Ecole Polytechnique Fédérale de Lausanne

Embedded System Design

CS-476

Motion Detection

Authors:

Vincent Roduit

Filippo Santiago Quadri

June 9, 2024

Abstract

Motion detection is widely used in security devices such as surveillance cameras, as well as in various other applications. While the specific requirements of these devices can vary, the fundamental principle remains the same: detecting whether an object has moved by computing the edges of an image and comparing two consecutive frames. The purpose of this project is to propose solutions that can accelerate this process, given the constraints of the board and the project's specifications. Techniques such as Direct Memory Access (DMA) and custom instructions have been utilized. This project has been implemented using C and Verilog.

Keywords— Verilog, C, Custom Instruction, Direct Memory Acess, Sobel Edge Detection

Contents

1 Introduction

The purpose of this project is to build a system capable of detecting movement from a camera. This system can be envisioned for implementation in a surveillance camera. Although the basic concept of motion detection remains the same across different devices, the specifications can vary significantly. For example, a surveillance camera might need to capture one frame per second, but this rate would be inadequate for detecting fast-moving objects under a microscope. Therefore, a list of specifications needs to be created to clearly understand the objectives.

The main objective of this project is to develop a motion detection system that fulfils the specifications. This system utilizes a combination of C programming and Verilog and is implemented on an FPGA board, namely the gecko4education [Gec].
The fundamental principle of motion detection is very simple. This involves analyzing two sequential frames given by a camera and identifying the changes in edges. Edges are significant change of intensity on a given image. These edges can easily be found by computing the convolution with a specific matrix.

There are two main types of edge detection: the maximum of the first derivative (using operators like Sobel or Prewitt) and zero crossing of the second derivative (such as Marr–Hildreth or Laplacian of Gaussian). The second method is less sensitive to noise due to prior filtering but requires floating-point convolution, making it less suitable for real-time applications. The Sobel operator, on the other hand, delivers excellent results for the "maximum of the first derivative" method and is computationally simple. This is why the Sobel operator was chosen for implementing motion detection.

This project aims to accelerate the process in order to meet the specifications. In fact, it can be seen in following sections that computing the process only in software is experienced to be very slow. Implementing key functions in hardware using Verilog and optimizing higher-level control tasks in C can achieve significant improvements regarding the speed of the system. Techniques such as Direct Memory Access (DMA) and custom instructions are employed to enhance data transfer rates and computational performance.

This document presents the design and implementation of the motion detection system, focusing on identifying bottlenecks and proposing solutions to address them. The report concludes with a discussion section and a summary, which will encapsulate the work done and explore potential future developments. The complete code generated is available in the git repository at this link.

2 Specifications

As stated in Section 1, the implementation can vary depending on the needs for the same behavior. In this context, specifications need to be established to provide guidelines for the project. Since the goal of this project is to develop a motion detection algorithm for a home surveillance camera, the following specifications have been chosen:

1. **The system should have at least 1 or 2 frames per second**: Higher resolutions are unnecessary, as even if intruders move quickly, one or two frames per second are sufficient to detect their movements.
2. **The system should have low power consumption**: The camera should operate for several months, or even years, without requiring a battery change. Therefore, the system should use minimal hardware (only what is necessary to meet the constraints) to avoid unnecessary power consumption and higher production costs. For this project, the CPU is assumed to be a low-energy embedded CPU, which is optimal for battery-powered applications. This makes it crucial to have an efficient software component, provided the constraints are met.

Based on these requirements, it is clear that a compromise must be found between efficiency and power consumption. Increasing the hardware components can accelerate the system but will also increase power consumption. Moreover, extensive hardware can significantly raise the cost of the system. A balance between these parameters must be set to optimize the overall performance.

3 Design Process

The development of this project has been done in an iterative fashion. It consists of these differents steps:

- Development of a new project version
- Inspection of the requirements
- Identification of the bottlenecks
- Putting new solutions into practice

These four steps are repeated until the specifications are met. This section aims to present these different processing steps in the same chronological order in which they were developed.

3.1 Software implementation

The initial phase of the process involves the complete implementation of the system in software. This serves as the foundational step in the development process, enabling the identification of initial bottlenecks and the possibility to check if the project is feasible (or if, for example, the algorithms and operators that are used need to be changed). By running the code on the board, the following behavior appears: one frame every approximately 5 seconds to compute the edges and do the comparison with the previous frame. This is clearly not sufficient, as described by the Section 2.

The code presented in Listing 1 shows this implementation. By inspecting this code, it is obvious that this solution is far from an optimal one. First, the pixels are not efficiently accessed. In fact, a pixel can be used in nine different convolutions. The proposed code don't use this fact and load the nine pixels every time. Even worse, the pixels are loaded form the DRAM. Thus, every time we want to access a new pixel, we need to access the bus. This explain the slow behavior of the system. Moreover, computing the Convolution in software (line 8 and 9) is also time consuming and constitute another bottleneck of the system.

Listing 1: C implementation of Sobel Algorithm

```
1  for (int line = 1; line < height - 1; line++) {
2      for (int pixel = 1; pixel < width - 1; pixel++) {
3        valueX = valueY = valueD = 0;
4        for (int dx = -1; dx < 2; dx++) {
5          for (int dy = -1; dy < 2; dy++) {
6            uint32_t index = ((line+dy)*width)+dx+pixel;
7            int32_t gray = grayscale[index];
8            valueX += gray*gx_array[dy+1][dx+1];
9            valueY += gray*gy_array[dy+1][dx+1];
10         }
11       }
12       result = (valueX < 0) ? -valueX : valueX;
13       result += (valueY < 0) ? -valueY : valueY;
14       sobelResult[line*width+pixel] = (result > threshold) ? 0xFF : 0;
15     }
16   }
```

3.2 Sobel as a Custom Instruction

The subsequent stage in the process involves conducting the Sobel convolution in hardware rather than performing it in C. This adjustment will decrease the time required by circumventing the CPU's multiplication tasks, which can be efficiently handled in hardware using shifts and additions. However, it will not entirely resolve the issue, as the primary challenge stems from memory access and bus utilization.

The custom instruction is used with the unique identifier 0xC (decimal 12). This instruction loads the nine pixels and then computes a Sobel convolution and returns the result. This implementation is summarised in the Listing 2. This solution is only a minor improvement as it does not improve the efficiency of pixel loading or bus usage. These aspects will be improved in the next parts.

Listing 2: Sobel Algorithm using Custom instruction

```
1     for (int line = 1; line < height - 1; line++) {
2     for (int pixel = 1; pixel < width - 1; pixel++) {
3       uint16_t matrix[9];
4       int cnt = 0;
5       for (int dx = -1; dx < 2; dx++) {
6         for (int dy = -1; dy < 2; dy++) {
7           uint32_t index = ((line+dx)*width)+dy+pixel;
8           matrix[cnt] = grayscale[index];
9           uint32_t gray = grayscale[index];
10          cnt += 1;
11        }
12      }
13      tmp_sobel_result = 0;
14      valueA = (matrix[3] << 24) | (matrix[2] << 16) | (matrix[1] << 8) |
            matrix[0];
15      valueB = 1;
16      asm volatile ("l.nios_rrr r0,%[in1],%[in2],0xC"::[in1]"r"(valueA),[
            in2]"r"(valueB));
17      valueA = (matrix[7] << 24) | (matrix[6] << 16) | (matrix[5] << 8) |
            matrix[4];
18      valueB = 2 | (matrix[8] << 8) | (threshold << 16);
19      asm volatile ("l.nios_rrr %[out1],%[in1],%[in2],0xC":[out1]"=r"(
            tmp_sobel_result):[in1]"r"(valueA),[in2]"r"(valueB));
20      sobelResult[line*width+pixel] = tmp_sobel_result > threshold ? 0xff
            : 0;
21    }
22  }
```

3.3 Direct Memory Access

Now that the Sobel convolution has been computed in hardware, the main problem to be solved is bus occupancy, which is closely related to pixel loading efficiency. The first approach is not perfect and consists of loading three lines at a time using a Direct Access Memory controller. Obviously, this first solution solves half of the problem, since one line is reloaded three times. Nevertheless, it is a first operational attempt.

This solution follows these steps:

- Load 3 lines with the help of the DMA.
- Read 2 blocks as presented in Figure 1 and store them into an array, named SobelStorage.
- Compute the four convolutions, one at a time and directly send it to the array SobelImage (without the use of the DMA), responsible of displaying the result through the VGA.

However, this solution was found to be inefficient. Although accessing data through the DMA improves speed, storing them directly into an array creates the same bottleneck as before. Additionally, computing the convolution requires accessing and isolating each pixel (noting that when reading in the CI Memory, we access 4 pixels at a time), making this process inefficient. This issue motivates the next significant improvement: computing four convolutions at a time and revising the data access/storing method.

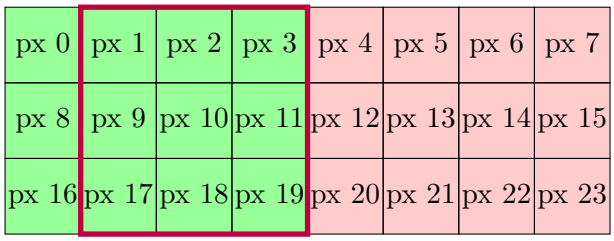
3.4 Four Convolutions

As stated before, only using the DMA for data transfer is inefficient. The hardware implementation needs to be updated, as well as the data access methods. Figure 1 and Table 1 help to understand the problem and support the development choice of performing four convolutions at a time. Because the camera converts pixels to grayscale on 8 bits and then stores them in 32-bit format into memory, each address contains four pixels. This means that loading an address is equivalent to reading four pixels. In order to use this information efficiently, it was decided to update the Verilog implementation to load and compute four convolutions at a time. To achieve this, three lines consisting of two consecutive addresses are read and sent to the *Sobel Edge Detection Custom Instruction*. The consecutive addresses are represented by the different colors in Figure 1. The four convolutions are then stored in the SRAM (CI Memory) associated to the DMA. Finally, when a full line has been calculated, the result is sent (via a burst) to the SobelImage array, which is used to display the result via the VGA. This solution has proved to be a significant improvement in terms of system speed, resulting in approximately one frame per second.

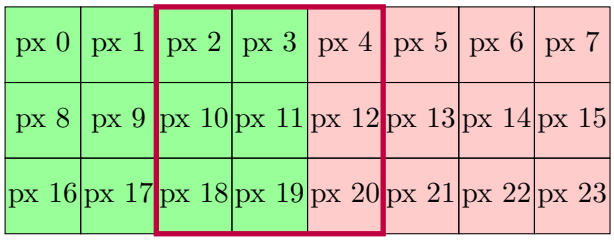
Adress	Content
$0x00	{px3,px2,px1,px0}
$0x01	{px7,px6,px5,px4}
...	...

Table 1: Pixel Storage

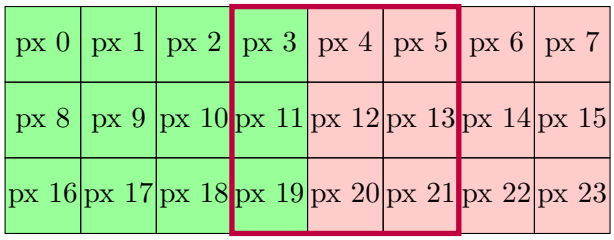
(a) Convolution 1



(b) Convolution 2



(c) Convolution 3



(d) Convolution 4

Figure 1: Structure of the pixels for Sobel Algorithm

3.5 Forward/Reverse Data Loading and Circular Buffer

Another improvement that was already discussed in previous sections is the loading of the pixels. Indeed, a *Forward/Reverse* computing and a *Circular Buffer* memory reorganization can be be implemented. The first one refers to what is called a block in this report. A block corresponds to a 3-by-4 matrix (4 pixels per line for 3 consecutive lines, pixels green or red in Fig. 1). In order to achieve this, the Verilog code and the software are both updated. The software alternates the loading of the pixels sent to the custom instruction and the convolution can be computed in the *forward* mode (like in Sec. 3.4) or in the *reverse mode*. This new way of loading datas removes unnecessary transfers. The bit 16 of the register ValueB is responsible of controlling the reverse mode of the module. When $ValueB[16] = 0$, convolutions are calculated according to pixel structure presented in Figure 1. The reverse order can be found in Appendix, Figure 2.

Furthermore, the lines are read efficiently through a Circular Buffer. In this process, the oldest line in memory (line that is no more needed for computing purposes) is replaced by the new line and the 3 other lines are read in the correct order to compute the *Edge Detection*. Figure 3, available in the Appendix (Section 7), illustrates the different steps of loading. In this figure, green represents lines loaded during the step, while red indicates old lines. Steps two to five are repeated until the

entire height of the image is computed. By inspecting this figure, it is evident that indexes must be properly managed to preserve the pixel order described in Figures 1 and 2.

The bottleneck of the system now lies on the comparison between two successive frames. In fact, this part suffers form the same behavior as already stated multiple times in this report (see Section 3.1), namely the bus occupancy. No DMA was used for this process, causing latency in the system. The next step of the process will solve this issue.

3.6 DMA use for frame comparison

This part aims to solve the main bottlenceks stated in the previous section. A DMA is used to load fragment of the old frame, the new frame, as well as the grayscale image. Then comparison is done and sent back through the DMA. This enhancement has demonstrated some excellent result, by dividing by a factor of two the time per frame (see Section 4 for more about results). The stall cycles are also dramatically reduced. The main factor slowing down the system is that the comparison is done on eight bit which is usless. As the edge detection return a binary value (0 or 255), which is encoded on 8 bits, only one bit is sufficient to capture this information.

3.7 Edge comparison on 1-bit

The final step in the process is to transform the size of each pixel from 8 bits to 1 bit. After calculating the Sobel convolution and comparing it to the threshold, the 8 bits only capture a binary value, black or white (or 0 and 255 in decimal format). There is no need to use eight bits at this stage. In this way, each pixel is downconverted to just one bit. Then, every 8 steps (i.e. every 32 turns), the result of the custom instruction is retrieved and stored in memory. The size of an image is then scaled from $480 * 640 * 8 = 2.46Mbits$ to $480 * 640 = 0.3Mbits = \frac{\text{Original Size}}{8}$, which is a significant improvement in size. Also, the comparison is done at 1 bit instead of 8 bits as before. This not only improves speed, but also memory efficiency, reducing the size by a factor of eight. This last improvement demonstrates good performance, reducing the stall cycles and increasing the frame per second.

4 Comparison Between Methods

This section aims to compare the different steps in order to show improvement or deterioration of each method. The analysis focuses on the two aspects detailed in Section 2.

4.1 Hardware size

Section 3 focuses on the performance of the system in terms of speed. However, as mentioned in section 2, an important aspect of the specification is the size of the hardware. Indeed, increasing the size of the system will increase both the price and the power consumption. A balance must therefore be struck between hardware and software. The table 2 summarises the percentage of resources used by each solution.

Method	Percentage
Software	46%
Sobel as CI	46%
DMA Alone	41%
4 Conv	41%
F/W Reads & CB	39%
DMA for Comparison	43%
1-bit Comparison	40%

Table 2: Percentage of device resources used

Looking at the table 2, it is obvious that the hardware improvement has no real impact on the system. In fact, the approximate size given by the synthesiser shows that the size is almost constant and does not explode with the implemented solutions. Considering this aspect, the faster solution can be chosen without compromising on power consumption.

4.2 Speed

Speed is the second most important factor. Since bus occupancy dramatically reduces speed, the solutions presented in section 3 tend to improve this component. Table 3 summarises the results obtained. Looking at this table, we can see that for the software implementation, the CPU waits for resources for about 80% of the time. Conversely, the Reverse-Forward reading tends to reduce this value to 0.68%, which can be explained by the reasons described in section 3. It can also be seen that 4 Conv and Reverse-Forward Reading both meet the requirements given in the specifications section. In order to facilitate the reading, the Table 3 is constructed as follow: the software is taken as reference for CPU Cycles, Stall Cycles and Bus idle columns. Then, for instance it can be observe that the DMA alone reduces the number of cycles by a factor of two. For reference, the number of cycles founded for the software are:

- CPU Cycles: 4.54×10^8
- Stall Cycles: 3.6×10^8
- Bus Idle: 1.78×10^8

Based on these results, the approach developed in this report can be approved. It showed an improvement in speed while keeping the hardware part reasonably small. Therefore, the last solution can be selected for implementation in the concrete system.

Method	CPU Cycles	Stall Cycles	Bus idle	SPF	FPS
Software	1	1	1	$6.11s$	$0.16fps$
Sobel CI	1.13	1.19	1.08	6.92	$0.14fps$
DMA alone	0.51	0.53	0.48	$3.12s$	$0.32fps$
4 Conv	0.15	0.13	0.15	$0.93s$	$1.08fps$
F/R Reads & CB	0.15	0.13	0.16	$0.93s$	$1.08fps$
DMA for Comparison	6.51×10^{-2}	1.96×10^{-2}	8.71×10^{-2}	$0.4s$	$2.51fps$
1-bit Comparison	5.44×10^{-2}	5.38×10^{-3}	8.31×10^{-2}	0.3	$3fps$

Table 3: Summary of method's cycles

5 Discussion

In projects of this nature, the avenues for improvement seem almost limitless. There's always the prospect of finding optimizations that are more efficient, faster, or more power-efficient. Yet, as often said in engineering circles, *There is no free lunch.* This quote underscores the reality that any optimization pursued is likely to incur losses elsewhere. Thus, it perpetually remains a matter of trade-offs.

In the realm of embedded systems, the true challenge lies in discerning when to halt the pursuit, that is, in identifying the optimal trade-off to achieve desired performance while minimizing all associated losses such as consumption, circuit size, and cost.

Applied to this scenario, for instance, the project could have continued exploring further avenues, perhaps delving into additional parallel convolutions or investigating alternative methods like LoG or image hashing [FJZ+17]. However, such pursuits may not necessarily be justified. The solutions already explored have successfully met the varied requirements, striking a commendable balance between speed and hardware footprint.

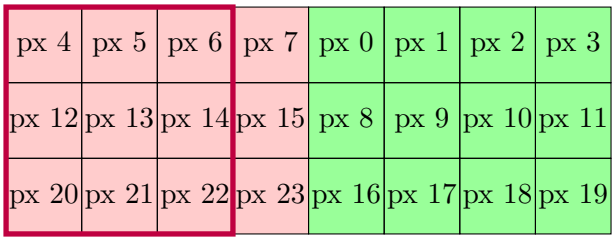
6 Summary

As explained in section 3, numerous solutions were explored, each aimed at addressing the bottlenecks stemming from the preceding solution.

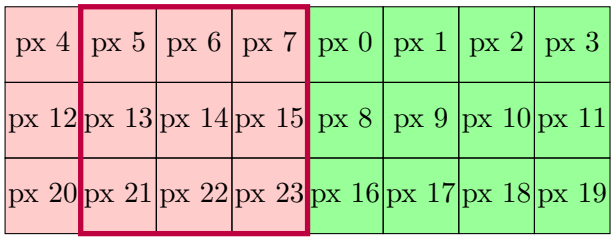
An initial all-software version proved functional but fell short of satisfying the frame rate constraint. Subsequently, the implementation of a custom instruction capable of computing Sobel alleviated some of the work from the CPU, yet failed to resolve the bus access problem (so the frame rate constraint). Consequently, integration of a DMA was pursued. In its initial iteration, the DMA was employed solely for loading pixel buffers, thereby obviating the need for accessing DRAM each time. While this reduced bus occupancy, convolution remained highly inefficient.

To reduce this inefficiency, four convolutions were performed simultaneously, accompanied by optimisations in memory access and management (detailed in section 3.5). As a final adjustment, the size of a pixel Sobel was reduced to one bit. The transfer of a binary value using an entire integer byte was identified as a source of inefficiency. This change resulted in an overall performance improvement, increasing speed and reducing the number of cycles required.

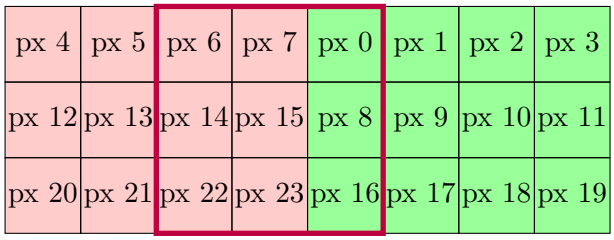
7 Appendix



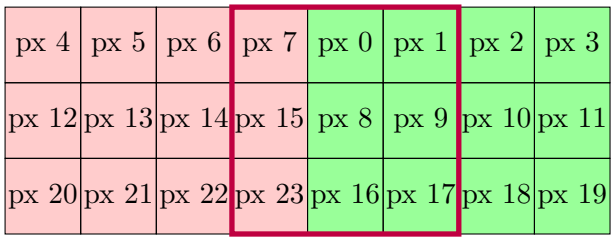
(a) Convolution 1



(b) Convolution 2

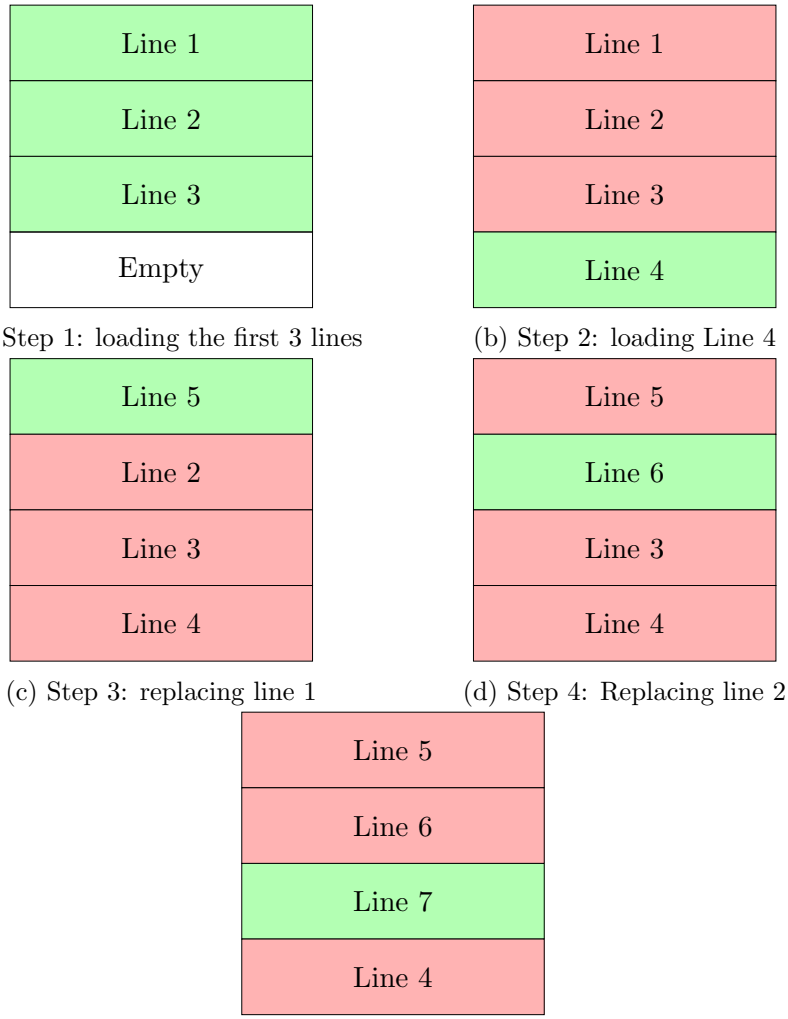


(c) Convolution 3



(d) Convolution 4

Figure 2: Structure of the pixels in reverse mode



(a) Step 1: loading the first 3 lines

(b) Step 2: loading Line 4

(c) Step 3: replacing line 1

(d) Step 4: Replacing line 2

(e) Step 5: Replacing line 3

Figure 3: Steps of the different line loading

References

[FJZ+17] Meng Fei, Zhixin Ju, Xin Zhen, et al. "Real-time visual tracking based on improved perceptual hashing". In: *Multimedia Tools and Applications* 76 (2017), pp. 4617–4634. DOI: 10.1007/s11042-016-3723-5.

[Gec] Gecko4Education. *Gecko4Education: EPFL*. https://gecko-wiki.ti.bfh.ch/gecko4education_epfl:start. Accessed: 2024-06-03.