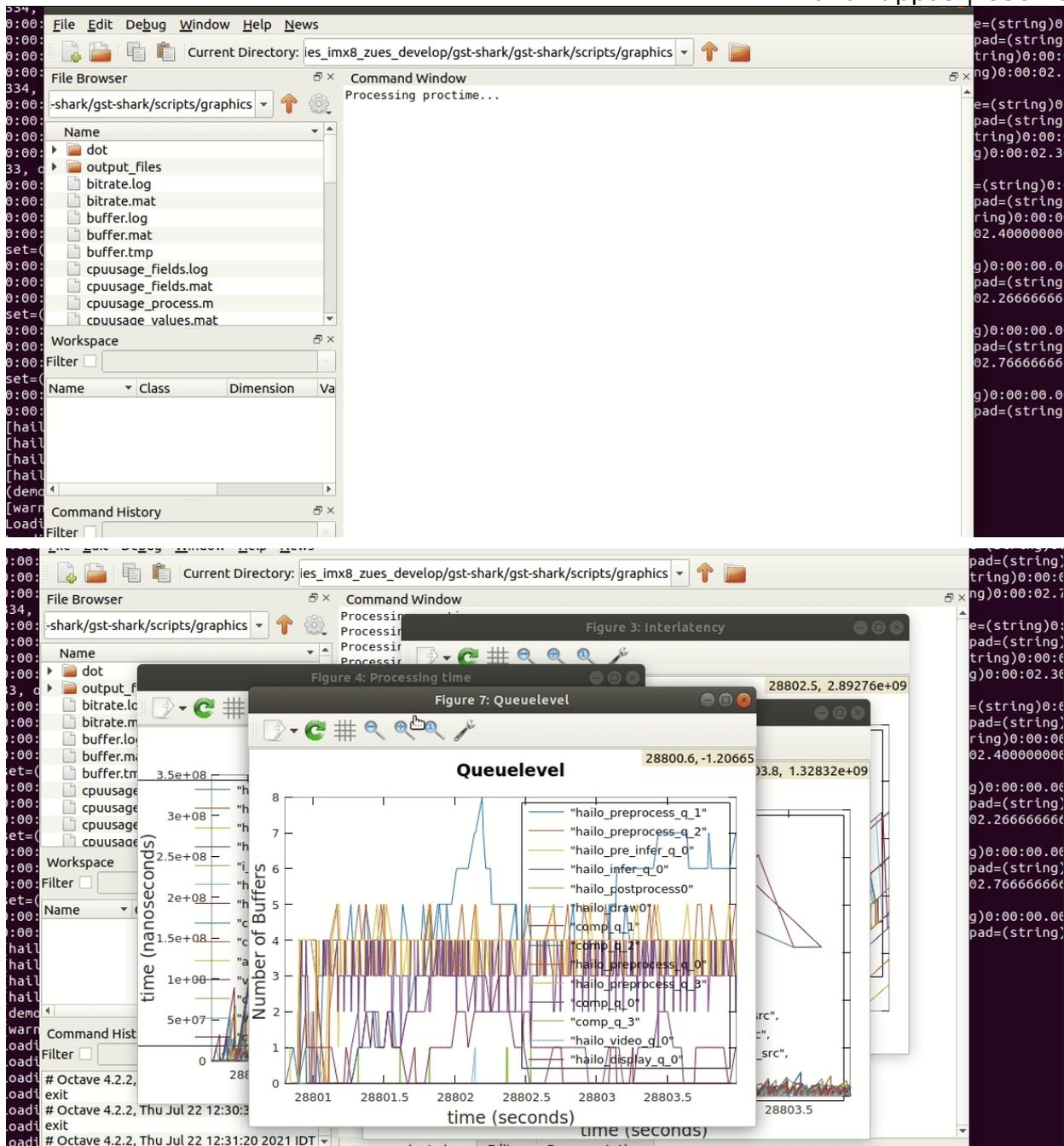
(demo_virtualenv) hailo@hail-168-lap:~/repos/demos$ gst_set_debug
export GST_TRACERS="cpuusage;proctime;interlatency;scheduletime;buffer;bitrate;framerate;queuelevel;graphic"
(demo_virtualenv) hailo@hail-168-lap:~/repos/demos$
    
```

Then just run your app. You will start seeing all kinds of tracer prints, and when the pipeline starts playing you should see the graphic plot load.

NOTE:: The graph will stay open as long as the pipeline runs. However if you have `GST_DEBUG_DUMP_DOT_DIR` set then afterwards a `.dot` file will be saved. [Click this file to reopen the graph.](#)



After you've run a gstreamer pipeline with tracers enabled, you can plot them using `gst-plot`. `gst-plot` will open an Octave window which will run the appropriate script to plot each tracer. Depending on how much data you have to plot this can take a while:



Each graph inspects a different metric of the pipeline, it is recommended to read more about what each one represents here:

- CPU Usage
- Processing Time
- InterLatency
- Schedule Time
- Buffer
- Bitrate
- Framerate
- Queue Level
- Graphic

Good luck, happy hunting.

Debugging with Gst-Instruments

gst-instruments is a set of performance profiling and data flow inspection tools for GStreamer pipelines.

- **gst-top-1.0** at the start of the pipeline will analyze and profile the run. (gst-top-1.0 gst-launch-1.0 ! audiotestsrc ! autovideosink)
- **gst-report-1.0** - generates performance report for input trace file.
- **gst-report-1.0 --dot gst-top.gsttracee | dot -Tsvg > perf.svg** - generates performance graph in DOT format.

[Read more in gst-instruments github page](#)

Writing Your Own Postprocess

Overview

If you want to add a network to the Tappas that is not already supported, then you will likely need to implement a new postprocess and drawing filter. Fortunately with the use of the [hailofilter](#), you don't need to create any new gstreamer elements, just provide the shared object file (.so) that applies your filter!

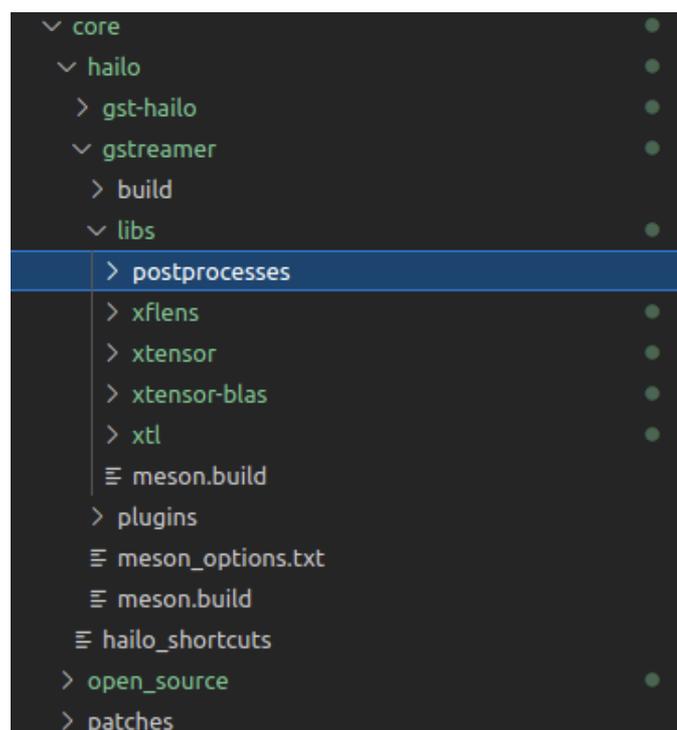
In this guide we will go over how to create such an so and what mechanisms/structures are available to you as you create your postprocess.

Getting Started

Where are all the files?

To begin your postprocess writing journey, it is good to understand where you can find all the relevant source files that already exist, and how to add your own.

From the `/hailo/` directory, you can find the `core/` folder. Inside this `core/` directory are a few subdirectories that host different types of sources files. The one we are interested in is `hailo/`. Here you will find source files for all kinds of Hailo tools, such as the `hailofilter`, the different metas provided, and the source files for the postprocesses of the networks that were already provided in the Tappas. Inside this directory is one titled `gstreamer/`, and inside that are two folders of interest: `libs/` and `plugins/`. The former contains the source code for all the postprocess and drawing functions packaged in the Tappas, while the latter contains source code for the `hailofilter`, `hailomuxer`, and the different metas/classes available. This guide will mostly focus on this `core/hailo/gstreamer/` directory, as it has everything we need to create and compile a new .so! You can take a moment to peruse around, when you are ready to continue enter the `postprocesses/` directory:



Preparing the Header File: Default Filter Function

We can create our new postprocess here in the `postprocesses/` folder. Create a new header file named `my_post.hpp`.

In the first line we want to import a useful class to our postprocess, so add the following include:

```
#include "hailo_frame.hpp"
```

A `hailo_frame` is a class that represents an image buffer that has a tensor or a number of tensors attached. You can find the class definition in `plugins/metadata/hailo_frame.hpp`. This class can also hold detection results, and will be the return value of your `.so`! Let's wrap up the header file by adding a function prototype for our filter, your whole header file should look like:

```
#include "hailo_frame.hpp"

G_BEGIN_DECLS
void filter(HailoFramePtr hailo_frame);
G_END_DECLS
```

Yes really, that's it! The `hailofilter` element does not expect much, just that the above `filter` function be provided. We will discuss [adding multiple filters in the same .so](#) later. Note that the `filter` function takes a `HailoFramePtr` as a parameter; this will provide you with the `hailo_frame` of each passing image.

Implementing filter()

Let's start implementing the actual filter so that you can see how to access and work with tensors. Start by creating a new file called `my_post.cpp`. Open it and include the following:

```
#include <gst/gst.h>
#include <iostream>
#include "my_post.hpp"
#include "hailo_detection.hpp"
```

The `<gst/gst.h>` include provides the gstreamer framework api, the `<iostream>` will allow us to print to the console, the `"my_post.hpp"` includes the header file we just wrote, and the `"hailo_detection.hpp"` will provide access to the `DetectionObject` struct which represents any detection object that we infer. You can find the source for `DetectionObject` in `plugins/metadata/hailo_detection.hpp`, later we will use it to attach detected objects to the frame.

For now add the following implementation for `filter()` so that we have a working postprocess we can test:

```
// Default filter function
void filter(HailoFramePtr hailo_frame)
```

```
{  
  std::cout << "My first postprocess!" << std::endl;  
}
```

That should be enough to try compiling and running a pipeline! Next we will see how to add our postprocess to the Meson project so that it compiles.

Compiling and Running

Building with Meson

Meson is an open source build system that puts an emphasis on speed and ease of use. **GStreamer** uses **meson** for all subprojects to generate build instructions to be executed by **ninja**, another build system focused solely on speed that requires a higher level build system (ie: meson) to generate its input files.

Like GStreamer, Tappas also uses Meson, and compiling new projects requires adjusting the `meson.build` files. Here we will discuss how to add yours.

In the `gstreamer/libs/` path you will find a `meson.build`, open it and add the following entry for our postprocess:

```
#####
# MY POST SOURCES
#####
my_post_sources = [
  'postprocesses/my_post.cpp',
]

my_post_lib = shared_library('my_post',
  my_post_sources,
  cpp_args : hailo_lib_args,
  link_args: hailo_ld_args,
  include_directories: project_inc,
  dependencies : plugin_deps + hailo_deps,
  gnu_symbol_visibility : 'default',
)
```

This should give meson all the information it needs to compile our postprocess. In short, we are providing paths to cpp compilers, linked libraries, included directories, and dependencies. Where are all these path variables coming from? Great question: from the parent meson project, you can read that meson file to see what packages and directories are available at [core/hailo/gstreamer/meson.build](#).

Compiling the .so

You should now be ready to compile your postprocess. To help streamline this process we have gone ahead and provided a script that handles most of the work. You can find this script at [/hailo/docker/scripts/install_hailo_gstreamer.sh](#). This script includes some flags that allow you do more specific operations, but the only one you need right now is `--skip-hailort`. This omits recompiling the **Hailort** package (where the **hailonet** element comes from). Since our postprocess does not affect or change **Hailort**, then we can save time by not recompiling. From the `/hailo/` folder you can run:

```
./docker/scripts/install_hailo_gstreamer.sh --skip-hailort
```

```

root@hailo_tappas:/hailo# ./docker/scripts/install_hailo_gstreamer.sh --skip-hailort
/hailo/core/hailo/gstreamer /hailo
The Meson build system
Version: 0.60.0
Source dir: /hailo/core/hailo/gstreamer
Build dir: /hailo/core/hailo/gstreamer/build
Build type: native build
Project name: gst-hailo-tools
Project version: 0.1.0
C compiler for the host machine: cc (gcc 7.5.0 "cc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0")
C linker for the host machine: cc ld.bfd 2.30
C++ compiler for the host machine: c++ (gcc 7.5.0 "c++ (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0")
C++ linker for the host machine: c++ ld.bfd 2.30
Host machine cpu family: x86_64
Host machine cpu: x86_64
Configuring config.h using configuration
Found pkg-config: /usr/bin/pkg-config (0.29.1)
Run-time dependency gstreamer-1.0 found: YES 1.14.5
Run-time dependency gstreamer-base-1.0 found: YES 1.14.5
Run-time dependency gstreamer-video-1.0 found: YES 1.14.5
Run-time dependency blas found: YES 3.7.1
Run-time dependency lapack found: YES 3.7.1
Library dl found: YES
Run-time dependency opencv4 found: YES 4.5.2
Build targets in project: 11

gst-hailo-tools 0.1.0

User defined options
  buildtype: release
  ldargs   : -L/usr/lib/x86_64-linux-gnu/gstreamer-1.0/,-lgsthailo
  libargs  : -I/hailo/core/hailo/gst-hailo,-std=c++14

```

If all goes well you should see some happy green **YES**, and our `.so` should appear in `apps/gstreamer/x86/lib/!`

```

v apps/x86
  > classification
  > detection
  > face_detection
  > face_landmarks
  v gstreamer
    ≡ libcenterpose_post.so
    ≡ libclassification.so
    ≡ libdebug.so
    ≡ libdetection_draw.so
    ≡ libface_detection_post.so
    ≡ libface_landmarks_draw.so
    ≡ libface_landmarks_post.so
    ≡ libgsthailo.so
    ≡ libgsthailometa.so
    ≡ libmobilenet_ssd_post.so
    ≡ libmy_post.so
    ≡ libsegmentation_draw.so
    ≡ libyolo_post.so

```

Running the `.so`

Congratulations! You've compiled your first postprocess! Now you are ready to run the postprocess and see the results. Since it is still so generic, we can try it. Run this test pipeline in your terminal to see if it works:

```
gst-launch-1.0 videotestsrc ! hailofilter so-  
path=$TAPPAS_WORKSPACE/apps/gstreamer/x86/libs/libmy_post.so !  
fakesink
```

See in the above pipeline that we gave the `hailofilter` the path to `libmy_post.so` in the `so-path` property. So now every time a buffer is received in that `hailofilter`'s sink pad, it calls the `filter()` function in `libmy_post.so`. The resulting app should just show the original video while our chosen text "My first postprocess!" prints in the console:

```
os=false debug=false ! videoconvert ! fpsdtspr  
overlay=false  
Setting pipeline to PAUSED ...  
[HailoRT] [warning] Desc page size value (102  
Pipeline is PREROLLING ...  
Redistribute latency...  
Redistribute latency...  
My first postprocess!  
Pipeline is PREROLLED ...  
Setting pipeline to PLAYING ...  
New clock: GstSystemClock  
My first postprocess!  
My first postprocess!
```

Filter Basics

Working with Tensors

Printing statements on every buffer is cool and all, but we would like a postprocess that can actually do operations on inference tensors. Let's take a look at how we can do that.

Head back to `my_post.cpp` and swap our print statement with the following:

```
// Get the output layers from the hailo frame.
std::vector<HailoTensorPtr> tensors = hailo_frame->get_tensors();
```

The `hailo_frame` has two ways of providing the output tensors of a network: via the `get_tensors()` and `get_tensors_by_name()` functions. The first (which we used here) returns an `std::vector` of `HailoTensorPtr` objects. These are an `std::shared_ptr` to a `HailoTensor`: a class that represents an output tensor of a network. `HailoTensor` holds all kinds of important tensor metadata besides the data itself; such as the width, height, number of channels, and even quantization parameters. You can see the full implementation for this class at [plugins/metadata/hailo_tensor.hpp](#).

`get_tensors_by_name()` also returns a `HailoTensorPtr` for each output layer, but this time as an `std::map` that pairs the `output layer names` with their corresponding `HailoTensorPtr`. This can be convenient if you want to perform operations on specific layers whose names you know in advanced.

So now we have a vector of `HailoTensorPtr` objects, lets get some information out of one, add the following lines to our `filter()` function:

```
// Get the first output tensor
HailoTensorPtr first_tensor = tensors[0];
std::cout << "Tensor: " << first_tensor->name;
std::cout << " has width: " << first_tensor->width;
std::cout << " height: " << first_tensor->height;
std::cout << " channels: " << first_tensor->channels << std::endl;
```

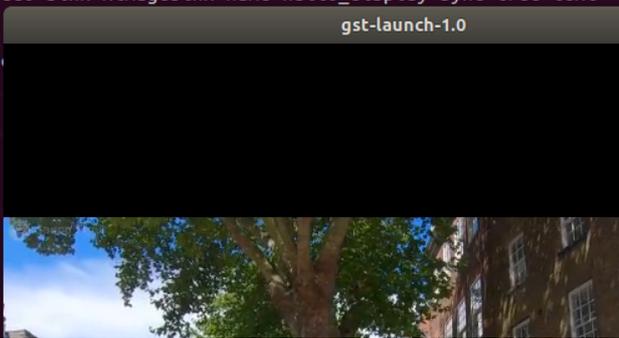
Recompile with the same [script we used earlier](#). Run a test pipeline, and this time see actual parameters of the tensor printed out:

```
gst-launch-1.0 filesrc
location=$TAPPAS_WORKSPACE/apps/gstreamer/x86/detection/detection.mp4
name=src_0 ! decodebin ! videoscale ! video/x-raw, pixel-aspect-
ratio=1/1 ! videoconvert ! queue ! hailonet hef-
path=$TAPPAS_WORKSPACE/apps/gstreamer/x86/detection/yolov5m.hef
debug=False is-active=true qos=false batch-size=8 ! queue leaky=no
max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! hailofilter
so-path=$TAPPAS_WORKSPACE/apps/gstreamer/x86/libs/libmy_post.so
qos=false debug=False ! videoconvert ! fpsdisplaysink video-
sink=ximagesink name=hailo_display sync=true text-overlay=false
```

```

root@hailo_tappas:/hailo# gst-launch-1.0 filesrc location=/hailo/apps/x86/detection/detection.mp4 name=src_0
! decodebin ! videoscale ! video/x-raw, pixel-aspect-ratio=1/1 ! videoconvert ! queue ! hailonet hef-path=/ha
ilo/apps/x86/detection/yolov5m.hef debug=False is-active=true qos=false batch-size=8 ! queue leaky=no max-siz
e-buffers=30 max-size-bytes=0 max-size-time=0 ! hailofilter so-path=/hailo/apps/x86/gstreamer/libmy_post.so q
os=false debug=False ! videoconvert ! fpsdisplaysink video-sink=ximagesink name=hailo_display sync=true text-
overlay=false
Setting pipeline to PAUSED ...
[HailoRT] [warning] Desc page size value (1024) is not
Pipeline is PREROLLING ...
Redistribute latency...
Redistribute latency...
Tensor: conv93 has width: 20 height: 20 channels: 255
Pipeline is PREROLLED ...
Setting pipeline to PLAYING ...
New clock: GstSystemClock
Tensor: conv93 has width: 20 height: 20 channels: 255
Tensor: conv93 has width: 20 height: 20 channels: 255
Tensor: conv93 has width: 20 height: 20 channels: 255
Tensor: conv93 has width: 20 height: 20 channels: 255
Tensor: conv93 has width: 20 height: 20 channels: 255

```



With a **HailoTensor** in hand, you have everything you need to perform your postprocess operations. You can access the actual tensor values from the **HailoTensor** with:

```
auto first_tensor_data = first_tensor->data;
```

Keep in mind that at this point the data is of type **uint8_t**. You will have to dequantize the tensor to a **float** if you want the full precision. Luckily the quantization parameters (scale and zero point) are also accessible through the **HailoTensor**.

Attaching Detection Objects to the Frame

Now that you know how to create a basic filter and access your inference tensor, let's take a look at how to add a detection object to your **hailo_frame**.

Remove the prints from the **filter()** function and replace them with the following function call:

```
std::vector<DetectionObject> detections = demo_detection_objects();
```

Here we are calling a function **demo_detection_objects()** that will return some detection objects. Copy the following function definition into your **my_post.cpp**:

```

std::vector<DetectionObject> demo_detection_objects()
{
    std::vector<DetectionObject> objects; // The detection objects we
    will eventually return
    DetectionObject first_detection = DetectionObject(0.2, 0.2, 0.2,
0.2, 0.99, 1);
    DetectionObject second_detection = DetectionObject(0.6, 0.6, 0.2,
0.2, 0.89, 1);
    objects.push_back(first_detection);
    objects.push_back(second_detection);

    return std::move(objects);
}

```

In this function we are creating two instances of a **DetectionObject** and pushing them into a vector that we return. Note that when creating a **DetectionObject**, we give a

series of parameters. The expected parameters are as follows:

```
DetectionObject(float xmin, float ymin, float height, float width,
float confidence, int class_id, int dataset_id=0)
```

NOTE: It is assumed that the `xmin`, `ymin`, `width`, and `height` given are a **percentage of the image size** (meaning, if the box is **half** as wide as the width of the image, then `width=0.5`). This protects the pipeline's ability to resize buffers without compromising the correct relative size of the detection boxes.

Looking back at the demo function we just introduced, we are adding two instances of `DetectionObject`: `first_detection` and `second_detection`. According to the parameters we saw, `first_detection` has an `xmin` 20% along the x axis, and a `ymin` 20% down the y axis. The `width` and `height` are also 20% of the image. The last two parameters, `confidence` and `class_id`, show that this instance has a 99% `confidence` for `class_id` 1. What label does `class_id` 1 imply? That depends on your **dataset!** Notice that the last parameter of `DetectionObject` is a `dataset_id` with default 0. The provided detection drawer, [which we will look at later](#), uses the `dataset_id` along with the `class_id` to look up the proper label within different datasets. Right now a few datasets are provided out of the box in the Tappas, you can find them in the file [libs/postprocess/common/labels.hpp](#). The default dataset is `COCO`, so a `class_id` of 1 is a `person`.

Now that we have a couple of `DetectionObject` in hand, let's add them to the original `hailo_frame`. There is a helper function we need in the [libs/postprocess/common/common.hpp](#) file, so include it into `my_post.cpp` now:

```
#include "common/common.hpp"
```

This file will no doubt have other features you will find useful, so it is recommended to keep the file handy.

With the include in place, let's add the following function call to the end of the `filter()` function:

```
// Update the frame with the found detections.
common::update_frame(hailo_frame, detections);
```

This function takes a `hailo_frame` and a `DetectionObject` vector, then adds each `DetectionObject` to the `hailo_frame`. Now that our detections have been added to the `hailo_frame` and the postprocess is done, we can clear the `tensors` vector to release the memory (add the command `tensors.clear()`; to the end of the `filter()` function to do so). To recap, our whole `my_post.cpp` should look like this:

```
#include <gst/gst.h>
#include <iostream>
#include "my_post.hpp"
#include "hailo_detection.hpp"
#include "common/common.hpp"
```

```

std::vector<DetectionObject> demo_detection_objects()
{
    std::vector<DetectionObject> objects; // The detection objects we
will eventually return
    DetectionObject first_detection = DetectionObject(0.2, 0.2, 0.2,
0.2, 0.99, 1);
    DetectionObject second_detection = DetectionObject(0.6, 0.6, 0.2,
0.2, 0.89, 1);
    objects.push_back(first_detection);
    objects.push_back(second_detection);

    return std::move(objects);
}

// Default filter function
void filter(HailoFramePtr hailo_frame)
{
    // Get the output layers from the hailo frame.
    std::vector<HailoTensorPtr> tensors = hailo_frame->get_tensors();
    std::vector<DetectionObject> detections = demo_detection_objects();

    // Update the frame with the found detections.
    common::update_frame(hailo_frame, detections);
    tensors.clear();
}

```

Recompile again and run the test pipeline, if all goes well then you should see the original video run with no problems! But we still don't see any detections? Don't worry, they are attached to each buffer, however no filter is drawing them onto the image itself. To see how our detection boxes can be drawn, read on to [Next Steps: Drawing](#).

Next Steps

Drawing

At this point we have a working postprocess that attaches two detection boxes to each passing buffer. But how do we get the GStreamer pipeline to draw those boxes onto the image? With another `hailofilter` of course! Just as we were able to add a `hailofilter` with an `.so` that added detection boxes, we can also add a **second** `hailofilter` to the pipeline that draws those boxes onto the image.

The Tappas already comes with an `.so` that knows how to draw attached `DetectionObject` instances: `libdetection_draw.so`. You can find the source for this `.so` at `libs/postprocesses/detection_draw.cpp`, inside are good examples of not only how to extract/draw `DetectionObject` instances from a `hailo_frame`, but also how to extract landmarks and draw them.

Since our postprocess attaches detections as `DetectionObject` instances, we can immediately make use of the `libdetection_draw.so`. All we need to do is insert another `hailofilter` element into our pipeline after the first, but this time with the path to `libdetection_draw.so` given in the `so-path` property instead of the path to `libmy_post.so`:

```
gst-launch-1.0 filesrc
location=$TAPPAS_WORKSPACE/apps/gstreamer/x86/detection/detection.mp4
name=src_0 ! decodebin ! videoscale ! video/x-raw, pixel-aspect-
ratio=1/1 ! videoconvert ! queue ! hailonet hef-
path=$TAPPAS_WORKSPACE/apps/gstreamer/x86/detection/yolov5m.hef
debug=False is-active=true qos=false batch-size=8 ! queue leaky=no
max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! hailofilter
so-path=$TAPPAS_WORKSPACE/apps/gstreamer/x86/libs/libmy_post.so
qos=false debug=False ! queue leaky=no max-size-buffers=30 max-size-
bytes=0 max-size-time=0 ! hailofilter so-
path=$TAPPAS_WORKSPACE/apps/gstreamer/x86/libs/libdetection_draw.so
qos=false debug=False ! videoconvert ! fpsdisplaysink video-
sink=ximagesink name=hailo_display sync=true text-overlay=false
```

Run the expanded pipeline above to see the original video, but this time with the two detection boxes we added!



As expected, both boxes are labeled as **person**, and each is shown with the assigned **confidence**. Obviously, the two boxes don't move or match any object in the video; this is because we hardcoded their values for the sake of this tutorial. It is up to you to extract the correct numbers from the inferred tensor of your network, as you can see among the postprocesses already implemented in the Tappas each network can be different. We hope that this guide gives you a strong starting point on your development journey, good luck!

Multiple Filters in One .so

While the **hailofilter** always calls on a **filter()** function by default, you can provide the element access to other functions in your **.so** to call instead. This may be of interest if you are developing a postprocess that applies to multiple networks, but each network needs slightly different starting parameters (in the Tappas case, multiple flavors of the **Yolo detection network are handled via the same .so**).

So how do you do it? Simply by declaring the extra functions in the header file, then pointing the **hailofilter** to that function via the **function-name** property.

Let's look at the yolo networks as an example, open up

[libs/postprocesses/yolo_postprocess.hpp](#) to see what functions are made available to the **hailofilter**:

```
#ifndef _HAILO_YOLO_POST_HPP_
#define _HAILO_YOLO_POST_HPP_
#include "hailo_frame.hpp"

G_BEGIN_DECLS
void filter(HailoFramePtr hailo_frame);
void yolov3(HailoFramePtr hailo_frame);
void yolov4(HailoFramePtr hailo_frame);
void yolov5(HailoFramePtr hailo_frame);
void yolov5_no_persons(HailoFramePtr hailo_frame);
G_END_DECLS
#endif
```

Any of the functions declared here can be given as a `function-name` property to the `hailofilter` element. Consider this pipeline for running the `Yolov5` network:

```
gst-launch-1.0 filesrc
location=$TAPPAS_WORKSPACE/apps/gstreamer/x86/detection/detection.mp4
name=src_0 ! decodebin ! videoscale ! video/x-raw, pixel-aspect-
ratio=1/1 ! videoconvert ! queue leaky=no max-size-buffers=30 max-
size-bytes=0 max-size-time=0 ! hailonet hef-
path=$TAPPAS_WORKSPACE/apps/gstreamer/x86/detection/yolov5m.hef
debug=False is-active=true qos=false batch-size=1 ! queue leaky=no
max-size-buffers=30 max-size-bytes=0 max-size-time=0 ! hailofilter
function-name=yolov5 so-
path=$TAPPAS_WORKSPACE/apps/gstreamer/x86/libs//libyolo_post.so
qos=false debug=False ! queue leaky=no max-size-buffers=30 max-size-
bytes=0 max-size-time=0 ! hailofilter so-
path=$TAPPAS_WORKSPACE/apps/gstreamer/x86/libs//libdetection_draw.so
qos=false debug=False ! videoconvert ! fpsdisplaysink video-
sink=ximagesink name=hailo_display sync=true text-overlay=false
```

The first `hailofilter` above that performs the post-process points to `libyolo_post.so` in the `so-path`, but it also includes the property `function-name=yolov5`. This lets the `hailofilter` know that instead of the default `filter()` function it should call on the `yolov5` function instead.

Writing Your Own Python Postprocess

Overview

If you want to add a network to the TAPPAS that is not already supported, then you will likely need to implement a new postprocess. Fortunately with the use of the [hailopython](#), you don't need to create any new GStreamer elements, just provide a Python module that applies your post-processing!

In this guide we will go over how to create such a python module and what mechanisms/structures are available to you as you create your postprocess.

Getting Started

hailopython requires a module and a Python function.

Python module template

Here is a template for a Python module for the hailopython element.

```
# Import hailo module
import hailo
# Import GStreamer
import gi
gi.require_version('Gst', '1.0')
from gi.repository import Gst

# Create 'run' function, that accepts 2 parameters - a Gst.Buffer
# object and a hailo.HailoROI object.
# `run` is default function name if no name is provided
def run(buffer: Gst.Buffer, roi: hailo.HailoROI):
    print("My first Python postprocess!")
```

To call it, create a pipeline with hailopython:

```
gst-launch-1.0 videotestsrc ! hailopython
module=$PATH_TO_MODULE/my_module.py ! autovideosink
```

Extracting the tensors

One of the first steps in each postprocess is to get the output tensors. This can be done by one of `roi.get_tensor(tensor_name)` or `roi.get_tensors()`

```
def run(buffer: Gst.Buffer, roi: hailo.HailoROI):
    for tensor in roi.get_tensors():
        print(tensor.name())
        my_tensor = roi.get_tensor("output_layer_name")
        print(f"shape is
{my_tensor.height()}X{my_tensor.width()}X{my_tensor.features()}")
```

After doing that you might want to convert this object of type HailoTensor to a numpy array on which you can perform post-processing operations more conveniently. This is a fairly simple step, you just use `np.array` on a given HailoTensor.

Notice that `np.array` has a parameter that determines whether we copy the memory or using the original buffer.

```
def run(buffer: Gst.Buffer, roi: hailo.HailoROI):
    my_tensor = roi.get_tensor("output_layer_name")
    # To create a numpy array with new memory
    my_array = np.array(my_tensor)
    # To create a numpy array with original memory
    my_array = np.array(my_tensor, copy=False)
```

There are some other methods in HailoTensor, you are welcome to perform `dir(my_tensor)` or `help(my_tensor)`.

Adding results

After you process your net results and come up with post-processed results, you can use them however you want. Here we will show you how to add them to the original image in order to draw them later by `hailooverlay` element. In order to add post-processed result to the original image - use the `roi.add_object` method. This method adds a HailoObject object to our image. There are several types of objects that are currently supported: `hailo.HailoClassification` - Classification of the image. `hailo.HailoDetection` - Detection in the image. `hailo.HailoLandmarks` - Landmarks in the image.

You can create one of these objects and then add it with the `roi.add_object` method.

```
def run(buffer: Gst.Buffer, roi: hailo.HailoROI):
    classification = hailo.HailoClassification(type='animal',
    index=1, label='horse', confidence=0.67)
    # You can also create a classification without class id (index).
    classification = hailo.HailoClassification(type='animal',
    label='horse', confidence=0.67)
    roi.add_object(classification)
```

You can also add objects to detections:

```
def run(buffer: Gst.Buffer, roi: hailo.HailoROI):
    # Adds a person detection in the bottom right quarter of the
    # image. (normalized only)
    person_bbox = hailo.HailoBBox(xmin=0.5, ymin=0.5, width=0.5,
    height=0.5)
    person = hailo.HailoDetection(bbox=person_bbox, label='person',
    confidence=0.97)
    roi.add_object(person)
    # Now, Adds a face to the person, at the top of the person.
    # (normalized only)
    face_bbox = hailo.HailoBBox(xmin=0.0, ymin=0.0, width=1,
```

```

height=0.2)
    face = hailo.HailoDetection(bbox=face_bbox, label='face',
confidence=0.84)
    person.add_detection(face)
    # No need to add the face to the roi because it is already in the
    person that is in the roi.

```

Next Steps

Drawing

In order to draw your postprocessed results on the original image use the `hailooverlay` element. It is already familiar with our `HailoObject` types and knows how to draw classifications, detections, and landmarks onto the image.

```

gst-launch-1.0 filesrc
location=$TAPPAS_WORKSPACE/apps/gstreamer/x86/detection/detection.mp4
name=src_0 ! decodebin \
! videoscale ! video/x-raw, pixel-aspect-ratio=1/1 ! videoconvert !
queue leaky=no max-size-buffers=30 \
max-size-bytes=0 max-size-time=0 ! hailonet hef-
path=$TAPPAS_WORKSPACE/apps/gstreamer/x86/detection/yolov5m.hef \
debug=False is-active=true qos=false batch-size=8 ! queue leaky=no
max-size-buffers=30 max-size-bytes=0 \
max-size-time=0 ! hailopython
module=$TAPPAS_WORKSPACE/apps/gstreamer/x86/detection/my_module.py
qos=false ! queue \
leaky=no max-size-buffers=30 max-size-bytes=0 max-size-time=0 !
hailooverlay qos=false ! videoconvert ! \
fpsdisplaysink video-sink=ximagesink name=hailo_display sync=true
text-overlay=false

```

This is the standard detection pipeline with a python module for post-processing.

Multiple functions in one Python module

There is an option to write several post-process functions in the same module. In order to run each of them you just need to add the `function` property to the `hailopython` element:

```

import hailo
import gi
gi.require_version('Gst', '1.0')
from gi.repository import Gst

def post_process_function(buffer: Gst.Buffer, roi: hailo.HailoROI):
    print("My first Python postprocess!")

def other_post_function(buffer: Gst.Buffer, roi: hailo.HailoROI):
    print("Other Python postprocess!")

```

```
gst-launch-1.0 videotestsrc ! hailopython  
module=$PATH_TO_MODULE/my_module.py function=other_post_function !  
autovideosink
```