Laboratory Exercise 10

Using Hardware Acceleration

CPUs are general-purpose computational devices that are capable of a wide variety of tasks. This level of flexibility however, comes at a cost; CPUs are slower at some tasks than specialized devices. Because of this factor, some applications can execute more quickly if parts of the implementation run on a specialized device, such as a custom circuit in a field-programmable gate array (FPGA). This approach, of offloading computation to a faster device, is known as *hardware acceleration*.

In this laboratory exercise, we will implement two versions of an image processing application that detects the edges of an image. The first version will run entirely on the CPU. We will assume that the reader is using the DE1-SoC board to run their programs. If a different board is being used, the same instructions will mostly apply, but we will point out any differences as needed. The second version of the application will make use of hardware acceleration, by offloading most of the image processing operations to a hardware accelerator implemented inside an FPGA. We will assume that image files are represented in the *bitmap* (BMP) file format, using *24-bit true color*.

The Canny Edge-detection Technique

In this exercise, we will implement a variation of the *Canny edge detector*, which is a widely-used edge-detection scheme. Figures 1 and 2 show a sample image that is provided as the input to a Canny edge detector, as well as the resulting edge-detected output image.

The Canny edge-detection algorithm involves five stages which are applied to the input image in succession. The details of these stages are given below. As well, we will see how the sample input image from Figure 1 is transformed as it passes through each stage.

Figure 1: Original image.

Figure 2: Edge-detected image.

Stage 1: Grayscale Conversion

Figure 3 shows the state of our sample image at the end of the grayscale conversion stage. This stage converts the input 24-bit bitmap color image (8 bits each for red, green, and blue) into an 8-bit grayscale image. The grayscale value at each pixel is calculated as the average of the three 8-bit color values of the original image.

Figure 3: The sample input image after the grayscale conversion stage.

Stage 2: Gaussian Smoothing

Figure 4 shows the state of our sample image at the end of the Gaussian smoothing stage. In this stage, a Gaussian filter is used to smooth out the image, by modifying noisy pixels (pixels that are unlike their neighbouring pixels) to be more like their neighbours. Shown below is the 5 x 5 Gaussian filter operation that is applied to the image. Note that the * denotes convolution, A is the original image, and B is the resulting filtered image. The effect of this operation is that each pixel gets assigned the weighted average value of the 5 x 5 grid of pixels surrounding each pixel.

$$B = 1/159 \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * A$$

Figure 4: The sample input image after the Gaussian smoothing stage.

Stage 3: Sobel Operator

Figure 5 shows the state of our sample image at the end of the Sobel operator stage. This stage overwrites each pixel with the overall intensity gradient at that pixel. To calculate the overall intensity gradient, the gradient is first calculated in the horizontal (C_x) and vertical (C_y) directions across the pixel, using the matrices below:

$$C_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * B \qquad C_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * B$$

The magnitudes of the two gradients are then added to calculate the overall gradient intensity value for each pixel, resulting in the image C:

$$C = 0.5|C_x| + 0.5|C_y|$$

In image C, which is the final output of this stage, the edges of the original image are highlighted as brighter pixels. Non-edges, which are areas with low intensity gradients, appear as darker pixels.

Figure 5: The sample input image after the Sobel operator stage.

To illustrate the effect of the Sobel operator, let us examine different image boundaries that may exist in an image as shown in Figure 6. For each 3 x 3 image shown, we can use the Sobel operator to calculate the intensity gradient at the center pixel. Let us examine the vertical boundary example, where there is a boundary between darker pixels on the left side of the image, and brighter pixels on the right side. In this example, $|C_x|$ of the center pixel can be calculated as |{-1*2 + 0*98 + 1*181} + {-2*1 + 0*94 + 2*178} + {-1*6 + 0*91 + 1*184}| = 711. In the vertical direction, $|C_y|$ for this pixel is calculated to be 7. The intensity gradient for this pixel is high in the horizontal direction, and low in the vertical direction, which is to be expected along a vertical boundary. The total intensity gradient for this pixel, is $0.5 * 711 + 0.5 * 7 = 359$ which saturates the 8-bit grayscale channel to value 255. The Sobel operator would therefore detect this boundary as a strong edge, and the center pixel would become bright with a value of 255.

Vertical Boundary		
2	98	181
1	94	178
6	91	184

Horizontal Boundary		
188	185	181
86	94	91
4	3	3

Diagonal Boundary		
100	185	181
5	94	178
2	3	91

Non-Boundary		
137	145	129
133	140	138
129	130	131

Figure 6: Examples of boundaries in an image.

In horizontal boundary case, we see that the vertical gradient is high ($|C_y|$ = 726) and the horizontal gradient is low ($|C_x|$ = 2). For the diagonal boundary case, the gradient is high in both directions with $|C_x|$ = 516 and $|C_y|$ = 552. Finally, for the non-boundary case, the gradients are low in both directions with $|C_x|$ = 4 and $|C_y|$ = 36. As the gradients are low, the center pixel would become dark with a value of $0.5 * 4 + 0.5 * 36 = 20$.

Stage 4: Non-Maximum Suppression

Figure 7 shows the state of our sample image at the end of the non-maximum suppression stage. This stage aims to thin the thick and/or blurry edges that may have resulted from the Sobel operator stage. Thick edges are problematic as many applications of edge detection benefit from the edges being as thin as possible. For example, to accurately calculate the surface area of an object, thin edges are desired as to not overlap with the surface. The non-maximum suppression stage thins the edges by removing the weaker (non-maximum) pixels of each edge, and keeping only the maxima. Figure 8 shows the effect of non-maximum suppression on a sample image containing a blurry vertical line. Notice that the vertical line, which is originally three-pixels wide, becomes one-pixel wide.

Figure 7: The sample input image after the non-maximum suppression stage.

0	41	134	45	0
0	43	135	46	0
0	35	136	41	0
0	41	132	35	0
0	44	131	41	0

Non-maximum Suppression →

0	0	134	0	0
0	0	135	0	0
0	0	136	0	0
0	0	132	0	0
0	0	131	0	0

Figure 8: The effect of nonmaximum suppression on a blurred vertical line.

Stage 5: Hysteresis

Figure 9 shows the end result of the hysteresis stage. The goal of the hysteresis stage is to remove any pixel that does not belong to an edge. Additionally, weak edges are erased altogether by removing any pixel that does not meet a user-defined intensity threshold. The hysteresis algorithm examines each pixel to determine whether:

1. the pixel exceeds the user-defined threshold value, and
2. there exists at least one adjacent pixel (horizontally, vertically, or diagonally) that exceeds the intensity threshold.

If both criteria are met, the pixel is preserved. Otherwise, the pixel is removed by turning it black.

Figure 9: The sample input image after the hysteresis stage.

Part I

Write a C-language program that implements the five stages of the canny edge detector as described above, and run your code on the DE1-SoC Computer. Some sample images are provided along with this exercise. Start with the provided skeleton code in Figure 10. It contains functionality for loading and storing bitmap images. The *main* function for the program is given in Part d of the figure. Once a 24-bit true-color image is loaded into memory, the code calls functions to transform the pixels according to the five edge-detection stages, and then writes the resulting image into an output file. The first step of edge-detection, in which the image is converted to grayscale, is provided in the code, but you have to write the other stages. The main program also uses a function, *draw_image*, to render the image on a video display. This function uses the character-device driver called *video*, which can be found in your Linux* file-system in the folder `/home/root/Linux_Libraries/drivers`. This driver is described in the tutorial *Using Linux on DE-series Boards*. The code in Figure 10 also shows how to measure the runtime of your program, which will allow you to compare with the hardware-accelerated version.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <intelfpgaup/video.h>

typedef unsigned char byte;
struct pixel { // Ordering of pixel colors in BMP files is b, g, r
   byte b;
   byte g;
   byte r;
};

// Read BMP file and extract the header (store in header) and pixel values
// (store in data)
int read_bmp(char *bmp, byte **header, struct pixel **data, int *wid, int *ht) {
   struct pixel *data_;    // temporary pointer to pixel data
   byte *header_;          // temporary pointer to header data
   int width_, height_;    // temporary variables for width and height

   FILE *file = fopen (bmp, "rb");
   if (!file) return -1;
   // read the 54-byte header
   header_ = malloc (54);
   fread (header_, sizeof(byte), 54, file);

   // get height and width of image
   width_ = *(int*) &header_[18];   // width given by 4 bytes at offset 18
   height_ = *(int*) &header_[22];  // height given by 4 bytes at offset 22
   // Read in the image
   int size = width_ * height_;
   data_= malloc (size * sizeof(struct pixel));
   fread (data_, sizeof(struct pixel), size, file); // read the data
   fclose (file);

   *header = header_;   // return pointer to caller
   *data = data_;       // return pointer to caller
   *wid = width_;       // return value to caller
   *ht = height_;       // return value to caller
   return 0;
}
```

Figure 10: An outline of the software for edge detection (Part a).

```
// Determine the 8-bit grayscale value by averaging the r, g, and b fields.
// Store the 8 bit grayscale value in the r field.
void convert_to_grayscale(struct pixel *data, int width, int height) {
   int x, y;

   for (y = 0; y < height; y++) {
      for (x = 0; x < width; x++) {
         // For each pixel store the grayscale value in its r field
         (*(data + y*width + x)).r = ((*(data + y*width + x)).r + (*(data +
             y*width + x)).g +
             (*(data + y*width + x)).b)/3;
      }
   }
}

// Write the grayscale image to disk. The 8-bit grayscale values should be
// inside the r field of each pixel.
void write_grayscale_bmp(char *bmp, byte *header, struct pixel *data, int width,
    int height) {
   FILE* file = fopen (bmp, "wb");

   // write the 54-byte header
   fwrite (header, sizeof(byte), 54, file);
   int y, x;

   // the r field of the pixel has the grayscale value. Copy to g and b.
   for (y = 0; y < height; y++) {
      for (x = 0; x < width; x++) {
         (*(data + y*width + x)).b = (*(data + y*width + x)).r;
         (*(data + y*width + x)).g = (*(data + y*width + x)).r;
      }
   }
   int size = width * height;
   fwrite (data, sizeof(struct pixel), size, file); // write the rest of the data
   fclose (file);
}

// Draw the image pixels on the video display
void draw_image (struct pixel *data, int width, int height, int screen_x, int
    screen_y) {
   // ... code not shown
}
```

Figure 10. An outline of the software for edge detection (Part *b*).

```
// Gaussian blur. Operate on the .r fields of the pixels only.
void gaussian_blur( ) {
   unsigned int gaussian_filter[5][5] = {
      { 2, 4, 5, 4, 2 },
      { 4, 9,12, 9, 4 },
      { 5,12,15,12, 5 },
      { 4, 9,12, 9, 4 },
      { 2, 4, 5, 4, 2 }
   };
   // ... code not shown
}

void sobel_filter( ) {
   // Definition of Sobel filter in horizontal and vertical directions
   int horizontal_operator[3][3] = {
      { -1,  0,  1 },
      { -2,  0,  2 },
      { -1,  0,  1 }
   };
   int vertical_operator[3][3] = {
      { -1,  -2,  -1 },
      { 0,  0,  0 },
      { 1,  2,  1 }
   };
   // ... code not shown
}

void non_maximum_suppressor( ) {
   // ... code not shown
}

// Only keep pixels that are next to at least one strong pixel.
void hysteresis_filter( ) {
   #define strong_pixel_threshold 32   // example value
   // ... code not shown
}
```

Figure 10. An outline of the software for edge detection (Part *c*).

```
int main(int argc, char *argv[]) {
   struct pixel *data;                          // used to hold the image pixels
   byte *header;                                // used to hold the image header
   int width, height;                           // the dimensions of the image
   int screen_x, screen_y, char_x, char_y;      // video screen dimensions
   time_t start, end;                           // used to measure run-time

   // Check inputs
   if (argc < 2) {
      printf ("Usage: edgedetect <BMP filename>\n");
      return 0;
   }

   // Open input image file (24-bit bitmap image)
   if (read_bmp (argv[1], &header, &data, &width, &height) < 0) {
      printf ("Failed to read BMP\n");
      return 0;
   }

   if (!video_open ( )) // open the video display driver
   {
      printf ("Error: could not open video device\n");
      return -1;
   }
   video_read (&screen_x, &screen_y, &char_x, &char_y);   // video screen size
   draw_image (data, width, height, screen_x, screen_y);

   /*****************************************
   *          IMAGE PROCESSING STAGES          *
   *****************************************/

   // Start measuring time
   start = clock();

   convert_to_grayscale(data, width, height);
   // gaussian_blur (...);
   // sobel_filter (...);
   // non_maximum_suppressor (...);
   // hysteresis_filter (...);

   end = clock();

   printf ("TIME ELAPSED: %.0f ms\n", ((double) (end - start)) * 1000 /
       CLOCKS_PER_SEC);

   write_grayscale_bmp ("edges.bmp", header, data, width, height);

   printf ("Press return to continue");
   getchar ();
   draw_image (data, width, height, screen_x, screen_y);
   video_close ( );
   return 0;
}
```

Figure 10. An outline of the software for edge detection (Part *d*).

Part II

For Part I you used the DE1-SoC Computer system to run your program. For this part, you will use a different computer system, which is illustrated in Figure 11. This system include an edge-detection mechanism that is implemented as a hardware circuit in the FPGA device. As indicated in the figure, the hardware edge-detection mechanism consists of five components that are connected in sequence. The first component is called *Mem-to-Stream DMA*. This component is a *direct memory access* (DMA) controller that reads a 24-bit color image from memory. As the DMA reads these pixels, it *streams* (sends in sequence) them to the *RGB24 to Grayscale* color-space converter. The converter implements stage one of the canny edge detector, converting the 24-bit color pixels into 8-bit grayscale pixels. The grayscale pixels are then streamed to the *Edge Detector* component, which implements stages two to five. The *Grayscale-to-RGB24* color-space converter then transforms the grayscale image back into the RGB24 format, which the *Stream-to-Mem DMA* then writes back into memory.

Figure 11 shows two types of memory ports in the FPGA: an SDRAM port, and an Onchip-memory port. The physical address ranges of these memories are `0xC0000000` to `0xC3FFFFFF` for SDRAM and `0xC8000000` to `0xC803FFFF` for Onchip memory. If you are using the DE1-SoC or DE10-Standard boards, the edge-detection mechanism and video-out port are connected to the SDRAM port. If you are using the DE10-Nano board, then it does not include an SDRAM memory and the edge-detection mechanism, and video-out port, are connected to the Onchip memory. When the SDRAM memory is being used, the edge-detection mechanism assumes that the image size is 480×480 pixels, and the image is stored in the memory starting at address `0xC0000000`. The edge-detected output image is saved to the address `0xC2000000`. When the Onchip memory is being used, the image size is 240×240 pixels, the image starts at address `0xC8000000`, and the edge-detected image is saved at `0xC8040000`.

The DMA controllers shown in Figure 11, as well as the video-out port, require that each pixel in memory is *word-aligned*. Every 32 bits in memory should store one pixel, where bits 31 to 24 are (unused) padding bits, 23 to 16 are the red component, 15 to 8 are green, and 7 to 0 are blue.

When the SDRAM is being used the video-out port is initialized to read pixel data starting at the address `0xC0000000`. When Onchip memory is being used the video-out port is set up to read pixel data from the address `0xC8000000`.

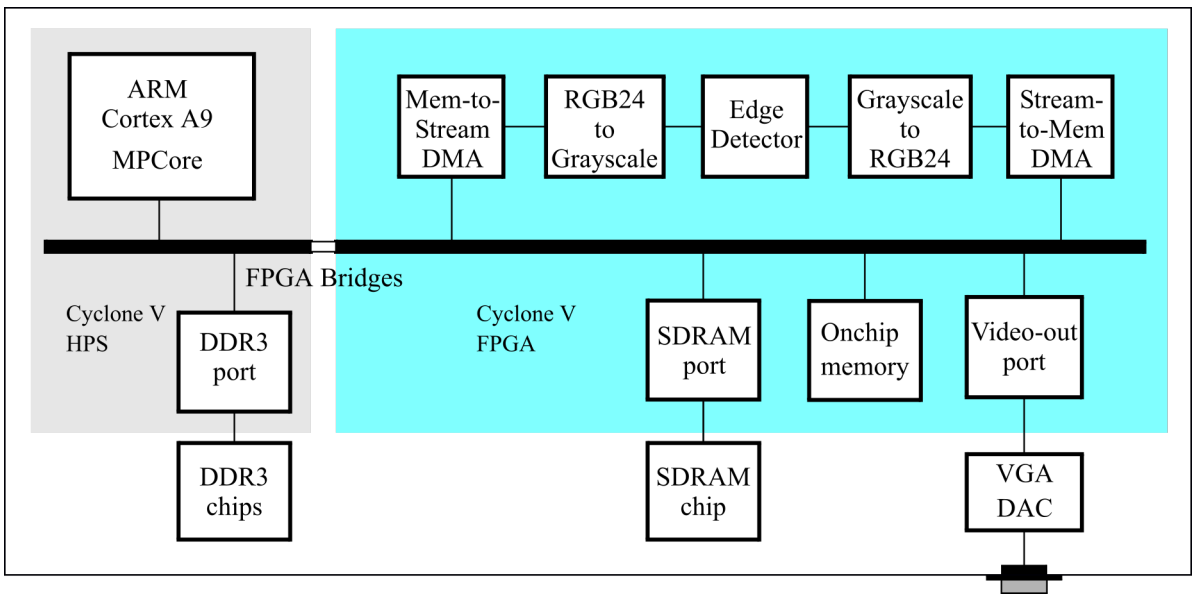

Figure 11: The components of the edge-detection system.

Perform the following:

1. Program the FPGA with the *Edge_Detector_System.rbf* file, following the instructions provided in the appendix of the tutorial *Using Linux on DE-series Boards*. For this part you will make use of only the video-out capability of the system; the edge-detection capability will be used in Part III.

2. Consider the code in Figure 12. It first calls *read_bmp*, shown in Figure 10a, to read a bitmap image from a file into a data structure. The code then generates virtual addresses that allows access to SDRAM memory at physical address `0xC0000000`, and the FPGA lightweight bridge at address `0xFF200000`. The functions *open_physical* and *map_physical* use the */dev/mem* mechanism and the Linux system function *mmap* to provide virtual memory mappings of physical addresses. More details are provided in the tutorial *Using Linux on DE-series Boards*.

3. You are to write the function *memcpy_consecutive_to_padded* to complete the code in Figure 12. This function has to write a 32-bit value into the SDRAM buffer corresponding to each 24-bit pixel in the image.

4. Compile and test your code. Some sample images that have the correct dimensions are provided with this lab exercise (480×480 pixels for the DE1-SoC and DE10-Standard boards, and 240×240 for the DE10-Nano). You should see the bitmap image, upside down, on the video display.

5. Write a function called *flip* that flips the image vertically before writing it into the SDRAM buffer. Test your code to see that the image is now displayed right-side up.

Part III

For this part you are to extend your program from Part II so that it makes use of the hardware edge-detection mechanism. You will need to access the programming registers of the DMA controllers, which are illustrated in Figure 13. The register at the `Base` address is called the *Buffer* register, and the one at address `Base+4` is the *Backbuffer* register. Each of these registers stores the address of a memory buffer. The `Buffer` register stores the address of the buffer that is *currently* being used by the DMA controller.

The operation of the DMA controller can be turned *off* by writing the value `0` into the *Status* register, at address `Base+12`. Writing the value `0x4` turns the DMA controller *on*, so that it performs transfers from/to memory starting at the address in the *Buffer* register.

It is possible for software to directly write into the *Backbuffer* register to change its contents, but not the *Buffer* register. To change the *Buffer* register it is necessary to perform a *swap* operation, explained below.

A buffer register swap is caused by writing the value `1` to the *Buffer* register. This write operation does not directly modify the content of the *Buffer* register, but instead causes the contents of the *Buffer* and *Backbuffer* registers to be swapped. The swap operation does not happen right away; it occurs at the end of the current DMA operation, when all pixels have been transferred. Software can poll the value of the S bit in the *Status* register to see when a DMA operation has been completed (an entire image has been processed). Writing the value 1 into the *Buffer* register causes S to be set to 1. Then, when the DMA operation is completed, S is reset back to 0.

The addresses of the programming registers for the three DMA controllers shown in Figure 11 are given in Table 1. The last two columns in the table give the initial contents of the *Buffer* and *Backbuffer* registers for each DMA. The columns labeled SDRAM and Onchip show the contents when the SDRAM memory or Onchip memory are being used, respectively. As shown, the *Steam-to-Mem* and *Mem-to-Stream* DMAs have the same value in both the *Buffer* and *Backbuffer* registers. It is still possible to initiate a swap to determine when a DMA operation has been completed. For the video-port DMA controller, the *Buffer* address is set to the start of memory, and the *Backbuffer* address points to the start of the edge-detected image. Performing a swap operation allows either the original image or the generated edge-detected image to be displayed.

```
int main(int argc, char *argv[]){
   struct pixel *data;      // used to hold the image pixels
   byte *header;            // used to hold the image header
   int width, height;       // image size
   int fd = -1;             // used to open/read/etc. the image file
   void *SDRAM_virtual;
   void *LW_virtual;

   // Pointer to the DMA controller for the original image
   volatile unsigned int *mem_to_stream_dma = NULL;

   // Check inputs
   if (argc < 2){
      printf ("Usage: edgedetect <BMP filename>\n");
      return 0;
   }
   // Open input image file (24-bit bitmap image)
   if (read_bmp (argv[1], &header, &data, &width, &height) < 0){
      printf ("Failed to read BMP\n");
      return 0;
   }
   printf ("Image width = %d pixels, Image height = %d pixels\n", width, height);

   if ((fd = open_physical (fd)) == -1)   // Open /dev/mem
      return (-1);
   SDRAM_virtual = map_physical (fd, 0xC0000000, 0x03FFFFFF);
   LW_virtual = map_physical (fd, 0xFF200000, 0x00005000);
   if ((LW_virtual == NULL)  (SDRAM_virtual == NULL))
      return (0);

   // Set up pointer to edge-detection DMA controller
   mem_to_stream_dma = (volatile unsigned int *)(LW_virtual + 0x3100);
   *(mem_to_stream_dma+3) = 0; // Turn off edge-detection hardware DMA

   // Write the image to the memory used for video-out and edge-detection
   memcpy_consecutive_to_padded (data, SDRAM_virtual, width*height);

   free(header);
   free(data);

   unmap_physical (SDRAM_virtual, 0x03FFFFFF);  // release mem mapping
   unmap_physical (LW_virtual, 0x00005000);     // release mem mapping
   close_physical (fd);                         // close /dev/mem

   return 0;
}
```

Figure 12: The software code for Part II.

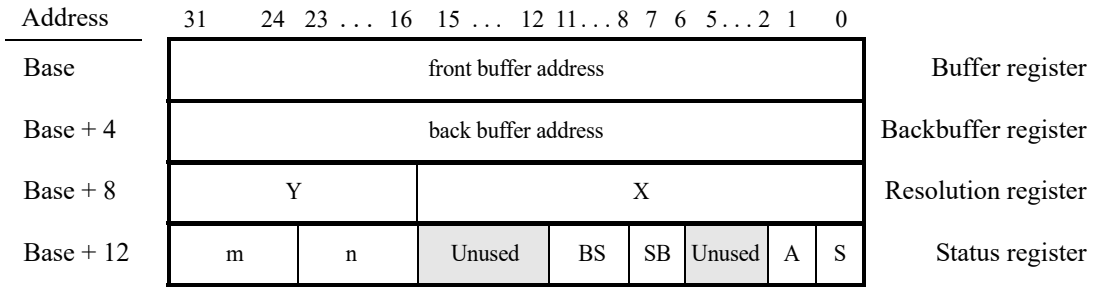

Figure 13: DMA controller registers.

DMA	Address	Register	SDRAM	Onchip
Video-out	0xFF203020	Buffer	0xC0000000	0xC8000000
	0xFF203024	Backbuffer	0xC2000000	0xC8040000
	0xFF203028	Resolution		
	0xFF20302C	Status		
Mem-to-Stream	0xFF203100	Buffer	0xC0000000	0xC8000000
	0xFF203104	Backbuffer	0xC0000000	0xC8000000
	0xFF203108	Resolution		
	0xFF20310C	Status		
Stream-to-Mem	0xFF203120	Buffer	0xC2000000	0xC8040000
	0xFF203124	Backbuffer	0xC2000000	0xC8040000
	0xFF203128	Resolution		
	0xFF20312C	Status		

Table 1: DMA register addresses.

Your program should do the following:

1. Disable the *Mem-to-Stream* and *Stream-to-Mem* DMA controllers. Recall that you will first need to obtain virtual addresses for accessing the physical addresses in Table 1. Refer to the tutorial *Using Linux on DE-series Boards* if needed.
2. Copy the pixels of the input image to memory at address `0xC0000000` (or `0xC8000000`). The bitmap image should now appear on the video display.
3. Enable the DMAs to start the edge-detection operation. Then, perform a swap operation and wait until both DMA controllers are finished.
4. Disable the DMAs.
5. Perform a swap operation for the video DMA, so that the edge-detected image appears on the display.
6. Save the edge-detected image into a file *edges.bmp*, for later display.
7. Test your program by using the images provided with this lab exercise. For the DE1-SoC and DE10-Standard boards 480×480 images are provided, and for the DE10-Nano 240×240 images are included. Compare the run-time of your program with the software-only solution from Part I, to see the benefits of using hardware acceleration.