# DEEPSTREAM 4.0 PLUGIN MANUAL

SWE-SWDOCDPSTR-002-PGRF | August 6, 2019
Advance Information | Subject to Change

**Application Note**

# DOCUMENT CHANGE HISTORY

| Date | Author | Revision History |
|---|---|---|
| Nov. 19, 2018 | Bhushan Rupde, Jonathan Sachs | Release 3.0 (Initial release) |
| Aug. 6, 2019 | Bhushan Rupde, Jonathan Sachs | Release 4.0 (Unified release) |

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# 1.0 INTRODUCTION

DeepStream SDK is based on the GStreamer framework. This manual describes the DeepStream GStreamer plugins and the DeepStream input, outputs, and control parameters.

DeepStream SDK is supported on systems that contain an NVIDIA® Jetson™ module or an NVIDIA dGPU adapter.[1]

The manual is intended for engineers who want to develop DeepStream applications or additional plugins using the DeepStream SDK. It also contains information about metadata used in the SDK. Developers can add custom metadata as well.

The manual describes the methods defined in the SDK for implementing custom inferencing layers using the `IPlugin` interface of TensorRT™.

You can refer the sample examples shipped with the SDK as you use this manual to familiarize yourself with DeepStream application and plugin development.

[1] This manual uses the term *dGPU* ("discrete GPU") to refer to NVIDIA GPU expansion card products such as NVIDIA® Tesla® T4 and P4, NVIDIA® GeForce® GTX 1080, and NVIDIA® GeForce® RTX 2080. This version of DeepStream SDK runs on specific dGPU products on x86_64 platforms supported by NVIDIA driver 418+ and NVIDIA® TensorRT™ 5.1 and later versions.

# 2.0 GSTREAMER PLUGIN DETAILS

## 2.1 GST-NVINFER

The `Gst-nvinfer` plugin does inferencing on input data using NVIDIA® TensorRT™.

The plugin accepts batched NV12/RGBA buffers from upstream. The `NvDsBatchMeta` structure must already be attached to the Gst Buffers.

The low-level library (`libnvds_infer`) operates on any of INT8 RGB, BGR, or GRAY data with dimension of Network Height and Network Width.

The `Gst-nvinfer` plugin performs transforms (format conversion and scaling), on the input frame based on network requirements, and passes the transformed data to the low-level library.

The low-level library preprocesses the transformed frames (performs normalization and mean subtraction) and produces final float RGB/BGR/GRAY planar data which is passed to the TensorRT engine for inferencing. The output type generated by the low-level library depends on the network type.

The pre-processing function is:

$$y = net\text{-}scale\text{-}factor * (x - mean)$$

Where:

- $x$ is the  input pixel value. It is an *int8* with range [0,255].
- *mean* is the corresponding mean value, read either from the mean file or as `offsets[c]`, where `c` is the channel to which the input pixel belongs, and `offsets` is the array specified in the configuration file. It is a *float*.
- *net-scale-factor* is the pixel scaling factor specified in the configuration file. It is a *float*.
- $y$ is the corresponding output pixel value. It is a *float*.

`nvinfer` currently works on the following type of networks:

- Multi-class object detection
- Multi-label classification
- Segmentation

The `Gst-nvinfer` plugin can work in two modes:

- **Primary mode**: Operates on full frames
- **Secondary mode**: Operates on objects added in the meta by upstream components

When the plugin is operating as a secondary classifier along with the tracker, it tries to improve performance by avoiding re-inferencing on the same objects in every frame. It does this by caching the classification output in a map with the object's unique ID as the key. The object is inferred upon only when it is first seen in a frame (based on its object ID) or when the size (bounding box area) of the object increases by 20% or more. This optimization is possible only when the tracker is added as an upstream element.

Detailed documentation of the TensorRT interface is available at:

```
https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html
```

The plugin supports the `IPlugin` interface for custom layers. Refer to section IPlugin Interface for details.

The plugin also supports the interface for custom functions for parsing outputs of object detectors and initialization of non-image input layers in cases where there are more than one input layer.

Refer to `sources/includes/nvdsinfer_custom_impl.h` for the custom method implementations for custom models.

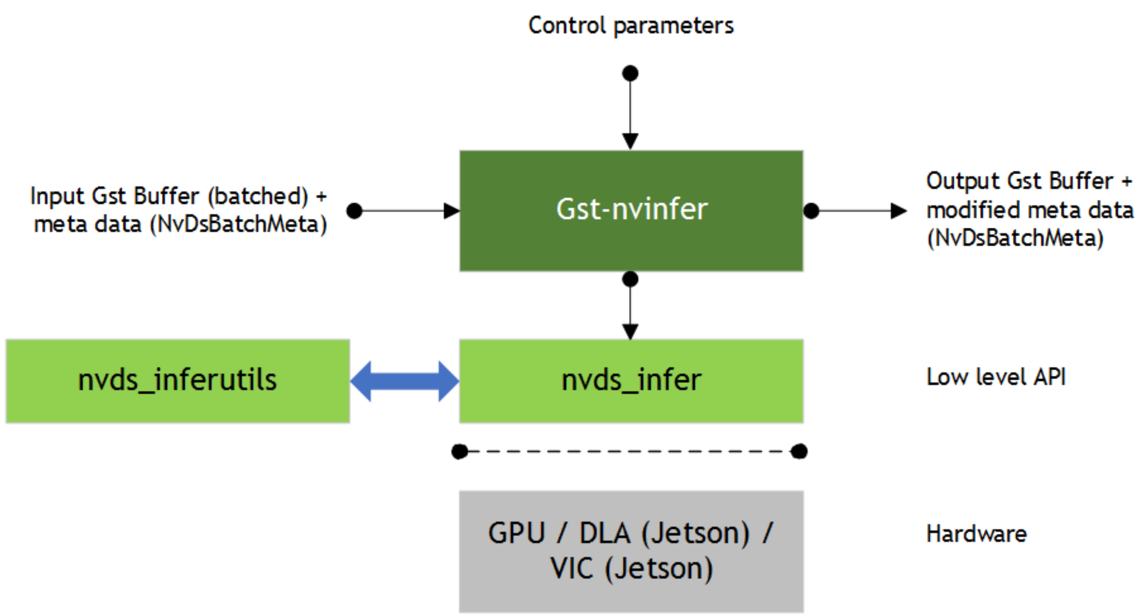


Figure 1. Gst-nvinfer inputs and outputs

Downstream components receive a Gst Buffer with unmodified contents plus the metadata created from the inference output of the `Gst-nvinfer` plugin.

The plugin can be used for cascaded inferencing. That is, it can perform **primary inferencing** directly on input data, then perform **secondary inferencing** on the results of primary inferencing, and so on. See the sample application `deepstream-test2` for more details.

### 2.1.1 Inputs and Outputs

This section summarizes the inputs, outputs, and communication facilities of the Gst-nvinfer plugin.

- Inputs
  - Gst Buffer
  - `NvDsBatchMeta` (attaching `NvDsFrameMeta`)
  - Caffe Model and Caffe Prototxt
  - ONNX
  - UFF file
  - TLT Encoded Model and Key
  - Offline: Supports engine files generated by Transfer Learning Toolkit SDK Model converters

Layers: Supports all layers supported by TensorRT, see:

```
https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html
```

- Control parameters: `Gst-nvinfer` gets control parameters from a configuration file. You can specify this by setting the property `config-file-path`. For details, see Gst-nvinfer File Configuration Specifications. Other control parameters that can be set through `GObject` properties are:
  - Batch size
  - Inference interval
  - Attach inference tensor outputs as buffer metadata
- The parameters set through the GObject properties override the parameters in the Gst-nvinfer configuration file.
- Outputs
  - Gst Buffer
  - Depending on network type and configured parameters, one or more of:
    - NvDsObjectMeta
    - NvDsClassifierMeta
    - NvDsInferSegmentationMeta
    - NvDsInferTensorMeta

### 2.1.2 Features

Table 1 summarizes the features of the plugin.

Table 1. Features of the Gst-nvinfer plugin

| Feature | Description | Release |
|---|---|---|
| Transfer-Learning-Toolkit encoded model support | — | DS 4.0 |
| Gray input model support | Support for models with single channel gray input | DS 4.0 |
| Tensor output as meta | Raw tensor output is attached as meta data to Gst Buffers and flowed through the pipeline | DS 4.0 |
| Segmentation model | Supports segmentation model | DS 4.0 |
| Maintain input aspect ratio | Configurable support for maintaining aspect ratio when scaling input frame to network resolution | DS 4.0 |

| Feature | Description | Release |
|---|---|---|
| Custom cuda engine creation interface | Interface for generating CUDA engines from TensorRT `INetworkDefinition` and `IBuilder` APIs instead of model files | DS 4.0 |
| Caffe Model support | — | DS 2.0 |
| UFF Model support | — | DS 3.0 |
| ONNX Model support | — | DS 3.0 |
| Multiple modes of operation | Support for cascaded inferencing | DS 2.0 |
| Asynchronous mode of operation for secondary inferencing | Infer asynchronously for secondary classifiers | DS 2.0 |
| Grouping using CV::Group rectangles | For detector bounding box clustering | DS 2.0 |
| Configurable batch-size processing | User can configure batch size for processing | DS 2.0 |
| No Restriction on number of output blobs | Supports any number of output blobs | DS 3.0 |
| Configurable number of detected classes (detectors) | Supports configurable number of detected classes | DS 3.0 |
| Support for Classes: configurable (> 32) | Support any number of classes | DS 3.0 |
| Application access to raw inference output | Application can access inference output buffers for user specified layer | DS 3.0 |
| Support for single shot detector (SSD) | — | DS 3.0 |
| Secondary GPU Inference Engines (GIEs) operate as detector on primary bounding box | Support secondary inferencing as detector | DS 2.0 |
| Multiclass secondary support | Support multiple classifier network outputs | DS 2.0 |
| Grouping using DBSCAN | For detector bounding box clustering | DS 3.0 |
| Loading an external lib containing IPlugin implementation for custom layers (IPluginCreator & IPluginFactory) | Supports loading (`dlopen()` ) a library containing IPlugin implementation for custom layers | DS 3.0 |
| Multi GPU | Select GPU on which we want to run inference | DS 2.0 |
| Detection width height configuration | Filter out detected objects based on min/max object size threshold | DS 2.0 |
| Allow user to register custom parser | Supports final output layer bounding box parsing for custom detector network | DS 2.0 |
| Bounding box filtering based on configurable object size | Supports inferencing in secondary mode objects meeting min/max size threshold | DS 2.0 |
| Configurable operation interval | Interval for inferencing (number of batched buffers skipped) | DS 2.0 |
| Select Top and bottom regions of interest (RoIs) | Removes detected objects in top and bottom areas | DS 2.0 |
| Operate on Specific object type (Secondary mode) | Process only objects of define classes for secondary inferencing | DS 2.0 |

| Feature | Description | Release |
|---|---|---|
| Configurable blob names for parsing bounding box (detector) | Support configurable names for output blobs for detectors | DS 2.0 |
| Allow configuration file input | Support configuration file as input (mandatory in DS 3.0) | DS 2.0 |
| Allow selection of class id for operation | Supports secondary inferencing based on class ID | DS 2.0 |
| Support for Full Frame Inference: Primary as a classifier | Can work as classifier as well in primary mode | DS 2.0 |
| Multiclass secondary support | Support multiple classifier network outputs | DS 2.0 |
| Secondary GIEs operate as detector on primary bounding box<br>Support secondary inferencing as detector | — | DS 2.0 |
| Supports FP16, FP32 and INT8 models<br>FP16 and INT8 are platform dependent | — | DS 2.0 |
| Supports TensorRT Engine file as input | — | DS 2.0 |
| Inference input layer initialization<br>Initializing non-video input layers in case of more than one input layers | — | DS 3.0 |
| Support for FasterRCNN | — | DS 3.0 |
| Support for Yolo detector (YoloV3/V3-tiny/V2/V2-tiny) | — | DS 4.0 |

### 2.1.3 Gst-nvinfer File Configuration Specifications

The `Gst-nvinfer` configuration file uses a "Key File" format described in:

```
https://specifications.freedesktop.org/desktop-entry-spec/latest
```

The `[property]` group configures the general behavior of the plugin. It is the only mandatory group.

The `[class-attrs-all]` group configures detection parameters for all classes.

The `[class-attrs-<class-id>]` group configures detection parameters for a class specified by `<class-id>`. For example, the `[class-attrs-23]` group configures detection parameters for class ID 23. This type of group has the same keys as `[class-attrs-all]`.

Table 2 and Table 3, respectively describe the keys supported for `[property]` groups and `[class-attrs-…]` groups.

Table 2. Gst-nvinfer plugin, [property] group, supported keys

| Property | Meaning | Type and Range | Example Notes | Network Types / *Applicable to GIEs (Primary/Seconday)* |
|---|---|---|---|---|
| num-detected-classes | Number of classes detected by the network | Integer, >0 | num-detected-classes=91 | Detector<br>*Both* |
| net-scale-factor | Pixel normalization factor | Float, >0.0 | net-scale-factor=0.031 | All<br>*Both* |
| model-file | Pathname of the caffemodel file. Not required if `model-engine-file` is used | String | model-file=/home/ubuntu/model.caffemodel | All<br>*Both* |
| proto-file | Pathname of the prototxt file. Not required if `model-engine-file` is used | String | proto-file=/home/**ubuntu**/model.prototxt | All<br>*Both* |
| int8-calib-file | Pathname of the INT8 calibration file for dynamic range adjustment with an FP32 model | String | int8-calib-file=/home/ubuntu/int8_calib | All<br>*Both* |
| batch-size | Number of frames or objects to be inferred together in a batch | Integer, >0 | batch-size=30 | All<br>*Both* |
| model-engine-file | Pathname of the serialized model engine file | String | model-engine-file=/home/ubuntu/model.engine | All<br>*Both* |
| uff-file | Pathname of the UFF model file | String | uff-file=/home/ubuntu/model.uff | All<br>*Both* |
| onnx-file | Pathname of the ONNX model file | String | onnx-file=/home/ubuntu/model.onnx | All<br>*Both* |
| enable-dbscan | Indicates whether to use DBSCAN or the OpenCV `groupRectangles()` function for grouping detected objects | Boolean | enable-dbscan=1 | Detector<br>*Both* |
| labelfile-path | Pathname of a text file containing the labels for the model | String | labelfile-path=/home/ubuntu/model_labels.txt | Detector & classifier<br>*Both* |
| mean-file | Pathname of mean data file (PPM format) | String | mean-file=/home/ubuntu/model_meanfile.ppm | All<br>*Both* |

| | | | | Network Types / *Applicable to GIEs (Primary/Seconday)* |
|---|---|---|---|---|
| **Property** | **Meaning** | **Type and Range** | **Example *Notes*** | |
| gie-unique-id | Unique ID to be assigned to the GIE to enable the application and other elements to identify detected bounding boxes and labels | Integer, >0 | gie-unique-id=2 | All<br>*Both* |
| operate-on-gie-id | Unique ID of the GIE on whose metadata (bounding boxes) this GIE is to operate on | Integer, >0 | operate-on-gie-id=1 | All<br>*Both* |
| operate-on-class-ids | Class IDs of the parent GIE on which this GIE is to operate on | Semicolon delimited integer array | operate-on-class-ids=1;2<br>*Operates on objects with class IDs 1, 2 generated by parent GIE* | All<br>*Both* |
| interval | Specifies the number of consecutive batches to be skipped for inference | Integer, >0 | interval=1 | All<br>*Primary* |
| input-object-min-width | Secondary GIE infers only on objects with this minimum width | Integer, ≥0 | input-object-min-width=40 | All<br>*Secondary* |
| input-object-min-height | Secondary GIE infers only on objects with this minimum height | Integer, ≥0 | input-object-min-height=40 | All<br>*Secondary* |
| input-object-max-width | Secondary GIE infers only on objects with this maximum width | Integer, ≥0 | input-object-max-width=256<br>*0 disables the threshold* | All<br>*Secondary* |
| input-object-max-height | Secondary GIE infers only on objects with this maximum height | Integer, ≥0 | input-object-max-height=256<br>*0 disables the threshold* | All<br>*Secondary* |
| uff-input-dims | Dimensions of the UFF model | channel; height; width; input-order<br>All integers, ≥0 | input-dims=3;224;224;0<br>*Possible values for input-order are:*<br>*0: NCHW*<br>*1: NHWC* | All<br>*Both* |
| network-mode | Data format to be used by inference | Integer<br>0: FP32<br>1: INT8<br>2: FP16 | network-mode=0 | All<br>*Both* |

| Property | Meaning | Type and Range | Example *Notes* | Network Types / ***Applicable to GIEs (Primary/Seconday)*** |
|---|---|---|---|---|
| offsets | Array of mean values of color components to be subtracted from each pixel. Array length must equal the number of color components in the frame. The plugin multiplies mean values by `net-scale-factor`. | Semicolon delimited float array, all values ≥0 | offsets=77.5;21.2;11.8 | All<br>*Both* |
| output-blob-names | Array of output layer names | Semicolon delimited string array | **For detector**:<br>output-blob-names=coverage;bbox<br>**For multi-label classifiers**:<br>output-blob-names=coverage_attrib1;coverage_attrib2 | All<br>*Both* |
| parse-bbox-func-name | Name of the custom bounding box parsing function. If not specified, Gst-nvinfer uses the internal function for the resnet model provided by the SDK. | String | parse-bbox-func-name=<br>parse_bbox_resnet | Detector<br>*Both* |
| custom-lib-path | Absolute pathname of a library containing custom method implementations for custom models | String | custom-lib-path=<br>/home/ubuntu/<br>libresnet_custom_impl.so | All<br>*Both* |
| model-color-format | Color format required by the model. | Integer<br>0: RGB<br>1: BGR | model-color-format=0 | All<br>*Both* |

| Property | Meaning | Type and Range | Example *Notes* | Network Types / *Applicable to GIEs (Primary/Seconday)* |
|---|---|---|---|---|
| classifier-async-mode | Enables inference on detected objects and asynchronous metadata attachments. Works only when tracker-ids are attached. Pushes buffer downstream without waiting for inference results. Attaches metadata after the inference results are available to next Gst Buffer in its internal queue. | Boolean | classifier-async-mode=1 | Classifier *Secondary* |
| process-mode | Mode (primary or secondary) in which the element is to operate on | Integer 1=Primary 2=Secondary | gie-mode=1 | All *Both* |
| classifier-threshold | Minimum threshold label probability. The GIE outputs the label having the highest probability if it is greater than this threshold | Float, ≥0 | classifier-threshold=0.4 | Classifier *Both* |
| uff-input-blob-name | Name of the input blob in the UFF file | String | uff-input-blob-name=Input_1 | All *Both* |
| secondary-reinfer-interval | Reinference interval for objects, in frames | Integer, ≥0 | secondary-reinfer-interval=15 | Classifier *Secondary* |
| output-tensor-meta | Gst-nvinfer attaches raw tensor output as Gst Buffer metadata. | Boolean | output-tensor-meta=1 | All *Both* |
| enable-dla | Indicates whether to use the DLA engine for inferencing.<br>**Note**: DLA is supported only on Jetson AGX Xavier™. Currently work in progress. | Boolean | enable-dla=1 | All *Both* |
| use-dla-core | DLA core to be used.<br>**Note**: Supported only on Jetson AGX Xavier™. Currently work in progress. | Integer, ≥0 | use-dla-core=0 | All *Both* |

| Network Types / *Applicable to GIEs (Primary/Seconday)* | | | | |
|---|---|---|---|---|
| **Property** | **Meaning** | **Type and Range** | **Example** ***Notes*** | |
| network-type | Type of network | Integer<br>0: Detector<br>1: Classifier<br>2: Segmentation | network-type=1 | All<br>*Both* |
| maintain-aspect-ratio | Indicates whether to maintain aspect ratio while scaling input. | Boolean | maintain-aspect-ratio=1 | All<br>*Both* |
| parse-classifier-func-name | Name of the custom classifier output parsing function. If not specified, Gst-nvinfer uses the internal parsing function for softmax layers. | String | parse-classifier-func-name= parse_bbox_softmax | Classifier<br>*Both* |
| custom-network-config | Pathname of the configuration file for custom networks available in the custom interface for creating CUDA engines. | String | custom-network-config=/home/ ubuntu/ network.config | All<br>*Both* |
| tlt-encoded-model | Pathname of the Transfer Learning Toolkit (TLT) encoded model. | String | tlt-encoded-model= /home/ubuntu/ model.etlt | All<br>*Both* |
| tlt-model-key | Key for the TLT encoded model. | String | tlt-model-key=abc | All<br>*Both* |
| segmentation-threshold | Confidence threshold for the segmentation model to output a valid class for a pixel. If confidence is less than this threshold, class output for that pixel is −1. | Float, ≥0.0 | segmentation-threshold=0.3 | Segmentation<br>*Both* |

Table 3. Gst-nvinfer plugin, [class-attrs-...] groups, supported keys

| Detector or Classifier / *Applicable to GIEs (Primary/Seconday)* | | | | |
|---|---|---|---|---|
| **Name** | **Description** | **Type and Range** | **Example** ***Notes*** | |
| threshold | Detection threshold | Float, >=0 | threshold=0.5 | Object detector<br>*Both* |

| Name | Description | Type and Range | Example Notes | Detector or Classifier / *Applicable to GIEs (Primary/Seconday)* |
|---|---|---|---|---|
| eps | Epsilon values for OpenCV `grouprectangles()` function and DBSCAN algorithm | Float, >=0 | eps=0.2 | Object detector *Both* |
| group-threshold | Threshold value for rectangle merging for OpenCV `grouprectangles()` function | Integer, >=0 | group-threshold=1 *0 disables the clustering functionality* | Object detector *Both* |
| minBoxes | Minimum number of points required to form a dense region for DBSCAN algorithm | Integer, ≥0 | minBoxes=1 *0 disables the clustering functionality* | Object detector *Both* |
| roi-top-offset | Offset of the RoI from the top of the frame. Only objects within the RoI are output. | Integer, ≥0 | roi-top-offset=200 | Object detector *Both* |
| roi-bottom-offset | Offset of the RoI from the bottom of the frame. Only objects within the RoI are output. | Integer, ≥0 | roi-bottom-offset=200 | Object detector *Both* |
| detected-min-w | Minimum width in pixels of detected objects to be output by the GIE | Integer, ≥0 | detected-min-w=64 | Object detector *Both* |
| detected-min-h | Minimum height in pixels of detected objects to be output by the GIE | Integer, ≥0 | detected-min-h=64 | Object detector *Both* |
| detected-max-w | Maximum width in pixels of detected objects to be output by the GIE | Integer, ≥0 | detected-max-w=200 *0 disables the property* | Object detector *Both* |
| detected-max-h | Maximum height in pixels of detected objects to be output by the GIE | Integer, ≥0 | detected-max-h=200 *0 disables the property* | Object detector *Both* |

### 2.1.4 Gst Properties

The values set through Gst properties override the values of properties in the configuration file. The application does this for certain properties that it needs to set programmatically.

Table 4 describes the `Gst-nvinfer` plugin's Gst properties.

### Table 4. Gst-nvinfer plugin, Gst properties

| Property | Meaning | Type and Range | Example Notes |
|---|---|---|---|
| config-file-path | Absolute pathname of configuration file for the Gst-nvinfer element | String | config-file-path= config_infer_primary.txt |
| process-mode | Infer Processing Mode<br>1=Primary Mode<br>2=Secondary Mode | Integer, 1 or 2 | process-mode=1 |
| unique-id | Unique ID identifying metadata generated by this GIE | Integer,<br>0 to 4,294,967,295 | unique-id=1 |
| infer-on-gie-id | See operate-on-gie-id in the configuration file table | Integer,<br>0 to 4,294,967,295 | infer-on-gie-id=1 |
| infer-on-class-ids | See operate-on-class-ids in the configuration file table | An array of colon-separated integers (class-ids) | infer-on-class-ids=1:2:4 |
| model-engine-file | Absolute pathname of the pre-generated serialized engine file for the mode | String | model-engine-file= model_b1_fp32.engine |
| batch-size | Number of frames/objects to be inferred together in a batch | Integer,<br>1 – 4,294,967,295 | batch-size=4 |
| Interval | Number of consecutive batches to be skipped for inference | Integer, 0 to 32 | interval=0 |
| gpu-id | Device ID of GPU to use for pre-processing/inference (dGPU only) | Integer,<br>0-4,294,967,295 | gpu-id=1 |
| raw-output-file-write | Pathname of raw inference output file | Boolean | raw-output-file-write=1 |
| raw-output-generated-callback | Pointer to the raw output generated callback function | Pointer | *Cannot be set through* gst-launch |
| raw-output-generated-userdata | Pointer to user data to be supplied with `raw-output-generated-callback` | Pointer | *Cannot be set through* gst-launch |
| output-tensor-meta | Indicates whether to attach tensor outputs as meta on `GstBuffer`. | Boolean | output-tensor-meta=0 |

## 2.1.5 Tensor Metadata

The Gst-nvinfer plugin can attach raw output tensor data generated by a TensorRT inference engine as metadata. It is added as an `NvDsInferTensorMeta` in the *frame_user_meta_list* member of `NvDsFrameMeta` for primary (full-frame) mode, or in the *obj_user_meta_list* member of `NvDsObjectMeta` for secondary (object) mode.

**To read or parse inference raw tensor data of output layers**

1. Enable property `output-tensor-meta`, or enable the same-named attribute in the configuration file for the Gst-nvinfer plugin.
2. When operating as primary GIE, `NvDsInferTensorMeta` is attached to each frame's (each `NvDsFrameMeta` object's) *frame_user_meta_list*. When operating as secondary GIE, `NvDsInferTensorMeta` is attached to each each NvDsObjectMeta object's *obj_user_meta_list*.

   Metadata attached by Gst-nvinfer can be accessed in a GStreamer pad probe attached downstream from the Gst-nvinfer instance.
3. The `NvDsInferTensorMeta` object's metadata type is set to `NVDSINFER_TENSOR_OUTPUT_META`. To get this metadata you must iterate over the `NvDsUserMeta` user metadata objects in the list referenced by *frame_user_meta_list* or *obj_user_meta_list*.

For more information about Gst-infer tensor metadata usage, see the source code in `sources/apps/sample_apps/deepstream_infer_tensor_meta-test.cpp`, provided in the DeepStream SDK samples.

### 2.1.6 Segmentation Metadata

The Gst-nvinfer plugin attaches the output of the segmentation model as user meta in an instance of `NvDsInferSegmentationMeta` with `meta_type` set to `NVDSINFER_SEGMENTATION_META`. The user meta is added to the `frame_user_meta_list` member of `NvDsFrameMeta` for primary (full-frame) mode, or the `obj_user_meta_list` member of `NvDsObjectMeta` for secondary (object) mode.

For guidance on how to access user metadata, see User/Custom Metadata Addition inside NvDsBatchMeta, and Tensor Metadata, above.

## 2.2 GST-NVTRACKER

This plugin tracks detected objects and gives each new object a unique ID.

The plugin adapts a low-level tracker library to the pipeline. It supports any low-level library that implements the low-level API, including the three reference implementations, the NvDCF, KLT, and IOU trackers.

As part of this API, the plugin queries the low-level library for capabilities and requirements concerning input format and memory type. It then converts input buffers into the format requested by the low-level library. For example, the KLT tracker uses Luma-only format; NvDCF uses NV12 or RGBA; and IOU requires no buffer at all.

The low-level capabilities also include support for batch processing across multiple input streams. Batch processing is typically more efficient than processing each stream independently. If a low-level library supports batch processing, that is the preferred mode of operation. However, this preference can be overridden with the `enable-batch-process` configuration option if the low-level library supports both batch and per-stream modes.

The plugin accepts NV12/RGBA data from the upstream component and scales (converts) the input buffer to a buffer in the format required by the low-level library, with tracker width and height. (Tracker width and height must be specified in the configuration file's `[tracker]` section.)

The low -level tracker library is selected via the `ll-lib-file` configuration option in the tracker configuration section. The selected low-level library may also require its own configuration file, wich can be specified via the `ll-config-file` option.

The three reference low level libraries support different algorithms:

- The KLT tracker uses a CPU-based implementation of the Kanade Lucas Tomasi (KLT) tracker algorithm. This library requires no configuration file.
- The Intersection of Union (IOU) tracker uses the intersection of the detector's bounding boxes across frames to determine the object's unique ID. This library takes an optional configuration file.
- The Nv-adapted Discriminative Correlation Filter (NvDCF) tracker uses a correlation filter-based online discriminative learning algorithm, coupled with a Hungarian

algorithm for data association in multi-object tracking. This library accepts an optional configuration file.

`

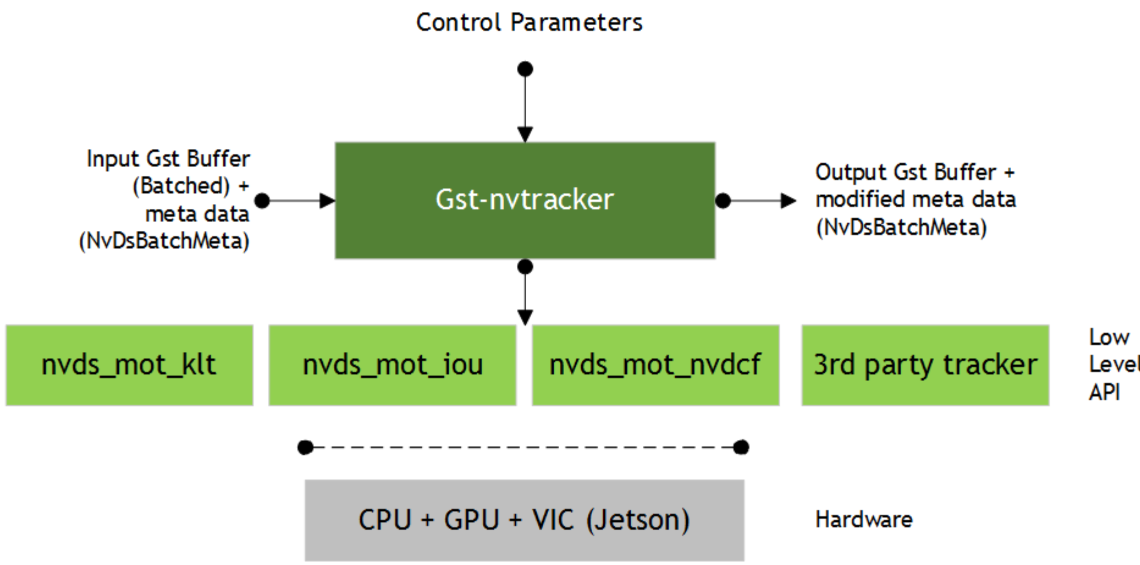


Figure 2. Gst-nvtracker inputs and outputs

### 2.2.1 Inputs and Outputs

This section summarizes the inputs, outputs, and communication facilities of the `Gst-nvtracker` plugin.

- Inputs
  - Gst Buffer (batched)
  - `NvDsBatchMeta`

  Formats supported are NV12 and RGBA.
- Control parameters
  - `tracker-width`
  - `tracker-height`
  - `gpu-id` (for dGPU only)
  - `ll-lib-file`
  - `ll-config-file`
  - `enable-batch-process`
- Output
  - Gst Buffer (provided as an input)
  - `NvDsBatchMeta` (Updated by `Gst-nvtracker`with tracked object coordinates and object IDs)

### 2.2.2 Features

Table 5 summarizes the features of the plugin.

Table 5. Features of the Gst-nvtracker plugin

| Feature | Description | Release |
|---|---|---|
| Configurable tracker width/height | Frames are internally scaled to specified resolution for tracking | DS2.0 |
| Multi-stream CPU/GPU tracker | Supports tracking on batched buffers consisting of frames from different sources | DS2.0 |
| NV12 Input | — | DS2.0 |
| RGBA Input | — | DS 3.0 |
| Allows low FPS tracking | IOU tracker | DS 3.0 |
| Configurable GPU device | User can select GPU for internal scaling/color format conversions and tracking | DS2.0 |
| Dynamic addition/deletion of sources at runtime | Supports tracking on new sources added at runtime and cleanup of resources when sources are removed | DS 3.0 |
| Support for user's choice of low-level library | Dynamically loads user selected low-level library | DS 4.0 |
| Support for batch processing | Supports sending frames from multiple input streams to the low-level library as a batch if the low-level library advertises capability to handle that | DS 4.0 |
| Support for multiple buffer formats as input to low-level library | Converts input buffer to formats requested by the low-level library, for up to 4 formats per frame | DS 4.0 |

### 2.2.3 Gst Properties

Table 6 describes the Gst properties of the `Gst-nvtracker` plugin.

Table 6. Gst-nvtracker plugin, Gst Properties

| Property | Meaning | Type and Range | Example *Notes* |
|---|---|---|---|
| tracker-width | Frame width at which the tracker is to operate, in pixels. | Integer, 0 to 4,294,967,295 | tracker-width=640 |
| tracker-height | Frame height at which the tracker is to operate, in pixels. | Integer, 0 to 4,294,967,295 | tracker-height=368 |
| ll-lib-file | Pathname of the low-level tracker library to be loaded by `Gst-nvtracker`. | String | ll-lib-file=/opt/nvidia/deepstream/libnvds_nvdcf.so |

| Property | Meaning | Type and Range | Example *Notes* |
|---|---|---|---|
| ll-config-file | Configuration file for the low-level library if needed. | Path to configuration file | ll-config-file=/opt/nvidia/deepstream/tracker_config.yml |
| gpu-id | ID of the GPU on which device/unified memory is to be allocated, and with which buffer copy/scaling is to be done. (dGPU only.) | Integer, 0 to 4,294,967,295 | gpu-id=1 |
| enable-batch-process | Enables/disables batch processing mode. Only effective if the low-level library supports both batch and per-stream processing. (Optional.) | Boolean | enable-batch-process=1 |

## 2.2.4 Custom Low-Level Library

To write a custom low-level tracker library, implement the API defined in `sources/includes/nvdstracker.h`. Parts of the API refer to `sources/includes/nvbufsurface.h`.

The names of API functions and data structures are prefixed with `NvMOT`, which stands for *NVIDIA Multi-Object Tracker*.

This is the general flow of the API from a low-level library perspective:

1. The first required function is:

```
NvMOTStatus NvMOT_Query(
    uint16_t customConfigFilePathSize,
    char* pCustomConfigFilePath,
    NvMOTQuery *pQuery
);
```

   The plugin uses this function to query the low-level library's capabilities and requirements before it starts any processing sessions (contexts) with the library. Queried properties include the input frame memory format (e.g., RGBA or NV12), memory type (e.g., NVIDIA® CUDA® device or CPU mapped NVMM), and support for batch processing.

   The plugin performs this query once, and its results apply to all contexts established with the low-level library. If a low-level library configuration file is specified, it is provided in the query for the library to consult.

   The query reply structure `NvMOTQuery` contains the following fields:

- `NvMOTCompute computeConfig`: Reports compute targets supported by the library. The plugin currently only echoes the reported value when initiating a context.
- `uint8_t numTransforms`: The number of color formats required by the low-level library. The valid range for this field is 0 to `NVMOT_MAX_TRANSFORMS`. Set this to 0 if the library does not require any visual data. Note that 0 does not mean that untransformed data will be passed to the library.
- `NvBufSurfaceColorFormat colorFormats[NVMOT_MAX_TRANSFORMS]`: The list of color formats required by the low-level library. Only the first `numTransforms` entries are valid.
- `NvBufSurfaceMemType memType`: Memory type for the transform buffers. The plugin allocates buffers of this type to store color and scale-converted frames, and the buffers are passed to the low-level library for each frame. Note that support is currently limited to the following types:

  dGPU: `NVBUF_MEM_CUDA_PINNED`
  `NVBUF_MEM_CUDA_UNIFIED`

  Jetson: `NVBUF_MEM_SURFACE_ARRAY`
- `bool supportBatchProcessing`: True if the low-library support batch processing across multiple streams; otherwise false.

2. After the query, and before any frames arrive, the plugin must initialize a context with the low-level library by calling:

```
NvMOTStatus NvMOT_Init(
    NvMOTConfig *pConfigIn,
    NvMOTContextHandle *pContextHandle,
    NvMOTConfigResponse *pConfigResponse
);
```

   The context handle is opaque outside the low-level library. In batch processing mode, the plugin requests a single context for all input streams. In per-stream processing mode, the plugin makes this call for each input stream so that each stream has its own context.

   This call includes a configuration request for the context. The low-level library has an opportunity to:

- Review the configuration, and create a context only if the request is accepted. If any part of the configuration request is rejected, no context is created, and the return status must be set to `NvMOTStatus_Error`. The `pConfigResponse` field can optionally contain status for specific configuration items.
- Pre-allocate resources based on the configuration.

| Note: | |
|---|---|
| | • **In the `NvMOTMiscConfig` structure, the `logMsg` field is currently unsupported and uninitialized.**<br>• **The `customConfigFilePath` pointer is only valid during the call.** |

3. Once a context is initialized, the plugin sends frame data along with detected object bounding boxes to the low-level library each time it receives such data from upstream. It always presents the data as a batch of frames, although the batch contains only a single frame in per-stream processing contexts. Each batch is guaranteed to contain at most one frame from each stream.

   The function call for this processing is:

```
NvMOTStatus NvMOT_Process(NvMOTContextHandle contextHandle,
    NvMOTProcessParams *pParams,
    NvMOTTrackedObjBatch *pTrackedObjectsBatch
);
```

   Where:

   - `pParams` is a pointer to the input batch of frames to process. The structure contains a list of one or more frames, with at most one frame from each stream. No two frame entries have the same `streamID`. Each entry of frame data contains a list of one or more buffers in the color formats required by the low-level library, as well as a list of object descriptors for the frame. Most libraries require at most one color format.
   - `pTrackedObjectsBatch` is a pointer to the output batch of object descriptors. It is pre-populated with a value for `numFilled`, the number of frames included in the input parameters.

     If a frame has no output object descriptors, it is still counted in `numFilled` and is represented with an empty list entry (`NvMOTTrackedObjList`). An empty list entry has the correct `streamID` set and `numFilled` set to 0.

| Note: | |
|---|---|
| | **The output object descriptor `NvMOTTrackedObj` contains a pointer to the associated input object, `associatedObjectIn`. You must set this to the associated input object only for the frame where the input object is passed in. For example:**<br>• **Frame 0: NvMOTObjToTrack X is passed in. The tracker assigns it ID 1, and the output object associatedObjectIn points to X.**<br>• **Frame 1: Inference is skipped, so there is no input object. The tracker finds object 1, and the output object `associatedObjectIn` points to NULL.**<br>• **Frame 2: `NvMOTObjToTrack` Y is passed in. The tracker identifies it as object 1. The output object 1 has `associatedObjectIn` pointing to Y.** |

4. When all processing is complete, the plugin calls this function to clean up the context:

```
void NvMOT_DeInit(NvMOTContextHandle contextHandle);
```

### 2.2.5 Low-Level Tracker Library Comparisons and Tradeoffs

DeepStream 4.0 provides three low-level tracker libraries which have different resource requirements and performance characteristics, in terms of accuracy, robustness, and efficiency, allowing you to choose the best tracker based on you use case. See the following table for comparison.

Table 7. Tracker library comparison

| Tracker | Computational Load | | Pros | Cons | Best Use Cases |
|---|---|---|---|---|---|
| | GPU | CPU | | | |
| IOU | X | Very Low | Light-weight | No visual features for matching, so prone to frequent tracker ID switches and failures.<br>Not suitable for fast moving scene. | Objects are sparsely located, with distinct sizes.<br>Detector is expected to run every frame or very frequently (ex. every alternate frame). |
| KLT | X | High | Works reasonably well for simple scenes | High CPU utilization.<br>Susceptible to change in the visual appearance due to noise and perturbations, such as shadow, non-rigid deformation, out-of-plane rotation, and partial occlusion.<br>Cannot work on objects with low textures. | Objects with strong textures and simpler background.<br>Ideal for high CPU resource availability. |
| NvDCF | Medium | Low | Highly robust against partial occlusions, shadow, and other transient visual changes.<br><br>Less frequent ID switches. | Slower than KLT and IOU due to increased computational complexity.<br>Reduces the total number of streams processed. | Multi-object, complex scenes with partial occlusion. |

### 2.2.6 NvDCF Low-Level Tracker

NvDCF is a reference implementation of the custom low-level tracker library that supports multi-stream, multi-object tracking in a batch mode using a discriminative correlation filter (DCF) based approach for visual object tracking and a Hungarian algorithm for data association.

NvDCF preallocates memory during initialization based on:

- The number of streams to be processed
- The maximum number of objects to be tracked per stream (denoted as `maxTargetsPerStream` in a configuration file for the NvDCF low-level library, `tracker_config.yml`)

Once the number of objects being tracked reaches the configured maximum value, any new objects will be discarded until resources for some existing tracked objects are released. Note that the number of objects being tracked includes objects that are tracked in Shadow Mode (described below). Therefore, NVIDIA recommends that you make `maxTargetsPerStream` large enough to accommodate the maximum number of objects of interest that may appear in a frame, as well as the past objects that may be tracked in shadow mode. Also, note that GPU memory usage by NvDCF is linearly proportional to the total number of objects being tracked, which is (number of video streams) × (`maxTargetsPerStream`).

DCF-based trackers typically apply an exponential moving average for temporal consistency when the optimal correlation filter is created and updated. The learning rate for this moving average can be configured as filterLr. The standard deviation for Gaussian for desired response when creating an optimal DCF filter can also be configured as `gaussianSigma`.

DCF-based trackers also define a search region around the detected target location large enough for the same target to be detected in the search region in the next frame. The `SearchRegionPaddingScale` property determines the size of the search region as a multiple of the target's bounding box size. For example, with `SearchRegionPaddingScale: 3`, the size of the search region would be:

$$searchregionwidth = w + 3 * (w * h)^{1/2}$$

$$searchregionheight = h + 3 * (w * h)^{1/2}$$

Where $w$ and $h$ are the width and height of the target's bounding box.

Once the search region is defined for each target, the image patches for the search regions are cropped and scaled to a predefined feature image size, then the visual features are extracted. The `featureImgSizeLevel` property defines the size of the feature image. A lower value of `featureImgSizeLevel` causes NvDCF to use a smaller feature size, increasing GPU performance at the cost of accuracy and robustness.

Consider the relationship between `featureImgSizeLevel` and `SearchRegionPaddingScale` when configuring the parameters. If `SearchRegionPaddingScale` is increased while `featureImgSizeLevel` is fixed, the number of pixels corresponding to the target in the feature images is effectively decreased.

The `minDetectorConfidence` property sets confidence level below which object detection results are filtered out.

To achieve robust tracking, NvDCF employs two approaches to handling false alarms from PGIE detectors: **late activation** for handling false positives and **shadow tracking** for false negatives. Whenever a new object is detected a new tracker is instantiated in temporary mode. It must be activated to be considered as a valid target. Before it is activated it undergoes a probationary period, defined by `probationAge`, in temporary mode. If the object is not detected in more than `earlyTerminationAge` consecutive frames during the period, the tracker is terminated early.

Once the tracker for an object is activated, it is put into inactive mode only when (1) no matching detector input is found during the data association, or (2) the tracker confidence falls below a threshold defined by `minTrackerConfidence`. The per-object tracker will be put into active mode again if a matching detector input is found. The length of period during which a per-object tracker is in inactive mode is called the **shadow tracking age**; if it reaches the threshold defined by `maxShadowTrackingAge`, the tracker is terminated. If the bounding box of an object being tracked goes partially out of the image frame and so its visibility falls below a predefined threshold defined by `minVisibiilty4Tracking`, the tracker is also terminated.

Note that `probationAge` is counted against a clock that is incremented at every frame, while `maxShadowTrackingAge` and `earlyTerminationAge` are counted against a clock incremented only when the shadow tracking age is incremented. When the PGIE detector runs on every frame (i.e., `interval=0` in the `[primary-gie]` section of the `deepstream-app` configuration file), for example, all the ages are incremented based on the same clock. If the PGIE detector runs on every other frame (i.e., `interval` is set to 1 in `[primary-gie]`) and `probationAge: 12`, it will yield almost the same results as `interval=0` with `probationAge: 24`, because `shadowTrackingAge` would be incremented at a half speed compared to the case with PGIE `interval=0`.

Table 8 summaries the configuration parameters for an NvDCF low-level tracker.

### Table 8. NvDCF low-level tracker, configuration properties

| Property | Meaning | Type and Range | Example *Notes* |
|---|---|---|---|
| maxTargetsPerStream | Max number of targets to track per stream | Integer, 0 to 65535 | maxTargetsPerStream: 30 |

| Property | Meaning | Type and Range | Example *Notes* |
|---|---|---|---|
| filterLr | Learning rate for DCF filter in exponential moving average | Float, 0.0 to 1.0 | filterLr: 0.11 |
| gaussianSigma | Standard deviation for Gaussian for desired response | Float, >0.0 | gaussianSigma: 0.75 |
| minDetectorConfidence | Minimum detector confidence for a valid object | Float, -inf to inf | minDetectorConfidence: 0.0 |
| minTrackerConfidence | Minimum detector confidence for a valid target | Float, 0.0 to 1.0 | minTrackerConfidence: 0.6 |
| featureImgSizeLevel | Size of a feature image | Integer, 1 to 5 | featureImgSizeLevel: 1 |
| SearchRegionPaddingScale | Search region size | Integer, 1 to 3 | SearchRegionPaddingScale: 3 |
| maxShadowTrackingAge | Maximum length of shadow tracking | Integer, ≥0 | maxShadowTrackingAge: 9 |
| probationAge | Length of probationary period | Integer, ≥0 | probationAge: 12 |
| earlyTerminationAge | Early termination age | Integer, ≥0 | earlyTerminationAge: 2 |
| minVisibiilty4Tracking | Minimum visibility of target bounding box to be considered valid | Float, 0.0 to 1.0 | minVisibiilty4Tracking: 0.1 |

## 2.3 GST-NVSTREAMMUX

The `Gst-nvstreammux` plugin forms a batch of frames from multiple input sources. When connecting a source to `nvstreammux` (the muxer), a new pad must be requested from the muxer using `gst_element_get_request_pad()` and the pad template `"sink_%u"`. For more information, see `link_element_to_streammux_sink_pad()` in the DeepStream app source code.

The muxer forms a batched buffer of `batch-size` frames. (`batch-size` is specified using the `gst object` property.)

If the muxer's output format and input format are the same, the muxer forwards the frames from that source as a part of the muxer's output batched buffer. The frames are returned to the source when muxer gets back its output buffer. If the resolution is not the same, the muxer scales frames from the input into the batched buffer and then returns the input buffers to the upstream component.

The muxer pushes the batch downstream when the batch is filled or the batch formation timeout `batched-pushed-timeout` is reached. The timeout starts running when the first buffer for a new batch is collected.

The muxer uses a round-robin algorithm to collect frames from the sources. It tries to collect an average of ( `batch-size/num-source` ) frames per batch from each source (if all sources are live and their frame rates are all the same). The number varies for each source, though, depending on the sources' frame rates.

The muxer outputs a single resolution (i.e. all frames in the batch have the same resolution). This resolution can be specified using the `width` and `height` properties. The muxer scales all input frames to this resolution. The `enable-padding` property can be set to `true` to preserve the input aspect ratio while scaling by padding with black bands.

For DGPU platforms, the GPU to use for scaling and memory allocations can be specified with the `gpu-id` property.

For each source that needs scaling to the muxer's output resolution, the muxer creates a buffer pool and allocates four buffers each of size:

`output-width * output-height *` *f*

Where *f* is 1.5 for NV12 format, or 4.0 for RGBA. The memory type is determined by the `nvbuf-memory-type` property.

Set the `live-source` property to `true` to inform the muxer that the sources are live. In this case the muxer attaches the PTS of the last copied input buffer to the batched Gst Buffer's PTS. If the property is set to `false`, the muxer calculates timestamps based on the frame rate of the source which first negotiates capabilities with the muxer.

The muxer attaches an `NvDsBatchMeta` metadata structure to the output batched buffer. This meta contains information about the frames copied into the batch (e.g. source ID of the frame, original resolutions of the input frames, original buffer PTS of the input frames). The source connected to the `Sink_N` pad will have `pad_index` N in `NvDsBatchMeta`.

The muxer supports addition and deletion of sources at run time. When the muxer receives a buffer from a new source, it sends a `GST_NVEVENT_PAD_ADDED` event. When a muxer sink pad is removed, the muxer sends a `GST_NVEVENT_PAD_DELETED` event. Both events contains the source ID of the source being added or removed (see `sources/includes/gst-nvevent.h`). Downstream elements can reconfigure when they receive these events. Additionally, the muxer also sends a `GST_NVEVENT_STREAM_EOS` to indicate EOS from the source.

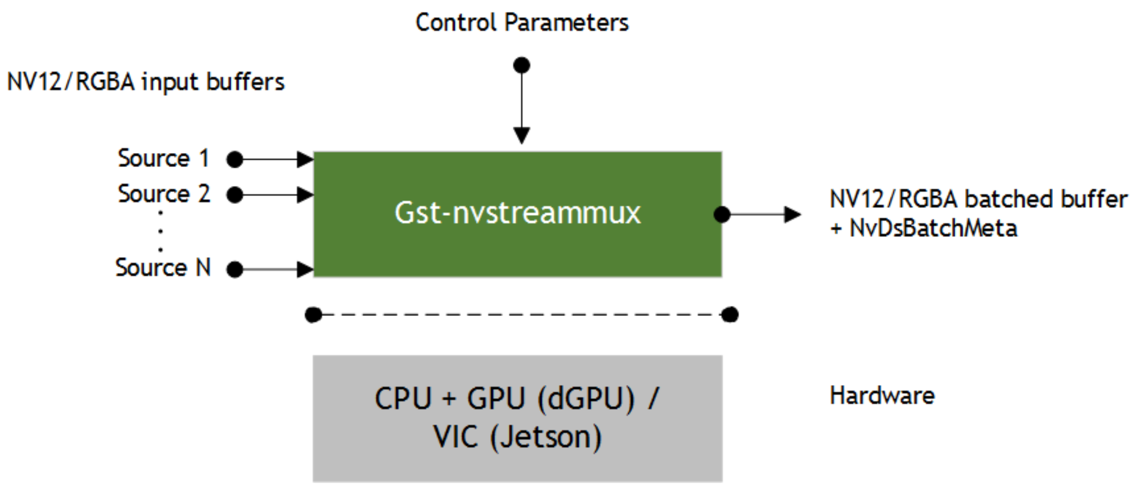


Figure 3. The Gst-nvstreammux plugin

## 2.3.1 Inputs and Outputs

- Inputs
  - NV12/RGBA buffers from an arbitrary number of sources
- Control Parameters
  - `batch-size`
  - `batched-push-timeout`
  - `width`
  - `height`
  - `enable-padding`
  - `gpu-id` (dGPU only)
  - `live-source`
  - `nvbuf-memory-type`
- Output
  - NV12/RGBA batched buffer
  - `GstNvBatchMeta` (meta containing information about individual frames in the batched buffer)

## 2.3.2 Features

Table 9 summarizes the features of the plugin.

Table 9. Features of the Gst-nvstreammux plugin

| Feature | Description | Release |
|---|---|---|
| Configurable batch size | — | DS 2.0 |
| Configurable batching timeout | — | DS 2.0 |
| Allows multiple input streams with different resolutions | — | DS 2.0 |
| Allows multiple input streams with different frame rates | — | DS 2.0 |
| Scales to user-determined resolution in muxer | — | DS 2.0 |
| Scales while maintaining aspect ratio with padding | — | DS 2.0 |
| Multi-GPU support | — | DS 2.0 |
| Input stream DRC support | — | DS 3.0 |
| User-configurable CUDA memory type (Pinned/Device/Unified) for output buffers | — | DS 3.0 |
| Custom message to inform application of EOS from individual sources | — | DS 3.0 |
| Supports adding and deleting run time sinkpads (input sources) and sending custom events to notify downstream components | — | DS 3.0 |
| Supports RGBA data handling at output | — | DS 3.0 |

### 2.3.3 Gst Properties

Table 10 describes the Gst-nvstreammux plugin's Gst properties.

Table 10. Gst-nvstreammux plugin, Gst properties

| Property | Meaning | Type and Range | Example *Notes* |
|---|---|---|---|
| batch-size | Maximum number of frames in a batch. | Integer, 0 to 4,294,967,295 | batch-size=30 |
| batched-push-timeout | Timeout in microseconds to wait after the first buffer is available to push the batch even if a complete batch is not formed. | Signed integer, -1 to 2,147,483,647 | batched-push-timeout=40000 *40 msec* |
| width | If non-zero, muxer scales input frames to this width. | Integer, 0 to 4,294,967,295 | width=1280 |
| height | If non-zero, muxer scales input frames to this height. | Integer, 0 to 4,294,967,295 | height=720 |

| Property | Meaning | Type and Range | Example<br>*Notes* |
|---|---|---|---|
| enable-padding | Maintains aspect ratio by padding with black borders when scaling input frames. | Boolean | enable-padding=1 |
| gpu-id | ID of the GPU on which to allocate device or unified memory to be used for copying or scaling buffers. (dGPU only.) | Integer,<br>0 to 4,294,967,295 | gpu-id=1 |
| live-source | Indicates to muxer that sources are live, e.g. live feeds like an RTSP or USB camera. | Boolean | live-source=1 |
| nvbuf-memory-type | Type of memory to be allocated.<br>**For dGPU**:<br>0 (`nvbuf-mem-default`): Default memory, `cuda-device`<br>1 (`nvbuf-mem-cuda-pinned`): Pinned/Host CUDA memory<br>2 (`nvbuf-mem-cuda-device`) Device CUDA memory<br>3 (`nvbuf-mem-cuda-unified`): Unified CUDA memory<br>**For Jetson**:<br>0 (`nvbuf-mem-default`): Default memory, surface array<br>4 (`nvbuf-mem-surface-array`): Surface array memory | Integer, 0-4 | nvbuf-memory-type=1 |

## 2.4 GST-NVSTREAMDEMUX

The `Gst-nvstreamdemux` plugin demuxes batched frames into individual buffers. It creates a separate Gst Buffer for each frame in the batch. It does not copy the video frames. Each Gst Buffer contains a pointer to the corresponding frame in the batch.

The plugin pushes the unbatched Gst Buffer objects downstream on the pad corresponding to each frame's source. The plugin gets this information through the `NvDsBatchMeta` attached by `Gst-nvstreammux`. The original buffer timestamps (PTS) of individual frames are also attached back to the Gst Buffer.

Since there is no frame copy, the input Gst Buffer is not returned upstream immediately. When all of the non-batched Gst Buffer objects demuxed from an input batched Gst Buffer are returned to the demuxer by the downstream component, the input batched Gst Buffer is returned upstream.

The demuxer does not scale the buffer back to the source's original resolution even if Gst-nvstreammux has scaled the buffers.

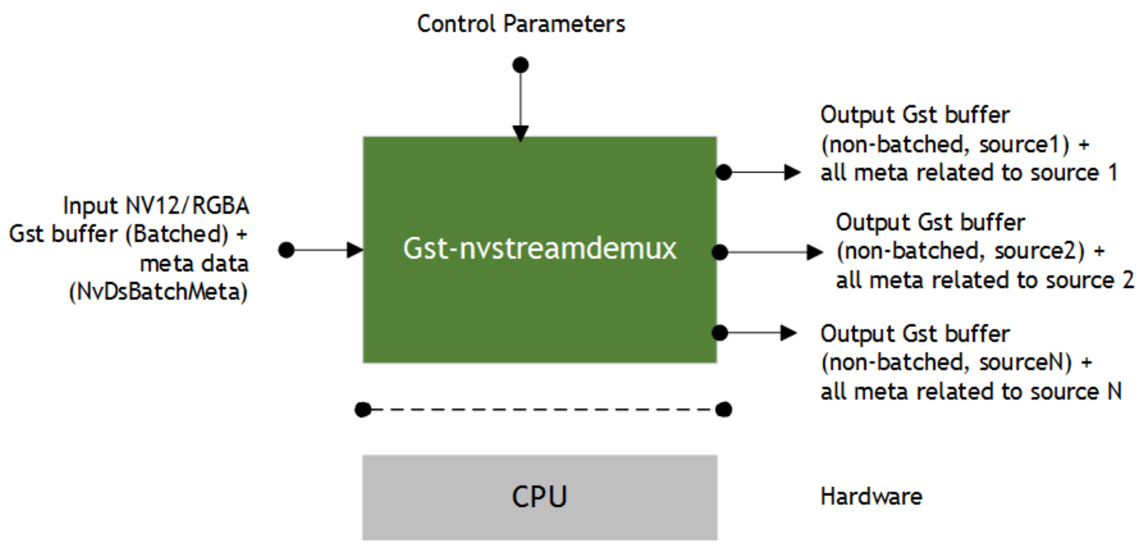


Figure 4. The Gst-nvstreamdemux plugin

## 2.4.1 Inputs and Outputs

- Inputs
  - Gst Buffer (batched)
  - `NvDsBatchMeta`
  - Other meta
- Control parameters
  - *None*
- Output
  - Gst Buffer (non-batched, single source)
  - Meta related to each Gst Buffer source

## 2.5 GST-NVMULTISTREAMTILER

The `Gst-nvmultistreamtiler` plugin composites a 2D tile from batched buffers. The plugin accepts batched NV12/RGBA data from upstream components. The plugin composites the tile based on stream IDs, obtained from `NvDsBatchMeta` and `NvDsFrameMeta` in row-major order (starting from source 0, left to right across the top row, then across the next row). Each source frame is scaled to the corresponding location in the tiled output buffer. The plugin can reconfigure if a new source is added and it exceeds the space allocated for tiles. It also maintains a cache of old frames to avoid display flicker if one source has a lower frame rate than other sources.

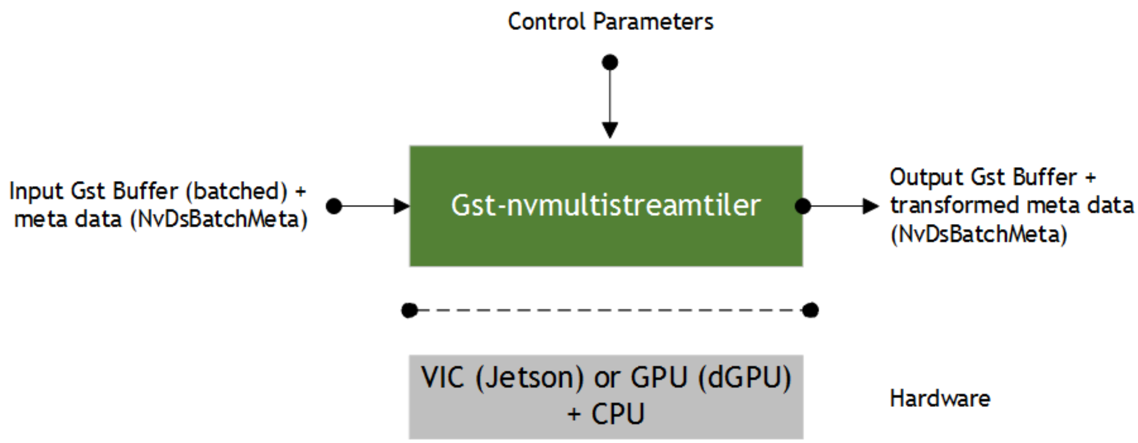


Figure 5. The Gst-nvmultistreamtiler plugin

### 2.5.1 Inputs and Outputs

- Inputs
  - Gst Buffer batched buffer
  - `NvDsBatchMeta` with Gst Buffer batched (batch is one or more buffers)

  Formats supported: NV12/RGBA
- Control Parameters
  - `rows`
  - `columns`
  - `width`
  - `height`
  - `gpu-id` (dGPU only)
  - `show-source`
  - `nvbuf-memory-type`
  - `custom-tile-config`

- Output
  - Gst Buffer (single frame) with composited input frames
  - Transformed metadata (`NvDsBatchMeta`)

  Formats supported: NV12/RGBA

## 2.5.2 Features

Table 11 summarizes the features of the plugin.

### Table 11. Features of the Gst-nvmultistreamtiler plugin

| Feature | Description | Release |
|---|---|---|
| Composites a 2D tile of input buffers | — | DS 2.0 |
| Scales bounding box with metadata coordinates according to scaling and position in tile | — | DS 2.0 |
| Multi-GPU support | — | DS 2.0 |
| Shows expanded preview for a single source | — | DS 2.0 |
| User configurable CUDA memory type (Pinned/Device/Unified) for output buffers | — | DS 3.0 |
| Reconfigures 2D tile for new sources added at runtime | — | DS 3.0 |

## 2.5.3 Gst Properties

Table 12 describes the `Gst-nvmultistreamtiler` plugin's Gst properties.

### Table 12. Gst-nvmultistreamtiler plugin, Gst properties

| Property | Meaning | Type and Range | Example *Notes* |
|---|---|---|---|
| Rows | Number of rows in 2D tiled output | Integer, 1 to 4,294,967,295 | row=2 |
| Columns | Number of columns in 2D tiled output | Integer, 1 to 4,294,967,295 | columns=2 |
| Width | Width of 2D tiled output in pixels | Integer, 16 to 4,294,967,295 | width=1920 |
| Height | Height of 2D tiled output in pixels | Integer, 16 to 4,294,967,295 | height=1080 |

| Property | Meaning | Type and Range | Example<br>*Notes* |
|---|---|---|---|
| show-source | Scale and show frames from a single source.<br>-1: composite and show all sources<br>For values ≥0, frames from that source are zoomed. | Signed integer, −1 to 2,147,483,647 | show-source=2 |
| gpu-id | ID of the GPU on which device/unified memory is to be allocated, and in which buffers are copied or scaled. (dGPU only.) | Integer,<br>0 to 4,294,967,295 | gpu-id=1 |
| nvbuf-memory-type | Type of CUDA memory to be allocated.<br>**For dGPU**:<br>0 (`nvbuf-mem-default`): Default memory, `cuda-device`<br>1 (`nvbuf-mem-cuda-pinned`): Pinned/Host CUDA memory<br>2 (`nvbuf-mem-cuda-device`) Device CUDA memory<br>3 (`nvbuf-mem-cuda-unified`): Unified CUDA memory<br>**For Jetson**:<br>0 (`nvbuf-mem-default`): Default memory, surface array<br>4 (`nvbuf-mem-surface-array`): Surface array memory | Integer, 0-4 | nvbuf-memory-type=1 |
| custom-tile-config | Custom tile position and resolution. Can be configured programmatically for all or none of the sources. | Values of enum `CustomTileConfig` | Reserved for future use.<br>Default: null. |

# 2.6 GST-NVDSOSD

This plugin draws bounding boxes, text, and region of interest (**RoI**) 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.

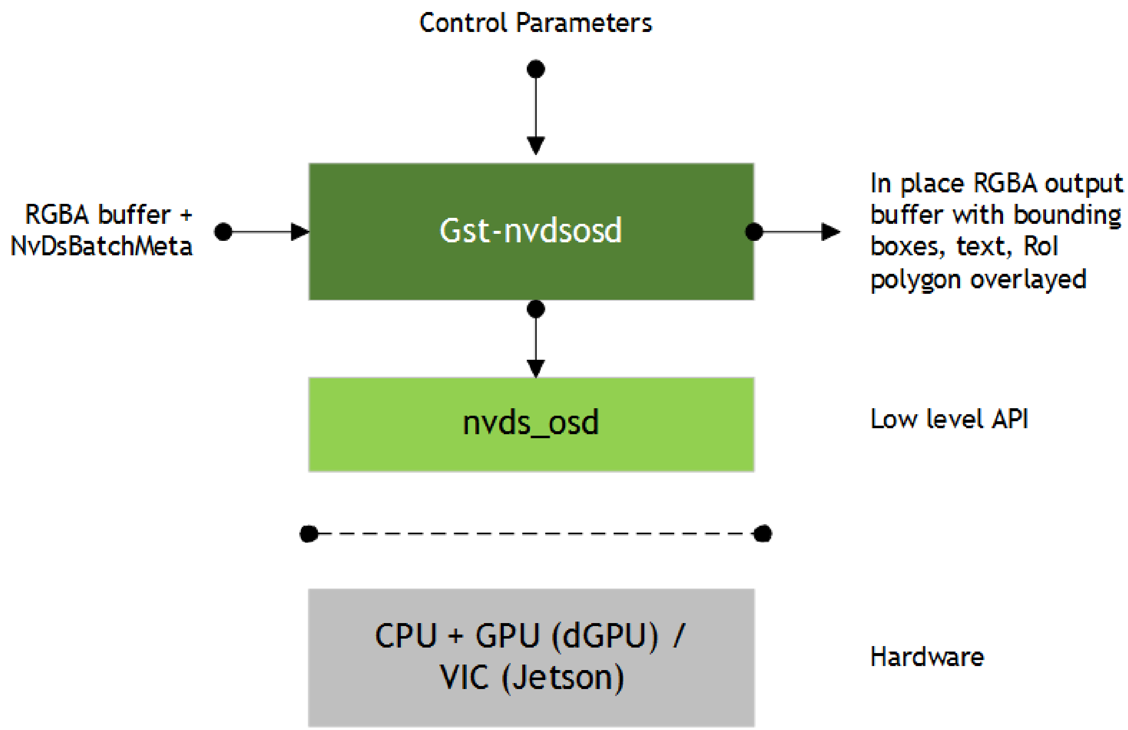


Figure 6. The Gst-nvdsosd plugin

## 2.6.1 Inputs and Outputs

- Inputs
  - RGBA buffer
  - `NvDsBatchMeta` (holds `NvDsFrameMeta` consisting of bounding boxes, text parameters, and lines parameters)
  - `NvDsLineMeta` (RoI polygon)
- Control parameters
  - `gpu-id` (dGPU only)
  - `display-clock`
  - `clock-font`
  - `clock-font-size`
  - `x-clock-offset`
  - `y-clock-offset`
  - `clock-color`

- Output
  - RGBA buffer modified in place to overlay bounding boxes, texts, and polygons represented in the metadata

## 2.6.2 Features

Table 13 summarizes the features of the plugin.

Table 13. Features of the Gst-nvdsosd plugin

| Feature | Description | Release |
|---|---|---|
| Supports drawing polygon lines | — | DS 3.0 |
| Supports drawing text using Pango and Cairo libraries | — | DS 2.0 |
| VIC (Jetson only) and GPU support for drawing bounding boxes | — | DS 2.0 |

## 2.6.3 Gst Properties

Table 14 describes the Gst properties of the `Gst-nvdsosd` plugin.

Table 14. Gst-nvdsosd plugin, Gst Properties

| Property | Meaning | Type and Range | Example *Notes* |
|---|---|---|---|
| gpu-id | Device ID of the GPU to be used for operation **(dGPU only)** | Integer, 0 to 4,294,967,295 | gpu-id=0 |
| display-clock | Indicates whether to display system clock | Boolean | display-clock=0 |
| clock-font | Name of Pango font to use for the clock | String | clock-font=Arial |
| clock-font-size | Font size to use for the clock | Integer, 0-60 | clock-font-size=2 |
| x-clock-offset | X offset of the clock | Integer, 0 to 4,294,967,295 | x-clock-offset=100 |
| y-clock-offset | Y offset of the clock | Integer, 0 to 4,294,967,295 | y-clock-offset=50 |
| clock-color | Color of the clock to be set while display, in the order 0xRGBA | Integer, 0 to 4,294,967,295 | clock-color=0xff0000ff *(Clock is red with alpha=1)* |

## 2.7 GST-NVVIDEOCONVERT

This plugin performs video color format conversion. It accepts NVMM memory as well as RAW (memory allocated using `calloc()` or `malloc()` ), and provides NVMM or RAW memory at the output.

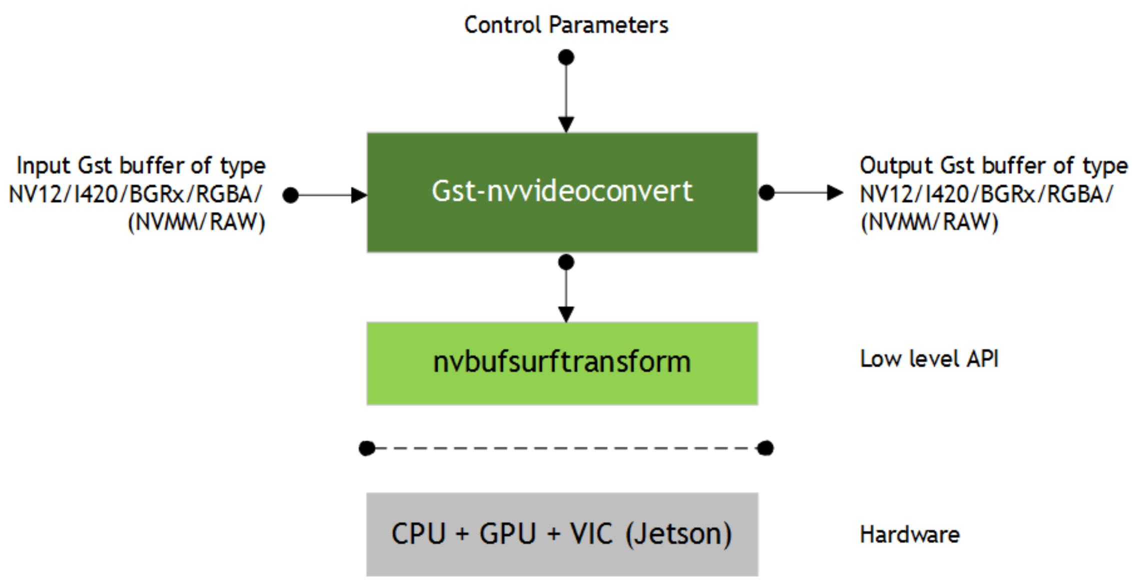


Figure 7. The Gst-nvvideoconvert plugin

### 2.7.1 Inputs and Outputs

- Inputs
  - Gst Buffer batched buffer
  - `NvDsBatchMeta`

  Format: NV12, I420, BGRx, RGBA (NVMM/RAW)
- Control parameters
  - `gpu-id` (dGPU only)
  - `nvbuf-memory-type`
  - `src-crop`
  - `dst-crop`
  - `interpolation-method`
  - `compute-hw`
- Output
  - Gst Buffer
  - `NvDsBatchMeta`
  - Format: NV12, I420, BGRx, RGBA (NVMM/RAW)

## 2.7.2 Features

This plugin supports batched scaling and conversion in single call for NVMM to NVMM, RAW to NVMM, and NVMM to RAW buffer types. It does not support RAW to RAW conversion. The plugin supports cropping of the input and output frames.

## 2.7.3 Gst Properties

Table 15 describes the Gst properties of the `Gst-nvvideoconvert` plugin.

Table 15. Gst-nvvideoconvert plugin, Gst Properties

| Property | Meaning | Type and Range | Example *Notes* |
|---|---|---|---|
| nvbuf-memory-type | Type of memory to be allocated.<br>**For dGPU**:<br>0 (`nvbuf-mem-default`): Default memory, `cuda-device`<br>1 (`nvbuf-mem-cuda-pinned`): Pinned/Host CUDA memory<br>2 (`nvbuf-mem-cuda-device`) Device CUDA memory<br>3 (`nvbuf-mem-cuda-unified`): Unified CUDA memory<br>**For Jetson**:<br>0 (`nvbuf-mem-default`): Default memory, surface array<br>4 (`nvbuf-mem-surface-array`): Surface array memory | enum GstNvVidConvBufMemoryType | |
| src-crop | Pixel location: left:top:width:height | String | 20; 40; 150; 100 |
| dst-crop | Pixel location: left:top:width:height | String | 20; 40; 150; 100 |

| Property | Meaning | Type and Range | Example *Notes* |
|---|---|---|---|
| interpolation-method | Interpolation method.<br>0: Nearest<br>1: Bilinear<br>2: Algo-1 (GPU—Cubic, VIC—5 Tap)<br>3: Algo-2 (GPU—Super, VIC—10 Tap)<br>4: Algo-3 (GPU—LanzoS, VIC—Smart)<br>5: Algo-4 (GPU—Ignored, VIC—Nicest)<br>6: Default (GPU—Nearest, VIC—Nearest) | enum GstInterpolationMethod | interpolation-method=1<br>*Default value is 6.* |
| compute-hw | Type of computing hardware<br>0: Default (GPU for gDPU, VIC for Jetson)<br>1: GPU<br>2: VIC | enum GstComputeHW | compute-hw=0<br>*Default value is 0.* |
| gpu-id | Device ID of GPU to use for format conversion | Integer,<br>0 to 4,294,967,295 | gpu-id=0 |
| output-buffers | Number of Output Buffers for the buffer pool | Unsigned integer,<br>1 to 4,294,967,295 | output-buffers=4 |

## 2.8 GST-NVDEWARPER

This plugin dewarps 360° camera input. It accepts `gpu-id` and `config-file` as properties. Based on the selected configuration of surfaces, it can generate a maximum of four dewarped surfaces. It currently supports dewarping of two projection types, `NVDS_META_SURFACE_FISH_PUSHBROOM` and `NVDS_META_SURFACE_FISH_VERTCYL`. Both of these are used in 360-D use case.

The plugin performs its function in these steps:

1. Reads the configuration file and creates a vector of surface configurations. It supports a maximum of four dewarp surface configurations.
2. Receives the 360-D frame from the decoder; based on the configuration, generates up to four dewarped surfaces.
3. Scales these surfaces down to network / selected dewarper output resolution using NPP APIs.
4. Pushes a buffer containing the dewarped surfaces to the downstream component.

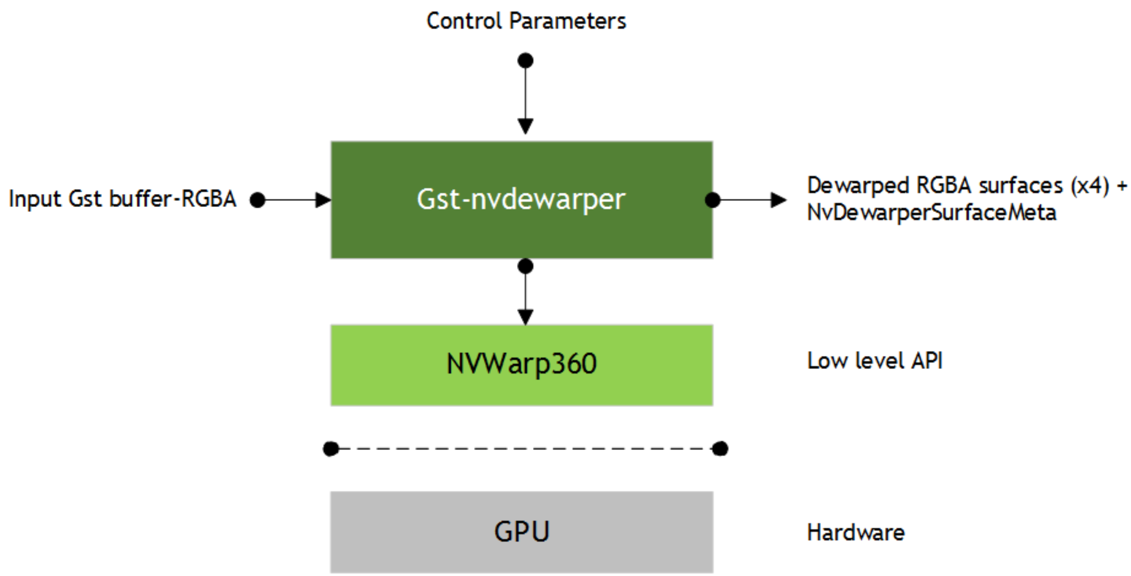


Figure 8. The Gst-nvdewarper plugin

### 2.8.1 Inputs and Outputs

- Inputs
  - A buffer containing a 360-D frame in RGBA format
- Control parameters
  - `gpu-id`; selects the GPU ID (dGPU only)
  - `config-file`, containing the pathname of the dewarper configuration file
- Output
  - Dewarped RGBA surfaces
  - `NvDewarperSurfaceMeta` with information associated with each surface (`projection_type`, `surface_index`, and `source_id`), and the number of valid dewarped surfaces in the buffer (`num_filled_surfaces`)

### 2.8.2 Features

Table 16 summarizes the features of the plugin.

Table 16. Features of the Gst-nvdewarper plugin

| Feature | Description | Release |
|---|---|---|
| Configure number of dewarped surfaces | Supports a maximum of four dewarper surfaces. | DS 3.0 |

| Feature | Description | Release |
|---|---|---|
| Configure per-surface projection type | Currently supports FishPushBroom and FishVertRadCyd projections. | DS 3.0 |
| Configure per-surface index | Surface index to be set in case of multiple surfaces having same projection type. | DS 3.0 |
| Configure per-surface width and height | | DS 3.0 |
| Configure per-surface dewarping parameters | Per-surface configurable yaw, roll, pitch, top angle, bottom angle, and focal length dewarping parameters. | DS 3.0 |
| Configurable dewarper output resolution | Creates a batch of up to four surfaces of a specified output resolution; internally scales all dewarper surfaces to output resolution. | DS 3.0 |
| Configurable NVDS CUDA memory type | — | DS 3.0 |
| Multi-GPU support | — | DS 3.0 |
| Aisle view CSV calibration file support | If set, properties in the [surface<n>] group are ignored. | DS 3.0 |
| Spot view CSV calibration file support | If set, properties in the [surface<n>] group are ignored. | DS 3.0 |
| Configure source id | Sets the source ID information in the NvDewarperSurfaceMeta. | DS 4.0 |
| Configurable number of output buffers | Number of allocated output dewarper buffers. Each buffer contains four dewarped output surfaces. | DS 4.0 |

## 2.8.3 Configuration File Parameters

The configuration file specifies per-surface configuration parameters in [surface<n>] groups, where <n> is an integer from 0 to 3, representing dewarped surfaces 0 to 3.

Table 17. Gst-nvdewarper plugin, configuration file, [surface<n>] parameters

| Property | Meaning | Type and Range | Example *Notes* |
|---|---|---|---|
| output-width | Scale dewarped surfaces to specified output width | Integer, >0 | output-width=960 |
| output-height | Scale dewarped surfaces to specified output height | Integer, >0 | output-height=752 |
| dewarp-dump-frames | Number of dewarped frames to dump. | Integer, >0 | dewarp-dump-frames=10 |
| projection-type | Selects projection type. Supported types are:<br>1: PushBroom<br>2: VertRadCyl | Integer, 1 or 2 | projection-type=1 |

| Property | Meaning | Type and Range | Example<br>*Notes* |
|---|---|---|---|
| surface-index | An index that distinguishes surfaces of the same projection type. | Integer, ≥0 | surface-index=0 |
| width | Dewarped surface width. | Integer, >0 | width=3886 |
| height | Dewarped surface height. | Integer, >0 | height=666 |
| top-angle | Top field of view angle, in degrees. | Float,<br>−180.0 to 180.0 | top-angle=0 |
| bottom-angle | Bottom field of view angle, in degrees. | Float,<br>−180.0 to 180.0 | bottom-angle=0 |
| pitch | Viewing parameter pitch in degrees. | Float,<br>0.0 to 360.0 | pitch=90 |
| yaw | Viewing parameter yaw in degrees. | Float,<br>0.0 to 360.0 | yaw=0 |
| roll | Viewing parameter roll in degrees. | Float,<br>0.0 to 360.0 | roll=0 |
| focal-length | Focal length of camera lens, in pixels per radian. | Float, >0.0 | focal-length=437 |
| aisle-calibration-file | Pathname of the configuration file for aisle view. Set for the 360-D application only.<br>If set, properties in the `[surface<n>]` group are ignored.<br>The configuration file is a CSV file with columns like `sensorId` and `cameraId`, and dewarping parameters like `top-angle`, `bottom-angle`, `yaw`, `roll`, `pitch`, `focal-length`, `width,` and `height`. | String | aisle-calibration-file=<br>csv_files/nvaisle_2M.csv |
| spot-calibration-file | Pathname of the configuration file for spot view. Set for the 360-D application only.<br>If set, properties in the `[surface<n>]` group are ignored.<br>The configuration file is a CSV file with columns like `sensorId` and `cameraId`, and dewarping parameters like `top-angle`, `bottom-angle`, `yaw`, `roll`, `pitch`, `focal-length`, `width`, and `height`. | String | spot-calibration-file=<br>csv_files/nvspot_2M.csv<br>*For an example of a spot view configuration file, see the file in the example above.* |

This plugin can be tested with the one of the following pipelines.

- For dGPU:

```
gst-launch-1.0 filesrc location=streams/sample_cam6.mp4 ! qtdemux !
h264parse ! nvv4l2decoder ! nvvideoconvert ! nvdewarper config-
file=config_dewarper.txt source-id=6 nvbuf-memory-type=3 ! m.sink_0
nvstreammux name=m width=960 height=752 batch-size=4 num-surfaces-
per-frame=4 ! nvmultistreamtiler ! nveglglessink
```

- For Jetson:

```
gst-launch-1.0 filesrc location= streams/sample_cam6.mp4 ! qtdemux !
h264parse ! nvv4l2decoder  ! nvvideoconvert  ! nvdewarper config-
file=config_dewarper.txt source-id=6  ! m.sink_0 nvstreammux name=m
width=960 height=752 batch-size=4 num-surfaces-per-frame=4 !
nvmultistreamtiler ! nvegltransform ! nveglglessink
```

The `Gst-nvdewarper` plugin always outputs a GStreamer buffer which contains the maximum number of dewarped surfaces (currently four surfaces are supported). These dewarped surfaces are scaled to the output resolution (`output-width` × `output-height`) set in the configuration file located at `configs/deepstream-app/config_dewarper.txt`.

Also, the batch size to be set on `Gst-nvstreammux` must be a multiple of the maximum number of dewarped surfaces (currently four).

### 2.8.4 Gst Properties

Table 18 describes the `Gst-nvdewarper` plugin's Gst properties.

Table 18. Gst-nvdewarper plugin, Gst properties

| Property | Meaning | Type and Range | Example and Notes |
|---|---|---|---|
| config-file | Absolute pathname of configuration file for the Gst-nvdewarper element | String | config-file= configs/ deepstream-app/ config_dewarper.txt |
| gpu-id | Device ID of the GPU to be used (dGPU only) | Integer, 0 to 4,294,967,295 | gpu-id=0 |
| source-id | Source ID, e.g. camera ID | Integer, 0 to 4,294,967,295 | source-id=6 |
| num-output-buffers | Number of output buffers to be allocated | Integer, 0 to 4,294,967,295 | num-output-buffers=4 |

| Property | Meaning | Type and Range | Example and Notes |
|---|---|---|---|
| nvbuf-memory-type | Type of memory to be allocated.<br>**For dGPU**:<br>0 (`nvbuf-mem-default`): Default memory, `cuda-device`<br>1 (`nvbuf-mem-cuda-pinned`): Pinned/Host CUDA memory<br>2 (`nvbuf-mem-cuda-device`) Device CUDA memory<br>3 (`nvbuf-mem-cuda-unified`): Unified CUDA memory<br>**For Jetson**:<br>0 (`nvbuf-mem-default`): Default memory, surface array<br>4 (`nvbuf-mem-surface-array`): Surface array memory | Integer, 0 to 4 | nvbuf-memory-type=3 |

## 2.9 GST-NVOF

NVIDIA GPUs, starting with the dGPU Turing generation and Jetson Xavier generation, contain a hardware accelerator for computing optical flow. Optical flow vectors are useful in various use cases such as object detection and tracking, video frame rate up-conversion, depth estimation, stitching, and so on.

The gst-nvof plugin collects a pair of NV12 images and passes it to the low-level optical flow library. The low-level library returns a map of flow vectors between the two frames as its output.

The map of flow vectors is encapsulated in the `NvDsOpticalFlowMeta` structure and is added as a user meta with `meta_type` set to `NVDS_OPTICAL_FLOW_META`. The user meta is added to the `frame_user_meta_list` member of `NvDsFrameMeta`.

For guidance on how to access user metadata, see User/Custom Metadata Addition inside NvDsBatchMeta and Tensor Metadata.

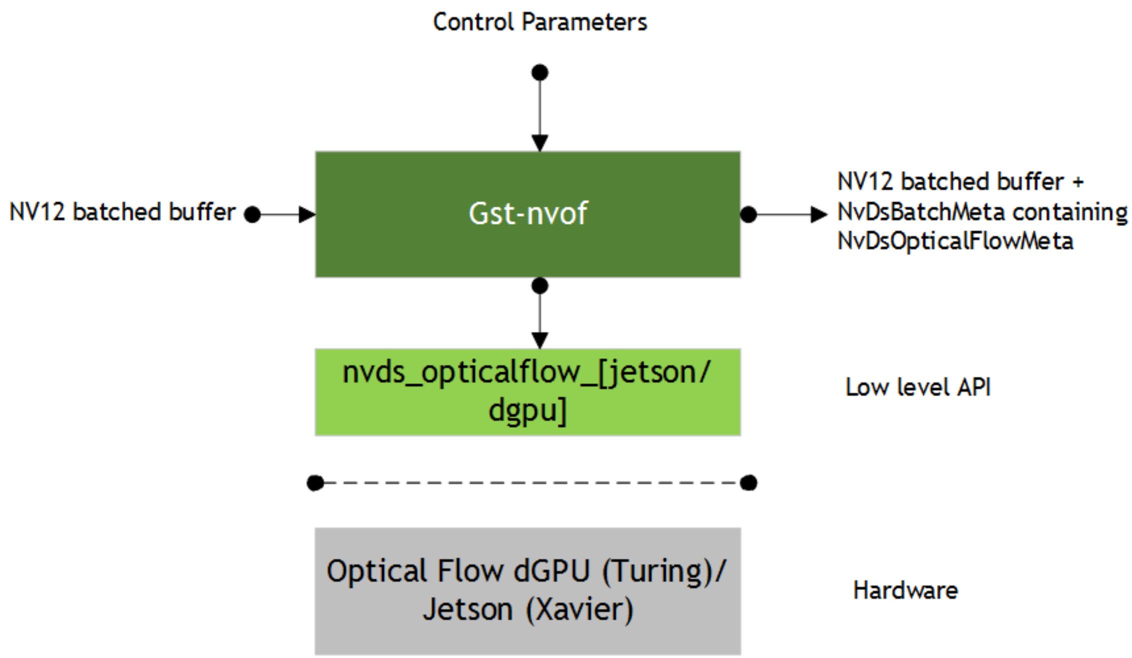


Figure 9. The Gst-nvof plugin

## 2.9.1 Inputs and Outputs

- Inputs
  - GStreamer buffer containing NV12 frame(s)
- Control parameters
  - `gpu-id`: selects the GPU ID (valid only for dGPU platforms)
  - `dump-of-meta`: enables dumping of optical flow map vector into a `.bin` file
  - `preset-level`: sets the preset level
  - `pool-size`: sets the pool size
  - `grid-size`: sets the grid size
- Outputs
  - GStreamer buffer containing NV12 frame(s)
  - `NvDsOpticalFlowMeta` metadata

## 2.9.2 Features

Table 19 summarizes the features of the plugin.

Table 19. Features of the Gst-nvof plugin

| Feature | Description | Release |
|---|---|---|
| Configure GPU selection | Sets the gpu ID to be used for optical flow operation (valid only for dGPU platforms) | DS 4.0 |
| Configure dumping of optical flow metadata | Enables dumping of optical flow output (motion vector data) | DS 4.0 |
| Configure preset level | Sets the desired preset level | DS 4.0 |
| Configure grid size | Sets the flow vector block size | DS 4.0 |

## 2.9.3 Gst Properties

Table 20 describes the Gst properties of the `Gst-nvof` plugin.

Table 20. Gst-nvof plugin, Gst properties

| Property | Meaning | Type and Range | Example *Notes* |
|---|---|---|---|
| gpu-id | Device ID of the GPU to be used for decoding (dGPU only). | Integer,<br>0 to 4,294,967,295 | gpu-id=0 |
| dump-of-meta | Dumps optical flow output into a `.bin` file. | Boolean | dump-of-meta=1 |
| preset-level | Selects a preset level, default preset level is 0 i.e. NV_OF_PERF_LEVEL_FAST<br>Possible values are:<br>0 (`NV_OF_PERF_LEVEL_FAST`): high performance, low quality.<br>1 (`NV_OF_PERF_LEVEL_MEDIUM`): intermediate performance and quality.<br>2 (`NV_OF_PERF_LEVEL_SLOW`): low performance, best quality (valid only for dGPU platforms). | Enum, 0 to 2 | preset-level=0 |
| grid-size | Selects the grid size. The hardware generates flow vectors blockwise, one vector for each block of 4×4 pixels. Currently only the 4x4 grid size is supported. | Enum, 0 | grid-size=0 |
| pool-size | Sets the number of internal motion vector output buffers to be allocated. | Integer,<br>1 to 4,294,967,295 | pool-size=7 |

## 2.10 GST-NVOFVISUAL

The `Gst-nvofvisual` plugin is useful for visualizing motion vector data. The visualization method is similar to the OpenCV reference source code in:

```
https://github.com/opencv/opencv/blob/master/samples/gpu/optical_flow.cpp
```

The plugin solves the optical flow problem by computing the magnitude and direction of optical flow from a two-channel array of flow vectors. It then visualizes the angle (direction) of flow by hue and the distance (magnitude) of flow by value of Hue Saturation Value (**HSV**) color representation. The strength of HSV is always set to a maximum of 255 for optimal visibility.

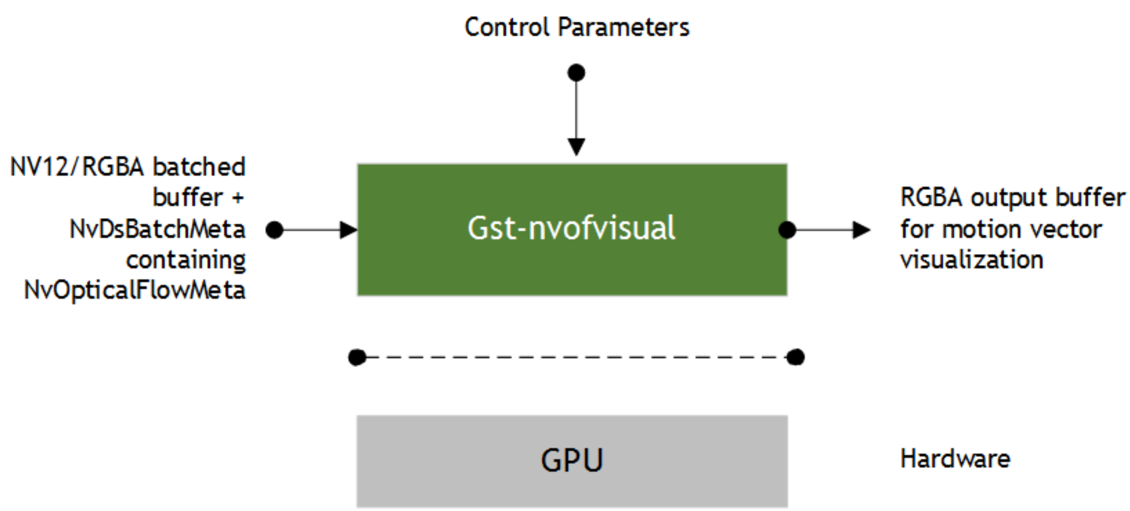


Figure 10. The Gst-nvofvisual plugin

### 2.10.1 Inputs and Outputs

- Inputs
  - GStreamer buffer containing NV12/RGBA frame(s)
  - `NvDsOpticalFlowMeta` containing the motion vector (MV) data generated by the gst-nvof plugin
- Control parameters
  - `gpu-id,` selects the GPU ID
- Output
  - GStreamer buffer containing RGBA frame(s)

- RGBA buffer generated by transforming MV data into color-coded RGBA image reference

### 2.10.2 Features

Table 21 summarizes the features of the plugin.

Table 21. Features of the Gst-nvofvisual plugin

| Feature | Description | Release |
|---|---|---|
| Configure GPU selection | Sets the GPU ID to be used for optical flow visualization operations (valid only for dGPU platforms) | DS 4.0 |

### 2.10.3 Gst Properties

Table 22 describes the Gst properties of the `Gst-nvofvisual` plugin.

Table 22. Gst-nvofvisual plugin, Gst Properties

| Property | Meaning | Type and Range | Example *Notes* |
|---|---|---|---|
| gpu-id | Device ID of the GPU to be used **(dGPU only)** | Integer, 0 to 4,294,967,295 | gpu-id=0 |

## 2.11 GST-NVSEGVISUAL

The `Gst-nvsegvisual` plugin visualizes segmentation results. Segmentation is based on image recognition, except that the classifications occur at the pixel level as opposed to the image level as with image recognition. The segmentation output size is generally same as the input size.

For more information, see the segmentation training reference at:

```
https://github.com/qubvel/segmentation_models
```

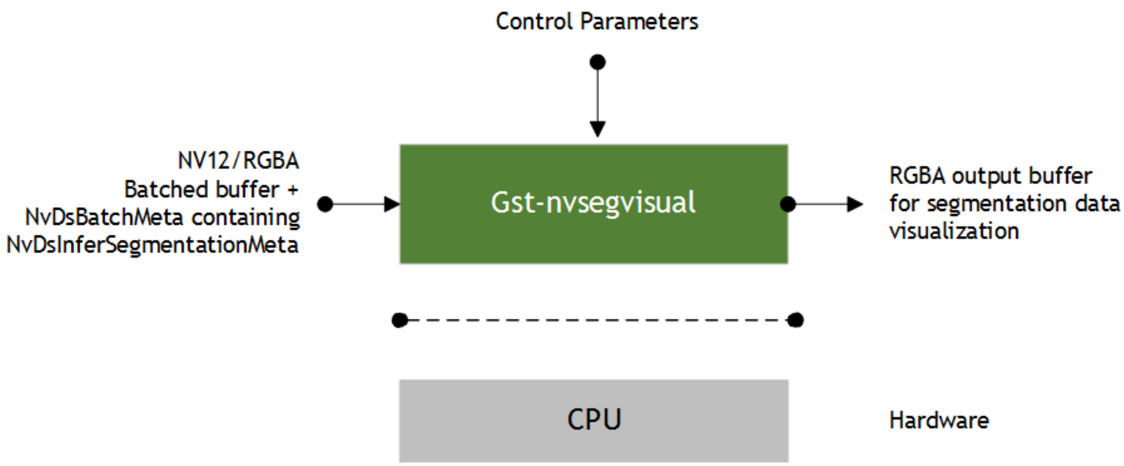


Figure 11. The Gst-nvsegvisual plugin

## 2.11.1 Inputs and Outputs

- Inputs
  - GStreamer buffer containing NV12/RGBA frame(s)
  - `NvDsInferSegmentationMeta` containing class numbers, pixel class map, width, height, etc. generated by gst-nvinfer.
  - `gpu-id`: selects the GPU ID
  - width, set according the segmentation output size
  - height, set according the segmentation output size
- Output

  This plugin allocates different colors for different classes. For example, the industrial model's output has only one representing defective areas. Thus defective areas and background have different colors. The semantic model outputs four classes with four different colors: car, pedestrian, bicycle, and background.

  This plugin shows only the segmentation output. It does not overlay output on the original NV12 frame.

Table 23 summarizes the features of the plugin.

Table 23. Features of the Gst-nvsegvisual plugin

| Feature | Description | Release |
|---|---|---|
| Configure GPU selection | Sets the GPU ID to be used for segmentation visualization operations (valid only for dGPU platforms) | DS 4.0 |
| Configure width | Sets width according to the segmentation output size | DS 4.0 |

| Feature | Description | Release |
|---|---|---|
| Configure height | Sets height according to the segmentation output size | DS 4.0 |

### 2.11.2 Gst Properties

Table 24 describes the Gst properties of the `Gst-nvsegvisual` plugin.

Table 24. Gst-nvsegvisual plugin, Gst Properties

| Property | Meaning | Type and Range | Example and Notes |
|---|---|---|---|
| gpu-id | Device ID of the GPU to be used for decoding | Integer, 0 to 4,294,967,295 | gpu-id=0 |
| width | Segmentation output width | Integer, 0 to 4,294,967,295 | width=512 |
| height | Segmentation output height | Integer, 0 to 4,294,967,295 | height=512 |

## 2.12 GST-NVVIDEO4LINUX2

DeepStream extends the open source V4L2 codec plugins (here called Gst-v4l2) to support hardware-accelerated codecs.

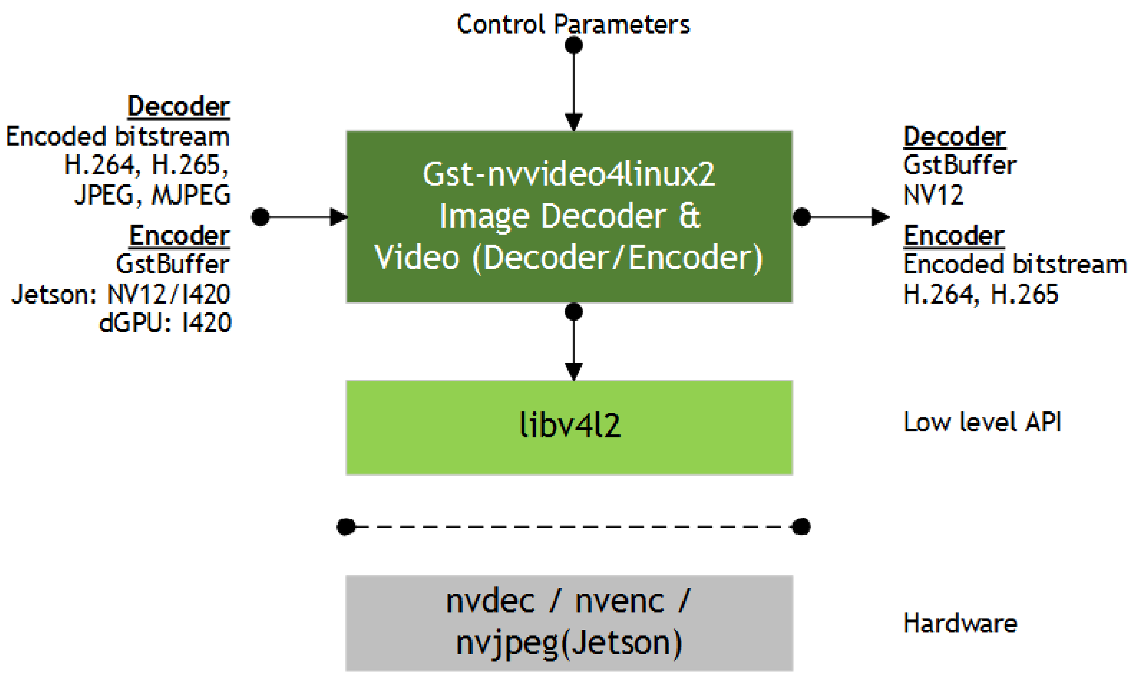


Figure 12. The Gst-nvvideo4linux2 decoder plugin

## 2.12.1 Decoder

The OSS Gst-nvvideo4linux2 plugin leverages the hardware decoding engines on Jetson and DGPU platforms by interfacing with `libv4l2` plugins on those platforms. It supports H.264, H.265, JPEG and MJPEG formats.

The plugin accepts an encoded bitstream & NVDEC h/w engine to decoder the bitstream. The decoded output is in NV12 format.

| Note: | **When you use the v4l2 decoder use for decoding JPEG images, you must use the open source `jpegparse` plugin before the decoder to parse encoded JPEG images.** |
|---|---|

### 2.12.1.1 Inputs and Outputs

- Inputs
  - An encoded bitstream. Supported formats are H.264, H.265, JPEG and MJPEG
- Control Parameters
  - `gpu-id`
  - `num-extra-surfaces`
  - `skip-frames`
  - `cudadec-memtype`

- `drop-frame-interval`

- Output
  - Gst Buffer with decoded output in NV12 format

### 2.12.1.2 Features

| Feature | Description | Release |
|---|---|---|
| Supports H.264 decode | — | DS 4.0 |
| Supports H.265 decode | — | DS 4.0 |
| Supports JPEG/MJPEG decode | _ | DS 4.0 |
| User-configurable CUDA memory type (Pinned/Device/Unified) for output buffers | — | DS 4.0 |

### 2.12.1.3 Configuration Parameters

| Property | Meaning | Type and Range | Example *Notes* | Platforms |
|---|---|---|---|---|
| gpu-id | Device ID of GPU to use for decoding. | Integer,<br>0 to 4,294,967,295 | gpu-id=0 | dGPU |
| num-extra-surfaces | Number of surfaces in addition to min decode surfaces given by the V4L2 driver. | Integer, 1 to 24 | num-decode-surfaces=24<br>*Default: 0* | dGPU<br>Jetson |
| skip-frames | Type of frames to skip during decoding. Represented internally by enum *SkipFrame*.<br>0 (`decode_all`): decode all frames<br>1 (`decode_non_ref`): decode non-ref frames<br>2 (`decode_key`): decode key frames | Integer,<br>0, 1, or 2 | skip-frames=0<br>*Default: 0* | dGPU<br>Jetson |
| drop-frame-interval | Interval to drop the frames, e.g. a value of 5 means the decoder receives every fifth frame, and others are dropped. | Integer,<br>1 to 30 | *Default: 0* | dGPU<br>Jetson |
| cudadec-memtype | Memory type for CUDA decoder buffers. Represented internally by enum `CudaDecMemType`.<br>0 (`memtype_device`): Device<br>1 (`memtype_pinned`): Host Pinned<br>2 (`memtype_unified`): Unified | Integer,<br>0, 1, or 2 | cuda-memory-type=1<br>*Default: 2* | dGPU |

### 2.12.2 Encoder

The OSS Gst-nvvideo4linux2 plugin leverages the hardware accelerated encoding engine available on Jetson and dGPU platforms by interfacing with `libv4l2` plugins on those platforms. The plugin accepts RAW data in I420 format. It uses the NVENC hardware engine to encode RAW input. Encoded output is elementary bitstream supported formats.

#### 2.12.2.1 Inputs and Outputs

- Inputs
  - RAW input in I420 format
- Control parameters
  - `gpu-id` (dGPU only)
  - `profile`
  - `bitrate`
  - `control-rate`
  - `iframeinterval`
- Output
  - Gst Buffer with encoded output in H264, H265, VP8 or VP9 format.

#### 2.12.2.2 Features

| Feature | Description | Release |
|---|---|---|
| Supports H.264 encode | — | DS 4.0 |
| Supports H.265 encode | — | DS 4.0 |

#### 2.12.2.3 Configuration Parameters

| Property | Meaning | Type and Range | Example Notes | Platforms |
|---|---|---|---|---|
| gpu-id | Device ID of GPU to used. | Integer,<br>0 to 4,294,967,295 | gpu-id=0 | dGPU |
| bitrate | Sets bitrate for encoding, in bits/seconds. | Integer,<br>0 to 4,294,967,295 | Default:4000000 | dGPU<br>Jetson |
| iframeinterval | Encoding intra-frame occurrence frequency. | Unsigned integer,<br>0 to 4,294,967,295 | *Default: 30* | dGPU<br>Jetson |

| | | | | |
|---|---|---|---|---|
| Profile | H.264/H.265 encoder profile; represented internally by enum `GstV4l2VideoEncProfileType`.<br>**For H.264:**<br>0 (`GST_V4L2_H264_VIDENC_BASELINE_PROFILE`): Baseline<br>2 (`GST_V4L2_H264_VIDENC_MAIN_PROFILE`): Main<br>4 (`GST_V4L2_H264_VIDENC_HIGH_PROFILE`): High<br>**For H.265:**<br>0 (`GST_V4L2_H265_VIDENC_MAIN_PROFILE`): Main<br>1 (`GST_V4L2_H265_VIDENC_MAIN10_PROFILE`): Main10 | Values of enum `GstV4l2VideoEncProfileType` | Default Baseline<br>*Default: 0* | dGPU<br>Jetson |

# 2.13 GST-NVJPEGDEC

The Gst-nvjpegdec plugin decodes images on both dGPU and Jetson platforms. It is the preferred method for decoding JPEG images.

On the dGPU platform this plugin is based on the `libnvjpeg` library, part of the CUDA toolkit. On Jetson it uses a platform-specific hardware accelerator.

The plugin uses an internal software parser to parse JPEG streams. Thus there is no need to use a `jpegparse` open source plugin separately to parse the encoded frame.

The plugin accepts a JPEG encoded bitstream and produces RGBA output on the dGPU platform, and produces I420 output on the Jetson platform.

## 2.13.1 Inputs and Outputs

- Inputs
  - Elementary JPEG
- Control parameters
  - `gpu-id` (dGPU only)
  - DeepStream (Jetson only)
- Output
  - Gst Buffer with decoded output in RGBA format

### 2.13.2 Features

| Feature | Description | Release |
|---|---|---|
| Supports JPEG Decode | — | DS 4.0 |
| Supports MJPEG Decode | — | DS 4.0 |

### 2.13.3 Configuration Parameters

| Property | Meaning | Type and Range | Example and Notes | Platforms |
|---|---|---|---|---|
| gpu-id | Device ID of GPU to use for decoding. | Integer,<br>0 to 4,294,967,295 | gpu-id=0 | dGPU |
| DeepStream | Applicable only for Jetson; required for outputting buffer with new `NvBufSurface` or Legacy Buffer | Boolean | DeepStream=1 | Jetson |

## 2.14 GST-NVMSGCONV

The Gst-nvmsgconv plugin parses `NVDS_EVENT_MSG_META` (`NvDsEventMsgMeta`) type metadata attached to the buffer as user metadata of frame meta and generates the schema payload. For the batched buffer, metadata of all objects of a frame must be under the corresponding frame meta.

The generated payload (`NvDsPayload`) is attached back to the input buffer as `NVDS_PAYLOAD_META` type user metadata.

DeepStream 4.0 supports two variations of the schema, full and minimal. The Gst-nvmsgconv plugin can be configured to use either one of the schemas.

By default, the plugin uses the full DeepStream schema to generate the payload in JSON format. The full schema supports elaborate semantics for object detection, analytics modules, events, location, and sensor. Each payload has information about a single object.

You can use the minimal variation of the schema to communicate minimal information with the back end. This provides a small footprint for the payload to be transmitted from DeepStream to a message broker. Each payload can have information for multiple objects in the frame.

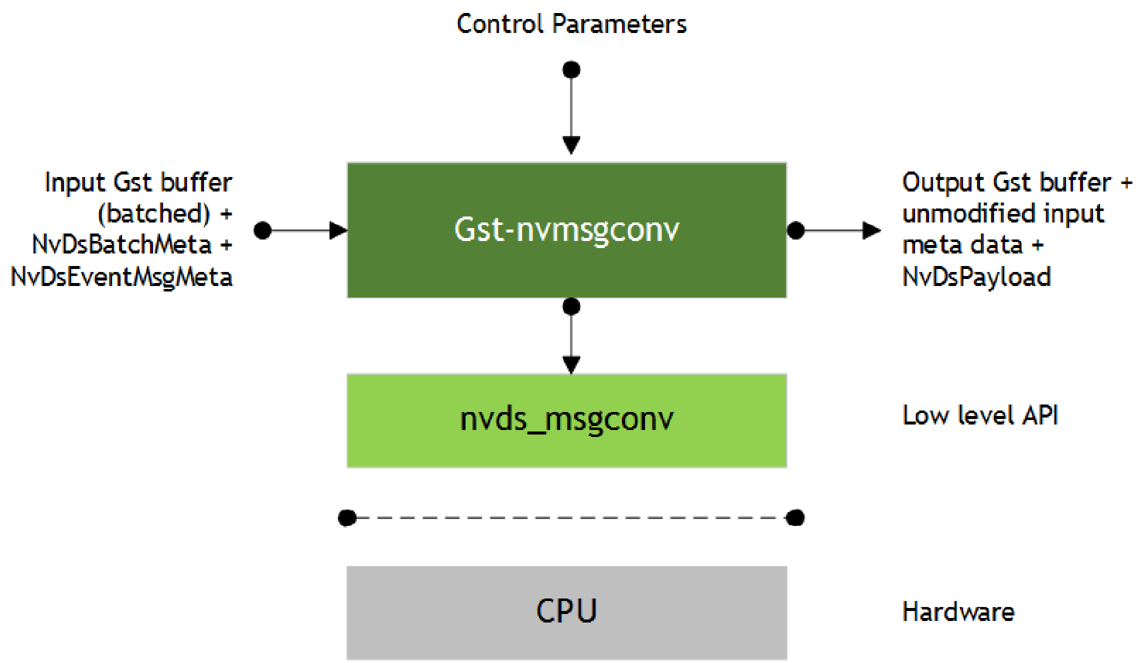


Figure 13. The Gst-nvmsgconv plugin

## 2.14.1 Inputs and Outputs

- Inputs
  - Gst Buffer with `NvDsEventMsgMeta`
- Control parameters
  - `config`
  - `msg2p-lib`
  - `payload-type`
  - `comp-id`
- Output
  - Same Gst Buffer with additional `NvDsPayload` metadata. This metadata contains information about the payload generated by the plugin.

## 2.14.2 Features

Table 25 summarizes the features of the plugin.

Table 25. Features of the Gst-nvmsgconv plugin

| Feature | Description | Release |
|---|---|---|
| Payload in JSON format | Message payload is generated in JSON format | DS 3.0 |
| Supports DeepStream schema specification | DeepStream schema spec implementation for messages | DS 3.0 |
| Custom schema specification | Provision for custom schemas for messages | DS 3.0 |
| Key-value file parsing for static properties | Read static properties of sensor/place/module in the form of key-value pair from a text file | DS 3.0 |
| CSV file parsing for static properties | Read static properties of sensor/place/module from a CSV file | DS 3.0 |
| DeepStream 4.0 minimalistic schema | Minimal variation of the DeepStream message schema | DS 4.0 |

## 2.14.3 Gst Properties

Table 26 describes the `Gst-nvmsgconv` plugin's Gst properties.

Table 26. Gst-nvmsgconv plugin, Gst properties

| Property | Meaning | Type and Range | Example *Notes* | Platforms |
|---|---|---|---|---|
| config | Absolute pathname of a configuration file that defines static properties of various sensors, places, and modules. | String | config=msgconv_config.txt<br>or<br>config=msgconv_config.csv | dGPU<br>Jetson |
| msg2p-lib | Absolute pathname of the library containing a custom implementation of the `nvds_msg2p_*` interface for custom payload generation. | String | msg2p-lib=libnvds_msgconv_custom.so | dGPU<br>Jetson |
| payload-type | Type of schema payload to be generated. Possible values are:<br>`PAYLOAD_DEEPSTREAM`: Payload using DeepStream schema.<br>`PAYLOAD_DEEPSTREAM_MINIMAL`: Payload using minimal DeepStream schema.<br>`PAYLOAD_CUSTOM`: Payload using custom schemas. | Integer,<br>0 to 4,294,967,295 | payload-type=0<br>or<br>payload-type=257 | dGPU<br>Jetson |

| Property | Meaning | Type and Range | Example<br>*Notes* | Platforms |
|---|---|---|---|---|
| comp-id | Component ID of the plugin from which metadata is to be processed. | Integer,<br>0 to 4,294,967,295 | comp-id=2<br>*Default is* *`NvDsEventMsgMeta`* | dGPU<br>Jetson |

### 2.14.4 Schema Customization

This plugin can be used to implement a custom schema in two ways:

- **By modifying the payload generator library**: To perform a simple customization of DeepStream schema fields, modify the low level payload generation library file `sources/libs/nvmsgconv/nvmsgconv.cpp`.
- **By implementing the `nvds_msg2p` interface**: If a library that implements the custom schema needs to be integrated with the DeepStream SDK, wrap the library in the `nvds_msg2p` interface and set the plugin's `msg2p-lib` property to the library's name. Set the `payload-type` property to `PAYLOAD_CUSTOM`.

See `sources/libs/nvmsgconv/nvmsgconv.cpp` for an example implementation of the `nvds_msg2p` interface.

### 2.14.5 Payload with Custom Objects

You can add a group of custom objects to the `NvDsEventMsgMeta` structure in the `extMsg` field and specify their size in the `extMsgSize` field. The meta copy (`copy_func`) and free (`release_func`) functions must handle the custom fields accordingly.

The payload generator library handles some standard types of objects (Vehicle, Person, Face, etc.) and generates the payload according to the schema selected. To handle custom object types, you must modify the payload generator library `nvmsgconv.cpp`.

See `deepstream-test4` for details about adding custom objects as `NVDS_EVENT_MSG_META` user metadata with buffers for generating a custom payload to send to back end.

## 2.15 GST-NVMSGBROKER

This plugin sends payload messages to the server using a specified communication protocol. It accepts any buffer that has `NvDsPayload` metadata attached, and uses the `nvds_msgapi_*` interface to send the messages to the server. You must implement the `nvds_msgapi_*` interface for the protocol to be used and specify the implementing library in the `proto-lib` property.

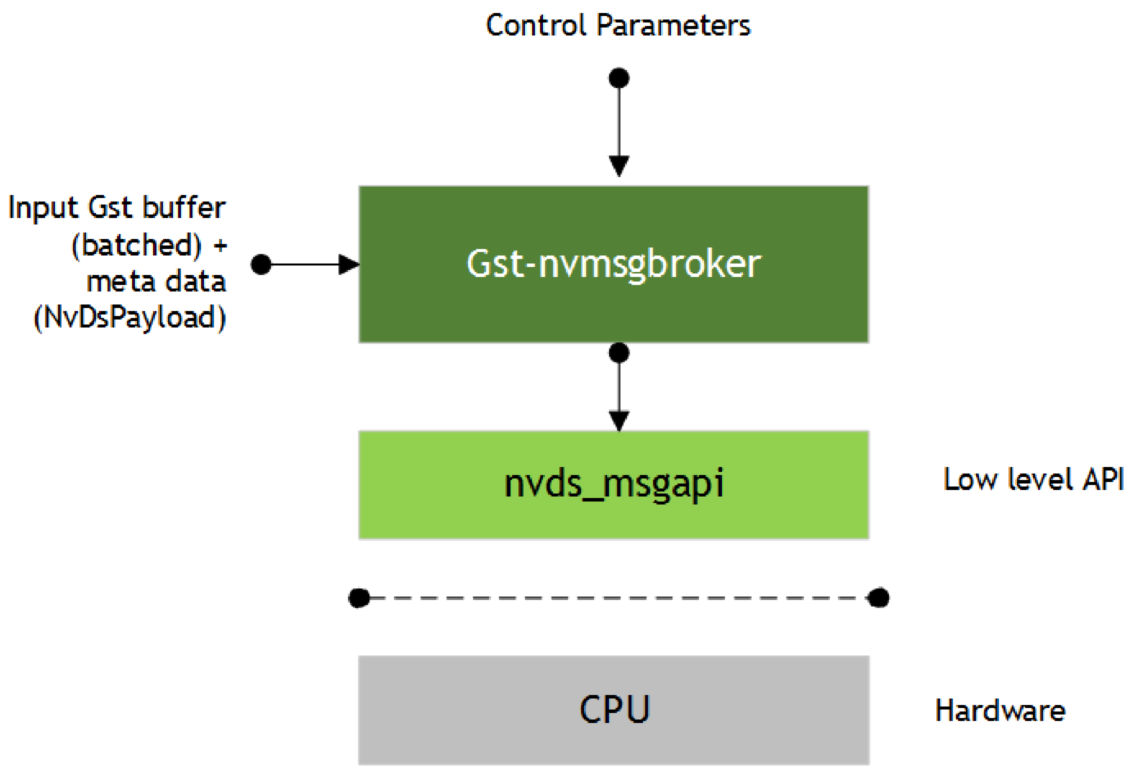


Figure 14. The Gst-nvmsgbroker plugin

## 2.15.1 Inputs and Outputs

- Inputs
  - Gst Buffer with `NvDsPayload`
- Control parameters
  - `Config`
  - `conn-str`
  - `proto-lib`
  - `comp-id`
  - `topic`
- Output
  - *None, as this is a sink type component*

## 2.15.2 Features

Table 27 summarizes the features of the `Gst-nvmsgbroker` plugin.

Table 27. Features of the Gst-nvmsgbroker plugin

| Feature | Description | Release |
|---|---|---|
| Payload in JSON format | Accepts message payload in JSON format | DS 3.0 |
| Kafka protocol support | Kafka protocol adapter implementation | DS 3.0 |
| Azure IOT support | Integration with Azure IOT framework | DS 4.0 |
| AMQP support | AMQP 0-9-1 protocol adapter implementation | DS 4.0 |
| Custom protocol support | Provision to support custom protocol through a custom implementation of the adapter interface | DS 3.0 |
| Configurable parameters | Protocol specific options through configuration file | DS 3.0 |

## 2.15.3 Gst Properties

Table 28 describes the Gst properties of the `Gst-nvmsgbroker` plugin.

Table 28. Gst-nvmsgbroker plugin, Gst Properties

| Property | Meaning | Type and Range | Example *Notes* | Platforms |
|---|---|---|---|---|
| config | Absolute pathname of configuration file required by `nvds_msgapi_*` interface | String | config=msgapi_ config.txt | dGPU Jetson |
| conn-str | Connection string as end point for communication with server | String Format must be <name>;<port>; <topic-name> | conn-str=foo.bar.com;80 or conn-str=foo.bar.com; 80;dsapp1 | dGPU Jetson |
| proto-lib | Absolute pathname of library that contains the protocol adapter as an implementation of `nvds_msgapi_*` | String | proto-lib=libnvds_ kafka_proto.so | dGPU Jetson |
| comp-id | ID of component from which metadata should be processed | Integer, 0 to 4,294,967,295 | comp-id=3 *Default: plugin processes metadata from any component* | dGPU Jetson |
| topic | Message topic name | String | topic=dsapp1 | dGPU Jetson |

## 2.15.4 nvds_msgapi: Protocol Adapter Interface

You can use the NVIDIA DeepStream messaging interface, `nvds_msgapi`, to implement a custom protocol message handler and integrate it with DeepStream applications. Such a message handler, known as a **protocol adapter**, enables you to integrate DeepStream applications with backend data sources, such as data stored in the cloud.

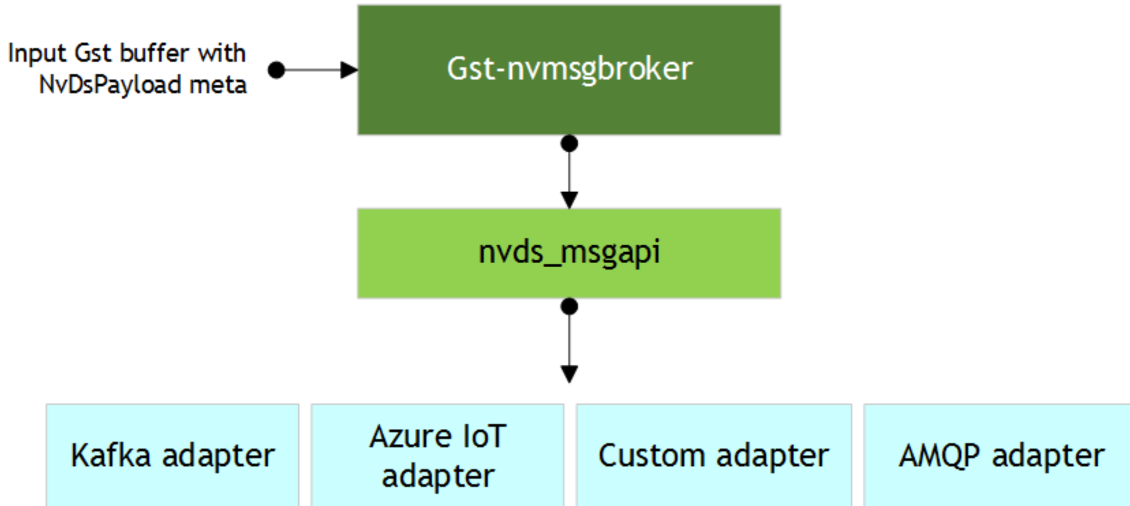


Figure 15. The Gst-nvmsgbroker plugin calling the nvds_msgapi interface

The `Gst-nvmsgbroker` plugin calls the functions in your protocol adapter as shown in Figure 15. These functions support:

- Creating a connection
- Sending messages by synchronous or asynchronous means
- Terminating the connection
- Coordinating the client's and protocol adapter's use of CPU resources and threads
- Getting the protocol adapter's version number

The `nvds_msgapi` interface is defined in the header file `source/includes/nvds_msgapi.h`. This header file defines a set of function pointers which provide an interface analogous to an interface in C++.

The following sections describe the methods defined by the `nvds_msgapi` interface.

### 2.15.4.1 nvds_msgapi_connect(): Create a Connection

```
NvDsMsgApiHandle nvds_msgapi_connect(char *connection_str,
        nvds_msgapi_connect_cb_t connect_cb, char *config_path
);
```

The function accepts a connection string and configures a connection. The adapter implementation can choose whether or not the function actually makes a connection to accommodate connectionless protocols such as HTTP.

#### Parameters

- *connection_str*: A pointer to a string that specifies connection parameters in the general format `"<url>;<port>;<specifier>"`.
  - `<url>` and `<port>` specify the network address of the remote entity.
  - `<specifier>` specifies information specific to a protocol. Its content depends on the protocol's implementation. It may be a topic for messaging, for example, or a client identifier for making the connection.

  Note that this connection string format is not binding, and a particular adapter may omit some fields (eg: `specifier`) from its format, provided the omission is described in its documentation.

  A special case of such connection string adaptation is where the adapter expects all connection parameters to be specified as fields in the configuration file (see config path below), in which case the connection string is passed as NULL.
- *connect_cb*: A callback function for events associated with the connection.
- *config_path*: The pathname of a configuration file that defines protocol parameters used by the adapter.

#### Return Value

A handle for use in subsequent interface calls if successful, or NULL otherwise.

### 2.15.4.2 nvds_msgapi_send() and nvds_msgapi_send_async(): Send an event

```
NvDsMsgApiErrorType nvds_msgapi_send(NvDsMsgApiHandle *h_ptr,
        char *topic, uint8_t *payload, size_t nbuf
);

NvDsMsgApiErrorType nvds_msgapi_send_async(NvDsMsgApiHandle h_ptr,
        char *topic, const uint8_t *payload, size_t nbuf,
        nvds_msgapi_send_cb_t send_callback, void *user_ptr
);
```

Both functions send data to the endpoint of a connection. They accept a message topic and a message payload.

The `nvds_send()` function is synchronous. The `nvds_msgapi_send_async()` function is asynchronous; it accepts a callback function that is called when the "send" operation is completed.

Both functions allow the API client to control execution of the adapter logic by calling `nvds_msgapi_do_work()`. See the description of the `nvds_msgapi_do_work()` function.

**Parameters**

- *h_ptr*: A handle for the connection, obtained by a call to `nvds_msgapi_connect()`.
- *topic*: A pointer to a string that specifies a topic for the message; may be NULL if *topic* is not meaningful for the semantics of the protocol adapter.
- *payload*: A pointer to a byte array that contains the payload for the message.
- *nbuf*: Number of bytes to be sent.
- *send_callback*: A pointer to a callback function that the asynchronous function calls when the "send" operation is complete. The signature of the callback function is of type `nvds_msgapi_send_cb_t`, defined as:

  ```
  typedef void (*nvds_msgapi_send_cb_t)(void *user_ptr,
          NvDsMsgApiErrorType completion_flag
  );
  ```

  Where the callback's parameters are:

  - *user_ptr*: The user pointer (*user_ptr*) from the call to `nvds_msgapi_send()` or `nvds_msgapi_send_async()` that initiated the "send" operation. Enables the callback function to identify the initiating call.
  - *completion_flag*: A code that indicates the completion status of the asynchronous send operation.

#### 2.15.4.3 nvds_msgapi_do_work(): Incremental Execution of Adapter Logic

```
void nvds_msgapi_do_work();
```

The protocol adapter must periodically surrender control to the client during processing of `nvds_msgapi_send()` and `nvds_msgapi_send_async()` calls. The client must periodically call `nvsd_msgapi_do_work()` to let the protocol adapter resume execution. This ensures that the protocol adapter receives sufficient CPU resources. The client can use this convention to control the protocol adapter's use of multi-threading and thread scheduling. The protocol adapter can use it to support heartbeat functionality, if the underlying protocol requires that.

The `nvds_msgapi_do_work()` convention is needed when the protocol adapter executes in the client thread. Alternatively, the protocol adapter may execute time-consuming operations in its own thread. In this case the protocol adapter need not surrender control to the client, the client need not call `nvsd_msgapi_do_work()`, and the implementation of `nvds_msgapi_do_work()` may be a no-op.

The protocol adapter's documentation must specify whether the client must call `nvds_msgapi_do_work()`, and if so, how often.

#### 2.15.4.4 nvds_msgapi_disconnect(): Terminate a Connection

```
NvDsMsgApiErrorType nvds_msgapi_disconnect(NvDsMsgApiHandle h_ptr);
```

The function terminates the connection, if the underlying protocol requires it, and frees resources associated with *h_ptr*.

**Parameters**

- *h_ptr*: A handle for the connection, obtained by a call to `nvds_msgapi_connect()`.

#### 2.15.4.5 nvds_msgapi_getversion(): Get Version Number

```
char *nvds_msgapi_getversion();
```

This function returns a string that identifies the `nvds_msgapi` version supported by this protocol adapter implementation. The string must use the format `"<major>.<minor>"`, where `<major>` is a major version number and `<minor>` is a minor version number. A change in the major version number indicates an API change that may cause incompatibility. When the major version number changes, the minor version number is reset to 1.

### 2.15.5 nvds_kafka_proto: Kafka Protocol Adapter

The DeepStream 3.0 release includes a protocol adapter that supports Apache Kafka. The adapter provides out-of-the-box capability for DeepStream applications to publish messages to Kafka brokers.

#### 2.15.5.1 Installing Dependencies

The Kafka adapter uses `librdkafka` for the underlying protocol implementation. This library must be installed prior to use.

To install `librdkakfa`, enter these commands:

```
git clone https://github.com/edenhill/librdkafka.git
cd librdkafka
git reset --hard 7101c2310341ab3f4675fc565f64f0967e135a6a
./configure
make
sudo make install
sudo cp /usr/local/lib/librdkafka* /opt/nvidia/deepstream/deepstream-
4.0/lib
```

Install additional dependencies:

```
sudo apt-get install libglib2.0 libglib2.0-dev
sudo apt-get install libjansson4 libjansson-dev
```

#### 2.15.5.2 Using the Adapter

You can use the Kafka adapter in an application by setting the `Gst-nvmsgbroker` plugin's `proto-lib` property to the pathname of the adapter's shared library, `libnvds_kafka_proto.so`. The plugin's `conn-str` property must be set to a string with format:

```
<kafka broker address>;<port>;<topic-name>
```

This instantiates the `Gst-nvmsgbroker` plugin and makes it use the Kafka protocol adapter to publish messages that the application sends to the broker at the specified broker address and topic.

#### 2.15.5.3 Configuring Protocol Settings

You can define configuration setting for the Kafka protocol adapter as described by the documentation at:

```
https://github.com/edenhill/librdkafka/blob/master/CONFIGURATION.md
```

You can set these options in the `Gst-nvmsgbroker` configuration file. Like the rest of DeepStream, the configuration file use the gkey format. The Kafka settings must be in a group named `[message-broker]`, and must be specified as part of a key named `proto-cfg`. The settings can be a series of key-value pairs separated by semicolons, for example:

```
[message-broker]
proto-cfg="message.timeout.ms=2000;retries=5"
```

The Kafka adapter lets you specify the name of the field in messages that is to be used to define the partition key. For each message, the specified message field is extracted and send to the topic partitioner along with the message. The partitioner uses it to identify the partition in the Kafka cluster that handles the message. The partition key information must be specified in the Gst-nvmsgbroker configuration file's `[message-broker]` group, using an entry named `partition-key`.

Fields embedded in a hierarchy of JSON objects in the message are specified using dotted notation. For example, for the sample JSON message shown below, the `id` field in the `sensor` object is identified as `sensor.id`,

```
{
    "sensor" {
        "id": "cam1"
    }
}
```

| Note: | **For the DeepStream reference application and the 360-D application, both distributed with the DeepStream SDK, you can add the `proto-cfg` setting to the `[message-broker]` group of the top level configuration file passed to the application.** |
|---|---|

### 2.15.5.4 Programmatic Integration

You can integrate the Kafka adapter into custom user code by using the `nvds_msgapi` interface to call its functions. Note the following points with regard to the functions defined by the interface:

- The connection string passed to the `nvdm_msgapi_connect()` has the format `<kafka broker address>;<port>;<topic-name>`.
- For both "send" functions, the topic name must match the topic name passed to `nvds_msgapi_connect()`.
- The application must call `nvds_msgapi_do_work()` at least once a second, and preferably more often. The frequency of calls to `nvds_msgapi_do_work()` determines the rate at which messages waiting to be sent are processed.
- It is safe for multiple application threads to share connection handles. The library `librdkafka` is thread-safe, so Kafka protocol adapter does not need to implement separate locking mechanisms for functions calling directly to this library.
- The Kafka protocol adapter expects the client to manage usage and retirement of the connection handle. The client must ensure that once a handle is disconnected, it is not used for either a "send" call or a call to `nvds_msgapi_do_work()`. While the library attempts to ensure graceful failure if the application calls these functions with retired handles, it does not do so in a thread-safe manner.

### 2.15.5.5 Monitor Adapter Execution

The Kafka adapter generates log messages based on the `nvds_logger` framework to help you monitor execution. The adapter generates separate logs for the `INFO`, `DEBUG`, and `ERROR` severity levels, as described in nvds_logger: The Logger Framework. You can limit the log messages generated by setting the level at which log messages are filtered as part of the logging setup script.

**Note:** **If the severity level is set to `DEBUG`, the `nvds_logger` framework logs the entire contents of each message sent by the Kafka protocol adapter.**

### 2.15.6 Azure MQTT Protocol Adapter Libraries

The DeepStream 4.0 release includes protocol adapters that supports direct messaging from device to cloud (using the Azure device client adapter) and through Azure IoT Edge runtime (using the Azure module client adapter). The adapters provide out-of-the-box capability for DeepStream applications to publish messages to Azure IoT Hub using the MQTT protocol.

The Azure IoT protocol adapters are encapsulated by their respective shared libraries found within the `deepstream` package at the location:

```
/opt/nvidia/deepstream/deepstream-4.0/lib
```

The Azure device client adapter library is named `libnvds_azure_proto.so`.

The Azure module client adapter library is named `libnvds_azure_edge_proto.so`.

#### 2.15.6.1 Installing Dependencies

Azure adapters use `libiothub_client.so` from the Azure IoT C SDK (v1.2.8) for the underlying protocol implementation. After you install the `deepstream` package you can find the precompiled library at:

```
/opt/nvidia/deepstream/deepstream-4.0/lib/libiothub_client.so
```

You can also compile `libiothub_client.so` manually by entering these commands:

```
git clone -b 2018-07-11 --recursive https://github.com/Azure/azure-iot-
sdk-c.git
cd azure-iot-sdk-c
mkdir cmake
cd cmake
cmake ..
cmake --build .  # append '-- -j <n>' to run <n> jobs in parallel
```

To install some other required dependencies, enter one of these commands.

- For an x86 computer using Ubuntu 18.04:

```
sudo apt-get install -y libcurl3 libssl-dev uuid-dev libglib2.0
libglib2.0-dev
```

- For other platforms or OSes:

```
sudo apt-get install -y libcurl4-openssl-dev libssl-dev uuid-dev
libglib2.0 libglib2.0-dev
```

### 2.15.6.2 Setting Up Azure IoT

Azure IoT adapter needs a functioning Azure IoT Hub instance to which is can publish messages. To set up an Azure IoT Hub instance if required, see the instructions at:

```
https://docs.microsoft.com/en-us/azure/iot-hub/tutorial-connectivity
```

After you create the Azure IoT instance, create a device entry corresponding to the device that is running DeepStream.

To set up Azure IoT Edge runtime on the edge device, see the instructions at:

```
https://docs.microsoft.com/en-us/azure/iot-edge/how-to-install-iot-
edge-linux
```

### 2.15.6.3 Configuring Adapter Settings

Place Azure IoT specific information in a custom configuration file named, e.g., `cfg_azure.txt`. The entries in the configuration file vary slightly between the Azure device client and the module client.

- For an Azure device client:

```
[message-broker]
connection_str = HostName=<my-hub>.azure-
devices.net;DeviceId=<device_id>;
SharedAccessKey=<my-policy-key>
shared_access_key = <my-policy-key>
custom_msg_properties =  <key1>=<value1>; <key2>=<value2>;
<key3>=<value3>;
```

- For an Azure module client:

```
[message-broker]
#custom_msg_properties =  <key1>=<value1>; <key2>=<value2>;
<key3>=<value3>;
```

Here is useful information about some of the configuration file properties:

- **connection_str**: You can obtain the Azure connection string from the Azure IoT Hub web interface. A connection string uniquely identifies each device associated with the IoT Hub instance. It is under the “Primary Connection String” entry in the “Device detail” section.
- **shared_access_key**: You can obtain the shared access key from the “Primary key” entry in the “Device detail” section.
- **custom_msg_properties**: Use this property to embed custom key/value pairs in the MQTT messages sent from the device to Azure IoT. You can embed multiple key values separated by semicolons, as in this example:

```
custom_msg_properties = ex2: key1=value1;key2=value2;key3=value3;
```

| Note: | The `connection_str`, `shared_access_key`, and `custom_msg_properties` strings are each limited to 512 characters. |
|---|---|

#### 2.15.6.4 Using the Adapter

To use the Azure device client adapter in an application, set the `Gst-nvmsgbroker` plugin’s `proto-lib` property to the pathname of the adapter’s shared `library - libnvds_azure_proto.so` for the device client case, or `libnvds_azure_edge_proto.so` for the module client case.

The next step in using the adapter is to specify the connection details. The procedure for specifying connection details is different for the Azure device client and module client cases, as described in the following sections.

##### Connection Details for the Device Client Adapter

Set the plugin’s `conn-str` property to the full Azure connection string in the format:

```
HostName=<my-hub>.azure-
devices.net;DeviceId=<device_id>;SharedAccessKey=<my-policy-key>
```

Alternatively, you can specify the connection string details in the Azure configuration file:

```
[message-broker]
connection_str = HostName=<my-hub>.azure-
devices.net;DeviceId=<device_id>;SharedAccessKey=<my-policy-key>
```

#### Connection Details for the Module Client Adapter

Leave the connection string empty, since the Azure IoT Edge library automatically fetches the connection string from from the file `/etc/iotedge/config.yaml`.

Once the connection details have been configured, you can integrate the Azure device client and module client adapters into custom user code by using the `nvds_msgapi` interface to call its functions. Note the following points with regard to the functions defined by the interface:

- The connection string passed to `nvds_msgapi_connect()` may be NULL for both the Azure device client and the module client. For the device client the Azure configuration file has an option to specify a connection string. For the module client the connection string is always specified in `/etc/iotedge/config.yaml`.
- Both “send” functions use the topic name specified in the `Gst-nvmsgbroker` plugin’s property “topic.” It may be null.
- The application must call `nvds_msgapi_do_work()` after each call to `nvds_msgapi_send_async()`. The frequency of calls to `nvds_msgapi_do_work()` determines the rate at which messages waiting to be sent are processed.
- It is safe for multiple application threads to share connection handles. The library `libiothubclient` is thread-safe, so Azure protocol adapters need not implement separate locking mechanisms for functions calling this library directly.
- The Azure protocol adapters expects the client to manage usage and retirement of the connection handle. The client must ensure that once a handle is disconnected, it is not used for either a “send” call or a call to `nvds_msgapi_do_work()`. While the library attempts to ensure graceful failure if the application calls these functions with retired handles, it does not do so in a thread-safe manner.

### 2.15.6.5 Monitor Adapter Execution

The Azure device client and module client use different logging mechanisms.

#### Azure device client library log messages

The Azure device client adapter uses the `nvds_logger` framework to generate log messages which can help you monitor execution. The adapter generates separate logs for the `INFO`, `DEBUG`, and `ERROR` severity levels, as described in nvds_logger: Logging

[Framework](#). You can limit the generated log messages by setting the level at which log messages are filtered in the logging setup script.

| Note: | **If the severity level is set to `DEBUG`, the `nvds_logger` framework logs the entire contents of each message sent by the Azure device client protocol adapter.** |
|---|---|

**Azure Module Client Library Log Messages**

The log messages from the Azure module client adapter library are emitted to `stdout`, and the log output is captured in the docker/iotedge module logs.

#### 2.15.6.6 Message Topics and Routes

You can specify a message topic in a GStreamer property `topic`. However, the Azure device client and module client use the `topic` property in different ways.

The Azure device client does not support topics. Thus the value of the `topic` property is ignored, and you cannot use it to filter messages on Azure IoT Hub.

The Azure module client uses the `topic` property to determine the **route** of messages, i.e. how messages are passed within a system. For more information about message routes, see:

```
https://docs.microsoft.com/en-us/azure/iot-edge/module-
composition#declare-routes)
```

### 2.15.7 AMQP Protocol Adapter

DeepStream release 4.0 includes an AMQP protocol adapter that DeepStream applications can use out of the box to publish messages using AMQP 0-9-1 message protocol.

The AMQP protocol adapter shared library is located in the `deepstream` package at:

```
/opt/nvidia/deepstream/deepstream-4.0/lib/libnvds_amqp_proto.so
```

#### 2.15.7.1 Installing Dependencies

AMQP protocol adapter for DeepStream uses the `librabbitmq.so` library, built from `rabbitmq-c` (v0.8.0) for the underlying AMQP protocol implementation. To build the library, enter these commands:

```
git clone -b v0.8.0  --recursive https://github.com/alanxz/rabbitmq-
c.git
mkdir build && cd build
```

```
cmake ..
cmake --build .
```

To copy the built `librabbitmq.so` library to its final location, enter this command.

- For x86:

  ```
  sudo cp ./librabbitmq/librabbitmq.so.4 /usr/lib/
  ```

- For Jetson:

  ```
  sudo cp ./librabbitmq/librabbitmq.so.4 /usr/lib/aarch64-linux-gnu/
  ```

Install additional dependencies:

```
sudo apt-get install libglib2.0 libglib2.0-dev
```

#### AMQP broker

The AMQP protocol communicates with an AMQP 0-9-1 compliant message broker. If you do not have a functioning broker already, you can deploy one by installing the `rabbitmq-server` package, available at:

```
https://www.rabbitmq.com/install-debian.html
```

You can install this package on your local system or on the remote machine where you want the broker to be installed.

To install the package, enter the command:

```
sudo apt-get install rabbitmq-server
```

To determine whether the `rabbitmq` service is running, enter the command:

```
sudo service rabbitmq-server status
```

If `rabbitmq` is not running, enter this command to start it:

```
sudo service rabbitmq-server start
```

#### 2.15.7.2 Configure Adapter Settings

You can place AMQP protocol adapter specific information in a custom configuration named, for example, `cfg_amqp.txt`. Here is an example of configuration file entries for an AMQP broker installed on the local machine:

```
[message-broker]
hostname = localhost
username = guest
password = guest
port = 5672
exchange = amq.topic
topic = topicname
```

The properties in the configuration file are:

- **hostname**: Hostname of the host on which the AMQP broker is installed
- **username**: Username used to log in to the broker
- **password**: Password used to log in to the broker
- **port**: Port used to communicate with the AMQP broker
- **exchange**: Name of the exchange on which to publish messages
- **topic**: Message topic

#### 2.15.7.3 Using the adapter

To use the AMQP protocol client adapter in a DeepStream application, set the `Gst-nvmsgbroker` plugin's `proto-lib` property to the pathname of the adapter's shared library, `libnvds_amqp_proto.so`.

```
proto-lib = <path to libnvds_amqp_proto.so>
```

You can specify the AMQP connection details in the AMQP adapter specific configuration file (e.g., `cfg_amqp.txt`) as described above. This is the recommended method. The path to the AMQP configuration file is specified by the Gst property `config`:

```
config = <path to cfg_amqp.txt>
```

Alternatively, you can specify the AMQP protocol's hostname, port number, and username in the Gst plugin's `conn-str` property, and specify the password in the configuration file. In the Gst properties:

```
conn-str = hostname;5672;username
config   = <pathname of AMQP configuration file>
```

In the AMPQ configuration file:

```
[message-broker]
password = <password>
```

You can set the `Gst-nvmsgbroker` plugin's `topic` property to specify the message topic.

```
topic =  <topicname>
```

Alternatively, you can specify a topic in the AMQP configuration file (cfg_amqp.txt). In the Gst properties, set:

```
config    = <path to cfg_amqp.txt>
```

In the AMQP configuration file:

```
[message-broker]
Topic = topicname
```

#### 2.15.7.4 Programmatic Integration

Once you have configured the connection, you can integrate the AMQP protocol adapter into your application by using the `nvds_msgapi` interface to call its functions. Note the following points about the functions defined by the interface:

- The connection string passed to `nvds_msgapi_connect()` has the format `Hostname;<port>;username`.
- For both "send" functions, the topic name is specified either by the `Gst-nvmsgbroker` plugin's `topic` property or by the `topic` parameter in the AMQP configuration file.
- The application must call `nvds_msgapi_do_work()` after each call to `nvds_msgapi_send_async()`. The frequency of calls to `nvds_msgapi_do_work()` determines the rate at which messages waiting to be sent are processed.

The AMQP protocol adapter expects the client to manage usage and retirement of the connection handle. The client must ensure that once a handle is disconnected, it is not used for either a "send" call or a call to `nvds_msgapi_do_work()`. While the library attempts to ensure graceful failure, if the application calls these functions with retired handles, it does not do so in a thread-safe manner.

**Note:** As stated at https://github.com/alanxz/rabbitmq-c#threading, you cannot share a socket, an `amqp_connection_state_t`, or a channel between threads using the `librabbitmq` library. This library is designed for use by event-driven, single-threaded applications, and does not yet meet the requirements of threaded applications.

To deal with this limitation, your application must open an AMQP connection (and an associated socket) per thread. If it needs to access a single AMQP connection or any of its channels from more than one thread,

you must implement an appropriate locking mechanism. It is generally simpler to have a connection dedicated to each thread.

#### 2.15.7.5 Monitor Adapter Execution

The AMQP protocol adapter uses the `nvds_logger` framework to generate log messages which can help you monitor execution. The adapter generates separate logs for the `INFO`, `DEBUG`, and `ERROR` severity levels, as described in nvds_logger: Logging Framework. You can limit the log messages being generated by setting the level at which log messages are filtered in the logging setup script.

**Note:** If the severity level is set to `DEBUG`, `nvds_logger` logs the entire contents of each message sent by the AMQP protocol adapter.

### 2.15.8 nvds_logger: Logging Framework

DeepStream provides a logging framework named `nvds_logger`. The Kafka protocol adapter uses this framework to generate a run time log. `nvds_logger` is based on `syslog`, and offers many related features, including:

- Choice of priorities (log levels)
- Log filtering and redirection
- Shared logging across different DeepStream instances running concurrently
- Log retirement and management using `logrotate`
- Cross-platform support

#### 2.15.8.1 Enabling Logging

To enable logging, run the `setup_nvds_logger.sh` script. Note that this script must be run with `sudo`. You may have to modify the permissions associated with this script to make it executable.

The script accepts an optional parameter specifying the pathname of log file to be written. By default, the pathname is `/tmp/nvds/ds.log`.

Once logging is enabled, you can access the generated log messages by reading the log file.

By default, you must have `sudo` permissions to read the log file. Standard techniques for syslog-based logging configuration can eliminate this requirement.

#### 2.15.8.2 Filtering Logs

`nvds_logger` allows logs to be associate with a severity level similar to that which syslog offers. You can filter log messages based on severity level by modifying the setup script. By default, the script enables logging for messages at the `INFO` level (level 6) and above. You can modify this as outlined in the comments in the script:

```
# Modify log severity level as required and rerun this script
#              0       Emergency: system is unusable
#              1       Alert: action must be taken immediately
#              2       Critical: critical conditions
#              3       Error: error conditions
#              4       Warning: warning conditions
#              5       Notice: normal but significant condition
#              6       Informational: informational messages
#              7       Debug: debug-level messages
# refer https://tools.ietf.org/html/rfc5424.html for more information

echo "if (\$syslogtag contains 'DSLOG') and (\$syslogseverity <= 6)
then $nvdslogfilepath" >> 11-nvds.conf
```

#### 2.15.8.3 Retiring and Managing Logs

It is recommended that you limit the size of log files by retiring them periodically. `logrotate` is a popular utility for this purpose. You can use it in cron jobs so that the log files are automatically archived periodically, and are discarded after a desired interval.

#### 2.15.8.4 Generating Logs

You can implement modules that use the logger by including `sources/includes/nvds_logger.h` in the source code and linking to the `libnvds_logger.so` library.

Generating logs programmatically involves three steps:

1. Call `nvds_log_open()` before you write any log messages.
2. Call `nvds_log()` to write log messages.
3. Call `nvds_log_close()` upon completion to flush and close the logs.

Note the `nvds_logger` is a process-based logging mechanism, so the recommended procedure is to call `nvds_log_open()` from the main application routine rather than the individual plugins. Similarly, call `nvds_log_close()` from the main application when it shuts down the application before exit.

# 3.0 METADATA IN THE DEEPSTREAM SDK

Each Gst Buffer has associated metadata. The DeepStream SDK attaches the DeepStream metadata object, `NvDsBatchMeta`, described in the following sections.

## 3.1 NVDSBATCHMETA: BASIC METADATA STRUCTURE

DeepStream uses an extensible standard structure for metadata. The basic metadata structure `NvDsBatchMeta` starts with batch level metadata, created inside the Gst-nvstreammux plugin. Subsidiary metadata structures hold frame, object, classifier, and label data. DeepStream also provides a mechanism for adding user-specific metadata at the batch, frame, or object level.

DeepStream attaches metadata to a Gst Buffer by attaching an `NvDsBatchMeta` structure and setting `GstNvDsMetaType.meta_type` to `NVDS_BATCH_GST_META` in the Gst-nvstreammux plugin. When your application processes the Gst Buffer, it can iterate over the attached metadata to find `NVDS_BATCH_GST_META`.

The function `gst_buffer_get_nvds_batch_meta()` extracts `NvDsBatchMeta` from the Gst Buffer. (See the declaration in `sources/include/gstnvdsmeta.h`.) See the `deepstream-test1` sample application for an example of this function's usage. For more details, see *NVIDIA DeepStream SDK API Reference*.

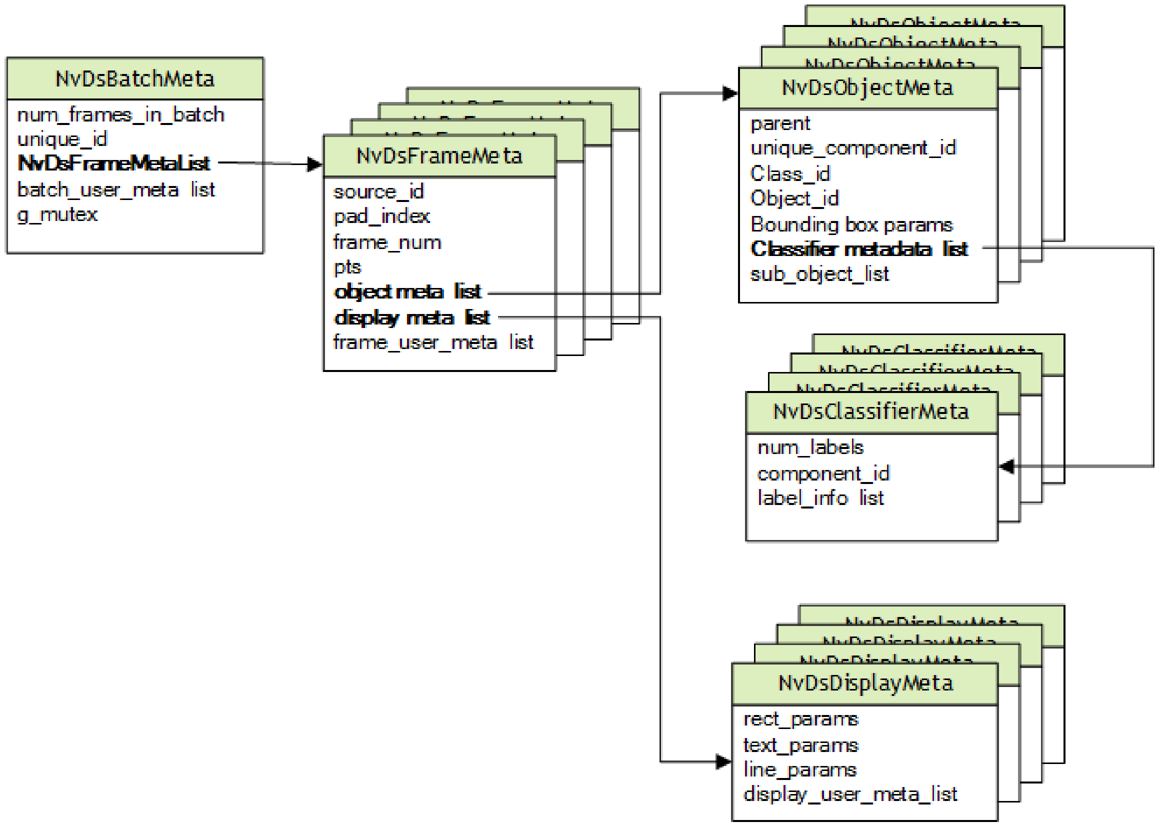


Figure 16. DeepStream metadata hierarchy

## 3.2 USER/CUSTOM METADATA ADDITION INSIDE NVDSBATCHMETA

To attach user-specific metadata at the batch, frame, or object level within `NvDsBatchMeta`, you must acquire an instance of `NvDsUserMeta` from the user meta pool by calling `nvds_acquire_user_meta_from_pool()`. (See `sources/includes/nvdsmeta.h` for details.) Then you must initialize `NvDsUserMeta`. The members you must set are `user_meta_data`, `meta_type`, `copy_func`, and `release_func`.

For more details, see the sample application source code in `sources/apps/sample_apps/deepstream-user-metadata-test/deepstream_user_metadata_app.c`.

## 3.3 ADDING CUSTOM META IN GST PLUGINS UPSTREAM FROM GST-NVSTREAMMUX

The DeepStream SDK creates batch level metadata in the Gst-nvstreammux plugin. It holds `NvDsBatchMeta` metadata in a hierarchy of batches, frames within batches, and objects within frames.

### To add metadata to the plugin before Gst-nvstreammux

*This procedure introduces metadata to the DeepStream pipeline at a plugin before Gst-nvstreammux.*

1. Set the plugin's following members of the plugin's `NvDsUserMeta` structure:
   - `copy_func`
   - `free_func`
   - `meta_type`
   - `gst_to_nvds_meta_transform_func`
   - `gst_to_nvds_meta_release_func`
2. Attach the metadata by calling `gst_buffer_add_nvds_meta()` and set the meta_type in the `NvDsMeta` instance returned by `gst_buffer_add_nvds_meta()`.
3. The Gst-nvstreammux plugin transforms the input gst-meta created in step 2 from the Gst Buffer into an `NvDsUserMeta` object associated with the corresponding `NvDsFrameMeta` object. It adds this object to the `frame_user_data` list.
4. Search the `frame_user_meta` list in the `NvDsFrameMeta` object for the `meta_type` that was set in step 2, and access the attached metadata.

See the sample application source code in `sources/apps/sample_apps/deepstream-gst-metadata-test/deepstream_gst_metadata.c` for more details. If gst meta is not attached with `gst_buffer_add_nvds_meta()`,it is not transformed into DeepStream metadata. It is still be available in the Gst Buffer, though.

# 4.0 IPLUGIN INTERFACE

DeepStream 4.0 supports TensorRT™ plugins for custom layers. The `Gst-nvinfer` plugin now has support for the `IPluginV2` and `IPluginCreator` interface, introduced in TensorRT 5.0. For caffemodels and for backward compatibility with existing plugins, it also supports the following interfaces:

- `nvinfer1::IPluginFactory`
- `nvuffparser::IPluginFactory`
- `nvuffparser::IPluginFactoryExt`
- `nvcaffeparser1::IPluginFactory`
- `nvcaffeparser1::IPluginFactoryExt`
- `nvcaffeparser1::IPluginFactoryV2`

See the TensorRT documentation for details on new and deprecated plugin interfaces.

## 4.1 HOW TO USE IPLUGINCREATOR

To use the new `IPluginCreator` interface you must implement the interface in an independent custom library. This library must be passed to the `Gst-nvinfer` plugin through its configuration file by specifying the library's pathname with the `custom-lib-path` key.

`Gst-nvinfer` opens the library with `dlopen()`, which causes the plugin to be registered with TensorRT. There is no further direct interaction between the custom library and `Gst-nvinfer`. TensorRT calls the custom plugin functions as required.

The SSD sample provided with the SDK provides an example of using the `IPluginV2` and `IPluginCreator` interface. This sample has been adapted from TensorRT.

## 4.2 HOW TO USE IPLUGINFACTORY

To use the `IPluginFactory` interface, you must implement the interface in an independent custom library. Pass this library to the `Gst-nvinfer` plugin through the plugin's configuration file by specifying the library's pathname in the `custom-lib-path` key. The custom library must implement the applicable functions:

- `NvDsInferPluginFactoryCaffeGet`
- `NvDsInferPluginFactoryCaffeDestroy`
- `NvDsInferPluginFactoryUffGet`
- `NvDsInferPluginFactoryUffDestroy`
- `NvDsInferPluginFactoryRuntimeGet`
- `NvDsInferPluginFactoryRuntimeDestroy`

These structures are defined in `nvdsinfer_custom_impl.h`. The function definitions must be named as in the header file. `Gst-nvinfer` opens the custom library with `dlopen()` and looks for the names.

#### For Caffe Files

During parsing and building of a caffe network, `Gst-nvinfer` looks for `NvDsInferPluginFactoryCaffeGet`. If found, it calls the function to get the `IPluginFactory` instance. Depending on the type of `IPluginFactory` returned, `Gst-nvinfer` sets the factory using one of the `ICaffeParser` interface's methods `setPluginFactory()`, `setPluginFactoryExt()`, or `setPluginFactoryV2()`.

After the network has been built and serialized, `Gst-nvinfer` looks for `NvDsInferPluginFactoryCaffeDestroy` and calls it to destroy the `IPluginFactory` instance.

#### For Uff Files

During parsing and building of a caffe network, `Gst-nvinfer` looks for `NvDsInferPluginFactoryUffGet`. If found, it calls the function to get the `IPluginFactory` instance. Depending on the type of `IPluginFactory` returned, `Gst-nvinfer` sets the factory using one of the `IUffParser` inteface's methods `setPluginFactory()` or `setPluginFactoryExt()`.

After the network has been built and serialized, `Gst-nvinfer` looks for `NvDsInferPluginFactoryUffDestroy` and calls it to destroy the `IPluginFactory` instance.

#### During Deserialization

If deserializing the models requires an instance of `NvInfer1::IPluginFactory`, the custom library must also implement `NvDsInferPluginFactoryRuntimeGet()` and optionally `NvDsInferPluginFactoryRuntimeDestroy()`. During deserialization, `Gst-nvinfer` calls the library's `NvDsInferPluginFactoryRuntimeGet()`

`function` to get the `IPluginFactory` instance, then calls `NvDsInferPluginFactoryRuntimeDestroy` to destroy the instance if it finds that function during `Gst-nvinfer` deinitialization.

The `FasterRCNN` sample provided with the SDK provides an example of using the `IPluginV2+nvcaffeparser1::IPluginFactoryV2` interface with DeepStream. This sample has been adapted from TensorRT. It also provides an example of using the legacy `IPlugin + nvcaffeparser1::IPluginFactory + Gst-nvinfer 1::IPluginFactory` interface for backward compatibility.

# 5.0 DOCKER CONTAINERS

DeepStream 4.0 provides Docker containers for both dGPU and Jetson platforms. These containers provide a convenient, out-of-the-box way to deploy DeepStream applications by packaging all associated dependencies within the container. The associated Docker images are hosted on the NVIDIA container registry in the NGC web portal at https://ngc.nvidia.com. They leverage the nvidia-docker package, which enables access to GPU resources from containers, as required by DeepStream applications. The rest of this section describes the features supported by the DeepStream Docker container for the dGPU and Jetson platforms.

| Note: | **The DeepStream 4.0 containers for dGPU and Jetson are distinct, so you must take care to get the right image for your platform.** |
|---|---|

## 5.1 A DOCKER CONTAINER FOR DGPU

The Deeptream 4.0 container for dGPU is kept in the “Inference” section of the NGC web portal. The “Container” page gives instructions for pulling and running the container, along with a description of its contents.

Unlike the container in DeepStream 3.0, the dGPU DeepStream 4.0 container supports DeepStream application development within the container. It contains the same build tools and development libraries as the DeepStream 4.0 SDK.

In a typical scenario, you build, execute and debug a DeepStream application within the DeepStream container. Once your application is ready, you can create your own Docker container holding your application files (binaries, libraries, models, configuration file, etc.), using the DeepStream 4.0 container as a base image and adding your application-specific files to it. Here is a snippet which shows how a Dockerfile for creating your own Docker container might look:

```
FROM docker pull nvcr.io/nvidia/deepstream:4.0-19.07
ADD mydsapp  /root/apps/
# To get video driver libraries at runtime (libnvidia-
encode.so/libnvcuvid.so)
ENV NVIDIA_DRIVER_CAPABILITIES $NVIDIA_DRIVER_CAPABILITIES,video
```

This Dockerfile copies your application (from directory `mydsapp`) into the container (pathname `/root/apps`). Note that you must ensure that the DeepStream 4.0 image location from NGC is accurate.

## 5.2 A DOCKER CONTAINER FOR JETSON

DeepStream 4.0 supports containers on the Jetson platform. As of JetPack release 4.2.1, NVIDIA Container Runtime for Jetson has been added, allowing you to run GPU-enabled containers on Jetson devices. Leveraging this capability, DeepStream 4.0 can be run inside containers on Jetson devices using Docker images made available on NGC.

A DeepStream 4.0 container for Jetson is present in the "Inference" section of the NGC container registry. Pull the container and execute it according to the instructions on the container page on NGC.

The DeepStream container expects CUDA, TensorRT, and VisionWorks to be installed on the Jetson device, since it is mounted within the container from the host. Make sure that these utilities are installed using JetPack on your Jetson prior to launching the DeepStream container.

Note that the Jetson Docker containers are for deployment only. They do not support DeepStream software development within a container. You can build applications natively on the Jetson target and create containers for them by adding binaries to your Docker images. Alternatively, you can generate Jetson containers from your workstation using instructions in the NVIDIA Container Runtime for Jetson documentation. See the section "Building Jetson Containers on an x86 Workstation."

# 6.0 TROUBLESHOOTING

If you run into to trouble while using DeepStream, consider the following solutions.

- **Problem**: You are migrating from DeepStream 3.0 to DeepStream 4.0.

  **Solution**: You must clean up the DeepStream 3.0 libraries and binaries. The one of these commands to clean up:

  - **For dGPU**: Enter this command:

    ```
    $ sudo rm -rf /usr/local/deepstream /usr/lib/x86_64-linux-
    gnu/gstreamer-1.0/libnvdsgst_* /usr/lib/x86_64-linux-
    gnu/gstreamer-1.0/libgstnv* /usr/bin/deepstream* /usr/lib/x86_64-
    linux-gnu/libv4l/plugins/libcuvidv4l2_plugin.so
    ```

  - **For Jetson**: Flash the target device with the latest release of JetPack.

- **Problem**: “NvDsBatchMeta not found for input buffer” error while running DeepStream pipeline.

  **Solution**: The Gst-nvstreammux plugin is not in the pipeline. Starting with DeepStream 4.0, Gst-nvstreammux is a required plugin.

  This is an example pipeline:

  Gst-nvv4l2decoder → Gst-nvstreammux → Gst-nvinfer → Gst-nvtracker → Gst-nvmultistreamtiler → Gst-nvvideoconvert → Gst-nvosd → Gst-nveglglessink

- **Problem**: The DeepStream reference application fails to launch, or any plugin fails to load.

  **Solution**: Try clearing the GStreamer cache by running the command:

  ```
  $ rm -rf ${HOME}/.cache/gstreamer-1.0
  ```

  Also run this command if there is an issue with loading any of the plugins. Warnings or errors for failing plugins are displayed on the terminal.

```
$ gst-inspect-1.0
```

Then run this command to find missing dependencies:

```
$ldd <plugin>.so
```

Where <plugin> is the name of the plugin that failed to load.

- **Problem**: Application fails to run when the neural network is changed.

  **Solution**: Be sure that the network parameters are updated for the corresponding [GIE] group in the configuration file (e.g. `source30_720p_dec_infer-resnet_tiled_display_int8.txt`). Also be sure that the `Gst-nvinfer` plugin's configuration file is updated accordingly.

  When the model is changed, make sure that the application is not using old engine files.

- **Problem**: The DeepStream application is running slowly (Jetson only).

  **Solution**: Ensure that Jetson clocks are set high. Run these commands to set Jetson clocks high.

```
$ sudo nvpmodel -m <mode> --for MAX perf and power mode is 0
$ sudo jetson_clocks
```

- **Problem**: The DeepStream application is running slowly.

  **Solution 1**: One of the plugins in the pipeline may be running slowly.

  You can measure the latency of each plugin in the pipeline to determine whether one of them is slow.

  - To enable frame latency measurement, run this command on the console:

```
$ export NVDS_ENABLE_LATENCY_MEASUREMENT=1
```

  - To enable latency for all plugins, run this command on the console:

```
$ export NVDS_ENABLE_COMPONENT_LATENCY_MEASUREMENT=1
```

  **Solution 2** (dGPU only): Ensure that your GPU card is in the PCI slot with the greatest bus width.

  **Solution 3**: In the configuration file's `[streammux]` group, set `batched-push-timeout` to (1/max_fps).

**Solution 4**: In the configuration file's `[streammux]` group, set `width` and `height` to the stream's resolution.

**Solution 5**: For RTSP streaming input, in the configuration file's `[streammux]` group, set `live-source=1`. Also make sure that all `[sink#]` groups have the `sync` property set to 0.

**Solution 6**: If secondary inferencing is enabled, try to increase `batch-size` in the the configuration file's `[secondary-gie#]` group in case the number of objects to be inferred is greater than the `batch-size` setting.

**Solution 7**: On Jetson, use Gst-nvoverlaysink instead of Gst-nveglglessink as nveglglessink requires GPU utilization.

**Solution 8**: If the GPU is bottlenecking performance, try increasing the interval at which th primary detector infers on input frames by modifying the `interval` property of `[primary-gie]` group in the application configuration, or the `interval` property of the Gst-nvinfer configuration file

**Solution 9**: If the elements in the pipeline are getting starved for buffers (you can check if CPU/GPU utilization is low), try increasing the number of buffers allocated by the decoder by setting the `num-extra-surfaces` property of the `[source#]` group in the application or the `num-extra-surfaces` property of Gst-nvv4l2decoder element.

**Solution 10**: If you are running the application inside docker/on console and it delivers low FPS, set `qos=0` in the configuration file's `[sink0]` group.

The issue is caused by initial load. With `qos` set to 1, as the property's default value in the `[sink0]` group, decodebin starts dropping frames.

- **Problem**: On NVIDIA® Jetson Nano™, `deepstream-segmentation-test` starts as expected, but crashes after a few minutes. The system reboots.

  **Solution** : NVIDIA recommends that you power the Jetson module through the DC power connector when running this app. USB adapters may not be able to handle the transients.

- **Problem**: Errors occur when `deepstream-app` is run with a number of streams greater than 100. For example:

```
(deepstream-app:15751): GStreamer-CRITICAL **: 19:25:29.810:
gst_poll_write_control: assertion 'set != NULL' failed.
```

  **Solution**: run this command on the console:

```
ulimit -Sn 4096
```

  Then run `deepstream-app` again.

### Notice



### Trademarks



### Copyright