Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering
InES Institute of Embedded Systems

4K HEVC Video Processing with GPU Optimization on Jetson TX1

Tobias Kammacher, Matthias Frei, Matthias Rosenthal

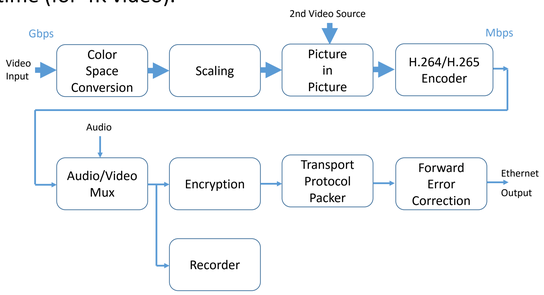
Zurich University of Applied Sciences

High Performance Multimedia and Data Acquisition Research Group

Learn how to capture and process 4K video (HEVC encoding, scaling, mixing) on the TX1 and how to integrate the powerful GPU for complex tasks. 4K video previously required custom hardware or high-performance desktop processors. The heterogeneous system architecture of the TX1 allows to process these tasks on a single chip. The main challenges lie in the optimal utilization of the different hardware resources of the TX1 (CPU, GPU, dedicated hardware blocks) and in the software frameworks.

Streaming Use Case and Video Capture

In order to stream video a set of functions has to be applied to the captured video input. These functions have to perform sufficiently to process data rates of multiple Gbps in real time (for 4K video).



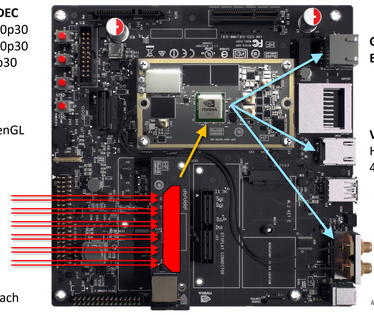
The TX1 provides 12 MIPI CSI-2 lanes which are capable of processing up to 1.5 Gbps of video input data per lane. The TX1 provides powerful hardware-accelerated video scaling/conversion, a H.264/265 CODEC and the CPU/GPU for video processing.

Data Flow for Processing / Streaming Video Input

Video CODEC
H.265 2160p30
H.264 2160p30
VP8 2160p30
GPU
256 Core
Maxwell
CUDA, OpenGL
CPU

Video Input
12 x
CSI-2
Lanes

Max
1.5 Gbps each



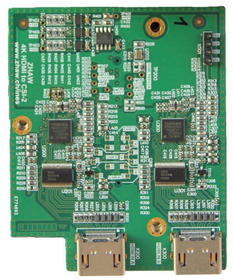
Gigabit Ethernet

Video Output
HDMI 2.0
4096x2160 60Hz

WIFI
802.1ac 2x2

Image: anandtech.com

Our Custom HDMI-to-CSI converter Board that plugs into the TX1

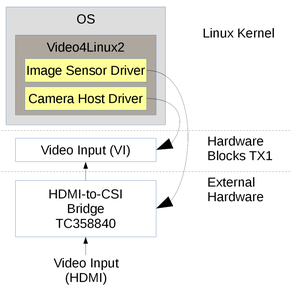


For our use case we had to capture 4K and 1080p simultaneously from two HDMI sources. The Toshiba TC358840 is a HDMI-to-CSI bridge IC, which can convert 4K HDMI inputs to 8 lanes CSI-2. Officially MIPI CSI-2 only supports up to 4 lanes in parallel. Therefore it was necessary to expand the drivers and implement a 4+4 lane «dual link» configuration.

In terms of software involved, video capturing is handled by the Video4Linux2 (V4L2) framework as part of the Linux kernel. Three primary drivers are necessary:

- Camera Host Driver: *tegra_vi2* (or *vi2*)
- Image Sensor / Subdevice Driver: *tc358840*
- Video Buffer: *videobuf2*

On the basis of the Linux4Tegra kernel, «Avionic Design» has developed a driver for capturing 1080p60 on the TK1. We have ported the capture driver to the TX1 and also adapted the sensor driver from the predecessor IC to the tc358840. This allows us to capture 4K and 1080p60 at the same time and further process the video with the TX1 platform and our custom HDMI input board.



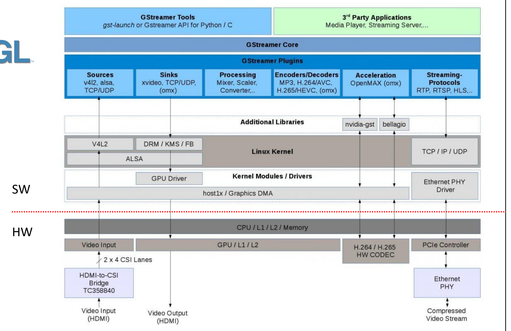
Blog: blog.zhaw.ch/high-performance
Website: ines.zhaw.ch
E-Mail: rosn@zhaw.ch / kamm@zhaw.ch

Check out our Driver Implementation on Github:
github.com/ines-hpmm

Video Processing with GStreamer

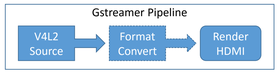
Captured video frames are passed to user space, where they can be processed by any application supporting the V4L2 API. Nvidia primarily supports GStreamer, which is a pipeline-based multimedia framework. Any audio/video processing application can be put together by combining plugins. Each plugin performs one well-defined task.

Nvidia provides hardware-acceleration for video scaling, format conversion and rendering to HDMI out. There are also general-purpose plugins that use OpenGL to provide GPU acceleration for scaling, conversions and rendering.



The method used for passing video buffers through the pipeline has a severe impact on the overall performance. These methods are called IO-modes:

- **MMAP**: Memory allocated by driver and memory-mapped into user space
- **Userptr**: Memory allocated by user space and passed to driver (DMA required)
- **DMAbuf**: Memory allocated by one driver; accessible from a different driver



A simple example for a GStreamer pipeline is capturing video from a HDMI source, (possibly performing a format conversion) and rendering it.

A few examples for different methods have been compared. The first one uses the *MMAP mode* and profits from the hardware-accelerated format conversion and rendering. The disadvantage is that the video data needs to be copied to the proprietary NVMM memory and can then not be modified easily.

```
gst-launch-1.0 v4l2src io-mode=2 ! 'video/x-raw, format=UYVY' !
nvvidconv ! 'video/x-raw(memory:NVMM), format=I420' ! nvoverlaysink
```

Alternatively the *Userptr mode* can be used, which has the advantage that buffers are passed between GStreamer plugins without the need to copy the data. Therefore CPU load is reduced drastically. An additional advantage is, that the X-Server video sink can be used, which is able to directly process the UYVY format (thus no format conversion is necessary).

```
gst-launch-1.0 v4l2src io-mode=3 ! 'video/x-raw, format=UYVY' !
xvimagesink
```

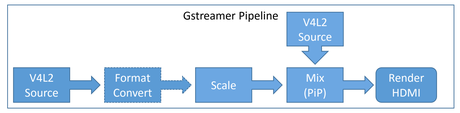
For the example above for capturing 4K video and rendering it, an overview over the CPU load is given in the table below. The best performance is achieved using the Userptr mode and xvimagesink.

Pipeline \ IO Mode	R/W	MMAP	Userptr	DMAbuf
xvimagesink	30 fps 100% CPU	20 fps 70% CPU	**30 fps** **30% CPU**	**20 – 30 fps** **100% CPU ***
videoconvert & nvoverlaysink	Not supported	6 fps 180% CPU	30 fps 50% CPU	6 fps 180% CPU
nvvidconv & nvoverlaysink (with NVMM)	26 fps 80% CPU	**30 fps** **80% CPU**	Not supported	Not supported
nvvidconv(NVMM) & nvvidconv & xvimagesink	18 fps 85% CPU	20 fps 80% CPU	Not supported	Not supported
GStreamer Version	**1.2.4**	**1.2.4**	**1.8.0**	**1.8.0**

100% CPU = 1 CPU core
* Some Buffers missed

Complex Video Processing and GPU Optimization

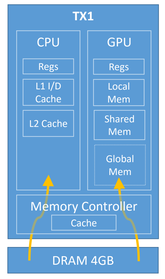
For complex video processing tasks, the Gstreamer plugins that are available by default may not perform sufficiently. In our use case we want to scale and mix a 4K and a 1080p video source. This can be done with the GStreamer *compositor* plugin, but the CPU load is very high and the resulting performance is around 1 FPS. Using OpenGL accelerated plugins, the performance can be improved to around 7 FPS.



To improve performance even more, a custom plugin for mixing two video sources can be implemented using the GPU.

On the TX1 platform the CPU and GPU share DRAM and all memory access is handled by a common memory controller. The different methods for accessing memory from the GPU/CUDA are:

- Unified Virtual Addressing
  - Single virtual address space for CPU, GPU
  - Memory copy required (Duplication)
- Zero Copy (Memory Mapped)
  - Pinned memory accessed via GPU DMA
  - Always uncached
- Unified Memory (Managed Memory Pool)
  - Automatically migrate data between host and device
  - Cache operations



An example plugin was implemented in C/CUDA for mixing a 1080p overlay on top of a 4K video stream. The implementation was tested with different memory access methods. The results for processing a single frame are given below:

Unified Virtual Addressing *
Step 1: cudaMemcpy() to GPU (12.5 ms)
Step 2: Execute kernel (9-11 ms)
Step 3: cudaMemcpy() to host (7.2 ms) **-> Total: 30 ms**

Zero Copy (Memory Mapped)
Step 1: cudaMallocHost(): Allocate memory on host **
Step 2: Execute kernel (23.5 – 25.7 ms) **-> Total: 25 ms**

Managed Memory
Step 1: cudaMallocManaged(): Allocate shared memory **
Step 2: Execute kernel (9-11 ms)
Step 3: synchronize with CPU (0.2 ms) **-> Total: 10 ms**

* Upload 4K + 1080p, Download 4K
** One time only operation

The example shows that the memory access method has a noticable influence on performance and that the best method depends on the use case. For mixing two video streams, *Managed Memory* is the preferred method. On the other hand for while for overlaying a fix logo onto a video stream, *UVA* may be preferrable because the logo image has to be uploaded to the GPU only once.

In conclusion we found that high data rate video processing tasks are best handled by the GPU. Encoding can be done very efficiently by the hardware-accelerated CODEC and for additional functionality the CPU provides the highest flexibility at low data rates.