_alloc_conherent() and image size 2048x1243

Embedded Systems Hands-On 2 Report

Author:
Tran Minh Quang
Matrikel-Nr: 2697987

TECHNISCHE UNIVERSITÄT DARMSTADT

Contents

1 Task 3: Character Device Driver

1.1 Driver implementation for Multiplier IP task 2

The general idea of using the driver is shown in the figure

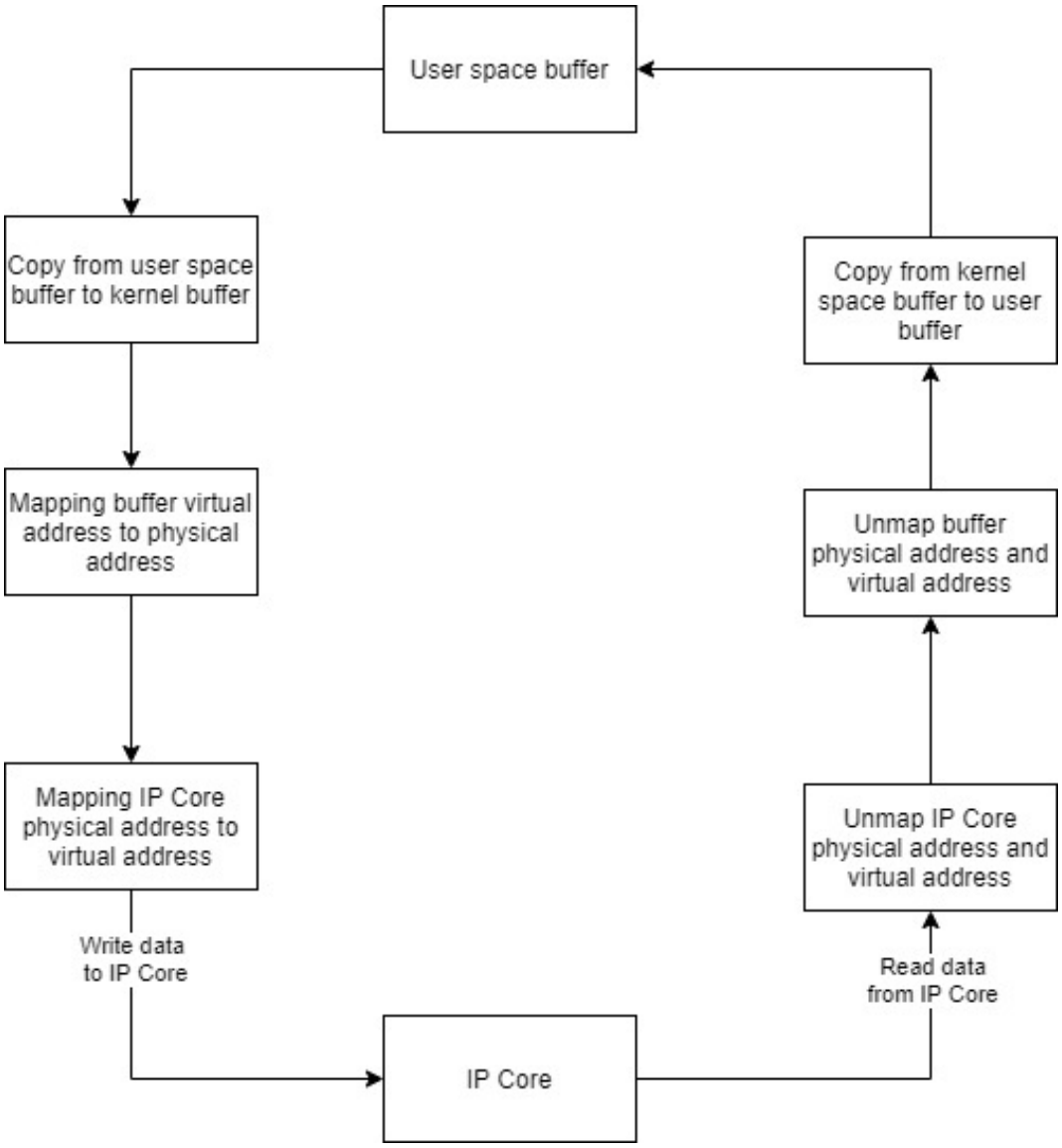

Figure 1: Character device driver usage diagram

The process in the diagram above can be done with the following important functions:

```
0   /* Open function performs the initialization of a device
    * and is used to open the driver in user space.
2   * We can ask for kernel buffer here with kmalloc()
    * or dma_alloc_coherent() function. */
4   static int device_open(struct inode *inode, struct file *file);

6   /* The write function write data from user space to kernel space.
    * In this function, copy_from_user() is used to copy data.
8   * Then ioremap function is used to map the base address of the IP core to virtual address.
    * With this physical address we can write data to IP core by the function iowrite64(). */
10  static int device_write(struct file *file, const char __user *user_buffer,size_t size, loff_t *
    offset);

12  /* The read function transfer data from kernel space to user space.
    * In this function, copy_to_user() is used to copy data.
14  * Then ioremap function is used to map the base address of the IP core to virtual address.
```

```
     * With this physical address we can read data from IP core by the function ioread64(). */
    static int device_read(struct file *file, char __user *user_buffer, size_t size, loff_t *offset);

    /* The release function releases device-specific resources.
     * In this function, kfree() or dma_free_coherent()
     * can be used to release kernel buffer.
     * iounmap() can also be used to unmap I/O memory from kernel address space. */
    static int device_release(struct inode *inode, struct file *file);
```

Listing 1: Basic functions for character device

For example, the simple implementation of **device_write()** function is as follow:

```
static ssize_t device_write(struct file*filp, const char __user*buf,size_t len, loff_t*off)
{
  mapped = ioremap(base_addr,24);
  copy_from_user(data_buf,buf,len);
  write_buf = (uint64_t*)data_buf;
  iowrite64(*(write_buf),mapped);
  iowrite64(*(write_buf+1),mapped + 8);
  return len;
}
```

Listing 2: device_write() function

1.2 User space program

At user space, we can use following functions to transfer data to the driver:

```
//Open device driver
int fd = open("/dev/my_character_device",O_RDWR);
//Transfer data with the driver
pwrite(fd, operand, 16, 0);
pread(fd, &result, 8,0);
```

Listing 3: device_write() function

As the result, the driver works perfectly for Multiplier IP core of task 2 :

add hình ở đây

2 Task 4: RGB to GrayScale

2.1 General Idea of RGB to gray conversion

2.1.1 Read and write color image with **CImg** library

To read and write the image, we can use following functions:

```
static ssize_t device_write(struct file*filp, const char __user*buf,size_t len, loff_t*off)
{
   //Read the image
   CImg<unsigned char> image("saigon.jpg");
   unsigned char*ptr = image.data(0,0);

   //Write image to kernel space
   pwrite(fd,ptr,3*image_size,0);

   //Read result image from kernel space
   pread(fd,Gray_img,width*height,0);

   // Write back the image
   CImg<unsigned char>gotback_image(Gray_img,width,height,1,1,true);

    gotback_image.save("saigon_gray.png");
}
```

Listing 4: device_write() function

The most important here is the order of the pixel red, green and blue are saved with CImag. The whole red pixel are saved in the first region, then the green and the blue one. This order is needed to read the data correctly in the IP Core.

2.1.2 RGB to Grayscale calculator

From a resource on internet [1], I get the formular to convert from RGB to gray picture as follow:

$$Grayscale = 0\ 299 * Red\_Value + 0\ 587 * Green\_Value + 0\ 114 * Blue\_Value. \tag{0.1}$$

This formula requires fixed point number. For that reason, the library **FixedPoint :: *** and **fxptMult()** functions should be used to have the correct result.

2.1.3 RGB to Grayscale acceleration

As from the equation 0.1, we realise that the computations of the red, green and blue pixels are independent. For that reason, we can compute them in parallel to gain conversion time. As the data bus of AXI4 is 64 bytes for lite, 128 bytes for full and the pixels needed for the computation are saved in different region, we cannot read each byte for each color for one computation. This is inefficient. Instead of that we can read full burst length for each color each time and store them in different FIFOs. Afterward, the pixels are taken out sequentially and computed in parallel. This idea is discribed in figure 2.

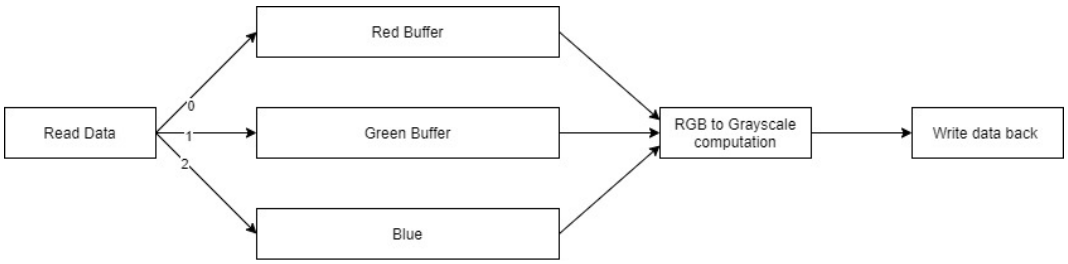

Figure 2: The general Idea of RGB to Grayscale conversion

Data is evenly distributed in turn into red, green and blue respectively. Right after three sets data of each color are available, the rule for computation is fired and write back the result to memory.

2.2 Burst length adjustment and kernel page size crossing problem

In some cases, we could have the situation where the AXI transfer have to across the end of the page before the transfer is finished. For example, the start address is not at the beginning of the 4096 bytes page, but the burst length is 256 that will transfer 4096 bytes each time as in figure 3.

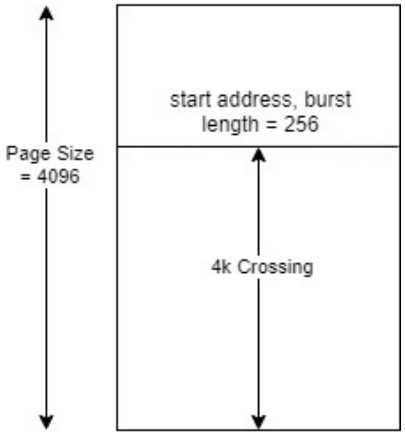

Figure 3: Page size crossing situation example

Another problem is that the burst length does not match the size of the image all the time. Very often we have to read or write extra bytes. To avoid this and the page size crossing problem, I have implemented a function to adjust the burst length for that specific situation to match exactly the page size. Afterward, the normal burst length can be used a gain as normal.

2.3 From AXI-Lite to AXI-Full

From the implementation of AXI-Lite in task 2, we can move to AXI-Full easily by the available functions in BlueAXI library.

```
//Set start address and the burst length for the write transfer.
axi4_write_addr(master_write, curr_addr, curr_burst);

//Send all the beat step by step and set the last bit signal for the last beat.
axi4_write_data(master_write, gray_pixel, 16'hffff, True);

//Get response from salve
master_write.response.get();

//Response to master
slave_write.response.put(AXI4_Lite_Write_Rs_Pkg{resp: OKAY});

//Set start address and the burst length for the read transfer.
axi4_read_data(master_read, curr_addr, curr_burst);

//Return value to master
slave_read.response.put(AXI4_Lite_Read_Rs_Pkg{ data: zExtend(pack(conversion_finished)), resp: OKAY});
```

Listing 5: AXI-Full function function

2.4 Evaluation

2.4.1 Conversion result

As the result for RGB to Grayscale converter, I represent here two different image with the size of 2048x1243 and 1280x720.

Figure 4: Original 2048x1243 picture

Figure 5: Grayscale 2048x1243 picture

Figure 6: Original 1280x720 picture

Figure 7: Grayscale 1280x720 picture

2.4.2 Comparison between two driver approaches

When I try my Grayscale converter with some picture, I always get randomly horizontal stripes as in figure 8 in the result and also core dump error, when I run the code many times.

Figure 8: Horizontal stripes in the result

After long time debug, I realize that the reason for this is because the kernel does not provide enough memory for the image and the IP overwrites the other memory areas. I tried to implement two different kernel approaches and find out that the driver has the impact on the performance as well.

The two approaches are:

- **dma_map_single()**: To use this mapping function we have to ask for kernel buffer by kmalloc() function. With this we can only get an memory around 4MB and that is not enough for run a full HD picture. However, this function has the performance advantage. We can read and write faster with this method.
- **dma_alloc_coherent()**: This function return a kernel buffer and also its virtual address. We can get much larger buffer with this method. However, the read and writing time is slower.

As in figure 10, we can see that dma_alloc_conherent() always take more time for the same bitstream and image size. This difference will become more pronounced the larger the image size. I have a better example when we get to the sobel filter section.

Image size = 1280x720	AXI-Lite	AXI-Full 4	AXI-Full 16	AXI-Full 256
dma_alloc_coherent()	0.0623994s	0.0522661s	0.0522465s	0.0522369s
dma_map_single()	0.0517502s	0.0424728s	0.041758s	0.0414705s

Figure 9: Driver approaches comparision

2.4.3 Comparison between AXI-Lite and AXI-Full with different burst length

In this section, I represent the result when I run different AXI4 configurations with different image sizes.

Image size = 1280x720	AXI-Lite	AXI-Full 4	AXI-Full 16	AXI-Full 256
Conversion time (s)	0,0517502	0,0424728	0,041758	0,0414705
Bandwidth (MB/s)	17,8	21,7	22,07	22,2
FPS	19	23,5	23,9	24

Figure 10: AXI4 with dma_map_single() and image size 1280x720

Image size = 1280x720	AXI-Lite	AXI-Full 4	AXI-Full 16	AXI-Full 256
Conversion time (s)	0,0623994	0,0522661	0,0522465	0,0522369
Bandwidth (MB/s)	14,8	17,63	17,64	17,64
FPS	16	19	19	19

Figure 11: AXI4 with dma_alloc_conherent() and image size 1280x720

Image size = 2048x1243	AXI-Lite	AXI-Full 4	AXI-Full 16	AXI-Full 256
Conversion time (s)	0,114558	0,0841386	0,0830493	0,0828176
Bandwidth (MB/s)	22,2	30,3	30,7	30,7
FPS	8,72	11,9	12	12

Figure 12: AXI4 with dma_alloc_conherent() and image size 2048x1243

From the result in the above table, we can see that there There is a significant improvement when switching from AXI-Lite to AXI-Full. However, there are no significant differences between AXI-Full 4, 16 and 256. In addition, with dma_alloc_coherent() we can run the conversion with larger picture, but it reduces the performance.

2.4.4 Resource utilization

3 Task 5: Sobel Filter

3.1 General Idea of Sobel Filter implementation

3.2 Evaluation

3.2.1 Comparison between two driver approaches

3.2.2 Resource utilization

3.2.3 Comparison between different kernel size

References

[1] Rgb to grayscale formular. https://www.dynamsoft.com/blog/insights/image-processing/image-processing-101-color-space-conversion/.