Application Note

Building a Driver and Occupancy Monitoring System with an RGB-IR Camera

Jianzhong Xu and Reese Grimsley

ABSTRACT

This application note details the implementation of a vision pipeline using an RGB-IR sensor on the AM62A platform. This document includes a reference design for driver and occupancy monitoring system with video telephony, utilizing the OX05B1S sensor and the AM62A SK EVM.

Table of Contents

List of Figures

List of Tables

Trademarks

Arm® and Cortex® are registered trademarks of Arm Limited.
All trademarks are the property of their respective owners.

1 Introduction

With the growing demand for image sensing in both visible and infrared light, RGB-IR sensors, which capture both RGB and IR images with a single camera, are becoming increasingly popular. Effectively processing and leveraging RGB-IR data is essential for various artificial intelligence applications, including driver monitoring and occupancy monitoring (also known as cabin monitoring), robotics, security surveillance, and smart home systems. The AM62A SoC is an excellent choice for building such intelligent systems, as detailed in [1].

This application note focuses on implementing a vision processing pipeline on the AM62A SoC, from an RGB-IR camera to the AI engine, for a driver and occupancy monitoring system (DMS or OMS) with support for video calls or video recording. This document begins by outlining the essential hardware and software components, followed by a reference design and benchmarks.

2 Building Blocks of an RGB-IR Vision Pipeline

As demonstrated in [1], the AM62A offers a range of unique features that make the device an excellent choice for building applications that require both RGB and IR image data. The following sections provide details on the relevant components for building RGB-IR image processing pipelines.

2.1 CSI Receiver

The camera serial interface (CSI) receiver (Rx) on the AM62A complies with MIPI CSI v1.2 and supports up to 16 virtual channels. This receiver contains a DMA wrapper that transfers the captured image data to memory via DMA. This wrapper creates multiple DMA contexts, each dedicated to storing data from one virtual channel to the memory.

The CSI Rx driver in the Processor SDK Linux (or EdgeAI SDK) for AM62A [2] is compliant to the V4L2 framework. The driver creates a V4L2 video device node for each DMA context which is used to capture image data from each virtual channel and store to the memory. User-space applications can then access the captured image data via the video node associated with each virtual channel.

An RGB-IR sensor typically uses a 4x4 mosaic pattern containing both color and IR pixels. An external IR illuminator can be used to provide adequate IR lighting and can be periodically turned on and off, allowing the sensor to capture either color (RGB) dominant or IR dominant images. The sensor can be synchronized with the illuminator to send RGB dominant images through one virtual channel when the illuminator is off and IR dominant images through another channel when the illuminator is on. To handle these images separately, the CSI Rx driver creates two video device nodes: one for receiving RGB data (when the IR illuminator is off) and another for receiving IR data (when the IR illuminator is on).

2.2 Image Signal Processor

The Image Signal Processor (ISP), also known as the Vision Pre-processing Accelerator (VPAC), on the AM62A device provides essential vision processing functions at the pixel level. The ISP consists of three sub-modules: the Vision Imaging Sub-System (VISS), the Multi-Scalar (MSC), and the Lens Distortion Correction (LDC).

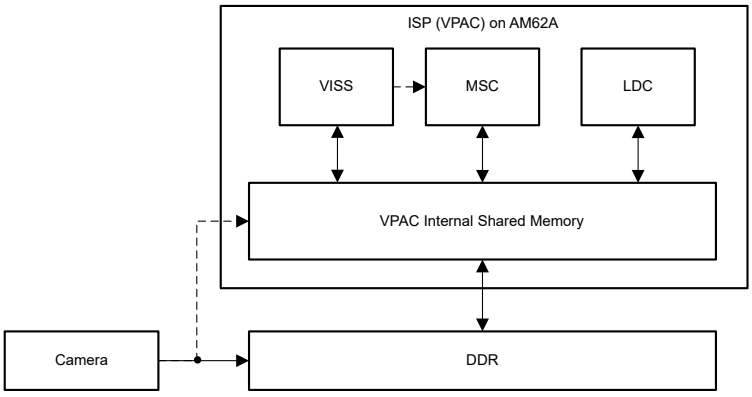

Figure 2-1. AM62A Image Signal Processor Overview

The VISS sub-module processes the raw images and produces demosaiced color images and IR images. The VISS is sometimes also referred as ISP. The MSC sub-module can produce up to 10 downscaled or cropped images. The LDC sub-module performs perspective and geometric transforms and is mostly used to correct lens distortion.

The sub-modules of the AM62A ISP (VPAC) can operate in either memory-to-memory mode or on-the-fly mode:

- **Memory-to-memory mode**: The sub-module reads data from memory and writes processed data back to memory.
- **On-the-fly mode**: The camera data is sent directly to the Vision Imaging Sub-System (VISS), and the VISS output is sent directly to the Multi-Scalar (MSC) without being stored in memory first.

The VISS processes raw image data in a 4x4 RGB-IR pattern and produces two output streams: one RGB stream and one IR stream, as illustrated in Figure 2-2.

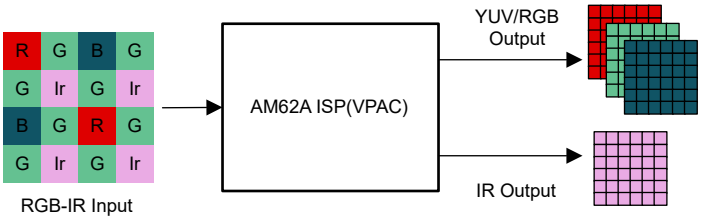

Figure 2-2. RGB-IR Processing by AM62A ISP

The 4x4 image data first undergoes raw pixel processing, such as decompanding, WDR merge, defective pixel correction, and lens shading correction. After raw pixel processing, the image data is split into two paths, the RGB processing path and the IR processing path.

- **RGB Processing Path**: This path involves typical color processing functions such as demosaicing, auto white balancing, color correction, and noise filtering. Additionally, IR contamination is removed from the RGB data.
- **IR Processing Path**: This path is simpler and involves only upsampling and tone mapping.

For applications such as driver monitoring that only need IR images, the IR output is utilized. For applications like video calls or recording that require color images, the YUV or RGB output from the ISP is used. The occupancy (or cabin) monitoring can also use the RGB output to enhance the detection accuracy.

2.3 Video Processing Unit

The AM62A SoC includes a hardware accelerator for encoding and decoding, known as the Video Processing Unit (VPU), which supports H.264 and H.265 encoding and decoding. The VPU can packetize the RGB stream produced by the ISP for use in video calls or recording. For more details about the VPU applications and performance, please refer to [3].

2.4 TI Deep Learning Acceleration

Deep learning and neural networks are an increasingly popular strategy to extract meaning and information from imagery and other data. TI's AM6xA and TDA4x SoC's use an in-house developed hardware IP, the C7xMMA, with TI Deep Learning (TIDL) software to accelerate neural network inference.

The C7xMMA is a tightly coupled C7x SIMD DSP and matrix multiplier accelerator (MMA). The architecture is highly effective for Convolution Neural Networks (CNNs), which are a common type of neural network used for vision processing. In most CNNs, matrix multiplication and similar operations compose at least 98% of the total operations. In this way, MMA's have a large impact on the computational efficiency of neural network acceleration for vision tasks, such as object detection, pixel-level segmentation, and key-point detection.

Figure 2-3 depicts a general development flow for TIDL on AM6xA and TDA4x processors. This development flow can be entered from multiple points. TI provides GUI-based and command line-based tools that enable users to:

- Bring data (BYOD) and train a TI model
- Bring pretrained model (BYOM) of a custom architecture

- Evaluate a pre-trained and pre-optimized model from TI's Model Zoo.

Where each of these development actions feeds into the next. Developers compile a model for the target SoC and can test accuracy on PC before deploying to the target. The compilation tools and accelerator are invoked through open source runtimes like Tensorflow Lite, ONNX Runtime or TVM. These runtimes provide a familiar API and allow unaccelerated layers to run on the Arm® A cores, easing usability for a broad host of models. Each of these open source runtimes (OSRT) leverage the TIDL runtime (TIDL_RT) under-the-hood.

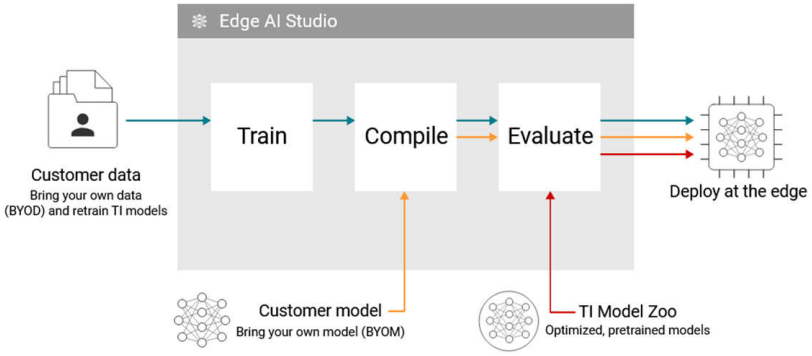

Figure 2-3. TI Deep Learning Development Flow

2.5 GStreamer and TIOVX Frameworks

Two vision frameworks are supported on AM6xA devices:GStreamer and OpenVX. To be precise, TI has implemented and conforms to the OpenVX standard; this implementation is designated TI OpenVX (TIOVX). These frameworks enable on-chip hardware accelerators, like C7xMMA and ISP. The frameworks handle IPC and data management under the hood to reduce application-level complexity.

GStreamer (GST) is an Open Source, Linux-based framework for multimedia pipelines. Pipelines are constructed of plugins that implement a function, like capturing an image, changing data formats, scaling sizes, compressing, or writing to a file. Many community plugins are available, but the best performance is attained by using TI-provided plugins that leverage on-chip accelerators. These are provided within the Edge AI SDK and source code is available for modifying or extending TI plugins. GST is particularly effective for building and testing pipelines quickly – in addition to C++, Python, and other language support, GST pipelines can be run straight from command line. For example:

v4l2src device=/dev/video-usb-cam0 ! image/jpeg, width=1280, height=720 ! jpegdec ! video/x-raw, format=NV12 ! kmssink driver-name=tidss sync=true

The above pipeline consists of individual plugins separated by '!' delimiters. This pipeline reads images from a USB camera with v4l2, decodes the JPEG-encoded images, converts the frames' encoding to NV12, and pushes frames to the display subsystem (DSS) through Linux's KMS/DRM interface for visualization on a monitor. Pipelines can be large and complex, yet still be run from command line with no coding otherwise.

TIOVX is a lower-level framework for building vision pipelines on heterogeneous SoCs. Analogous to GST, pipelines are composed of nodes within an acyclic graph, where each node runs a function on a target core. For example, processing a raw frame on the ISP or running a neural network on C7xMMA. TIOVX applications are written in C/C++, and require more knowledge of the SOC. Under the hood, GST uses TIOVX to communicate with hardware accelerators. GST requires more interaction with Linux to pass control signals from plugin, whereas TIOVX allows cores to communicate more directly. TIOVX is portable between operating systems, including Linux and QNX, and is appropriate for functional safety (FuSa) certified applications. This makes TIOVX an excellent choice in automotive and other ASIL and SIL-rated use cases. The TIOVX framework is slightly more efficient than GST in terms of interrupt usage; however, frame rate, accelerator utilization, and DDR bandwidth are near parity between GST and TIOVX.

Note that TIOVX applications restrict the user to TIDL_RT for deep learning models; open source runtimes (OSRT) like ONNX Runtime cannot be used at runtime through TIOVX. One role of the OSRT is to supply a

backup implementation for layers that TIDL does not support. Therefore, a TIOVX-only application's direct use of TIDL_RT implies that all layers in the neural network must be supported by TIDL.

In most cases, GST is sufficient for building vision-processing applications on TI AM6xA SoCs running Linux. If functional safety and another operating system are required, TIOVX is a more appropriate choice.

3 Performance Considerations and Benchmarking Tools

When building a vision application, the following performance metrics need to be considered and benchmarked:

- **End-to-end latency**: The latency between capturing an image and generating the analytics result must be as low as possible to allow for timely decision-making and responsive actions.
- **Video throughput (frames per second)**: Images must be captured and processed at the desired frame rate without frame drops.
- **CPU load**: The load on general-purpose CPU cores (A53 in the case of AM62A) due to the vision pipeline must be minimal, as all image processing is done on hardware accelerators.
- **DDR utilization**: The DDR read and write operations by the vision pipeline must leave enough bandwidth for other system tasks.
- **Hardware accelerators (HWA) load (ISP, VPU, C7x/MMA)**: HWAs are dedicated to specific functionalities and cannot be used for other purposes. The HWAs can be utilized up to 100% by the vision pipeline with some margin.

The EdgeAI SDK for AM62A provides several tools to benchmark these performance metrics:

- **Perf_stats tool [5]**: Measures the load on CPU cores and HWAs, as well as DDR utilization.
- **GStreamer debug trace**: By setting the environment variable GST_DEBUG_FILE, GStreamer debug messages can be redirected to a file. An EdgeAI SDK script (/opt/edgeai-gst-apps/scripts/gst_tracers/parse_gst_tracers.py) can process these messages and estimate the processing time for each element in the GStreamer pipeline.
- **GStreamer plugin *fpsdisplaysink*** : Displays the throughput of the pipeline in frames per second (fps).
- **Custom GStreamer plugin *tiperfoverlay*** : Projects the CPU loads, DDR utilization, HWA loads, and fps on the display or prints them on the terminal console.

4 Reference Design

This section describes a vision pipeline reference design for driver and occupancy monitoring with support for video calls or recording, using a single RGB-IR image sensor. This section covers the hardware setup, software design, tools, and example applications with benchmarking results.

4.1 Camera Module

The LI-OX05B1S-MIPI-137H camera module from Leopard Imaging was used to capture RGB-IR images. The camera was connected to the AM62A SK EVM using two adapter cables: FAW-1233-03 and LI-FPC22-IPEX-PI, as shown in Figure 4-1.

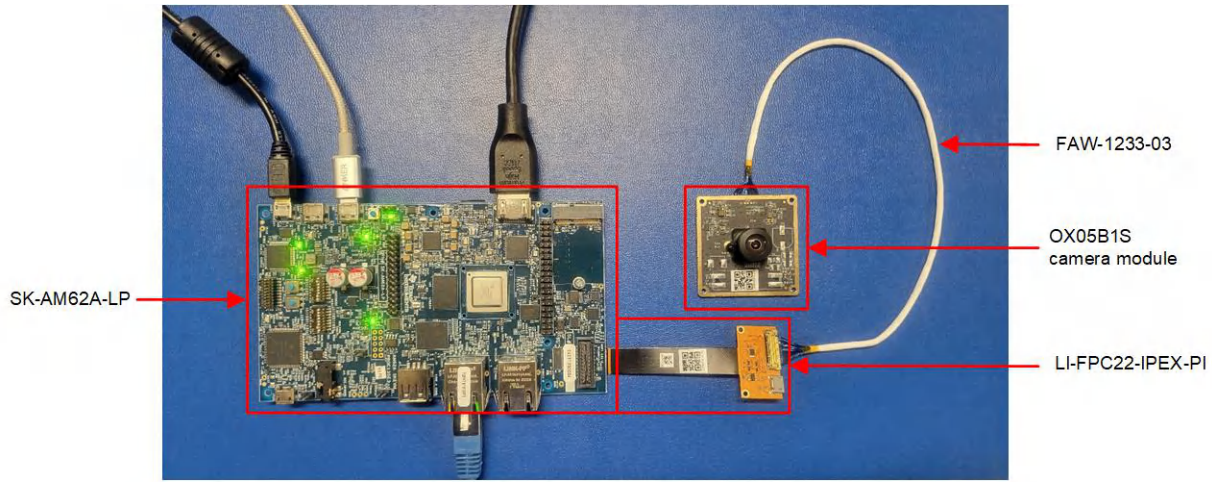

Figure 4-1. Reference Design Hardware Setup

4.2 Sensor Driver

The driver for the OX05B1S sensor is provided through the Processor SDK Linux (or EdgeAI SDK) for AM62A, located in source file drivers/media/i2c/ox05b1s.c within the linux-kernel. The driver configures the sensor to operate in two modes:

- **Mode A (virtual channel 0)**: This mode can be synchronized with turning on an external IR illuminator, allowing the sensor to capture and send IR-dominant streams.
- **Mode B (virtual channel 1)**: This mode can be synchronized with turning off the external IR illuminator, enabling the sensor to capture and send RGB-dominant streams.

These two modes can alternate, with each mode running for a different number of frames to achieve the desired frame rate for each stream. For instance, both streams can run at 30 fps, or one stream can run at 15 fps while the other runs at 45 fps.

4.3 CSI-2 Rx Driver

The V4L2-compliant CSI-2 Rx driver, included in the Processor SDK, receives image data from the sensor and differentiates the RGB-dominant and IR-dominant streams based on the virtual channel numbers. The driver then stores each stream in DDR using dedicated DMA contexts. Two video device nodes are created in user space, allowing applications to retrieve the RGB-dominant and IR-dominant image data respectively. The data flow from the sensor to the video device nodes is modeled by the V4L2 framework as the media device topology. This topology can be displayed in text by the *media-ctl --print* command and visualized using the Linux dot utility.

Figure 4-2 shows the media device topology in text, based on SDK 10.1. This topology contains two streams from the sensor (ox05b 4-0036) to CSI-2 Rx (cdns_csi2rx.30101000.csi-bridge), then to the DMA wrapper (30102000.ticsi2rx). The DMA wrapper uses two DMA contexts to transfer the image data to DDR, with each context linked to a device node (/dev/video3 and /dev/video4). User-space applications can then access the raw image data from these two device nodes.

```
root@am62axx-evm:~# media-ctl -p

Device topology
- entity 1: 30102000.ticsi2rx (7 pads, 7 links, 2 routes)
            type V4L2 subdev subtype Unknown flags 0
            device node name /dev/v4l-subdev0
        routes:
                0/0 -> 1/0 [ACTIVE]
                0/1 -> 2/0 [ACTIVE]
        pad0: Sink
                [stream:0 fmt:SBGGI10_1X10/2592x1944 field:none colorspace:srgb]
                [stream:1 fmt:SBGGI10_1X10/2592x1944 field:none colorspace:srgb]
                <- "cdns_csi2rx.30101000.csi-bridge":1 [ENABLED,IMMUTABLE]
        pad1: Source
                [stream:0 fmt:SBGGI10_1X10/2592x1944 field:none colorspace:srgb]
                -> "30102000.ticsi2rx context 0":0 [ENABLED,IMMUTABLE]
        pad2: Source
                [stream:0 fmt:SBGGI10_1X10/2592x1944 field:none colorspace:srgb]
                -> "30102000.ticsi2rx context 1":0 [ENABLED,IMMUTABLE]

- entity 9: cdns_csi2rx.30101000.csi-bridge (5 pads, 2 links, 2 routes)
            type V4L2 subdev subtype Unknown flags 0
            device node name /dev/v4l-subdev1
        routes:
                0/0 -> 1/0 [ACTIVE]
                0/1 -> 1/1 [ACTIVE]
        pad0: Sink
                [stream:0 fmt:SBGGI10_1X10/2592x1944 field:none colorspace:srgb]
                [stream:1 fmt:SBGGI10_1X10/2592x1944 field:none colorspace:srgb]
                <- "ox05b 4-0036":0 [ENABLED,IMMUTABLE]
        pad1: Source
                [stream:0 fmt:SBGGI10_1X10/2592x1944 field:none colorspace:srgb]
                [stream:1 fmt:SBGGI10_1X10/2592x1944 field:none colorspace:srgb]
                -> "30102000.ticsi2rx":0 [ENABLED,IMMUTABLE]

- entity 15: ox05b 4-0036 (1 pad, 1 link, 2 routes)
             type V4L2 subdev subtype Sensor flags 0
             device node name /dev/v4l-subdev2
        routes:
                0/0 -> 0/0 [ACTIVE]
                0/0 -> 0/1 [ACTIVE]
        pad0: Source
                [stream:0 fmt:SBGGI10_1X10/2592x1944@1/60 field:none colorspace:srgb]
                [stream:1 fmt:SBGGI10_1X10/2592x1944@1/60 field:none colorspace:srgb]
                -> "cdns_csi2rx.30101000.csi-bridge":0 [ENABLED,IMMUTABLE]

- entity 21: 30102000.ticsi2rx context 0 (1 pad, 1 link)
             type Node subtype V4L flags 0
             device node name /dev/video3
        pad0: Sink
                <- "30102000.ticsi2rx":1 [ENABLED,IMMUTABLE]

- entity 27: 30102000.ticsi2rx context 1 (1 pad, 1 link)
             type Node subtype V4L flags 0
             device node name /dev/video4
        pad0: Sink
                <- "30102000.ticsi2rx":2 [ENABLED,IMMUTABLE]
```

2 DMA contexts at the CSI Rx driver

2 streams from the sensor

Video device node for IR stream

Video device node for RGB stream

Figure 4-2. Media Device Toplogy in Text

The media device topology can also be visualized using the Linux dot utility. Run the following command on EVM to generate a dot file:

```
root@am62axx-evm:~# media-ctl --print-dot > media.dot
```

Then run the following command on a Linux PC to generate a png image file, as shown below.

```
$ dot -Tpng media-top.dot -o media-top.png
```

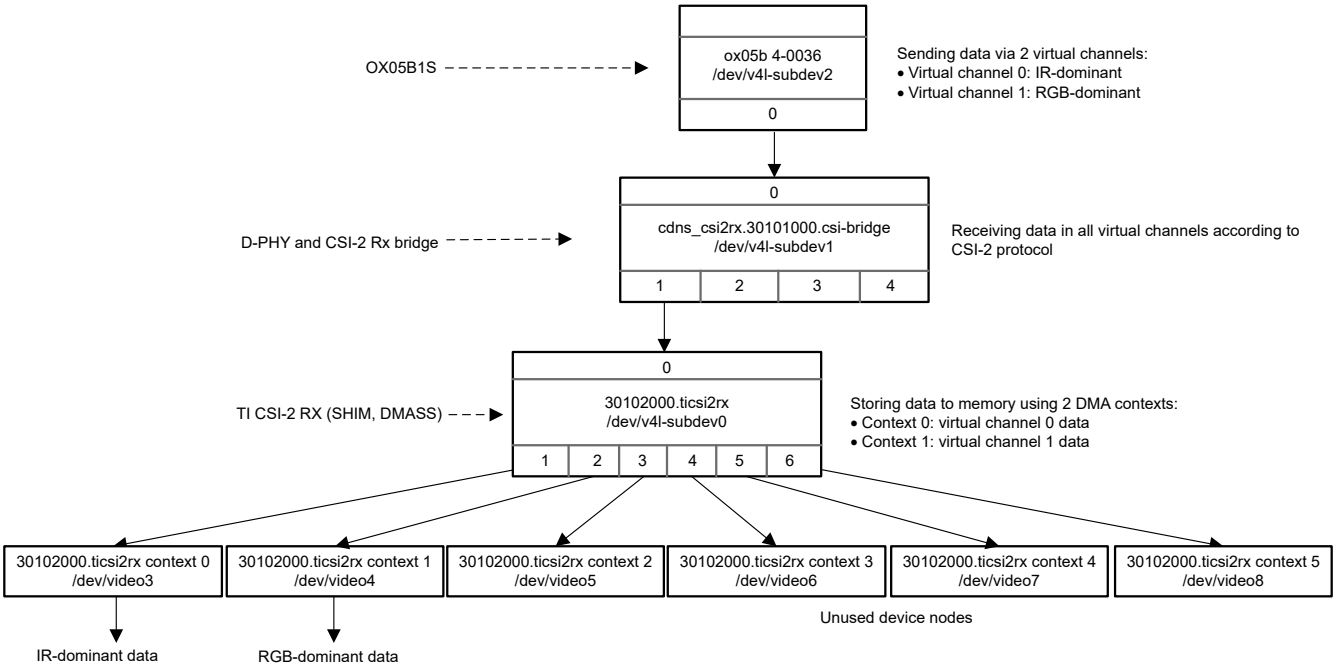

Figure 4-3. Media Device Topology Visualization

The OX05B1S driver in the SDK configures the sensor to transmit IR-dominant data through virtual channel 0 and RGB-dominant data through virtual channel 1. DMA context 0 is used to store the data from virtual channel 0. As a result, the first video device node created by the CSI2 Rx driver (/dev/video3 in the example above) is used to receive IR-dominant data. Similarly, the second device node (/dev/video4 in the example above) is used to receive RGB-dominant data.

4.4 Image Processing

The OX05B1S is a raw sensor, and the captured raw images are processed by the ISP. To achieve the best image quality for RGB applications, the ISP must be tuned. For details on how to tune the ISP for a specific sensor, refer to the AM6xA ISP Tuning Guide [4]. Pre-tuned ISP configuration binaries for the OX05B1S are provided by the SDK under /opt/imaging/ox05b1s/linear on the target.

The AM62A ISP produces two output frames for each RGB-IR input frame: one YUV frame and one IR frame. In this reference design, the OX05B1S is configured to alternate between RGB-dominant and IR-dominant streams, each at 30 fps. In the vision pipeline implemented by GStreamer or TIOVX, only the RGB output frames are used for the RGB-dominant stream, while the IR frames are discarded, and vice versa for the IR-dominant stream.

While processing the RGB-dominant stream, the ISP also generates statistics that can be used for auto exposure and gain control (AE) and auto white balancing (AWB). The Processor SDK provides AE and AWB algorithms (also referred to as "2A") that are used in this reference design to adjust the sensor exposure and gain and white balancing gains. The exposure and gain adjustments are sent to the sensor, while the white balancing gains are provided to the ISP.

4.5 Deep Learning for Driver and Occupancy Monitoring

Driver monitoring systems (DMS) and occupancy monitoring systems (OMS) are typically separate processing paths from an image analysis and deep-learning perspective. In both cases, the IR frames from the RBG-IR camera are typically used. This way, the vehicle interior can be sufficiently illuminated with non-visible light to allow accurate monitoring while preserving the driver's nighttime vision.

The images are therefore analyzed as single-channel, grayscale images. Rather than providing 3-channel RGB data, a single channel is being processed, thereby reducing processing requirements and DDR bandwidth. However, analyzing single-channel (for example, grayscale) images implies that the neural network models are also trained on such data, whereas typical models are trained for 3-channel RGB. TIDL is fully capable of processing an arbitrary number of input channels and resolutions.

Deep Learning for Driver Monitoring

Driver monitoring must determine when the driver is or is not attentive to the road. This can nominally be identified as fatigue or distractions. In both cases, the driver's head position, gaze and eyes are of primary interest. Eye and eyelid movements occur rapidly, so analysis must be at an appropriate frame rate, often around 30 FPS. Local regulatory standards like Euro NCAP can alter this requirement. Simpler DMS systems can use head-pose only, but this is unable to handle difficult *lizard* type scenarios, in which the driver's head position points toward the road but the eyes are looking elsewhere, such as a cell phone.

A typical flow for DMS can look like the following in Figure 4-4. Please note there are several viable approaches and techniques. For example, some systems use head-pose detection instead of gaze detection to determine driver distractedness.

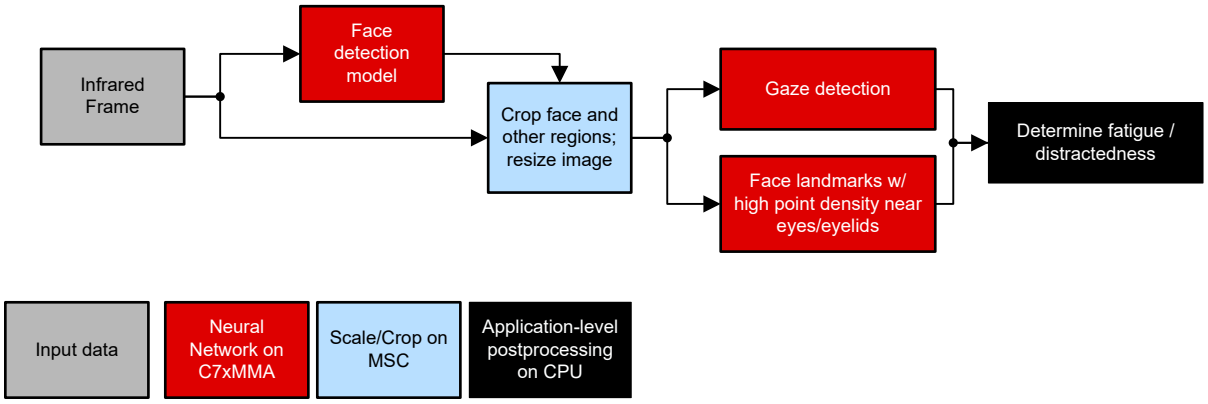

Figure 4-4. Driver Monitoring Image Analysis Flow

The deep learning models provide information about the driver's attentiveness. However, some degree of postprocessing across frames is required. For example, a single frame showing closed-eyes can be a blink, but several in a row can indicate drowsiness or microsleep. Similarly, looking away from the road in front of the vehicle can indicate distraction or be a necessary driving activity, like looking in the direction of an upcoming turn. In this way, deep learning algorithms for DMS must provide sufficiently high frame rate to enable such tracking from across multiple frames.

Deep Learning for Occupancy Monitoring

Occupancy monitoring collects information about which seats are occupied within the vehicle and how seatbelts are being utilized. This changes less quickly than a driver's head position and eye movements, so the frame rate requirements are lower; 1 to 5 FPS are acceptable in most circumstances. However, the region of interest is larger, typically the entire vehicle's interior as opposed to the driver's seat only. Therefore, models must run at higher resolution and have higher processing requirements. OMS are responsible for checking which seats are occupied, if seatbelts are used correctly, and how airbags must be deployed in case of a crash.

An example data flow for occupancy monitoring is shown in Figure 4-5. A single image can be processed by multiple stages of neural networks to determine how many passengers are present, how the passengers are positioned, and how this affects airbag deployments.

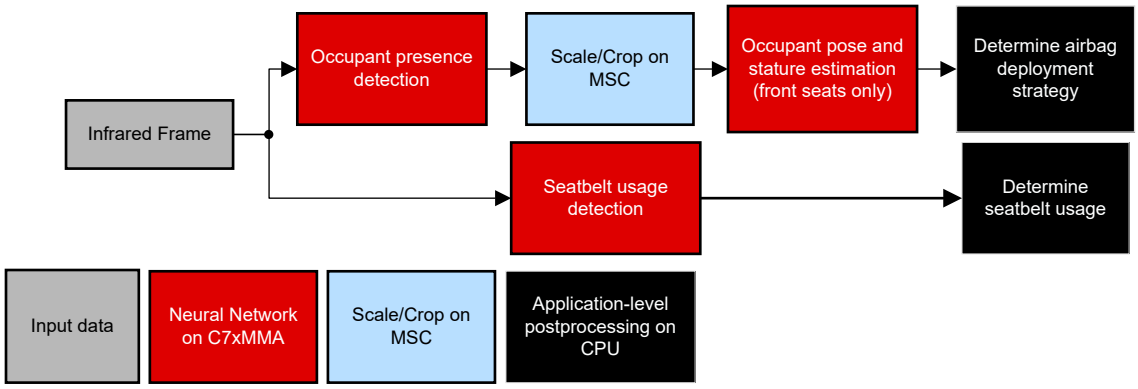

Figure 4-5. Occupancy Monitoring Image Analysis Flow

Operating Multiple Models with TIDL

TIDL allows multiple deep learning models to be loaded at the same time. So long as the models' weights and configurations fit into the available persistent DDR space, the application can initialize and run multiple models in any order. No special handling is required for concurrent calls to TIDL.

Several of the models described in this report have different frame rate requirements and different levels of complexity. TIDL enables prioritization and preemption of models to fit such applications. For example, the DMS models, which require high frame rate, can be run at a higher priority to make sure the models run quickly enough with respect to the inter-frame latency. Then OMS models, which have lower FPS requirement and can be larger, can run at lower priority to take advantage of unused cycles between DMS frames. Developers need to analyze the runtime latencies of the models and make sure there is sufficient headroom to run each model at the required frame rate and within a latency bound.

4.6 Reference Code and Applications

Find example code for the applications in the next section on the Texas Instruments GitHub [6]. Tests and benchmarks for this application note were run on the AM62A74 processor on the SK-AM62A-LP starter-kit board. The device is running the Linux SDK for Edge AI applications, version 10.00.00.08.

5 Application Examples and Benchmarking

This section presents a few examples using the LI-OX05B1S-MIPI-137H camera and the benchmarking results for each example.

5.1 Application 1: Single-stream Capture and Visualization with GST

This is a simple example of streaming from the camera to a display and visualizing the RGB data. This example demonstrates how the performance metrics listed in Section 3 were benchmarked. The components of the GStreamer pipeline of this example are shown below.

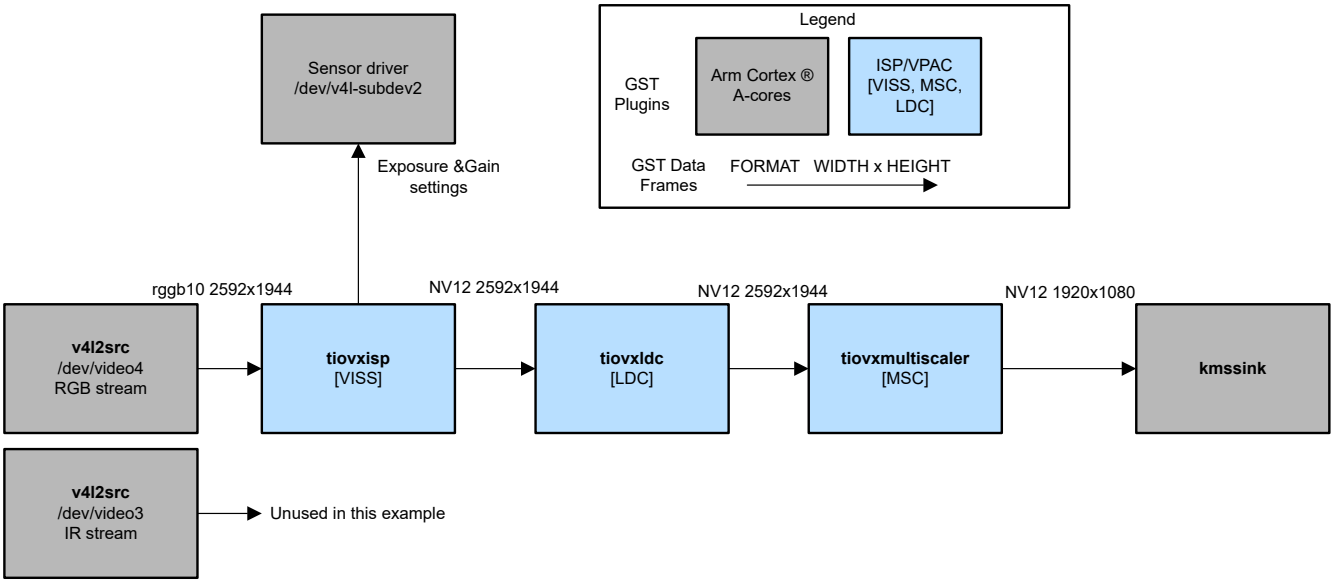

Figure 5-1. RGB Stream Capture And Display With Gstreamer

The corresponding GST command can be found in the GitHub repo [6]. To benchmark the performance metrics, this command was run in one terminal console and the perf_stats tool [5] was run in another terminal console simultaneously.

- Pipeline throughput (fps) was printed out in the first console, as shown in Figure 5-2. The throughput was also shown on the display.
- CPU load, HWA load, and DDR utilization were printed out in the second console and shown on the display as well. These performance metrics were constantly updated while the GST command was running. Figure 5-3 shows a screen capture of a single update.
- After stopping the GST pipeline, run "/opt/edgeai-gst-apps/scripts/gst_tracers/parse_gst_tracers.py /run/trace.log" to generate the latency measurements of each element in the pipeline, as shown in Figure 5-4. The latencies for tiovxisp0 (VISS), tiovxldc0 (LDC), and tiovxmultiscaler0 (MSC) shown in the figure are as expected:
  - For VISS and LDC, latency is approximately 5MPixel/375MHz + overhead ≈ 14-15msec, where 375MHz is the ISP(VPAC) operating clock frequency.
  - For MSC, the YUV data can be processed simultaneously or separately. When processed separately (the default configuration), luma (Y) plane latency is about 14-15 msec, the same as VISS and LDC, while chroma (UV) plane latency is about half of luma plane latency, or 7 msec. Then total latency for MSC is about 21 msec. This latency can be reduced to 14 msec by configuring the MSC to process both planes simultaneously.

```
/GstPipeline:pipeline0/GstFPSDisplaySink:RGB: last-message = rendered: 61654, dropped: 0, current: 29.83, average: 29.84
/GstPipeline:pipeline0/GstFPSDisplaySink:RGB: last-message = rendered: 61804, dropped: 0, current: 29.84, average: 29.84
/GstPipeline:pipeline0/GstFPSDisplaySink:RGB: last-message = rendered: 61954, dropped: 0, current: 29.83, average: 29.84
/GstPipeline:pipeline0/GstFPSDisplaySink:RGB: last-message = rendered: 62104, dropped: 0, current: 29.83, average: 29.84
/GstPipeline:pipeline0/GstFPSDisplaySink:RGB: last-message = rendered: 62254, dropped: 0, current: 29.83, average: 29.84
/GstPipeline:pipeline0/GstFPSDisplaySink:RGB: last-message = rendered: 62404, dropped: 0, current: 29.83, average: 29.84
/GstPipeline:pipeline0/GstFPSDisplaySink:RGB: last-message = rendered: 62554, dropped: 0, current: 29.83, average: 29.84
/GstPipeline:pipeline0/GstFPSDisplaySink:RGB: last-message = rendered: 62704, dropped: 0, current: 29.83, average: 29.84
/GstPipeline:pipeline0/GstFPSDisplaySink:RGB: last-message = rendered: 62854, dropped: 0, current: 29.83, average: 29.84
/GstPipeline:pipeline0/GstFPSDisplaySink:RGB: last-message = rendered: 63004, dropped: 0, current: 29.84, average: 29.84
```

No frame drop

30fps

Figure 5-2. Pipeline Throughput of Single Stream Capture and Visualization

```
Summary of CPU load,
====================

CPU: mpu1_0: TOTAL LOAD =   8.43 % ( HWI =   0. 0 %, SWI =   0. 0 % )
CPU: mcu1_0: TOTAL LOAD =   1.97 % ( HWI =   0. 0 %, SWI =   0. 0 % )
CPU:  c7x_1: TOTAL LOAD =   0. 4 % ( HWI =   0. 0 %, SWI =   0. 0 % )

HWA performance statistics,
===========================

HWA:   VISS: LOAD =  47.18 % ( 172 MP/s )
HWA:   LDC : LOAD =  43.60 % ( 158 MP/s )
HWA:   MSC1: LOAD =  69.25 % ( 170 MP/s )

DDR performance statistics,
===========================

DDR: READ  BW: AVG =   1291 MB/s
DDR: WRITE BW: AVG =    955 MB/s
DDR: TOTAL BW: AVG =   2246 MB/s

SoC temperature statistics
==========================

thermal_zone0(DDR):     56.73 degree Celsius
thermal_zone1(CPU):     50.38 degree Celsius
thermal_zone2(C7x):     50.82 degree Celsius
```

CPU load

HWA load

DDR utilization

Figure 5-3. CPU, HWA, and DDR Load of Single Stream Capture and Visualization

element	latency	out-latancy	out-fps	frames
capsfilter2	0.16	33.51	29	5701
queue3	0.09	33.51	29	5701
tiperfoverlay0	2.99	33.51	29	5701
queue4	0.26	33.51	29	5701
tiovxldc0	14.36	33.51	29	5702
capsfilter1	0.15	33.51	29	5701
queue2	0.09	33.51	29	5701
capsfilter0	0.08	33.51	29	5701
queue0	0.13	33.51	29	5701
tiovxisp0	14.87	33.51	29	5701
queue1	0.18	33.51	29	5701
tiovxmultiscaler0	20.94	33.51	29	5701

LDC

VISS

MSC

Figure 5-4. Pipeline Element Latencies of Single Stream Capture and Visualization

5.2 Application 2: Dual-stream Capture and Visualization with GST and TIOVX Frameworks

This section analyzes an application that visualizes both RGB and Infrared streams simultaneously.

This is composed in both GStreamer and TIOVX to allow comparison between the two. Shown in Figure 5-5 is the application in GStreamer and Figure 5-6 shows the equivalent application constructed with TIOVX. The GST commands and TIOVX code can be found in the GitHub [6].

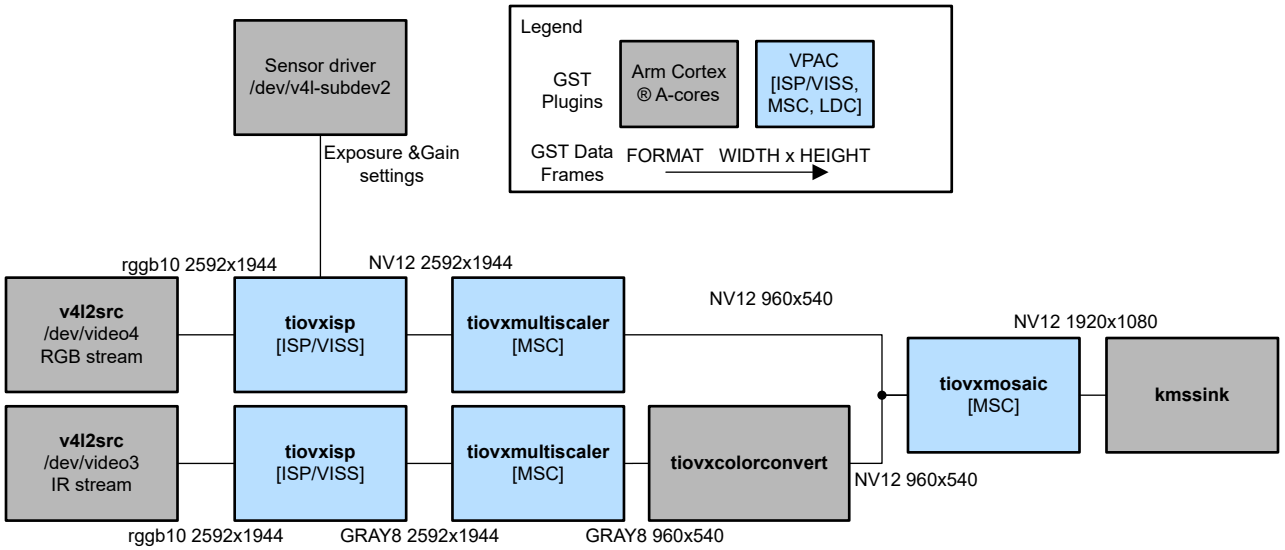

Figure 5-5. Dual Stream Capture and Display with GStreamer

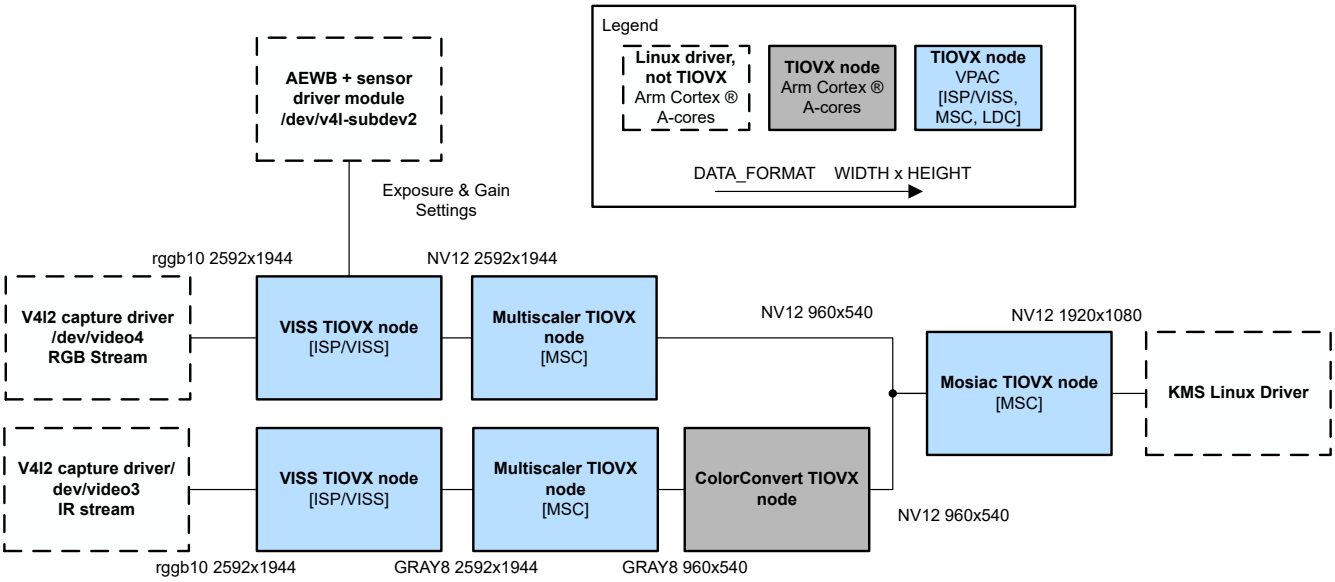

Figure 5-6. Dual Stream Capture and Display with TIOVX

Frames arrive through v4l2 from the corresponding /dev/videoX entries for RGB and IR streams and are processed by the on-chip ISP. Both streams are downscaled to a resolution that fit into the monitor, and the IR stream (in grayscale) is converted to the same color format as the RGB stream. Then, the streams are combined into a single frame with a *mosaic* feature before displaying to a monitor through the Linux KMS or DRM interface.

The TIOVX and GStreamer applications are equivalent in terms of the processing functions involved, however, there are a few key differences. The TIOVX application builds a TIOVX graph to handle the inner body of the application, which, in this case, is the ISP, down-scaling, color-conversion, and image-merging (mosaic) features. Input from V4L2 and output through KMS or DRM is handled using Linux-level APIs outside the TIOVX graph. However, GStreamer has numerous plugins available to implement these API calls by plugins. The TIOVX application is compiled into a binary application and run whereas the GStreamer pipeline can be represented using a single string that can be run from the command line.

Performance Statistics and SoC Resource Utilization

This section analyzes resource utilization on AM62A74 while running these applications. The measurement application receives remote core utilizations through TIOVX and reads other information like DDR utilization and temperature through memory-mapped registers. This *perf_stats* application is part of the SDK under the /opt/edgeai-gst-apps/scripts/perf_stats directory. The sampling interval for SoC utilization is 500ms; these are collected across a 20 second window (600 frames per RGB, IR stream) and averaged together into the double bar-chart shown in Figure 5-7.

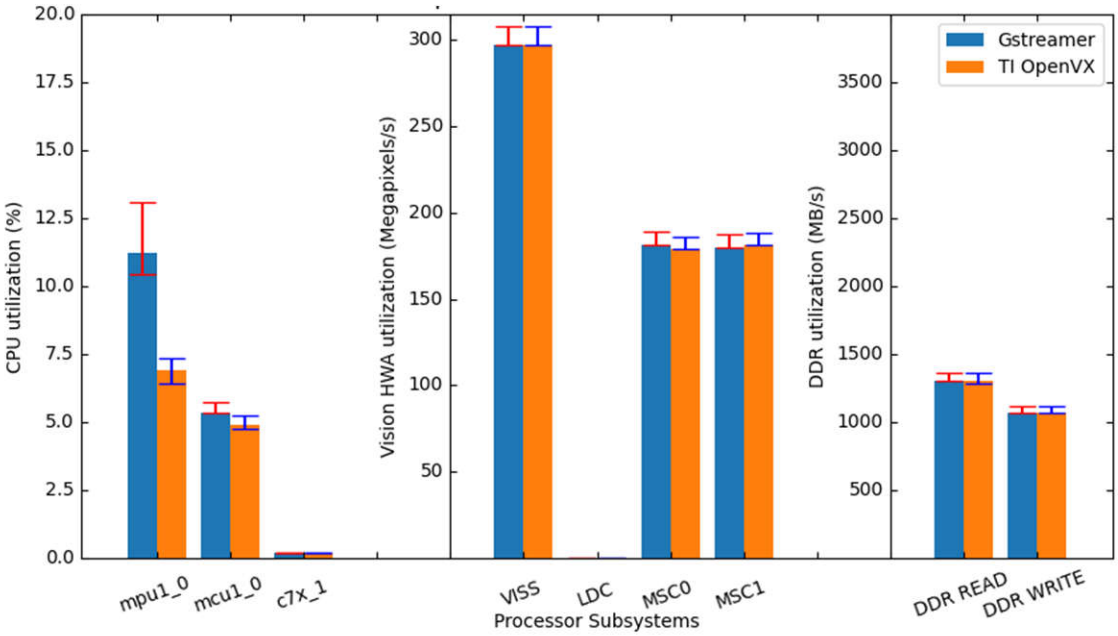

Figure 5-7. Utilization Comparison of GStreamer and TIOVX For Dual-stream Visualization Pipeline

Error bars in this chart represent the 25th and 75th percentile. Between the two frameworks, accelerator and DDR utilization is at parity for the application running at 30FPS (per input stream) without frame drops. Notably, the MPU (quad Arm® Cortex®-A53 in SMP mode) denoted as mpu1_0 in Figure 5-7, has higher utilization for GStreamer than TIOVX. This is due to increased signalling between the CPU complex and any remote core or accelerator. C7xMMA is unused in this application. Otherwise, Core or HWA utilizations are very similar between the two application frameworks, with TIOVX being slightly more efficient.

Table 5-1 shows latency through individual components of the pipeline. Frame capture and display latency is not included. For each processing task in the application, compare the latency for GStreamer and TIOVX and the total latencies. From here, the TIOVX is generally faster, especially for color conversion; the infrared path is most affected by the difference in application frameworks.

Table 5-1. GStreamer vs. TIOVX Latencies

Function	GStreamer (ms)	TIOVX (ms)
VISS ISP (Infrared)	18.5	13.9
VISS ISP (RGB)	17.6	14.1
MSC Downscaling (Infrared)	14.3	13.7
MSC Downscaling (RGB)	21.2	20.5
Color conversion (Infrared->NV12)	19.2	0.64
Mosaic combining images (RGB + IR)	5.5	4.7
Sum latencies (IR path)	**57.5**	**32.9**
Sum latencies (RGB path)	**44.3**	**39.3**

GStreamer internally implements TIOVX nodes for each plugin, so TIOVX nodes always measure faster than the equivalent plugin in GStreamer. Measurements in GStreamer are captured before and after the plugin runs from Linux, whereas measurements in TIOVX can be captured and reported by the remote core running the operation. There is a noticeable improvement[1] in TIOVX compared to GStreamer, especially for the color conversion from grayscale to NV12 format[2].

Another way to compare these applications is in regards interrupts and how frequent inter-processor communication is (that is, A53 messaging C7x, R5F messaging A53, and so forth). Fewer interrupts is better, as this allows the processor to more quickly address pending signals from different peripherals and accelerators. Measure interrupt count from Linux to see the number of interrupts the A53's (running Linux) received for each application before and after running for 600 frames.

Number of interrupts to Linux across 20 second duration (600 frames per stream). GStreamer shows more interrupts than TIOVX because Cortex A53 cores running Linux must be notified between each plugin/pipeline element. Interrupt counts for individual core's mailbox are captured from /proc/interrupts.

Table 5-2. GStreamer vs. TIOVX Interrupt Counts

	GStreamer Application	TIOVX Application
DM R5F (manage VPAC)	13,469	10,589
C7x	0 (unused)	0 (unused)
MCU R5F	0 (unused)	0 (unused)

The data in Table 5-2 reflects the general trend that GStreamer is less efficient in terms of CPU interrupts than TIOVX. This is because TIOVX allows all cores to communicate directly, whereas GStreamer requires the cores to flow through the Linux host (A53). Adding AI processing with TIDL shows a similar pattern for C7x interrupts.

5.3 Application 3: Representative OMS-DMS + Video Telephony Pipeline in GStreamer

In this example application, a representative GStreamer pipeline is run for occupancy monitoring, driver monitoring and video telephony on live stream from an OX05B1S camera.

For OMS and DMS, the infrared frames are passed through a segmentation and object detection model, respectively. These models emulate a seatbelt detection and face detection task, respectively. The RGB frames are H.265 encoded and saved into a file. This represents a starting point for an RGB-IR in-cabin monitoring applications. Note that the deep learning models used here are not optimized [3]for DMS/OMS. A final product needs to extend this to include more DMS and OMS models and postprocessing for a robust and feature-rich product.

[1] In some GStreamer pipelines, individual plugin latency can be impacted by adjacent plugins and presence of queues, since these queues affect how GStreamer chooses to multithread plugins and portions of the pipeline. This also explains why the GStreamer application's latencies in Table 5-1 exceed those shown in Figure 5-4.

[2] Note that color conversion is not necessary in a practical application; this is included to allow RGB and Ir streams to be simultaneously visualized by stitching frames. Both frames must be in the same format. A production application does not require this color conversion step.

[3] The best models use Infrared frames as grayscale images, although the models profiled here use RGB data. The models profiled in this app note have higher DDR utilization than designed models for this use case.

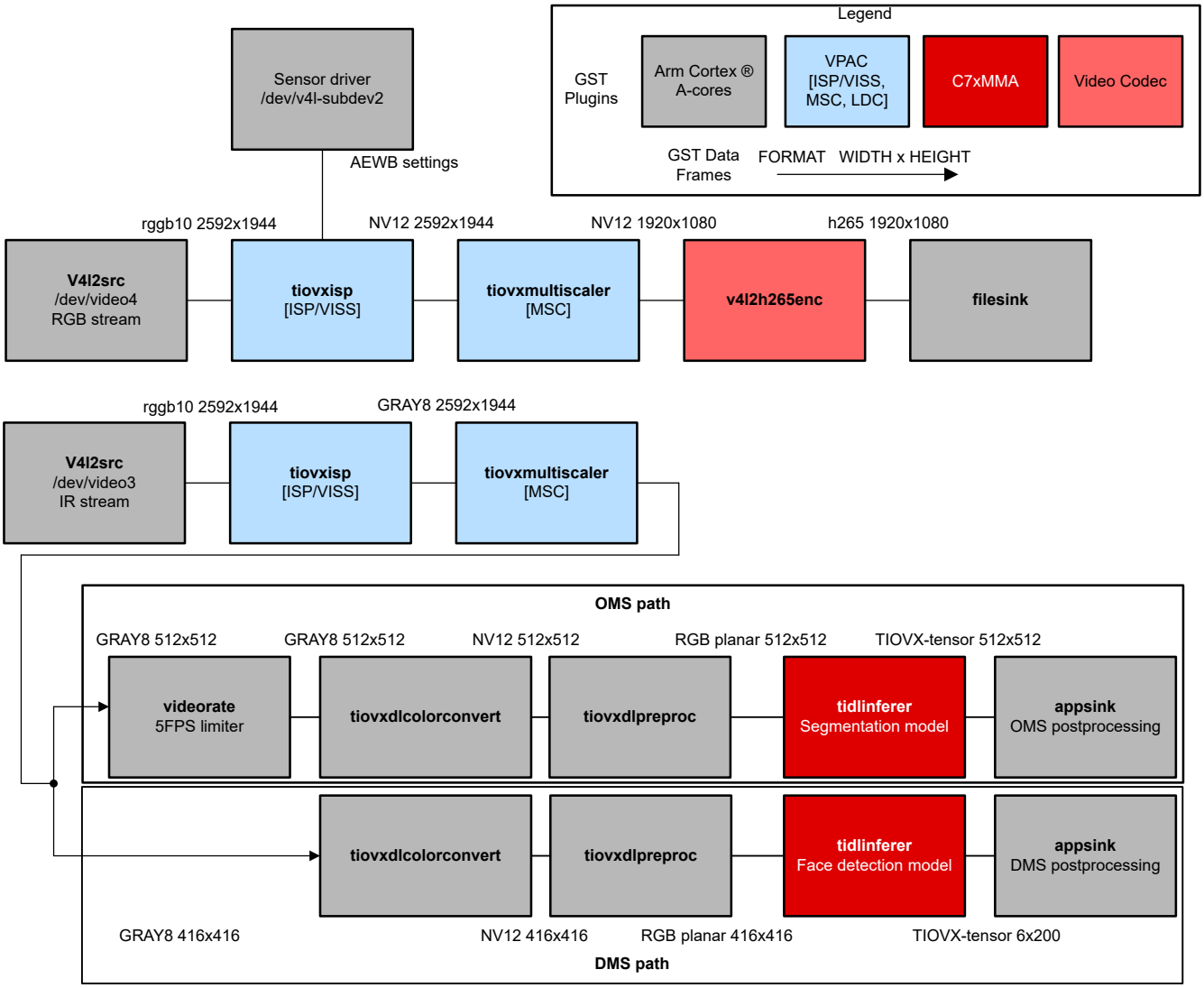

Figure 5-8. GStreamer Application Flow For Dual-stream RGB-IR For DMS-OMS With Video Recording

In this application, depicted by Figure 5-8, the RGB frames are kept only for file storage by scaling and encoding in H.265 format. Infrared input is processed and scaled down before splitting into OMS and DMS paths. DMS runs at 30 FPS whereas OMS runs more slowly at 5 FPS (with excess frames dropped in OMS path). From here, the infrared frames are prepared for deep learning by scaling, converting colorspace to RGB (although final models can be optimized for grayscale input), preprocessed, and run through a CNN on the C7x deep learning accelerator. Any post-processing for these models run in an application code exposed by the GStreamer *appsink* plugin.

Figure 5-9 shows the core load on AM62A while running this application. The VISS-ISP hardware accelerator is nearly maxed out for the 315 MP/s capacity. However, there is plenty of headroom left on processing cores. The 4x Arm® Cortex®-A53s are only used approximately 26%, leaving plenty of space for application code and other services. The C7xMMA AI accelerator is less than 50% loaded, leaving headroom for other AI models, and many further optimizations can be performed to maximize IR image analysis. The 32-bit, 3200 MT/s DDR bus shows utilization of 35%, and can reach approximately 50-60% before this shared resource becomes contended for. Note that DDR utilization must be strongly considered when optimizing the overall system.

Individually, the OMS, DMS, and telephony/recording portions of the pipeline have latencies of approximately 85ms, 51ms, and 67ms, respectively (not including frame-capture latency).

Figure 5-9 shows compute, accelerator, DDR utilization for the DMS-OMS reference GStreamer application. Note that models here are off-the-shelf from TI model zoo and used to emulate a realistic load for a simple

DMS/OMS application. There is headroom for further deep learning and image analysis algorithms, including opportunity to optimize and reduce DDR load.

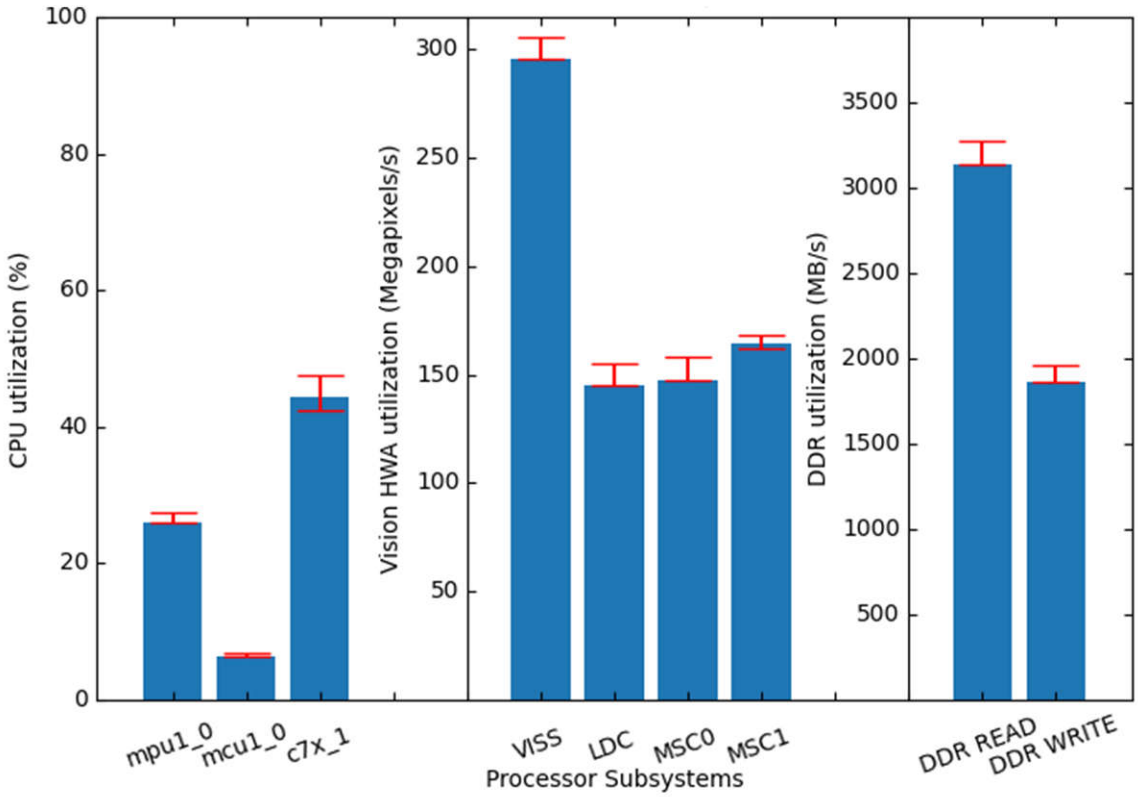

Figure 5-9. Core Utilizations for RGB-Ir DMS/OMS Application

6 Summary

This application note describes, implements, and benchmarks a reference design for driver and occupancy monitoring systems using RGB-IR image sensors with the AM62A processor. An OX05B1S camera module is used to create RGB and Infrared vision processing pipelines that leverage hardware accelerates on AM62A for this 5 Megapixel use case. We explore the fundamental components of a representative OMS/DMS pipeline and measure utilization and latency across said pipelines in GStreamer and TI OpenVX.

7 References

1. Texas Instruments, Driver and Occupancy Monitoring Systems on AM62A, technical white paper
2. Texas Instruments, Processor SDK Linux for edge AI applications on AM62A, web site
3. Texas Instruments, Multimedia Applications on AM62A, application note
4. Texas Instruments, AM6xA ISP Tuning Guide, application note
5. Texas Instruments, Perf_stats tool, website
6. Texas Instruments, EdgeAI-RGB-IR, code repository

8 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Revision * (February 2025) to Revision A (March 2025)	Page
• Updated link between RGB and IR dominant streams and Linux video device nodes: /dev/video3 for IR and /dev/video4 for RGB	12

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265