(https://www.nvidia.com/en-us/deep-learning-ai/education/)

Deployment for Intelligent Video Analytics using DeepStream

Welcome to the NVIDIA DeepStream SDK Lab!

In the previous classes, we reviewed deep network models for intelligent video analytics, video stream decoding, data preparation, optimization and deployment techniques. We saw how deployment could be a delicate and laborious task given the diversity of objects, environments and camera types, let alone live video stream handling and scalability support. To address these issues, DeepStream offers a complete framework for deploying AI-based solutions for Intelligent Video Analytics (IVA) projects.

Figure 1. DeepStream's reference application output

DeepStream allows you to focus on the core deep learning and IP components of your IVA system rather than designing an end-to-end solution from scratch.

Please note that this training assumes you are familiar with fundamentals of video processing, computer vision based deep learning tasks, including image classification and object detection, as well as the basics of the C and C++ programming languages.

This tutorial covers the following topics:

In this lab, we will guide you through a series of sample programs, each highlighting different aspects of DeepStream. By the end of this lab, you will have a strong understanding of how DeepStream applications work and how you can get started building your own.

Use Case Brainstorming

The possibilities for intelligent video analytics applications **at scale** are boundless. Take a few minutes and think about a use case. Try to think through a use case in your industry or domain of expertise. Feel free to research ideas or discuss with your neighbor or a TA.

A loose framework for one possible way to approach the task is given below. Fill in the cell with notes from your own brainstorming session.

Brainstorm Intelligent Video Analytics Application: IDEA: VIDEO TASK (DETECTION/CLASSIFICATION/SEGMENTATION/etc.): PROPOSED NEURAL NETWORK: POTENTIAL IMPACT: POTENTIAL COMPLICATIONS:

While recent advancements in deep learning and computer vision offer human-like classification and object detection capabilities, building IVA applications to utilize these techniques is a challenging task. Building video analytics applications still takes extensive effort, not to mention tuning and optimizing of these systems is tedious.

1. DeepStream Overview

1.1 Simplifying IVA Applications

DeepStream simplifies building IVA applications by separating the application into components managed and built by the user and components managed by DeepStream.

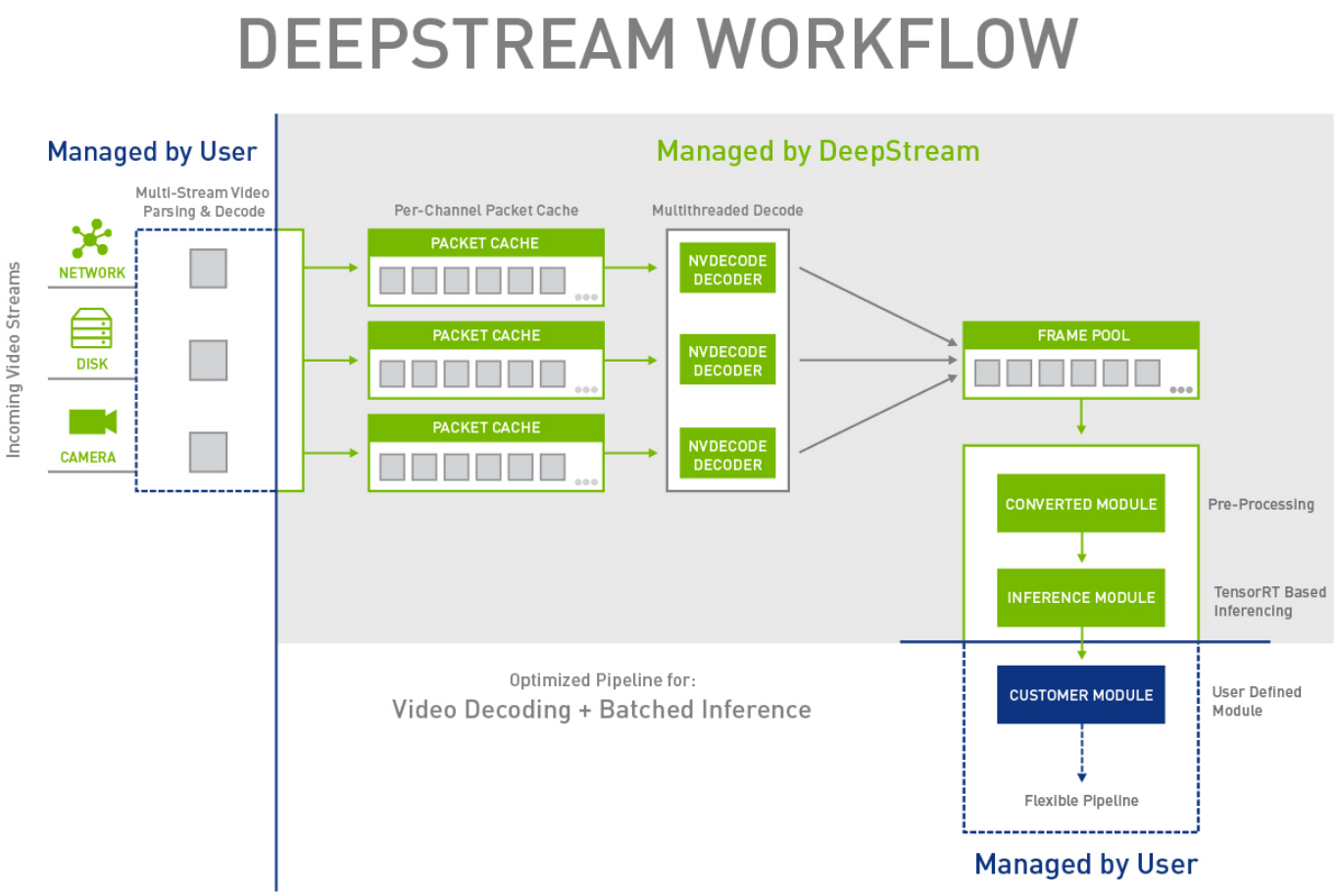


Figure 2. Video analysis tasks by DeepStream vs developer

As developers, we build components to manage important business tasks like:

- Selecting the kind and number of video streams we want to analyze
- Choosing the type of analysis, we want to do on the video
- Handling and interacting with the results of our analysis

We **don't** need to build components to manage difficult tasks like:

- Efficiently leverage the GPU for accelerated processing and inference
- Efficiently process data from multiple video streams at once
- Keeping track of metadata associated with each frame of video from multiple sources
- Optimizing our pipeline for maximum data throughput
- Optimizing our neural networks for high-speed inference

These are **mundane** tasks that the DeepStream SDK manages for us. That lets us focus on the more **important**

1.2 Performance

To give an overview of performance improvement when using DeepStream SDK, we will scrutinize the DeepStream-app reference application included within the release package. Figure 3 shows that the performance is doubled using DeepStream 3.0 when tested on T4 compared to the previous version P4, while it consumes the same amount of power and equal number of streams. The reference application includes a primary detector, three classifiers and a tracker.

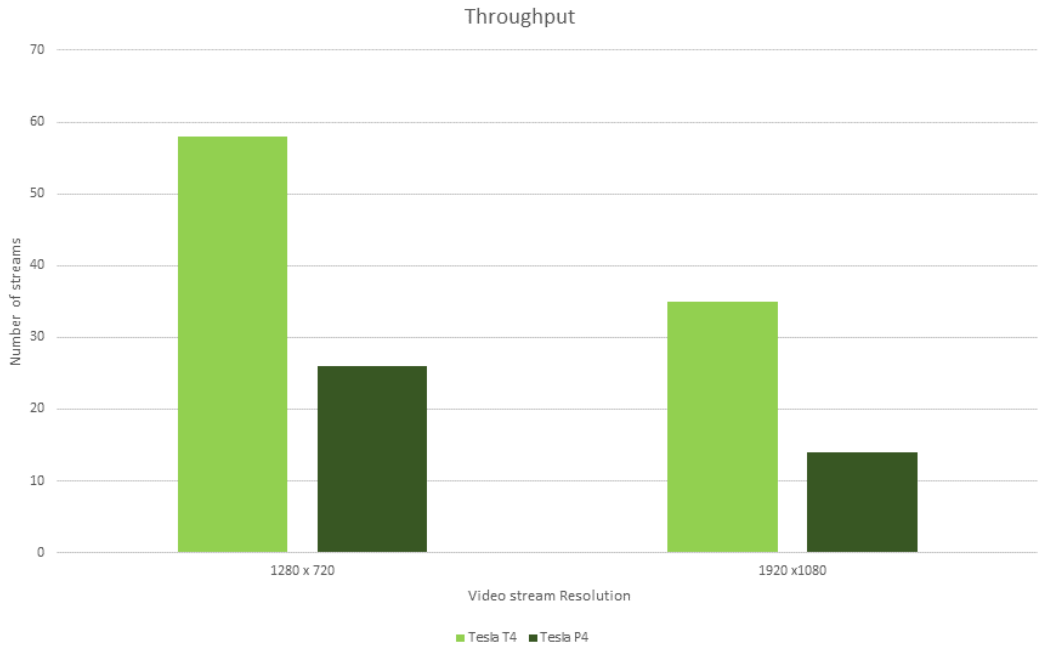
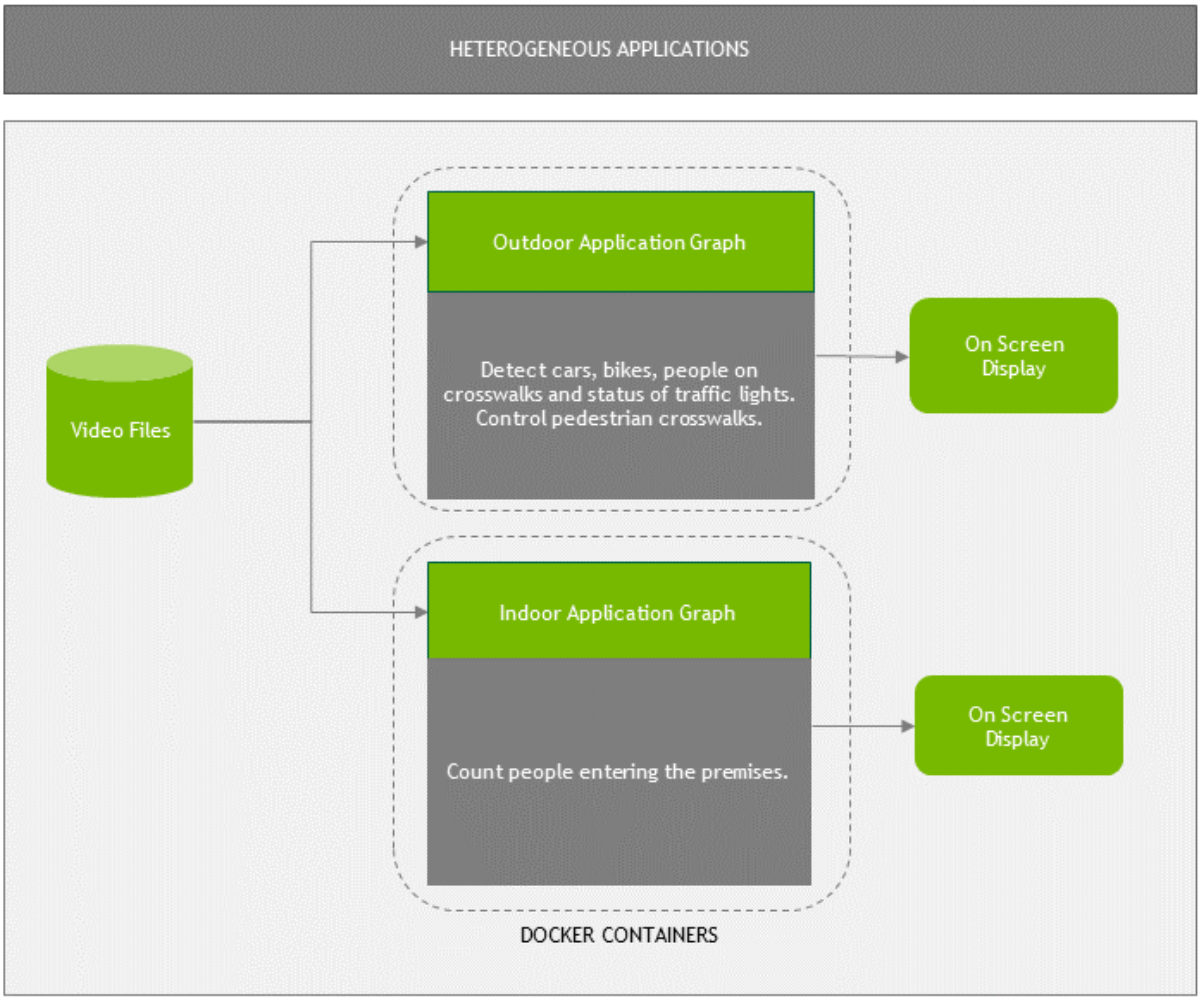


Figure 3. Performance measurement for the DeepStream reference application

1.3 Scalability

DeepStream provides scalability at different levels of the system hierarchy. For example, the DeepStream SDK 3.0 supports processing a higher number of concurrent streams, in addition to utilizing multiple GPUs upon availability. Furthermore DeepStream-in-containers provide flexibility to the deployment phase as shown below:


HETEROGENEOUS APPLICATIONS
Video Files
Outdoor Application Graph
Detect cars, bikes, people on crosswalks and status of traffic lights. Control pedestrian crosswalks.
On Screen Display
Indoor Application Graph
Count people entering the premises.
On Screen Display
DOCKER CONTAINERS

1.4 DeepStream SDK overview

The DeepStream SDK consists a set of building blocks which bridge the gap between low level APIs (such as TensorRT, Video Codec SDK) and the user application. By utilizing the DeepStream SDK, you can accelerate development of the IVA applications by focusing on building core deep learning models instead of designing end-to-end applications from scratch.

Below, you can see a schematic presentation of the DeepStream SDK in a series of potential applications.

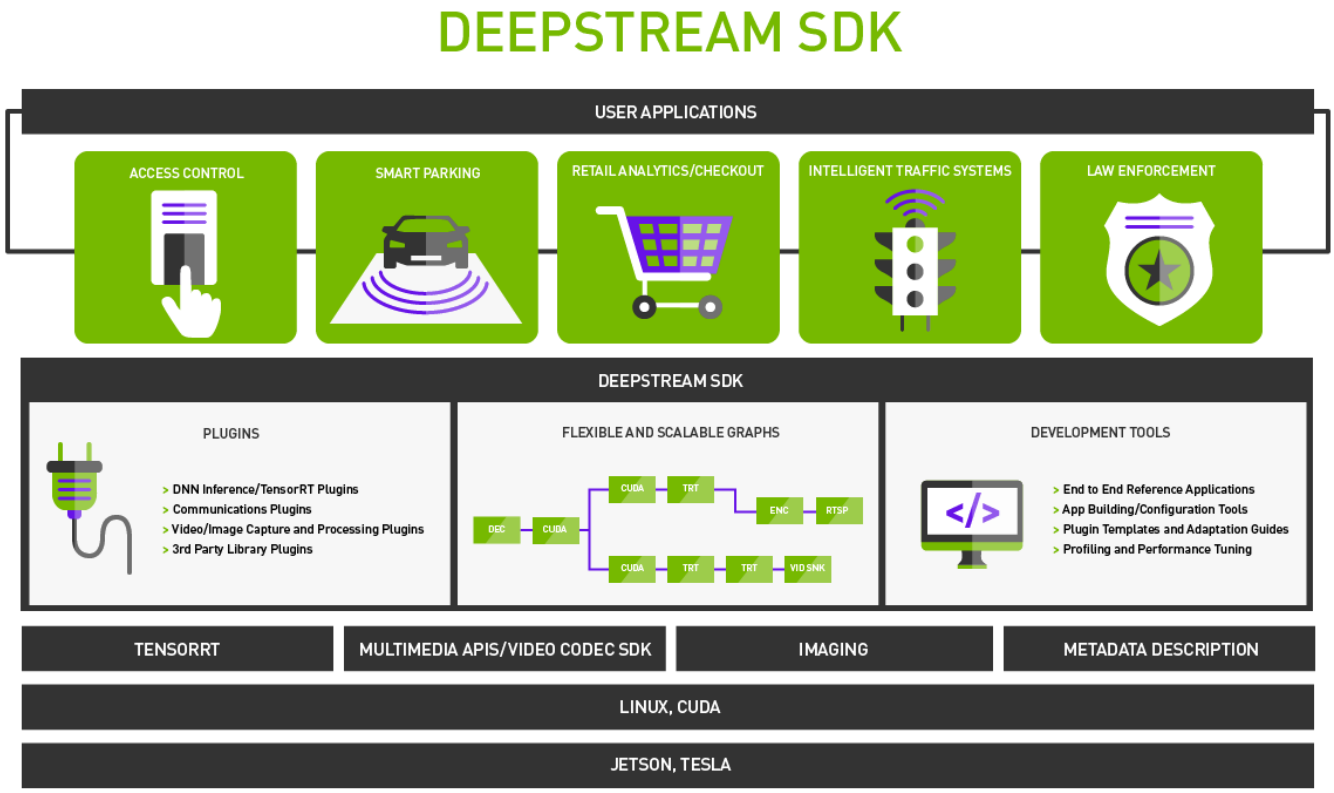


Figure 5. DeepStream application structure

In addition, the DeepStream SDK extends these capabilities by providing several other hardware accelerated building blocks. This includes support for TensorRT 5, CUDA 10, and Turing GPUs. In addition, DeepStream applications can be deployed as a part of a larger multi-GPU cluster or a microservice in containers. This allows highly flexible system architectures and opens new application capabilities.

Below, you can see a shortened list of new capabilities provided by DeepStream:

- Allowing addition and removal of video stream dynamically and during the pipeline execution, in addition to frame rate and resolution adjustments
- Extending the video processing capabilities by supporting custom layers, and user-defined parsing of detector outputs
- Providing Support for 360-degree camera using GPU-accelerated dewarping libraries
- Augmenting the meta-data with application-specific, user-defined insights
- Providing pruned and efficient inference models

- Getting detailed performance analysis with the NVIDIA Nsight system profiler tool.

The DeepStream SDK is based on the **GStreamer multimedia framework** and provides a pipeline of GPU accelerated plugins as shown in figure 6. The SDK facilitates application implementation procedure by providing plugins for video inputs, video decoding, image pre-processing, TensorRT-based inference, object tracking and display. You can utilize these capabilities to assemble flexible, multi-stream video analytics applications.

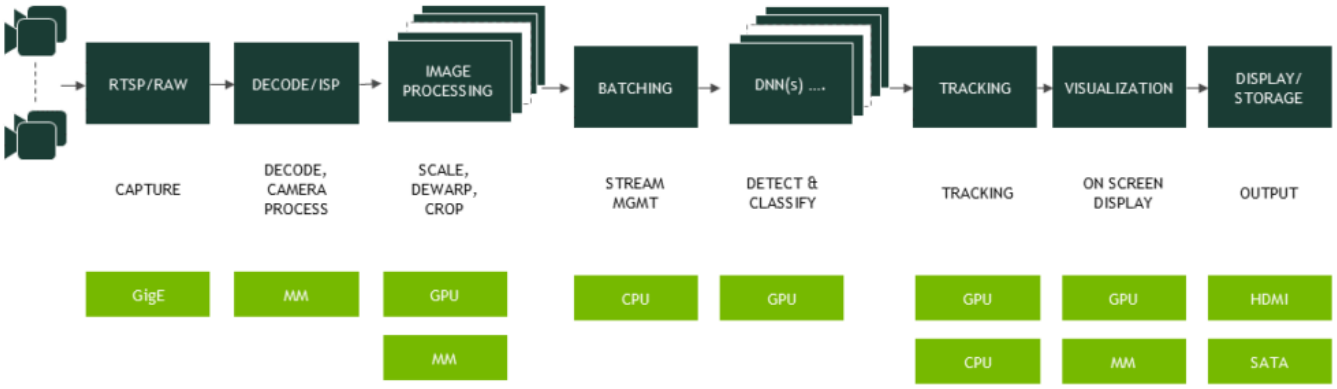


Figure 6. Sample DeepStream pipeline

2. GStreamer Foundations

The DeepStream SDK is based on the open source GStreamer multimedia framework (https://gstreamer.freedesktop.org/). There are a few key concepts in GStreamer that we need to touch on before getting started. These include Elements, Pads, Buffers, and Caps. We will be describing them at a high level, but encourage those who are interested in the details to read the GStreamer Basics (https://gstreamer.freedesktop.org/documentation/application-development/basics/index.html) documentation to learn more.

2.1 Elements

Elements are the core building block with which we make pipelines. Every process in-between the source (i.e. input of the pipeline, e.g. camera and video files) and sink elements (e.g. screen display) is passed through elements. Video decoding and encoding, neural network inference, and displaying text on top of video streams are examples of "element". DeepStream allow us to instantiate elements and weave them into pipelines.

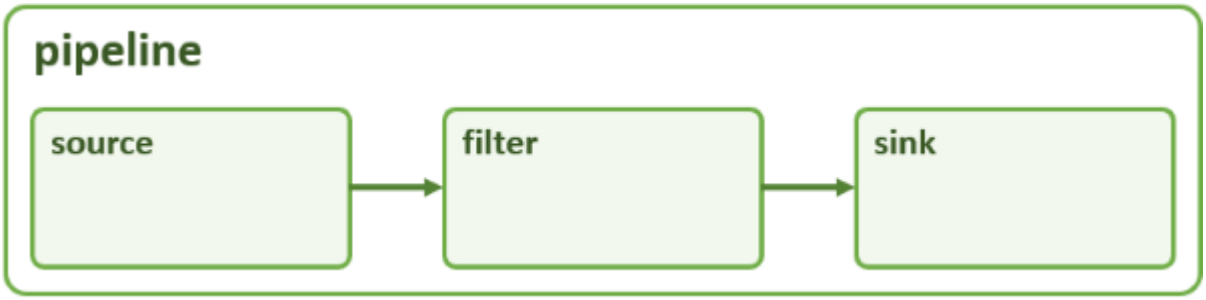


Figure 7. Example pipeline with three **elements**

2.2 Pads

Pads are the interfaces between elements. When data flows from element to another element in a pipeline, it flows from the sink pad of one element to the source pad of another. Note that each element might have zero, one or many source/sink elements.

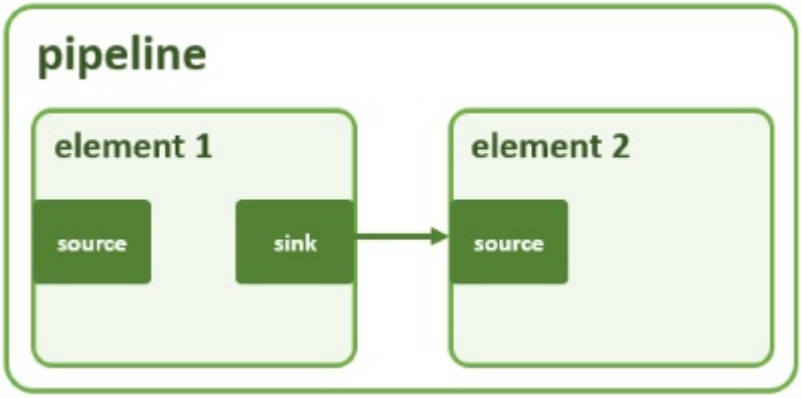


Figure 8. Elements with source/sink pads

2.3 Caps (Capabilities)

Caps (or Capabilities), are the data types that a pad is permitted to utilize or emit. Because pads can allow multiple data types, sometimes it's the data flow is ambiguous. Pads are "negotiated" in order to explicitly define the type of data that can flow through the pad. Caps streamline this process and allow elements of our pipeline with ambiguous pads to negotiate the correct data flow process. Later in this course, we will use caps to pass certain video data types (NV12, RGB) to the downstream elements in the pipeline.

2.4 Buffers

Buffers carry the data that will passed on through the pipeline. Buffers are timestamped, contain metadata such as how many elements are using it, flags, and pointers to objects in memory. When we write application code, we rely on accessing data attached to the buffer.

2.5 Plugin Based Architecture

DeepStream applications can be thought of as pipelines consisting of individual components (plugins). Each plugin represents a functional block like inference using TensorRT or multi-stream decode. Where applicable,

In [ ]:
```
!gst-inspect-1.0 h264parse
```

We see a lot of useful information, but for now let's focus on the `Descriptions`. By inspecting the `h264parse` plugin, we can see that this is meant for parsing H.264 streams. Cameras typically stream encoded data to a processor. We commonly use MPEG-4 (H.264) (https://en.wikipedia.org/wiki/H.264/MPEG-4_AVC) for compression and encoding, but other options like H.265, VC1, and MPEG-2, to name a few, are available. Compression facilitates accelerated processing by reducing the amount of data transmitted from one place to another. When we are building a pipeline, we can use this plugin if we need to parse H.264 video streams. Let's inspect a DeepStream specific plugin: `nvosd`. If you get stuck, you can go to the Answer Key.

In [ ]:
```
# Accomplish the command by yourself; remember to include the ! to start your
 command
```

This plugin is designed to draw to rectangles and text. By weaving this plugin into a pipeline, we can easily display bounding boxes and class labels on our video streams based on the results of a inference model.

3. Setup Prerequisites

DeepStream has the following requirements:

```
Ubuntu 16.04
NVIDIA driver 410+
CUDA 10
TensorRT 5.0.2+
GStreamer 1.8.3
OpenCV 3.4.1 [Download and compile OpenCV-3.4.0 with CUDA 10]
```

For the current lab session all the prerequisites are already installed. To install prerequisites on another machine, refer to section “Quick Start Guide” in the user guide file included in **DeepStream_Release/docs**.

3.1 Installation Procedure

Run the following commands. With this, the SDK installation is complete.

In [ ]:
```
!export PKG_CONFIG_PATH=/usr/lib/x86_64-linux-gnu/gstreamer-1.0
!$PKG_CONFIG_PATH
```

In [ ]:
```
%%bash

make -C DeepStream_Release/sources/apps/deepstream-test1 clean
make -C DeepStream_Release/sources/apps/deepstream-test2 clean
make -C DeepStream_Release/sources/apps/deepstream-test3 clean
make -C DeepStream_Release/sources/apps/deepstream-test4 clean
make -C DeepStream_Release/sources/libs/nvparsebbox clean
make -C DeepStream_Release/sources/gst-plugins/gst-dsexample clean
```

In [ ]:
```
# Clear the cache
!rm -rf $HOME/.cache/gstreamer-1.0/
```

3.2 Directory Layout

When you download the DeepStream package and untar it, you will find similar folders to what is described next. We will briefly mention the contents through several of the directories as they will be particularly useful for you when designing your own applications.

- In the **samples** directory, configuration files required for the reference application are provided.
- In the **samples/Models** directory, four ResNet based neural networks are provided as a part of the package.
  - **Primary Detector** is ResNet10 based 4 class detector. It detects vehicle, person, two-wheeler, and road sign. The network resolution of this model is 640x368 pixels.
  - **Secondary_CarColor**: It is 12 class color classifier.
  - **Secondary_CarMake**: It is 20 class car make classifier.
  - **Secondary_VehicleTypes**: It is 6 class car-type classifier.
  - The three secondary models are ResNet18 based car classifiers. These Networks operate on 224x224 resolution.
- In the **sources/apps** directory, four test applications and sample template plugin (to integrate custom IP) are provided. We will go through some of these applications in this lab.
- In the **includes** directory, you can find header files to develop your own application and plugin.
- **nvparsebbox** contains the sample code for a bounding box parser function. This parser function is for ResNet-10 based neural network detector. We will see more about it in lab exercise.
- binaries.tbz2 file contains NVIDIA GStreamer plugins and their supporting libraries.

4. Building a sample application

In this section, we are going to “Build a sample application”. This is the “Hello world” of DeepStream SDK. This will introduce you to the building blocks of the DeepStream SDK, how to implement a simple application using these building blocks and gives an overview of metadata handling. This application decodes the input file and then detects the objects in the frame using neural networks.

Since this is the first exercise introducing the pipeline declaration, we will spend extra time describing the code and building blocks of the pipeline.

At the end of this section, you will be familiar with how to write an application based on the Deepstream SDK.

4.1 Application Walk-Through

You can review the source code deepstream_test1_app.c (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/apps/deepstream-test1/deepstream_test1_app.c). Let's quickly walk through what's going on in our application.

Initialization and setting up a pipeline

First, we initialize the GStreamer container and create a container element called `pipeline` in **main** using `gst_pipeline_new` which is going to contain all the elements shown in the pipeline diagram we saw previously.

```
/* Standard GStreamer initialization */
  gst_init (&argc, &argv);
  loop = g_main_loop_new (NULL, FALSE);
  /* Create gstreamer elements */
  /* Create Pipeline element that will form a connection of other elements */
  pipeline = gst_pipeline_new ("dstest1-pipeline");
```

Setting the video source

Then, we create all the elements required in the pipeline using `gst_element_factory_make` . We start by specifing the input source as a video file on disk:

```
/* Source element for reading from the file */
  source = gst_element_factory_make ("filesrc", "file-source");
```

H264 parser and encoder

We need to create h264 parser and decoders. Our video source is encoded using h264 codec and `nvdec_h264` allows us to perform GPU-accelerated decoding:

```
/* Since the data format in the input file is elementary h264 stream,
   * we need a h264parser */
  h264parser = gst_element_factory_make ("h264parse", "h264-parser");
  /* Use nvdec_h264 for hardware accelerated decode on GPU */
  decoder = gst_element_factory_make ("nvdec_h264", "nvh264-decoder");
```

nvinfer, the inference plugin

The `nvinfer plugin` provides TensorRT-based inference for detection and tracking. The low-level library (libnvds_infer) operates either on float RGB or BGR planar data with dimension of Network Height and Network Width. The plugin accepts NV12/RGBA data from upstream components like the decoder, muxer, and dewarper.

The Gst-nvinfer plugin also performs preprocessing operations like format conversion, scaling, mean subtraction, and produces final float RGB/BGR planar data which is passed to the low-level library. The low-level library uses the TensorRT engine for inferencing. It outputs each classified object's class and each detected object's

bounding boxes (Bboxes) after clustering.

```
/* Use nvinfer to run inferencing on decoder's output,
   * behavior of inferencing is set through config file */
  pgie = gst_element_factory_make ("nvinfer", "primary-nvinference-engine");
```

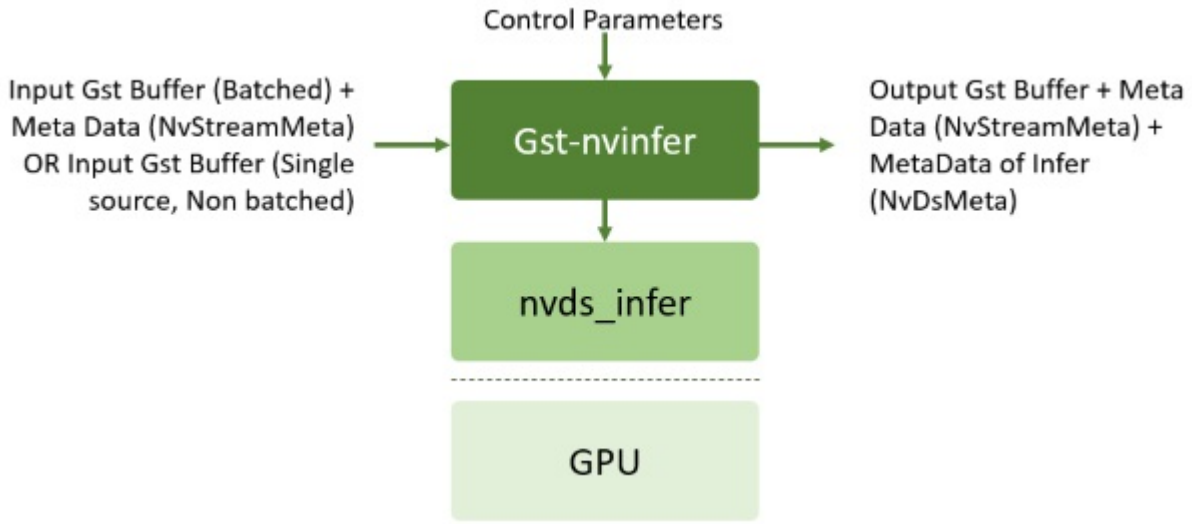


Figure 9. The nvinfer plugin

nvvidconv for video conversions

Next, we create the `nvvidconv` plugin that performs color format conversions, which is required to make data ready for the `nvosd` plugin:

```
/* Use convertor to convert from NV12 to RGBA as required by nvosd */
  nvvidconv = gst_element_factory_make ("nvvidconv", "nvvideo-converter");
```

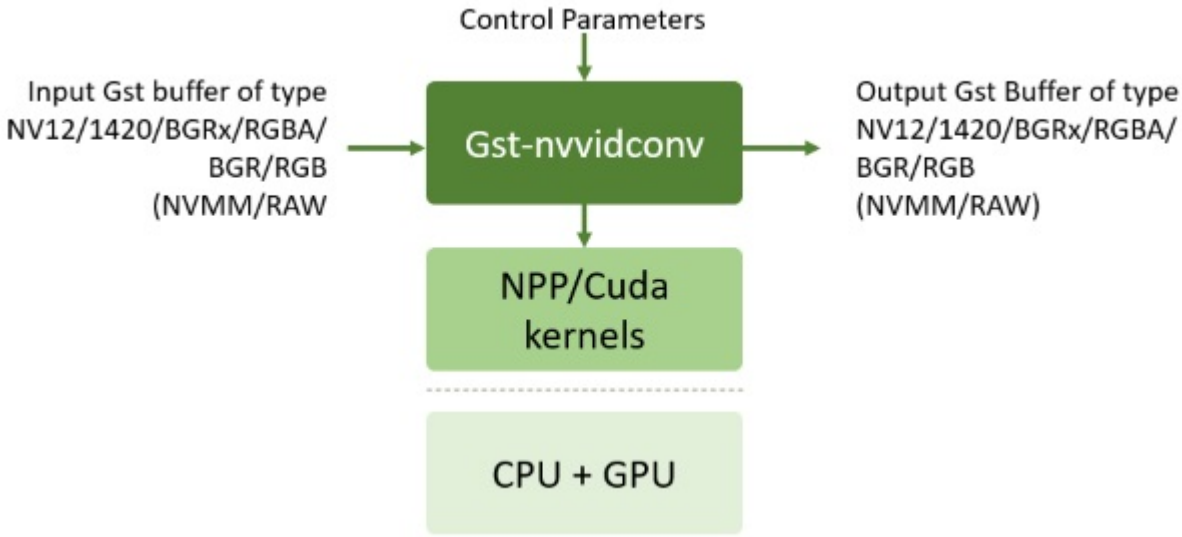


Figure 10. The nvvidconv plugin

Annotating using nvosd

The `nvosd` plugin draws bounding boxes, text, and RoI (Regions of Interest) polygons (Polygons are presented as a set of lines).

The plugin accepts an RGBA buffer with attached metadata from the upstream component. It draws bounding boxes, which may be shaded depending on the configuration (e.g. width, color, and opacity) of a given bounding box. It also draws text and RoI polygons at specified locations in the frame. Text and polygon parameters are configurable through metadata.

```
/* Create OSD to draw on the converted RGBA buffer */
  nvosd = gst_element_factory_make ("nvosd", "nv-onscreendisplay");
```

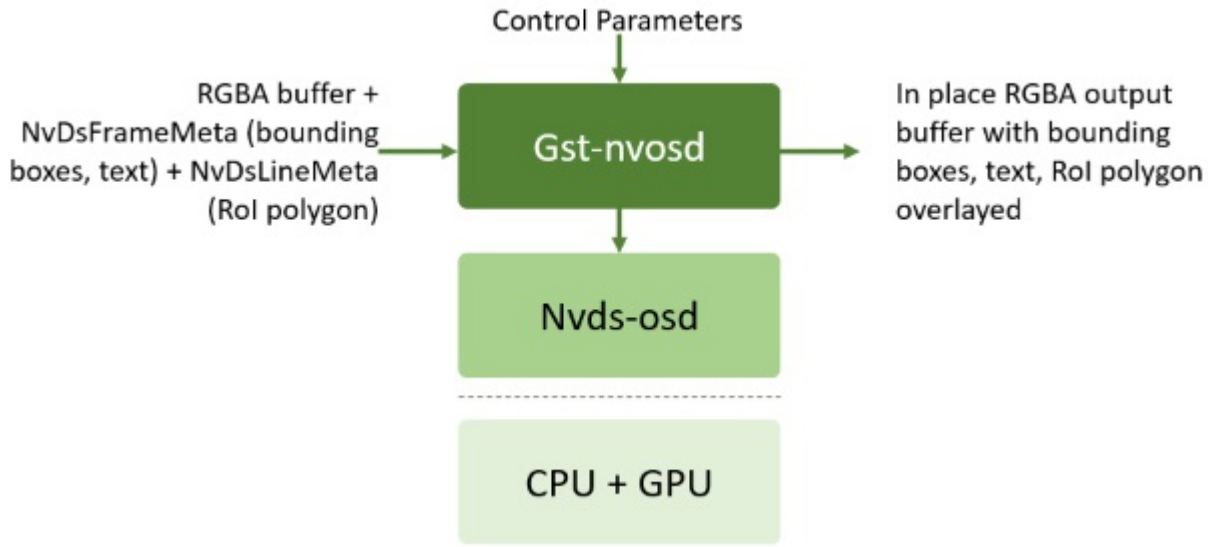


Figure 11. The nvosd plugin

Writing the stream back to the disk

Next, we are creating another nvvidconv to convert the video back to the NV12 format (after being consumed by the `nvosd` element)

```
nvvidconv1 = gst_element_factory_make("nvvidconv", "nvvideo-converter1");
  videoconvert = gst_element_factory_make("videoconvert", "converter");
```

We would like to save video files on the drive using the sink element of the pipeline. To save a video file, we need to encode the stream using the h264 codec of `x264enc` element and pass it to the `filesink` to be written on disk.

```
x264enc = gst_element_factory_make("x264enc", "h264 encoder");
  qtmux = gst_element_factory_make("qtmux", "muxer");

  /* Finally render the osd output */
  sink = gst_element_factory_make ("filesink", "filesink");
```

More on caps...

The `capsfilter` does not modify the incoming data, however it allows only certain types of data to be passed through the element. We use these filters to pass the required video data formats through the `nvosd` element.

```
/* caps filter for nvvidconv to convert NV12 to RGBA as nvosd expects input
   * in RGBA format */
  filter1 = gst_element_factory_make ("capsfilter", "filter1");
  filter2 = gst_element_factory_make ("capsfilter", "filter2");
  filter3 = gst_element_factory_make ("capsfilter", "filter3");
  filter4 = gst_element_factory_make ("capsfilter", "filter4");
```

Setting element properties...

Before creating the actual pipeline, we need to set properties of some of the elements. for example, the `filesrc` element requires the name of a video file. Below, you can review a few examples of properties we have set in the source file:

```
/* we set the input filename to the source element */
  g_object_set (G_OBJECT (source), "location", argv[1], NULL);
  g_object_set(G_OBJECT(sink), "location", "out_test1.mp4", NULL);

  /* Set all the necessary properties of the nvinfer element,
   * the necessary ones are : */

  g_object_set (G_OBJECT (pgie),
```

Accessing Metadata Results

We use probes (https://gstreamer.freedesktop.org/documentation/application-development/advanced/pipeline-manipulation.html#using-probes) to access our metadata. Probes are simply callback functions that interact with the pads of elements. They let us easily interact with data flowing through our pipeline.

```
/* Let's add probe to get informed of the meta data generated, we add probe to
   * the sink pad of the osd element, since by that time, the buffer would have
   * had got all the metadata. */
  osd_sink_pad = gst_element_get_static_pad (nvosd, "sink");
  if (!osd_sink_pad)
    g_print ("Unable to get sink pad\n");
  else {
    osd_probe_id = gst_pad_add_probe (osd_sink_pad, GST_PAD_PROBE_TYPE_BUFFER,
        osd_sink_pad_buffer_probe, NULL, NULL);
```

Inside the callback function, **osd_sink_pad_buffer_probe**, we iterate through all the metadata types which are attached to the buffer. The DeepStream components attach metadata of type `NVDS_META_FRAME_INFO` to the buffer. We access the object metadata to count the number of vehicles in the frame. In this example we are using 4 class detectors (vehicle, person, two-wheeler and road sign). The class_id of class “vehicle” is 0. So, we will count the objects having class id = 0. In the same way, we can find the number of people in the frame.

```
osd_sink_pad_buffer_probe (GstPad * pad, GstPadProbeInfo * info,
    gpointer u_data)
{
...
    while ((gst_meta = gst_buffer_iterate_meta (buf, &state))) {
        if (gst_meta_api_type_has_tag (gst_meta->info->api, _nvdsmeta_quark)) {
          nvdsmeta = (NvDsMeta *) gst_meta;
          /* We are interested only in intercepting Meta of type
           * "NVDS_META_FRAME_INFO" as they are from infer elements. */
          if (nvdsmeta->meta_type == NVDS_META_FRAME_INFO) {
            frame_meta = (NvDsFrameMeta *) nvdsmeta->meta_data;
...
            for (rect_index = 0; rect_index < num_rects; rect_index++) {
              //show class names, count etc. Refer to the source file for more info
rmation
...
            }
          }
        }
      }
...
      return GST_PAD_PROBE_OK;
}
```

We will learn more about handling metadata and probes in Exercise 2 and Exercise 6.

Adding elements to the pipeline and linking them together

We configure these elements with the required parameters so that each component does one of the operations we need. Next, we add all these elements to the pipeline (`gst_bin_add_many`). This function accepts a list of elements to be added, ending with NULL. Individual elements also can be added with `gst_bin_add` .

```
gst_bin_add_many (GST_BIN (pipeline),
      source, h264parser, decoder, pgie,
      filter1, nvvidconv, filter2, nvosd, nvvidconv1, filter3,
      videoconvert, filter4, x264enc, qtmux, sink, NULL);
```

Note: While `gst_bin_add_many` adds elements to the pipeline, but it does not link them together. By other means, the elements of the above function are not ordered. In order to link the items, we need to call the `gst_element_link_many` function.

We link all these elements in the order data will flow through the pipeline (`gst_element_link_many`).

```
gst_element_link_many (source, h264parser, decoder, pgie, filter1,
      nvvidconv, filter2, nvosd, nvvidconv1, filter3,
      videoconvert, filter4,
      x264enc, qtmux, sink, NULL);
```

Running the pipeline

With a pipeline defined and the elements linked, we install a callback function on the sink pad of the OSD component and then we put the pipeline in “PLAYING” state. When the buffer arrives on the sink pad of the OSD component, the callback function is called. When the pipeline is finished, we put the pipeline into the "NULL" state to clean up.

```
g_print ("Now playing: %s\n", argv[1]);
  gst_element_set_state (pipeline, GST_STATE_PLAYING);

  /* Wait till pipeline encounters an error or EOS */

  g_print ("Running...\n");
  g_main_loop_run (loop);
```

4.2 Using Configuration Files

Configuration files tell our application how to utilize our deep learning model. The **NvInfer** component uses this configuration file for inference. Let's look at an example configuration file to go over the common parameters.

```
gpu-id=0
net-scale-factor=0.0039215697906911373
model-file=../../../samples/models/Primary_Detector/resnet10.caffemodel
proto-file=../../../samples/models/Primary_Detector/resnet10.prototxt
labelfile-path=../../../samples/models/Primary_Detector/labels.txt
int8-calib-file=../../../samples/models/Primary_Detector/cal_trt4.bin
batch-size=1
network-mode=1
num-detected-classes=4
interval=0
gie-unique-id=1
parse-func=4
output-blob-names=conv2d_bbox;conv2d_cov/Sigmoid
```

The config file contains the list of parameters required for the neural network as well as a few parameters required for the application.

- **gpu-id**: It indicates which GPU to be used for inference in case of multiple GPUs.
- **net-scale-factor**: It is a pixel normalization factor. A parameter required for ResNet model.
- **model-file**: It is the path to the Caffe model file
- **proto-file**: It is the path to prototext file. This contains the network architecture.
- **labelfile-path**: A path to the text file containing the labels of the objects (vehicle, person, two wheeler, road sign). This file is used by the DeepStream reference application.
- **int8-calib-file**: calibration file for dynamic range adjustment with an FP32 model.
- **batch-size**: The number of frames or objects to be inferred together in a batch.
- **network-mode**: This is the data format to be used by inference. We can set it to int8 or FP32.
- **parse-func**: It is the type of detector bounding box parser function. Set it to 0 for your own trained model and it’s bounding-box parser function. We will see more about this in our 3rd lab exercise.
- **output-blob-names**: It is the name of the name of the coverage layer for the Caffe model.

4.3 Pipeline Architecture

This is the pipeline architecture of the test application.

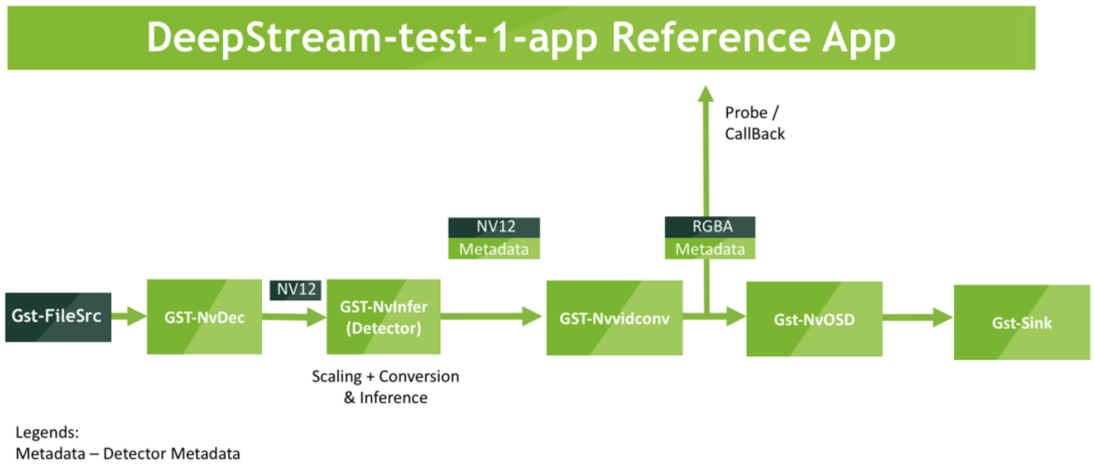


Figure 12. deepstream-test1 pipeline

In a nutshell:

To begin, the **FileSrc** component will read data from the compressed file and feed it to the decoder.

NvDec will decode the incoming data and the output frames will be given to the **NvInfer** component.

The **NvInfer** component is configured with Resnet-10 based neural network parameters (e.g. model file, prototxt, batch-size) to detect and classify the objects. After inference, it attaches the output metadata - *i.e. object type, it's bounding boxes* - to the buffer.

Then the video data on the buffer, along with its metadata, is sent to the on-screen display component to draw bounding boxes and texts describing attributes of the objects (using **Nvvidconv** along the way to convert the format).

To access the metadata in the application, we install a callback function on the sink pad of the **OSD** component. The application can then use this metadata to solve the given problem (in this case, counting the number of persons, number of vehicles, etc.).

Exercise 1

We are going to build and run the source file discussed above. There are three **#TODO** sections that you need to complete before building the application. Open deepstream_test1_app.c (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/apps/deepstream-test1/deepstream_test1_app.c) and complete all **TODOs** within it (leave the file opened and continue the following steps to get hints regarding how to fix them one by one correctly). You will see the implementation details of what we just went through above.

When you are done, run the following cell again to make the application and move to the next cell.

If you get stuck, you can reference the answer key here (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/apps/deepstream-test1/exercise_1_answer_key.txt).

In [ ]:
```
!make -C DeepStream_Release/sources/apps/deepstream-test1 clean
!make -C DeepStream_Release/sources/apps/deepstream-test1/

!cd DeepStream_Release/sources/apps/deepstream-test1 && \
./deepstream-test1-app  ../../../samples/streams/sample_720p.h264
```

Expected Output

Console Output

When you run the application, at each iteration, you should see the frame number, number of objects in the frame, vehicle count and person count.

```
.....
make: Entering directory '/dli/tasks/l-iv-04/task/DeepStream_Release/sources/apps/d
eepstream-test1'
rm -rf deepstream_test1_app.o deepstream-test1-app
make: Leaving directory '/dli/tasks/l-iv-04/task/DeepStream_Release/sources/apps/de
epstream-test1'
make: Entering directory '/dli/tasks/l-iv-04/task/DeepStream_Release/sources/apps/d
eepstream-test1'
cc -I../../includes `pkg-config --cflags gstreamer-1.0`   -c -o deepstream_test1_ap
p.o deepstream_test1_app.c
cc -o deepstream-test1-app deepstream_test1_app.o `pkg-config --libs gstreamer-1.0`
make: Leaving directory '/dli/tasks/l-iv-04/task/DeepStream_Release/sources/apps/de
epstream-test1'
Now playing: ../../../samples/streams/sample_720p.h264
>>> Generating new TRT model engine
Using INT8 data type.

 ***** Storing serialized engine file as /dli/tasks/l-iv-04/task/DeepStream_Releas
e/sources/apps/deepstream-test1/../../../samples/models/Primary_Detector/resnet10.c
affemodel_b1_int8.engine batchsize = 1 *****

Running...
End of stream
Returned, stopping playback
Deleting pipeline
```

Display Output

Check the output:

In [ ]:
```
from IPython.display import HTML
HTML("""
    <video width="640" height="480" controls>
      <source src="DeepStream_Release/sources/apps/deepstream-test1/out_test1.
mp4" type="video/mp4">
    </video>
""".format())
```

In the output video, we can see green bounding boxes and class labels around detected cars and people. The number of detections in each frame of the video corresponds to the log output printed, while the `deepstream-test1` application was running. What's controlling how many detections are made?

Changing the Configuration

The ResNet model generates class prediction scores between 0 and 1 for each of the four classes; we must choose a threshold for the predictions to decide when we should apply the label for any given class. We can change many parameters in the configuration, but this one is critical to get results from our model. Let's see what happens to the output video, if we increase our threshold for considering something to be class `vehicle`.

Let's start fresh by removing our old compiled application and output.

In [ ]:
```
!make -C DeepStream_Release/sources/apps/deepstream-test1 clean
```

Now we can change the configuration and recompile our application. Open the configuration (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/apps/deepstream-test1/dstest1_pgie_config.txt) and update the first element of `class-thresholds` to be `0.95`. This corresponds to the vehicle class. Save the file, then remake the application with the code below.

In [ ]:
```
!make -C DeepStream_Release/sources/apps/deepstream-test1/
```

With a higher threshold for predicting class `vehicle`, we would expect to see fewer detections. Let's see how this plays out when we run the application.

In [ ]:
```
!cd DeepStream_Release/sources/apps/deepstream-test1 && \
./deepstream-test1-app  ../../../samples/streams/sample_720p.h264
```

In [ ]:
```
from IPython.display import HTML
HTML("""
    <video width="640" height="480" controls>
      <source src="DeepStream_Release/sources/apps/deepstream-test1/out_test1.
mp4" type="video/mp4">
    </video>
""".format())
```

Right away, it is clear we are detecting fewer cars. In the printed output, we can see fewer detections of class car in almost every single frame. In the video output, we see fewer green bounding boxes around cars.

With just a simple tweak to a configuration parameter, DeepStream lets us interact with our deep learning model and get very different results.

5. Build A Multi-DNN Application

In this section, we will incorporate object tracking, cascaded inferencing and metadata handling by adding more neural networks to the pipeline. DeepStream makes it easy to build a relatively complex solution with only a few, simple additional steps.

5.1 Pipeline

Below, we can explore the architecture diagram of the application. Here, we have 3 additional models that identify car color, make and type respectively. Plugging additional models are like the original classifier, however there are configuration considerations that we are going to explore later in this section.

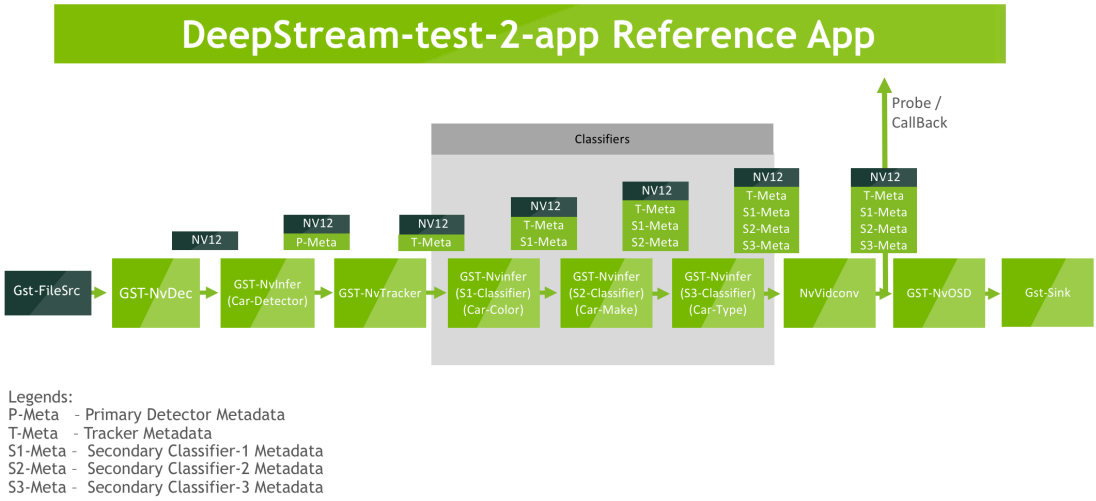


Figure 13. deepstream-test2 pipeline

You should notice that, while there is more going on than the previous exercise, the plugins are generally the same. We still need to use `NvDec`, `NvInver`, `NvOSD`, and most of the other components. We are using a new plugin though. After the first NvInfer component, we add the **NvTracker** component to track the objects and generate unique-ids for them.

NvTracker element

The `NvTracker` plugin operates on Luma data and tracks detected objects and gives each new object a unique ID.

```
/* We need to have a tracker to track the identified objects */
  nvtracker = gst_element_factory_make ("nvtracker", "tracker");
```

The plugin accepts NV12/RGBA data from the upstream component and scales (converts) the input buffer to a Luma buffer with a specific tracker width and height. (Tracker width and height must be specified in the configuration file's [tracker] section.)

The low-level library uses a CPU based implementation of the **Kanade Lucas Tomasi (KLT)** tracker algorithm. The plugin also supports the Intersection of Union (IoU) tracker algorithm, which uses the intersection of the detector's bounding boxes across frames to determine the object's unique ID.

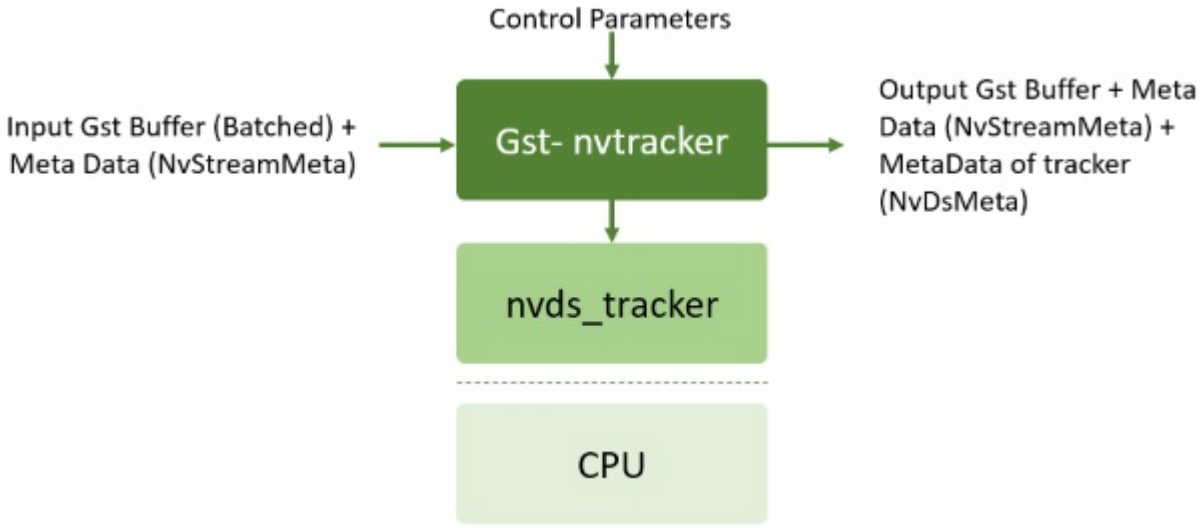


Figure 14. The nvtracker plugin

The tracker component updates the object's metadata with a tracker-id. After this component, we add three cascaded secondary neural network classifiers. These classifiers work on the objects detected as "vehicles or cars". The first classifier classifies the car color. The second classifier classifies the car make and the third classifier, classifies car type (e.g. coupe, sedan, etc.). Each classifier, after inference on a car object, will append the metadata to their results. Then, the application, using a callback function, can access the metadata to understand and analyze the attributes of the objects.

5.2 Application code walk-through

Creation of Elements & Linking

In the first exercise, we composed a pipeline by creating the elements and adding them to the pipeline. In this example, we create a tracker element and 3 additional NvInfer elements which serve as secondary classifiers. We configure them with appropriate network parameters as we did in the first example.

5.3 New Configuration Parameters

Because these secondary classifiers are only intended to execute on objects that we believe are vehicles, we will need to add new configuration parameters to generate this behavior.

Two new parameters, **operate-on-gie-id** and **operate-on-class-ids** will let us control this behavior.

The first, **operate-on-gie-id**, lets us configure a classifier to only execute on objects from a different classifier. In this case, we will configure the secondary classifier to only execute on objects detected by the primary classifier. The second, **operate-on-class-ids**, lets us configure a classifier to only execute on objects **of a specific class**. By combining these two, our secondary classifiers will be configured to only evaluate the make, model, and color of objects classified as cars by our primary model.

Open up the configuration file (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/apps/deepstream-test2/dstest2_sgie1_config.txt) for the first secondary classifier and create **operate-on-gie-id** and **operate-on-class-ids** parameters. Set **operate-on-gie-id** equal to 1, telling the classifier to execute on objects from the first detector network. Set **operate-on-class-ids** equal to 0, telling the classifier to execute only when the `class_id` is 0, corresponding to the `vehicle` class.

We have gone ahead and added the new parameters to the configurations for the other two secondary classifiers for you.

If you get stuck, you can reference the answer key here (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/apps/deepstream-test2/exercise_2_configuration_answer_key.txt).

Exercise 2

If you have added the two new configuration parameters and saved the file, continue to the next cell block to compile and run the application. But before running the application, there are TODO items to be completed in the source code. Open deepstream_test2_app.c (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/apps/deepstream-test2/deepstream_test2_app.c) and complete all **TODOs** within it (leave the file opened and continue the following steps to get hints regarding how to fix them one by one correctly).

If you are able to `make` your application successfully, but experience errors when trying to run the application, make sure your configuration file is correct and that you correctly completed all **TODOs** in deepstream_test2_app.c (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/apps/deepstream-test2/deepstream_test2_app.c).

If you are completely stuck, you can reference the answer key (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/apps/deepstream-test2/exercise_2_test2_app_answer_key.txt).

When you are done, run the cells below again to make the application and move to the next section.

In [ ]:
```
# clean up the build directory
!make -C DeepStream_Release/sources/apps/deepstream-test2 clean
```

In [ ]:
```
# make and run the deepstream test2 app
!make -C DeepStream_Release/sources/apps/deepstream-test2

! cd DeepStream_Release/sources/apps/deepstream-test2 && \
./deepstream-test2-app  ../../../samples/streams/sample_720p.h264
```

Console Output

On the screen you should see the frame number, number of objects in the frame, vehicle count and person count.

```
.....
Now playing: ../../../samples/streams/sample_720p.h264
>>> Generating new GIE model cache
Using INT8 data type.
...
Running...
[ INFO:0] Initialize OpenCL runtime...
Frame Number = 0 Number of objects = 4 Vehicle Count = 3 Person Count = 1
Frame Number = 1 Number of objects = 4 Vehicle Count = 3 Person Count = 1
Frame Number = 2 Number of objects = 4 Vehicle Count = 3 Person Count = 1
...
Frame Number = 594 Number of objects = 6 Vehicle Count = 6 Person Count = 0
Frame Number = 595 Number of objects = 6 Vehicle Count = 6 Person Count = 0
Frame Number = 596 Number of objects = 6 Vehicle Count = 6 Person Count = 0
End of stream
Returned, stopping playback
Deleting pipeline
```

Display Output

Let's look at our video output to see the results.

In [ ]:
```
from IPython.display import HTML
HTML("""
    <video width="640" height="480" controls>
      <source src="DeepStream_Release/sources/apps/deepstream-test2/out_test2_
app.mp4" type="video/mp4">
    </video>
""".format())
```

By using the secondary neural networks, we can classify cars into more specific categories than before. Notice that the `Person` category doesn't have the additional metadata, while the `Vehicle` category does. Let's dive into the details of handling metadata to see how we made this happen.

5.4 Accessing Metadata

Similar to the previous section, in the callback function, we access the metadata attached to buffer. Here, in this callback, we access the metadata added by the first classifier to get the color of the car. We can find the car make and the car type which are the outputs from the second and the third classifier. You can try it on your own and see the result on the console.

```
/* Lets add probe to get informed of the meta data generated, we add probe to
   * the sink pad of the osd element, since by that time, the buffer would have
   * had got all the metadata. */
  osd_sink_pad = gst_element_get_static_pad (nvosd, "sink");
  if (!osd_sink_pad)
    g_print ("Unable to get sink pad\n");
  else
    osd_probe_id = gst_pad_add_probe (osd_sink_pad, GST_PAD_PROBE_TYPE_BUFFER,
        osd_sink_pad_buffer_probe, NULL, NULL);
```

And the callback function is defined as:

```
static GstPadProbeReturn
osd_sink_pad_buffer_probe (GstPad * pad, GstPadProbeInfo * info,
    gpointer u_data)
{
...
  while ((gst_meta = gst_buffer_iterate_meta (buf, &state))) {
    if (gst_meta_api_type_has_tag (gst_meta->info->api, _nvdsmeta_quark)) {

      nvdsmeta = (NvDsMeta *) gst_meta;
        frame_meta->num_strings = 0;

        num_rects = frame_meta->num_rects;
        /* This means we have num_rects in frame_meta->obj_params,
         * now lets iterate through them */

        for (rect_index = 0; rect_index < num_rects; rect_index++) {
 ...
        }

      }
    }
  }
```

Our metadata handling is governed by two files: **gstnvdsmeta.h** and **nvosd.h**.

gstnvdsmeta.h (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/includes/gstnvdsmeta.h) defines the NVIDIA DeepStream Metadata structures used to describe metadata objects and nvosd.h (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/includes/nvosd.h) defines the NvOSD library to be used to draw rectangles and text over the video frame for given parameters.

Open gstnvdsmeta.h (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/includes/gstnvdsmeta.h) and review it. Pay attention to the following code snippet in the docstring:

```
gpointer state = NULL;
GstMeta *gst_meta;
NvDsMeta *nvdsmeta;
static GQuark _nvdsmeta_quark = 0;

if (!_nvdsmeta_quark)
    _nvdsmeta_quark = g_quark_from_static_string (NVDS_META_STRING);

  while ((gst_meta = gst_buffer_iterate_meta (buf, &state))) {
    if (gst_meta_api_type_has_tag (gst_meta->info->api, _nvdsmeta_quark)) {

      nvdsmeta = (NvDsMeta *) gst_meta;
      // Do something with this nvdsmeta
```

Because DeepStream metadata is attached to the buffer, we use a while loop to iterate through all the attached metadata. Within the loop, we write our business logic that determines what we display on the video output. Open deepstream_test2_app (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/apps/deepstream-test2/deepstream_test2_app.c) and scroll to line 92 to see the following code snippet:

```
/* We are interested only in intercepting Meta of type
       * "NVDS_META_FRAME_INFO" as they are from our infer elements. */
      if (nvdsmeta->meta_type == NVDS_META_FRAME_INFO) {
        frame_meta = (NvDsFrameMeta *) nvdsmeta->meta_data;
```

By intercepting the metadata from the **NvInfer** element and storing it in `frame_meta`, we can access the results of our deep learning models.

Notice that you can also control the on-screen display parameters defined in **nvosd.h**. These `txt_params` also come from the `frame_meta`.

Exercise 3

Change the font size to 24 (line 203 in **deepstream_test2_app**) and run the cell block below to create a new output video.

In [ ]:
```
# re-make and run the deepstream test2 app
!make -C DeepStream_Release/sources/apps/deepstream-test2

! cd DeepStream_Release/sources/apps/deepstream-test2 && \
./deepstream-test2-app  ../../../samples/streams/sample_720p.h264
```

Now, let's see how the output video changed.

In [ ]:
```
from IPython.display import HTML
HTML("""
    <video width="640" height="480" controls>
      <source src="DeepStream_Release/sources/apps/deepstream-test2/out_test2_
app.mp4" type="video/mp4">
    </video>
""".format())
```

In this section, we learned how to set up a pipeline with multiple, cascaded deep learning classifiers. Configuration file parameters are an important way for us to control the behavior of our models in a DeepStream application.

We also dove deeper into our callback function for handling metadata. We access all the metadata attached to the buffer by iterating through it one at a time. To access metadata and results from our deep neural networks, we perform our business logic on the metadata of type `NVDS_META_FRAME_INFO` .

At this point, you should have a sense of where you could include business logic components to utilize the results of a deep learning classifier.

6. Custom Parsers and Networks

Up until now, we have only used the sample neural networks and parsers.

In this section, we will see how to use the **NvInfer** component to accommodate any **TensorRT** compatible neural network for `detection` . With this, users can focus on developing their own neural networks and leverage DeepStream building blocks to quickly implement a solution around it for novel use cases.

6.1 Pipeline

Here, the pipeline architecture is the same as in exercise 1.

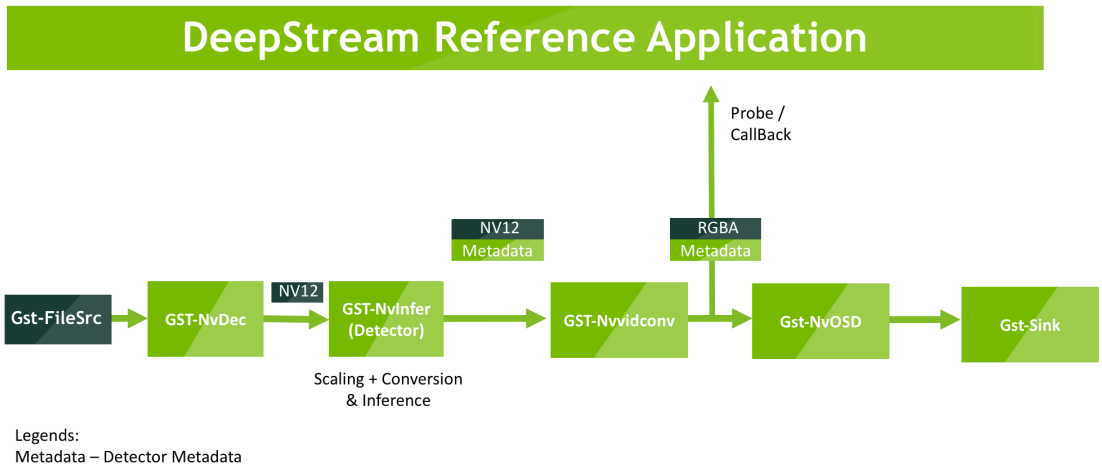


Figure 15. The custom parser pipeline

6.2 Using Custom Parsers

In the DeepStream package, we provide an example of a custom parser (inside the `nvparsebbox` directory). This sample parser function is for a ResNet10 based detector. For this exercise, we will use this plugin as a bounding box parser function of the custom model created by a developer. For now, we will use the primary detector in the package as our custom model, but you could always supply a new model via the configuration files as described previously.

To use this parser, we first need to generate a library containing this parser function and set the configuration parameters mentioned in the previous cell. It's also important to make sure you are setting the correct network parameters when you are using custom parsers, but we will get to that in a bit.

Exercise 4

Open nvdsparsebbox.cpp (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/libs/nvdsparsebbox2/nvdsparsebbox.cpp) and review the example model. There's a couple of key items to point out here. We will need to keep track of our function's name, `parse_bbox_custom_resnet` , so we can use it in the configuration file. We also need to fill in the if statement on line 20:

```
if(framenum == 0)
  {
    printf(<<<<FIXME>>>>);
  }
```

Fill in this `printf` with what you want to print when the custom parser is called when operating on the first frame. This serves as an indicator that our custom parser is in fact being used by DeepStream. Remember to wrap your text in double quotations marks, like this: `"example text"` . When you are ready, proceed to the next cell block to build the library.

If you run into errors, make sure your `printf` statement is correct and re-run the cell block below.

In [ ]:
```
import sys,os,os.path
os.environ['LD_LIBRARY_PATH']='/usr/local/lib/x86_64-linux-gnu:/usr/local/lib/
i386-linux-gnu:/usr/lib/x86_64-linux-gnu:/usr/lib/i386-linux-gnu:/tensorrt/inc
lude'
```

In [ ]:
```
!echo $LD_LIBRARY_PATH

# clean up before building
!make -C newDS/DeepStream_Release/sources/libs/nvdsparsebbox2 clean

# build the library
!make -C newDS/DeepStream_Release/sources/libs/nvdsparsebbox2
```

With the library built, there's one more step. We need tell DeepStream to use this function and library for parsing. To start, open dstest1_pgie_config.txt (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/apps/deepstream-test3/dstest1_pgie_config.txt) and review the configuration file. It should look familiar, as it's the same as the one from section 4. To use a custom parser, we need to add three additional parameters for the NvInfer component:

- **parse-bbox-func-name**
  - The name of the bounding box parser function of the detector
- **custom-lib-path**
  - The library that contains the parser function
- **parse-func**
  - Setting this equal to 0 indicates that we are using a detection algorithm. Example parameters are listed below.

```
parse-func=0
parse-bbox-func-name=parse_bbox_custom_resnet
custom-lib-path=/dli/tasks/l-iv-04/task/DeepStream_Release/sources/libs/nvdsparsebb
ox2/libnvdsparsebbox.so
```

Make sure that the parameter `parse-bbox-func-name` is set to be equal to the name of the function you defined in nvdsparsebbox.cpp (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/libs/nvdsparsebbox2/nvdsparsebbox.cpp).

Build and Run the Application

With the library built and configuration set, we are ready to build and run the application.

In [ ]:
```
!rm -rf $HOME/.cache/gstreamer-1.0/ && \
cd DeepStream_Release/sources/apps/deepstream-test3 && \
make && \
./deepstream-test3-app  ../../../samples/streams/sample_720p.h264
```

Notice that, after the first frame, our custom parser printed out a message. DeepStream is using this parser on every frame. We can look at the output video and see the results of our parser.

In [ ]:
```
from IPython.display import HTML
HTML("""
    <video width="640" height="480" controls>
      <source src="DeepStream_Release/sources/apps/deepstream-test3/out_test3_
app.mp4" type="video/mp4">
    </video>
""".format())
```

Console Output

```
...
#your_flag: Text inside the printf call
Frame Number = 0 Number of objects = 5 Vehicle Count = 3 Person Count = 2
....
Frame Number = 1428 Number of objects = 4 Vehicle Count = 4 Person Count = 0
Frame Number = 1429 Number of objects = 5 Vehicle Count = 5 Person Count = 0
Frame Number = 1430 Number of objects = 5 Vehicle Count = 5 Person Count = 0
Frame Number = 1431 Number of objects = 3 Vehicle Count = 3 Person Count = 0
Frame Number = 1432 Number of objects = 4 Vehicle Count = 4 Person Count = 0
Frame Number = 1433 Number of objects = 5 Vehicle Count = 5 Person Count = 0
Frame Number = 1434 Number of objects = 6 Vehicle Count = 6 Person Count = 0
Frame Number = 1435 Number of objects = 5 Vehicle Count = 5 Person Count = 0
Frame Number = 1436 Number of objects = 5 Vehicle Count = 5 Person Count = 0
Frame Number = 1437 Number of objects = 6 Vehicle Count = 6 Person Count = 0
Frame Number = 1438 Number of objects = 5 Vehicle Count = 5 Person Count = 0
End of stream
Returned, stopping playback
#your_flag: parse_bbox_custom_resnet is called
Deleting pipeline
```

In this exercise, we learned how to incorporate a custom parser into a DeepStream pipeline by leveraging the **NvInfer** plugin. **NvInfer** makes it easy to utilize custom neural networks and neural network parsers. When we do this, we need to make sure that we update our configuration file parameters with the relevant parameters.

7. Custom Plugins and OpenCV based Detection

In this exercise, we will see how to incorporate a custom plugin into DeepStream pipeline. We will use the custom plugin **DsExample** to use OpenCV (a commonly used computer vision library) to detect a moving car in a video stream.

OpenCV 3 is a pre-requisite for Deepstream so it's already at your disposal. The DsExample Makefile already includes necessary linking instructions for the OpenCV modules typically used for real-time processing. If you base your future custom plugins on this reference plugin, and you need other modules, you should include them in the Makefile. Students with experience in C and C++ will notice that we are building our library in C++ as opposed to C, which is necessary to use OpenCV.

7.1 Building Pipelines at the Command Line

In this section, we will also be composing our pipeline at the command line, which makes it easy for us to prototype.

In order to run the pipeline on the command line we are using a native Gstreamer program called `gst-launch-1.0` which builds and runs the pipeline and is called using the following parameter list:

```
gst-launch-1.0 [OPTIONS] PIPELINE-DESCRIPTION
```

Here, `description` is the list of pipeline elements and their corresponding properties. These elements are separated by exclamation marks.

When we compose pipelines on the command line instead of building them into applications, we sacrifice flexibility and advanced usage for ease of prototyping. For production usage, we would compose the pipeline in our application, as we have done in the previous three sections. For prototyping and learning, building the pipeline at the command line and visualizing can help us get a sense of what's happening. We will see this at the end of the exercise.

7.2 Pipeline Architecture

Figure 16. The custom plugin architecture

In this exercise, our pipeline will look very similar to the one from Exercise 1. The key difference is that here, we are building a pipeline that uses **DsExample** instead of **NvInfer** to perform the object detection. As with deepstream-test1 the metadata produced now by our inference/detection plugin is rendered by `nvosd` and later, displayed by the video `sink` .

7.3 Custom Plugin Library Structure

DsExample supports the DeepStream metadata for inter-plugin operability, which inputs uncompressed frames and outputs ROI metadata. The implementation of DsExample a could be useful as a proof of concept and experiments, as a template to start development and in order to shield your code form GStreamer specifics.

The DsExample DeepStream filter consists of 2 parts:

1. The custom library dsexample_lib_cpp (../../../../tree/tasks/l-iv-04/task/DeepStream_Release/sources/gst-plugins/gst-dsexample/dsexample_lib_cpp) that implements our custom detection algorithm.
2. The code that implements all the necessary mechanics for the GStreamer filter (gstdsexample.cpp (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/gst-plugins/gst-dsexample/gstdsexample.cpp) and the associated header file)

7.4 Motion Detection with OpenCV

Our approach to motion detection is straightforward: We analyze the differences in consecutive frames and, if the difference is above a threshold, consider that contour (https://docs.opencv.org/3.3.1/d4/d73/tutorial_py_contours_begin.html) to be moving. Once we have identified moving contours, we identify the bounding boxes for the contour and return that metadata attached to a `DsExampleOutput` object.

Please review the library code by opening dsexample_lib.cpp (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/gst-plugins/gst-dsexample/dsexample_lib_cpp/dsexample_lib.cpp) and the header file (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/gst-plugins/gst-dsexample/dsexample_lib_cpp/dsexample_lib.h). Pay particular attention to the function `DsExampleProcess`, which has a return type of `DsExampleOutput` in the `dsexample_lib.cpp`. This is where we are implementing the details of OpenCV based detection algorithm and calculating bounding boxes.

7.5 GStreamer Plugin Mechanics

Going through the low-level details of warping the library into a GStreamer plugin is beyond the scope of this lab. Some of the core functionality we implement include:

- Initializing and setting properties of our element
- Starting and stopping the output thread
- Wrapping around our custom library to execute when this element receives an input buffer from an upstream element
- Attaching resulting metadata to the frame

We encourage you to look through gstdsexample.cpp (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/gst-plugins/gst-dsexample/gstdsexample.cpp) and gstdsexample.h (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/gst-plugins/gst-dsexample/gstdsexample.h) and take note of the core components. You can download these files for reference, too.

With this background, we are ready to build the plugin and run our pipeline.

7.6 Compile and Run

In [ ]:
```
!rm -f DeepStream_Release/sources/gst-plugins/gst-dsexample/libgstnvdsexample.
so && \
make -C DeepStream_Release/sources/gst-plugins/gst-dsexample -f Makefile_cpp
```

We also need to copy the built library to the gstreamer directory and clear our cache. Once we have done that, we are ready to run the pipeline.

In [ ]:
```
!cp DeepStream_Release/sources/gst-plugins/gst-dsexample/libgstnvdsexample.so
 /usr/lib/x86_64-linux-gnu/gstreamer-1.0/
```

In [ ]:
```
!rm $HOME/.cache/gstreamer-1.0/registry.x86_64.bin
```

In [ ]:
```
# Run the pipeline

!GST_DEBUG_DUMP_DOT_DIR=./ gst-launch-1.0 filesrc location=DeepStream_Release/
samples/streams/car1.mp4 ! decodebin ! \
  nvvidconv ! "video/x-raw(memory:NVMM), format=(string)RGBA" ! \
  dsexample processing-width=160 processing-height=120 ! nvosd ! queue ! \
  nvvidconv ! "video/x-raw, format=(string)RGBA" ! videoconvert ! x264enc ! qt
mux ! filesink location=out_dsmotion.mp4
```

7.7 Visualizing the Pipeline

When the pipeline is finished, we can visualize it to get a better understanding of the data flow. By setting `GST_DEBUG_DUMP_DOT_DIR` before running the pipeline, DeepStream will create pipeline diagrams (graphs) to visualize the pipeline in different states.

Execute the following cell block, and then replace the `REPLACE_ME` in the cell beneath it with the output from this cell. Then run that one, too. The `dot` command comes from graphviz (https://www.graphviz.org/), and it lets us convert `.dot` files into standard image formats.

In [ ]:
```
!ls ./ | grep PLAYING_PAUSED
```

In [ ]:
```
!dot -Tpng ./0.00.17.691726599-gst-launch.PLAYING_PAUSED.dot > ./ex4_pipeline.
png
```

Now, open this link (ex4_pipeline.png) and view the pipeline diagram in the browser. There is a lot to see here. Each block corresponds to an element in our pipeline, and the arrows indicate the flow of data (from left to right). The first block on the left is the **filesrc block**. This is the beginning of our pipeline. Data flows from here into the large **decodebin** block, which contains multiple sub-blocks corresponding to different elements in our pipeline.

You may have noticed that there is a **Caps filter** (in fact, there are several throughout the pipeline). We have seen these in pipelines before (in our DeepStream application code), and briefly discussed Caps in the beginning of the lab.

When an element has more than one output pad, it might not be clear which one should be used. An example of this would be when an element outputs both audio and video, and the next element could operate on either of them. In this situation, GStreamer effectively chooses a pad randomly. This isn't what we want. **Caps filters** are our solution. They are essentially pass-through elements which only accepts media with the given capabilities, effectively resolving the ambiguity.

7.8 Evaluate the Output

We can now look at our output. Note that our motion detection algorithm was relatively successful. The bounding boxes generally follow the moving car, though not perfectly. The motion of the camera itself appears to be causing some issues.

In [ ]:
```
from IPython.display import HTML
HTML("""
<video width="640" height="480" controls>
  <source src="out_dsmotion.mp4" type="video/mp4">
</video>
""".format())
```

We learned that DeepStream supports custom plugins, and that we can use custom plugins in addition to or instead of existing plugins like **NvInfer**. We learned how to compose pipelines at the command line, and how that can help us in prototyping and rapidly visualizing our pipeline. We also learned how the basics of how a motion detection algorithm could work, and saw a simple implementation using OpenCV.

When we implement custom plugins though, we also must implement the code that handles all the necessary mechanics to treat it as a GStreamer plugin. Because of that, we covered the necessary functionality at a high level and downloaded the example implementation for future reference.

8. Object Detection on Multiple Input Streams

Up until now, we have looked at one video stream at a time. In some circumstances, though, we may want to use the same model to detect objects from multiple video streams. In this exercise, we will build a pipeline that accepts multiple input files and runs our object detection neural network on them. Like before, we will be building a pipeline at the command line for ease of explanation.

8.1 Batching Input Streams

DeepStream provides the plugins, **nvstreammux** and **nvstreamdemux**, to streamline processing multiple input streams. The video aggregator plugin (**nvstreammux**) forms a batch of buffers from the input sources and creates the necessary metadata to differentiate between the buffers of different source files. Once the batch is created, the DeepStream pipeline continues as expected. For example, in our pipeline, we will do object detection using a TensorRT optimized neural network via the NvInfer plugin to detect objects in the batch of frames.

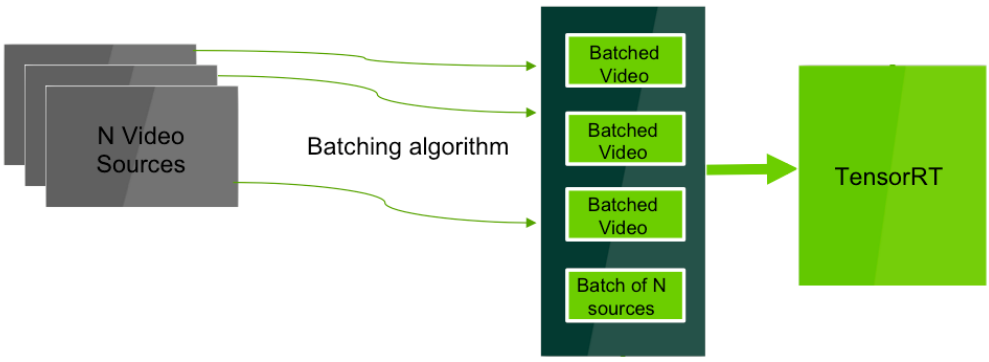


Figure 17. Multiple input pipeline

When we have finished inferencing, we need to separate the buffers so we can create output files for each of the input streams. We use the plugin **nvstreamdemux** to go from 1 batch to N separate streams. Because our DeepStream pipeline keeps track of the metadata, this process is simple and completely painless.

8.2 Composing a Pipeline

We are ready to compose a pipeline that runs detection on two video streams. Our pipeline is going to look a lot the one in the previous exercise. But there will be three key differences:

1. Because we are operating on multiple inputs, we need to create video parsing, decoding, conversion, and sink elements for each input. We do them separately to allow for inputs that require different types of processing
2. We include **nvstreammux** and its `name` property, to batch the input streams
3. We include **nvstreamdemux** and its `name` property, to separate the batched frames

We use the value supplied to the `name` property to easily access information in these two elements. When we want to access the first input stream from our batch, we do that with the notation `name.src_0`. In the same way, when we want to `sink` the first input stream from our batch, we can do that with the notation `name.sink_0`.

Look at the pipeline supplied below. It's almost working correctly, but not quite.

Exercise 5

Replace the `<<FIXME>>`s with the appropriate DeepStream plugins and references. As mentioned above, we need to compose a pipeline that batches the streams together, runs inference, and then separates the data. If you get stuck, you can look at the Answer Key, but we encourage you to try a few times before doing so. We have provided the information you need to solve this with a bit of trial and error. Remember, we want use `nvstreammux` before we use `nvstreamdemux`.

In [ ]:
```
!gst-launch-1.0 nvstreammux name=mux batch-size=1 width=1280 height=720 ! nvin
fer config-file-path=./DeepStream_Release/samples/configs/deepstream-app/confi
g_infer_primary.txt ! nvstreamdemux name=demux \
filesrc location=./DeepStream_Release/samples/streams/1.264 ! h264parse ! nvde
c_h264 ! queue ! mux.sink_0 \
filesrc location=./DeepStream_Release/samples/streams/2.264 ! decodebin ! queu
e ! mux.sink_1 \
<<FIXME>> ! "video/x-raw(memory:NVMM), format=NV12" ! queue ! nvvidconv ! "vid
eo/x-raw(memory:NVMM), format=RGBA" ! nvosd font-size=15 ! nvvidconv ! "video/
x-raw, format=RGBA" ! videoconvert ! "video/x-raw, format=NV12" ! x264enc ! qt
mux ! filesink location=./out3.mp4 \
<<FIXME>> ! "video/x-raw(memory:NVMM), format=NV12" ! queue ! nvvidconv ! "vid
eo/x-raw(memory:NVMM), format=RGBA" ! nvosd font-size=15 ! nvvidconv ! "video/
x-raw, format=RGBA" ! videoconvert ! "video/x-raw, format=NV12" ! x264enc ! qt
mux ! filesink location=./out4.mp4
```

If the pipeline was successful, you should see the printed output that looks like:

```
Setting pipeline to PAUSED ...
>>> Using TRT model serialized engine /dli/dl-deepstream-deployment/English/noteboo
ks/DeepStream_Release/samples/configs/deepstream-app/../../models/Primary_Detector/
resnet10.caffemodel_b30_int8.engine crypto flags(0)
Pipeline is PREROLLING ...
Redistribute latency...
Redistribute latency...
Redistribute latency...
Pipeline is PREROLLED ...
Setting pipeline to PLAYING ...
New clock: GstSystemClock
Got EOS from element "pipeline0".
Execution ended after 0:00:25.710410137
Setting pipeline to PAUSED ...
Setting pipeline to READY ...
Setting pipeline to NULL ...
Freeing pipeline ...
```

If your output looks like this, you are ready to look at the results!

In [ ]:
```
from IPython.display import HTML
HTML("""
    <video width="640" height="480" controls>
      <source src="./out3.mp4" type="video/mp4">
    </video>
""".format())
```

In [ ]:
```
from IPython.display import HTML
HTML("""
    <video width="640" height="480" controls>
      <source src="./out4.mp4" type="video/mp4">
    </video>
""".format())
```

Notice that one video is much longer than the other (in fact, one is a 19 second clip taken from the other), yet DeepStream was able to process both at the same time!

In this section, we learned how DeepStream makes it easy to handle multiple video streams at once. Using nvstreammux and nvstreamdemux, we can batch and separate video streams for efficient processing and analysis.

9. Dewarping 360° camera streams

All the video files we have examined so far, had flat rectangular frames, making the inference process immediate and straightforward. 360° camera feeds (also known as omnidirectional cameras) however, are growing in demand and consequently rapid pre-processing and *dewarping* become vital tasks.

The 360° camera's field of view covers the entire sphere (or a full circle of view) and provides a wide visual field coverage. these raw camera feeds are not useful for inference unless we `dewarp` them into flat video feeds and process them separately. `Dewarping` is a technique that mathematically corrects the deformed (warped) camera feeds so that all the curved lines turn back into straight lines. This is process is computationally expensive and delays the inference by expanding the preprocess step resulting in significant delays

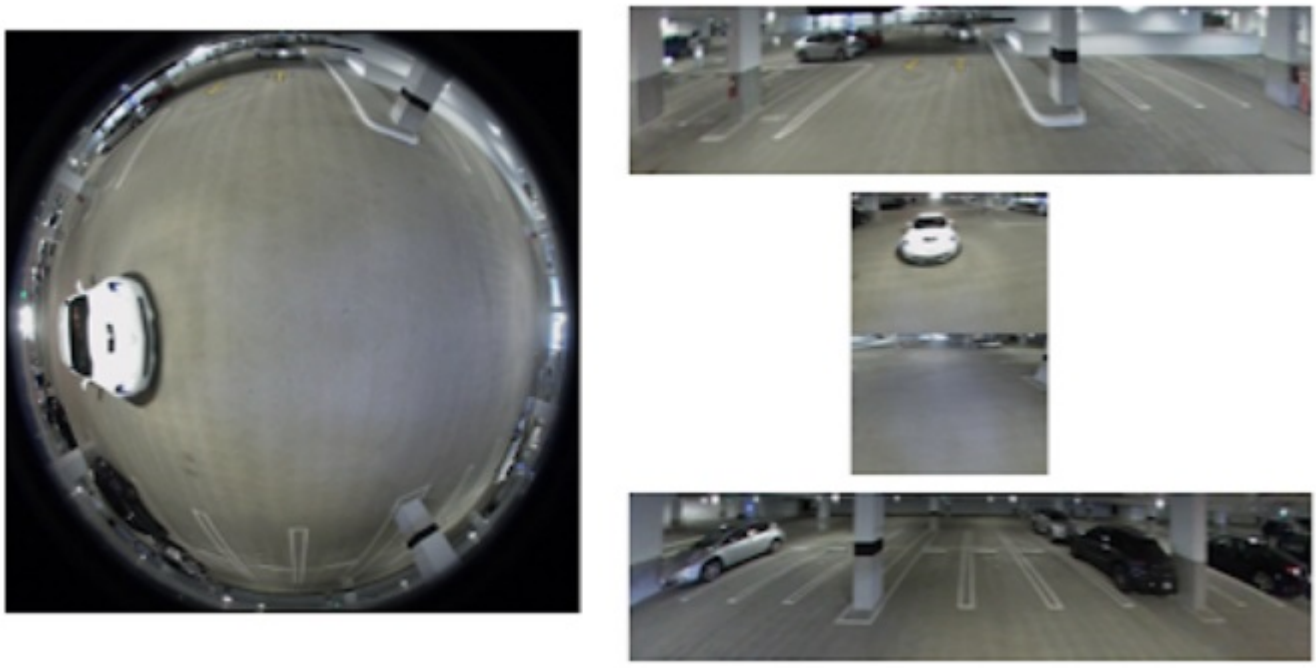
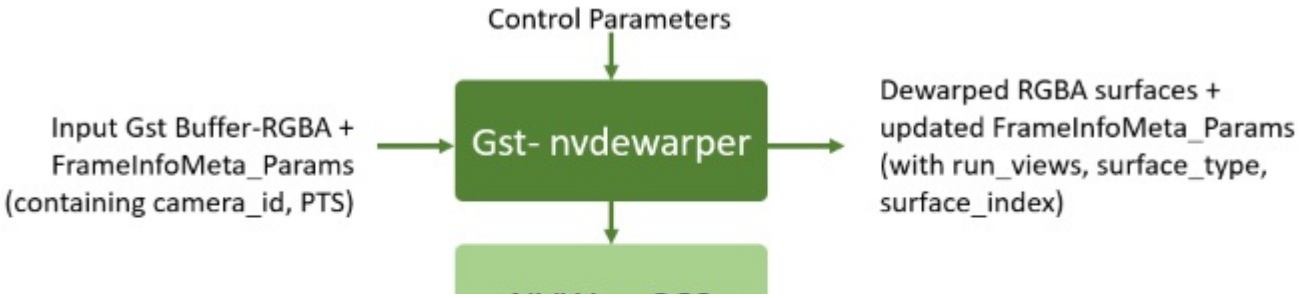


Figure 18. Dewarp process, left: 360° camera feed - right: 4 dewarped surfaces

In this section, we will work with the GPU-accelerated `gst-nvdewarper` plugin for dewarping camera feeds. The gst-nvdewarper plugin included in the SDK provides hardware-accelerated solutions to dewarp and transform an image to a planar projection. Reducing distortion makes these images more suitable for processing with existing deep learning models and viewing. The current plugin supports `pushbroom` and `vertically panned radical cylinder` projections.



In addition, the dewarper plugin requires a `config-file` which specifies the camera specs and the output videos properties. You can open the Open configuration file (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/apps/deepstream-test5/config_dewarper.txt) and inspect the paramters;

- **projection-type**: Selects projection type. Supported projection types are:1 = PushBroom, 2 = VertRadCyl
- **surface-index**: An index that distinguishes surfaces of the same projection type
- **width**: Dewarped surface width
- **height**: Dewarped surface height
- **top-angle**: Top field of view angle, in degrees
- **bottom-angle**: Bottom field of view angle, in degrees
- **pitch**: Viewing parameter pitch in degrees
- **yaw**: Viewing parameter yaw in degrees
- **roll**: Viewing parameter roll in degrees
- **focal-length**: Focal length of camera lens, in pixels per radian

The following pipeline takes a 360° camera feed and after decoding, the `nvdewarper` plugin dewaros the feed into four surfaces. We do not apply any inference for brevity and only save one of the surfaces as output:

In [ ]:
```
!gst-launch-1.0 filesrc location=DeepStream_Release/sources/apps/deepstream-te
st5/sample_cam6.mp4 ! \
qtdemux ! h264parse ! nvdec_h264 ! nvvidconv ! nvdewarper config-file=DeepStre
am_Release/sources/apps/deepstream-test5/config_dewarper.txt ! \
m.sink_0 nvstreammux name=m width=960 height=752 batch-size=4 ! \
nvvidconv ! capsfilter caps="video/x-raw, format=RGBA" ! videoconvert ! capsfi
lter caps="video/x-raw, format=NV12" ! \
x264enc ! qtmux ! filesink location=DeepStream_Release/sources/apps/deepstream
-test5/out.mp4
```

You may view the resulting video by executing the following cell:

In [ ]:
```
from IPython.display import HTML
HTML("""
    <video width="640" height="480" controls>
      <source src="./DeepStream_Release/sources/apps/deepstream-test5/out.mp4"
type="video/mp4">
    </video>
""".format())
```

10. Using "nvmsgconv" and "nvmsgbroker" plugins in the pipeline

An important capability of DeepStream which is added in version 3.0, is to encapsulate the generated metadata (tracks, bounding boxes, etc.) as messages and sending them for further analysis. These data could be used for a variety of tasks including anomaly detection, building long-term trends on location and movement and providing information dashboards for remote viewing, and more.

Let's consider the metadata generated by DeepStream first. At the end of the pipeline, the generated metadata represents the complete set of information extracted from the elements of the pipeline. We need to make this information accessible to the external consumers as part of a message for further analysis or long term archival.

10.1 The gst-nvmsgconv plugin

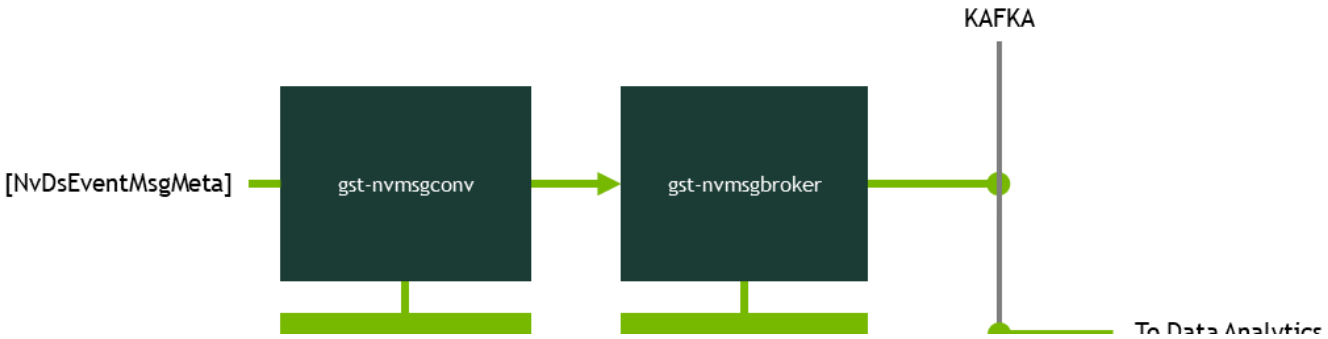
The `gst-nvmsgconv` plugin accepts a metadata structure of `NvDsEventMsgMeta` type and generates the corresponding **message payload**. this plugin provides a comprehensive JSON-based schema description has been defined that specifies events based on associated objects, location and time of occurrence, and underlying sensor information while specifying attributes for each of these event properties.

By default, the `NvDsEventMsgMeta` plugin generates messages based on the DeepStream schema description. However, it also allows the user to register their own metadata-to-payload converter functions for additional customizability. Using custom metadata descriptions in combination with user-defined conversion functions gives the user the ability to implement a fully custom event description and messaging capability that perfectly meets their needs.

10.2 The gst-nvmsgbroker plugin

After generating the augmented message with user data, the `gst-nvmsgbroker` plugin provides a message delivery mechanism using the **Apache Kafka** protocol to Kafka message brokers. Kafka serves as a conduit into a backend event analysis system, including those executing in the cloud. The high message throughput support, combined with reliability offered by the Kafka framework, enables the backend architecture to scale to support numerous DeepStream applications continuously sending messages.

The `gst-nvmsgbroker` plugin is implemented in a protocol agnostic manner and leverages protocol adapters in the form of shared libraries. These can be modified to support any user defined protocol. As we do not utilize a kafka instance, we are going to make a customized gst-nvmsgbroker (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/apps/deepstream-test4/broker/broker.cpp) plugin that dumps the metadata inside a text file instead of connecting to the kafka sever.



First, we are going to create a broker by implementing the `nvds_msgapi.h` API that removes all network communication codes and simply writes the metadata into a file. There are five main functions that this api implements:

- **nvds_msgapi_connect**: which connects to the broker and returns a message handler of type `NvDsMsgApiHandle`
- **nvds_msgapi_send**: for sending the payload string to the target synchronously
- **nvds_msgapi_send_async**: for sending the payload string to the target asynchronously
- **nvds_msgapi_disconnect**: terminating the connection and freeing up the resources
- **nvds_msgapi_getversion**: get the broker API version

Now let's build our broker and then we will see how to use this custom broker in our app.

In [ ]:
```
!rm DeepStream_Release/sources/apps/deepstream-test4/broker/*.so
!rm DeepStream_Release/sources/apps/deepstream-test4/logs.txt
!make -C DeepStream_Release/sources/apps/deepstream-test4/broker
```

While our broker has a minimal implementation of the broker class, the deepstream SDK is shipped with the full kafka implementation of the broker. You can look at the implementation here.

10.3 The "nvmsgconv" and "nvmsgbroker" example application

In this part, we provide an example (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/apps/deepstream-test4/deepstream_test4_app.c) of utilizing the "nvmsgconv" and "nvmsgbroker" plugins. This sample builds on top of the deepstream-test1 sample to demonstrate how to:

- Use "nvmsgconv" and "nvmsgbroker" plugins in the pipeline.
- Create NVDS_META_EVENT_MSG type of meta and attach to buffer.
- Use NVDS_META_EVENT_MSG for different types of objects e.g. vehicle, person etc.
- Provide copy / free functions if meta data is extended through "extMsg" field.

First, we make elements of both plugins:

```
/* Create msg converter to generate payload from buffer metadata */
  msgconv = gst_element_factory_make ("nvmsgconv", "nvmsg-converter");
  /* Create msg broker to send payload to server */
  msgbroker = gst_element_factory_make ("nvmsgbroker", "nvmsg-broker");
```

The `nvmsgconv` plugin uses NVDS_META_EVENT_MSG type of metadata from the buffer and generates the "DeepStream Schema" payload in Json format. Static properties of schema are read from configuration file in the form of key-value pair.

```
g_object_set (G_OBJECT(msgconv), "config", "dstest4_msgconv_config.txt", NULL);
```

Check `dstest4_msgconv_config.txt` for reference. Generated payload is attached as `NVDS_META_PAYLOAD` type metadata to the buffer. Note that in addition to common fields provided in `NvDsEventMsgMeta` structure, user can also create custom objects and attach to buffer as `NVDS_META_EVENT_MSG` metadata. To do that `NvDsEventMsgMeta` provides "extMsg" and `extMsgSize` fields. User can

create custom structure, fill that structure and assign the pointer of that structure as `extMsg` and set the `extMsgSize` accordingly. If custom object contains fields that can't be simply mem copied, then user should also provide function to copy and free those objects.

Finally, `nvmsgbroker` plugin extracts `NVDS_META_PAYLOAD` type of metadata from the buffer and sends that payload to the server using protocol adaptor APIs. We use the library we built previously as the broker library:

```
#define PROTOCOL_ADAPTOR_LIB  "broker/broker.so"
...
g_object_set (G_OBJECT(msgbroker), "proto-lib", PROTOCOL_ADAPTOR_LIB,
              "conn-str", CONNECTION_STRING, "sync", FALSE, NULL);
```

In [ ]:
```
!rm DeepStream_Release/sources/apps/deepstream-test4/deepstream-test4-app
!rm DeepStream_Release/sources/apps/deepstream-test4/deepstream_test4_app.o
!make -C DeepStream_Release/sources/apps/deepstream-test4/

!cd DeepStream_Release/sources/apps/deepstream-test4 && \
./deepstream-test4-app  ../../../samples/streams/sample_720p.h264
```

You may have noticed that we write the broker message into the logs.txt file. Let's have a look at the content of this file:

In [ ]:
```
!head DeepStream_Release/sources/apps/deepstream-test4/logs.txt -n 1000
```

As you can see the meta-data from the `dstest4_msgconv_config.txt` is attached as `NVDS_META_PAYLOAD` type metadata to the buffer as a custom buffer. Moreover, you can add the meta-data dynamically to other elements of the pipeline in a similar manner by adding a probe to the element event and attaching `NVDS_META_PAYLOAD` data.

11. Performance Analysis

11.1 Generating Performance KPIs

In this section, we will learn how to generate performance KPIs for the workload created in section 4. The key metrics we will highlight are:

- Throughput: Using gst probes installed in the display plugin
- GPU Performance Metrics: Using nvidia-smi
- Latency: Using gst logs / gst tracer

11.2 Throughput

Just as we previously used probes to access metadata, we can also use probes (https://gstreamer.freedesktop.org/documentation/application-development/advanced/pipeline-manipulation.html#using-probes) to measure throughput and get a sense of the dataflow. As we saw before, we installed our function for metadata handling with `gst_pad_add_probe` .

Below, instead of handling metadata, we are going to install a second probe that will log the time it takes for a frame to pass through the on screen display plugin. We are going to print out our frame rate for a single video stream, but we could expand this to generate metrics such as minimum, maximum, and average frame throughput at a granular level for each video stream.

Open deepstream_test1_app.c (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/apps/deepstream-test1/deepstream_test1_app.c) and uncomment the following lines of code (lines 348 and 349 unless you have added new lines to the file).

```
perf_probe_id = gst_pad_add_probe (osd_sink_pad, GST_PAD_PROBE_TYPE_BUFFER, sink_bi
n_buf_probe, NULL, NULL);
```

With this line, we are adding another user-defined callback function, `sink_bin_buf_probe` , to our pipeline (it's defined beginning at line 130 in the code). We have copied the callback function below for ease of explanation.

```
static GstPadProbeReturn
sink_bin_buf_probe (GstPad * pad, GstPadProbeInfo * info, gpointer u_data)
{

  static struct timeval last_fps_time;
  static struct timeval curr_fps_time;
  static int numbuff = 0;

  /* this parameter determines after how many frames fps metrics are reported */
  const int FPS_DISPLAY_FREQ = 10;

  if ((numbuff++ % FPS_DISPLAY_FREQ) == 0) {
    //last fps is not recorded
    if (last_fps_time.tv_sec == 0
        && last_fps_time.tv_usec == 0)
      /* first time around */
```

In [ ]:
```
!make -C DeepStream_Release/sources/apps/deepstream-test1 clean

!make -C DeepStream_Release/sources/apps/deepstream-test1
```

With the application built, we are ready to run it.

In [ ]:
```
!cd DeepStream_Release/sources/apps/deepstream-test1 && \
./deepstream-test1-app  ../../../samples/streams/sample_720p.h264
```

If you scroll through the output, you can see that we are operating with a high frame rate, but that it does vary. Logging the minimum, maximum, and average frame rates for your applications is highly recommended to understand any potentially unexpected performance issues.

11.3 GPU Performance Metrics

We can use `nvidia-smi` to explore the GPU performance metrics while our application is running. GPU utilization is something we want to pay attention to, and we will discuss it below. Run the cell below to re-run the application while logging the results of `nvidia-smi` .

In [ ]:
```
%%bash
cd DeepStream_Release/sources/apps/deepstream-test1
nvidia-smi dmon -i 0 -s ucmt -c 20 > smi.log &
./deepstream-test1-app  ../../../samples/streams/sample_720p.h264
```

Open smi.log (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/apps/deepstream-test1/smi.log) to investigate our utilization metrics. If you have the file open while the pipeline is running, you will need to refresh the `smi.log` page to see updated metrics. You can also see example output below:

```
# gpu     sm    mem    enc    dec   mclk   pclk     fb   bar1 rxpci txpci
# Idx      %      %      %      %    MHz    MHz     MB     MB  MB/s  MB/s
    0      0      0      0      0    877   1312      0      2    13     2
    0      8      0      0      0    877   1312    640      7     0    92
    0     79     14      0      0    877   1530   1192      7   828     9
    0     57      9      0      0    877   1530   1180      7    40    11
    0      4      0      0      2    877   1530    896      7   208     0
    0      4      0      0      3    877   1530    896      7   206   236
    0      4      0      0      3    877   1530    896      7     0   236
    0      4      0      0      3    877   1312    896      7   207     0
    0      4      0      0      3    877   1312    896      7   206   236
    0      4      0      0      3    877   1312    896      7   206   236
    0      4      0      0      3    877   1312    896      7   205   235
    0      4      0      0      3    877   1312    896      7     0   236
    0      4      0      0      3    877   1312    896      7   210     0
    0      4      0      0      3    877   1312    896      7   205   236
    0      4      0      0      3    877   1312    896      7     0   236
    0      4      0      0      3    877   1312    896      7   206     0
    0      4      0      0      3    877   1312    896      7   208   236
    0      4      0      0      3    877   1312    896      7   206   236
    0      4      0      0      3    877   1312    896      7   210   232
    0      4      0      0      3    877   1312    896      7     0   235
```

Understanding nvidia-smi

The cell block above passed the following arguments to `nvidia-smi`:

- `dmon -i 0`:
  - Reports default metrics (device monitoring) for the devices selected by comma separated device list. In this case, we are reporting default metrics for GPU with index 0 since that is the GPU we are using.

- `-s ucmt`:
  - We can choose which metrics we want to display. In this case, we supplied `ucmt` to indicate we want metrics for
    - u: Utilization (SM, Memory, Encoder and Decoder Utilization in %)
    - c: Proc and Mem Clocks (in MHz)
    - m: Frame Buffer and Bar1 memory usage (in MB)
    - t: PCIe Rx and Tx Throughput in MB/s (Maxwell and above)
- `-c 20`
  - We can configure the number of iterations for which we are monitoring. In this case, we choose 20 iterations.

Let's dive a bit deeper into a few of the metrics we selected, since they are particularly useful to monitor.

Utilization metrics report how busy each GPU is over time and can be used to determine how much an application is using the GPUs in the system. In particular, the **sm** column tracks the percent of time over the past sample period during which one or more kernels was executing on the GPU. **fb** reports the GPU's frame buffer memory usage. For those interested in learning more, more detailed and comprehensive documentation can be accessed here (http://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf).

Take a quick look at your metrics in `smi.log`. Does it look like our GPU was highly utilized? Keep your answer in your head -- we will explore utilization in the next section.

11.4 Generating Latency Metrics with GStreamer Logs

We can also leverage GStreamer's built in debugging (https://gstreamer.freedesktop.org/documentation/tutorials/basic/debugging-tools.html#the-debug-log) capabilities to measure latency across our pipeline. By setting `GST_DEBUG` environment variables, we can control what kinds of logs we get. `GST_SCHEDULING:7` is the Trace setting, which logs all trace messages. This logs every time things like Buffers are modified, giving us detailed information about our pipeline.

Use the cell below to run our application with debugging set to the Trace setting.

In [ ]:
```
!cd DeepStream_Release/sources/apps/deepstream-test1 && \
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/opencv-3.4.1/lib && \
GST_DEBUG="GST_SCHEDULING:7" GST_DEBUG_FILE=trace.log ./deepstream-test1-app
../../../samples/streams/sample_720p.h264
```

Open trace.log (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/sources/apps/deepstream-test1/trace.log). This is a huge file, so we will highlight an instance of the decoder plugin latency from the trace log.

```
0:00:05.137672361  <omxh264dec-omxh264dec0:sink> calling chainfunction &gst_video_d
ecoder_chain with buffer buffer: 0x7f9c02bc00, pts 0:00:35.719066666, dts 0:00:35.7
19033333, dur 0:00:00.016683333, size 56047, offset 172403525, offset_end none, fla
gs 0x2400
0:00:05.170122161  <src_0:proxypad3> calling chainfunction &gst_proxy_pad_chain_def
ault with buffer buffer: 0x7f714c5020, pts 0:00:35.719066666, dts 99:99:99.99999999
9, dur 0:00:00.016683333, size 808, offset none, offset_end none, flags 0x0
```

You can see the time a frame entered at the input of the decoder plugin to when it entered the next input is **33ms** which would equate to roughly **30** frames per second.

11.5 More on GPU Utilization

In this last section, we will switch to using our section 5 application to explore how utilization metrics vary with the workload. Specifically, we will look at how utilization varies with:

- Different number of input streams
- The addition of secondary inference components to the pipeline

Based on these results, you will begin to see factors that ultimately saturate pipeline performance.

Open configs/deepstream-app/source4_720p_dec_infer-resnet_tracker_sgie_tiled_display_int8.txt (../../../../edit/tasks/l-iv-04/task/DeepStream_Release/samples/configs/deepstream-app/source4_720p_dec_infer-resnet_tracker_sgie_tiled_display_int8.txt) and keep it open to change stream count later on.

In [ ]:
```
 rm ~/.cache/gstreamer-1.0/ -rf
```

In [ ]:
```
!LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/include/opencv && \
cd DeepStream_Release/samples && \
sh -c 'nvidia-smi dmon -i 0 -s ucmt -c 20 > smi.log &' && \
deepstream-app -c configs/deepstream-app/source4_720p_dec_infer-resnet_tracker
_sgie_tiled_display_int8.txt
```

Open smi.log (../edit/DeepStream_Release/samples/smi.log) and observe decoder and SM utilization. It should look pretty similar to the output from the previous section, with several utilization metrics slightly higher.

Now vary stream count to 16 and 32 streams by modifying **num-sources** under the **[source0]** section within the source4_720p_dec_infer-resnet_tracker_sgie_tiled_display_int8.txt file you just opened. Also remember to change the **batch-size** in **[primary-gie]** to match the stream count. Then, re-run the application with the modified stream count.

What changes do you notice when you vary the stream count?

1. Are we still able to achieve 30 frames per second?
2. Does the GPU utilization increase or decrease?
3. What changes happen to our frame buffer memory usage?

Summary

Congratulations! You have used DeepStream SDK to design real-life IVA projects. By now, you should have an understanding of:

- The building blocks of DeepStream
- How to use DeepStream for single and multiple neural network-based inferencing
- How to access and handle metadata
- How to use the NvInfer component to accommodate any TensorRT compatible neural network detector
- How to customize a DeepStream pipeline and incorporate custom image processing using C++ code
- How to batch multiple input streams together into a single DeepStream pipeline
- How to use unwarp plugin for 360 camera streams
- how to utilize "nvmsgconv" and "nvmsgbroker" plugins in the pipeline
- Performance metrics and how to measure them

Whether you are well underway in developing your own IVA application or you are exploring potential opportunities, DeepStream can help you accelerate your IVA application development using Deep learning on NVIDIA hardware. We hope you have enjoyed this course!

Answer Key

Plugins

In [ ]:
```
!gst-inspect-1.0 nvosd
```

Click here to return to where you were.

Handling Multiple Inputs

In [ ]:
```
!gst-launch-1.0 nvstreammux name=mux batch-size=1 width=1280 height=720 ! nvin
fer config-file-path=./DeepStream_Release/samples/configs/deepstream-app/confi
g_infer_primary.txt ! nvstreamdemux name=demux \
filesrc location=./DeepStream_Release/samples/streams/1.264 ! h264parse ! nvde
c_h264 ! queue ! mux.sink_0 \
filesrc location=./DeepStream_Release/samples/streams/2.264 ! decodebin ! queu
e ! mux.sink_1 \
demux.src_0 ! "video/x-raw(memory:NVMM), format=NV12" ! queue ! nvvidconv ! "v
ideo/x-raw(memory:NVMM), format=RGBA" ! nvosd font-size=15 ! nvvidconv ! "vide
o/x-raw, format=RGBA" ! videoconvert ! "video/x-raw, format=NV12" ! x264enc !
 qtmux ! filesink location=./out3.mp4 \
demux.src_1 ! "video/x-raw(memory:NVMM), format=NV12" ! queue ! nvvidconv ! "v
ideo/x-raw(memory:NVMM), format=RGBA" ! nvosd font-size=15 ! nvvidconv ! "vide
o/x-raw, format=RGBA" ! videoconvert ! "video/x-raw, format=NV12" ! x264enc !
 qtmux ! filesink location=./out4.mp4
```