Hardware Accelerator for Real-Time Edge Detection and Movement Detection

Final project report

Embedded Systems Design - CS476

Authors

Francesco Poluzzi

Francesco Murande Escobar

September 27, 2024

Spring Semester

Abstract

Image processing has become a critical task in real-time applications. Typically, this task is executed by embedded systems, which are specialized electronic systems designed for specific functions rather than for general-purpose use.

Among the key image processing tasks are edge detection and motion detection. Edge detection identifies objects and can discern physical elements, while motion detection is essential for recognizing movement. Algorithms performing these tasks are often combined, as motion detection becomes more effective when it focuses on edges. This combination is fundamental in various practical applications such as surveillance systems, traffic monitoring, and autonomous driving.

A commonly used edge detection algorithm is the Sobel algorithm, which is computationally intensive. Executing this algorithm solely at the software level on an embedded system's processor can be slow and inefficient. However, embedded systems offer the possibility to customize circuits at the hardware level, which opens up opportunities for performance acceleration.

The aim of this report is to identify and address bottlenecks in the software implementation of the Sobel algorithm and motion detection, and to enhance performance using hardware accelerators. The project concentrates on boosting the efficiency of edge and motion detection processes through the use of FPGA-based hardware accelerators.

The report provides detailed explanations of both software and hardware implementations. It includes a thorough performance analysis of the software implementation and the rationale behind the chosen acceleration strategy.

The project has achieved significant speed improvements, ultimately fulfilling the goal of performing Sobel and motion detection in real-time.

Contents

1 Introduction **3**
1.1 Experiment Setup . . . 3
1.2 Sobel Algorithm . . . 3
1.3 Movement detection . . . 4

2 Software Implementation **4**
2.1 Grayscale . . . 4
2.2 Sobel Filtering . . . 6
2.3 Movement detection . . . 9
2.4 Observations: Cycle estimation . . . 12
2.5 Profiling . . . 12

3 Hardware Acceleration **12**
3.1 Near-memory computing . . . 13
3.2 Grayscale acceleration . . . 14
3.3 Sobel acceleration . . . 14
3.4 Movement detection acceleration . . . 17
3.5 Full program . . . 18

4 Results and discussions **19**
4.1 Step-by-step profiling . . . 19
4.2 Final results . . . 21

5 Conclusions **22**
5.1 Potential future improvements . . . 22

6 Appendix **23**

1 Introduction

This project was performed for the university course CS476 - Embedded Systems Design at École Polytechnique Fédérale de Lausanne (EPFL). The goal of the project was to accelerate a full-software edge detection algorithm followed by movement detection on an embedded-system board that was provided to us. This report will detail a thorough study and performance tracking of the software implementation, our reasoning for accelerating this program, and an explanation of the final accelerated version. The project involved both software development in C and hardware description in Verilog.

1.1 Experiment Setup

For the project, the Gecko4Education EPFL platform was used [1], which is based on the Altera Cyclone IV FPGA [2]. On this platform, an OR1200 processor simulator was implemented [3]. The OR1200 is a 32-bit scalar RISC processor with a Harvard microarchitecture, featuring a 5-stage integer pipeline, virtual memory support (MMU), and basic DSP capabilities. This simulator is capable of emulating OpenRISC-based computer systems at the instruction level, making it suitable for developing and testing the hardware acceleration of algorithms.

The Olimex OV7670 camera was utilized [4], providing a maximum resolution of 640x480 pixels at 30 frames per second (FPS). It is driven by a 24MHz clock oscillator. For the purposes of this project, the camera was operated in grayscale mode with a resolution of 640x480 (VGA), maintaining a frame rate of 15 FPS. This setup allows for sufficient image quality and frame rate for effective edge and motion detection.

Cutecom was used as the RS232 communication program.

1.2 Sobel Algorithm

The Sobel Algorithm is an edge detection method developed by Irwin Sobel and Gary Feldman at the Stanford Artificial Intelligence Laboratory (SAIL) in 1968. The Sobel operator performs a 2-D spatial gradient measurement on an image, emphasizing regions of high spatial frequency corresponding to edges. [5]

The algorithm works by convolving a 3x3 matrix (kernel) with the image pixels. At each iteration, the algorithm measures the change in the gradient of the pixels within the 3x3 kernel. A significant change in pixel intensity indicates the presence of an edge.

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A}$$

$$\mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

Figure 1: Horizontal and vertical gradient computation

- **A** indicates the 3x3 window of the frame, the central element of the **A** matrix is the one currently processed.

The Sobel operator uses two kernels, one for detecting changes in the horizontal direction (Gx) and another for the vertical direction (Gy). The magnitude of the gradient is then calculated using the formula:

$$\text{Gradient Magnitude} = \sqrt{(Gx)^2 + (Gy)^2}$$

This method highlights edges in the image, which means regions with a significant change in intensity. Setting a threshold we can determine which values of Gradient Magnitude are detected as edges, changing the sensitivity of the algorithm.

For simplicity, the Gradient Magnitude can be approximated using the absolute values of the gradients:

$$\text{Approximate Gradient Magnitude} = |Gx| + |Gy|$$

1.3 Movement detection

Motion Detection is the process of detecting moving objects within a video sequence. In Computer Vision, it involves detecting pixel-wise changes across consecutive video frames. The goal is to identify areas of the video where movement has occurred.

For this project, the desired output was to detect movement without losing the information of the frame, meaning that static objects should still be visible. Traditional motion detection algorithms such as Frame Differencing or Background Subtraction were not suitable for this purpose. Instead, a simpler approach was used, where pixels are assigned different colours based on their characteristics:

- **Gray:** Background pixels that do not change over time.
- **Black:** Steady edge pixels that form static edges in the image.
- **White:** Moving edge pixels that indicate the presence of motion.

This approach allows for the continuous visualization of both moving and static objects, giving a complete view of the scene while highlighting motion.

2 Software Implementation

A full software implementation for performing Sobel and movement detection was developed. The goal of this was both to understand the algorithms and to identify the software elements that significantly slow down the operation. A complete profiling of all steps of this implementation was performed, with the goal of identifying the parts that slow the process the most.

2.1 Grayscale

As the image produced by the camera is in RGB format, and the Sobel algorithm is performed on a grayscale image, a conversion is needed. We use the conventional transformation for this

purpose, which means a weighted sum of the Red, Green and Blue values to estimate the luminance.

2.1.1 Code

```
for (int line = 0; line < camParams.nrOfLinesPerImage; line++) {
     for (int pixel = 0; pixel < camParams.nrOfPixelsPerLine; pixel++) {
       uint16_t rgb =
          swap_u16(rgb565[line*camParams.nrOfPixelsPerLine+pixel]);
       uint32_t red1 = ((rgb >> 11) & 0x1F) << 3;
       uint32_t green1 = ((rgb >> 5) & 0x3F) << 2;
       uint32_t blue1 = (rgb & 0x1F) << 3;
       uint32_t gray = ((red1*54+green1*183+blue1*19) >> 8)&0xFF;
       grayscale[line*camParams.nrOfPixelsPerLine+pixel] = gray;
     }
   }
```

Code Block 1: Main loop of the grayscale conversion function

The program performs a weighted mean of the RGB value and stores it on an 8-bit unsigned integer.

2.1.2 Number of cycles estimation

To estimate the number of cycles we analyze the function step by step:

Loop Initialization and Increment:

- `for (int line = 0; line < camParams.nrOfLinesPerImage; line++)`: 1 comparison and 1 increment per iteration.
- `for (int pixel = 0; pixel < camParams.nrOfPixelsPerLine; pixel++)`: 1 comparison and 1 increment per iteration.

Index Calculation:

- `line * camParams.nrOfPixelsPerLine + pixel`: 1 multiplication and 1 addition

Memory Access:

- `rgb565[line * camParams.nrOfPixelsPerLine + pixel]`: 1 Access
- `swap_u16`: 1 function call and 1 operation
- `grayscale[line * camParams.nrOfPixelsPerLine + pixel] = gray`: 1 Access

Bitwise Operations and Shifts:

- `((rgb » 11) & 0x1F) « 3`: 1 shift, 1 mask, 1 shift.
- `((rgb » 5) & 0x3F) « 2`: 1 shift, 1 mask, 1 shift.
- `(rgb & 0x1F) « 3`: 1 mask, 1 shift.

Grayscale Calculation:

- `red1 * 54`: 1 operation

- `green1 * 183`: 1 operation
- `blue1 * 19`: 1 operation
- `red1 * 54 + green1 * 183 + blue1 * 19`: 2 additions
- `(red1 * 54 + green1 * 183 + blue1 * 19) >> 8`: 1 operation
- `& 0xFF`: 1 operation

Total Cycles per Inner Loop Iteration

It is evident how one of the most demanding tasks in this loop is the arithmetic and logic operations that are performed per pixel of the image. Additionally, we are also storing the calculated value at each iteration.

Total Cycles for the Entire Loop

Assuming the image size is `width` x `height`:

- Number of inner loop iterations: `width` x `height`
- Outer loop increment and comparison.

$$\begin{aligned}\text{Total cycles} = (\text{width} \times \text{height}) \times \text{Arithmetic or Logic Cycles} \\ + \text{height} \times (\text{Outer loop increment and comparison})\end{aligned}$$

2.1.3 Bottlenecks Hot-Spotting

Every part of the function could be considered a bottleneck: accessing the values, performing multiplications, additions, and shifting operations. Therefore, to optimize the overall functioning of our hardware accelerator, we will process the RGB values directly in the camera modules through hardwired shifting operations. Additionally, we will take advantage of bus parallelism (32-bit) by sending four grayscale pixels per word instead of two RGB pixels per word.

2.2 Sobel Filtering

The next function we will analyze is the one that takes a grayscale image as input, performs Sobel detection, and outputs a black-and-white image where all the edges are highlighted in white and the rest of the pixels are black.

2.2.1 Code

```
void edgeDetection( volatile uint8_t *grayscale,
                    volatile uint8_t *sobelResult,
                    int32_t width,
                    int32_t height,
                    int32_t threshold ) {
  const int32_t gx_array[3][3] = {{-1,0,1},
                                  {-2,0,2},
                                  {-1,0,1}};
  const int32_t gy_array[3][3] = { {1, 2, 1},
                                   {0, 0, 0},
```

```
                                    {-1,-2,-1}};
  int32_t valueX,valueY, result;
  for (int line = 1; line < height - 1; line++) {
    for (int pixel = 1; pixel < width - 1; pixel++) {
      valueX = valueY = 0;
      for (int dx = -1; dx < 2; dx++) {
        for (int dy = -1; dy < 2; dy++) {
          uint32_t index = ((line+dy)*width)+dx+pixel;
          int32_t gray = grayscale[index];
          valueX += gray*gx_array[dy+1][dx+1];
          valueY += gray*gy_array[dy+1][dx+1];
        }
      }
      result = (valueX < 0) ? -valueX : valueX;
      result += (valueY < 0) ? -valueY : valueY;
      sobelResult[line*width+pixel] = (result > threshold) ? 0xFF : 0;
    }
  }
}
```

Code Block 2: C function for performing Sobel filtering.

The program takes five input variables:

- `grayscale`: A pointer to the grayscale input image array.
- `sobelResult`: A pointer to the array where the edge detection result will be stored.
- `width`: An integer representing the image width in pixels (e.g., 640 in our camera settings).
- `height`: An integer representing the image height in pixels (e.g., 480 in our camera settings).
- `threshold`: An integer representing the edge detection sensitivity, adjustable for different detection levels.

To define the kernels of the filter, we use two internal constants represented as 3×3 arrays: `gx_array` and `gy_array`. These arrays correspond to the horizontal and vertical Sobel operators, respectively.

Additionally, we declare three variables: `valueX`, `valueY`, and `result`. These variables store the vertical gradient, horizontal gradient, and the final computation, respectively.

We iterate over each pixel in the image using two nested loops, excluding border pixels to ensure a 3×3 neighbourhood is present around each pixel. For each neighbouring pixel, we fetch the corresponding grayscale value from the `grayscale` array. The horizontal and vertical gradient values (`valueX` and `valueY`) are then calculated by multiplying the grayscale value with the corresponding values from `gx_array` and `gy_array`.

To optimize memory access, we perform both the x and y calculations in the same loop, thus minimizing unnecessary memory accesses.

Rather than computing the total gradient magnitude, we make an approximation by calculating the absolute values of `valueX` and `valueY`, and then summing them.

Finally, we compare the computed result against the specified threshold. If the magnitude exceeds the threshold, we set the corresponding pixel in `sobelResult` to 255 (white), indicating an edge. Otherwise, it is set to 0 (black), indicating no edge.

2.2.2 Number of cycles estimation

To estimate the number of cycles we analyze the function step by step.

Outer Loops (lines and pixels):

```
for (int line = 1; line < height - 1; line++) {
    for (int pixel = 1; pixel < width - 1; pixel++) {
```

These loops iterate over the entire image except for the border. The total number of iterations is $(\text{height} - 2) \times (\text{width} - 2)$.

Inner Loops (dx and dy):

```
for (int dx = -1; dx < 2; dx++) {
    for (int dy = -1; dy < 2; dy++) {
```

These loops iterate over a 3×3 kernel for each pixel, so they always run 9 times per pixel.

Computations Inside the Inner Loop:

- **Index Calculation**:

  ```
  uint32_t index = ((line+dy)*width) + dx + pixel;
  ```

  This involves two additions, one multiplication, and one assignment.

- **Grayscale Access and Accumulation**:

  ```
  int32_t gray = grayscale[index];
  valueX += gray * gx_array[dy+1][dx+1];
  valueY += gray * gy_array[dy+1][dx+1];
  ```

$$\text{Grayscale Access Cycles} = 1(\text{memory access})$$

$$\text{Accumulation Cycles} = 2(\text{multiplications}) + 2(\text{additions})$$

Total cycles for the inner loop:

$$\begin{aligned}\text{Total Inner Loop Cycles} = (\text{index calculation cycles}) + 1(\text{grayscale access cycles}) \\ + (\text{accumulation operation cycles})\end{aligned}$$

Since this runs 9 times:

$$\text{Cycles per Pixel} = 9 \times \text{Total Inner Loop Cycles}$$

Edge Detection Calculation:

```
result = (valueX < 0) ? -valueX : valueX;
result += (valueY < 0) ? -valueY : valueY;
sobelResult[line*width + pixel] = (result > threshold) ? 0xFF : 0;
```

There are 3 comparisons and 3 assignments:

$$\text{Edge Detection Cycles} = 3(\text{comparisons}) + 3(\text{assignments})$$

Total Cycles per Pixel:

$$\text{Total Cycles per Pixel} = (\text{inner loops cycles}) + (\text{edge detection cycles})$$

Total Cycles for the Entire Image

For an image of size 640x480:

$$\text{Total Cycles} = (640 - 2) \times (480 - 2) \times \text{Total Cycles per Pixel}$$

2.2.3 Bottleneck Hot-Spotting

One of the most demanding tasks in this algorithm is accessing the 8 surrounding pixels for each operation, followed by applying the filter through multiplication, accumulation, and addition.

Our goal with the accelerator is to efficiently access these surrounding pixels at the same time, and possibly speed up this access by processing subsequent pixels using already retrieved data. Additionally, since the convolution operations for this filter involve straightforward multiplications by a constant power of 2 and sign inversions, we could improve the algorithm by adding specialized modules that can handle these operations without needing a multiplier.

Avoiding the need to compute indices can be achieved by organizing the storage of pixels in a way that makes sequential processing easier.

In summary, the strategies we're considering include: storing values with spatial correlation to avoid index computation, using a larger buffer to minimize memory access and allow the processing of more pixels with preloaded data, and simplifying the convolution operation to remove the need for a multiplier.

2.3 Movement detection

2.3.1 Code

```
void movementDetection(
                        uint8_t *sobelResult,
                        uint8_t *movementResult,
                        int32_t width,
                        int32_t height) {

    uint8_t last_sobel;
        for (int i = 0; i < width * height; i++) {
            last_sobel=movementResult[i];
            if (sobelResult[i]  == 255) {
                if (last_sobel!=127) {
```

```
                movementResult[i] = 0;
            } else {
                movementResult[i] = 255;
            }
        } else {
            movementResult[i] = 127;
        }
    }
```

Code Block 3: C function for performing Sobel filtering.

The function `movementDetection` processes an image to detect movement by comparing current edge detection results with previous states. It updates the movementResult array based on the edges detected by the sobelResult array.

The function needs 3 parameters:

- `sobelResult`: A pointer to an array that contains the results of the edge detection of the current frame.
- `movementResult`: A pointer to an array that contains the values of the previous motion detection of the previous frame in input and stores the result of the movement detection of the current frame at the output.
- `width`: An integer representing the image width (e.g., 640 in our camera settings).
- `height`: An integer representing the image height (e.g., 480 in our camera settings).
- `last_sobel`: A local variable to store the previous value from movementResult.

The function uses a loop to iterate over all the pixels of a frame. For each iteration, the movement detection value of the previous frame is stored in the local variable last_sobel. Then, the edge detection value of the current frame is analyzed. If there is a strong edge (sobelResult[i] == 255) and the previous pixel was not part of the background (last_sobel != 127), meaning it was an edge, the new pixel is defined as a steady edge. If the previous pixel was part of the background, the new pixel is defined as an edge in motion.
Finally, if the new pixel is not an edge, the motion detection value is set to background.

2.3.2 Number of cycles estimation

To estimate the number of cycles required to perform the `movementDetection` we analyze the function step by step.

Initialization of Local Variable:

```
uint8_t last_sobel;
```

This is a variable initialization and we assumed takes 1 cycle.

Main Loop:

```
for (int i = 0; i < width * height; i++) {
```

This loop iterates over the entire image. For an image of size 640x480, the total number of iterations is $640 \times 480 = 307,200$.

Loop Body:

- **Assignment of `last_sobel`**:

  ```
  last_sobel = movementResult[i];
  ```

- **Conditional Check and Nested Conditionals**:

  ```
  if (sobelResult[i] == 255) {
      if (last_sobel != 127) {
          movementResult[i] = 0;
      } else {
          movementResult[i] = 255;
      }
  } else {
      movementResult[i] = 127;
  }
  ```

  Each comparison and assignment within the conditionals will take 1 cycle each. We need to consider the operations in detail:

  - `if (sobelResult[i] == 255)` comparison
  - If true, the nested `if (last_sobel != 127)` comparison
    - If true, assignment `movementResult[i] = 0`:
    - Else, assignment `movementResult[i] = 255`:
  - Else (if the first condition is false), assignment `movementResult[i] = 127`:

  We assume that either the nested comparison or the else block will be executed, but not both. Total cycles for the conditionals (worst-case scenario):

$$\begin{aligned}\text{Conditional Cycles} = &\ (\text{first comparison}) + (\text{nested comparison}) \\ &+ (\text{assignment in nested if}) + (\text{assignment in else})\end{aligned}$$

Total Cycles per Iteration:

$$\text{Total Cycles per Iteration} = 1(\text{assignment}) + (\text{conditionals cycles})$$

Total Cycles for the Entire Image

For the entire image of size 640x480 the total cycles:

$$\text{Total Cycles} = 307,200 \times \text{Total Cycles per Iteration}$$

2.3.3 Bottlenecks Hot-Spotting

In this function, the main processing-demanding task is storing the values of the previous frame and performing the comparison. The primary optimization strategy would be to ensure that both values required for the comparison are readily available. To achieve this, incorporating an additional buffer could be a plausible solution.

2.4 Observations: Cycle estimation

Estimating the number of cycles required for each function in a program executed on an FPGA is often done as an approximation because it's challenging to precisely determine the cycle count for each operation. The number of cycles estimation was indicative and was useful just to perform an initial analysis of the bottlenecks.

This complexity arises from factors such as how the compiler translates the code into hardware logic and how the design is executed within the FPGA's architecture.

However, it is a good strategy to identify operations that are repeated or dependent on certain parameters. By recognizing these patterns, we can optimize the design to reduce latency and improve overall performance.

2.5 Profiling

To better understand how the code is being executed in the FPGA we used profiler counters that measured separately each function. The results are presented in the following table:

Step	Total cycles	Stall cycles	Bus IDLE cycles
Grayscale conversion	37,988,398	2,630,903	19,859,240
Sobel algorithm computation	334,789,259	25,557,325	155,497,776
Movement detection	21,766,022	16,537,111	10,732,944

Table 1: Performance summary of the 2 developed implementations. Cycles counts are reported as per frame.

In our initial estimation, we found that the Sobel function is the most processing-demanding, followed by the Grayscale conversion, and finally, the Movement detection function. It's noteworthy that a significant portion of the cycles for the Sobel and Movement detection functions is spent in Bus-Idle states, which can account for almost half of the total cycles. This highlights a bottleneck in data movement within the system. To mitigate this issue, implementing a DMA-based accelerator could offload the processor from data movement tasks, thereby reducing the number of Bus-Idle cycles and improving overall efficiency.

Additionally, stall cycles indicate that certain operations occupy the processor for prolonged periods, potentially affecting overall performance. To address this, developing more efficient modules that handle these operations could help minimize stall cycles and optimize processor utilization.

3 Hardware Acceleration

Having developed a working full-software function to perform Sobel and movement detection, and having identified the bottlenecks of this implementation, our next step was to accelerate this function through hardware. As we are working on an application-specific embedded system, we can modify the hardware modules, add new ones, or remove others. To enable these

modules to communicate with the hardware, the most common approach is to use custom instructions. These modules can communicate with the program running on the microcontroller's CPU through explicit assembly calls in C. The advantage of these hardware modules is that they allow for the acceleration of some tasks relative to the CPU: hardware operations can be parallelized, custom instruction modules can be placed near the memory, and these modules can operate while the CPU is performing other operations in parallel. Taking advantage of custom instructions was our approach to accelerating the program.

Custom instructions communicate with the CPU through a set of signals. There are two 32-bit input registers, named *ValueA* and *ValueB*, and one output register called *Result*. Additionally, there is a *CustomID* field used by the CPU to select which custom instruction to invoke and a *Start* signal that signals the custom instruction module to begin operation. Once the custom instruction is activated, the program waits for the custom instruction module to issue a *Done* signal before it proceeds to fetch the next instructions. This setup ensures that the CPU synchronizes properly with the execution of custom instructions, maintaining operational integrity and timing correctness.

3.1 Near-memory computing

As observed in the software implementation, the biggest factor that slows the program is data movement. For every produced pixel, all the 8 pixels around it are read from the memory, and the newly computed pixel is stored. Every time a pixel is loaded or stored from the memory, the CPU has to perform a load/store instruction, which usually takes multiple clock cycles due to the time taken to communicate with the bus and the time taken by the SRAM to write or read a value. The increase in latency due to waiting for the bus can be completely eliminated by placing the computing part "near" the memory, so that there is no time needed to wait for the bus, and the only latency contribution is given by the time taken by the memory to store or output a value. This approach is known as near-memory computing [6] [7].

Direct Memory Access (DMA) is a system that allows hardware subsystems to access main system memory independently, bypassing the CPU to improve data transfer rates and free up processor time. In the virtual prototype we were working on, we had available a DMA module that communicated with a near-custom-instruction dual-port memory (which we'll call CI memory). Our DMA was programmable through custom instruction calls in the C program. It could be programmed on:

- Burst size: the size of the blocks of data for the communications between the DMA and the bus.
- Block size: the total amount of data to be transferred in a DMA transfer.
- Memory start address: the address of the custom-instruction memory where the DMA starts writing or reading. Transferred data gets written or read on consecutive addresses.
- Bus start address: the bus address where the DMA starts writing or reading. As our microcontroller is memory-mapped, it can point to either the main memory or to a peripheral. Transferred data gets written or read on consecutive addresses.
- Control register: writing to this register initiates a DMA transfer. If we write '1', the transfer is from the bus to CI memory; if we write '2', it's in the opposite direction.

Moreover, there is the Status register, a register that can only be read (through a custom instruction) and informs the user whether the DMA is transferring data, if it is in an idle state, or if it has encountered a bus error.

3.2 Grayscale acceleration

In the previous implementation, the program captured a coloured picture from the camera consisting of 16-bit RGB pixels. It then iterated over each pixel to convert it to grayscale before performing Sobel edge detection. This process involved reading and storing each pixel and calculating its grayscale value through multiplications.

This can be highly optimized by modifying our hardware modules. The module that takes images from the camera and writes them to the image vector (*camera.v*) currently processes one pixel at a time, writing it to the output vector as it is. In a previous assignment, we had modified this module to read four RGB pixels at once, performing their grayscale conversion in a hardwired manner using shifts and sums, and then storing these four pixels. This approach significantly reduces the time required to grayscale the image to essentially the time needed to capture the frame. Additionally, this method allows the camera module to have fewer iterations as it writes four pixels at a time, hence the time taken by *TakeSingleImageBlocking()* is likely to decrease.

3.3 Sobel acceleration

3.3.1 Image tiling

As mentioned previously, the Sobel algorithm essentially performs a 2-dimensional convolution on the image using a 3x3 filter. To optimize this operation through near-memory computing, we decided to move the image into the CI memory and then perform the convolution near this memory. However, this memory does not have the capability to store the entire image, meaning that the convolution operation must be divided into subgroups of the image. The image was therefore tiled into rectangular blocks. Performing the convolution on these blocks separately, however, is not acceptable as it does not consider the correlation between data at the edges of each block (the Sobel filter would not account for the positions between different image blocks). To address this, the blocks should also include the $f-1$ first columns of the successive blocks, and the $f-1$ rows of the block below, where f is the size of the filter, in our case, 3.

Our image consists of 640x480 pixels, and since our pixels are grayscale (occupying a Byte), this means that an image consists of 307,200 bytes. When considering 32-bit words, we need 76,800 words for the whole image. For an efficient division of the image, we decided to split the image into rectangular blocks of 16x12 words each. Then, to perform the correct tiling, the actual blocks over which we perform the convolution are of 17x14 words, totalling 238 words. This tiling ensures the inclusion of necessary overlapping areas for accurate convolution across block boundaries.

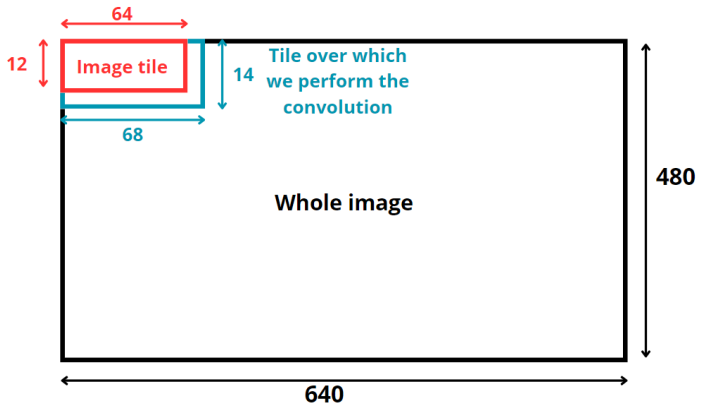

Figure 2: Illustration explaining how the image is tiled for performing Sobel. Reported numbers express the number of pixels.

3.3.2 DMA modifications

To facilitate rectangular tiling on the image, modifications to the DMA were necessary. Originally, the DMA transferred data to consecutive bus addresses starting from the specified bus start address. This approach resulted in storing image data as a continuous row rather than as a rectangular block. Since the DMA module automatically increments the bus address by one position for each word transferred, we needed a modification to adjust this incrementation. Specifically, after transferring a number of words equal to the burst size, the address increment needed to be modified by the length of an image row minus the length of a window row.

This modification ensures that subsequent bursts start at the correct position in the next row of the intended image block. However, it's important to note that with this modification, the DMA becomes fully customized for this specific tiling task and cannot be used for other types of transfers where block sizes are larger than the burst size. Also, the burst size of the transfer must be the length of the row of the image tile we want to transfer. This customization limits the DMA's flexibility but optimizes it for the specific requirements of our image-processing task.

```
//at end of window line, increase buststartaddress to skip the image line
s_busStartAddressShadowReg <= (s_dmaCurrentStateReg == INIT) ?
   s_busStartAddressReg :
    ((s_dmaCurrentStateReg==DO_READ && s_endTransactionInReg == 1'b1
       )||(s_dmaCurrentStateReg==END_WRITE_TRANSACTION) ) ?
       s_busStartAddressShadowReg + 32'd640 : s_busStartAddressShadowReg;
```

Code Block 4: Modifications performed on the already existent DMA module, observable at lines 176-177 of the file modules/ramdmaci/verilog/ramdmaci_sobel_movement_detection.

3.3.3 Dual port memory access

As mentioned, the ci memory is a dual-port one, meaning that it can be accessed by 2 different ports and read or written at the same time. this works by clocking the 2 ports on different clock edges. Previously, the 2 ports of the ci memory were connected one to the DMA and the other to the main CPU through the custom instruction interface. The user could either perform a

DMA transfer on the memory by writing on the DMA control register or perform a direct read or write of a single word, both by calling the same custom ID but with different configuration codes on the ValueA access. As for our purpose, the single-word access on the memory was not necessary anymore, so we decided to connect this second port to a module that performs Sobel filtering of the image. This way, sobel filtering on a part of the memory is possible while the other part gets written by DMA, allowing double buffering. Our memory was then divided into two blocks of $17 * 14 = 238$ 32-bit words for double buffering of the image tiles.

3.3.4 Sobel Module

We developed a module specifically designed to perform Sobel filtering on the image tile stored in the CI memory. This module was integrated within the DMA module itself to facilitate shared access to the same memory and enable effective intercommunication between the two modules. For simplicity and to maintain a unified interface, we decided to initiate Sobel filtering using the same custom ID as the DMA module, but with a specific word in the *ValueA* field. Additionally, the Sobel threshold value can be configured in the same manner.

The module operates by calculating the Sobel value for each word stored in the 238-word memory block opposite to the one used by the DMA. This means that the Sobel module uses a memory start address that is the opposite of the DMA module's start address (if the DMA starts at 0, Sobel starts at 238, and vice versa). Each word contains data for 4 pixels, and the Sobel values for these pixels are calculated in parallel. To compute the Sobel filter for a pixel, it is necessary to read the surrounding pixels from the memory. Therefore, while a single pixel requires reading from a 3x3 pixel window, calculating for four pixels at once necessitates reading from a 3x6 pixel window to ensure coverage of all surrounding pixels for each of the four pixels. The module iterates over all the possible 12x16 positions taken by the 3x6 window required for processing. For each of these positions, since the words contain 4 pixels, two words for each of the three rows of the window must be read. In the first word, all four pixels are read, while in the second word, only two are needed. Additionally, it is important to consider that the bus used reverses the endianness of the word, so pixels are stored in the inverted order within each word. For each window position, the module sequentially reads six words, gradually filling a window register. Also, the memory address from which the word is read needs to be incremented by the length of a row minus one (i.e., 16 positions) every two reads.

Once the window is filled, the Sobel value for 4 pixels is computed from it. The values are multiplied by constants, so the multiplications are performed through sums and shifts, and a multiplier is not required. After waiting a clock cycle for this computation to be performed, the computed 4 values, which will either be 0 or 255, are concatenated into a word using little endian logic (as the bus will switch the bytes again) and then stored back in the memory.

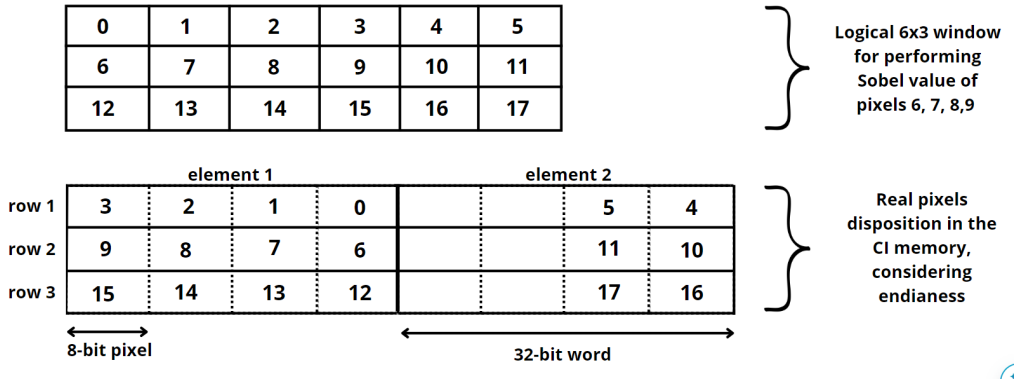

Figure 3: Illustration showing how the 6x3 window is read from the memory, considering the endianness switch performed by the bus.

3.4 Movement detection acceleration

As observed in the profiling of the software implementation, movement detection also represents a significant performance bottleneck. The issue with this operation is the need to store the entire image of the previous frame and compare each pixel with the new ones one by one. This results in a large number of loads and stores from the main memory, leading to a very slow operation. To try to accelerate this type of operation, we considered integrating this calculation with the Sobel filtering to reduce the number of steps.

To make this integration possible, a third buffer of $16 * 12 = 192$ words was added to the CI memory. In the program, we are writing the filtered Sobel image onto a vector for every frame. This means that right before writing a filtered image tile to this vector, the vector contains the previous values of the tile. The approach was then, right before calculating a Sobel tile, to store the previous tile in a third 238-word block of the memory. At this point, the only step added to the Sobel state machine was to perform an additional read of the 4 previous values of the 4 pixels that are computed. This modification allows the Sobel computation to adjust by writing 127 (gray) in any pixel that is not on an edge, writing 0 (black) in pixels that remained on an edge from previous cycles, and writing 255 (white) in pixels that are now on an edge and were not before, indicating that there was movement.

This implementation allows us to reduce the latency added by the movement detection to only one read cycle per produced pixel, plus the time taken to copy the previous image tiles to the third 238-word memory block through DMA. This significantly accelerates the movement detection process.

The full code of the developed modules that perform DMA transfers, Sobel filtering and movement detection is available in Code Block 5.

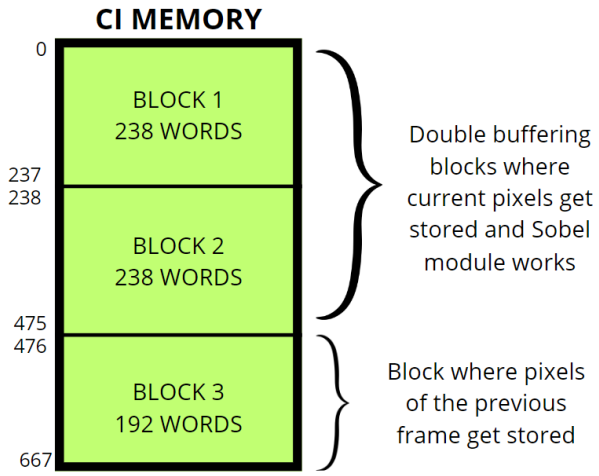

Figure 4: Illustration of the logical data disposition in the custom instruction memory.

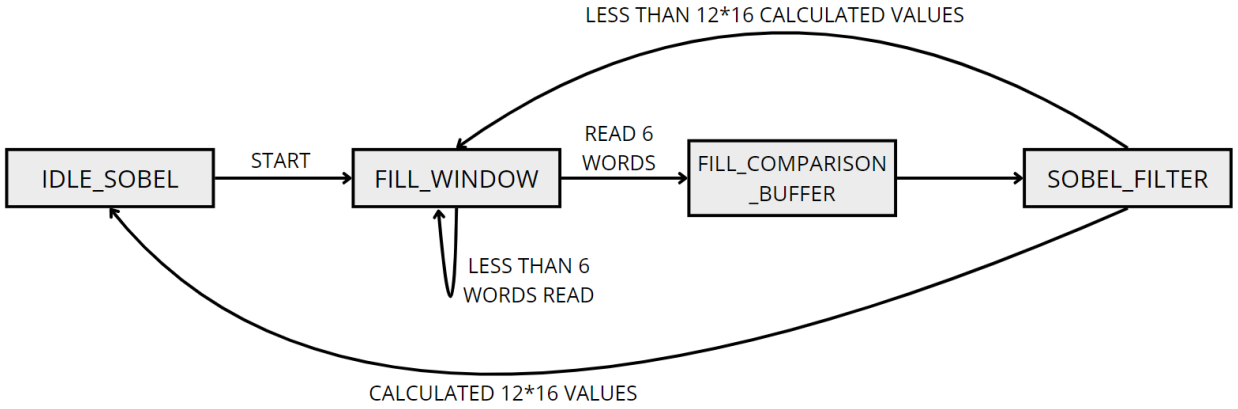

Figure 5: Simplified illustration of the state machine that performs the Sobel operation. IDLE_SOBEL is the state where no calculations are being performed. FILL_WINDOW is the state where the 6x3 pixels window is filled. In FILL_COMPARISON_BUFFER, the previous pixels are read, and in SOBEL_FILTER, the computation is performed.

3.5 Full program

Now, the final step was to take advantage of the developed hardware modules by creating a software function that effectively calls these custom instructions. As mentioned, our strategy involved taking advantage of the DMA module by using the double buffering technique.

Our program, through the *takeSingleImageBlocking()* function, captures a full grayscale image from the camera module and stores this image in a vector. The image is automatically tiled by the DMA, thanks to our modifications, and these tiles are saved in one of the two ping-pong buffers of the CI memory. Each iteration alternates the tile storage between the buffers.

While one buffer is being written to, Sobel filtering is performed on the other, which contains the image tile stored during the previous iteration. It is important to note that before starting this filtering, the 192 values of the filtered pixels from the same tile in previous frames are stored in the third buffer through DMA.

After each iteration, once the output pixel values have been computed, a DMA transfer must be initiated from the CI memory block containing them to the output image vector.

Some additional considerations have to be made:

- Before starting to iterate, the first image tile must be stored in the first buffer.
- The number of total iterations is $\frac{160\times480}{12\times16} = 400$.
- On the last iteration, only Sobel filtering is performed, and input pixels are not written to the CI memory.
- The input tiles that need to be stored have a size of 17×14, while the output tiles are 16×12. This means that the burst and block size must be switched each time, respectively, between 17 and 238 to 16 and 192.
- After each iteration, both input and output bus start addresses must be rewritten. Normally, in both cases, they are incremented by 64 (as these addresses are expressed in Bytes). However, after $\frac{160}{16} = 10$ iterations, when we reach the end of a row, we have to increase the bus start address by 64 plus the length of 11 image rows, so by $640 \times 11 + 64$.
- Every time a DMA transfer needs to be started, we have to wait for the previous running DMA transfers and Sobel computations to end. To do so, we read the Sobel and DMA status registers inside a *while(1)* loop.
- To avoid detecting movement on every pixel of the first frame, every pixel of the output vector gets initialized to 127 (gray) before starting the program.

The full C function that performs this operation is available on Code Block 6

4 Results and discussions

4.1 Step-by-step profiling

At this point, step-by-step profiling has been performed for both implementations to evaluate how the acceleration strategy has performed. The results are reported in Table 2. First and foremost, it is evident that almost all steps have been significantly accelerated.

Here, the most significant bottleneck of the accelerated version is the initialization of the frame, which is executed through the *TakeSingleImageBlocking()* C function. This function captures an image from the camera and writes this input into a C vector. As previously described, the camera module has been modified to output four pixels at a time, already in grayscale. This modification effectively almost halves the cycles taken by the network to read the image. However, this step remains the slowest part in this accelerated version.

Thanks to the modifications in the *camera.v* module, the cycles taken by the task of grayscaling the pixels are reduced to essentially zero, as the grayscale is computed in a hardwired manner using shifts and additions, taking less than a cycle.

The most noticeable difference is observed in the Sobel component, as this was the part that involved extensive memory operations, including reading the surrounding eight pixels for each pixel and writing the processed pixel back to memory. These operations were replaced by a

DMA transfer to the CI memory and the performance of Sobel filtering on a memory block. Since these operations occur in a ping-pong buffer, the DMA transfer and filtering happen simultaneously, meaning their delays do not need to be summed; instead, the total delay will be dictated by the slower of the two. This component greatly benefits from the advantages of near-memory computing, as the large volume of writes and reads takes only one cycle in this case, instead of the many cycles required to perform loads and stores through the system bus. This efficiency is observable in the reduction of idle cycles, which decreased from about 76% of the total cycles to 14%. This improvement means that the central CPU is used more efficiently, primarily because it spends far fewer cycles waiting for memory writes and reads.

Also, movement detection has been significantly accelerated. In this accelerated version, the time overhead introduced by this operation is primarily determined by the time taken to move the image tile of the previous frame to the third buffer of the CI memory. Additionally, an extra cycle is required for each produced pixel to read the pixel of the previous frame during the Sobel filtering. This low overhead has allowed us to greatly reduce the latency associated with movement detection, which previously involved reading the previous pixel and writing back the new one to the memory for each pixel. A significant difference is observable in the bus-idle cycles, which decreased from 45% to 16%. This reduction effectively indicates that the time taken by movement detection, essentially consisting of data movement on the bus performed by DMA, is more efficiently utilized, as the bus is almost always busy and actively moving data.

Furthermore, we must consider that in this accelerated version, the output pixels from the Sobel and movement detector are stored in the CI memory, unlike in the full-software version where they are stored in the output image vector. This means that this version requires an overhead in cycles due to the time taken by the DMA to move each image output tile to the output vector. However, this increase in latency is negligible considering the significant speed-ups obtained in other parts of the program.

Task	Version	Total cycles	Stall	Bus IDLE cycles
Frame initialization	Software	8,480,465	112	5,803,692
Frame initialization	Accelerated	4,546,114	282	3,158,094
Performing grayscale	Software	37,914,955	26,235,588	19,861,094
Performing grayscale	Accelerated	0	0	0
Sobel filtering	Software	334,951,131	255,702,422	155,571,670
Sobel filtering	Accelerated	908,164	128,949	508,836
Movement detection	Software	22,175,170	16,840,035	10,927,688
Movement detection	Accelerated	469,422	380,622	77,521
Moving results to output image	Software	0	0	0
Moving results to output image	Accelerated	948,308	134,666	40,766

Table 2: Step by step profiling result of the 2 implementations. In the software version, there is no time taken for moving data to the output image as the movement detection function directly writes values in that vector.

4.2 Final results

Finally, profiling has been performed on the full application. As evident from the results listed in Table 3, the overall improvement is significant. In particular:

- The total number of cycles needed to process a frame decreased by 97.5
- The stall cycles decreased by 99.75
- The bus-idle cycles decreased by 96.6
- The total frames per second increased by a factor of 38.

As discussed, these improvements are mainly due to fewer reads and writes being performed on the main SRAM memory. This is particularly evident in the decrease in stall cycles. A significant amount of time, during which the CPU is stalled waiting for the bus, is saved in this manner.

Regarding the final image, we can notice more image noise in the accelerated version, which is more visible due to the higher frame rate. This is primarily caused by the poor camera quality available. Nonetheless, the accelerated version effectively performs visible real-time edge and movement detection, which is essentially impossible in the other version that analyzes a frame every 5 seconds.

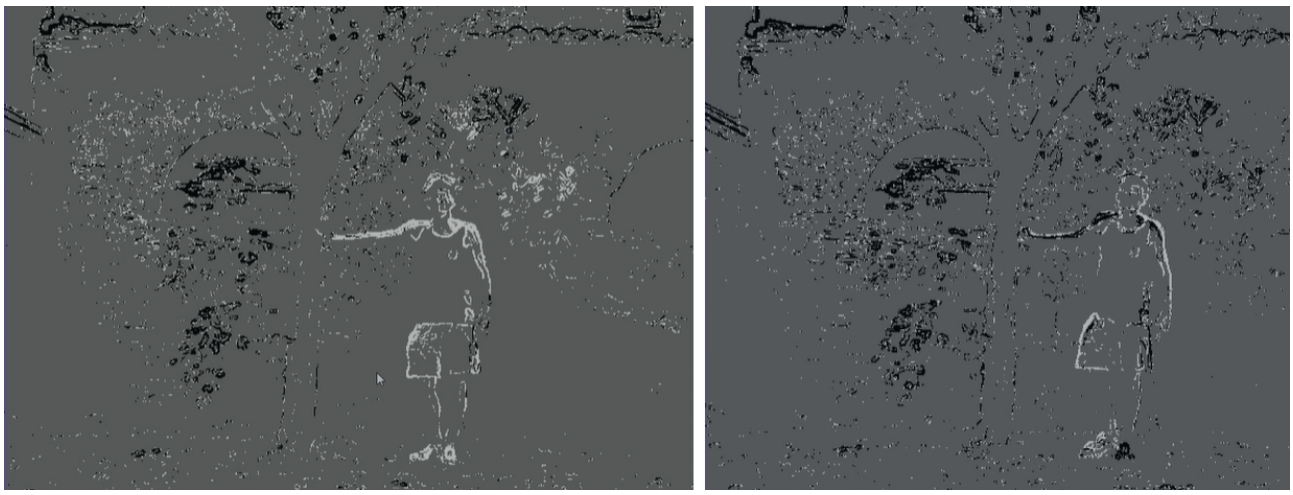

Figure 6: Screenshot comparison of software (left) and hardware (right) implementations. In the software version, the person appears fully white because, even though the person was not moving at the moment, it was in a different position in the previous frame taken 5 seconds earlier, hence movement is detected.

Version	Total cycles	Stall	Bus IDLE cycles	Frames per second
Software implementation	390,650,058	289,363,191	186,032,916	0.2
Accelerated version	9,834,322	719,008	6,358,410	7.6

Table 3: Performance summary of the 2 developed implementations. Cycles counts are reported as per frame. Frames per second are estimated considering our clock frequency of 74.24 MHz.

5 Conclusions

This report analyzed all the steps performed for the acceleration of an edge and movement detection application on an embedded FPGA board. It began by describing the context and the algorithms involved. Then, a detailed analysis of the software version was presented. At this stage, our reasoning on how to accelerate the process was discussed. Subsequently, an overview of the decided hardware acceleration strategy was fully described and documented. Finally, the results of the two implementations were reported and evaluated.

The first modification made to the system was performed on the camera module. The new version was capable of reading four pixels at a time, grayscaling them in parallel, and storing them in groups of four. This change not only accelerated the process of reading an image from the camera, but also reduced the time taken to grayscale the pixels to zero. The main bottleneck of the full software implementation was observed to be the data movement caused by reads and writes to the main memory of the microcontroller, which caused the main CPU to stall while waiting for the data to be ready. To solve this problem, a near-memory computing approach was adopted, meaning that calculations were moved closer to the memory. This allowed for direct memory reads and writes instead of passing through the bus with load and store instructions. Additionally, a double buffering approach was utilized, thanks to the dual-port nature of a custom instruction memory we instantiated. This approach allowed for the parallelization of computation with data movement, significantly reducing total latency.

This acceleration process led to positive outcomes, resulting in the significant speedups documented in Table 3.

5.1 Potential future improvements

At this point, we can conclude by reasoning on possible future improvements.

First, reducing the size of the output pixels could decrease the latency associated with data movement. In our current version, we are moving 8-bit pixels to the output of the network. However, these pixels only take three values, representing white, gray, and black, which means that a pixel can be represented with just 2 bits. Implementing this change could reduce the time taken to move the output values to the image vector by a factor of four. However, such a modification would require changes to the VGA functions, the ones that display the output image on the screen. Since these functions currently do not support 2-bit pixels, this capability would need to be implemented. An alternative to modifying the VGA function would be to extend the 2-bit pixels to a byte via software. However, the time lost due to waiting for memory writes and reads would likely be greater than the time taken to move more data through the DMA.

Another possible improvement could be transforming our CI memory into a vector register file, over which we can perform matrix multiplication. In fact, convolutions can be effectively performed through simple matrix multiplication with a prior data arrangement known as Im2Col [8]. Modifying the DMA to automatically arrange this data in the CI memory and instantiating a matrix multiplier in place of the Sobel module could significantly reduce computation time. However, with our filter size of 3, all the input data would need to be stored three times, resulting in a tripling of the data movement for the input. Since matrix multiplication and data

movement occur concurrently in the double buffering system, a study should be conducted to determine whether the advantages of simplified computation might be offset by an increase in data movement latency.

6 Appendix

```
module ramDmaCi #( parameter [7:0] customIdDMA = 8'h00 )
                  ( input wire         start,
                                       clock,
                                       reset,
                    input wire [31:0]  valueA,
                                       valueB,
                    input wire [7:0]   ciN,
                    output wire        done ,
                    output wire [31:0] result,
                    // Here the required bus signals are defined
                    output wire        requestTransaction,
                    input wire         transactionGranted,
                    input wire         endTransactionIn,
                                       dataValidIn,
                                       busErrorIn,
                                       busyIn,
                    input wire [31:0]  addressDataIn,
                    output reg         beginTransactionOut,
                                       readNotWriteOut,
                                       endTransactionOut,
                    output wire        dataValidOut,
                    output reg [3:0]   byteEnablesOut,
                    output reg [7:0]   burstSizeOut,
                    output wire [31:0] addressDataOut);
 localparam [7:0] customIdSobel = 8'd21;
  wire s_isMyCi = (ciN == customIdDMA) ? start : 1'b0;
  wire s_isSramWrite = (valueA[31:10] == 22'd0) ? s_isMyCi & valueA[9] :
     1'b0;
  wire s_isSramRead  = s_isMyCi & ~valueA[9];
  reg s_isSramReadReg;
  assign done   = (s_isMyCi & valueA[9]) | s_isSramReadReg | s_isMyCiSobel;
  always @(posedge clock) s_isSramReadReg = ~reset & s_isSramRead;
  reg[31:0] s_busStartAddressReg;
  reg[8:0]  s_memoryStartAddressReg;
  reg[9:0]  s_blockSizeReg;
  reg[7:0]  s_usedBurstSizeReg;
  //register that holds the treshold value for the sobel filter.
  //The treshold value idetermines the sensibility of the edge detection
  //The higher the value the less edges are detected (0 to 255)
  reg[7:0]  s_sobelTresholdReg;
  always @(posedge clock)
    begin
      s_busStartAddressReg    <= (reset == 1'b1) ? 32'd0 :
                                 (s_isMyCi == 1'b1 && valueA[12:9] ==
                                    4'b0011) ? valueB :
                                    s_busStartAddressReg;
      s_memoryStartAddressReg <= (reset == 1'b1) ? 9'd0 :
```

```
                                (s_isMyCi == 1'b1 && valueA[12:9] ==
                                   4'b0101) ? valueB[8:0] :
                                   s_memoryStartAddressReg;
    s_blockSizeReg          <= (reset == 1'b1) ? 10'd0 :
                                (s_isMyCi == 1'b1 && valueA[12:9] ==
                                   4'b0111) ? valueB[9:0] : s_blockSizeReg;
    s_usedBurstSizeReg      <= (reset == 1'b1) ? 8'd0 :
                                (s_isMyCi == 1'b1 && valueA[12:9] ==
                                   4'b1001) ? valueB[7:0] :
                                   s_usedBurstSizeReg;
    s_sobelTresholdReg      <= (reset==1'b1)  ? 8'd127 :
                                (s_isMyCi == 1'b1 && valueA[12:9] ==
                                   4'b1101) ? valueB[7:0] :
                                   s_sobelTresholdReg;
  end
reg s_endTransactionInReg, s_dataValidInReg;
reg [31:0] s_addressDataInReg;
always @(posedge clock)
  begin
    s_endTransactionInReg <= endTransactionIn;
    s_dataValidInReg      <= dataValidIn;
    s_addressDataInReg    <= addressDataIn;
  end
reg [9:0] s_ramCiAddressReg;
wire s_ramCiWriteEnable;
wire [31:0] s_busRamData;
dualPortSSRAM #( .bitwidth(32),
                 .nrOfEntries(668)) memory
               ( .clockA(clock),
                 .clockB(~clock),
                 .writeEnableA(writeEnableSobel),
                 .writeEnableB(s_ramCiWriteEnable),
                 .addressA(addressSobel),
                 .addressB(s_ramCiAddressReg),
                 .dataInA(s_sobelDataInReg),
                 .dataInB(s_addressDataInReg),
                 .dataOutA(sobelDataOut),
                 .dataOutB(s_busRamData));
localparam [3:0] IDLE = 4'd0;
localparam [3:0] INIT = 4'd1;
localparam [3:0] REQUEST_BUS = 4'd2;
localparam [3:0] SET_UP_TRANSACTION = 4'd3;
localparam [3:0] DO_READ = 4'd4;
localparam [3:0] WAIT_END = 4'd5;
localparam [3:0] DO_WRITE = 4'd6;
localparam [3:0] END_TRANSACTION_ERROR = 4'd7;
localparam [3:0] END_WRITE_TRANSACTION = 4'd8;
reg [3:0] s_dmaCurrentStateReg, s_dmaNextState;
reg       s_busErrorReg;
reg       s_isReadBurstReg;
reg[8:0]  s_wordsWrittenReg;
// a dma action is requested by the ci:
wire s_requestDmaIn = (valueA[12:9] == 4'b1011) ? s_isMyCi & valueB[0] &
   ~valueB[1] : 1'b0;
wire s_requestDmaOut = (valueA[12:9] == 4'b1011) ? s_isMyCi & ~valueB[0]
   & valueB[1] : 1'b0;
```

```
wire s_dmaIsBusy = (s_dmaCurrentStateReg == IDLE) ? 1'b0 : 1'b1;
wire s_dmaDone;
// here we define the next state
always @*
  case (s_dmaCurrentStateReg)
    IDLE                  : s_dmaNextState <= (s_requestDmaIn == 1'b1 ||
       s_requestDmaOut == 1'b1) ? INIT : IDLE;
    INIT                  : s_dmaNextState <= REQUEST_BUS;
    REQUEST_BUS           : s_dmaNextState <= (transactionGranted ==
       1'b1) ? SET_UP_TRANSACTION : REQUEST_BUS;
    SET_UP_TRANSACTION    : s_dmaNextState <= (s_isReadBurstReg == 1'b1)
       ? DO_READ : DO_WRITE;
    DO_READ               : s_dmaNextState <= (busErrorIn == 1'b1) ?
       WAIT_END:
                                              (s_endTransactionInReg ==
                                                 1'b1 && s_dmaDone ==
                                                 1'b1) ? IDLE :
                                              (s_endTransactionInReg ==
                                                 1'b1) ? REQUEST_BUS :
                                                 DO_READ;
    WAIT_END              : s_dmaNextState <= (s_endTransactionInReg ==
       1'b1) ? IDLE : WAIT_END;
    DO_WRITE              : s_dmaNextState <= (busErrorIn == 1'b1) ?
       END_TRANSACTION_ERROR :
                                              (s_wordsWrittenReg[8] ==
                                                 1'b1 && busyIn == 1'b0)
                                                 ? END_WRITE_TRANSACTION
                                                 : DO_WRITE;
    END_WRITE_TRANSACTION : s_dmaNextState <= (s_dmaDone == 1'b1) ? IDLE
       : REQUEST_BUS;
    default               : s_dmaNextState <= IDLE;
  endcase
always @(posedge clock)
  begin
    s_dmaCurrentStateReg <= (reset == 1'b1) ? IDLE : s_dmaNextState;
    s_busErrorReg        <= (reset == 1'b1 || s_dmaCurrentStateReg ==
       INIT) ? 1'b0 :
                            (s_dmaCurrentStateReg == WAIT_END ||
                               s_dmaCurrentStateReg ==
                               END_TRANSACTION_ERROR) ? 1'b1 :
                               s_busErrorReg;
    s_isReadBurstReg     <= (s_dmaCurrentStateReg == IDLE) ?
       s_requestDmaIn : s_isReadBurstReg;
  end
reg[31:0] s_busStartAddressShadowReg;
reg[9:0]  s_blockSizeShadowReg;
wire s_doBusWrite = (s_dmaCurrentStateReg == DO_WRITE) ? ~busyIn &
   ~s_wordsWrittenReg[8] : 1'b0;
/* the second condition is the special case where the end of transaction
   collides with the last data valid in */
assign s_dmaDone = (s_blockSizeShadowReg == 10'd0 ||
                    (s_blockSizeShadowReg == 10'd1 &&
                       s_endTransactionInReg == 1'b1 && s_dataValidInReg
                       == 1'b1)) ? 1'b1 : 1'b0;
assign s_ramCiWriteEnable = (s_dmaCurrentStateReg == DO_READ) ?
   s_dataValidInReg : 1'b0;
```

```
always @(posedge clock)
  begin
    //at end of window line, increase buststartaddress to skip the image
       line
    s_busStartAddressShadowReg <= (s_dmaCurrentStateReg == INIT) ?
       s_busStartAddressReg :
                                  ((s_dmaCurrentStateReg==DO_READ &&
                                     s_endTransactionInReg == 1'b1
                                     )||(s_dmaCurrentStateReg==END_WRITE_TRANSACTION
                                     ) ? s_busStartAddressShadowReg +
                                     32'd640 : s_busStartAddressShadowReg;
    s_blockSizeShadowReg       <= (s_dmaCurrentStateReg == INIT) ?
       s_blockSizeReg :
                                  (s_ramCiWriteEnable == 1'b1 ||
                                     s_doBusWrite == 1'b1) ?
                                     s_blockSizeShadowReg - 10'd1 :
                                     s_blockSizeShadowReg;
    s_ramCiAddressReg          <= (s_dmaCurrentStateReg == INIT) ?
       {1'b0,s_memoryStartAddressReg} :
                                  (s_ramCiWriteEnable == 1'b1 ||
                                     s_doBusWrite == 1'b1) ?
                                     s_ramCiAddressReg + 10'd1 :
                                     s_ramCiAddressReg;
  end
reg   s_dataOutValidReg;
reg [31:0] s_addressDataOutReg;
wire [9:0] s_maxBurstSize = {2'd0,s_usedBurstSizeReg} + 10'd1;
wire [9:0] s_restingBlockSize = s_blockSizeShadowReg - 10'd1;
wire [7:0] s_usedBurstSize = (s_blockSizeShadowReg > s_maxBurstSize) ?
   s_usedBurstSizeReg : s_restingBlockSize[7:0];
assign requestTransaction = (s_dmaCurrentStateReg == REQUEST_BUS) ? 1'd1
   : 1'd0;
assign dataValidOut = s_dataOutValidReg;
assign addressDataOut = s_addressDataOutReg;
always @(posedge clock)
  begin
    beginTransactionOut <= (s_dmaCurrentStateReg == SET_UP_TRANSACTION) ?
       1'b1 : 1'b0;
    readNotWriteOut     <= (s_dmaCurrentStateReg == SET_UP_TRANSACTION) ?
       s_isReadBurstReg : 1'b0;
    byteEnablesOut      <= (s_dmaCurrentStateReg == SET_UP_TRANSACTION) ?
       4'hF : 4'd0;
    burstSizeOut        <= (s_dmaCurrentStateReg == SET_UP_TRANSACTION) ?
       s_usedBurstSize : 8'd0;
    s_addressDataOutReg <= (s_dmaCurrentStateReg == DO_WRITE && busyIn ==
       1'b1) ? s_addressDataOutReg :
                           (s_doBusWrite == 1'b1) ? s_busRamData :
                           (s_dmaCurrentStateReg == SET_UP_TRANSACTION) ?
                              {s_busStartAddressShadowReg[31:2],2'd0} :
                              32'd0;
    s_wordsWrittenReg   <= (s_dmaCurrentStateReg == SET_UP_TRANSACTION) ?
       {1'b0,s_usedBurstSize} :
                           (s_doBusWrite == 1'b1) ? s_wordsWrittenReg -
                              9'd1 : s_wordsWrittenReg;
    endTransactionOut   <= (s_dmaCurrentStateReg == END_TRANSACTION_ERROR
       || s_dmaCurrentStateReg == END_WRITE_TRANSACTION) ? 1'b1 : 1'b0;
```

```
      s_dataOutValidReg    <= (busyIn == 1'b1 && s_dmaCurrentStateReg ==
         DO_WRITE) ? s_dataOutValidReg : s_doBusWrite;
    end
  reg[31:0] s_result;
  always @*
    case (valueA[12:10])
      3'b000    : s_result <= {31'd0,sobelStatusRegister};
      3'b001    : s_result <= s_busStartAddressReg;
      3'b010    : s_result <= {23'd0,s_memoryStartAddressReg};
      3'b011    : s_result <= {22'd0,s_blockSizeReg};
      3'b100    : s_result <= {24'd0,s_usedBurstSizeReg};
      3'b101    : s_result <= {30'd0,s_busErrorReg,s_dmaIsBusy};
      default   : s_result <= 32'd0;
    endcase
  assign result = (s_isSramReadReg == 1'b1) ? s_result : 32'd0;
  // SOBEL with movement detection
  wire isMyCiSobel = (ciN == customIdSobel) ? start : 1'b0;
  reg s_isMyCiSobel;
  always @(posedge clock) s_isMyCiSobel <= (reset==1'b1) ? 1'b0 :
     isMyCiSobel;
  reg [2:0] counterSobel;                    // counter for the 3x3 window
  wire sobel_done, counter_done;             // flags for the end of the
     sobel computation
  wire [4:0] windowIndex;
  assign windowIndex = (counterSobel == 3'd0||counterSobel == 3'd1) ? 5'd0:
                       (counterSobel == 3'd2) ? 5'd4:
                       (counterSobel == 3'd3) ? 5'd6:
                       (counterSobel == 3'd4) ? 5'd10:
                       (counterSobel == 3'd5) ? 5'd12:
                        5'd16;
  reg [7:0] window [17:0];                   // 3x6 window
  reg [8:0]  initialCounterWord;                    // counter for the
     pixels on the image, it extends the 32-bit address to a 8-bit one
  reg [7:0] counterWindow;                   // counter for the positions
     taken by the window
  wire [31:0] sobelDataOut;
  reg [31:0] s_sobelDataOut;                // output of the memory to the
     sobel modules
  always @* s_sobelDataOut <= sobelDataOut;
  wire [31:0] s_sobelDataIn;
  reg [31:0] s_sobelDataInReg;              // input of the memory from the
     sobel module
  always @(posedge clock) s_sobelDataInReg <= s_sobelDataIn;
  // start address for the sobel filtering, it is either 0 or 238, the
     opposite of the start address of the dma, to allow the ping pong buffer
  wire [9:0] sobelMemoryStartAddress;
  assign sobelMemoryStartAddress = (s_memoryStartAddressReg==9'd0)? 10'd238
     : 10'd0; // start address for the sobel memory allowing the ping pong
     buffer.
  wire [9:0] addressSobel = (s_SobelCurrentStateReg == SOBEL_FILTER) ?
     addressSobelWrite :
     (s_SobelCurrentStateReg==FILL_COMPARISON_BUFFER ||
     (s_SobelCurrentStateReg==FILL_WINDOW && counterSobel==3'd7))?
     addressSobelComparison :
                                addressSobelRead;
```

```
  reg [9:0] addressSobelRead;          // address to read the from the
     memory and fill the window
  reg [9:0] addressSobelWrite;         // address to write the
     sobel+movement detection output
  reg [9:0] addressSobelComparison;   // address to read the pixels of the
     previous frame, to perform the movement detection
  wire writeEnableSobel;
  assign writeEnableSobel = (s_SobelCurrentStateReg == SOBEL_FILTER ) ?
     1'b1 : 1'b0; // enable the write operation only when the sobel filter
     is applied
  assign counter_done = (counterSobel == 3'd7) ? 1'b1 : 1'b0;  // when the
     window has been filled
  assign sobel_done = (counterWindow == 8'd223) ? 1'b1 : 1'b0; // when the
     whole image has been processed
  localparam [1:0] IDLE_SOBEL = 2'd0;                 // idle state
  localparam [1:0] FILL_WINDOW = 2'd1;               // fill the 3x6 window
  localparam [1:0] FILL_COMPARISON_BUFFER = 2'd2;  // read the same pixel
     of the previous frame for movement detection
  localparam [1:0] SOBEL_FILTER = 2'd3;           // apply the sobel filter
     for 4 pixels at a time and perform the movement detection
  reg [1:0] s_SobelCurrentStateReg, s_SobelNextState;
  wire sobelStatusRegister;
  assign sobelStatusRegister = (s_SobelCurrentStateReg == IDLE_SOBEL) ?
     1'b0 : 1'b1; // to wait for the end of the sobel filter
  integer i;
  reg[31:0] comparisonBuffer;  // buffer to store the values of the
     previous frame
  // SOBEL state machine
 always @*
    case (s_SobelCurrentStateReg)
      IDLE_SOBEL       : s_SobelNextState <= (s_isMyCiSobel) ? FILL_WINDOW
         : IDLE_SOBEL;
      FILL_WINDOW      : s_SobelNextState  <= (counter_done) ?
         FILL_COMPARISON_BUFFER : FILL_WINDOW;
      FILL_COMPARISON_BUFFER : s_SobelNextState  <=  SOBEL_FILTER;
      SOBEL_FILTER     : s_SobelNextState  <=  (sobel_done) ? IDLE_SOBEL  :
         FILL_WINDOW;
      default          : s_SobelNextState  <= IDLE_SOBEL;
    endcase
always @(posedge clock)
    begin
      s_SobelCurrentStateReg <= (reset==1'b1) ? IDLE_SOBEL :
         s_SobelNextState;
    end
always @(posedge clock)
begin
    counterWindow <= (s_SobelCurrentStateReg == IDLE_SOBEL) ? 8'd0 :
                     (s_SobelCurrentStateReg == SOBEL_FILTER) ?
                        counterWindow + 1 : counterWindow;
    initialCounterWord <= (s_SobelCurrentStateReg == IDLE_SOBEL) ?
       sobelMemoryStartAddress:
                      (counterSobel==3'd5)? ((counterWindow[3:0] == 4'd15)
                         ? initialCounterWord + 9'd2 :
                      initialCounterWord + 9'd1) : initialCounterWord;
    addressSobelRead <= (s_SobelCurrentStateReg == SOBEL_FILTER ||
       s_SobelCurrentStateReg == IDLE_SOBEL)? {1'b0,initialCounterWord}:
```

```
                        (s_SobelCurrentStateReg == FILL_WINDOW)?
                        ((counterSobel == 3'd1 || counterSobel == 3'd3) ?
                           addressSobelRead + 10'd16 :(counterSobel > 4) ?
                           addressSobelRead :addressSobelRead + 10'd1) :
                         addressSobelRead;
    addressSobelComparison <= addressSobelWrite + 10'd239 +
       {1'b0,s_memoryStartAddressReg}; //position where to read the sobel
       values of the previous iteration
    addressSobelWrite <= (s_SobelCurrentStateReg == IDLE_SOBEL)?
       sobelMemoryStartAddress :
                       (s_SobelCurrentStateReg == SOBEL_FILTER)?
                          addressSobelWrite+1 : addressSobelWrite;
    comparisonBuffer <=(reset==1'b1)? 32'b0 : ( (s_SobelCurrentStateReg ==
       FILL_COMPARISON_BUFFER) ? s_sobelDataOut : comparisonBuffer);
                               if(s_SobelCurrentStateReg == IDLE_SOBEL)begin
                        for (i=0; i<18; i=i+1)begin
                               window[i] <= 8'd0 ;
                        end
    end
    if(counterSobel != 3'd7 && counterSobel!=3'd0 )begin
      window[windowIndex] <= (s_SobelCurrentStateReg == FILL_WINDOW) ?
         s_sobelDataOut[7:0] : window[windowIndex];
      window[windowIndex+1] <= (s_SobelCurrentStateReg == FILL_WINDOW) ?
         s_sobelDataOut[15:8] : window[windowIndex+1];
      window[windowIndex+2] <= (s_SobelCurrentStateReg == FILL_WINDOW &&
         (windowIndex==5'd0 || windowIndex==5'd12 || windowIndex==5'd6 ) )
         ? s_sobelDataOut[23:16] : window[windowIndex+2];
      window[windowIndex+3] <= (s_SobelCurrentStateReg == FILL_WINDOW &&
         (windowIndex==5'd0 || windowIndex==5'd12 || windowIndex==5'd6 ) )
         ? s_sobelDataOut[31:24] : window[windowIndex+3];
    end
    else begin
      window[windowIndex] <= window[windowIndex];
      window[windowIndex+1] <= window[windowIndex+1];
      window[windowIndex+2] <= window[windowIndex+2];
      window[windowIndex+3] <= window[windowIndex+3];
    end
    counterSobel <= (s_SobelCurrentStateReg == FILL_WINDOW) ? counterSobel
       + 1 : 3'd0;
end
/// Calculate Gx for 4 pixels
wire signed [9:0] Gx_0 = window[0] - window[2] + ((window[6]<<1)) -
   ((window[8])<<<1) + window[12] - window[14];
wire signed [9:0] Gx_1 = window[1] - window[3] + ((window[7]<<1)) -
   ((window[9])<<<1) + window[13] - window[15];
wire signed [9:0] Gx_2 = window[2] - window[4] + ((window[8]<<1)) -
   ((window[10])<<<1) + window[14] - window[16];
wire signed [9:0] Gx_3 = window[3] - window[5] + ((window[9]<<1)) -
   ((window[11])<<<1) + window[15] - window[17];
// Calculate Gy for 4 pixels
wire signed [9:0] Gy_0 = window[12] + (window[13]<<<1) + window[14] -
   (window[0] + (window[1]<<1) + window[2]);
wire signed [9:0] Gy_1 = window[13] + (window[14]<<<1) + window[15] -
   (window[1] + (window[2]<<1) + window[3]);
wire signed [9:0] Gy_2 = window[14] + (window[15]<<<1) + window[16] -
   (window[2] + (window[3]<<1) + window[4]);
```

```
wire signed [9:0] Gy_3 = window[15] + (window[16]<<<1) + window[17] -
    (window[3] + (window[4]<<1) + window[5]);
// Compute absolute values of Gx and Gy
wire [9:0] abs_Gx_0 = (Gx_0 < 0) ? -Gx_0 : Gx_0;
wire [9:0] abs_Gx_1 = (Gx_1 < 0) ? -Gx_1 : Gx_1;
wire [9:0] abs_Gx_2 = (Gx_2 < 0) ? -Gx_2 : Gx_2;
wire [9:0] abs_Gx_3 = (Gx_3 < 0) ? -Gx_3 : Gx_3;
wire [9:0] abs_Gy_0 = (Gy_0 < 0) ? -Gy_0 : Gy_0;
wire [9:0] abs_Gy_1 = (Gy_1 < 0) ? -Gy_1 : Gy_1;
wire [9:0] abs_Gy_2 = (Gy_2 < 0) ? -Gy_2 : Gy_2;
wire [9:0] abs_Gy_3 = (Gy_3 < 0) ? -Gy_3 : Gy_3;
// Sum the absolute values to get the gradient magnitude
wire [7:0] sum_0 =  (abs_Gx_0 + abs_Gy_0 < {2'd0, s_sobelTresholdReg}) ?
    8'd127 : ((comparisonBuffer[7]==comparisonBuffer[6])?  8'd0 : 8'd255) ;
wire [7:0] sum_1 =  (abs_Gx_1 + abs_Gy_1 < {2'd0, s_sobelTresholdReg}) ?
    8'd127 : ((comparisonBuffer[15]==comparisonBuffer[14])?  8'd0 : 8'd255) ;
wire [7:0] sum_2 =  (abs_Gx_2 + abs_Gy_2 < {2'd0, s_sobelTresholdReg}) ?
    8'd127 : ((comparisonBuffer[23]==comparisonBuffer[22])?  8'd0 : 8'd255) ;
wire [7:0] sum_3 =  (abs_Gx_3 + abs_Gy_3 < {2'd0, s_sobelTresholdReg}) ?
    8'd127 : ((comparisonBuffer[31]==comparisonBuffer[30])?  8'd0 : 8'd255) ;
// Combine the results into the output bus
assign s_sobelDataIn = {sum_3, sum_2, sum_1, sum_0};
endmodule
```

Code Block 5: Full code of the DMA + Sobel + Movement Detection module, saved in modules/ramdmaci/verilog/ramdmaci_sobel_movement_detection.

```
#include <stdio.h>
#include <ov7670.h>
#include <swap.h>
#include <vga.h>
// define profiling flags : decomment one to enable a specific profiling
// #define FULL_PROFILING                    // total cycles per frame
// #define MOVEMENT_DETECTION_PROFILING      // cycles for moving
    additional data required for movement detection
// #define SOBEL_PROFILING                   // cycles for moving  sobel
    input data and performing sobel
// #define CI_MEMORY_TO_OUTPUT_PROFILING     // cycles for moving the
    output values back to the output vector
//  #define INITIALIZATION_PROFILING     // cycles for reading the image
    and iniytializing for the first tile
// constants for the dma transfers
const uint32_t writeBusStartAddress = 0x00000600;
const uint32_t writeMemoryStartAddress = 0x00000A00;
const uint32_t writeControlRegister = 0x00001600;
const uint32_t readStatusRegister = 0x00001400;
const uint32_t writeBlockSize = 0x00000E00;
const uint32_t writeBurstSize = 0x00001200;
const uint32_t writeSingleWord = 0x00000200;
const uint32_t writeSobelTreshold = 0x00001A00;
//constants for image tiling
const uint32_t block_size = 17*14;
const uint32_t burst_size = 16;
const uint32_t block_size_output = 16*12;
const uint32_t burst_size_output = 15;
const uint32_t skip_line = 640*11+64;
```

```
//constants for double buffering
const uint32_t buff1 = 0;
const uint32_t buff2 = 238;
//sobel treshold : the lower, the more sensible (0-255)
const uint32_t sobel_treshold=90;
int main () {
  volatile uint8_t grayscale[640*480];
  volatile uint8_t sobelImage[640*480];
  volatile uint8_t movement[640*480];
  volatile uint32_t  cycles,stall,idle;
  volatile unsigned int *vga = (unsigned int *) 0X50000020;
  uint32_t result;
  uint32_t pixel1, pixel2;
  uint32_t buffer, lastrow;
  uint32_t read_addr,write_addr;
  camParameters camParams;
  vga_clear();
  printf("Initialising camera (this takes up to 3 seconds)!\n" );
  camParams = initOv7670(VGA);
  printf("Done!\n" );
  printf("NrOfPixels : %d\n", camParams.nrOfPixelsPerLine );
  result = (camParams.nrOfPixelsPerLine <= 320) ?
     camParams.nrOfPixelsPerLine | 0x80000000 : camParams.nrOfPixelsPerLine;
  vga[0] = swap_u32(result);
  printf("NrOfLines  : %d\n", camParams.nrOfLinesPerImage );
  result =  (camParams.nrOfLinesPerImage <= 240) ?
     camParams.nrOfLinesPerImage | 0x80000000 : camParams.nrOfLinesPerImage;
  vga[1] = swap_u32(result);
  printf("PCLK (kHz) : %d\n", camParams.pixelClockInkHz );
  printf("FPS        : %d\n", camParams.framesPerSecond );
  uint32_t grayPixels;
  vga[2] = swap_u32(2);
  vga[3] = swap_u32((uint32_t) &sobelImage[0]);
  // initialize all values to 127 (gray) for avoiding movement detection in
     the first frame
  for(int i=0; i<640*480; i++){
    sobelImage[i]=127;
  }
  asm volatile ("l.nios_rrr
     r0,%[in1],%[in2],0x14"::[in1]"r"(writeSobelTreshold),[in2]"r"(sobel_treshold));
  while(1){
        #ifdef FULL_PROFILING
          asm volatile ("l.nios_rrr r0,r0,%[in2],0xC"::[in2]"r"(7)); //
             start profiling
        #endif
        #ifdef INITIALIZATION_PROFILING
          asm volatile ("l.nios_rrr r0,r0,%[in2],0xC"::[in2]"r"(7)); //
             start profiling
        #endif
          uint32_t gray = (uint32_t ) &grayscale[0];
          uint32_t sobel = (uint32_t ) &sobelImage[641];
          takeSingleImageBlocking(gray);
          //transfer first image block to ci memory
          asm volatile ("l.nios_rrr
             r0,%[in1],%[in2],0x14"::[in1]"r"(writeMemoryStartAddress),[in2]"r"(buff2)
          asm volatile ("l.nios_rrr
```

```
            r0,%[in1],%[in2],0x14"::[in1]"r"(writeBusStartAddress),[in2]"r"(gray));
        asm volatile ("l.nios_rrr
            r0,%[in1],%[in2],0X14"::[in1]"r"(writeBlockSize),[in2]"r"(block_size));
        asm volatile ("l.nios_rrr
            r0,%[in1],%[in2],0x14"::[in1]"r"(writeBurstSize),[in2]"r"(burst_size));
        asm volatile ("l.nios_rrr
            r0,%[in1],%[in2],0x14"::[in1]"r"(writeControlRegister),[in2]"r"(1));
        buffer=0;
        while(1){
            asm volatile ("l.nios_rrr
               %[out1],%[in1],r0,0x14":[out1]"=r"(result):[in1]"r"(readStatusRegiste
               // read status register
            if(result==0) break;
        }
    #ifdef INITIALIZATION_PROFILING
            asm volatile ("l.nios_rrr
               r0,r0,%[in2],0xC"::[in2]"r"(7<<4)); //disable counters
    #endif
        for (int i = 1; i <= 400; i++) {
            lastrow=(i%10);
            //move the previous values of the sobel buffer to be
               compared with the ones that have to be computed
          #ifdef MOVEMENT_DETECTION_PROFILING
             asm volatile ("l.nios_rrr
                r0,r0,%[in2],0xC"::[in2]"r"(7)); // start profiling
          #endif
            asm volatile ("l.nios_rrr
               r0,%[in1],%[in2],0x14"::[in1]"r"(writeBusStartAddress),[in2]"r"(s
            asm volatile ("l.nios_rrr
               r0,%[in1],%[in2],0X14"::[in1]"r"(writeBlockSize),[in2]"r"(block_s
            asm volatile ("l.nios_rrr
               r0,%[in1],%[in2],0x14"::[in1]"r"(writeBurstSize),[in2]"r"(burst_s
            asm volatile ("l.nios_rrr
               r0,%[in1],%[in2],0x14"::[in1]"r"(writeMemoryStartAddress),[in2]"r
            asm volatile ("l.nios_rrr
               r0,%[in1],%[in2],0x14"::[in1]"r"(writeControlRegister),[in2]"r"(1
            while(1){
                asm volatile ("l.nios_rrr
                   %[out1],%[in1],r0,0x14":[out1]"=r"(result):[in1]"r"(readStatu
                   // read status register
                if(result==0) break;
            }
          #ifdef MOVEMENT_DETECTION_PROFILING
            asm volatile ("l.nios_rrr
               r0,r0,%[in2],0xC"::[in2]"r"(7<<4)); //disable counters
          #endif
          #ifdef SOBEL_PROFILING
              asm volatile ("l.nios_rrr
                 r0,r0,%[in2],0xC"::[in2]"r"(7)); // start profiling
          #endif
            if(i<=399){ // transfer a grayscale image block to one
               buffer (not for the last iteration)
                asm volatile ("l.nios_rrr
                   r0,%[in1],%[in2],0x14"::[in1]"r"(writeMemoryStartAddress),[
                if(lastrow==0){
                    gray+=skip_line; // skip 12 pixel lines for the
```

```
                        last block of a row
                }
                else{
                    gray+=64; // increment bus start address
                }
                asm volatile ("l.nios_rrr
                   r0,%[in1],%[in2],0X14"::[in1]"r"(writeBlockSize),[in2]"r"(b
                asm volatile ("l.nios_rrr
                   r0,%[in1],%[in2],0x14"::[in1]"r"(writeBurstSize),[in2]"r"(b
                asm volatile ("l.nios_rrr
                   r0,%[in1],%[in2],0x14"::[in1]"r"(writeBusStartAddress),[in2
                asm volatile ("l.nios_rrr
                   r0,%[in1],%[in2],0x14"::[in1]"r"(writeControlRegister),[in2
            }
            asm volatile ("l.nios_rrr r0,r0,r0,0x15"); //start sobel
               + movement detection
            while(1){ //wait for dma transfer to finish
                asm volatile ("l.nios_rrr
                   %[out1],%[in1],r0,0x14":[out1]"=r"(result):[in1]"r"(readStatu
                   // read status register
                if(result==0) break;
            }
            while(1){ //wait for sobel to finish
                asm volatile ("l.nios_rrr
                   %[out1],%[in1],%[in2],0x14":[out1]"=r"(result):[in1]"r"(0),[i
                   // read status register
                if(result==0) break;
            }
            #ifdef SOBEL_PROFILING
             asm volatile ("l.nios_rrr
                r0,r0,%[in2],0xC"::[in2]"r"(7<<4)); //disable counters
           #endif
           #ifdef CI_MEMORY_TO_OUTPUT_PROFILING
             asm volatile ("l.nios_rrr
                r0,r0,%[in2],0xC"::[in2]"r"(7)); // start profiling
          #endif
            // transfer computed values (sobel+movement detection
               outputs) from ci memory to second buffer
            asm volatile ("l.nios_rrr
               r0,%[in1],%[in2],0x14"::[in1]"r"(writeMemoryStartAddress),[in2]"r
            asm volatile ("l.nios_rrr
               r0,%[in1],%[in2],0X14"::[in1]"r"(writeBlockSize),[in2]"r"(block_s
            asm volatile ("l.nios_rrr
               r0,%[in1],%[in2],0x14"::[in1]"r"(writeBurstSize),[in2]"r"(burst_s
            asm volatile ("l.nios_rrr
               r0,%[in1],%[in2],0x14"::[in1]"r"(writeBusStartAddress),[in2]"r"(s
            if(lastrow==0){
                sobel+=skip_line;  // skip 12 pixel lines for the
                   last block of a row
            }
            else{
                sobel+=64; // increment bus start address
            }
            asm volatile ("l.nios_rrr
               r0,%[in1],%[in2],0x14"::[in1]"r"(writeControlRegister),[in2]"r"(2
            while(1){ // wait for dma transfer to finish
```

```
                    asm volatile ("l.nios_rrr
                       %[out1],%[in1],r0,0x14":[out1]"=r"(result):[in1]"r"(readStatu
                       // read status register
                    if(result==0) break;
                }
              #ifdef CI_MEMORY_TO_OUTPUT_PROFILING
                 asm volatile ("l.nios_rrr
                    r0,r0,%[in2],0xC"::[in2]"r"(7<<4)); //disable counters
              #endif
                buffer=!buffer; //switch buffer for the next iteration
            }
          // read profiling results
        #ifdef FULL_PROFILING
          asm volatile ("l.nios_rrr
             %[out1],r0,%[in2],0xC":[out1]"=r"(cycles):[in2]"r"(1<<8|7<<4));
          asm volatile ("l.nios_rrr
             %[out1],%[in1],%[in2],0xC":[out1]"=r"(stall):[in1]"r"(1),[in2]"r"(1<<9));
          asm volatile ("l.nios_rrr
             %[out1],%[in1],%[in2],0xC":[out1]"=r"(idle):[in1]"r"(2),[in2]"r"(1<<10));
          printf("nrOfCycles: %d %d %d\n", cycles, stall, idle);
        #endif
        #ifdef MOVEMENT_DETECTION_PROFILING
          asm volatile ("l.nios_rrr
             %[out1],r0,%[in2],0xC":[out1]"=r"(cycles):[in2]"r"(1<<8));
          asm volatile ("l.nios_rrr
             %[out1],%[in1],%[in2],0xC":[out1]"=r"(stall):[in1]"r"(1),[in2]"r"(1<<9));
          asm volatile ("l.nios_rrr
             %[out1],%[in1],%[in2],0xC":[out1]"=r"(idle):[in1]"r"(2),[in2]"r"(1<<10));
          printf("nrOfCycles for movement detection data movement overhead:
             %d %d %d\n", cycles, stall, idle);
        #endif
        #ifdef SOBEL_PROFILING
          asm volatile ("l.nios_rrr
             %[out1],r0,%[in2],0xC":[out1]"=r"(cycles):[in2]"r"(1<<8));
          asm volatile ("l.nios_rrr
             %[out1],%[in1],%[in2],0xC":[out1]"=r"(stall):[in1]"r"(1),[in2]"r"(1<<9));
          asm volatile ("l.nios_rrr
             %[out1],%[in1],%[in2],0xC":[out1]"=r"(idle):[in1]"r"(2),[in2]"r"(1<<10));
          printf("nrOfCycles for Sobel loading and filtering: %d %d %d\n",
             cycles, stall, idle);
        #endif
        #ifdef CI_MEMORY_TO_OUTPUT_PROFILING
          asm volatile ("l.nios_rrr
             %[out1],r0,%[in2],0xC":[out1]"=r"(cycles):[in2]"r"(1<<8|7<<4));
          asm volatile ("l.nios_rrr
             %[out1],%[in1],%[in2],0xC":[out1]"=r"(stall):[in1]"r"(1),[in2]"r"(1<<9));
          asm volatile ("l.nios_rrr
             %[out1],%[in1],%[in2],0xC":[out1]"=r"(idle):[in1]"r"(2),[in2]"r"(1<<10));
          printf("nrOfCycles for moving data from ci memory to sobel vector:
             %d %d %d\n", cycles, stall, idle);
        #endif
        #ifdef INITIALIZATION_PROFILING
          asm volatile ("l.nios_rrr
             %[out1],r0,%[in2],0xC":[out1]"=r"(cycles):[in2]"r"(1<<8|7<<4));
          asm volatile ("l.nios_rrr
             %[out1],%[in1],%[in2],0xC":[out1]"=r"(stall):[in1]"r"(1),[in2]"r"(1<<9));
```

```
        asm volatile ("l.nios_rrr
            %[out1],%[in1],%[in2],0xC":[out1]"=r"(idle):[in1]"r"(2),[in2]"r"(1<<10));
        printf("nrOfCycles for initializing the frame: %d %d %d\n", cycles,
            stall, idle);
      #endif
  }
}
```

Code Block 6: Full C program for performing the accelerated Sobel filtering with movement detection

References

[1] *Gecko4Education EPFL edition*. https://gecko-wiki.ti.bfh.ch/gecko4education_epfl:start.

[2] *Altera Cyclone IV FPGA*. https://www.intel.com/content/www/us/en/products/details/fpga/cyclone/iv.html.

[3] *OpenRisc OR1200 processor*. https://github.com/openrisc/or1200.

[4] *Olimex OV7670 camera*. https://www.olimex.com/Products/Components/Camera/CAMERA-OV7670/.

[5] N. Vasanthavada N. Kanopoulos and R. L. Baker. “"Design of an image edge detection filter using the Sobel operator"”. In: *IEEE Journal of Solid-State Circuits*. Ed. by N. Vasanthavada N. Kanopoulos and R. L. Baker. IEEE, 1988. Chap. vol. 23, no. 2, pp. 358–367. ISBN: doi: 10.1109/4.996. URL: https://ieeexplore.ieee.org/document/996.

[6] Sebastian A. “Memory devices and applications for in-memory computing”. In: *Nature nanotechnologies* (Mar. 2020), pp. 529–544.

[7] Verma N et al. “In-Memory Computing: Advances and Prospect”. In: *IEEE Solid-State Circuits Magazine* 11.3 (2019), pp. 42–55.

[8] *Convolutional Neural Network with Numpy (Fast))*. https://hackmd.io/@machine-learning/blog-post-cnnumpy-fast. 2022.