GStreamer on Texas Instruments OMAP35x Processors

Don Darling
Texas Instruments, Inc.
ddarling@ti.com

Chase Maupin
Texas Instruments, Inc.
chase.maupin@ti.com

Brijesh Singh
Texas Instruments, Inc.
bksingh@ti.com

Abstract

The Texas Instruments (TI) OMAP35x applications processors are targeted for embedded applications that need laptop-like performance with low power requirements. Combined with hardware accelerators for multimedia encoding and decoding, the OMAP35x is ideal for handheld multimedia devices. For OMAP35x processors that have both an ARM® and a digital signal processor (DSP), TI has created a GStreamer plugin that enables the use of the DSP and hardware accelerators for encode and decode operations while leveraging open source elements to provide common functionality such as AVI stream demuxing.

Often in the embedded applications space there are fewer computation and memory resources available than in a typical desktop system. On ARM+DSP systems, the DSP can be used for CPU-intensive tasks such as audio and video decoding to reduce the number of cycles consumed on the ARM processor. Likewise, additional hardware accelerators such as DMA engines can be used to move data without consuming ARM cycles. This leaves the ARM available to handle other operations such as running a web browser or media player, and thus provides a more feature-rich system. This paper covers the design of the TI GStreamer plugin, considerations for using GStreamer in an embedded environment, and the community project model used in ongoing development.

1 GStreamer Overview

GStreamer is an open source framework that simplifies the development of multimedia applications, such as media players and capture encoders. It encapsulates existing multimedia software components, such as codecs, filters, and platform-specific I/O operations, by using a standard interface and providing a uniform framework across applications.

The modular nature of GStreamer facilitates the addition of new functionality, transparent inclusion of component advancements and allows for flexibility in application development and testing. Developers can join modular elements together in a pipeline to easily create custom workflows.

GStreamer brings a lot of value-added features to OMAP35x, including audio and video synchronization, interaction with a wide variety of open source plugins (muxers, demuxers, codecs, and filters), and the ability to play multimedia clips such as those available from YouTube. Collaboration with the GStreamer community exposes many opportunities for code reuse, which aids in the stabilization and enrichment of existing components rather than replicating existing functionality. New GStreamer features are continuously being added, and the core libraries are actively supported by participants in the GStreamer community. Additional information about the GStreamer framework is available on the GStreamer project site [3].

2 The TI GStreamer Plugin

One benefit of using GStreamer as a multimedia framework is that the core libraries already build and run on ARM Linux. Only a GStreamer plugin is required to enable additional OMAP35x hardware features. The TI GStreamer plugin provides elements for GStreamer pipelines that enable the use of plug-and-play DSP codecs and certain hardware-accelerated operations, such as video frame resizing and accelerated memory copy operations.

In addition to enabling OMAP35x hardware features, the following additional goals needed to be addressed when writing the TI GStreamer plugin:

- The plugin should provide a robust, portable baseline implementation that serves as a stable starting point for customer application development.

- The plugin should be easy to build and install.
- Certain performance requirements need to be met beyond the basic utilization of the DSP and hardware accelerators. More detail on performance considerations will be addressed in section 3.
- The amount of custom TI code should be kept to a minimum by using open source solutions wherever possible.
- There should not be any additional restrictions imposed by the TI GStreamer plugin on the types of pipelines created. For example, the video decode elements should be able to interface with existing video sinks—not just the video sink from our plugin. Likewise, our video sink should also accept buffers from open source ARM video decoders. All elements in the plugin should be interchangeable with ARM-side equivalents when needed.
- The open source community should be able to use the plugin, customize it to meet their needs beyond what is provided in the baseline implementation, and contribute back where it makes sense.

The TI GStreamer plugin provides baseline support for eXpressDSP™ Digital Media (xDM[1]) plug-and-play codecs and a video sink for using video drivers not supported by any open source plugin. Multiple xDM versions are supported, making it easy to migrate between codecs that conform to different versions of the xDM specification.

TI is not supporting the productization of the GStreamer plugin or GStreamer-based solutions. Complete products may require additional development for custom boards, features not specific to TI hardware (i.e., visual effects) or the implementation of applications that provide multimedia functionality through GStreamer. However, many components such as demuxers, media players, and other common features and applications are already available in various open source projects.

3 Considerations for Embedded Systems

When working in an embedded system there are usually fewer computation and memory resources available than in the typical desktop system. Following are some of the key resource considerations while implementing the TI GStreamer plugin.

Limited CPU Resources:

In an ARM+DSP system, the ARM is sufficient for running Linux, driving the peripherals and perhaps running a simple interface application or browser. However, in CPU-intensive multimedia applications that perform operations on complex media streams, the ARM is simply not powerful enough to do all of the work and still achieve real-time playback or encoding. To meet real-time requirements, the TI GStreamer plugin must utilize the DSP and other hardware accelerators to off-load the work required to process audio and video from the ARM processor.

Memory Copies are Expensive:

When processing audio and video it is often necessary to copy data between buffers. For example, when displaying a video frame on a display subsystem such as the frame buffer, it may be necessary to copy data into buffers provided by the device driver. Video frames can be quite large. If the normal system `memcpy` routine is used, it would create a significant load on the ARM processor since real-time playback can require 30 frames to be copied every second. Hardware acceleration for buffer copies must be used to keep the ARM load low enough to perform other tasks such as demuxing a stream or managing a media playback interface.

Parallelizing I/O Operations:

In an embedded system the I/O devices available are often slower than those available on a typical desktop system. Audio and video files may be stored on media such as NAND flash or SD/MMC cards. This means that the I/O wait times are longer and have more of an impact on real-time performance. On embedded systems, the I/O operations must be performed while the DSP is performing encode or decode operations to ensure the DSP always has available data.

4 Software Stack

Figure 1 depicts what the software stack looks like on an OMAP35x system running a GStreamer-based ap-

[1] TI's xDM specification defines a uniform set of APIs across various multimedia codecs to ease integration and ensure interoperability. xDM is built over TI's eXpress DSP Algorithm Interoperability Standard (also known as xDAIS) specification [7].

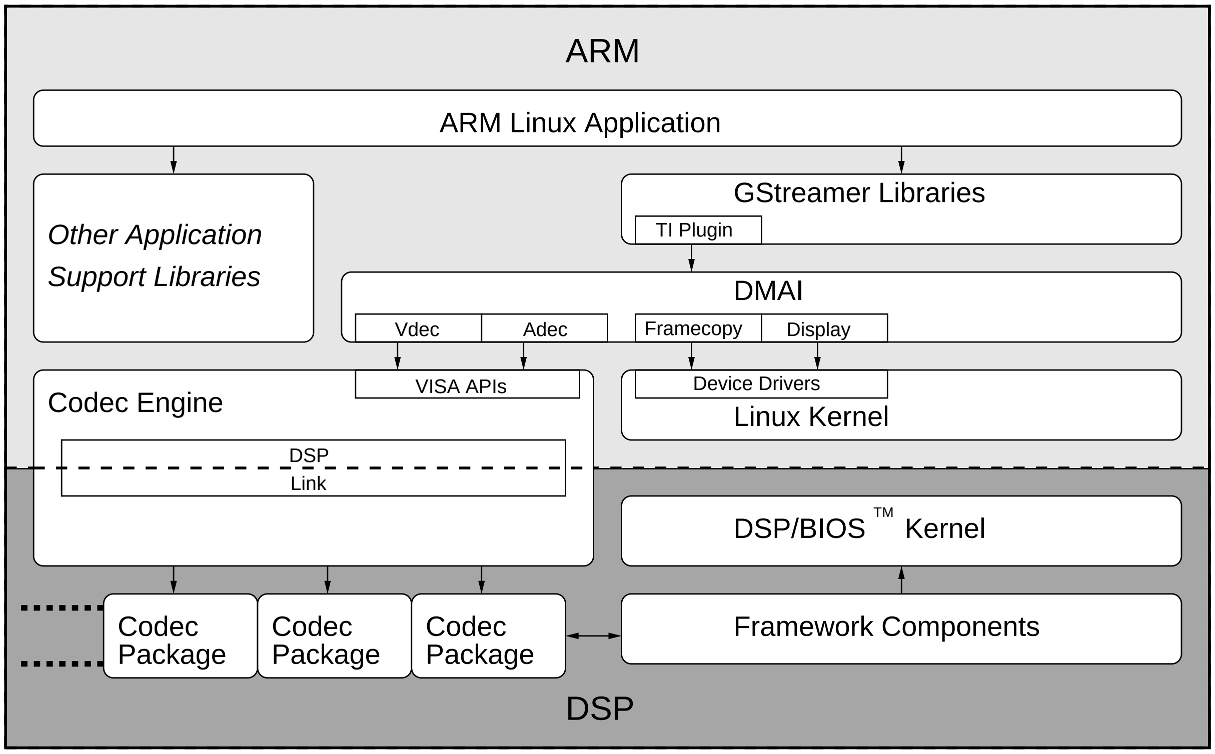


Figure 1: Software Stack for a GStreamer-based application using the TI GStreamer plugin.

plication. At the highest level there will be an ARM Linux application, such as a media player that is using the GStreamer library. At this level, developers familiar with Linux do not need to know a lot about programming for an embedded system. Other than cross-compiling their application, there are not a lot of differences between developing a GStreamer-based application on an OMAP35x and on a desktop system. The GStreamer library loads and interfaces with the TI GStreamer plugin, which handles all the details specific to the entitlement of OMAP35x hardware acceleration and use of the DSP. The core GStreamer library does not need to be aware of anything specific to the OMAP35x.

The TI GStreamer plugin interfaces with OMAP35x hardware using software components from the Digital Video Software Development Kit (DVSDK[2]). DVSDK components are all system tested for interoperability, providing a stable baseline for development. In the DVSDK software model, the DSP is mostly treated as a "black box" for running codecs—all peripherals are controlled using ARM-side Linux device drivers.

As part of the GStreamer framework, the TI GStreamer plugin also gains the ability to interface with many other open source GStreamer plugins that provide features such as:

- Demuxers for AVI, TS, and MP4 containers
- OSS and ALSA audio output
- V4L2 video capture
- ARM codecs including MP3 and AAC decoders

4.1 Portability and Reusability through the DaVinci™ Multimedia Application Interface (DMAI)

The most vital DVSDK component used by the TI GStreamer plugin is the DMAI [1], which enables portability to multiple TI platforms and newer DVSDK releases with minimal changes to the plugin code base. The interface with DMAI is also the boundary between the generic ARM Linux components and the DVSDK. DMAI directly and indirectly interfaces with all of the other software components of the DVSDK, providing a clean interface for interacting with hardware accelerators and DSP-side codecs. It should be noted that the

[2]DVSDK release notes and documentation are available from the TI web site [5].

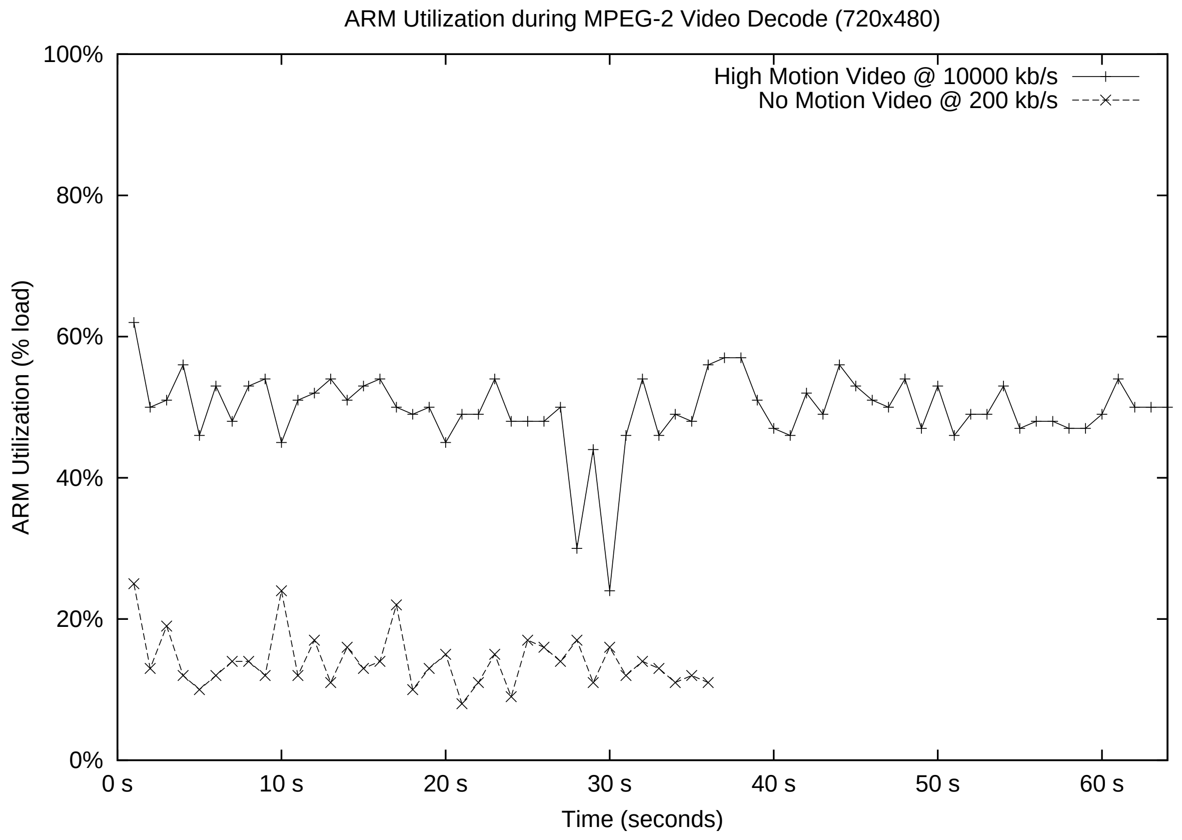


Figure 2: ARM utilization during MPEG-2 video decode.

TI GStreamer plugin also works on other TI platforms through use of the DMAI library. There is a single code base for the TI GStreamer plugin that is shared by all supported platforms.

The DMAI library provides a simple software interface but implements the many details of device driver and codec handshaking under the hood. It also provides a buffer abstraction that allows for the easy transfer of data between codecs, hardware accelerators and device drivers. Hardware acceleration is often provided without requiring developers to understand the platform-specific implementation details. For example, when using DMAI to perform a hardware-accelerated frame copy, DMAI can use a DMA operation on the OMAP35x, but will use the hardware resizer to perform a copy on platforms where a resizer could give better performance.

DMAI interfaces directly with the xDM interfaces of the available codecs and mostly abstracts out the differences between different xDM API versions. Where needed, it also abstracts out differences between device drivers and in some places, differences between kernel versions. For example, DMAI provides a display module that is configurable to use either the frame buffer or V4L2 API. The TI GStreamer plugin does not need any specialized code depending on the type of display it is using. Finally, DMAI aids in the error handling of low-level DVSDK components.

Since platform-specific code is abstracted by the DMAI library, the TI GStreamer plugin is mostly free of platform-specific code, making it extremely portable.

5 Performance

The graph in Figure 2 shows the ARM CPU utilization while decoding video files using the OMAP35x MPEG-2 DSP decoder. In this experiment, the decoder is run with two different NTSC-resolution video clips. The first video clip is designed to have zero-motion and a low bitrate to demonstrate the best-case ARM load. The second video clip is designed to have high-motion and a high bitrate to stress the system and demonstrate a worst-case ARM load.

Creation of MPEG-2 Test Files

No Motion @ 200kb/s:

```
$ gst-launch videotestsrc pattern=9 num-buffers=3600 ! \
      'video/x-raw-yuv, format=(fourcc)I420, width=720, height=480' ! \
      filesink location=sample_m2v.yuv
$ ffmpeg -pix_fmt yuv420p -s 720x480 -i sample_m2v.yuv -vcodec mpeg2video \
      -b 200000 sample_staticimage.m2v
```

High Motion @ 10000kb/s:

```
$ gst-launch videotestsrc pattern=1 num-buffers=3600 ! \
    'video/x-raw-yuv, format=(fourcc)I420, width=720, height=480' ! \
    filesink location=sample_m2v.yuv
$ ffmpeg -pix_fmt yuv420p -s 720x480 -i sample_m2v.yuv -vcodec mpeg2video \
    -b 10000000 sample_snow.m2v
```

Decode of MPEG-2 Test Files

No Motion @ 200kb/s:

```
$ gst-launch filesrc location=/mnt/sample_staticimage.m2v ! \
    TIViddec2 codecName=mpeg2dec engineName=decode ! fakesink
```

High Motion @ 10000kb/s:

```
$ gst-launch filesrc location=/mnt/sample_snow.m2v ! \
    TIViddec2 codecName=mpeg2dec engineName=decode ! fakesink
```

Figure 3: Steps to create and decode video test files for performance measurements.

Focus is put on MPEG-2 since it has a lower compression ratio than other video codecs. Since the ARM load is directly affected by the rate of data throughput, MPEG-2 is an upper-bound on the ARM load required to feed video data to the DSP codec. The same tests were performed with H.264 and MPEG-4 decoders yielding similar results at the same bitrates. Please note that H.264 and MPEG-4 exhibit better compression and their typical bitrate tends to be lower than MPEG-2.

On average, the ARM is not loaded more than 60 percent while decoding the high-bitrate video clip, and rarely went above 20 percent while decoding the low-bitrate clip. The decoders are allowed to run at maximum speed, and are not slowed down for real-time playback. This explains why the low-motion clip takes less time to decode than the high-motion clip, even though both clips are two minutes in duration. It should be noted that the video clips are read from an SD card, which contributes to part of the ARM load.

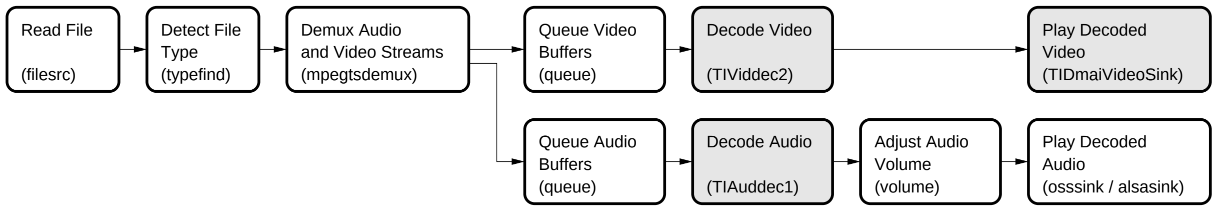
Figure 3 shows how GStreamer and FFmpeg[3] are used to create and decode the clips used for ARM load measurements.

6 Community Model

The TI GStreamer plugin is an open source project located at `http://gstreamer.ti.com` [4]. The project site provides a collaboration environment that includes:

- Source control using Subversion
- A wiki for documentation

[3]FFmpeg is a command-line utility for recording and converting audio and video streams. More information is available on the FFmpeg web site [2].



```
gst-launch filesrc location="video.ts" ! typefind ! mpegtsdemux name=demux \
    demux. ! 'video-x-h264' ! queue ! TIViddec2 ! TIDmaiVideoSink \
    demux. ! 'audio/mpeg' ! queue ! TIAuddec1 ! volume volume=5 ! alsasink
```

Figure 4: Example GStreamer pipeline. Shaded pipeline elements are provided by the TI GStreamer plugin.

- A package release system
- An issue and feature tracker
- Forums for support and discussion

An IRC channel (`#gst_ti`) is available on `irc.freenode.net` for developers interested in GStreamer on OMAP35x as well as other TI processors.

Anonymous access to the project and Subversion repository is supported. Account registration on the project site is optional but is needed for the submission of bug reports and patches. Developers interested in participating in the project can find answers to frequently asked questions, getting started guides and other project participation guidelines on the project site.

The TI GStreamer plugin project is community supported. TI is committed to enabling the community in their efforts to develop multimedia applications on TI processors using GStreamer. Community members are encouraged to use the forums and IRC channel to ask questions and discuss future development. For developers that want or need more support, a commercial support option is available from RidgeRun[4].

7 Plugin Design

Before diving into the design of the TI GStreamer plugin, an overview of the GStreamer pipeline model and how GStreamer plugins integrate into the framework should be discussed first.

[4]More information on RidgeRun is available on the RidgeRun web site [6]. Information on RidgeRun support for GStreamer is located at `http://www.ridgerun.com/products/gstreamer.shtml`.

7.1 The GStreamer Pipeline

A typical GStreamer pipeline starts with one or more source elements, uses zero or more filter elements and ends in a sink or multiple sinks. The example pipeline shown in Figure 4 demonstrates the demuxing and playback of a transport stream. An input file is first read using the `filesrc` element, parsed by the `typefind` element to ensure the input file is a transport stream, and then processed by the `mpegtsdemux` element, which demuxes the stream into its audio and video stream components. The video stream is sent through the `TIViddec2` element to decode the video using the DSP on the OMAP35x. Then it is finally sent to the `TIDmaiVideoSink` sink element to display the decoded video on the screen. The audio stream is processed by the `TIAuddec1` element to decode the audio on the DSP and reaches its destination at the `alsasink` or `osssink` element to play the decoded audio, depending on if the system uses an OSS sound driver or an ALSA sound driver.

Note that in the example pipeline, the TI GStreamer plugin is only contributing the `TIViddec2`, `TIAuddec1` and `TIDmaiVideoSink` elements. All other elements in the pipeline come from available open source plugins.

The main GStreamer distribution includes an application called `gst-launch`, which is a simple command line utility that allows you to construct and execute an arbitrary pipeline. It provides a flexible way to test pipelines without having to write entire GStreamer-based applications. The bottom half of Figure 4 shows the `gst-launch` command that would be used to construct and execute the pipeline shown.

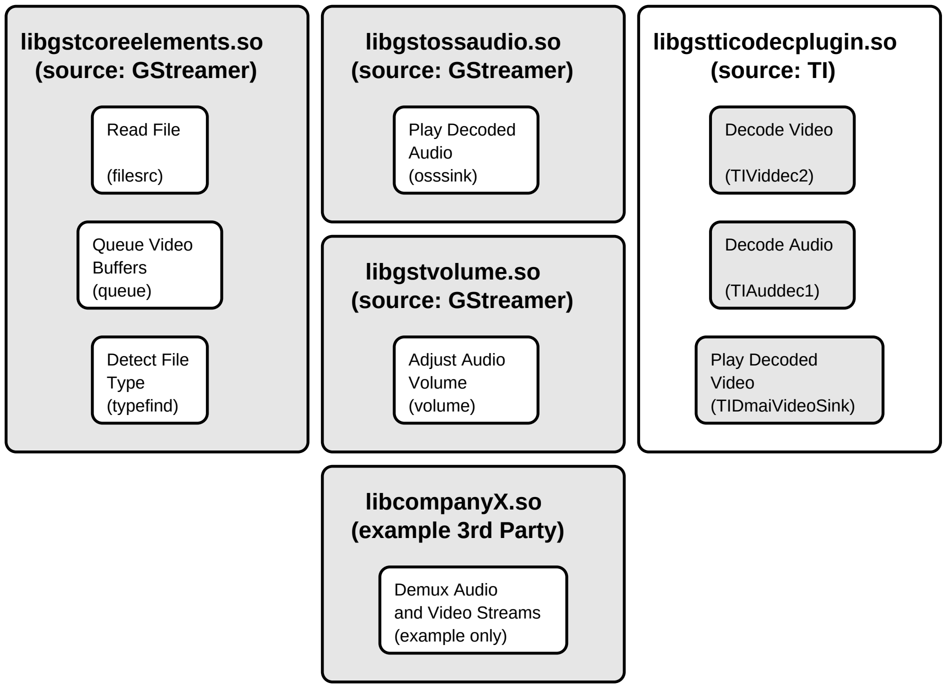


Figure 5: GStreamer pipeline elements and the the Linux shared objects that contain them. Shared object libraries may contain additional elements not shown here.

7.2 Shared Object Libraries

GStreamer filter elements are interchangeable, making it easy to perform different operations on a data stream. Further, GStreamer only needs to load into memory the plugins that contain elements for the desired pipeline, saving valuable system resources.

A GStreamer plugin typically maps to one or more shared object libraries on the Linux file system (see Figure 5). A single shared object library contains one or more pipeline elements. When a GStreamer-based application starts, it searches the shared object libraries for available elements. These shared object libraries can come from GStreamer itself or from other parties that provide custom GStreamer elements. The TI GStreamer plugin provides a shared object library that contains all of the pipeline elements that use the DSP and other hardware accelerators on the OMAP35x. These elements can connect and interact with pipeline elements from the main GStreamer base and from other third parties.

7.3 Decode Element Design

DSP decode algorithms require input buffers to be located in physically-contiguous memory and to have a full frame available for processing prior to being invoked. Physically-contiguous memory is allocated from memory regions shared by the ARM and DSP, allowing data to be passed between them without additional copy operations. However, these requirements also pose a problem as input buffers to the decode elements can be allocated from regular system memory and in some cases do not hold a complete frame. In order to use the decoder the input data must be prepared first so it is in a form usable by the DSP.

The `TIViddec2` decode element is implemented using two sub-threads (see Figure 6). The queue thread is in charge of preparing the input data for the DSP, and the decode thread invokes the DSP decoder when data is available for processing. The decode thread is a real-time thread to minimize the DSP idle time when there is data available for it to process. The queue thread copies incoming buffers into a physically-contiguous buffer for the DSP decoder. When there is enough data available to satisfy the DSP, the decode thread is signaled and DSP decoder is invoked. Since the code driving the DSP is in a separate thread, the queue thread continues to copy additional buffers into the physically-contiguous buffer while the DSP is running. When the DSP is finished, the decode thread pushes the decoded video frame to the downstream pipeline element.

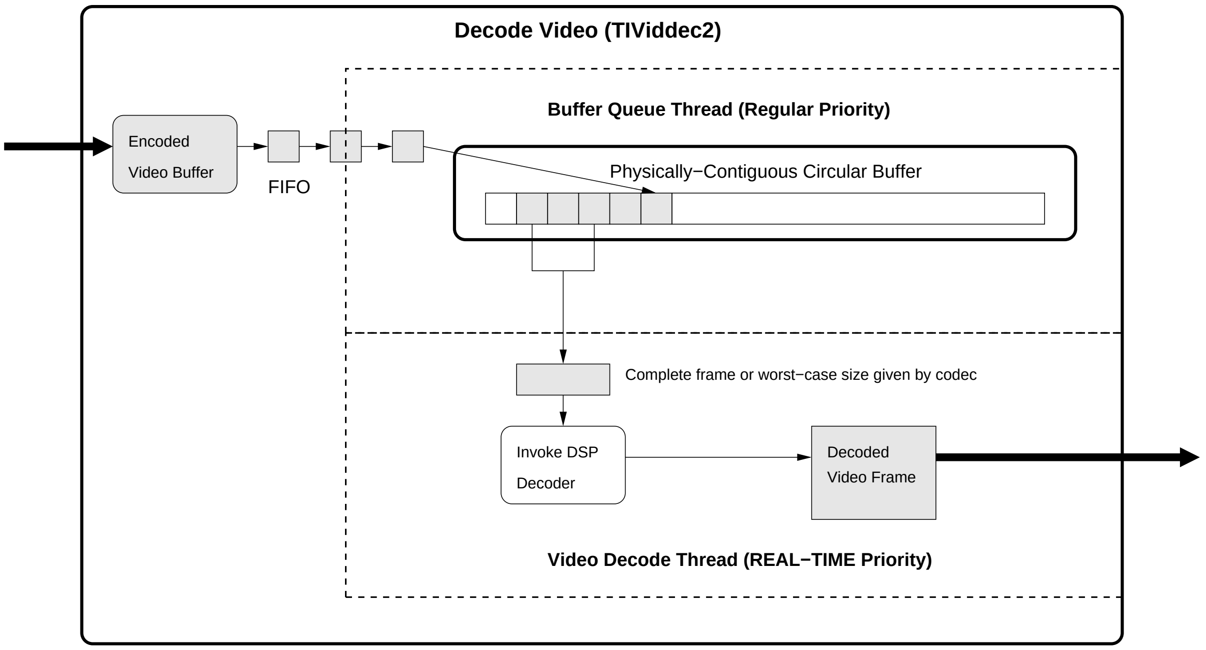

Decode Video (TIViddec2)
Buffer Queue Thread (Regular Priority)
Encoded Video Buffer
FIFO
Physically−Contiguous Circular Buffer
Complete frame or worst−case size given by codec
Invoke DSP Decoder
Decoded Video Frame
Video Decode Thread (REAL−TIME Priority)

Figure 6: TIViddec2 decode element design.

In some cases it cannot be determined when enough input buffers have been received to guarantee a complete frame is available for calling the DSP decoder. In these cases, the decoder specifies the worst-case amount of data needed before it can be invoked. If more data is passed to the decoder than is actually needed, the decoder returns how much data was consumed so the next time it is invoked the remaining unprocessed data can be given again, along with additional data queued up since the last invocation. To effectively handle this scenario, the physically-contiguous buffer is managed as a circular buffer. This allows the plugin to simply pass the location where the DSP should start processing the next frame and eliminates any need to copy unprocessed data into a new buffer.

All audio, video and imaging decode elements operate in the same manner as the `TIViddec2` video decode element.

7.4 Encode Element Design

The design of the `TIVidenc1` encode element is very similar to the `TIViddec2` decode element, but it has one minor difference. When performing an encode operation, there is a potential opportunity for a hardware-accelerated copy of the input buffer when it is known to come from a capture source element. In this case, the input buffer is already physically-contiguous in memory, which allows a hardware-accelerated copy into the element's physically-contiguous buffer. Input buffers received from a capture source also meet the requirements to be passed directly to the DSP. However, a copy is still needed as the capture source has few resources, and there is a risk of starvation if the input buffer is not released as soon as possible.

Although a capture source will typically give a complete frame in each input buffer, the physically-contiguous buffer in the encode element is still managed as a circular buffer. This enables support for encoding a stream from a file or network source where input buffers do not contain full frames. In this case, the encode element operates in an almost identical manner to the decode element.

All video and imaging encode elements operate in the same manner as the `TIVidenc1` video encode element.

7.5 Video Sink Design

When the `TIDmaiVideoSink` element receives its first decoded frame, it uses the metadata in the buffer to configure the display device. Several display sink properties are also configurable by GStreamer-based applications to control the selection of frame buffer or V4L2 output, display resolution, video standard and others.

TI GStreamer Plugin Elements	
Element Name	Description
`TIAuddec1`	Audio decoder for xDM 1.x codecs
`TIAuddec`	Audio decoder for xDM 0.9 codecs
`TIDmaiVideoSink`	Display sink for frame buffer and V4L2 display subsystems
`TIImgdec1`	Image decoder for xDM 1.x codecs
`TIImgdec`	Image decoder for xDM 0.9 codecs
`TIImgenc1`	Image encoder for xDM 1.x codecs
`TIImgenc`	Image encoder for xDM 0.9 codecs
`TIViddec2`	Video decoder for xDM 1.x codecs
`TIViddec`	Video decoder for xDM 0.9 codecs
`TIVidenc1`	Video encoder for xDM 1.x codecs
`TIVidenc`	Video encoder for xDM 0.9 codecs

Table 1: TI GStreamer Plugin Elements. DSP codecs currently available for OMAP35x all use the xDM 1.x specification, so a typical GStreamer pipeline on OMAP35x would use `TIViddec2`, `TIAuddec1` and `TIImgdec1` for decode and `TIVidenc1`, `TIImgenc1` for encode.

If no configuration parameters are specified to the `TIDmaiVideoSink` element, it uses reasonable defaults for OMAP35x based on recommendations from the DMAI library. It can optionally calculate the best supported resolution to fit the video clip being played.

Input buffers that come from the `TIViddec2` element can be detected by the video sink, in which case means the input buffers are physically-contiguous and hardware-acceleration is used to copy the input buffers into display buffers. If the input buffers do not come from `TIViddec2`, a regular `memcpy` is used to copy the input buffer.

8 Feature Summary and Future Work

A complete list of the elements provided by the TI GStreamer plugin is shown in Table 1. Support for audio encode is still missing but will be addressed in a future release. DSP codecs currently available for OMAP35x all use the xDM 1.x specification, so a typical GStreamer pipeline on OMAP35x would use `TIViddec2`, `TIAuddec1` and `TIImgdec1` for decode and `TIVidenc1`, `TIImgenc1` for encode. Elements that support xDM 0.9-based codecs are listed for completeness. Future work may eliminate the need for GStreamer-based applications to know which xDM version is used by the codecs. The TI GStreamer plugin should be up-to-date with support for new TI processors, updated DVSDK components and updated GStreamer base components. The complete list of planned work is available on the project site.

References

[1] DaVinci™ Multimedia Application Interface. Project site: `https://gforge.ti.com/gf/project/dmai/`.

[2] FFmpeg. Project site: `http://www.ffmpeg.org/`.

[3] GStreamer Open Source Multimedia Framework. Project site: `http://gstreamer.freedesktop.org/`.

[4] GStreamer on TI DaVinci™ and OMAP™ processors. Project site: `http://gstreamer.ti.com/`.

[5] OMAP3530 Digital Video Software Development Kit (DVSDK) 3.00.00.29 Release Notes. Online documentation: `https://www-a.ti.com/downloads/sds_support/targetcontent/dvsdk/oslinux_%dvsdk/v3_00_3530/exports/omap3530_3_00_00_29_release_notes.pdf`. Free login account required for viewing.

[6] RidgeRun – Embedded Solutions. Company site: `http://www.ridgerun.com/`.

[7] Texas Instruments, Inc. *xDAIS-DM (Digital Media) User Guide*, January 2007. Literature Number: SPRUEC8B.

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada