S9391
GstCUDA: Easy GStreamer and CUDA Integration

Eng. Daniel Garbanzo
MSc. Michael Grüner
GTC March 2019
Agenda

About RidgeRun
GStreamer Overview
CUDA Overview
GstCUDA Introduction
Application Examples
Performance Statistics
GstCUDA Demo on TX2
Q&A
About Us

- US Company - R&D Lab in Costa Rica
- 15 years of experience
- Embedded Linux and GStreamer experts
- Custom multimedia solutions
- Digital signal/image processing
- AI and Machine Learning solutions
- System optimization: CUDA, GStreamer, OpenCL, OpenGL, OpenVX, Vulkan
- Support for embedded and resource constrained systems
- Professional services, dedicated teams and specialized tools
• Complex multimedia applications require a lot of processing resources
• GStreamer offers a flexible way for creating multimedia applications
• CUDA offers high performance accelerated processing capabilities
- Open source framework for audio and video applications
- Based on a pipeline architecture
- Extensible design based on plugins (more than 1000 freely available)
- Automatic format and synchronization handling
- Tools for easy prototyping
● Each plugin represents a different processing module
● The plugins are linked and arranged in a pipeline
● Freedom to build arbitrary pipelines for different applications
Modular design lets you change your application easily!

Easily change from SW to HW accelerated processing
Easily change your application end use
Modular design lets you change your application easily!

Code equivalent:

gst-launch v4l2src ! videoconvereter ! x265enc ! mpegtsmux ! filesink

gst-launch v4l2src ! videoconvereter ! omxh265enc ! mpegtsmux ! udpsink
Development environment for high performance GPU-accelerated applications

General purpose data processing via parallel algorithm execution on GPU

Extensive development, debugging and profiling set of tools
GstCUDA Integrates the Best of Both Worlds
GstCUDA

Framework enabling easy integration of CUDA algorithms into GStreamer pipelines

Eliminates the need to learn GStreamer internals

Focus on your CUDA algorithm to reduce time to market!
What Does GstCUDA Solve?
Integration Complexities

- A lot of roadblocks between CUDA and GStreamer
- These are complex and time consuming
- Time is money!
Development Time

Without GstCUDA:
- 3 Months
- 10 days
- 5 days
Total = 3.5 months

With GstCUDA:
- 10 days
- 0.1 day
Total = 10.1 days

- Reduce development time
- Focus on the CUDA logic
- Minimize time to market
Performance Bottleneck

- Data transfers can be a bottleneck
- Memory copies can degrade performance
- Incompatibility between different memory types
Performance Bottleneck

Without GstCUDA

- Data transfers bottleneck cause poor performance
- Limited framerate at high resolutions

With GstCUDA

- Efficient memory handling improves performance
- Up to 2x 4K@60fps
Supported Platforms

- Focused for NVIDIA Embedded Platforms

Jetson TX1, TX2, TX2i and Nano

Jetson AGX Xavier
GstCUDA Key Features

- Allows CUDA algorithm easy integration into GStreamer pipelines
- Out of the box quick prototyping tools
- Fine-grained control of image memory layout (planes, strides, etc...)
- Automatic efficient memory handling
GstCUDA Key Features

- High performance for GStreamer/CUDA applications
- Zero memory copy interface between CUDA and GStreamer
- Direct handling of HW (NVMM) buffers
- Unified Memory allocation mechanism
Framework Overview
Quick Prototyping Elements

GStreamer elements for CUDA quick prototyping

Algorithms are loaded at runtime as plug-ins

Change the algorithm on the fly
Cudafilter Element

Video source → cudafilter → Video sink

location = median_filter.so

Median Filter CUDA algorithm

Single input / Single output configuration
Cudamux Element

Video source

IR

location = thermal_overlay.so

Video sink

Thermal Image Overlay
CUDA algorithm

Multiple input / Single output configuration
## CUDA Algorithm Interface

- Make your CUDA algorithm compatible by implementing these interfaces

### Cudafilter Interface

- `bool open();`
- `bool close();`
- `bool process (const GstCudaData &inbuf, GstCudaData &outbuf);`
- `bool process_ip (const GstCudaData &inbuf, GstCudaData &outbuf);`

### Cudamux Interface

- `bool open();`
- `bool close();`
- `bool process (vector<GstCudaData> &inbufs, GstCudaData &outbuf);`
- `bool process_ip (vector<GstCudaData> &inbufs, GstCudaData &outbuf);`
Buffer Processing Methods

process_ip
(In place)
Algorithm outputs are written to the input buffer

process
(Not in place)
Algorithm inputs and outputs are different buffers
Create Your Custom Element

- Some applications may require specialized elements
- GstCUDA provides bases classes to simplify development

**GstCUDABaseFilter:**
- Single-input / Single-output topology

**GstCUDABaseMISO:**
- Multiple-input / Single-output topology
GstCUDA Framework Usage Example

- Inherit parent classes and focus on the algorithm!

Diagram:
- Video source
- GstCUDA custom element
- Image Stitching
- Based on Multiple Input / Single Output Topology
GstCUDA Framework Summary

- The framework includes:

GstCUDA API
- Utils to handle memory interfaces
- GStreamer Unified Memory allocators
- Parent classes for different topologies

Quick prototyping elements
- Generic elements to evaluate custom algorithms
- Runtime loading of CUDA algorithms

Set of examples
- Complete GstCUDA element boilerplate
- CUDA algorithms for the prototyping elements
GstCUDA Application Areas Examples Video
Industrial Applications: Border Enhancement
Automation Applications: Hough Transform
Security Applications: Motion Detection/Estimation
Performance Statistics
Test Conditions

- Image convolution algorithm
  \[(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n - m]\]
- Stressing compute capabilities
- Variable convolution kernel size
- 1080p@240fps / 1080p@60fps stream input
- Cudafilter element
- Unified Memory allocator
- Jetson TX2 platform
- Not In-place

```
location = convolution.so
```
Varying Algorithm / Fixed Image Size

Framerate Stats

Average maximum framerate at different convolution kernel sizes

1080p@240ps input

- 3x3: 197 fps
- 5x5: 154 fps
- 7x7: 131 fps
- 9x9: 110 fps
- 11x11: 89 fps
- 13x13: 73 fps

Average Maximum Framerate [fps]
Varying Algorithm / Fixed Image Size

Processing Time Stats

Average processing time at different convolution kernel sizes

1080p@60fps input

<table>
<thead>
<tr>
<th>Kernel Size</th>
<th>Average Processing Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>3x3</td>
<td>3.13</td>
</tr>
<tr>
<td>5x5</td>
<td>4.21</td>
</tr>
<tr>
<td>7x7</td>
<td>5.55</td>
</tr>
<tr>
<td>9x9</td>
<td>7.14</td>
</tr>
<tr>
<td>11x11</td>
<td>9.33</td>
</tr>
<tr>
<td>13x13</td>
<td>11.94</td>
</tr>
</tbody>
</table>
Varying Algorithm / Fixed Image Size

**CPU Load Stats**

Average CPU load at different convolution kernel sizes
1080p@60fps input

**GPU Load Stats**

Average GPU load at different convolution kernel sizes
1080p@60fps input

*baseline = simple capture pipeline (without GstCUDA)
Fixed Algorithm / Varying Image Size

Test Conditions

- Memory copy algorithm
  \[ y(n,m) = x(n,m) \]
- Stressing data transfer
- Variable input resolution
- Cudafilter element
- Unified Memory allocator
- Jetson TX2 platform
- In-place vrs not In-place

\[ \text{location} = \text{memcpy.so} \]
Fixed Algorithm / Varying Image Size

Framerate Stats

Average maximum framerate at different resolutions

Note: Maximum Framerate limited to 245 fps by the video source
Fixed Algorithm / Varying Image Size

Processing Time Stats

Average processing time at different resolutions

- 4K@60fps Not-IP: 0.856 ms
- 4K@60fps IP: 0.723 ms
- 1080p@60fps Not-IP: 1.599 ms
- 1080p@60fps IP: 0.727 ms
- 720p@60fps Not-IP: 2.595 ms
- 720p@60fps IP: 4.195 ms

Average Processing Time [ms]
**Fixed Algorithm / Varying Image Size**

**CPU Load Stats**

- **Average CPU load at different resolutions**
  - 4K@60fps Not-IP
  - 4K@60fps IP
  - 1080p@60fps Not-IP
  - 1080p@60fps IP
  - 720p@60fps Not-IP
  - 720p@60fps IP
  - 4K@60fps baseline*
  - 1080p@60fps baseline*
  - 720p@60fps baseline*

**GPU Load Stats**

- **Average GPU load at different resolutions**
  - 4K@60fps Not-IP
  - 4K@60fps IP
  - 1080p@60fps Not-IP
  - 1080p@60fps IP
  - 720p@60fps Not-IP
  - 720p@60fps IP

*baseline = simple capture pipeline (without GstCUDA)*
Test Conditions

- Simple image mixing algorithm
  \[ y_{(n,m)} = 0.5(x_1(n,m) + x_2(n,m)) \]
- Stressing data transfer
- Variable input resolution
- Cudamux element
- Unified Memory allocator
- In-place=True
- Jetson TX2 platform

\texttt{location = mixer.so}
Fixed Algorithm / Varying Image Size

Framerate Stats

Average maximum framerate at different resolutions

- 4K: 139 fps
- 1080p: 224 fps
- 720p: 235 fps
- 1280x540: 239 fps

Note: Maximum framerate limited to 240fps by the video source.
Fixed Algorithm / Varying Image Size

**CPU Load Stats**

Average CPU load at different resolutions

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Average CPU load [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>4K@60fps</td>
<td>26.3</td>
</tr>
<tr>
<td>2x4K@60fps</td>
<td>16.5</td>
</tr>
<tr>
<td>2x720p@60fps</td>
<td>12.9</td>
</tr>
</tbody>
</table>

*baseline = simple capture pipeline (without GstCUDA)*

**GPU Load Stats**

Average GPU load at different resolutions

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Average GPU load [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>4K@60fps</td>
<td>16.5</td>
</tr>
<tr>
<td>1080p@60fps</td>
<td>3.1</td>
</tr>
<tr>
<td>720p@60fps</td>
<td>1.1</td>
</tr>
</tbody>
</table>
GstCUDA Live Demo on Jetson TX2
Sobel Filter 1080p60fps

Code equivalent:

gst-launch-1.0 nvcamerasrc sensor-id=2 fpsRange=60,60 ! "video/x-raw(memory:NVMM),width=1920,height=1080,framerate=60/1,format=I420" ! nvvidconv ! "video/x-raw" ! queue ! cudafilter in-place=false location=/borders.so ! queue ! nvoverlaysink
Resources

- GstCUDA wiki page:
  - gstcuda.ridgerun.com

- RidgeRun Website:
  - ridgerun.com

- RidgeRun Contact:
  - ridgerun.com/contact