CLARA AGX Documentation

Release 3.1

Nvidia Corporation

Aug 27, 2021

Introduction

Chapter 1

Overview

Clara AGX SDK is a collection of documentation, drivers, and reference applications that has been designed by NVIDIA to help developers build end-to-end streaming workflows for medical imaging applications. From training neural networks using the Transfer Learning Toolkit, to building a streaming inference application using DeepStream, to deployment on a Jetson AGX embedded device; the Clara AGX SDK takes an incremetal approach to introduce concepts and NVIDIA technology in a way that can be applied irrespective of previous development experience in the field of medical imaging.

While the ultimate goal of the Clara AGX SDK is to deploy applications on a Jetson AGX embedded device, the SDK includes x86 Linux versions of the sample applications in order to provide a cross-platform development environment that can be used even without access to Jetson AGX device. In any case, an NVIDIA GPU will be required in order to utilize the Transfer Learning Toolkit, DeepStream, TensorRT, CUDA, and any other NVIDIA software components that are used by the Clara AGX SDK (see *Prerequisites*).

Chapter 2

Clara AGX SDK Components

The Clara AGX SDK has been divided into a number of separate installation packages, as follows.

2.1 Clara AGX TLT

The Clara AGX TLT package provides a sample annotated data set as well as various scripts that are used to train a neural network and output a TensorRT runtime inference engine that can be used with the sample applications.

The sample data set consists of 300 images of the Learning Resources Anatomy Model that has been annotated with `stomach` and `intestines` regions. This data set can be used as-is with the *DeepStream and Endoscopy Sample Apps*, which include prebuilt TensorRT engines generated using this dataset. Alternatively, a custom data set and corresponding inference model may be annotated and generated by the user using utility scripts provided in the AGX TLT package and following the *Model Creation* section in this documentation.

Sample Image of the Learning Resources Anatomy Model

Tip: If you do not have the Learning Resources Anatomy Model available, the `stomach` and `intestines` inference on this model that is done by the sample application can be tested by simply pointing your camera device to a screen showing any of the images for this data set that are included with the AGX TLT package in the `data/organs` directory. By default, this directory is created by the SDK manager on the host machine at `~/nvidia/nvidia_sdk/Clara_AGX_3.1_Linux/Clara_AGX_TLT`.

2.2 Sample Applications

Building a powerful end-to-end inference application using DeepStream and TensorRT requires the use of a number of different software libraries that can be overwhelming if starting with no knowledge of these libraries. To help alleviate this problem, the AGX SDK includes three different sample applications that build on top of one another to incrementally introduce new components and features.

All of the sample applications included with the AGX SDK are written in C and/or C++.

2.2.1 GStreamer Sample

The GStreamer sample application is the most basic sample which demonstrates the use of the GStreamer framework in order to simply capture image frames from a camera and then render the captured video

stream onto the display. This sample does not make use of DeepStream, but since DeepStream functionality is exposed to applications via GStreamer framework plugins, a basic understanding of GStreamer is essential to write DeepStream applications.

The GStreamer sample is available for the Jetson AGX and Linux host, and are located at `/opt/nvidia/clara-agx-sdk/clara-agx-deepstream-sample/GStreamerJetsonSample` on AGX, and `/opt/nvidia/clara-agx-sdk/clara-agx-deepstream-sample/GStreamerLinuxHostSample` on the host machine.

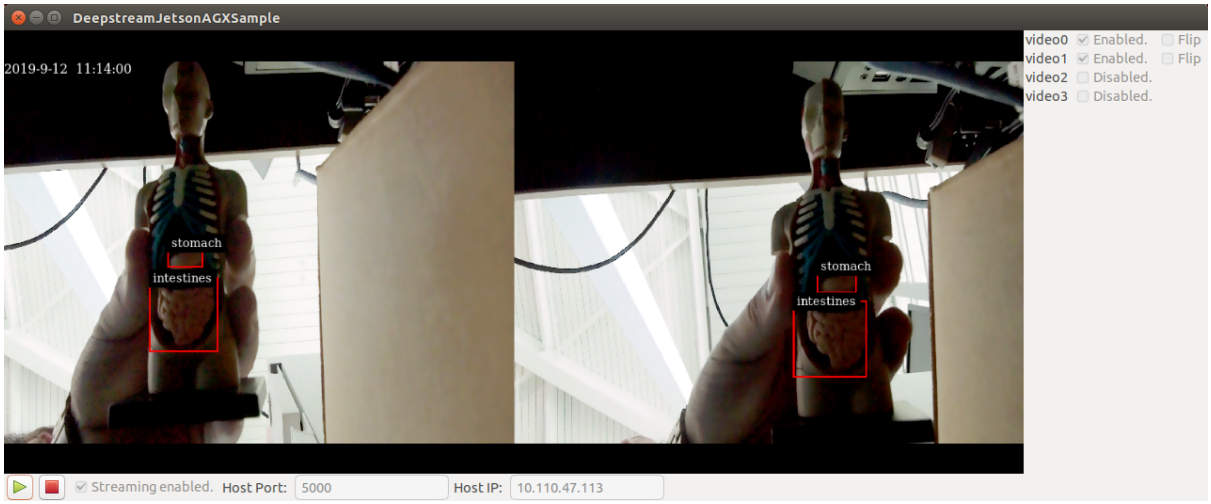
2.2.2 DeepStream Sample

The DeepStream sample application builds on top of the GStreamer sample by including the DeepStream inference plugin into the pipeline such that the `stomach` and `intestines` regions of the sample model are identified by bounding boxes rendered on top of the video stream. Alternatively, the sample model can be replaced with a user-provided model (by changing a configuration file used by the sample) to identify regions specified by the custom model.

This sample also expands on the GStreamer sample by providing multi-camera support, as well as the ability to send the captured and annotated video across a UDP H.264 video stream such that it can be viewed by another remote system.

The DeepStream sample is available for Jetson AGX and the Linux host, and are located at `/opt/nvidia/clara-agx-sdk/clara-agx-deepstream-sample/DeepstreamJetsonAGXSample` on AGX, and `/opt/nvidia/clara-agx-sdk/clara-agx-deepstream-sample/DeepstreamLinuxHostSample` on the host machine.

This sample will later be highlighted here: *DeepStream and Endoscopy Sample Apps*.



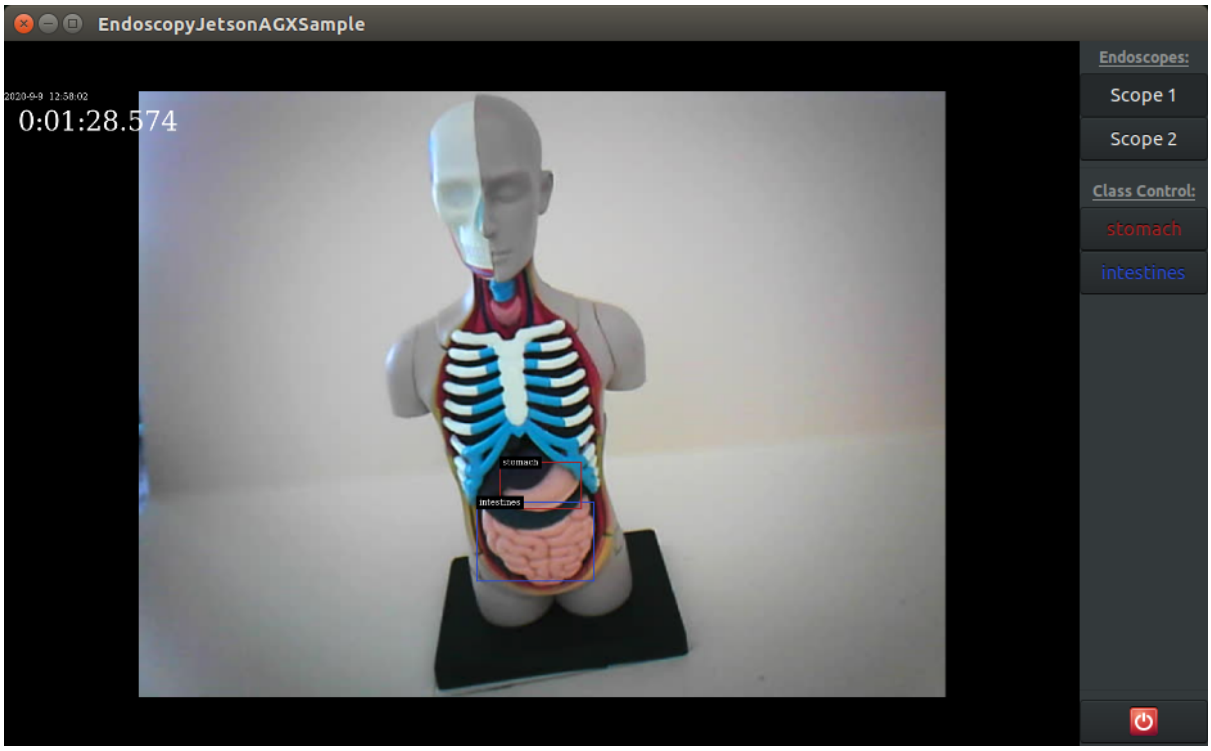
Multi-Stream DeepStream Sample Application

2.2.3 Endoscopy Sample

The Endoscopy sample builds on top of the DeepStream sample by adding runtime controls that are specific to the model and inferencing that is performed by DeepStream. These controls include the ability to switch between multiple inference models, and to enable/disable the annotation of the inference regions that are provided by the models. The UI is also modified to provide a more streamlined experience that might be desirable for an application that is deployed into medical imaging environments.

The Endoscopy sample is available only for Jetson AGX, and is located here: `/opt/nvidia/clara-agx-sdk/clara-agx-deepstream-sample/EndoscopyJetsonAGXSample`.

The endoscopy example is explored in more detail here: *DeepStream and Endoscopy Sample Apps*.



Endoscopy Sample Application

Chapter 3

Prerequisites

The Clara AGX SDK has been optimized to work with either the Jetson AGX Xavier or Clara AGX Development Kits. The following additional components are required to make use of various feature or samples included with the AGX SDK.

3.1 JetPack Software

The following components are required and will be installed onto the AGX device using the NVIDIA SDK Manager.

- JetPack 4.5.0
- DeepStream 5.1

3.2 CSI Camera Module

A CSI camera module is used for the execution of the DeepStream and endoscopy sample applications on the target AGX device. While many CSI camera modules should work with the samples, the following have been tested and are recommended.

- Clara AGX HDMI Input Module (Included with Clara AGX Development Kit)
- Leopard IMX-274
- Leopard IMX-390 with the GMSL kit

Note: The Leopard IMX-390 requires a custom kernel, which can be built using the L4T kernel build instructions

---

Important: Camera modules with CSI connectors are not supported in dGPU mode on Clara AGX Development kit.

3.3 USB Camera

The DeepStream Sample application may also be run on either the NVIDIA Xavier or the Linux x86 host machine using a USB camera. Many USB cameras should be compatible, but the following have been tested with the AGX SDK.

- Logitech C270
- Logitech C310

3.4 Linux x86 Host

During setup, the Clara AGX SDK requires a Linux x86 PC with the following hardware and software:

- An NVIDIA Quadro® or Tesla® GPU that supports CUDA 11.1.
  - If training a new model from a data set, the GPU must have at least 8GB of GDDR memory.
- At least 8GB of RAM
- Ubuntu 18.04
- NVIDIA Container Runtime

The following software is also required and will be installed by SDK Manager as part of the Clara AGX SDK installation.

- CUDA 11.1
- TensorRT 7.2
- DeepStream 5.1

After setup is complete the host machine is no longer needed to operate Clara AGX.

Chapter 4

SDK Installation

Installation of the Clara AGX SDK is automated via the NVIDIA SDK Manager.

4.1 SDK Manager Installation

Starting with Clara AGX SDK version 3.1, the NVIDIA SDK Manager is the only means of installation for the Clara AGX SDK. Please refer to the information button within the SDK Manager installation page to inspect where various components were installed or extracted to. More information may be found in the SDK Manager documentation.

Chapter 5

Storage Setup (m2 SSD)

Note: If the Clara AGX Developer Kit is reflashed with a new JetPack image, the partition table of the m2 drive will not be modified and the contents of the partition will be retained. In this case the **Create Partition** steps can be skipped, however the **Mount Partition** steps should be followed again in order to remount the partition.

Also note that any state, binaries, or docker images that persist on the m2 drive after flashing the system may be made incompatible with new libraries or components that are flashed onto the system. It may be required to recompile or rebuild these persistent objects to restore runtime compatibility with the system.

The Clara AGX Developer Kit includes a pre-installed 250GB m2 solid-state drive (SSD), but this drive is not partitioned or mounted by default. This page outlines the steps that should followed after the initial SDK installation in order to partition and format the drive for use.

Note: The following steps assume that the m2 drive is identified by the Clara AGX Developer Kit as `/dev/sda`. This is the case if no additional drives have been attached, but if other drives have been attached (such as USB drives) then the disk identifier may change. This can be verified by looking at the symlink to the drive that is created for the m2 hardware address on the system. If the symlink below shows something other than `../../sda`, replace all instances of `sda` in the instruction below with the identifier that is being used by your system:

```
$ ls -l /dev/disk/by-path/platform-14100000.pcie-pci-0001\:01\:00.0-ata-1
lrwxrwxrwx 1 root root 9 Jan 28 12:24 /dev/disk/by-path/platform-14100000.
↪pcie-pci-0001:01:00.0-ata-1 -> ../../sda
```

5.1 Create Partition

1. **Launch fdisk utility:**

   ```
   $ sudo fdisk /dev/sda
   ```

2. **Create a new primary partition.** Use the command **'n'**, then accept the defaults (press enter) for the next 4 questions to create a single partition that uses the entire drive:

```
Command (m for help): n
Partition type
   p   primary (0 primary, 0 extended, 4 free)
   e   extended (container for logical partitions)
Select (default p):

Using default response p.
Partition number (1-4, default 1):
First sector (2048-488397167, default 2048):
Last sector, +sectors or +size{K,M,G,T,P} (2048-488397167, default␣
↪488397167):

Created a new partition 1 of type 'Linux' and of size 232.9 GiB.
```

3. **Write the new partition table and exit.** Use the **'w'** command:

```
Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.
```

4. **Initialize ext4 filesystem on the new partition:**

```
$ sudo mkfs -t ext4 /dev/sda1
mke2fs 1.44.1 (24-Mar-2018)
Creating filesystem with 486400 4k blocks and 121680 inodes
Filesystem UUID: c3817b9c-eaa9-4423-ad5b-d6bae8aa44ea
Superblock backups stored on blocks:
  32768, 98304, 163840, 229376, 294912

Allocating group tables: done
Writing inode tables: done
Creating journal (8192 blocks): done
Writing superblocks and filesystem accounting information: done
```

5.2 Mount Partition

1. **Create a directory for the mount point.** These instructions will use the path `/media/m2`, but any path may be used if preferred.

```
$ sudo mkdir /media/m2
```

2. **Determine the UUID of the new partition.** The UUID will be displayed as a symlink to the `/dev/sda1` partition within the `/dev/disk/by-uuid` directory. For example, the following output shows that the UUID of the `/dev/sda1` partition is `4b2bb292-a4d8-4b7e-a8cc-bb799dfeb925`:

```
$ ls -l /dev/disk/by-uuid/ | grep sda1
lrwxrwxrwx 1 root root 10 Jan 28 10:05 4b2bb292-a4d8-4b7e-a8cc-
↪bb799dfeb925 -> ../../sda1
```

(continues on next page)

(continued from previous page)

3. **Add the fstab entry.** Using the mount path and the UUID from the previous steps, add the following line to the end of `/etc/fstab`:

```
UUID=4b2bb292-a4d8-4b7e-a8cc-bb799dfeb925 /media/m2 ext4 defaults 0 2
```

4. **Mount the partition.** The `/etc/fstab` entry above will mount the partition automatically at boot time. To instead mount the partition immediately without rebooting, use the `mount` command (and `df` to verify the mount):

```
$ sudo mount -a
$ df -h /dev/sda1
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1       229G  5.6M  229G   0% /media/m2
```

5.3 Move Docker Storage to m2 Partition

A complete installation of the Clara SDK leaves only about 10GB of storage remaining in the root 32GB filesystem (`/dev/mmcblk0p1`). When using Docker it is often the case that individual images will be many GB in size, and so this remaining disk space is generally insufficient for the storage needs of Docker images. For this reason it is highly recommended that the Docker daemon data directory be moved to a location on the new m2 partition. This can be done with the following steps:

1. **Create a new Docker data directory.** This is where Docker will store all of its data including build cache and container images. These instructions use the path `/media/m2/docker-data`, but another directory name can be used if preferred:

```
$ sudo mkdir /media/m2/docker-data
```

2. **Configure the Docker Daemon.** Add the following `data-root` configuration option to the `/etc/docker/daemon.json` file, pointing to the new data directory created above. Create the `/etc/docker/daemon.json` file if it does not already exist.

```
{
    "data-root": "/media/m2/docker-data"
}
```

Note: If existing configuration already exists in the `daemon.json` file, make sure to add a comma to the preceeding line before the `data-root` configuration, e.g.

```
{
    ...
    "default-runtime": "nvidia",
    "data-root": "/media/m2/docker-data"
}
```

3. **Restart the Docker Daemon.**

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

Chapter 6

Camera Setup and Verification

This page outlines how to verify the camera setup used with the Jetson target and x86 host devices to ensure they are functioning properly for use with the sample applications.

6.1 Jetson Camera Setup

1. Connect a CSI or USB camera module to your Clara AGX Development Kit or Jetson AGX module. Once connected, you may see what device(s) you have by the following:

   ```
   $ ls /dev/video*
   ```

   The format(s) of your device(s) may be seen by:

   ```
   $ v4l2-ctl -d /dev/video0 --list-formats
   ```

   If the above command fails (which may occur if the DeepStream example was not installed), then run:

   ```
   $ sudo apt install -y v4l-utils
   ```

   Please see the *Prerequisites* section for the list of compatible camera devices.

   Note: The CSI camera module, including the CSI HDMI input board included with the Clara AGX Development Kit, will enumerate as `/dev/video0` and any additional USB cameras will start to enumerate with `/dev/video1`. If there is no CSI module attached, USB cameras will start to enumerate with `/dev/video0`.

2. Run one of the following commands to prove that capture is working correctly. In all cases, a window should be rendered onto the display with a live stream from the capture.

   - CSI Camera

     ```
     $ gst-launch-1.0 nvarguscamerasrc ! 'video/x-raw(memory:NVMM),␣
     ↪width=(int)1920, height=(int)1080, format=(string)NV12,␣
     ↪framerate=(fraction)30/1' ! nvoverlaysink -e
     ```

Note: `nvarguscamerasrc` plugin is not available in dGPU mode on Clara AGX Development Kit.

- CSI HDMI Input Board

```
$ gst-launch-1.0 v4l2src io-mode=mmap device=/dev/video0 ! 'video/x-raw, format=(string)BGRA, width=(int)1920, height=(int)1080, framerate=(fraction)60/1' ! capssetter join=true caps='video/x-raw, format=(string)RGBA' ! nvvideoconvert ! 'video/x-raw(memory:NVMM), format=(string)RGBA' ! nveglglessink sync=0
```

- USB Camera

```
$ gst-launch-1.0 v4l2src device=/dev/video1 ! xvimagesink
```

Note: Replace `/dev/video0` in the commands above with the path that corresponds to the device being tested. See note about device enumeration in step 1, above.

3. Press **Ctrl+C** to exit the camera capture.

6.2 Linux x86 Camera Setup

1. Connect a USB camera to your Linux x86 host.
2. Run the following command to prove that capture is working correctly. A window should be rendered onto the display with a live stream from the camera.

```
$ gst-launch-1.0 v4l2src device=/dev/video1 ! xvimagesink
```

Note: Replace `/dev/video1` in the command above with the path that corresponds to the USB camera device being tested.

3. Press **Ctrl+C** to exit the camera capture.

6.3 Streaming Camera Setup Check

The DeepStream sample application offers the ability to stream the camera image from the target device to a remote host system using a UDP data stream. These steps can be followed to ensure that both the target and host systems are setup correctly for this streaming to work.

6.3.1 Jetson

1. Ensure the camera capture is working as described in the *Jetson Camera Setup* section, above.

2. Run the following command to start streaming the camera capture, replacing <IP_ADDRESS> with the IP address of the host system that will be receiving the stream.

   - CSI Camera

```
$ gst-launch-1.0 nvarguscamerasrc ! 'video/x-raw(memory:NVMM),␣
↪width=(int)1920, height=(int)1080, format=(string)NV12,␣
↪framerate=(fraction)30/1' ! \
omxh264enc control-rate=2 bitrate=4000000 ! 'video/x-h264, stream-
↪format=(string)byte-stream' ! \
h264parse ! rtph264pay mtu=1400 ! udpsink host=<IP_ADDRESS>␣
↪port=5000 sync=false async=false
```

   - CSI HDMI Input Board

```
$ gst-launch-1.0 v4l2src io-mode=mmap device=/dev/video0 ! 'video/x-
↪raw, format=(string)BGRA, width=(int)1920, height=(int)1080,␣
↪framerate=(fraction)60/1' ! capssetter join=true caps='video/x-raw,
↪ format=(string)RGBA' ! nvvideoconvert ! 'video/x-raw(memory:NVMM)␣
↪, format=(string)I420' ! nvv4l2h264enc ! h264parse ! rtph264pay !␣
↪udpsink host=<IP_ADDR> port=5000
```

   - USB Camera

```
$ gst-launch-1.0 v4l2src device=/dev/video0 ! nvvideoconvert !
↪'video/x-raw(memory:NVMM), format=(string)I420' ! nvv4l2h264enc !␣
↪h264parse ! rtph264pay ! udpsink host=<IP_ADDR> port=5000
```

Note: Replace `/dev/video0` in the commands above with the path that corresponds to the device being tested. See note about device enumeration in step 1 of the *Jetson Camera Setup*, above.

6.3.2 Linux Host

Use the following command to start streaming the camera capture on a Linux x86 host (close with **Ctrl+C**):

```
$ gst-launch-1.0 udpsrc port=5000 ! application/x-rtp,encoding-name=H264,␣
↪payload=96 ! rtph264depay ! queue ! avdec_h264 ! autovideosink sync=false
```

Chapter 7

Switching Between iGPU and dGPU

The Clara AGX Developer Kit can use either the Xavier AGX module GPU (iGPU, integrated GPU) or the RTX6000 add-in card GPU (dGPU, discrete GPU). Only one GPU can be used at a time, with the iGPU being the default after flashing the Clara AGX SDK with JetPack. Switching between the iGPU and dGPU is performed using the **nvgpuswitch.py** script contained in /usr/local/bin.

> **Warning:** The Clara AGX SDK components, such as the DeepStream sample applications, will be uninstalled during the process of switching GPUs. If you have made any changes to the source code installed by these packages (i.e. in `/opt/nvidia/clara-agx-sdk`), make sure to back up these changes before switching GPUs. See *Reinstalling Clara AGX SDK Packages* for more details.

To **view** the currently installed drivers and their version, use the query command:

```
$ nvgpuswitch.py query
iGPU (nvidia-l4t-cuda, 32.5.0-20201012161040)
```

To install the **dGPU** drivers, use the install command with the dGPU parameter (note that sudo must be used to install drivers):

```
$ sudo nvgpuswitch.py install dGPU
```

The install command will begin by printing out the list of commands that will be executed as part of the driver install, then will continue to execute those commands. This aids with debugging if any of the commands fail to execute for any reason. The following **debug arguments** may also be provided with the install command:

```
-d  Does a dry run, showing the commands that would be executed
    by the install but does not execute them.

-v  Enable verbose output (used with `-d' to describe each of the
    commands that would be run).

-i  Run commands interactively (asks before running each command).

-l [LOG]
   Writes a log of the install to the file `LOG'.
```

The dGPU driver install may be verified once again using the query command:

```
$ nvgpuswitch.py query
dGPU (cuda-drivers, 455.32.00-1)
```

After the dGPU drivers have been installed, rebooting the system will complete the switch to the dGPU. At this point the Ubuntu desktop will be output via DisplayPort on the dGPU, and so the display cable must be switched from the onboard HDMI to DisplayPort on the dGPU.

Note: CUDA installs its runtime binaries such as `nvcc` into its own versioned path that is not included by the default `$PATH` environment variable. Because of this, attempts to run commands like `nvcc` will fail on dGPU unless the CUDA 11.1 path is added to the `$PATH` variable. This can be done for the current user after the swtich to dGPU by adding the following line to `$HOME/.bashrc`:

```
export PATH=/usr/local/cuda-11.1/bin:$PATH
```

If at any time you want to switch back to **iGPU**, use the install command with the iGPU parameter:

```
$ sudo nvgpuswitch.py install iGPU
```

After the iGPU drivers have been installed, rebooting the system will complete the switch to the iGPU. At this point the Ubuntu desktop will be output via the onboard HDMI, and so the display cable must be switched from the DisplayPort on the dGPU to the onboard HDMI.

Note: The GPU settings will persist through reboots until it is changed again with nvgpuswitch.py.

7.1 Reinstalling Clara AGX SDK Packages

When switching between GPUs, CUDA is first uninstalled and then reinstalled by the script in order to provide the correct versions used by iGPU or dGPU (CUDA 10.2 and 11.1, respectively). Since many of the Clara AGX SDK installed packages depend on CUDA, this means that these Clara AGX SDK packages are also uninstalled when the active GPU is switched.

To reinstall the Clara AGX SDK packages after switching GPUs, the corresponding `*.deb` packages that were downloaded by SDK Manager during the initial installation can be copied to the Clara AGX Developer Kit and installed using `apt`. By default, SDK Manager downloads the `*.deb` packages to the following location:

```
~/Downloads/nvidia/sdkm_downloads
```

The current list of packages that must be reinstalled is as follows. Note that the version numbers may differ – if this is the case, use the latest version of the `arm64` package that exists in the download directory.

VisionWorks: `libvisionworks` and `libvisionworks-dev`

DeepStream 5.1: `deepstream-5.1_5.1.0-1_arm64.deb`

Clara AGX DeepStream Samples: `clara-agx-deepstream-sample-src_1.3.1-0_arm64.deb`

DeepStream GPU Video Test Source Plugin: `clara-agx-gstnvvideotestsrc_1.0.0-0_arm64.deb`

Use `apt` to reinstall these packages:

```
$ sudo apt install -y libvisionworks libvisionworks-dev
$ sudo apt install -y ./deepstream-5.1_5.1.0-1_arm64.deb
$ sudo apt install -y ./clara-agx-deepstream-sample-src_1.3.1-0_arm64.deb
$ sudo apt install -y ./clara-agx-gstnvvideotestsrc_1.0.0-0_arm64.deb
```

Chapter 8

Rivermax SDK

The Clara AGX Developer Kit can be used along with the NVIDIA Rivermax SDK to provide an extremely efficient network connection using the onboard ConnectX-6 network adapter that is further optimized for GPU workloads by using GPUDirect. This technology avoids unnecessary memory copies and CPU overhead by copying data directly to or from pinned GPU memory, and supports both the integrated GPU or the RTX6000 add-in dGPU.

The instructions below describe the steps required to install and test the Rivermax SDK with the Clara AGX Developer Kit. The test applications used by these instructions, `generic_sender` and `generic_receiver`, can then be used as samples in order to develop custom applications that use the Rivermax SDK to optimize data transfers using GPUDirect.

Note: The Rivermax SDK may also be installed onto the Clara AGX Developer Kit via SDK Manager by selecting it as an additional SDK during the JetPack installation. If Rivermax SDK was previously installed by SDK Manager, many of these instructions can be skipped (see additional notes in the steps below).

Note: Access to the Rivermax SDK Developer Program as well as a valid Rivermax software license is required to use the Rivermax SDK.

8.1 Installing Mellanox Drivers (OFED)

The Mellanox OpenFabrics Enterprise Distribution Drivers for Linux (OFED) must be installed in order to use the ConnectX-6 network adapter that is onboard the Clara AGX Developer Kit.

Note: If Rivermax SDK was previously installed via SDK Manager, OFED will already be installed and these steps can be skipped.

---

1. Download OFED version **5.4-1.0.3.0**:

   MLNX_OFED_LINUX-5.4-1.0.3.0-ubuntu18.04-aarch64.tgz

   If the above link does not work, navigate to the **Downloads** section on the main OFED page, select either **Current Versions** or **Archive Versions** to find version **5.4-1.0.3.0**, select **Ubuntu**, **Ubuntu**

18.04, **aarch64**, then download the **tgz** file.

Note: Newer versions of OFED have not been tested and may not work.

2. Install OFED:

```
$ sudo apt install -y apt-utils
$ tar -xvf MLNX_OFED_LINUX-5.4-1.0.3.0-ubuntu18.04-aarch64.tgz
$ cd MLNX_OFED_LINUX-5.4-1.0.3.0-ubuntu18.04-aarch64
$ sudo ./mlnxofedinstall --force --force-fw-update --vma --add-kernel-
↪support
$ sudo /etc/init.d/openibd restart
```

8.2 Installing GPUDirect

The GPUDirect drivers must be installed to enable the use of GPUDirect when using an RTX6000 add-in dGPU. When using the iGPU the CPU and GPU share the unified memory and the GPUDirect drivers are not required, so this step may be skipped when using the iGPU.

Note: **The GPUDirect drivers are not installed by SDK Manager**, even when Rivermax SDK is installed, so these steps must always be followed to enable GPUDirect support when using the dGPU.

1. Download GPUDirect Drivers for OFED:

   nvidia-peer-memory_1.1.tar.gz

   If the above link does not work, navigate to the **Downloads** section on the GPUDirect page.

2. Install GPUDirect:

```
$ mv nvidia-peer-memory_1.1.tar.gz nvidia-peer-memory_1.1.orig.tar.gz
$ tar -xvf nvidia-peer-memory_1.1.orig.tar.gz
$ cd nvidia-peer-memory-1.1
$ dpkg-buildpackage -us -uc
$ sudo dpkg -i ../nvidia-peer-memory_1.1-0_all.deb
$ sudo dpkg -i ../nvidia-peer-memory-dkms_1.1-0_all.deb
$ sudo service nv_peer_mem start
```

   Verify the `nv_peer_mem` service is running:

```
$ sudo service nv_peer_mem status
```

   Enable the `nv_peer_mem` service at boot time:

```
$ sudo systemctl enable nv_peer_mem
$ sudo /lib/systemd/systemd-sysv-install enable nv_peer_mem
```

8.3 Installing Rivermax SDK

Note: If Rivermax SDK was previously installed via SDK Manager, the download and install steps (1 and 2) can be skipped. **The Rivermax license must still be installed, however, so step 3 must still be followed**.

1. Download version **1.8.21** or newer of the Rivermax SDK from the NVIDIA Rivermax SDK developer page.
   a. Click **Get Started** and login using your NVIDIA developer account.
   b. Scroll down to **Downloads** and click **I Agree To the Terms of the NVIDIA Rivermax Software Licence Agreement**
   c. Select **Rivermax SDK 1.8.21**, **Linux**, then download **rivermax_ubuntu1804_1.8.21.tar.gz**. If a newer version is available, replace **1.8.21** in this and all following steps with the newer version that is available.
2. Install Rivermax SDK:

```
$ tar -xvf rivermax_ubuntu1804_1.8.21.tar.gz
$ sudo dpkg -i 1.8.21/Ubuntu.18.04/deb-dist/aarch64/rivermax_11.3.9.21_
↪arm64.deb
```

3. Install Rivermax License

   Using Rivermax requires a valid license, which can be purchased from the Rivermax Licenses page. Once the license file has been obtained, it must be placed onto the system using the following path:

```
/opt/mellanox/rivermax/rivermax.lic
```

8.4 Testing Rivermax and GPUDirect

Running the Rivermax sample applications requires two systems, a sender and a receiver, connected via ConnectX network adapters. If two Clara AGX Developer Kits are used then the onboard ConnectX-6 can be used on each system, but if only one Clara AGX is available then it's expected that another system with an add-in ConnectX network adapter will need to be used. Rivermax supports a wide array of platforms, including both Linux and Windows, but these instructions assume that another Linux based platform will be used as the sender device while the Clara AGX is used as the receiver.

1. Determine the logical name for the ConnectX devices that are used by each system. This can be done by using the `lshw -class network` command, finding the `product:` entry for the ConnectX device, and making note of the `logical name:` that corresponds to that device. For example, this output on a Clara AGX shows the onboard ConnectX-6 device using the `enp9s0f01` logical name (`lshw` output shortened for demonstration purposes).

```
$ sudo lshw -class network
*-network:0
     description: Ethernet interface
     product: MT28908 Family [ConnectX-6]
```

(continues on next page)

(continued from previous page)

```
    vendor: Mellanox Technologies
    physical id: 0
    bus info: pci@0000:09:00.0
    logical name: enp9s0f0
    version: 00
    serial: 48:b0:2d:13:9b:6b
    capacity: 10Gbit/s
    width: 64 bits
    clock: 33MHz
    capabilities: pciexpress vpd msix pm bus_master cap_list ethernet␣
↪physical 1000bt-fd 10000bt-fd autonegotiation
    configuration: autonegotiation=on broadcast=yes driver=mlx5_core␣
↪driverversion=5.4-1.0.3 duplex=full firmware=20.27.4006 (NVD0000000001)␣
↪ip=10.0.0.2 latency=0 link=yes multicast=yes
    resources: iomemory:180-17f irq:33 memory:1818000000-1819ffffff
```

The instructions that follow will use the `enp9s0f0` logical name for `ifconfig` commands, but these names should be replaced with the corresponding logical names as determined by this step.

2. Run the `generic_sender` application on the sending system.

   a. Bring up the network:

```
$ sudo ifconfig enp9s0f0 up 10.0.0.1
```

   b. Build the sample apps:

```
$ cd 1.8.21/apps
$ make
```

Note: The `1.8.21` path above corresponds to the path where the Rivermax SDK package was extracted in step 2 of the *Installing Rivermax SDK* section, above. If the Rivermax SDK was installed via SDK Manager, this path will be `$HOME/Documents/Rivermax/1.8.21`.

   e. Launch the `generic_sender` application:

```
$ sudo ./generic_sender -l 10.0.0.1 -d 10.0.0.2 -p 5001 -y 1462 -k␣
↪8192 -z 500 -v
...
+##############################################
| Sender index: 0
| Thread ID: 0x7fa1ffb1c0
| CPU core affinity: -1
| Number of streams in this thread: 1
| Memory address: 0x7f986e3010
| Memory length: 59883520[B]
| Memory key: 40308
+##############################################
| Stream index: 0
```

(continues on next page)

(continued from previous page)

```
| Source IP: 10.0.0.1
| Destination IP: 10.0.0.2
| Destination port: 5001
| Number of flows: 1
| Rate limit bps: 0
| Rate limit max burst in packets: 0
| Memory address: 0x7f986e3010
| Memory length: 59883520[B]
| Memory key: 40308
| Number of user requested chunks: 1
| Number of application chunks: 5
| Number of packets in chunk: 8192
| Packet's payload size: 1462
+**********************************************
```

3. Run the `generic_receiver` application on the receiving system.

   a. Bring up the network:

```
$ sudo ifconfig enp9s0f0 up 10.0.0.2
```

   b. Build the sample apps with GPUDirect support (`CUDA=y`):

```
$ cd 1.8.21/apps
$ make CUDA=y
```

> **Note:** The `1.8.21` path above corresponds to the path where the Rivermax SDK package was extracted in step 2 of the *Installing Rivermax SDK* section, above. If the Rivermax SDK was installed via SDK Manager, this path will be `$HOME/Documents/Rivermax/1.8.21`.

   c. Launch the `generic_receiver` application:

```
$ sudo ./generic_receiver -i 10.0.0.2 -m 10.0.0.2 -s 10.0.0.1 -p␣
↪5001 -g 0
...
Attached flow 1 to stream.
Running main receive loop...
Got 5877704 GPU packets | 68.75 Gbps during 1.00 sec
Got 5878240 GPU packets | 68.75 Gbps during 1.00 sec
Got 5878240 GPU packets | 68.75 Gbps during 1.00 sec
Got 5877704 GPU packets | 68.75 Gbps during 1.00 sec
Got 5878240 GPU packets | 68.75 Gbps during 1.00 sec
...
```

With both the `generic_sender` and `generic_receiver` processes active, the receiver will continue to print out received packet statistics every second. Both processes can then be terminated with `<ctrl-c>`

8.5 GPUDirect and CUDA Sample

GPUDirect is ideal for applications which receive data from the network adapter and then use the GPU to process the received data directly in GPU memory. The `generic_sender` and `generic_receiver` demo applications include a simple demonstration of the use of CUDA with received packets by using a CUDA kernel to compute and then compare a checksum of the packet against an expected checksum as provided by the sender. This additional checksum packet included by the sender also includes a packet sequence number that is used by the receiver to detect when any packets are lost during transmission.

In order to enable the CUDA checksum sample, append the `-x` parameter to the `generic_sender` and `generic_receiver` commands that are run above.

Due to the increased workload by the receiver when the checksum calculation is enabled, you will begin to see dropped packets and/or checksum errors if you try to maintain the same data rate from the sender as you did when the checksum was disabled (i.e. when all received packet data was simply discarded). Because of this the sleep parameter used by the sender, `-z`, should be increased until there are no more dropped packets or checksum errors. In this example, the sleep parameter was increased from `500` to `40000` in order to ensure the receiver can receive and process the sent packets without any errors or loss:

```
[Sender]
$ sudo ./generic_sender -l 10.0.0.1 -d 10.0.0.2 -p 5001 -y 1462 -k 8192 -z␣
↪40000 -v -x

[Receiver]
$ sudo ./generic_receiver -i 10.0.0.2 -m 10.0.0.2 -s 10.0.0.1 -p 5001 -g 0 -x
...
Got  203968 GPU packets | 2.40 Gbps during 1.02 sec | 0 dropped packets | 0␣
↪checksum errors
Got  200632 GPU packets | 2.36 Gbps during 1.00 sec | 0 dropped packets | 0␣
↪checksum errors
Got  203968 GPU packets | 2.40 Gbps during 1.01 sec | 0 dropped packets | 0␣
↪checksum errors
Got  201608 GPU packets | 2.37 Gbps during 1.01 sec | 0 dropped packets | 0␣
↪checksum errors
```

If you would like to write an application that uses Rivermax and GPUDirect for CUDA data processing, refer to the source code for the `generic_sender` and `generic_receiver` applications included with the Rivermax SDK in `generic_sender.cpp` and `generic_receiver.cpp`, respectively.

Note: The CUDA checksum calculation in the `generic_receiver` is included only to show how the data received through GPUDirect can be processed through CUDA. This example is **not optimized in any way**, and should not be used as an example of how to write a high-performance CUDA application. Please refer to the CUDA Best Practices Guide for an introduction to optimizing CUDA applications.

8.6 Troubleshooting

If running the driver installation or sample applications do not work, check the following.

1. The ConnectX network adapter is recognized by the system. For example, on a Linux system using a ConnectX-6 Dx add-in PCI card:

```
$ lspci
...
0000:05:00.0 Ethernet controller: Mellanox Technologies MT28841
0000:05:00.1 Ethernet controller: Mellanox Technologies MT28841
...
```

If the network adapter is not recognized, try rebooting the system and/or reseating the card in the PCI slot.

2. The ConnectX network adapter is recognized by the OFED driver. For example, on a Linux system using a ConnectX-6 Dx add-in PCI card:

```
  $ sudo mlxfwmanager
  ...
  Device Type:      ConnectX6DX
  Part Number:      MCX623106AC-CDA_Ax
  Description:      ConnectX-6 Dx EN adapter card; 100GbE; Dual-port␣
↪QSFP56; PCIe 4.0 x16; Crypto and Secure Boot
  PSID:             MT_0000000436
  PCI Device Name:  /dev/mst/mt4125_pciconf0
  Base GUID:        0c42a1030024053a
  Base MAC:         0c42a124053a
  Versions:         Current        Available
     FW             22.31.1014     N/A
     FW (Running)   22.30.1004     N/A
     PXE            3.6.0301       N/A
     UEFI           14.23.0017     N/A

If the device does not appear, first try rebooting and then
:ref:`reinstalling OFED <installing_ofed>` as described above.
```

3. The sender and reciever systems can ping each other:

```
$ ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.205 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.206 ms
...
```

If the systems can not ping each other, try bringing up the network interfaces again using the `ifconfig` commands.

4. The **nv_peer_mem** service is running:

```
$ sudo service nv_peer_mem status
* nv_peer_mem.service - LSB: Activates/Deactivates nv_peer_mem to \ start␣
↪at boot time.
   Loaded: loaded (/etc/init.d/nv_peer_mem; generated)
   Active: active (exited) since Mon 2021-01-25 16:45:08 MST; 9min ago
     Docs: man:systemd-sysv-generator(8)
  Process: 6847 ExecStart=/etc/init.d/nv_peer_mem start (code=exited,␣
↪status=0/SUCCESS)
```

(continues on next page)

(continued from previous page)

```
Jan 25 16:45:08 mccoy systemd[1]: Starting LSB: Activates/Deactivates nv_
↪peer_mem to \ start at boot time....
Jan 25 16:45:08 mccoy nv_peer_mem[6847]: starting... OK
Jan 25 16:45:08 mccoy systemd[1]: Started LSB: Activates/Deactivates nv_
↪peer_mem to \ start at boot time..
```

If the service is not running, try starting it again using `sudo service nv_peer_mem start`.

Chapter 9

AJA Video Systems

AJA provides a wide range of proven, professional video I/O devices, and a partnership between NVIDIA and AJA has led to the addition of Clara AGX support to the AJA NTV2 SDK and device drivers as of the NTV2 SDK 16.1 release.

GPU compute performance is a key component of the Clara AGX platform, and to optimize GPU based video processing applications the AJA drivers and SDK now offer RDMA support for NVIDIA GPUs. This feature allows video data to be captured directly from the AJA card to GPU memory, which significantly reduces latency and system PCI bandwidth for GPU video processing applications as sysmem to GPU copies are eliminated from the processing pipeline.

RDMA support from AJA devices has also been incorporated into the AJA GStreamer plugin in order to enable zero-copy GPU buffer integration with the DeepStream SDK. This support allows DeepStream applications to process video data along the entire pipeline, from the initial capture to final display, without ever leaving GPU memory.

The following instructions describe the steps required to setup and use an AJA device with the Clara AGX platform, including RDMA and DeepStream integration. Note that the AJA NTV2 SDK support for Clara AGX includes all of the AJA Developer Products, though the following instructions have only been verified for the Corvid 44 12G BNC and KONA HDMI products, specifically.

9.1 Installing the AJA NTV2 SDK and Drivers

1. Download the **AJA NTV2 SDK**

   a. Login to your AJA developer account at https://sdksupport.aja.com.

   > **Note:** Access to the AJA NTV2 SDK requires access to the AJA developer program, which can be requested from https://aja.com/developer/request.

   b. Under **Knowledgebase** on the left side, go to **NTV2 SDK Downloads**.

   c. Under **Current NTV2 SDKs**, go to **NTV2 SDK 16.1 for Linux, MacOS and Windows**.

   d. At the bottom of the page, under **Attachments**, download `ntv2sdklinux_16.1.0.3.zip` and `ntv2sdklinux_16.1.0.3_arm.zip`. The first zip file contains the NTV2 SDK itself, while the second zip file contains libraries and binaries that have been built for the arm64 architecture as required by Clara AGX.

Note: If a newer version of the 16.1 SDK has been released, replace `16.1.0.3` above and in all following steps with the version that is available for download.

e. Move the downloaded zip files to a new directory, e.g. `${HOME}/aja`. In all following steps, this path will be referred to as `AJA_BASE`.

```
$ export AJA_BASE=${HOME}/aja
$ mkdir ${AJA_BASE}
$ mv ~/Downloads/ntv2sdklinux_16.1.0.3.zip ${AJA_BASE}
$ mv ~/Downloads/ntv2sdklinux_16.1.0.3_arm.zip ${AJA_BASE}
```

2. Build and install the AJA drivers with RDMA support

a. Extract the NTV2 SDK zip files.

```
$ cd ${AJA_BASE}
$ export NTV2_SDK=ntv2sdklinux_16.1.0.3
$ unzip ${NTV2_SDK}.zip
$ unzip ${NTV2_SDK}_arm.zip
```

b. Copy the arm64 libraries to the main SDK directory.

```
$ cp ${AJA_BASE}/${NTV2_SDK}_arm/lib/* ${AJA_BASE}/${NTV2_SDK}/lib/
```

c. Build the AJA drivers with RDMA support.

```
$ cd ${AJA_BASE}/${NTV2_SDK}/ajadriver/linux
$ export AJA_RDMA=1
$ make
```

d. Load the AJA drivers.

```
$ sudo ${AJA_BASE}/${NTV2_SDK}/bin/load_ajantv2
  loaded ajantv2 driver module
  created node /dev/ajantv20
```

e. Ensure that the AJA driver and RDMA are working by using the `rdmawhacker` utility included with the NTV2 SDK.

```
$ cd ${AJA_BASE}/${NTV2_SDK}/ajaapps/crossplatform/rdmawhacker
$ make
$ ${AJA_BASE}/${NTV2_SDK}/bin/rdmawhacker

  DMA engine 1 WRITE 8388608 bytes  rate: 4678.72 MB/sec  584.84␣
↪xfers/sec

  <Ctrl-C to exit>
```

9.2 Installing the AJA GStreamer Plugin

1. Clone the **ntv2-gst** git repo.

```
$ cd ${AJA_BASE}
$ git clone https://github.com/aja-video/ntv2-gst.git
```

2. Setup the build environment and paths.

```
$ sudo apt install libgstreamer1.0-dev libgstreamer-plugins-base1.0-dev
$ sudo ln -sf /usr/lib/aarch64-linux-gnu/tegra/libnvdsbufferpool.so /opt/
↪nvidia/deepstream/deepstream-5.1/lib/
$ export GST_CUDA=/usr/local/cuda-11.1/targets/sbsa-linux
$ export GST_DEEPSTREAM=/opt/nvidia/deepstream/deepstream-5.1
$ export GST_NTV2=${AJA_BASE}/${NTV2_SDK}
```

Note: If the following error is output when creating the symlink using the `ln` command above, make sure that DeepStream has been installed as described in *Reinstalling Clara AGX SDK Packages*.

```
ln: target '/opt/nvidia/deepstream/deepstream-5.1/lib/' is not a directory
```

3. Build and install the plugin.

```
$ cd ${AJA_BASE}/ntv2-gst/gst-plugin
$ ./autogen.sh
$ make
$ sudo make install
$ sudo ln -sf /usr/local/lib/gstreamer-1.0/libgstaja.so /usr/lib/aarch64-
↪linux-gnu/gstreamer-1.0/
```

9.3 Testing the AJA GStreamer Plugin

The following `gst-launch-1.0` command can be used to test the AJA GStreamer plugin.

```
$ gst-launch-1.0 ajavideosrc mode=720p5994-rgba nvmm=true ! nv3dsink
```

The above assumes that a 720p @ 59.94 FPS video stream is being broadcast to SDI channel 1 (i.e. plugin index 0) of the AJA SDI video capture card. This video stream may be generated by any SDI source, but for testing purposes it may be useful to use the `ntv2player` utility that is included with the NTV2 SDK in order to generate a known test pattern. This test pattern can even be generated by the same AJA card that is receiving the stream by connecting a loopback between two channels on the same AJA card.

For example, if a loopback is created by connecting an SDI cable between channels 1 and 2 on the same AJA card, the following command can be used to output a test signal on channel 2 that is then received on channel 1 using the `gst-launch-1.0` command above.

```
$ ${AJA_BASE}/${NTV2_SDK}_arm/bin/ntv2player -v=720p -c=2
```

To use AJA KONA HDMI card `input-mode=hdmi` needs to be set in `gst-launch-1.0` command. For reference, see the following command.

```
$ gst-launch-1.0 ajavideosrc mode=UHDp30-rgba input-mode=hdmi nvmm=true !␣
↪nv3dsink
```

If issues are encounted when running these commands, see *Troubleshooting*, below.

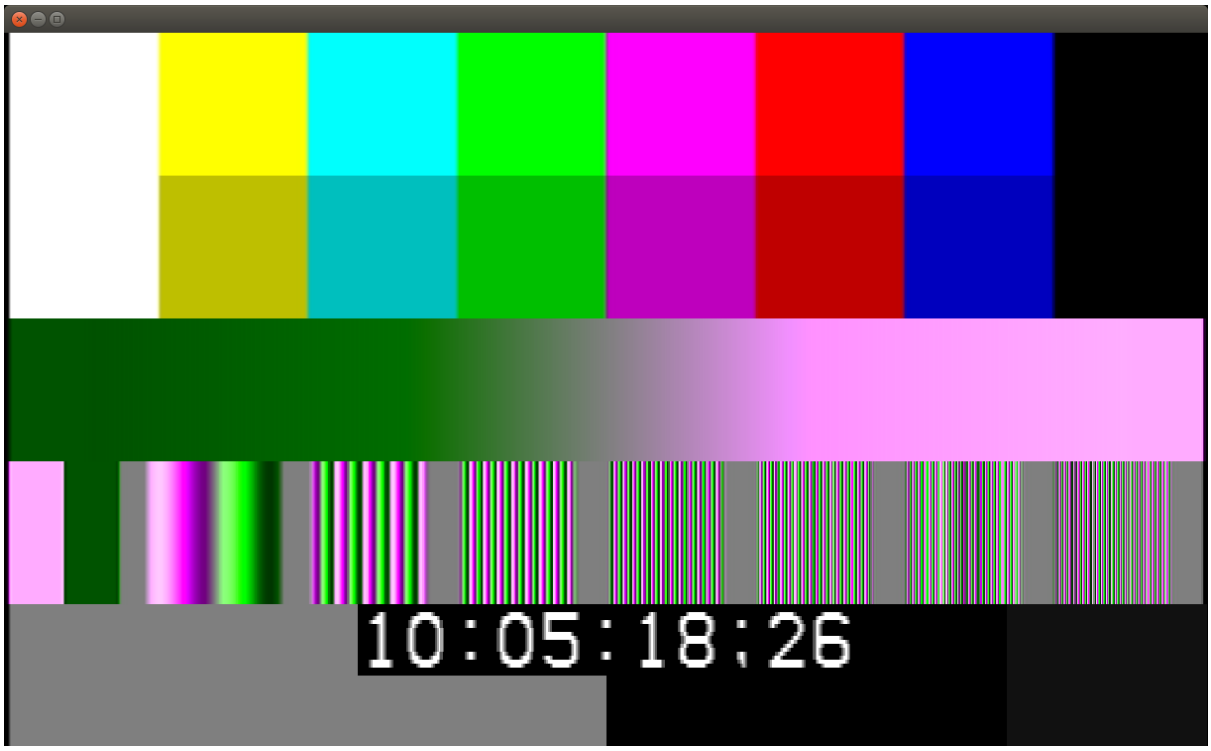


Fig. 9.1: AJA GStreamer plugin receiving ntv2player test pattern

Note: The `nvmm=true` parameter that is provided to the `ajavideosrc` plugin is what enables the use of RDMA so that the buffers written by the plugin are output directly to GPU memory via NVMM buffers (i.e. using the GStreamer caps `video/x-raw(memory:NVMM)`).

Enabling RDMA also requires the use of an `rgba` mode due to GPU support limitations. See *Problem 3* in the *Troubleshooting* section below for more details.

9.4 Using DeepStream with the AJA GStreamer Plugin

Since the AJA plugin using RDMA outputs to GPU (`video/x-raw(memory:NVMM)`) buffers directly, an RDMA stream from the AJA plugin may be passed directly into DeepStream without requiring any additional buffer conversions or transfers between sysmem and GPU. Assuming that the Clara AGX DeepStream sample is installed, the following command will run a GStreamer pipeline that captures a stream from the AJA card using RDMA and passes it through DeepStream using the sample model files included with the DeepStream sample.

```
$ gst-launch-1.0 ajavideosrc mode=720p5994-rgba nvmm=true \
    ! m.sink_0 nvstreammux name=m width=1280 height=720 batch-size=1 \
    ! nvinfer config-file-path=/opt/nvidia/clara-agx-sdk/clara-agx-deepstream-
↪sample/build/DeepstreamJetsonAGXSample/dsjas_nvinfer_config.txt \
    ! nvdsosd ! nv3dsink sync=false
```

If issues are encounted when running this commands, see *Troubleshooting*, below.

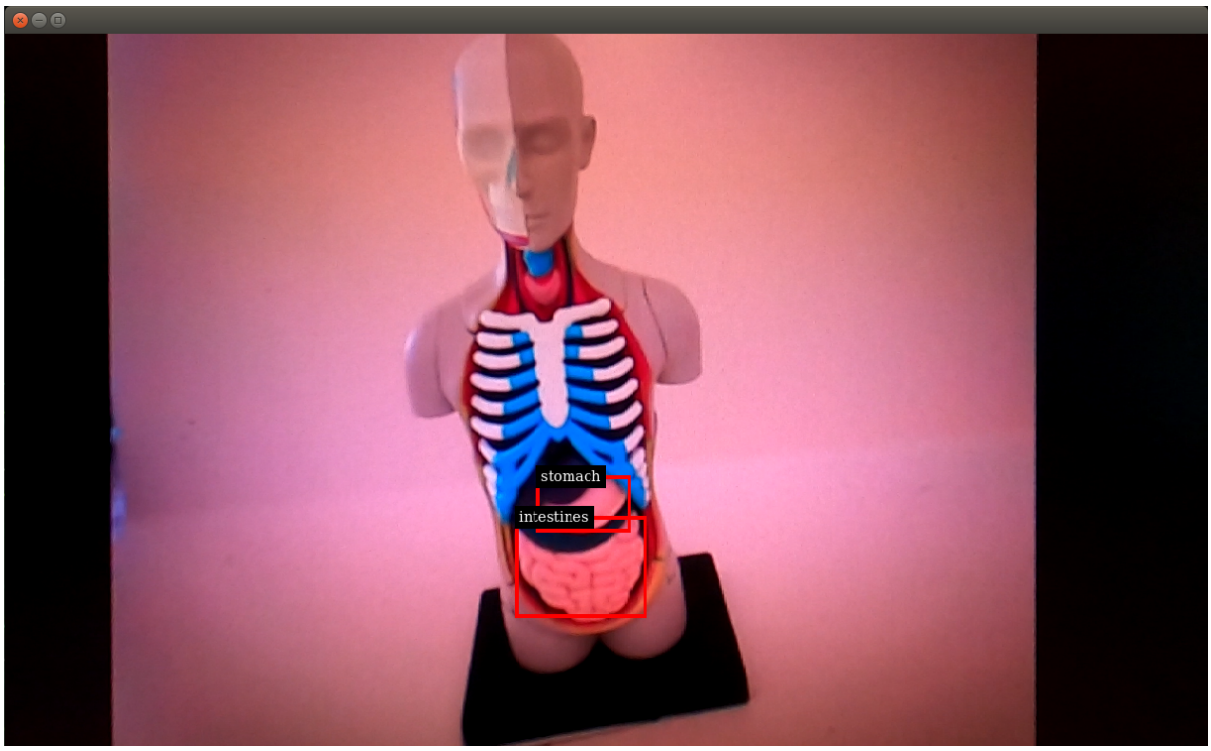


Fig. 9.2: AJA integration with DeepStream

9.5 Troubleshooting

If the *gst-launch-1.0* commands described in steps above fail to render a video stream onto the display, the following steps may help resolve the issue.

1. **Problem:** The command fails with the error:

   ```
   ERROR: from element /GstPipeline:pipeline0/GstNv3dSink:nv3dsink0:
   GStreamer error: state change failed and some element failed to post
   a proper error message with the reason for the failure.
   ```

   Solution: Make sure the `DISPLAY` environment variable is set (replace `:0` with the ID of the X11 display you are using):

   ```
   $ export DISPLAY=:0
   ```

2. **Problem:** The command fails with the error:

```
WARNING: from element /GstPipeline:pipeline0/
↪GstAjaVideoSrc:ajavideosrc0: Signal lost
Additional debug info:
gstajavideosrc.cpp(1325): gst_aja_video_src_create (): /
↪GstPipeline:pipeline0/GstAjaVideoSrc:ajavideosrc0:
No input source was detected - video frames invalid
```

Solution: This error occurs when the `mode` parameter provided to the `ajavideosrc` plugin does not match the video format that is being received by the AJA card.

The video format that is being received by the AJA card as well as the format that is being requested by the plugin can be checked by using the `ntv2watcher` utility that is included with the NTV2 SDK:

```
$ sudo apt install libqt5multimedia5
$ ${AJA_BASE}/${NTV2_SDK}_arm/bin/ntv2watcher
```

In the app, select *Routing* on the left navigation bar to see the current status and pipeline setup of the AJA card:

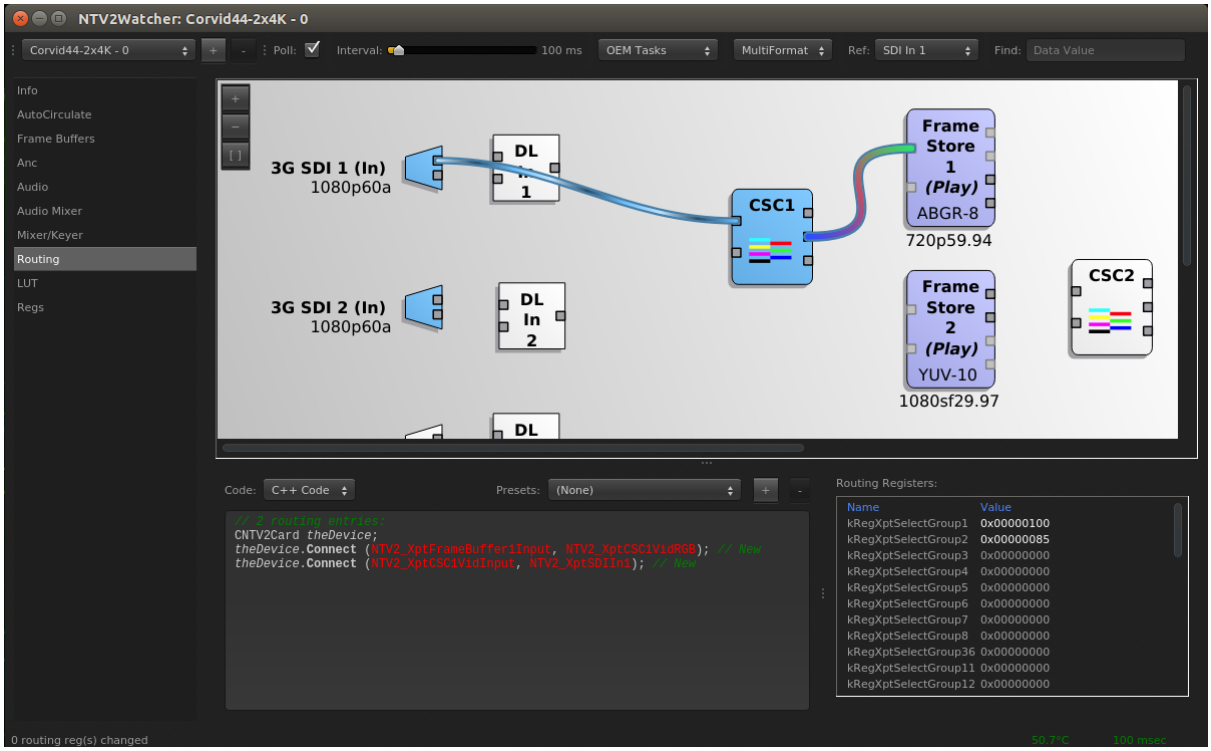


Fig. 9.3: Routing display of the ntv2watcher utility

In this example, we can see that SDI channel 1 is currently receiving a `1080p60a` video signal while the pipeline setup by the plugin is expecting a `720p59.94` signal (Frame Store 1). To resolve this, the `mode` parameter for the plugin should be changed to match the signal that is being received.

To view this list of modes that are supported by the plugin, use the `gst-inspect-1.0` utility (the following output is shortened for illustration purposes):

```
$ gst-inspect-1.0 ajavideosrc
...
mode : Video Mode to use for playback
       flags: readable, writable
       Enum "GstAjaRawModes" Default: 15, "HD720 8Bit 59.94p"
          (0): NTSC 8Bit 23.98i              - ntsc-2398
          (1): NTSC 8Bit 24i                 - ntsc-24
          (2): NTSC 8Bit 59.94i              - ntsc
          (3): NTSC 8Bit RGBA 23.98i         - ntsc-2398-rgba
          (4): NTSC 8Bit RGBA 24i            - ntsc-24-rgba
          (5): NTSC 8Bit RGBA 59.94i         - ntsc-rgba
          ...
          (38): HD1080 8Bit RGBA 23.98p      - 1080p2398-rgba
          (39): HD1080 8Bit RGBA 24p         - 1080p24-rgba
          (40): HD1080 8Bit RGBA 25p         - 1080p25-rgba
          (41): HD1080 8Bit RGBA 29.97p      - 1080p2997-rgba
          (42): HD1080 8Bit RGBA 30p         - 1080p30-rgba
          (43): HD1080 8Bit RGBA 50i         - 1080i50-rgba
          (44): HD1080 8Bit RGBA 50p         - 1080p50-rgba
          (45): HD1080 8Bit RGBA 59.94i      - 1080i5994-rgba
          (46): HD1080 8Bit RGBA 59.94p      - 1080p5994-rgba
          (47): HD1080 8Bit RGBA 60i         - 1080i60-rgba
          (48): HD1080 8Bit RGBA 60p         - 1080p60-rgba
          ...
```

In this particular example, the `1080p60-rgba` mode can be used for the plugin in order to match the signal that is being received:

```
$ gst-launch-1.0 ajavideosrc mode=1080p60-rgba nvmm=true ! nv3dsink
```

3. **Problem:** The command fails with the error:

```
WARNING: erroneous pipeline: could not link ajavideosrc0 to nv3dsink0
```

Solution: The reason for this error is that the video format that is being requested by the `ajavideosrc` plugin (i.e. the `mode` parameter) is not supported by the `nv3dsink` plugin.

The most likely reason for this incompatibility is that the `nv3dsink` – and all NVIDIA GStreamer/DeepStream plugins – do not support the packed `UYVY` (8-bit) or `v210` (10-bit) YUV 4:2:2 formats that are output by the `ajavideosrc` plugin. Because of this, an RGBA mode must be used with the `ajavideosrc` plugin when it is used with NVIDIA plugins.

In order to select an RGBA mode, ensure that the `mode` parameter that is given to the `ajavideosrc` plugin has the `-rgba` suffix, e.g. `1080p60-rgba`.

4. **Problem:** The command fails with the error:

```
ERROR: [TRT]: INVALID_CONFIG: Deserialize the cuda engine failed.
```

Solution: The reason for this error is that the DeepStream engine files for the DeepStream sample application have not been generated. See *Creating TensorRT Engines* for the steps to resolve this issue.

Chapter 10

Data Set Generation

The Clara AGX SDK includes a sample data set that consists of 300 images of the Learning Resources Anatomy Model that has been annotated with `stomach` and `intestines` regions. The sample applications by default use a model that has been trained using this sample dataset, allowing the samples to be run out of the box without having to generate a new data set or train a new model. This sample data set has been included in the Clara AGX TLT package in the `data/organs` subdirectory, and is also the default in all training-related scripts included in this package.

Once the Clara AGX SDK sample applications have been installed and tested, a common next step for the user is to create their own data set and to train a new inference model that can be loaded by the sample application in order to perform inference using the user-provided data set.

There are many ways for data sets to be captured and annotated for training purposes, and these instructions describe just one method that has been used and simplified using the scripts included in the Clara AGX TLT package.

These instructions must be followed on an x86 Linux PC with a camera device available.

10.1 Capturing Images

The `bin/capture.sh` script assists in capturing images for the data set from a camera attached to the PC. When run, this script will run continuously and take pictures whenever the spacebar is pressed (or until ESC is used to quit).

This script depends on FFMPEG, which can be installed with the following command:

```
$ sudo apt install ffmpeg
```

We recommend capturing at least 300 images in a wide range of angles and lighting conditions in order to generate a useful inference model. If 300 pictures does not produce acceptable inference results once the model has been trained, it may be desirable to revisit this step and take even more.

Store the images output by this step into a single directory (e.g. `/shared/tlt/workspace/data/organs`).

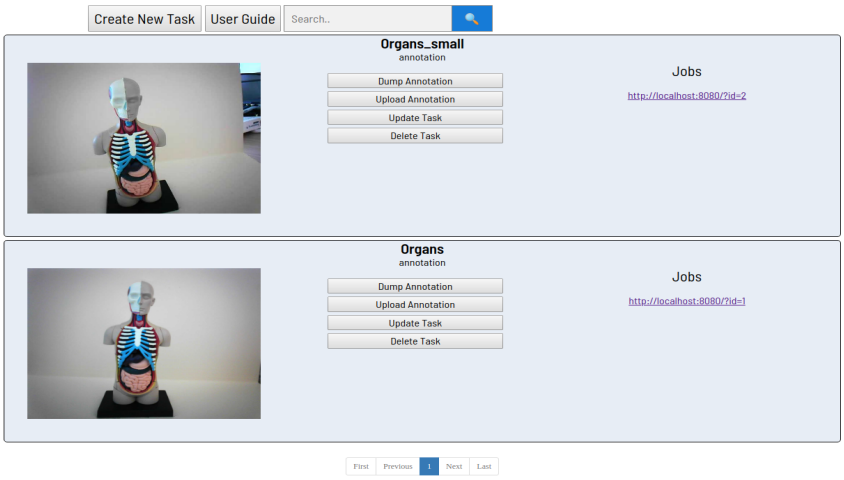
Note: If the camera you are using for this step is not `/dev/video0`, you will need to change the device via the `--dev <dev>` parameter.

Note: The steps in this documentation assume that you are overwriting the sample *organs* data set with your own custom data set. This simplifies the process since many of the scripts included have been hard coded to use specific paths and the *organs* data set name. If you would like to change the name or path for your data set, it may be required to modify the scripts that are used to reflect the new names or paths.

10.2 Labeling the Data Set

Once you have a data set, you will need to label it. These instructions use CVAT from OpenCV to perform labeling.

1. Install CVAT using these instructions.
2. Use Google Chrome to navigate to http://localhost:8080/dashboard/. This will launch CVAT.
3. Register a new user for CVAT. See the CVAT User Guide for more details.

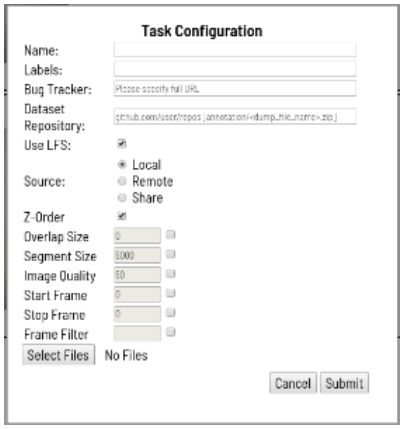


4. Click **Create New Task** and specify the following:

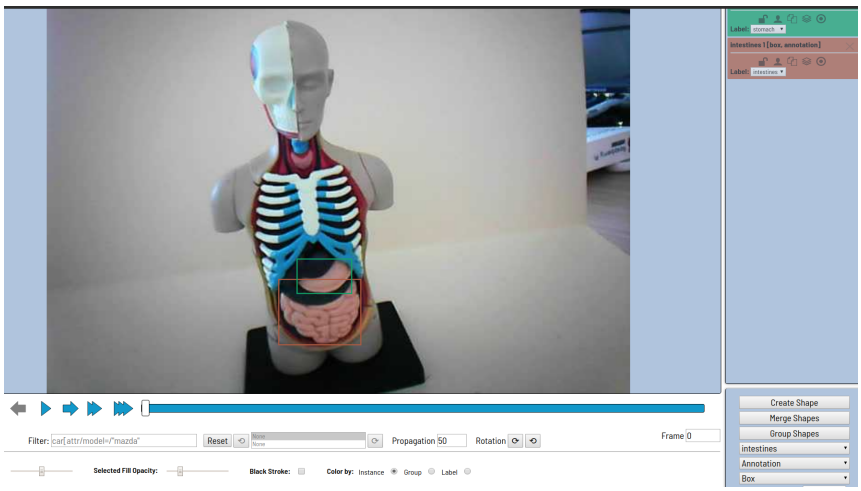
a. **Name**: Enter a name for the task (e.g. “Organs”).

b. **Labels**: Add all labels that will be used with the data set as a space-separated list (e.g. “stomach intestines liver”).

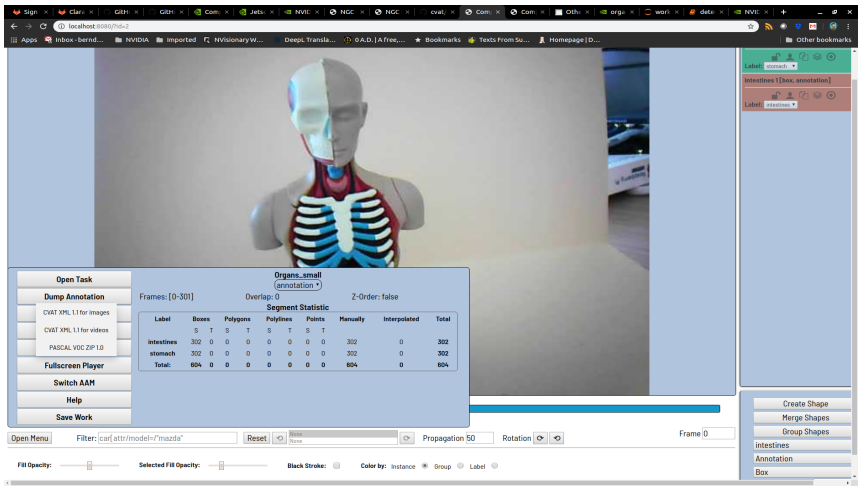
c. **Select Files**: Select the folder containing your image data set.



5. Click **Submit** to create the task.
6. Click the **Jobs** link associated with your new task.
7. Click **Create Shape** to create a bounding box for the selected label. Repeat this step for all labels added in step 4b.



8. Go to the next image and repeat step 7. Repeat this step until all images are labeled.
9. Select **Open Menu > Save Work**.
10. Select **Open Menu > Dump Annotation**. Then select **PASCAL VOC ZIP 1.0**. This should generate a ZIP file containing the annotations as XML files.



11. Copy the generated ZIP file to the same directory as your original image data set (e.g. `/shared/tlt/workspace/data/organs`).

Chapter 11

Training the Neural Network

Once a data set has been created and annotated, the Transfer Learning Toolkit (TLT) can be used to train a neural network and output a model that can used for inference by DeepStream and the sample application.

These instructions can be used to train a model using a custom data set that was generated in the previous step, or can simply be followed as-is as an exercise to regenerate a model from the sample data set. Note that the sample applications already include the sample models, however, so this step is optional if not replacing the sample model in the application.

Additionally many pre-trained models for object detection, segmentation, pose estimation, heart rate estimation, and more can be found as part of the Transfer Learning Toolkit. Further documentation for training these models can be found here: https://docs.nvidia.com/tlt/archive/tlt-30/text/quickstart/deepstream_integration.html

1. Get an NGC account via https://ngc.nvidia.com/signin.
2. Follow the steps to Install the TLT launcher from the Transfer Learning Toolkit 3.0 Documentation: https://docs.nvidia.com/tlt/archive/tlt-30/text/tlt_launcher.html#installing-the-launcher
3. Configure the TLT launcher so that the root of the Clara AGX TLT directory is mounted to `/workspace/tlt-experiments` in the TLT container. This can be done using the `bin/setup_tlt_mounts.sh` utility script in the Clara AGX TLT directory:

   ```
   $ bin/setup_tlt_mounts.sh
   ```

   For complete details about the TLT launcher configuration, see Running the TLT launcher.
4. Run the `bin/detectnet.sh` script and provide the name of the dataset.

   ```
   $ bin/detectnet.sh --dataset <dataset name>
   ```

   Where `<dataset name>` would be `organs` if using the sample dataset.

Note: The `bin/detectnet.sh` script requires the dataset to meet these requirements:

- All images start with `<dataset-name>-`.
- All annotations start with `<dataset-name>-`.
- The images and annotation files start at '0' and are numbered consecutively without missing numbers.
- The annotations ZIP file is named `<dataset-name>.zip`.

- The model-specific spec files are located in `spec/<dataset-name>/`.

5. The previous step trains the neural network using TLT and outputs the trained models (the `*.etlt` files) to the Clara AGX TLT directory in `results/experiment_dir_final`.

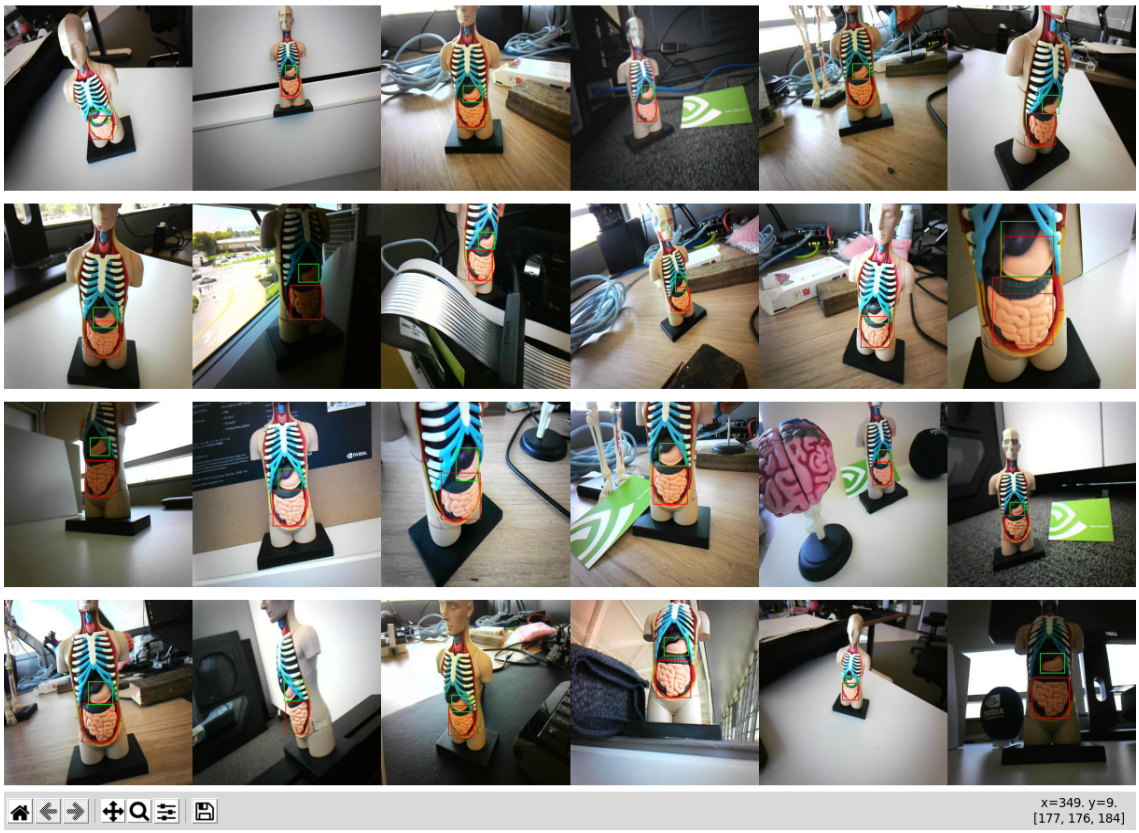
   In order to use the new models with the DeepStream sample application, the models must be run through the `tlt-converter` utility in order to generate the TensorRT engine files (`*.trt`). See *Creating TensorRT Engines* for the steps to generate the engine files.

6. If a display is available, the results of the trained model can be visualized by running `bin/visualize_test_results.py`. This will render a selection of test images that have been annotated using the inference results from using the newly trained model.

   Note that this utility depends on the `matplotlib` package, which also depends on the `python3-tk` package for the GUI to work when using `virtualenv`.

```
$ pip3 install matplotlib
$ sudo apt-get install python3-tk

$ python3 bin/visualize_test_results.py
```



Chapter 12

Creating TensorRT Engines

In order to use TLT-trained models (`*.etlt`) with the DeepStream sample application, the `tlt-converter` utility must be used in order to generate new TensorRT engine files (`*.trt`) on the device that they will be used. TensorRT engine files are specific to the platform, GPU, and TensorRT version that is being used, and so these steps must be followed if any of these change.

Important: Perform these steps on the device that will be using the engine files (i.e. the device that will be running the DeepStream application).

---

1. Download the TensorRT `tlt-converter` binary for your platform and extract it to a location accessible through $PATH (e.g. `/usr/bin`).
   - Jetson AGX (TensorRT 7.2): https://developer.nvidia.com/tlt-converter-1
   - Linux x86 (TensorRT 7.2): https://developer.nvidia.com/cuda111-cudnn80-trt72
2. Copy the following model files to a new local directory.
   - `calibration.bin`
   - `resnet18_detector.etlt`
   - `resnet18_detector_int8.etlt`

   If you are simply regenerating the engine files from the models that were included with the DeepStream sample application (that is, you did not train your own model files), these files will be located in `/opt/nvidia/clara-agx-sdk/clara-agx-deepstream-sample/model/organs/`.

   If you are generating the engine files for a new model that you trained using the *Training the Neural Network* steps above, these files will be located in the `results/experiment_dir_final` directory that was output by the training steps.
3. Copy the `bin/detectnet_convert.sh` script from the Clara AGX TLT package to the same directory as the model files above.
4. Run the `./detectnet_convert.sh` script.
5. The previous step creates the TensorRT engine files (`*.trt`) and outputs them alongside the model files in the current directory. To use these new engines, copy them to the DeepStream application's `model` directory (e.g. `/opt/nvidia/clara-agx-sdk/clara-agx-deepstream-sample/build/DeepstreamLinuxHostSample/model`) or reference them from the DeepStream appli-

cation configuration. Please see https://docs.nvidia.com/tlt/tlt-user-guide/text/deepstream_tlt_integration.html for more details integrating DeepStream with TLT.

Chapter 13

DeepStream and Endoscopy Sample Apps

13.1 Clara AGX Developer Kit and Jetson AGX

13.1.1 Installation

Install the Debian file for the Clara AGX DeepStream sample using the following command. If Clara AGX SDK was installed using **NVIDIA SDK Manager**, this step can be skipped.

```
$ sudo apt install ./clara-agx-deepstream-sample-src_*_arm64.deb
```

Note: This package has the following dependencies, which will be automatically installed with the above command:

- cmake
- v4l-utils
- libgtk2.0-dev
- libgtk-3-dev
- libgstreamer1.0-dev
- libgstreamer-plugins-base1.0-dev

Important: If you are re-installing or updating the Clara AGX DeepStream Sample package, you need to remove the old version first. See *Updating the DeepStream or Endoscopy Sample Apps* section below for more information.

13.1.2 Running Apps on Clara AGX Development Kit dGPU mode

The DeepStream and Endoscopy sample apps work in dGPU mode on Clara AGX Development Kit; however, the TensorRT engine files that are included with the sample apps are only compatible with the iGPU. Running the apps with the dGPU requires new TensorRT engine files to be generated. To do so, follow instructions given under *Creating TensorRT Engines*.

Important: Camera modules with CSI connectors are not supported in dGPU mode on Clara AGX Development Kit.

13.1.3 Endoscopy Sample Application

Follow these steps to run the Endoscopy Sample Application:

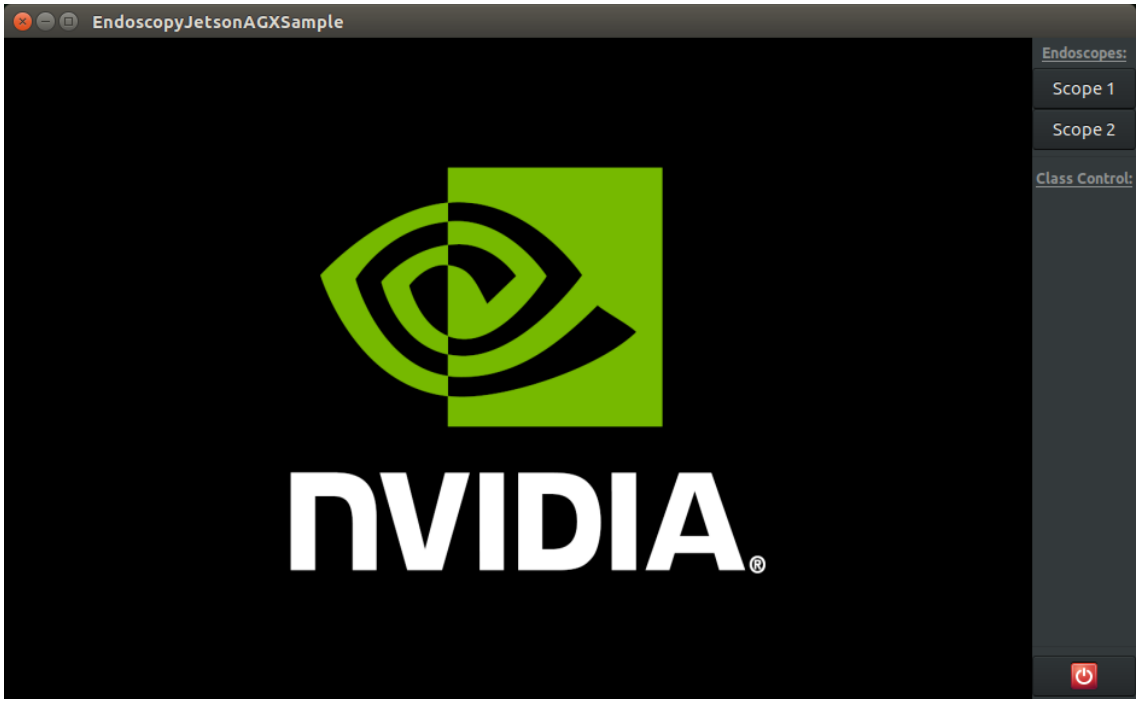
1. Navigate to the installation directory:

```
$ cd /opt/nvidia/clara-agx-sdk/clara-agx-deepstream-sample/build/
↪EndoscopyJetsonAGXSample
```

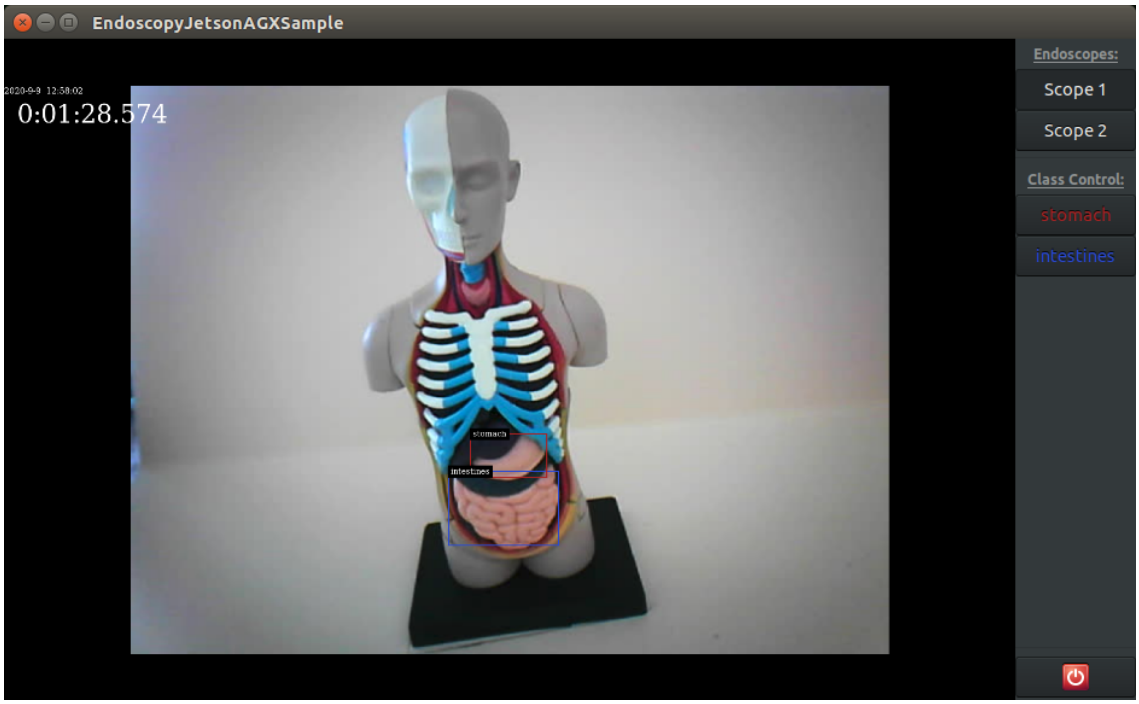
2. Run the application:

```
$ ./EndoscopyJetsonAGXSample
```

The application window should appear.



- All connected cameras should be visible on the top right.
- Select a source by clicking one of the buttons named "`Endoscope <N>`", where <N> corresponds to the video source.
- The main window area should show "Loading…" on the bottom while the camera is being initialized. When ready, the main window area will start displaying the live video stream from the camera.



- Click any of the "`Class Control`" buttons to dynamically enable or disable the bouding boxes and labels of detected objects.
- Click the button on the bottom right to quit the application.

Tip: By default, the app will launch with the FP16 network enabled. You can switch the network to FP32 or INT8 by changing the "`model-engine-file`" and "`network mode`" in the `ejas_nvinfer_config.txt` file.

Tip: By default, the app launches with the configuration parameters defined in the `ejas_config.txt` file. You can either modify start-up parameters there or create your own configuration file and start the application with `$ ./EndoscopyJetsonAGXSample -c <your_config_file>`

13.1.4 DeepStream Sample Application

Follow these steps to run the DeepStream Sample Application:

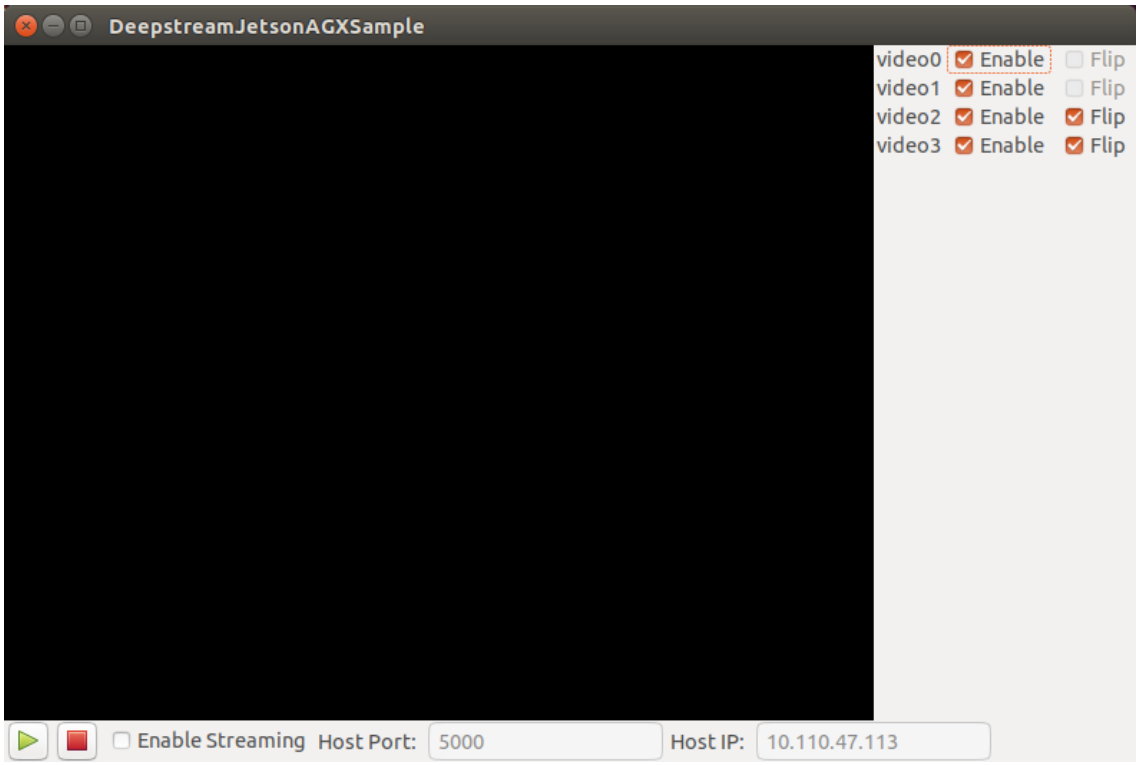
1. Navigate to the installation directory:

```
$ cd /opt/nvidia/clara-agx-sdk/clara-agx-deepstream-sample/build/
↪DeepstreamJetsonAGXSample
```

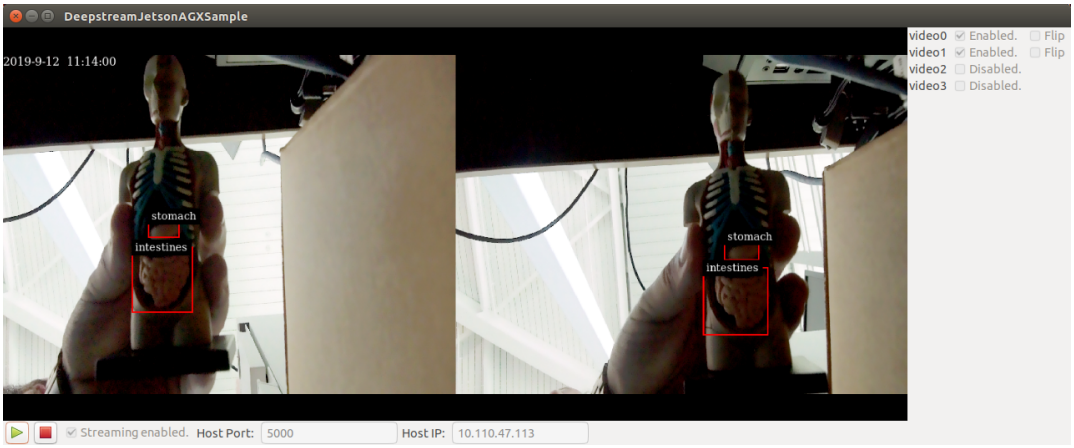
2. Run the application:

```
$ ./DeepstreamJetsonAGXSample
```

3. The application window should appear.



- All connected cameras should be visible on the right.
- Use **Enable** to select which camera streams to process. This only works before you play the stream the first time after starting the application.
- Use **Flip** to flip the video stream across the horizontal axis (if available). This only works before you play the stream the first time after starting the application.
- Check the **Enable Streaming** box on the bottom to stream the processed output as a raw h.264 UDP stream. Once this box is checked, you can define the **Host Port** and **Host IP** address of the streaming destination. See the *Streaming DeepStream Sample to Host* section below for details on how to receive the stream on the host.



- Click the green arrow to start video capture and object detection.

Important: If streaming video from multiple camera sensors at the same time, ensure that they have compatible capture mode. To check capture modes, use: `$ v4l2-ctl -d /dev/video<N>`

`--list-formats-ext` where `N` corresponds to the video source.

- Close the app to end the capture session.

Tip: By default the app launches with the FP16 network enabled. You can switch the network to FP32 or INT8 by changing the "`model-engine-file`" and "`network mode`" in the "dsjas_nvinfer_config.txt" file.

13.2 Linux x86 DeepStream Sample

Note: The Linux x86 AGX DeepStream Sample App only works with USB cameras. To avoid bandwidth issues, ensure that each USB camera is connected to its own USB controller. Depending on the motherboard in use, it is typically sufficient to connect one camera to the front and one to the back of the PC.

Note: On Linux x86, you may need to re-convert the TRT engines for your specific GPU architecture. Unlike with Tegra, it is difficult to predict the exact version of GPU and software running on your PC. In this case, the `DeepstreamLinuxHostSample` application may fail during playback with errors like the following:

```
ERROR: ../nvdsinfer/nvdsinfer_model_builder.cpp:1523 Deserialize engine␣
↪failed because file path: /opt/nvidia/clara-agx-sdk/clara-agx-deepstream-
↪sample/build/DeepstreamLinuxHostSample/model/resnet18_detector_fp16.trt␣
↪open error
0:00:05.776069825 25480 0x55de1633ed00 WARN  nvinfer gstnvinfer.cpp:616:gst_
↪nvinfer_logger:<primary-nvinference-engine> NvDsInferContext[UID 1]:␣
↪Warning from NvDsInferContextImpl::deserializeEngineAndBackend() <nvdsinfer_
↪context_impl.cpp:1691> [UID = 1]: deserialize engine from file :/opt/nvidia/
↪clara-agx-sdk/clara-agx-deepstream-sample/build/DeepstreamLinuxHostSample/
↪model/resnet18_detector_fp16.trt failed
0:00:05.776097743 25480 0x55de1633ed00 WARN  nvinfer gstnvinfer.cpp:616:gst_
↪nvinfer_logger:<primary-nvinference-engine> NvDsInferContext[UID 1]:␣
↪Warning from NvDsInferContextImpl::generateBackendContext() <nvdsinfer_
↪context_impl.cpp:1798> [UID = 1]: deserialize backend context from engine␣
↪from file :/opt/nvidia/clara-agx-sdk/clara-agx-deepstream-sample/build/
↪DeepstreamLinuxHostSample/model/resnet18_detector_fp16.trt failed, try␣
↪rebuild
0:00:05.776107361 25480 0x55de1633ed00 INFO  nvinfer gstnvinfer.cpp:619:gst_
↪nvinfer_logger:<primary-nvinference-engine> NvDsInferContext[UID 1]: Info␣
↪from NvDsInferContextImpl::buildModel() <nvdsinfer_context_impl.cpp:1716>␣
↪[UID = 1]: Trying to create engine from model files
ERROR: ../nvdsinfer/nvdsinfer_model_builder.cpp:934 failed to build network␣
↪since there is no model file matched.
ERROR: ../nvdsinfer/nvdsinfer_model_builder.cpp:872 failed to build network.
0:00:05.776325716 25480 0x55de1633ed00 ERROR nvinfer gstnvinfer.cpp:613:gst_
↪nvinfer_logger:<primary-nvinference-engine> NvDsInferContext[UID 1]: Error␣
↪in NvDsInferContextImpl::buildModel() <nvdsinfer_context_impl.cpp:1736>␣
↪[UID = 1]: build engine file failed
```

(continues on next page)

(continued from previous page)

```
0:00:05.776339221 25480 0x55de1633ed00 ERROR nvinfer gstnvinfer.cpp:613:gst_
↪nvinfer_logger:<primary-nvinference-engine> NvDsInferContext[UID 1]: Error␣
↪in NvDsInferContextImpl::generateBackendContext() <nvdsinfer_context_impl.
↪cpp:1822> [UID = 1]: build backend context failed
0:00:05.776346270 25480 0x55de1633ed00 ERROR nvinfer gstnvinfer.cpp:613:gst_
↪nvinfer_logger:<primary-nvinference-engine> NvDsInferContext[UID 1]: Error␣
↪in NvDsInferContextImpl::initialize() <nvdsinfer_context_impl.cpp:1149>␣
↪[UID = 1]: generate backend failed, check config file settings
0:00:05.776376300 25480 0x55de1633ed00 WARN  nvinfer gstnvinfer.cpp:812:gst_
↪nvinfer_start:<primary-nvinference-engine> error: Failed to create␣
↪NvDsInferContext instance
0:00:05.776381357 25480 0x55de1633ed00 WARN  nvinfer gstnvinfer.cpp:812:gst_
↪nvinfer_start:<primary-nvinference-engine> error: Config file path: dslhs_
↪nvinfer_config.txt, NvDsInfer Error: NVDSINFER_CONFIG_FAILED
Playing...
ERROR from element primary-nvinference-engine: Failed to create␣
↪NvDsInferContext instance
Error details: gstnvinfer.cpp(812): gst_nvinfer_start (): /GstPipeline:camera-
↪player/GstBin:Inference Bin/GstNvInfer:primary-nvinference-engine:
Config file path: dslhs_nvinfer_config.txt, NvDsInfer Error: NVDSINFER_CONFIG_
↪FAILED
Returned, stopping playback
```

If this is case, please follow the instructions under *Model Creation > Creating TensorRT Engines*.

1. Ensure all Linux x86 prerequisites are installed as described on the *Prerequisites* page.
2. Install the Debian file for the Clara AGX DeepStream Sample:

```
$ sudo apt install ./clara-agx-deepstream-sample-src_*_amd64.deb
```

Important: If you are re-installing or updating the Clara AGX DeepStream Sample package, you need to remove the old version first. See the *Updating the DeepStream or Endoscopy Sample Apps* section below for more information.

3. Navigate to the installation directory:

```
$ cd /opt/nvidia/clara-agx-sdk/clara-agx-deepstream-sample/build/
↪DeepstreamLinuxHostSample
```

4. Run the application:

```
$ ./DeepstreamLinuxHostSample
```

The application window should appear. See Step 5 of the *Clara AGX Developer Kit and Jetson AGX* section above for more details about the application.

13.3 Streaming DeepStream Sample to Host

Run the following command on the x86 Linux host in order to receive the video stream output by the DeepStream sample application:

```
$ gst-launch-1.0 udpsrc port=5000 ! application/x-rtp,encoding-name=H264,␣
↪payload=96 ! rtph264depay ! queue ! avdec_h264 ! autovideosink sync=false
```

13.4 Updating the DeepStream or Endoscopy Sample Apps

If you wish to update the Clara AGX DeepStream Sample package to a new version, you must remove the currently installed Debian package first. To do so, use the following command:

```
$ sudo apt remove clara-agx-deepstream-sample-src
```

Then, perform the installation as described in the steps above.

13.5 Using Docker for the DeepStream or Endoscopy Sample Apps

Docker provides lightweight reusable containers for deploying and sharing your applications.

Due to the way containers are used with Jetson, with host-side libraries mounted into the container in order to force library compatibility with the host drivers, there are two ways to build a container image containing the AGX samples:

1. Build the sample applications on the Jetson, outside of a container, then copy the executables into an image for use in a container.
2. Build the sample applications within a container.

These instructions document the process for #2, since it should be relatively straightforward to adapt the instructions to use method #1.

13.5.1 Configure the Container Runtime

The DeepStream libraries are not mounted into containers by default, and so an additional configuration file must be added to tell the container runtime to mount these components into containers.

1. Add the following to a new file, `/etc/nvidia-container-runtime/host-files-for-container.d/deepstream.csv`:

   ```
   dir, /opt/nvidia/deepstream/deepstream-5.1
   sym, /opt/nvidia/deepstream/deepstream
   ```

2. Configure Docker to default to use nvidia-docker with `"default-runtime": "nvidia"` in `/etc/docker/daemon.json`:

   ```
   {
      "runtimes": {
   ```

(continues on next page)

(continued from previous page)

```
        "nvidia": {
                "path": "nvidia-container-runtime",
                "runtimeArgs": []
        }
    },
    "default-runtime": "nvidia"
}
```

3. Restart the Docker daemon:

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

13.5.2 Build the Container Image

After installing the SDK sample code via instructions outlined in *Installation*, navigate to the `/opt/nvidia/clara-agx-sdk/clara-agx-deepstream-sample` directory and run:

```
$ sudo docker build -t clara-agx-samples -f Dockerfile.igpu .
```

13.5.3 Running the Samples in a Container

Since the samples are graphical applications and need permissions to access the X server, the DISPLAY environment must be set and root (the user running the container) must have permissions to access the X server:

```
$ export DISPLAY=:0.0
$ xhost +si:localuser:root
```

Then, using the clara-agx-samples image that was previously built, the samples can be run using the following:

```
$ sudo docker run --rm \
  --device /dev/video0:/dev/video0 \
  --env DISPLAY=$DISPLAY --volume /tmp/.X11-unix/:/tmp/.X11-unix \
  --network host --volume /tmp:/tmp \
  --workdir /opt/nvidia/clara-agx-sdk/clara-agx-deepstream-sample/build/
↪DeepstreamJetsonAGXSample \
  clara-agx-samples ./DeepstreamJetsonAGXSample
```

If you have no `/dev/video0` device this means you do not have a camera attached. Please see *Jetson Camera Setup*.

The various arguments to `docker run` do the following:

- `--device /dev/video0:/dev/video0`: Give the container access to the camera device `/dev/video0`
- `--env DISPLAY=$DISPLAY --volume /tmp/.X11-unix/:/tmp/.X11-unix`: Give the container access to the X desktop

- `--network host --volume /tmp:/tmp`: Give the container access to the Argus socket used by the Camera API
- `--workdir /opt/nvidia/clara-agx-sdk/clara-agx-deepstream-sample/build/DeepstreamJetsonAGXSample clara-agx-samples ./DeepstreamJetsonAGXSample`: Run the DeepstreamJetsonAGXSample from the clara-agx-samples image.

Chapter 14

DeepStream GPU Video Source Plugin

DeepStream SDK is based on the GStreamer framework, and provides a collection of proprietary GStreamer plugins that provide various GPU accelerated features for optimizing video analytic applications.

While DeepStream provides many plugins to meet the needs of a wide variety of use cases, it may be the case that a new plugin may be required in order to allow DeepStream integration with custom data input, output, or processing algorithms. In the case of Clara AGX, which provides a high performance edge computing platform for medical video analytics applications, one of the most common requirements will be the ability to capture video data from an external hardware device such that the data can be efficiently processed on the GPU using DeepStream or some other proprietary algorithms. For example, an ultrasound device may need to input raw data directly into the Clara AGX GPU so that it can be beamformed on the GPU before being passed through a model in DeepStream for AI inference and annotation.

Due to the proprietary nature of third party hardware devices or processing algorithms it is not possible for NVIDIA to provide a generic GStreamer plugin that will work with all third party products, and it will generally be the partner's responsibility to develop their own plugins. That being said, we also recognize that writing GStreamer plugins may be difficult for developers without any GStreamer experience, and writing these plugins such that they can work with GPU buffers and integrate with DeepStream complicates this process even further.

To help partners get started with writing GPU and DeepStream-enabled GStreamer plugins, the Clara AGX SDK includes a GPU Video Test Source GStreamer plugin, `nvvideotestsrc`. This sample plugin produces a stream of GPU buffers that are allocated, filled with a test pattern using CUDA, and then output by the plugin such that they can be consumed by downstream DeepStream components. When consumed only by other DeepStream or GPU-enabled plugins, this means that the buffers can go from end-to-end directly in GPU memory, avoiding any copies to/from system memory throughout the pipeline.

14.1 Running the GPU Video Test Source

The DeepStream GPU Video Test Source Plugin is automatically installed with the Clara AGX SDK. To test this plugin, the following command can be used.

```
$ gst-launch-1.0 nvvideotestsrc ! nv3dsink
```

This command, using the default `nvvideotestsrc` plugin parameters, will bring up a window showing an SMPTE color bars test pattern.

Fig. 14.1: nvvideotestsrc plugin SMPTE color bars pattern

Additional test patterns are available, some of which include animation. The pattern that is generated by the plugin can be configured using the `pattern` parameter. For example, to generate a Mandelbrot set pattern, which animates to zoom in and out of the image, the following command can be used.

```
$ gst-launch-1.0 nvvideotestsrc pattern=mandelbrot ! nv3dsink
```

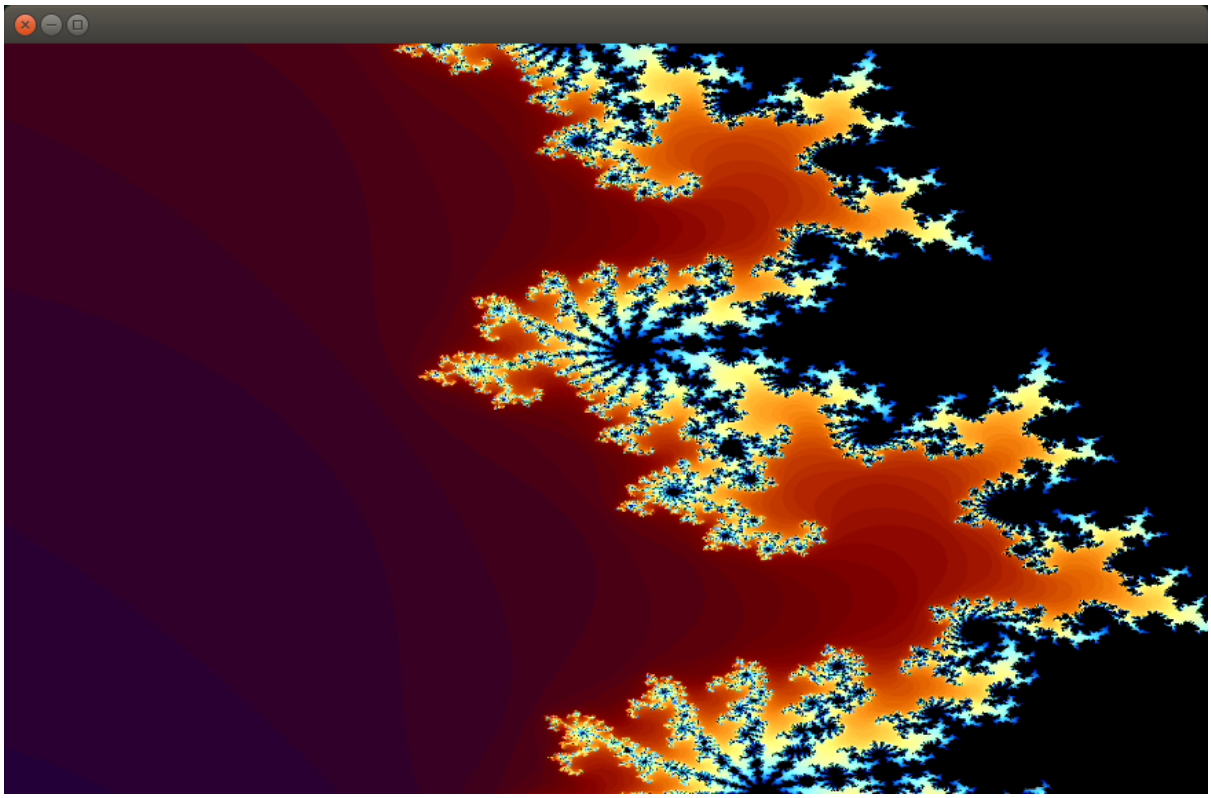


Fig. 14.2: nvvideotestsrc plugin Mandelbrot set pattern

Unless a downstream component specifically requests a specific video resolution or framerate, the plugin will default to a 1280x720 stream at 60FPS. To request a specific resolution or framerate, the GStreamer caps can also be provided to the `gst-launch-1.0` command between the `nvvideotestsrc` and downstream plugin. For example, the following command will tell the plugin to generate a 1920x1080 stream at 30FPS.

```
$ gst-launch-1.0 nvvideotestsrc ! 'video/x-raw(memory:NVMM), format=NV12,␣
↪width=1920, height=1080, framerate=30/1' ! nv3dsink
```

Note: If any of these commands fail with an error from element `nv3dsink`, make sure that the `DISPLAY` environment variable is set, e.g.: `$ export DISPLAY=:0`

14.2 Accessing the GPU Video Test Source Code

The source code for the GPU Video Test Source plugin is installed to the following path.

```
/opt/nvidia/clara-agx-sdk/clara-agx-gstnvvideotestsrc
```

The plugin is compiled and installed from this location during the Clara AGX SDK installation. If any changes to the plugin source code are made, the plugin can be recompiled and installed with the following.

```
$ cd /opt/nvidia/clara-agx-sdk/clara-agx-gstnvvideotestsrc/build
$ sudo make && sudo make install
```

14.3 Custom GPU Source with RDMA Support

A primary use for Clara AGX is as an edge device that captures video frames from an external device into GPU memory for further processing and analytics. In this case, it is recommended that the device drivers for the external device take advantage of GPUDirect RDMA such that data from the device is copied directly into GPU memory so that it doesn't need to first pass through system memory. Assuming the device drivers support RDMA, a GStreamer plugin can then use this RDMA support to accelerate DeepStream and GPU-enabled pipelines.

There are a number of resources that can be helpful for developers to add GPUDirect RDMA support to their DeepStream and GPU-enabled Clara AGX products.

1. The GPUDirect RDMA Documentation.
2. The jetson-rdma-picoevb GPUDirect RDMA sample drivers and applications.

   This sample provides a minimal hardware-based demonstration of GPUDirect RDMA including both the kernel drivers and userspace applications required to interface with an inexpensive and off-the-shelf FPGA board (PicoEVB). This is a good starting point for learning how to add RDMA support to your device drivers.
3. The `nvvideotestsrc` plugin. The source code for this plugin provides some basic placeholders and documentation that help point developers in the right direction when wanting to add RDMA support to a GStreamer source plugin. Specifically, the `gst_nv_video_test_src_fill` method – which is the GStreamer callback responsible for filling the GPU buffers – contains the following.

```
gst_buffer_map(buffer, &map, GST_MAP_READWRITE);

// The memory of a GstBuffer allocated by an NvDsBufferPool contains
// the NvBufSurface descriptor which then describes the actual GPU
// buffer allocation(s) in its surfaceList.
NvBufSurface *surf = (NvBufSurface*)map.data;

// Use CUDA to fill the GPU buffer with a test pattern.
//
// NOTE: In this source, we currently only fill the GPU buffer using CUDA.
//       This source could be modified to fill the buffer instead with␣
↪other
//       mechanisms such as mapped CPU writes or RDMA transfers:
//
//       1) To use mapped CPU writes, the GPU buffer could be mapped into
//          the CPU address space using NvBufSurfaceMap.
//
```

(continues on next page)

(continued from previous page)

```
//       2) To use RDMA transfers from another hardware device, the GPU
//          address for the buffer(s) (i.e. the `dataPtr` member of the
//          NvBufSurfaceParams) could be passed here to the device driver
//          that is responsible for performing the RDMA transfer into the
//          buffer. Details on how RDMA to NVIDIA GPUs can be performed␣
↪by
//          device drivers is provided in a demonstration application
//          available at https://github.com/NVIDIA/jetson-rdma-picoevb.
//
//       For more details on the NvBufSurface API, see nvbufsurface.h
//
gst_nv_video_test_src_cuda_prepare(self, surf->surfaceList);
self->cuda_fill_image(self);

// Set the numFilled field of the surface with the number of surfaces
// that have been filled (always 1 in this example).
// This metadata is required by other downstream DeepStream plugins.
surf->numFilled = 1;

gst_buffer_unmap(buffer, &map);
```

4. The *AJA Video Systems NTV2 SDK* and *GStreamer plugin*.

   The use of AJA Video Systems hardware with Clara AGX is documented in the *AJA Video Systems* section. This setup requires the installation of the *AJA NTV2 SDK and Drivers* and the *AJA GStreamer plugin*. These components are both open source and provide real world examples of video capture device drivers and a GStreamer plugin that use GPUDirect RDMA, respectively.

   In the case of the device driver support within the *NTV2 SDK*, all of the RDMA support is enabled at compile time using the `AJA_RDMA` flag, and so the RDMA specifics within the driver can be located using the `#ifdef AJA_RDMA` directives within the source code.

   In the case of the *GStreamer plugin*, the RDMA support was added to the plugin by this change: Add NVMM RDMA support for NVIDIA GPU buffer output.

Chapter 15

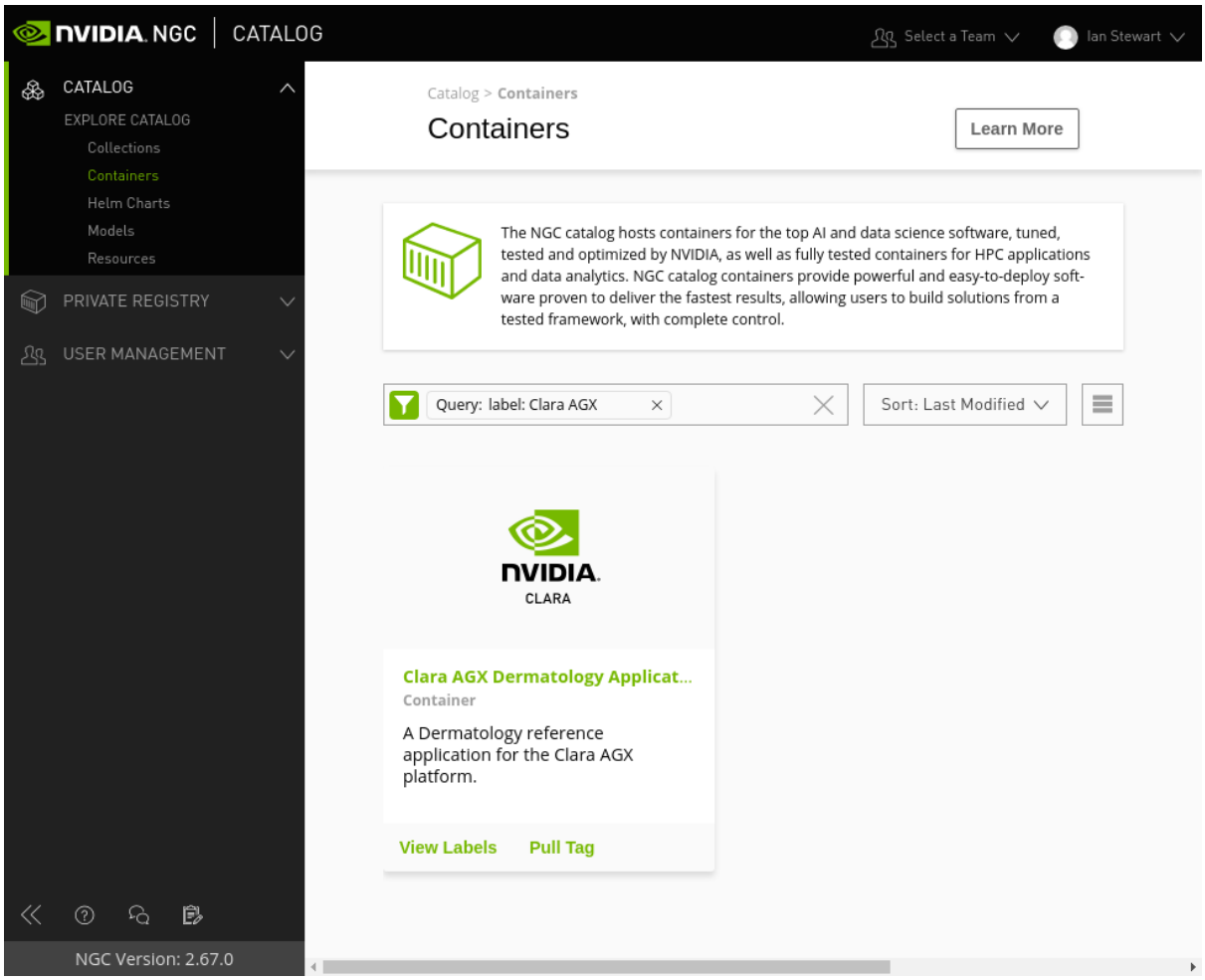
NGC Containers

In addition to the samples and packages that are installed locally as part of the Clara AGX SDK, containerized samples are also provided via the NVIDIA GPU Cloud (NGC).

In order to access the Clara AGX containers, you must first create an account and login to NGC via https://ngc.nvidia.com/signin. Once logged into NGC, the `Catalog` will provide access to all of the NVIDIA-provided containers, models, and other resources. In order to narrow this down to display just the containers provided as part of Clara AGX, the `Clara AGX` label can be used as a search query by typing `label: Clara AGX` into the search bar, or by using the following link:

Clara AGX Containers on NGC

The Clara AGX containers that are posted to NGC may be updated separately from the Clara AGX SDK releases, and so these containers may not be documented here. Please refer to the description page for the individual containers on NGC for any additional documentation related to these containers.


NVIDIA NGC
CATALOG
Select a Team
Ian Stewart
CATALOG
EXPLORE CATALOG
Collections
Containers
Helm Charts
Models
Resources
PRIVATE REGISTRY
USER MANAGEMENT
NGC Version: 2.67.0
Catalog > Containers
Containers
Learn More
The NGC catalog hosts containers for the top AI and data science software, tuned, tested and optimized by NVIDIA, as well as fully tested containers for HPC applications and data analytics. NGC catalog containers provide powerful and easy-to-deploy software proven to deliver the fastest results, allowing users to build solutions from a tested framework, with complete control.
Query: label: Clara AGX
Sort: Last Modified
NVIDIA
CLARA
Clara AGX Dermatology Applicat...
Container
A Dermatology reference application for the Clara AGX platform.
View Labels
Pull Tag