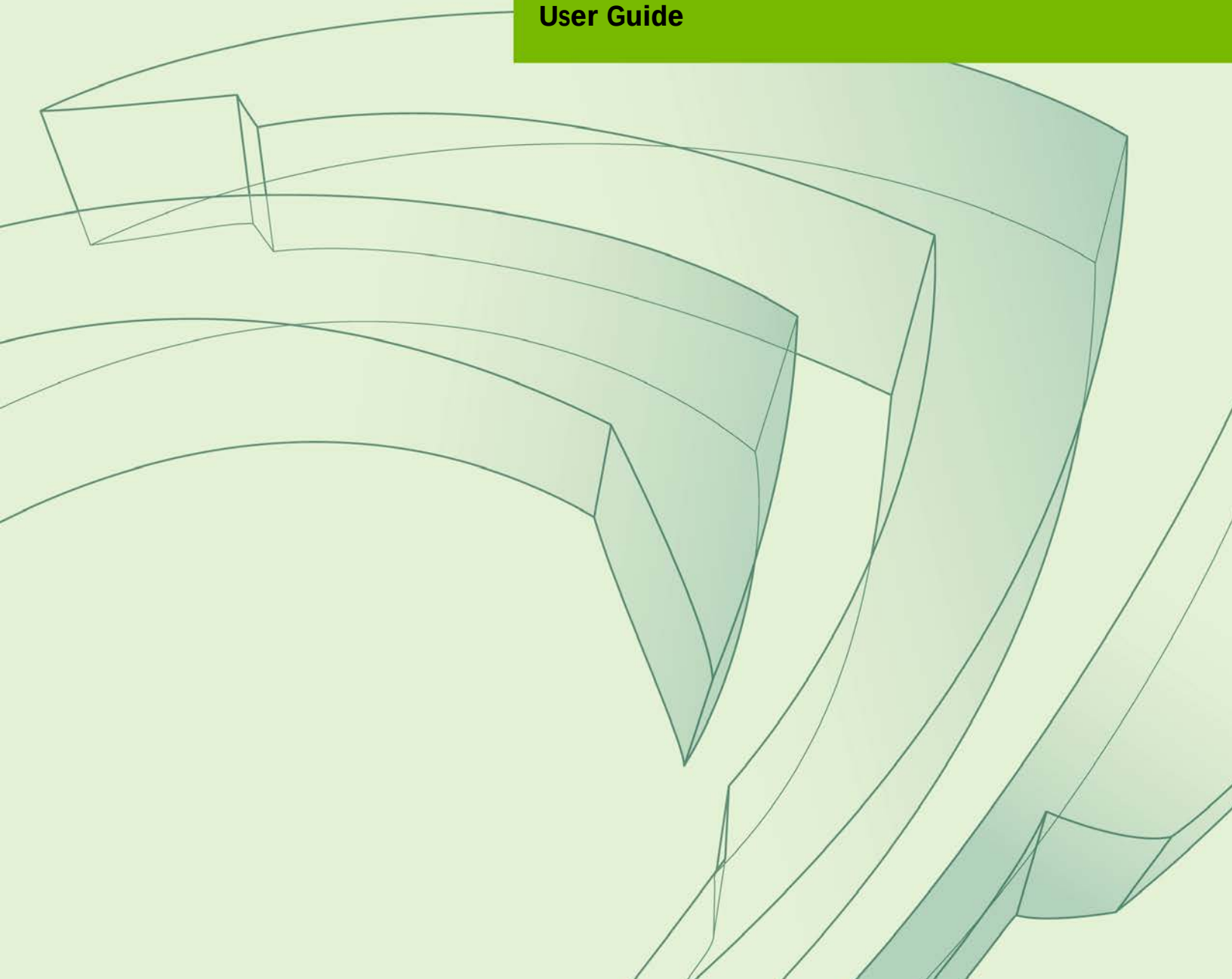
DEEPSTREAM SDK

DU-08633-001_v04 | December 2017

User Guide



DOCUMENT CHANGE HISTORY

DU-08633-001_v04

Version	Date	Authors	Description of Change
01	05/26/2017	RX	Add workflow and instructions for two samples
02	06/09/2017	XL	Added feature descriptions
03	06/23/2017	BP	Added building and running instructions for the third sample
04	12/21/2017	BP	Added Section 4.4 nvSmartDecInfer

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF CODE

Chapter 1. INTRODUCTION

One of the biggest challenges of AI is understanding video content. Applications with video content are endless: live stream content filtering, face detection and recognition, traffic surveillance, etc. Use of deep-learning based techniques in conjunction with GPU architectures have made tackling several of these use cases feasible and realistic. Even so, use of deep learning for video is computationally expensive. The requirements are even more stringent for live steaming video use cases since they often involve real time analysis requiring custom logic that customers can deploy using flexible workflows.

DeepStream provides an easy-to-use and high-performance SDK for video content analysis, which simplifies development of high-performance video analytics applications powered by deep learning. DeepStream enables customer to make optimum use of underlying GPU architectures, including hardware decoding support, thereby achieving high levels of efficiency, performance, and scale. Furthermore, DeepStream provides a flexible plug-in mechanism for the user to incorporate their custom functionality to video analytics applications to meet their unique needs.

DeepStream provides a high-level C++ API for GPU-accelerated video decoding, inference. DeepStream is built on the top of NVCODEC and NVIDIA® TensorRT™, which are responsible for video decoding and deep learning inference, respectively.

The following are the key features of DeepStream:

- Deploys widely-used neural network models such as GoogleNet, AlexNet, etc. for real-time image classification and object detection.
- Supports neural networks implemented using Caffe and TensorFlow frameworks.

- Supports common video formats: H.264, HEVC/H.265, MPEG-2, MPEG-4, VP9, and VC1[1].
- Takes inference with full precision float type (FP32) or optimized precision[2] (FP16 and INT8).
- Provides flexible analytics workflow which allows users to implement a plug-in to define their inference workflow.

[1] Supported format is dependent on specific GPU model. The support matrix can be found at https://developer.nvidia.com/nvidia-video-codec-sdk.
[2] Optimized precision inference requires hardware support, the support device can be found in the TensorRT manual.

Chapter 2. INSTALLATION

2.1 SYSTEM REQUIREMENTS

DeepStream has the following software dependencies:

- Ubuntu 16.04 LTS (with GCC 5.4)
- NVIDIA Display Driver R384
- NVIDIA VideoSDK 8.0
- NVIDIA CUDA® 9.0
- cuDNN 7 & TensorRT 3.0

NVIDIA recommends that DeepStream be run on a hardware platform with an NVIDIA Tesla® P4 or P40 graphics card. While the processor and memory requirements would be application dependent, the hardware platform used for execution of the samples shipped as part of the DeepStream SDK was an Intel Broadwell E5-2690 v4@2.60GHz 3.5GHz Turbo, with 128 GB System RAM.

2.2 DIRECTORY LAYOUT

The DeepStream SDK consists of two main parts: the library and the workflow demonstration samples. The installed DeepStream package includes the directories /lib, /include, /doc, and /samples.

- The dynamic library **`libdeepstream.so`** is in the /lib directory.
- There are two header files: **`deepStream.h`** and **`module.h`**.
  - **`deepStream.h`** includes the definition of decoded output, supported data type, inference parameters, profiler class, and DeepStream worker, as well as the declaration of functions.

  - **`module.h`** is the header file for plug-in implementation. This file is not mandatory for applications without plug-ins.

- The /samples folder includes examples of decoding, decoding and inference, and plug-in implementations. More information can be found in the Samples chapter.

Chapter 3. WORKFLOW

Live video stream analysis requires real-time decoding and neural network inference. For the decoding work, multiple threads execute in parallel, and feed various input streams to the GPU hardware decoder. For the inference part, one main thread handles all batched inference tasks by calling the TensorRT inference engine. The user can configure the maximum batch size, up to the number of decoded frames that may be fed to the rest of the pipeline. The plug-in system allows users to add more complex workflows into the pipeline.

3.1 DECODING AND INFERENCE WORKFLOW

The input to DeepStream consists of multiple video channels from local video files (H.264, HEVC, etc.) or online streaming videos. The DeepStream workflow is straightforward, and includes the steps shown below.

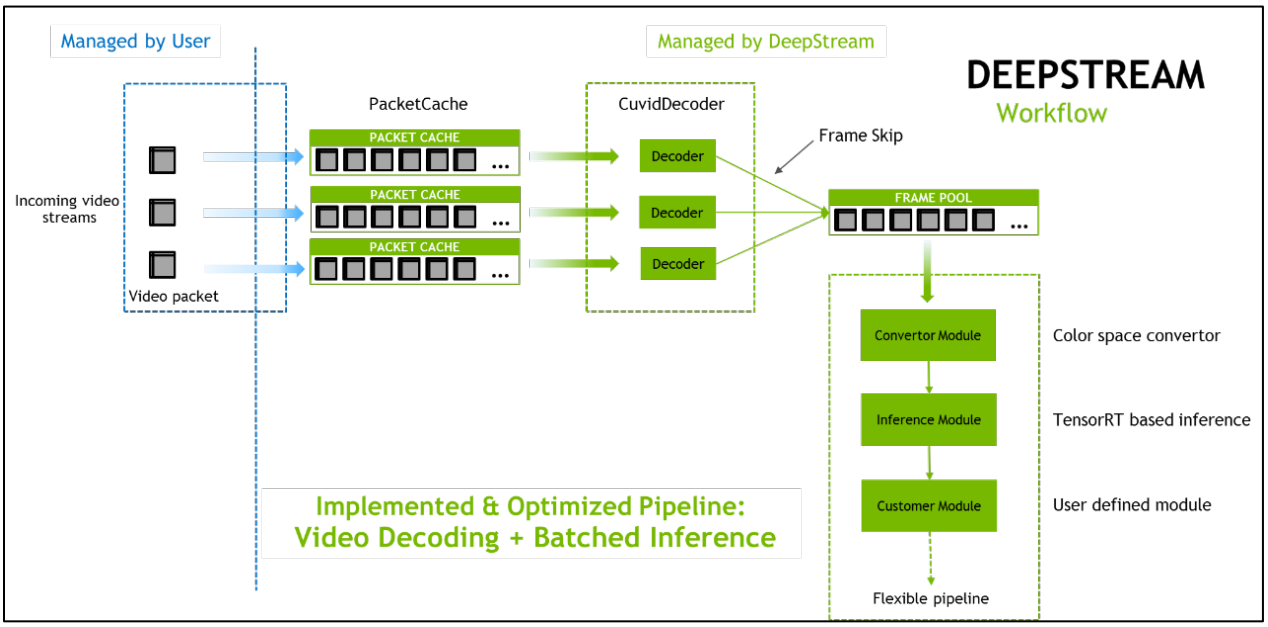


Figure 3-1: DeepStream Workflow

1. A DeviceWorker should be created specifying the number of input video channels, the GPU device ID to be used for executing the pipeline, and using an optional parameter to specify the maximum batch size of decoded frames passed to inference module.
2. Videos are parsed and the resulting packets are fed into appropriate channels within the packet cache maintained by the DeviceWorker.
3. DeviceWorker begins to decode once it gets the input packets and analyzes the decoded frame.
   - A frame pool in DeviceWorker is used for gathering the decoded frames from all decoders. The analysis pipeline in DeviceWorker fetches a batch of frames for the next process.
   - Typically, the analysis pipeline in DeviceWorker consists of a pair of pre-defined modules, for "color space conversion" and inference.
   - All the decoders share the same CUDA context (primary context) within multiple host threads. The analysis pipeline is in one host thread.

From a programming viewpoint, the user should build the pipeline by defining and configuring the DeviceWorker, and then executing the workflow by calling DeviceWorker->start(). The pipeline will be stopped by DeviceWorker->stop().

3.2 PLUG-IN MECHANISM

In DeepStream, the workflow that follows decoding is called the "analysis pipeline". Besides the pre-defined *color space converter* and *TensorRT-based inference* modules, the user can define their own modules. There are two main classes: IStreamTensor, which defines input and output format of this module, and IModule, which is the implementation of the module mechanism.

3.2.1 IStreamTensor

All modules use the IStreamTensor format, which allows the data to have a shape consisting of up to four dimensions. The tensor type can be float, nv12_frame, object coordination, or a user-defined type.

Code 3-1: Tensor Type Definitions

```
typedef enum {
     FLOAT_TENSOR      = 0,        //!< float tensor
     NV12_FRAME        = 1,        //!< nv12 frames
     OBJ_COORD         = 2,        //!< coords of object
     CUSTOMER_TYPE     = 3         //!< user-defined type
} TENSOR_TYPE;
```

The data can be stored in the GPU or CPU, as specified by MEMORY_TYPE.

Code 3-2: Memory Type Definitions

```
typedef enum {
     GPU_DATA = 0,    //!< gpu data
     CPU_DATA = 1     //!< cpu data
} MEMORY_TYPE;
```

Data and information are fetched from the previous module as IStreamTensor. The functions in IStreamTensor are listed in Code 3-3.

Code 3-3: IStreamTensor Member Functions

```
virtual void *getGpuData() = 0;
virtual const void *getConstGpuData() = 0;
virtual void *getCpuData() = 0;
virtual const void *getConstCpuData() = 0;
virtual size_t getElemSize() const = 0;
virtual MEMORY_TYPE getMemoryType() const = 0;
virtual TENSOR_TYPE getTensorType() const = 0;
virtual std::vector<int>& getShape() = 0;
```

```
virtual std::vector<TRACE_INFO > getTraceInfos() = 0;
virtual int getMaxBatch() const = 0;

virtual void setShape(const std::vector<int>& shape) = 0;
virtual void setShape(const int n, const int c, const int h, const
int w) = 0;
virtual void setTraceInfo(std::vector<TRACE_INFO >& vTraceInfos) = 0;
virtual void destroy() = 0;
```

For the input of module, you can use functions with the "get" prefix to acquire the information and data. For the output of the module, the "set" prefix function helps to set the information. IStreamTensor should be created with createStreamTensor.

Code 3-4: createStreamTensor Global Function

```
inline IStreamTensor *createStreamTensor(const int nMaxLen, const
size_t nElemSize, const TENSOR_TYPE Ttype, const MEMORY_TYPE Mtype,
const int deviceId) {
   return reinterpret_cast<IStreamTensor
  *>(createStreamTensorInternal(nMaxLen, nElemSize, Ttype, Mtype,
  deviceId));
}
```

Be sure to pass or update the Trace information, which includes information about the frames index, video index, etc. This can be used for tracing attributes of detected objects.

3.2.2 Module

Each module should include initialize(), execute(), and destroy(). The module can be added into DeviceWorker by IDeviceWorker->addCustomerTask (defined in **deepStream.h**). The DeepStream system executes "initialize" when adding a task, and calls the "execute" function of each module serially when executing DeviceWorker->start(). The "destroy" function is executed when calling DeviceWorker->destroy().

Chapter 4. SAMPLES

DeepStream provides four samples (located in the /samples directory):

- **decPerf**
  Uses DeepStream to test the performance of video decoding.
- **nvDecInfer_classification**
  Uses DeepStream to test the video decoding and inference for classification.
- **nvDecInfer_detection**
  Uses DeepStream to test the video decoding and inference for detection.
- **nvSmartDecInfer_detection**
  Illustrates the "smart decoding" feature whereby only I-frames in the input video are retrieved for decoding and inference.

The /samples directory includes the following subdirectories:

- */common*
  Contains header files shared by all samples
- */data/model*
  Contains GoogleNet and Resnet10 pre-trained models for classification and detection use cases respectively
- */data/video*
  Contains videos to use with the samples

4.1 DECPERF

The decPerf sample is used to test the performance of GPU hardware decoding and shows the following pre-analysis workflow:

- feeding packets into the pipeline
- adding the decoding task
- profiling the decoding performance

4.1.1 DeviceWorker

- DataProvider and FileDataProvider Class Definitions

  The definition of these two classes is in the header file **dataProvider.h**. FileDataProvider is used to load a packet from a video file. The dataProvider interface is shown in Code 4-1.

  Code 4-1: dataProvider Interface

```
// get data from the data provider. If return false, it means no
more data can be load.
bool getData(uint8_t **ppBuf, int *pnBuf);
// reset the dataProvider to reload from the video file from the
beginning.
void reload();
```

- Creating a DeviceWorker

  DeviceWorker is responsible for the entire DeepStream workflow. It includes multi-channel decoding and maintains the analysis pipeline. DeviceWorker is created by the createDeviceWorker function with the number of channels (g_nChannels) and GPU ID (g_devID) as parameters. There is an optional third parameter (maximum batch size) that specifies the maximum number of frames that DeepStream shall attempt to batch together for processing in the pipeline. The implementation uses this option to set the size of the "frame pool" (shown in Figure 3-1), so that the decoder copies up to as many decoded frames into the pool before they are processed by the analysis thread. Batching is particularly useful in use cases involving low channel count where processing one frame from each input video at a time can be inefficient. If the parameter is left out, it shall default to the number of channels.

  Code 4-2: Creating DeviceWorker

```
// Create a deviceWorker on a GPU device, the user needs to set the
channel number.
```

```
IDeviceWorker *pDeviceWorker = createDeviceWorker(g_nChannels,
g_devID, g_maxBatchSize);
```

- Adding Decoding Task in the DeviceWorker

  There is only one parameter for DeviceWorker->addDecodeTask,

  Code 4-3: Add decoding task

```
// Add decode task, the parameter is the format of codec.
pDeviceWorker->addDecodeTask(cudaVideoCodec_H264);
```

- Running the DeviceWorker

  After adding the decode task, DeviceWorker will create N decoders (N == g_nChannels). The decoding works will be submitted in parallel to the GPU decoder through multiple host threads.

  Code 4-4: DeviceWorker (decoding and analysis pipeline) start and stop

```
// Start and stop the DeviceWorker.
pDeviceWorker->start();
pDeviceWorker->stop();
```

- Pushing Packets into DeviceWorker

  DeviceWorker will be in a suspended state until the user pushes video packets into it. DecPerf provides an example by using userPushPacket() defined in the sample.

  Code 4-5: Pushing Video Packets to DeviceWorker

```
// User push video packets into a packet cache
std::vector<std::thread > vUserThreads;
for (int i = 0; i < g_nChannels; ++i) {
    vUserThreads.push_back( std::thread(userPushPacket,
                                        vpDataProviders[i],
                                        pDeviceWorker,
                                        i
                                        ) );
}

// wait for user push threads
for (auto& th : vUserThreads) {
    th.join();
}
```

4.1.2 Profiler

The decPerf sample illustrates profiling decode operations being performed during execution using the IDecodeProfiler interface.

The DecodeProfiler class implements the IDecodeProfiler interface and implements the necessary reportDecodeTime method for profiling decode operations. One instance of this class is created and registered for each instance of the decoder associated per channel.

Code 4-6: Setting Decoding Profiler

```
pDeviceWorker->setDecodeProfiler(g_vpDecProfilers[i], i);
```

For each decoding channel, the callback function reportDecodeTime() will be called once each frame is decoded. The information of frame index, video channel, device ID and the time of decoding this frame will be recorded.

4.1.3 Callback

Besides the plug-in mechanism introduced in Section 3.2, DeepStream provides a callback mechanism to get data from the decoder for a simple case where a plug-in is unnecessary. The callback function is defined by the user. The callback function should be passed to DeviceWorker by the setDecCallback function.

- Callback of Decoding

Code 4-7: Callback of Decoding

```
typedef struct {
     int frameIndex_;                    //!< Frame index
     int videoIndex_;                    //!< Video index
     int nWidthPixels_;                  //!< Frame width
     int nHeightPixels_;                 //!< Frame height
     uint8_t *dpFrame_;                  //!< Frame data (nv12
format)
     size_t frameSize_;                  //!< Frame size in bytes
     cudaStream_t stream_;               //!< CUDA stream
} DEC_OUTPUT

typedef void (*DECODER_CALLBACK)(void *pUserData, DEC_OUTPUT
*decOutput);
```

- Set Decode callback function in DeviceWorker

  Code 4-8: Setting Decode callback Function in Deviceworker

```
/** \brief Set Decode callback function
    *
    * User can define his/her own callback function to get the NV12
frame.
    * \param pUserData The data defined by user.
    * \param callback The callback function defined by user.
    * \param channel The channel index of video.
*/
virtual void setDecCallback(void *pUserData, DECODER_CALLBACK
callback, const int channel) = 0;
```

4.1.4 Running script

Build the sample by running make in the /decPerf directory.

The dependencies listed in the System Requirements need to be installed before running the build.

To run the sample, execute the **run.sh** script within the /decPerf directory. The various configuration options in the script are shown below:

```
----------------------------------------------------------------------
-devID: The device ID of GPU
-channels: The number of video channels
-fileList: The file path list, format: file1,file2,file3,…
-endlessLoop: If value equals 1, the application will reload the video at the end of
video.

../bin/sample_decPerf        -devID=${DEV_ID}       -channels=${CHANNELS}  \
                             -fileList=${FILE_LIST} -endlessLoop=1;
----------------------------------------------------------------------
```

4.1.5 Running log

Results from a sample execution is shown in the log below. Highlights in the log are annotated in red.

```
------------------------------------------------------------------------
./run.sh
[DEBUG][11:51:32] Device ID: 0
[DEBUG][11:51:32] Video channels: 2
[DEBUG][11:51:32] Endless Loop: 1
[DEBUG][11:51:32] Device name: Tesla P4
[DEBUG][11:51:32] ========== Video Parameters Begin =============
[DEBUG][11:51:32]    Video codec     : AVC/H.264
[DEBUG][11:51:32]    Frame rate      : 30/1 = 30 fps
[DEBUG][11:51:32]    Sequence format : Progressive
[DEBUG][11:51:32]    Coded frame size: [1280, 720]
[DEBUG][11:51:32]    Display area    : [0, 0, 1280, 720]
[DEBUG][11:51:32]    Chroma format   : YUV 420
[DEBUG][11:51:32] ========== Video Parameters End   =============
[DEBUG][11:51:32] ========== Video Parameters Begin =============
[DEBUG][11:51:32]    Video codec     : AVC/H.264
[DEBUG][11:51:32]    Frame rate      : 30/1 = 30 fps
[DEBUG][11:51:32]    Sequence format : Progressive
[DEBUG][11:51:32]    Coded frame size: [1280, 720]
[DEBUG][11:51:32]    Display area    : [0, 0, 1280, 720]
[DEBUG][11:51:32]    Chroma format   : YUV 420
[DEBUG][11:51:32] ========== Video Parameters End   =============
[DEBUG][11:51:33] Video [0]:  Decode Performance: 718.89 frames/second || Decoded Frames:
500
[DEBUG][11:51:33] Video [1]:  Decode Performance: 711.68 frames/second || Decoded Frames:
500 <-- decode performance for each channel
[DEBUG][11:51:33] Video [0]:  Decode Performance: 762.77 frames/second || Decoded Frames:
1000
[DEBUG][11:51:33] Video [1]:  Decode Performance: 748.20 frames/second || Decoded Frames:
1000
[DEBUG][11:51:34] Video [0]:  Decode Performance: 770.27 frames/second || Decoded Frames:
1500
[DEBUG][11:51:34] Video [1]:  Decode Performance: 738.86 frames/second || Decoded Frames:
1500
[DEBUG][11:51:35] Video [0]:  Decode Performance: 758.35 frames/second || Decoded Frames:
2000
[DEBUG][11:51:35] Video [1]:  Decode Performance: 766.09 frames/second || Decoded Frames:
2000
------------------------------------------------------------------------
```

4.2 NVDECINFER

Two versions of nvDecInfer samples are provided that illustrate use of the SDK to build decode+inference workflows. These are named nvDecInfer_classification and nvDecInfer_detection, representing classification and detection use cases. The underlying design and architecture for both of these samples is largely common. We describe the nvDecInfer_classification sample in detail, with the understanding that the detection sample implementation is largely analogous.

4.2.1 nvDecInfer_classification

The nvDecInfer_classification samples demonstrate typical usage of video decode and inference. Decoded frames are converted into BGR planar format, and use TensorRT with GoogleNet to implement inferencing. There is a user-defined plug-in to print probabilities of top-5 results into a log file.

4.2.1.1 Initializing the DeepStream library

All samples need to first initialize the DeepStream library before it can be used, by calling the **deepStreamInit()** function declared in deepStream.h.

4.2.1.2 Adding a module into the analysis pipeline

- Module: color space convertor

  The format of decoded frame is NV12 (YUV420), which is converted to RGB planar for the inference model.

  Code 4-9: Color Space Converter

```
// Add frame paser
IModule* pConvertor = pDeviceWorker-
>addColorSpaceConvertorTask(BGR_PLANAR);
```

- Module: inference

  The inference module can accept Caffe and UFF (Universal Framework Format) models, as supported in TensorRT 3.0. Note that UFF models provide a means by which to run TensorFlow-trained models through a TensorFlow-to-UFF conversion process as outlined in the TensorRT 3.0 documentation.

- Caffe Model

For executing Caffe models, the inference module needs a network description file (prototxt), a trained weight file (caffemodel), the input and output names, and a batch size as parameters. Note that this is the batch size used for inference in TensorRT. While typically this batch size would be the same as the maxBatchSize parameter specified while creating the DeviceWorker object as part of the createDeviceWorker() call, the library allows these to be different with the only requirement being that the inference batch size cannot be smaller than the number of input channels.

This part is also an example of connecting modules. The first parameter of addInferenceTask is a previous module (the color space converter module in this case) and the output index of its associated tensor to be used as input for inference.

Code 4-10: Adding Inference Task for Caffe models

```
// Add inference task
std::string inputLayerName = "data";
std::vector<std::string > outputLayerNames(1, "prob");
IModule*pInferModule=pDeviceWorker->addInferenceTask(
std::make_pair(pConvertor, 0),

     g_deployFile,

     g_modelFile,

     g_meanFile,

     inputLayerName,

     outputLayerNames,

     g_nChannels
);
```

- UFF models

For executing UFF models, the inference module needs the following parameters: UFF model description file (with .uff extension), the input and output layer names, the inference batch size, and the dimensionality of the input tensor specified based on the number of color channels in pixel format, and the height and width of the input video.

The function to be called for adding a uff-based inference layer is addInferenceTask_uff. As with the Caffe interface, the first parameter of addInferenceTask_uff is a previous module along with index of its output tensor to be used as input of inference.

Code 4-11: Adding Inference Task for UFF models

```
// Add inference task
std::string inputLayerName("input");
std::vector<std::string > outputLayerNames{"spatial_avg"};
IModule *pInferModule = pDeviceWorker->addInferenceTask_uff(
        std::make_pair(pConvertor, 0),

        g_uffFile,

        g_meanFile,

        g_nC, g_nH, g_nW, /* dimensions of input tensor */

        inputLayerName,

        outputLayerNames,

        g_nChannels);
```

- User-defined accuracy checking module

This module is an example of a user-defined module. The probability of top 5 results will be recorded into a log file. The user defined module inherits from the IModule class in the **module.h** file. The previous module should be specified when adding a module into the pipeline by DeviceWorker->addCustomerTask.

Code 4-12: Defining and Adding User-defined Module into Pipeline

```
// user-defined module inherits from IModule
class UserDefinedModule : public IModule {
…
};

// adding module into pipeline
PRE_MODULE_LIST preModules;
preModules.push_back(std::make_pair(pInferModule, 0));
UserDefinedModule *pAccurancyModule = new
UserDefinedModule(preModules, g_validationFile, g_synsetFile,
g_nChannels, logger);
assert(nullptr != pAccurancyModule);
pDeviceWorker->addCustomerTask(pAccurancyModule);
```

4.2.2 Profiler

Each module (pre-defined or user-defined) can define their profiler, and is called during DeepStream execution.

Code 4-13: Defining and Setting Module Profiler

```
// define module profiler
class ModuleProfiler : public IModuleProfiler {…}

// setup module profiler
pConvertor->setProfiler(g_pConvertorProfiler);
pInferModule->setProfiler(g_pInferProfiler);
pAccuracyModule->setProfiler(g_pAccuracyProfiler);
```

4.2.3 Callback

Each module can have their own callback function to get the result.

Code 4-14: Module Callback Function

```
typedef void (*MODULE_CALLBACK)(void *pUserData,
std::vector<IStreamTensor *>& out);
virtual void setCallback(void *pUserData, MODULE_CALLBACK callback)
= 0;
```

4.2.4 Running the sample

Build the sample by running `make` in the `/nvDecInfer_classification` directory.

Dependencies listed in the System Requirements need to be installed before running the build.

In order to execute the sample, the user needs to put together a sample video. NVIDIA provides a script (**`generate_video.sh`**) within the directory that generates a video that consists of images from the ImageNet dataset stitched together. Refer to http://image-net.org/download-faq for terms of use of the video and these images.

Run the sample as follows:

1. It not already installed, install `ffmpeg`.

```
sudo apt-get update
sudo apt-get install ffmpeg
```

2. Execute the **`generate_video.sh`** script to auto-generate a sample video (named **`sample_224x224.h264`**).
3. Execute the **`run.sh`** script.

The salient configuration options that the user can configure in the script are shown below.

```
------------------------------------------------------------------------
-channels: The number of video channels
-fileList: The file path list, format: file1,file2,file3,…
-deployFile: The path to the deploy file
-moduleFile: The path to the model file
-meanFile: The path to the mean file
-synsetFile: The synset file
-validationFile: The label file
-endlessLoop: If value equals 1, the application will reload the video at the end of
video.

../bin/sample_classification -nChannels=${CHANNELS}        \
                             -fileList=${FILE_LIST}        \
                             -deployFile=${DEPLOY}         \
                             -modelFile=${MODEL}           \
                             -meanFile=${MEAN}             \
                             -synsetFile=${SYNSET}         \
                             -validationFile=${VAL}        \
                             -endlessLoop=0
------------------------------------------------------------------------
```

4.2.5 Running log

Results from the sample execution are shown in the log below. Highlights in the log are denoted in red.

```
------------------------------------------------------------------------
./run.sh
[DEBUG][12:03:58] Video channels: 2
[DEBUG][12:03:58] Endless Loop: 0
[DEBUG][12:03:58] Device name: Tesla P4
[DEBUG][12:03:59] Use FP32 data type.
[DEBUG][12:04:01] ========== Network Parameters Begin ==========
[DEBUG][12:04:01] Network Input:
[DEBUG][12:04:01]    >Batch   :2
[DEBUG][12:04:01]    >Channel :3
[DEBUG][12:04:01]    >Height  :224
[DEBUG][12:04:01]    >Width   :224
[DEBUG][12:04:01] Network Output [0]
[DEBUG][12:04:01]    >Channel :1000
```

```
[DEBUG][12:04:01]    >Height  :1
[DEBUG][12:04:01]    >Width   :1
[DEBUG][12:04:01] Mean values = [103.907, 116.572,122.602]
[DEBUG][12:04:01] =========== Network Parameters End   ===========
[DEBUG][12:04:01] =========== Video Parameters Begin =============
[DEBUG][12:04:01]    Video codec     : AVC/H.264
[DEBUG][12:04:01]    Frame rate      : 25/1 = 25 fps
[DEBUG][12:04:01]    Sequence format : Progressive
[DEBUG][12:04:01]    Coded frame size: [224, 224]
[DEBUG][12:04:01]    Display area    : [0, 0, 224, 224]
[DEBUG][12:04:01]    Chroma format   : YUV 420
[DEBUG][12:04:01] =========== Video Parameters End   =============
[DEBUG][12:04:01] =========== Video Parameters Begin =============
[DEBUG][12:04:01]    Video codec     : AVC/H.264
[DEBUG][12:04:01]    Frame rate      : 25/1 = 25 fps
[DEBUG][12:04:01]    Sequence format : Progressive
[DEBUG][12:04:01]    Coded frame size: [224, 224]
[DEBUG][12:04:01]    Display area    : [0, 0, 224, 224]
[DEBUG][12:04:01]    Chroma format   : YUV 420
[DEBUG][12:04:01] =========== Video Parameters End   =============
[DEBUG][12:04:05] Video[1] Decoding Performance: 31.03 frames/second || Total Frames: 100
<- Decode performance for each channel
[DEBUG][12:04:05] Video[0] Decoding Performance: 31.04 frames/second || Total Frames: 100
[DEBUG][12:04:05] Analysis Pipeline Performance: 62.02 frames/second || Total Frames: 200
<- Combined end to end decode+inference performance across all channels
[DEBUG][12:04:08] Video[0] Decoding Performance: 30.69 frames/second || Total Frames: 200
[DEBUG][12:04:08] Video[1] Decoding Performance: 30.69 frames/second || Total Frames: 200
[DEBUG][12:04:08] Analysis Pipeline Performance: 61.39 frames/second || Total Frames: 400
[DEBUG][12:04:11] Video[1] Decoding Performance: 30.13 frames/second || Total Frames: 300
[DEBUG][12:04:11] Video[0] Decoding Performance: 30.08 frames/second || Total Frames: 300
[DEBUG][12:04:11] Analysis Pipeline Performance: 60.15 frames/second || Total Frames: 600
-----------------------------------------------------------------------------
```

4.3 NVDECINFER_DETECTION

The nvDecInfer_detection sample demonstrates use of a ResNet-10 network to implement detection use case using DeepStream. The network supports detection of four classes of objects: cars, people, road signs, and two-wheelers. It leverages support within TensorRT for optimizing trained networks to reduced INT8 precision that can then be deployed on NVIDIA Tesla® P4 GPUs, leading to efficiency gains. Note that the network is unpruned and is provided for illustrative purposes only while offering no guarantees for accuracy or performance. The sample can be executed by running the **run.sh** script, which uses the **sample_720p.h264** video in the samples/data/video directory for input. The sample video as well as other input parameters can be configured as required in the **run.sh** script.

4.3.1 Building the sample

Build the sample by running make in the /nvDecInfer_detection directory.

Besides the dependencies listed in the System Requirements, the detection samples require other software to be installed as explained below.

The detection sample can potentially render the bounding boxes of objects being detected as part of the GUI. To support this, the user is required to install a few dependent software packages that are necessary, as listed below:

- Mesa-dev packages

```
sudo apt-get install build-essential
sudo apt-get install libgl1-mesa-dev
```

- libglu

```
sudo apt-get install libglu1-mesa-dev
```

- freeglut

```
sudo apt-get install freeglut3-dev
```

- openCV

```
sudo apt-get install libopencv-dev python-opencv
```

- glew

  Install from project webpage: http://glew.sourceforge.net/index.html

4.3.2 Running the sample

By default, information about detected objects is sent to per-channel log files in KITTI format under the /logs directory. Only the type of object and bounding box coordinate fields are populated in the log.

GUI visualization of the results is disabled by default. It can be enabled using the "-gui 1" option. Note that a window manager needs to be running to support this use case.

The salient options in the script that the user can configure are shown below.

```
----------------------------------------------------------------------------
-channels: The number of video channels
-fileList: The file path list, format: file1,file2,file3,…
-deployFile: The path to the deploy file
-moduleFile: The path to the model file
-meanFile: The path to the mean file
-synsetFile: The synset file
-validationFile: The label file
-gui: enable gui (outputs kitti logs by default)
-endlessLoop: If value equals 1, the application will reload the video at the end of
video.

../bin/sample_detection      -nChannels=${CHANNELS}        \
                             -fileList=${FILE_LIST}        \
                             -deployFile=${DEPLOY}         \
                             -modelFile=${MODEL}           \
                             -meanFile=${MEAN}             \
                             -synsetFile=${SYNSET}         \
                             -validationFile=${VAL}        \
                             -endlessLoop=0
----------------------------------------------------------------------------
```

4.4 NVSMARTDECINFER_DETECTION

The detection and classification samples described previously decode and infer on every frame, which may not be necessary for certain inference workloads. The nvSmartDecInfer_detection sample implements an optimization of this model whereby only I-frames in the input video are injected into the DeepStream pipeline and subsequently decoded and inferred. This reduces the per stream workload on the GPU, which may allow more streams to be processed as part of the pipeline.

The nvSmartDecInfer_detection sample illustrates application of the smart decode capability to the nvDecInfer_detection sample described previously. The crux of the functionality is in the FileDataProvider::getData() function, which now parses the input video for the next I-frame rather than retrieving every frame. The functionality for parsing of the video file to detect I-frames is performed by the findIFrame() function that maintains a state machine as it progressively parses bytes from the file to identify the next I-frame.

4.4.1 Running the sample

Build the sample using the same procedure as outline for the detection sample as outlined in section 4.3.1.

As with the other samples, the smart decode sample can be executed by running the **run.sh** script. Its parameters are identical to the nvDecInfer_detection sample, except that on-screen display (enabled using the -gui option) is not supported since it is only the I-frames that flow through the pipeline making continuous rendering of video not possible. The logs output by the sample show the number of I-frames that are analyzed by the DeepStream pipeline, indicating a frequency of one I-frame every 250 frames for the provided video.

```
$ ./run.sh
 [DEBUG][11:16:23] Video channels: 1
[ERROR][11:16:23] Warning: No mean files.
[DEBUG][11:16:23] GUI disabled. KITTI log files will be generated.
[DEBUG][11:16:23] Endless Loop: 0
[DEBUG][11:16:23] Device name: Tesla P4
[DEBUG][11:16:23] Use INT8 data type.
[DEBUG][11:16:25] ========== Network Parameters Begin ==========
[DEBUG][11:16:25] Network Input:
[DEBUG][11:16:25]    >Batch   :1
[DEBUG][11:16:25]    >Channel :3
[DEBUG][11:16:25]    >Height  :368
[DEBUG][11:16:25]    >Width   :640
[DEBUG][11:16:25] Network Output [0]
[DEBUG][11:16:25]    >Channel :4
[DEBUG][11:16:25]    >Height  :23
```

```
[DEBUG][11:16:25]    >Width   :40
[DEBUG][11:16:25] Network Output [1]
[DEBUG][11:16:25]    >Channel :16
[DEBUG][11:16:25]    >Height  :23
[DEBUG][11:16:25]    >Width   :40
[DEBUG][11:16:25] ========== Network Parameters End   ==========
[DEBUG][11:16:25] Set SPS for smart decoding
[DEBUG][11:16:25] Set PPS for smart decoding
[DEBUG][11:16:25] the index of 1 I Frame in the video is 0
[DEBUG][11:16:25] ========== Video Parameters Begin =============
[DEBUG][11:16:25]    Video codec     : AVC/H.264
[DEBUG][11:16:25]    Frame rate      : 30/1 = 30 fps
[DEBUG][11:16:25]    Sequence format : Progressive
[DEBUG][11:16:25]    Coded frame size: [1280, 720]
[DEBUG][11:16:25]    Display area    : [0, 0, 1280, 720]
[DEBUG][11:16:25]    Chroma format   : YUV 420
[DEBUG][11:16:25] ========== Video Parameters End   =============
[DEBUG][11:16:25] the index of 2 I Frame in the video is 250
[DEBUG][11:16:25] the index of 3 I Frame in the video is 500
[DEBUG][11:16:25] the index of 4 I Frame in the video is 750
[DEBUG][11:16:25] the index of 5 I Frame in the video is 1000
[DEBUG][11:16:25] the index of 6 I Frame in the video is 1250
[DEBUG][11:16:25] User: Ending...
```

Notice

Trademarks

Copyright