Hardware accelerated video streaming with V4L2

on i.MX6Q

05/01/2014

Gabriel Huau

Embedded software engineer
SESSION OVERVIEW

1. Introduction

2. Simple V4L2 application

3. V4L2 application using OpenGL

4. V4L2 application using OpenGL and vendor specific features

5. Conclusion
• Embedded Software Engineer at **Adeneo Embedded** (Bellevue, WA)
  ▶ Linux / Android
    ♦ BSP Adaptation
    ♦ Driver Development
    ♦ System Integration
  ▶ Former U-Boot maintainer of the Mini2440
Introduction
WHAT'S V4L2?

- Video For Linux version 2
- Common framework
- API to access video devices (/dev/videoX)
- Not only video: audio, controls (brightness/contrast/hue), output, ...
SET YOUR GOALS

- Resolution: HD, full HD, VGA, …
- Frame rate to achieve: does it matter?
- Image processing: rotation, scaling, post processing effects, …
- Hardware availability:
  - CPU performances
  - GPU
  - Image Processing IP (IPU, DISPC, …)
WHY ARE WE HERE?

- V4L2 application development
- Optimization process and trade-offs
- Showing real customer solutions
HARDWARE SELECTION

- Freescale i.MX6Q SabreLite
- Popular platform
- Geared towards multimedia
Simple V4L2 application
ARCHITECTURE

Query Capabilities → Cropping Area → Video Format

Request Buffer → Query Buffer → Start Streaming

Dequeue → Rendering → Queue
Different ways to handle video capture buffers:

- **V4L2_MMAP**: memory mapping => allocated by the kernel
- **V4L2_USERPTR**: user memory => allocated by the user application
- **Others**: DMABUF, read/write

Only MMAP will be covered in this presentation.

**Warning**

Drivers don't necessarily support every method
Query capabilities:

```c
ioctl(fd, VIDIOC_QUERYCAP, &cap);

if (!(cap.capabilities & V4L2_CAP_VIDEO_CAPTURE))
    exit(EXIT_FAILURE);

if (!(cap.capabilities & V4L2_CAP_STREAMING))
    exit(EXIT_FAILURE);
```

**Warning**

Every V4L2 driver does not necessarily support both Streaming and Video Capture
ARCHITECTURE

Reset cropping area:

1. `ioctl(fd, VIDIOC_CROPCAP, &cropcap);`
2. `crop.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;`
3. `crop.c = cropcap.defrect;`
4. `ioctl(fd, VIDIOC_S_CROP, &crop);`

The area to capture/view needs to be defined
ARCHITECTURE

Set video format:

1. `fmt.fmt.pix.width = WIDTH;`
2. `fmt.fmt.pix.height = HEIGHT;`
3. `fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_NV12;`
4. `fmt.fmt.pix.field = V4L2_FIELD_ANY;`
5. `ioctl(fd, VIDIOC_S_FMT, &fmt);`

Warning

`VIDIOC_ENUM_FRAMESIZES` should be used to enumerate supported resolution
ARCHITECTURE

Request buffers:

1. `req.count = 4;
2. `req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
3. `req.memory = V4L2_MEMORY_MMAP;
4. `ioctl(v4l2_fd, VIDIOC_REQBUFS, &req);

4 capture buffers need to be allocated to store video frame from the camera
Query buffers:

```c
for (n_buffers = 0; n_buffers < req.count; n_buffers++) {
    buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    buf.memory = V4L2_MEMORY_MMAP;
    buf.index = n_buffers;

    ioctl(v4l2_fd, VIDIOC_QUERYBUF, &buf);
    buffers[n_buffers].length = buf.length;
    buffers[n_buffers].start = mmap(NULL, buf.length,
                                    PROT_READ | PROT_WRITE, MAP_SHARED,
                                    v4l2_fd, buf.m.offset);
}
```

- Memory information such as size/addresses need to be retrieved and stored in the User Application
- Need to keep a mapping between V4L2 index buffers and memory information
Start capturing frames:

```c
for (i = 0; i < n_buffers; ++i) {
    buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    buf.memory = V4L2_MEMORY_MMAP;
    buf.index = i;
    ioctl(v4l2_fd, VIDIOC_QBUF, &buf);
}

type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
ioctl(v4l2_fd, VIDIOC_STREAMON, &type);
```

Capture buffers need to be queued to be filled by the V4L2 framework
ARCHITECTURE

Rendering loop:

1 /* Dequeue */
2 ioctl(v4l2_fd, VIDIOC_DQBUF, &buf);
3
4 /* Conversion from NV12 to RGB */
5 frame = convert_nv12_to_rgb(buffers[buf.index].start);
6 display(frame);
7
8 /* Queue buffer for next frame */
9 ioctl(v4l2_fd, VIDIOC_QBUF, &buf);

Framebuffer pixel format is RGB
DEMONSTRATION
CONCLUSION

Advantages:

- Easy to implement

Drawbacks:

- Poor performances
- Cannot do any 'real time' geometric transformation (rotation/scaling)
V4L2 application using OpenGL
ARCHITECTURE

What we had:

1. Dequeue
2. Conversion by CPU
3. memcpy to framebuffer
4. Queue
What we are going to do:

- Using GPU with OpenGL
- Do the conversion on the GPU via shader
SHADERS

- GPU process unit
- Different types: Vertex, Fragment, Geometry
- Piece of code executed by the GPU
- Vertex shader: draw shapes (quad, triangles, ...)
- Fragment shader: transform every pixel (YUV conversion for example) => has access to OpenGL textures
Generate two textures for planar Y and UV:

1. `glGenTextures (2, textures);

- A Texture is an image container for the GPU
- No 'standard' support in OpenGL for YUV texture
/* Dequeue */

1  glActiveTexture(GL_TEXTURE0);
2  /* Y planar */
3  glBindTexture(GL_TEXTURE_2D, textures[0]);
4  /* Queue */
5  glTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE, width, height, 0,
6       GL_LUMINANCE, GL_UNSIGNED_BYTE, in);

• Map the first texture (Y planar) to an OpenGL internal format => GL_LUMINANCE

• GL_LUMINANCE has a size of 8 bits, exactly as the Y planar!
/* Dequeue */

```c
3  glActiveTexture(GL_TEXTURE1);
4  /* UV planar */
5  in += (width*height);
6  glBindTexture(GL_TEXTURE_2D, textures[1]);
7  glTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE_ALPHA, width/2,
               height/2, 0, GL_LUMINANCE_ALPHA, GL_UNSIGNED_BYTE, in);
```

/* Queue */

- Map the second texture (UV planar) to an OpenGL internal format => GL_LUMINANCE_ALPHA
- GL_LUMINANCE_ALPHA has a size of 16 bits, exactly as the UV planar!
- Shaders have everything now!
Example of vertex shader:

```cpp
void main(void) {
    opos = texpos;
    gl_Position = vec4(position, 1.0);
}
```

- `opos` is the texture position => pass to the Fragment Shader for color conversion.
- `gl_Position` is the vertex position.
Example of fragment shader:

```c
void main(void) {
    yuv.x = texture2D(Ytex, opos).r;
    yuv.yz = texture2D(UVtex, opos).ra;
    yuv += offset;
    r = dot(yuv, rcoeff);
    g = dot(yuv, gcoeff);
    b = dot(yuv, bcoeff);
    gl_FragColor = vec4(r, g, b, 1);
}
```

- `texture2D(Ytex, opos).r` => `GL_LUMINANCE` texture
- `texture2D(Ytex, opos).ra` => `GL_LUMINANCE_ALPHA` texture
- Do the conversion using the GPU
To summarize:

- Copy V4L2 buffer to OpenGL textures
- Vertex Shader: draw a quad => the viewport
- Fragment Shader: convert and fill the quad/triangles => the video
- Display the frame
DEMONSTRATION
CONCLUSION

Advantages:

- Decent performances
- Can handle geometric transformation (rotation/scaling)
- Relax the CPU load
- Generic solution (if your board has a GPU ...)

Drawbacks:

- Need some OpenGL skills
V4L2 application using OpenGL and vendor specific features
ARCHITECTURE

What we had:

1. Create two textures (Y/UV)
2. Initialize Vertex/Fragment shader
3. Dequeue
4. memcpy frame to OpenGL textures
5. Convert YUV to RGB by GPU
6. Render frame
7. Queue
ARCHITECTURE

What we are going to do:

- Handle YUV OpenGL Texture directly => no need the conversion by shader anymore!
RENDERING LOOP

1 /* Get a GPU pointer */
2 glGenTextures(1, &tex);
3
4 glGenBuffers(1, &buf);
5
6 glBindTexture(GL_TEXTURE_2D, textures[0]);
7 glBindBuffer(GL_ARRAY_BUFFER, buffers[buf.index].start);
8
9 /* Dequeue */
10 ... 
11
12 /* Queue */
13 ... 
14

- pTexel is a pointer directly to a GPU memory
- Conversion is done by the GPU before processing shaders
- Handle different YUV formats
**SHADERS UPDATE**

**Vertex shader:**

```cpp
void main(void) {
    opos = texpos;
    gl_Position = vec4(position, 1.0);
}
```

**Fragment shader:**

```cpp
void main(void) {
    yuv = texture2D(YUVtex, opos);
    gl_FragColor = vec4(yuv, 1);
}
```
To summarize:

- Copy V4L2 buffer to OpenGL textures
- Vertex Shader: draw a quad => the viewport
- Fragment Shader: fill the quad => the video
- Display the frame
ARCHITECTURE

What we had:

1. Create a YUV texture
2. Initialize Vertex/Fragment shader
3. Dequeue
4. memcpy frame to OpenGL textures
5. Render frame
6. Queue
What we are going to do:

- Remove memcpy by using DMA

1. Create a YUV texture
2. Initialize Vertex/Fragment shader
3. Dequeue
4. Map Physical/Virtual memory to GPU (DMA)
5. Render frame
6. Queue
RENDERING LOOP

1 /* Dequeue */
2 ...
3
4 glBindTexture (GL_TEXTURE_2D, textures[0]);
5 /* Physical and Virtual addresses */
6 glTexDirectVIVMap(GL_TEXTURE_2D, width, height, GL_VIV_NV12, &
   buffers[buf.index].start, &(buffers[buf.index].offset));
7 glTexDirectInvalidateVIV(GL_TEXTURE_2D);
8
9 /* Queue */
10 ...

- No more memcpy()
- GPU knows the physical address in RAM
To summarize:

- Copy V4L2 buffer to OpenGL textures by using the DMA
- Vertex Shader: draw a quad => the viewport
- Fragment Shader: fill the quad => the video
- Display the frame
DEMONSTRATION
CONCLUSION

Advantages:

- No more memory copy (memcpy)
- Good performances: can handle fullHD (1080p) at 60FPS
- Handle geometric transformation (rotation/scaling)
- Application is less complex => no conversion code needed anymore

Drawbacks:

- Need some OpenGL skills and GPU API
CONCLUSION

• Highly hardware dependent
• Other hardware solutions: IPU (Image Processing Unit), DISPC (Display Controller), ...
• GStreamer support and features
QUESTIONS?
REFERENCES

- Fourcc: http://www.fourcc.org/
- Kernel Documentation: https://www.kernel.org/v4l2-framework.txt
- Freescale GPU VDK
- GStreamer for i.MX: https://github.com/Freescale/gstreamer-imx.git