DeepStream SDK: Towards Large Scale Deployment of Intelligent Video Analytics Systems
November 1, 2017
Jeremy Furtek
MOTIVATION
Proliferation of Video Data

Content Filtering
Ad Injection
Parking Management
Traffic Engineering
Retail Analytics
Securing Critical Infrastructure
In-Vehicle Analytics
Law Enforcement
Advances in Deep Learning: Classification and Beyond

- **Image Classification** (AlexNet, 2012)
  - What the computer sees:
    - 82% cat
    - 10% dog
    - 2% not
    - 1% mug

- **Object Detection** (R-FCN, 2016)
  - Input image
  - Ground truth
  - ENet output

- **Semantic Image Segmentation** (ENet, 2016)

- **Image Compression** (WaveOne, 2017)

- **Structure From Motion** (SfM-Net, 2017)

- **Pose-Aware Face Recognition** (USC, 2017)
WHAT IS DEEPSTREAM?
DEEP LEARNING

TRAINING
Learning a new capability from existing data

INFEERENCE
Applying this capability to new data

Untrained Neural Network Model

Deep Learning Framework

TRAINING DATASET

Trained Model
New Capability

NEW DATA
App or Service Featuring Capability

Trained Model
Optimized for Performance
DEEP LEARNING WITH DEEPSTREAM

**TRAINING**
Learning a new capability from existing data

- Untrained Neural Network Model
- Deep Learning Framework
- Training Dataset
- Trained Model
- New Capability

**INFEERENCE**
Applying this capability to new data

- Video Data
- DeepStream Enabled App or Service
- Trained Model
  - Optimized for Performance
VIDEO ANALYTICS WITH DEEP LEARNING ON NVIDIA GPUS: AN OVERVIEW
From Video to Analytics With Deep Learning...
From Video to Analytics With Deep Learning...

**Video Decode**

Fully-accelerated hardware decoding on NVIDIA GPUs (NVDEC) is accessible via the Video Decode API.

- **Decode HW**
  - Formats:
    - MPEG-2
    - VC1
    - VP8
    - VP9
    - H.264
    - H.265
    - Lossless
  - Bit depth:
    - 8 bit
    - 10 bit
  - Color:
    - YUV 4:2:0
  - Resolution:
    - Up to 8K

- **Encode HW**
  - Formats:
    - H.264
    - H.265
    - Lossless
  - Bit depth:
    - 8 bit
    - 10 bit
  - Color:
    - YUV 4:4:4
    - YUV 4:2:0
  - Resolution:
    - Up to 8K

---

* See support diagram for previous NVIDIA HW generations
** 4:2:2 is not natively supported as HW
*** Support is codec dependent
From Video to Analytics With Deep Learning...
Custom CUDA kernels and/or optimized libraries (e.g. NVIDIA Performance Primitives, or NPP) enable high performance image conversions.

```c
__global__ void resize(uint8_t* dst, const uint8_t* src, int dst_w, int dst_h, int src_w, int src_h)
{
    ...
}
```
From Video to Analytics With Deep Learning...
Inference

TensorRT* provides a neural network optimizer and a high performance runtime library for inference deployment on NVIDIA GPUs.

Step 1: Optimize a trained neural network

Step 2: Perform real-time inference with the TensorRT runtime

*The current DeepStream release supports TensorRT 2.1.

<table>
<thead>
<tr>
<th></th>
<th>inference results</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>0.9</td>
</tr>
<tr>
<td>dog</td>
<td>0.05</td>
</tr>
<tr>
<td>tree</td>
<td>0.05</td>
</tr>
</tbody>
</table>
From Video to Analytics With Deep Learning...

- **Video Decode**: Compressed video bitstream is decoded to a video frame.
- **Image Conversion**: The video frame is converted to a YCbCr image.
- **Inference**: The YCbCr image is converted to a BGR image and fed into the inference process, which outputs confidence scores for categories such as cat, dog, and tree.
From Video to Analytics With Deep Learning...

Display/Storage

Processing of inference output is application dependent.

Some common options are:

- Real-time display with overlay
- Alert generation
- Database insertion
- Augmented video encoding
- “Slew-to-Cue” camera control
- Tracker update
- Custom post-processing algorithms (CUDA, OpenCV, ...)

Image Conversion

Inference

Display / Storage

1

0

(encoded video)

0

1

(decoded video)

frame

BGR

(image)

(YCbCr image)

cat

dog

tree

0.9

0.05

0.05

[Image of a street scene with objects detected and labeled]
Video to Analytics With the DeepStream SDK

DeepStream runtime library (libdeepstream)

1. Video Decode
2. Image Conversion
3. Inference
4. Display / Storage

- Compressed video bitstream
- Decoded video frame
- BGR image
- Inference results

(YCbCr image)

<table>
<thead>
<tr>
<th></th>
<th>cat</th>
<th>dog</th>
<th>tree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.9</td>
<td>0.05</td>
<td>0.05</td>
</tr>
</tbody>
</table>
USING THE DEEPSTREAM LIBRARY
DeepStream from the Ground Up

Tensors

DeepStream tensors are objects that implement the IStreamTensor interface and represent a 4-dimensional array of values stored in NCHW order.

typedef enum {
    FLOAT_TENSOR,       // float32
    NV12_FRAME,        // decoded YCbCr
    OBJ_COORD,         // detection bounding box
    CUSTOMER_TYPE     // user defined
} TENSOR_TYPE;

typedef enum {
    GPU_DATA,          // device memory
    CPU_DATA,          // host memory
} MEMORY_TYPE;

IStreamTensor* createStreamTensor(int maxlen, size_t elemsize, TENSOR_TYPE Ttype, MEMORY_TYPE Mtype, int deviceID);
DeepStream from the Ground Up

**Modules**

*DeepStream modules* are objects that implement the `IModule` interface and operate on one or more input *tensors* to produce one or more output *tensors*.

```
void IModule::execute(const ModuleContext& ctx, const std::vector<IStreamTensor*>& in, const std::vector<IStreamTensor*>& out) = 0;
```
DeepStream from the Ground Up

Building the DeepStream dataflow graph

DeepStream modules maintain connectivity information using a `std::vector` of `<IModule*, int>` pairs that represent the inputs to the module.

![Diagram showing the dataflow graph with modules A, B, and C, and their connections through output[A], output[B], and output[C]. The table for module C shows inputs `&A` with index 1 and `&B` with index 0.]
Using the *DeepStream* Library

Step 1: Create a device worker object

```c
// Create a device worker
IDeviceWorker* pDW = createDeviceWorker(1, // num channels
                                         0); // GPU device ID
```
Using the *DeepStream* Library

**Step 2: Add a decode task**

```cpp
// Create a device worker
IDeviceWorker* pDW = createDeviceWorker(1, 0);

// Add a decode task, specifying the CODEC type
pDW->addDecodeTask(cudaVideoCodec_H264);
```
Using the *DeepStream* Library

**Step 3: Add an image conversion module**

```c
// Create a device worker
IDeviceWorker* pDW = createDeviceWorker(1, 0);

// Add a decode task, specifying the CODEC type
pDW->addDecodeTask(cudaVideoCodec_H264);

// Add a module for color conversion.
// Color conversion module outputs:
// 0: BGR_PLANAR
// 1: NV12 (YCbCr)
IModule* pCC = pDW->addColorSpaceConvertorTask(BGR_PLANAR);
```

Caffe uses BGR ordering (not RGB)
Inference in \textit{DeepStream}: CAFFE and TensorRT

\begin{itemize}
\item \texttt{caffe train} -solver solver.prototxt
\end{itemize}
Using the DeepStream Library

Step 4: Add an inference module

```cpp
// Add a module for color conversion
// Color conversion module outputs:
// 0: BGR_PLANAR
// 1: NV12 (YCbCr)
IModule* pCC = pDW->addColorSpaceConvertorTask(BGR_PLANAR);

// Add an inference module using output 0 from color conversion
std::vector<std::pair<IModule*, int>> inf_inputs{std::make_pair(pCC, 0)};
IModule* pInf = pDW->addInferenceTask(inf_inputs,
  prototxt_file.c_str(),
  model_file.c_str(),
  input_layer_name,
  output_layer_names_vector,
  num_channels,
  &inference_params);
```
Using the DeepStream Library

Step 5: Add custom output processor(s)

```cpp
// Add a parser module using output 0 from inference
std::vector<std::pair<IModule*, int>> parser_inputs;
parser_inputs.push_back(std::make_pair(pInf, 0));
IModule* pParse = new UserInferenceParser(parser_inputs);
pDW->addCustomerTask(pParse);

// Add a display module using image and parser outputs
std::vector<std::pair<IModule*, int>> display_inputs;
display_inputs.push_back(std::make_pair(pCC, 0));
display_inputs.push_back(std::make_pair(pParse, 0));
IModule* pDisp = new UserDisplayModule(display_inputs);
pDW->addCustomerTask(pDisp);
```
Using the DeepStream Library

Step 6: Start the device worker

```cpp
// Add a display module using image conversion and parser outputs
std::vector<std::pair<IModule*, int>> display_inputs;
display_inputs.push_back(std::make_pair(pCC, 0)); // BGR output
display_inputs.push_back(std::make_pair(pParse, 0));
IModule* pDisp = new UserDisplayModule(display_inputs);
pDW->addCustomerTask(pDisp);

// Start the device worker...
pDW->start();
```
Using the *DeepStream* Library

**Step 7: Initiate data input**

```
// Start the device worker...
pDW->start();

// Initiate data input by pushing data to the device worker
while(1)
{
    // Read data from file/socket/database...
pDW->pushPacket(data, num_bytes, channel_num);
}
```
SCALING WITH DEEPSTREAM
DECODE PERFORMANCE

NVDEC H.264 YUV 4:2:0

Number of 30fps Streams / NVDEC

<table>
<thead>
<tr>
<th></th>
<th>4096 x 4096</th>
<th>3840 x 2160</th>
<th>2560 x 1440</th>
</tr>
</thead>
<tbody>
<tr>
<td>TESLA M60</td>
<td>4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>TESLA P40</td>
<td>5</td>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>
Scaling Video Analytics With DeepStream: Multiple Channels

**DeepStream** runtime library (libdeepstream)

- **Video Decode** → **Image Conversion** → **Inference**
- **Display / Storage**

---

- **GPU 0**
  - **channel 0**
  - **channel 1**
  - **channel 2**
  - **channel 3**
Faster Inference with INT8 Integers

GPUs with Compute Capability 6.1 and higher support dot product instructions with 8-bit operands. (Examples of GPUs with this capability are Tesla P4 and P40.)
## Inference with INT8: Accuracy

### Results - Accuracy

<table>
<thead>
<tr>
<th>NETWORK</th>
<th>Top1</th>
<th>Top5</th>
<th>Calibration using 5 batches</th>
<th>Calibration using 10 batches</th>
<th>Calibration using 50 batches</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Top1</td>
<td>Top5</td>
<td>Top1</td>
</tr>
<tr>
<td>Resnet-50</td>
<td>73.23%</td>
<td>91.18%</td>
<td>73.03%</td>
<td>91.15%</td>
<td>73.02%</td>
</tr>
<tr>
<td>Resnet-101</td>
<td>74.39%</td>
<td>91.78%</td>
<td>74.52%</td>
<td>91.64%</td>
<td>74.38%</td>
</tr>
<tr>
<td>Resnet-152</td>
<td>74.78%</td>
<td>91.82%</td>
<td>74.62%</td>
<td>91.82%</td>
<td>74.66%</td>
</tr>
<tr>
<td>VGG-19</td>
<td>68.41%</td>
<td>88.78%</td>
<td>68.42%</td>
<td>88.69%</td>
<td>68.42%</td>
</tr>
<tr>
<td>Googlenet</td>
<td>68.57%</td>
<td>88.83%</td>
<td>68.21%</td>
<td>88.67%</td>
<td>68.10%</td>
</tr>
<tr>
<td>Alexnet</td>
<td>57.08%</td>
<td>80.08%</td>
<td>57.00%</td>
<td>79.98%</td>
<td>57.00%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NETWORK</th>
<th>Top1</th>
<th>Top5</th>
<th>Diff Top1</th>
<th>Diff Top5</th>
<th>Diff Top1</th>
<th>Diff Top5</th>
<th>Diff Top1</th>
<th>Diff Top5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resnet-50</td>
<td>73.23%</td>
<td>91.18%</td>
<td>0.20%</td>
<td>0.03%</td>
<td>0.22%</td>
<td>0.13%</td>
<td>0.13%</td>
<td>0.12%</td>
</tr>
<tr>
<td>Resnet-101</td>
<td>74.39%</td>
<td>91.78%</td>
<td>-0.13%</td>
<td>0.14%</td>
<td>0.01%</td>
<td>0.09%</td>
<td>-0.01%</td>
<td>0.06%</td>
</tr>
<tr>
<td>Resnet-152</td>
<td>74.78%</td>
<td>91.82%</td>
<td>0.15%</td>
<td>0.01%</td>
<td>0.11%</td>
<td>0.01%</td>
<td>0.08%</td>
<td>0.05%</td>
</tr>
<tr>
<td>VGG-19</td>
<td>68.41%</td>
<td>88.78%</td>
<td>-0.02%</td>
<td>0.09%</td>
<td>-0.01%</td>
<td>0.10%</td>
<td>0.03%</td>
<td>0.07%</td>
</tr>
<tr>
<td>Googlenet</td>
<td>68.57%</td>
<td>88.83%</td>
<td>0.36%</td>
<td>0.16%</td>
<td>0.46%</td>
<td>0.25%</td>
<td>0.45%</td>
<td>0.19%</td>
</tr>
<tr>
<td>Alexnet</td>
<td>57.08%</td>
<td>80.06%</td>
<td>0.08%</td>
<td>0.08%</td>
<td>0.08%</td>
<td>0.07%</td>
<td>0.03%</td>
<td>-0.01%</td>
</tr>
</tbody>
</table>

TensorRT 2.1, all optimizations enabled. ILSVRC2012 validation dataset, batch = 25 images. Accuracy was measured on 500 batches which were not used for the calibration.

Inference with INT8: Performance

Results - Performance

Using INT8 Inference in DeepStream
Step 1: Run TensorRT offline to generate a calibration table file

- trained.caffemodel (BINARYPROTO)
- trained weights
- network description (deploy)
- deploy.protoxt
- TensorRT
- CUDA Engine
- calibration table
Using int8 Inference in DeepStream

Step 2: Provide the path to the calibration file when creating the inference task

```c++
// Add an inference module using output 0 from color conversion
std::vector<std::pair<IModule*, int>> inf_inputs{std::make_pair(pCC, 0)};

inference_params.calibrationTableFile = calibration_file.c_str();

IModule* pInf = pDW->addInferenceTask(inf_inputs,
  prototxt_file.c_str(),
  model_file.c_str(),
  input_layer_name,
  output_layer_names_vector,
  num_channels,
  &inference_params);
```
Scaling Video Analytics With DeepStream: Multiple GPUs

**GPU 0**

- **DeepStream device worker (device ID = 0)**
  - Channel 0: Video Decode → Image Conversion → Inference
  - Channel 1: Video Decode → Image Conversion → Inference

**GPU 1**

- **DeepStream device worker (device ID = 1)**
  - Channel 0: Video Decode → Image Conversion → Inference
  - Channel 1: Video Decode → Image Conversion → Inference

Display / Storage
Examples (Videos)
DeepStream Resources

- **DeepStream SDK Download**
  
  https://developer.nvidia.com/deepstream-sdk

- **NVIDIA Parallel For All Blog:**
  
  *DeepStream*: Next Generation Video Analytics for Smart Cities
  

- **NVIDIA Developer Forums**
  
  https://devtalk.nvidia.com/

- **Caffe Model Zoo**
  
  https://github.com/BVLC/caffe/wiki/Model-Zoo

- **NVIDIA Video Encode and Decode GPU Support Matrix**
  
NVIDIA DEEP LEARNING INSTITUTE

Training available as online self-paced labs and instructor-led workshops


Find or request an instructor-led workshop at [www.nvidia.com/dli](http://www.nvidia.com/dli)

Educators can download the Teaching Kit at [developer.nvidia.com/teaching-kit](http://developer.nvidia.com/teaching-kit) and contact nvdli@nvidia.com for info on the University Ambassador Program.

Caffe2  |  Microsoft Cognitive Toolkit  |  mxnet
TensorFlow  |  PyTorch

- Fundamentals
- Autonomous Vehicles
- Healthcare
- Intelligent Video Analytics
- Robotics
- Game Development & Digital Content
- Finance
- Parallel Computing
- Virtual Reality