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

ABOUT THE PRESENTER

- 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?

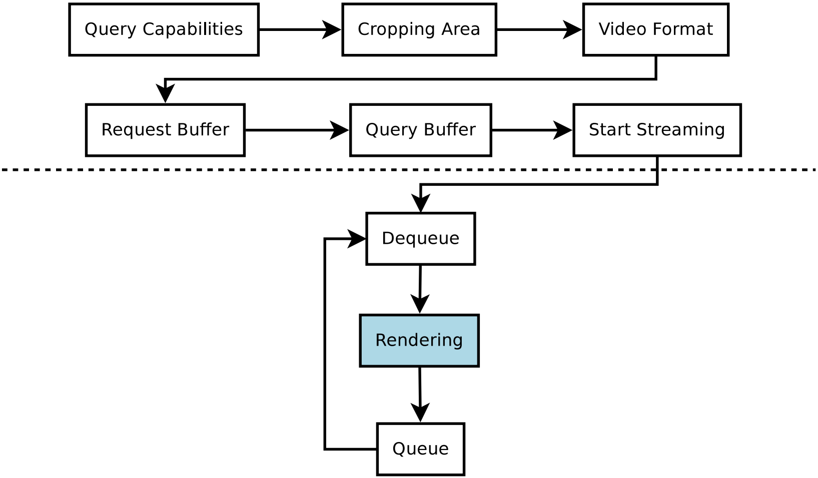
- V4L2 application development
- Optimization process and trade-offs
- Showing real customer solutions

HARDWARE SELECTION

- Freescale i.MX6Q SabreLite
- Popular platform
- Geared towards multimedia

Simple V4L2 application

ARCHITECTURE



MEMORY MANAGEMENT

Different ways to handle video capture buffers:

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

Only MMAP will be covered in this presentation.

Warning

Drivers don't necessarily support every method

ARCHITECTURE

Query capabilities:

```
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

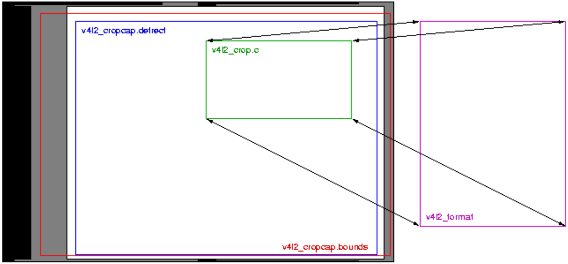
ARCHITECTURE

Reset cropping area:

```
ioctl(fd, VIDIOC_CROPCAP, &cropcap);

crop.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
crop.c = cropcap.defrect;
ioctl(fd, VIDIOC_S_CROP, &crop);
```

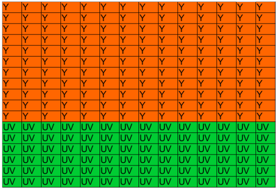
The area to capture/view needs to be defined



ARCHITECTURE

Set video format:

```c
fmt.fmt.pix.width = WIDTH;
fmt.fmt.pix.height = HEIGHT;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_NV12;
fmt.fmt.pix.field = V4L2_FIELD_ANY;
ioctl(fd, VIDIOC_S_FMT, &fmt);
```



Warning

VIDIOC_ENUM_FRAMESIZES should be used to enumerate supported resolution

ARCHITECTURE

Request buffers:

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

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

ARCHITECTURE

Query buffers:

```
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/adresses need to be retrieved and stored in the User Application
- Need to keep a mapping between V4L2 index buffers and memory information

ARCHITECTURE

Start capturing frames:

```
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:

```
/* Dequeue */
ioctl(v4l2_fd, VIDIOC_DQBUF, &buf);

/* Conversion from NV12 to RGB */
frame = convert_nv12_to_rgb(buffers[buf.index].start);
display(frame);

/* Queue buffer for next frame */
ioctl(v4l2_fd, VIDIOC_QBUF, &buf);
```

Framebuffer pixel format is RGB

DEMONSTRATION

CONCLUSION

Advantages:

- Easy to implement

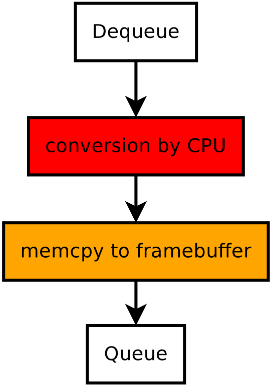
Drawbacks:

- Poor performances
- Cannot do any 'real time' geometric transformation (rotation/scaling)

V4L2 application using OpenGL

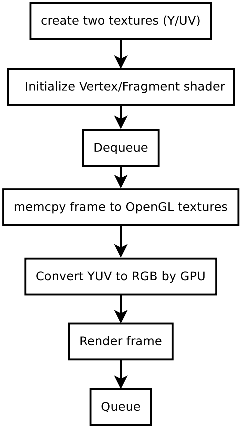
ARCHITECTURE

What we had:



ARCHITECTURE

What we are going to do:



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

SHADERS

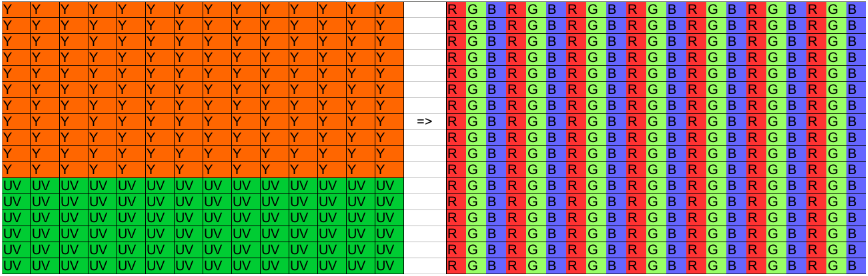
- GPU process unit
- Different types: Vertex, Fragment, Geometry
- Piece of code executed by the GPU
- Vertex shader: draw shapes (quad, triangles, ...)
- Fragment shader: transform every pixel (YUV conversion for example) => has access to OpenGL textures

TEXTURES

Generate two textures for planar Y and UV:

```
glGenTextures (2, textures);
```

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



RENDERING LOOP

```
/* Dequeue */

glActiveTexture(GL_TEXTURE0);
/* Y planar */
glBindTexture(GL_TEXTURE_2D, textures[0]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE, width, height, 0,
    GL_LUMINANCE, GL_UNSIGNED_BYTE, in);

/* Queue */
```

- 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!

RENDERING LOOP

```
/* Dequeue */

glActiveTexture(GL_TEXTURE1);
/* UV planar */
in += (width*height);
glBindTexture(GL_TEXTURE_2D, textures[1]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE_ALPHA, width/2,
    height/2, 0, GL_LUMINANCE_ALPHA, GL_UNSIGNED_BYTE, in);

/* Queue */
```

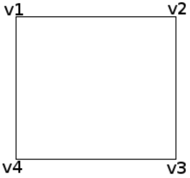
- Map the second texture (UV planar) to an OpenGL internal format => GL_LUMINANCE_ALPHA
- GL_LUMINANCE_ALPHA has a size of 16 bits, exactly as the UV planar!
- Shaders have everything now!

SHADERS

Example of vertex shader:

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

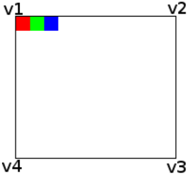
- *opos* is the texture position => pass to the Fragment Shader for color conversion.
- gl_Position is the vertex position.



ARCHITECTURE

Example of fragment shader:

```
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

ARCHITECTURE

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 ...)

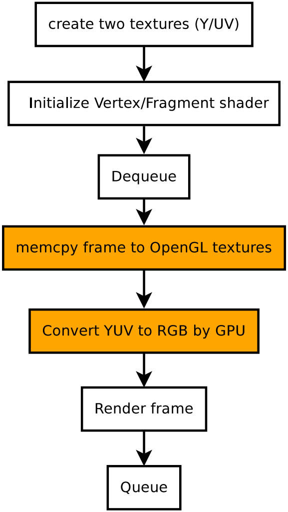
Drawbacks:

- Need some OpenGL skills

V4L2 application using OpenGL and vendor specific features

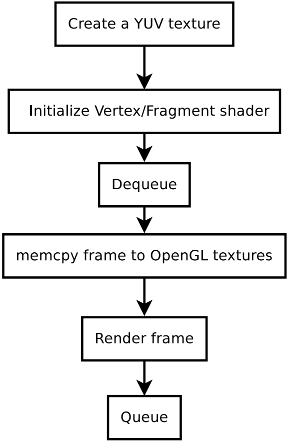
ARCHITECTURE

What we had:



ARCHITECTURE

What we are going to do:



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

RENDERING LOOP

```
/* Get a GPU pointer */
glTexDirectVIV (GL_TEXTURE_2D, WIDTH, HEIGHT, GL_VIV_NV12, &
     pTexel);

/* Dequeue */
...

glBindTexture(GL_TEXTURE_2D, textures[0]);
memcpy(pTexel, buffers[buf.index].start, width * height * 3/2);
glTexDirectInvalidateVIV(GL_TEXTURE_2D);

/* Queue */
...
```

- 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:

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

Fragment shader:

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

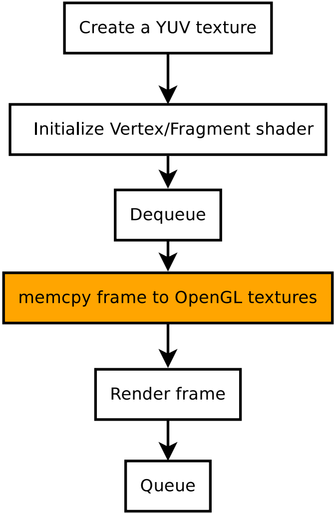
ARCHITECTURE

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

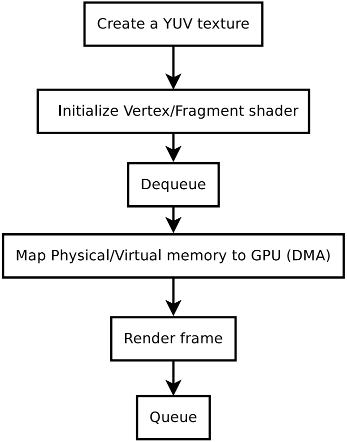
ARCHITECTURE

What we had:



ARCHITECTURE

What we are going to do:



- Remove memcpy by using DMA

RENDERING LOOP

```
/* Dequeue */
...

glBindTexture (GL_TEXTURE_2D, textures[0]);
/* Physical and Virtual addresses */
glTexDirectVIVMap(GL_TEXTURE_2D, width, height, GL_VIV_NV12, &
    buffers[buf.index].start, &(buffers[buf.index].offset));
glTexDirectInvalidateVIV(GL_TEXTURE_2D);

/* Queue */
...
```

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

ARCHITECTURE

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:

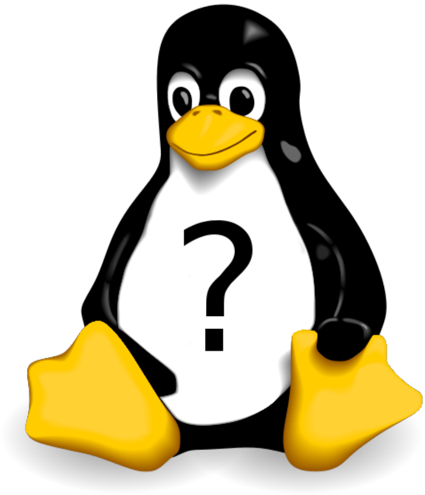
- Need some OpenGL skills and GPU API

Conclusion

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