École Polytechnique Fédérale de Lausanne

Embedded System Design

CS-476

Movement Detection with Sobel Operator and Perceptual Hashing

Authors:
Giorgio Ajmone
Alessandro Cardinale

June 9, 2024

Contents

1 Introduction

In modern computing, especially for timing-constrained applications, software operations often fall short, necessitating hardware solutions.

The goal of this project is to implement a real-time movement detection mechanism running on a Field Programmable Gate Array. The main contribution of this work consists of the hardware/software co-design of different accelerators to optimize the numerous phases involved in image processing.

In general, detecting movement is a challenging computation that requires different steps to process the images coming from a camera module and to output the result using a VGA interface. It consists of several phases that acquire and pre-process the image to make it suitable for the movement detection task.

As mentioned, the first steps to pre-process the input image usually involve image acquisition followed by the conversion from RGB encoding to grayscale values. The goal of this phase is to reduce the noise while representing the image in a more compressed way.

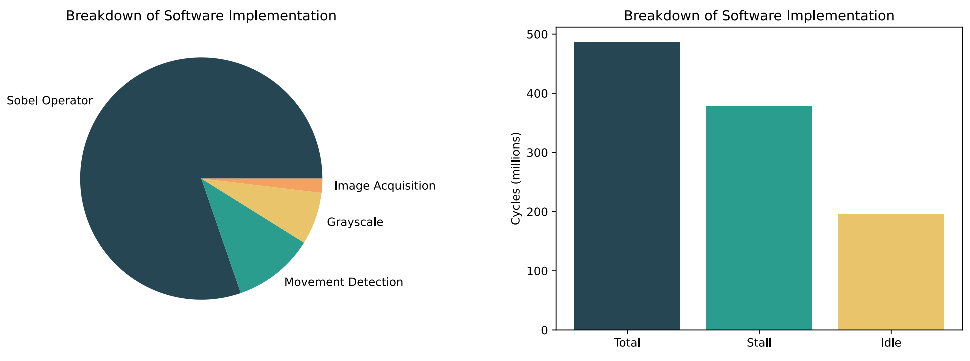
The second step, in this case the most critical phase, consists of the Sobel Operator as the algorithm for edge detection. This operator works by applying a pair of convolutional kernels, represented by 3x3 matrices, to an image to estimate the gradient of the image intensity at each pixel. These kernels detect changes in intensity in both the horizontal and vertical directions. The resulting gradient is computed as a combination of the two and it highlights areas where the intensity changes sharply, corresponding to edges within the image.

The last step is represented by the movement detection task on the pre-processed image. The standard way of detecting movement consists of a pixel-by-pixel comparison of two consecutive images. In addition, the final design proposes to leverage a particular class of mathematical functions, known as Locality-Sensitive Hashing functions or Perceptual hashing, to identify changes in the image in addition to the software-based comparison.

The final implementation leverages Verilog and C to enhance a OpenRISC processor running on an Altera Cyclone IV FPGA as part of a GECKO4Education board. Starting from a baseline system consisting of a camera module, a VGA interface and a processor, the proposed design includes a hardware accelerator to perform Sobel Operator and a dedicated hashing module. The software component of the project involves the algorithmic optimization of the pixel-by-pixel comparison in addition to several techniques to mask the costs of the different phases.

In the final part of the report, results are discussed and the improvements derived from hardware solutions are shown. The hardware-accelerated version exhibits a 28x improvement over a naive pure software-based implementation used as a baseline.

2 Profiling



(a) (b)

Figure 1: *Profiling of the naive software-based implementation.*

The initial phase of this project involves the development of a purely software-based version of the original concept. This version serves as the baseline for performance analysis, allowing us to evaluate the contributions of the different computational stages and identify potential bottlenecks. This initial analysis also provides insights into the theoretical performance limits achievable by the software implementation.

To evaluate the performance of the proposed solution, it is necessary to incorporate instrumentation capable of accurately measuring the performance of various functions. This is achieved through a combined approach involving both software and hardware tools. On the hardware side, performance counters are integrated to monitor various execution characteristics, including the total number of cycles, stall cycles, and idle cycles. On the software side, custom instructions are embedded within the code to control these performance counters. These instructions are responsible for starting, stopping, and reading the performance counters at strategic points in the code. This combination of hardware and software instrumentation allows for a detailed profiling of the execution phases, allowing us to identify performance-critical sections.

Figure 1 presents the results of the performance analysis conducted on the naive software-only implementation, considering a single iteration of the process. The analysis reveals that the application of the Sobel Operator is the most computationally intensive stage, accounting for approximately 80% of the total execution cycles. The second most demanding task is movement detection, which consumes around 10% of the total cycles. This stage is followed, in terms of computational costs, by the conversion from RGB to grayscale, which, while less demanding than the previous stages, still represents a notable portion of the computational load. Interestingly, the image capturing phase contributes the least to the total cycle count.

It is important to notice that, in the naive implementation, all the phases are sequential and blocking towards other phases. In this case, all the stages are exposed with respect to the total computation which means it is possible to identify a portion of the total number of cycles dedicated to each task without any overlap.

2.1 Image Acquisition

```c
void takeSingleImageBlocking(uint32_t framebuffer) {
  uint32_t result;
  asm volatile ("l.nios_rrr r0,%[in1],%[in2],0x7"::[in1]"r"(5)
   ,[in2]"r"(framebuffer));
  asm volatile ("l.nios_rrr r0,%[in1],%[in2],0x7"::[in1]"r"(6)
   ,[in2]"r"(2));
  do {
    asm volatile ("l.nios_rrc %[out1],%[in1],r0,0x7":[out1]"=r"
   (result):[in1]"r"(7));
  } while (result == 0);
}
```

Listing 1: Image Acquisition

The initial phase of each iteration involves transferring the image to be processed from the camera. This phase is critical yet occupies less than 2% of the total computation time dedicated to processing an image. The image from the camera is stored in a 640x480 array, where each pixel occupies a uint16_t. The transfer process is managed autonomously by the camera's Direct Memory Access (DMA) controller, with the CPU in the naive implementation merely waiting for the transfer to complete before proceeding to the next steps.

During this phase, the CPU remains idle, which, while having a limited impact on overall performance, represents an inefficiency that can be mitigated. Using the blocking version of the image acquisition requires the core to keep spinning on the flag that shows the status of the transfer without being able to perform any possible computation. As explained in the introduction, in the naive software implementation, the image acquisition process is completely exposed in terms of costs.

2.2 Grayscale Conversion

```c
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;
  }
}
```

Listing 2: Grayscale Conversion

The initial processing phase for each image, after acquisition, involves converting each pixel from the 16-bit RGB565 representation to the 8-bit grayscale representation. This phase accounts for approximately 7% of the total computation time. For each pixel, a series of bit-wise operations are performed to do the conversion. The result of this processing is stored in a 640x480 array of uint8_t.

Apart from compiler optimizations, each conversion requires the CPU to perform 6 bit-wise shift operations and 4 bit-wise and operations to extract and realign the bit encoding the RGB values. In addition, to convert to grayscale encoding, it is necessary to perform 3 multiplications, that while being optimizable by the compiler still account for a notable portion of the computation. Although most of the instructions are independent, when performed by the CPU they become sequential and blocking to each other.

2.3 Sobel Operator

```
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) ? 0
   xFF : 0;
    }
}
```

Listing 3: Edge Detection

The third phase of the process involves performing edge detection on the image that has been converted to grayscale. This step is accomplished by applying the Sobel operator, which consists of convoluting the image with two specific kernels representing Gaussian filters. These filters are designed to detect changes in intensity in the horizontal and vertical directions, gradients, respectively. The results of these convolutions are then combined using an approximation of the Euclidean distance to produce the final edge-detected image. This step is critical for identifying the edges within the image, which are essential for the subsequent movement detection phase.

As shown in the introduction, this task is the most computationally intensive part of the entire process, representing the largest single contributor to the total computation time. Specifically, the edge detection phase accounts for approximately 80% of the total computation, consuming more than 390 million cycles out of a total of 487 million cycles. The high computational cost is primarily due to the significant number of multiplications, memory accesses, and comparisons that are required.

For each pixel in the image, the naive software implementation performs a series of operations: 28 multiplications, 28 memory accesses, and 3 comparisons. These operations are necessary for applying the Sobel kernels and combining their results. Additionally, the implementation involves several for loops to iterate over the pixels and perform these operations sequentially. Each of these operations is carried out one after the other, despite the fact that many of them are independent and could, in theory, be performed simultaneously. This sequential execution contributes to the high cycle count and overall computational load.

2.4 Movement Detection

```
for (int pixel = 0; pixel < camParams.nrOfLinesPerImage*
   camParams.nrOfPixelsPerLine; pixel++) {

  if(sobelA[pixel] == sobelB[pixel]){
    if(sobelA[pixel] == 0) dataToVga[pixel] = 0x0000;
    else dataToVga[pixel] = 0xFFFF;
  }
  else{
    dataToVga[pixel] = 0x08F0;
  }
}
```

Listing 4: Movement Detection

The final phase of image processing involves the task of movement detection. This task is performed by comparing each pixel with the corresponding pixel from the previous image at the same position. If the two pixels differ, it indicates a change in the image, suggesting possible movement. This phase contributes approximately 10% to the total computational cost, consuming around 52 million cycles. The primary cost in this phase is due to the 3 memory accesses and 2 comparisons performed for each pixel.

Moreover, one of the main challenges for this task lies in the limited opportunities for optimizations. Indeed, this phase necessitates storing in memory the previous image in its entirety since there is a dependency between pixels in the same position belonging to two consecutive images. In a concrete way, it means that the shortest dependency distance, the shortest distance between a pixel's first acquisition and where it will be needed for the next comparison, is always a whole image.

3 Design

3.1 Image Acquisition

In our design, we propose to employ a ping-pong buffer system. This method involves doubling the memory allocated for storing image pixels, thereby decoupling the image transfer from the computation. The idea is that, in this way, the cost of the image acquisition from the camera is not exposed to the computation but overlapped and hidden. In the final implementation, three different buffers are employed to allow to store the previous, current and future images at the same time. The rotation is performed using pointers to avoid transferring the data from one buffer to the other.

In more detail, the ping-pong buffer system works by using two buffers in alternation. While one buffer is used to store the incoming image data, the other buffer is simultaneously used for image processing. This concurrent approach significantly enhances efficiency by ensuring that the CPU is not left idle waiting for the DMA transfer to complete. Instead, the CPU can continue processing the previously transferred image while the new image data is being loaded into the second buffer.

Implementing this system requires using the non-blocking version of the image acquisition function. This function enables the CPU to initiate the transfer and then immediately proceed with other tasks. Synchronization between image transfer and processing is crucial in this setup. This is achieved by reading the transfer status through custom instructions, which ensures that the CPU only processes complete and valid image data.

The primary advantage of this approach is the substantial reduction in idle time, leading to more efficient use of computational resources. The only overhead introduced is the additional memory required for the second buffer and the minor cost associated with synchronization. However, these costs are outweighed by the performance gains from having a continuous processing pipeline.

Finally, the portion of the cost exposed depends on the rest of the tasks. Indeed, in case the subsequent tasks summed together were to be less demanding than the cost of the image transfer, the CPU would stall waiting for the transfer to complete. On the other hand, in case the cost of the rest of the computation were to be bigger than the image transfer, it would mean a virtual price to acquire a new image being zero.

3.2 Grayscale Conversion

In the software version, the grayscale conversion is performed after the entire image has been transferred. However, as demonstrated during the course, it is possible to offload the conversion to hardware and perform it on each pixel during the acquisition phase. With this modification, the pixels are processed in a pipeline, and the cost is integrated into that of the image acquisition. The only disadvantage of this solution is a slight increase in the critical path between pixel acquisition and its storage in the internal line buffer of the camera depending on the implementation.

On the other hand, a secondary advantage, beyond performance improvements, is the reduced storage cost. Each pixel can be represented with half the bits compared to the original solution. This reduction in storage requirements can lead to significant savings in memory usage, particularly in systems with limited storage capacity.

In our final design, we have implemented, using the module developed during the course, the grayscale conversion inside the camera module. To be able to output the converted version of each pixel, it is necessary to add some additional ports to the provided camera module. In particular, an 8-bit wide port for the grayscale converted value, a port to signal when the value read on the port is valid and two additional ports to forward the vsync and hsync signal coming from the camera itself. The valid signal refers to the grayscale port and stays high for one clock cycle and down for one clock cycle. This behavior is related to the way the camera processes each pixel, one every two clock cycles. The vsync and hsync ports are connected to the negative edge of the two internal signals inside the camera. They become high for the clock cycle where the internal signals exhibit a negative edge. These two ports are used to synchronize the computation with the frame and line of an image.

In summary, converting the image pixels from the 16-bit RGB565 format to the 8-bit grayscale format is a step that can be optimized by leveraging hardware capabilities. This optimization not only enhances performance by integrating the conversion into the image acquisition pipeline but also reduces storage requirements by halving the bit representation of each pixel. The trade-off of a minor increase in the critical path is outweighed by the benefits in computational efficiency and memory usage.

3.3 Sobel Operator

3.3.1 Algorithm

The goal is to embed the Sobel conversion in the stream of data acquired by the camera, by creating a sub-module directly connected to the camera. In this way, the filtered version of each frame will be stored in the DRAM to perform movement detection is software.
The Sobel accelerator will host the logic to perform the conversion and a DMA to interface with the bus architecture.

From an algorithmic point of view, the main challenge to obtain a streaming-like filtering is the mismatch between the way the data is given by the camera, i.e. line-wise, and the way the Sobel operator requires it, i.e. square-wise.
To do so it can be noticed that, in the most general case, each pixel contributes to nine windows (or equivalently nine centres) as shown in Figure 2, where the grid represents the frame, the yellow square a pixel and the light blue square a window containing that pixel. This premise can lead to the following logic: once a pixel is received, it must be accumulated in every window it takes part in. Formally, this translate in the

following equation:

$$d_n[i] \;=\; d_n[i-1] \;+\; pixel \cdot W_n \qquad n = 0, 1, 2, \ldots, 8 \tag{1}$$

where n represents the window number, i the step in the computation of the gradient d (either X or Y) and W the value of the Sobel matrix operator in the position occupied by the pixel in the 3x3 window. In the following, this operation will be denoted as accumulation.

To translate this algorithm in hardware one can use 9 memories that will store the gradient d of each window. In this context, the accumulation equation becomes:

$$writeData_n \;=\; readData_n \;+\; pixel \cdot W_n \qquad n = 0, 1, 2, \ldots, 8 \tag{2}$$

where $writeData$ and $readData$ are the input and output words of the memories. In words, equation 2 tells that the new data that will be written will be the old data stored in memory plus the new (filtered) pixel.

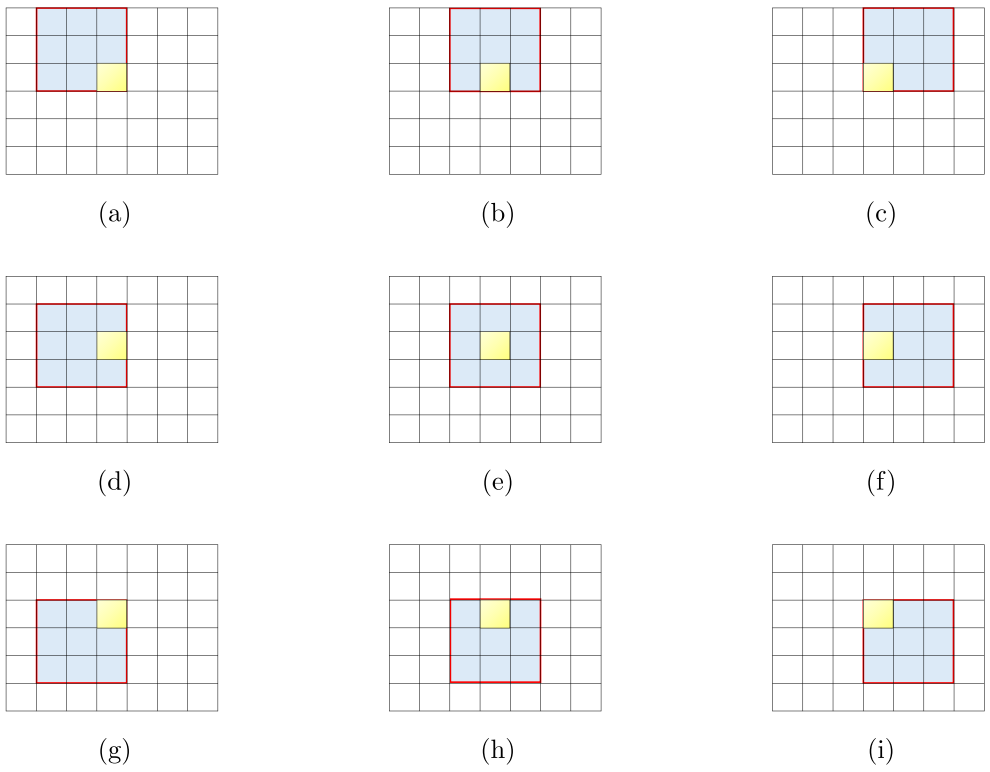


Figure 2: *Windows (light blue squares) seen by a single Pixel (yellow square).*

Taken that as premise, one must construct a rigorous algorithm to implement in hardware. One can think of the frame as a grid of pixels. As the Sobel operator acts on a 3x3 window, the frame-grid can be divided into triplets of rows and columns that repeat periodically using a modulo-3 indexing technique. Figure 3 shows the indexing.

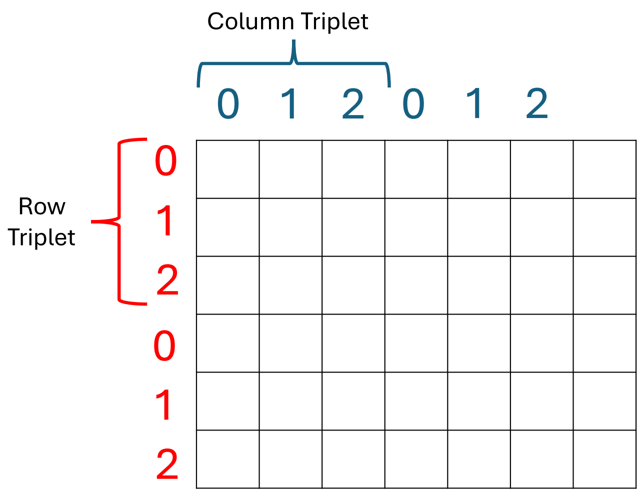


Figure 3: *Subdivision of the frame in a grid. Each row and column has an index. Indexing is performed in modulo-3 to be complaint with the 3x3 nature of the Sobel operator.*

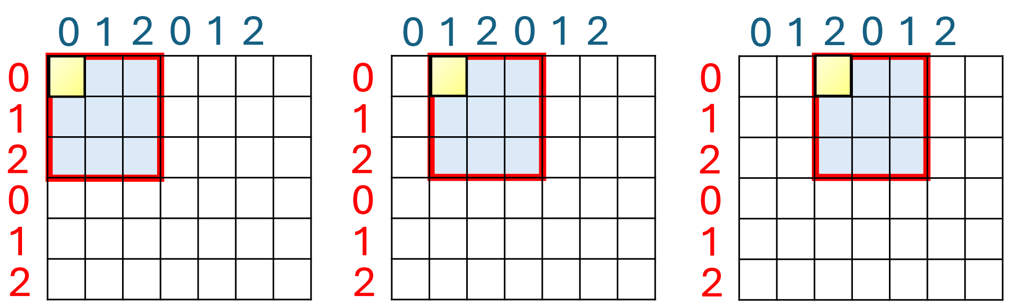
This convention allows to efficiently create a $(row, column)_{tripletR,tripletC}$ notation to denote the pixels. In this notation *row* and *column* represent the index of the row and the column where the pixel sits while *tripletR* and *tripletC* are the number of the row and column triplet to which the pixel belong.
For instance, the first pixel received will be the denoted by $(0, 0)_{0,0}$ as its row and column indexes are $(0, 0)$ and it belongs to the first triplet of both rows and columns. The second and third pixel, then, will be $(0, 1)_{0,0}$ and $(0, 2)_{0,0}$. The fourth pixel will be denoted instead by $(0, 0)_{0,1}$ because the modulo-3 nature of the indexing brings the column index back to 0, but the pixel now belongs to the second triplet of columns.

Once defined the indexing technique, it is possible to create a formal window - memory address mapping, as they are strongly correlated. The idea followed is that whenever a pixel is received a new window is opened. The new window is simply a specific address in a specific memory. The memory management is given by the following equation that describes, given a pixel, which memory is opened:

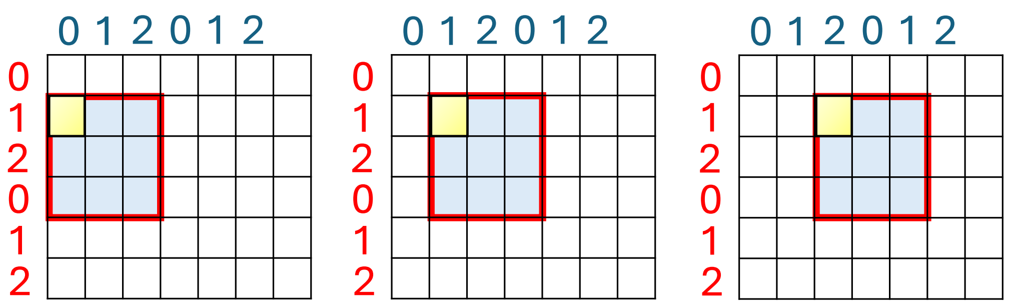
$$Memory = 3 \cdot row + column \tag{3}$$

The variables are the same defined before.
The outcome of equation 3 is shown in Figure 4. One can clearly see the realisation of the equation above by looking at the indexes of the pixels in yellow (top-left thus "opening pixels" for each window). The result is that row 0 hosts windows belonging to memories 0,1,2, row 1 refers to memories 3,4,5 and row 2 to memories 6,7,8.

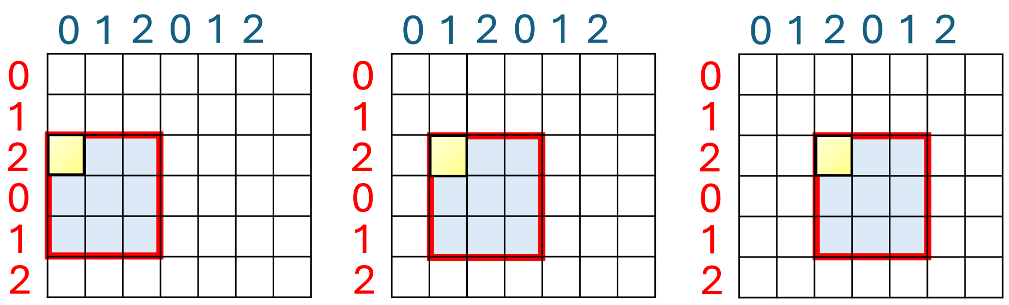
Regarding addresses, the system works such that the address in a specific memory is updated every time a new column triplet begins that is, once every three pixels. If so, memories whose modulo-3 division gives the same result, e.g. 1,4,7, update at the



(a) *Windows - Memories 0,1,2*



(b) *Windows - Memories 3,4,5*



(c) *Windows - Memories 6,7,8*

Figure 4: *Windows - Memories mapping with their corresponding top-left pixels, i.e. the pixels that open the window.*

same time, that is when the *column* value is equal to their modulo-3. So, memories (0,3,6), (1,4,7) and (2,5,8) will update their addresses at the same time. More formally, given an arbitrary $(row, column)$ pixel, the following relation can be written:

$$M = 3 \cdot [0,1,2] + column \cdot [1,1,1]; \quad (4a)$$

$$Address_M = tripletC \quad (4b)$$

To make an example, using the usual notation, as we receive the pixel $(1,1)_{0,0}$ then the address in memories 1,4 and 7 is set to 0 and it will remain 0 up to the pixel $(1,1)_{0,1}$ where it will become 1. Then at the pixel $(1,1)_{0,2}$ it will be set to 2 and so on up to the end of the line. At the beginning of every new line it will be reset to 0 as also *tripletC* becomes 0 again.

Of course, one must not forget equations 1 and 2 that describe practically how the convolution is performed.
Combining these two parts, one ends up with a new version of equation 2 that takes into account the opening of a new window. This is done by introducing the overwrite operation so that, each pixel received performs 8 accumulations and 1 overwriting.
The memory-address pair that is overwritten is determined by equations 3 and 4(a),(b) and it corresponds to the window on which the pixel occupies the top-left corner position.
The other 8 memory-address pairs where accumulation occurs are the ones mapped to all the other windows where the pixel is contained.
A modified version of equation 2 is found below.

$$overwrite_n \; = \; n \; - \; (3 \cdot row \; + \; column) \; = \; n \; - \; Memory_{ow} \tag{5a}$$

$$writeData_n \; = \; readData_n \cdot overwrite_n \; + \; pixel \cdot W_n \tag{5b}$$

The set of equations above simply signifies that if one is addressing the memory where overwriting must occur (defined uding equation 3), then the *writeData* takes only the value of the filtered pixel as *readData* is multiplied by 0.
For the sake of simplicity, the address is not taken into account in such equation because it simply follows equations 4(a),(b).

Regarding the choice of the weight W, recall that, when computing Sobel, the 3x3 convolution window directly refers to the 3x3 Sobel operator for both X and Y gradient. Using the following version of the Sobel operators

$$Sobel_X \; = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad Sobel_Y \; = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{6}$$

one maps each pixel of the window with the element of the matrix according to its positions. So, e.g. the top-left corner pixel of the window will be multiplied by the top-left corner element of the matrix (-1 for X and 1 for Y) and so on.
Also this mapping can be efficiently implemented with the indexing technique previously explained. If one refers again to Figure 4, it is enough to overlap the matrix to the light-blue window and see which indexes of the frame-grid correspond to which element of the matrix for all the possible 9 frames.
For instance, in the second image of Figure 4(c), the $(0, 0)$ element of the matrix corresponds to the index $(2, 1)$ of the frame, $(0, 1)$ element with the index $(2, 2)$ and so on for every element.
Since each window corresponds to a memory then the same pixel (denoted by a unique index) will be multiplied by a different weight for every memory. As an example, the pixel in position $(2, 2)_{0,0}$ in the X gradient computations will be multiplied by 1 for memory 0 and memory 6, 0 for memories 1,4,7, -1 for memories 2 and 8, 2 for memory 3 and -2 for memory 5. Refer to Figure 4 to recall the window memory mapping.
Mathematically, one can express this relation as:

$$Sobel_{row} \; = \; [row \; - \; \frac{n - (n \; mod \; 3)}{3}] \; mod \; 3 \tag{7a}$$

$$Sobel_{col} \; = \; (column \; - \; n \; mod \; 3) \; mod \; 3 \tag{7b}$$

$$W_n \;=\; SobelMatrix(Sobel_{row},\; Sobel_{col}) \tag{7c}$$

where $SobelMatrix$ is a placeholder for X and Y Sobel operators and $Sobel_{row}$ and $Sobel_{col}$ are the row and column indexes of the matrix derived from the pixel position in the frame-grid.

The final part to be described is how the X and Y gradients are combined to obtain the Sobel image.
In the algorithm proposed, once a window is completed, the gradient of its center pixel is directly taken for further computations. Therefore, one must define when a window is finished. This is done again using the indexing technique.
Recalling that in the proposed algorithm each window of the frame is mapped to a specific address in a specific memory, then the pixel that completes a generic window mapped to memory n (i.e. the one in the bottom-right corner) is the one whose index representation satisfies the following relations:

$$row \;=\; [\frac{n - (n \; mod\, 3)}{3} \;+\; 2] \; mod\, 3 \tag{8a}$$

$$column \;=\; [(n \; mod\, 3) \;+\; 2] \; mod\, 3 \tag{8b}$$

After that the absolute value of both X and Y gradient of such window are taken and compared with a threshold. If the whole gradient is greater then the threshold, an edge has been detected and the pixel is assigned the value 1, otherwise, the value 0.
In this way, the image has been compressed by a factor equal to the amount of bits of the pixel. In the present work, by a factor of 16 as the initial pixel was 16-bits long and the final just 1 bit long.

Now it is possible to write the proposed algorithm to implement Sobel edge detection. In the pseudo-code $Memory$ has been renamed with the pedix ow to stress the fact that that is the memory where overwrite occurs.

Algorithm 1 Proposed Sobel convolution algorithm for hardware

```
Input: Frame given sequentially pixel by pixel
Output: Sobel version of the frame

Initialize: $i \leftarrow 0$
for $P \in Frame$ do
    $Memory_{ow} \leftarrow 3 \cdot row(P) + column(P)$

    for $n \leftarrow 0$ to $8$ do
        if $n\,mod\,3 = column(P)$ then
            $Address_n \leftarrow tripletC(P)$
        end if

        $Sobel_{row} \leftarrow [row(P) - \frac{n-(n\,mod\,3)}{3}]\,mod\,3$
        $Sobel_{col} \leftarrow (column(P) - n\,mod\,3)\,mod\,3$
        $W_n^X \leftarrow SobelX(Sobel_{row},\ Sobel_{col})$
        $W_n^Y \leftarrow SobelY(Sobel_{row},\ Sobel_{col})$

        if $n = Memory_{ow}$ then
            $writeDataX_n \leftarrow P \cdot W_n^X$
            $readDataX_n[Address_n] \leftarrow writeDataX_n$

            $writeDataY_n \leftarrow P \cdot W_n^Y$
            $readDataY_n[Address_n] \leftarrow writeDataY_n$
        else
            $writeDataX_n \leftarrow readDataX_n[Address_n] + P \cdot W_n^X$
            $readDataX_n[Address_n] \leftarrow writeDataX_n$

            $writeDataY_n \leftarrow readDataY_n[Address_n] + P \cdot W_n^Y$
            $readDataY_n[Address_n] \leftarrow writeDataY_n$
        end if

        $row_{closing} \leftarrow [\frac{n-(n\,mod\,3)}{3} + 2]\,mod\,3$
        $column_{closing} \leftarrow [(n\,mod\,3) +; 2]\,mod\,3$
        if $row(P) = row_{closing}\ \&\ column(P) = column_{closing}$ then
            $G_X \leftarrow |writeDataX_n|$
            $G_Y \leftarrow |writeDataY_n|$
            $G_{tot} \leftarrow G_Y + G_X$
            if $G_{tot} \geq threshold$ then
                $sobelFrame[i] \leftarrow 1$
            else
                $sobelFrame[i] \leftarrow 0$
            end if
            $i \leftarrow i + 1$
        end if
    end for
end for
```

3.3.2 Hardware Implementation

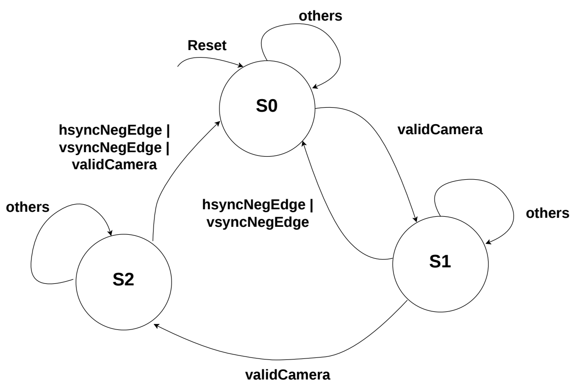
In the following an RTL implemetation of the previously described algorithm is proposed. The circuit is a standard FSM-Datapath micro-architecture.

Finite State Machine:
From the previous section, it is clear that the algorithm has been constructed such that everything is controlled by the modulo-3 row-column indexing of the frame-grid. Therefore, it would be wise to relate the control logic of the system to this indexing. To do so efficiently in hardware, two one-hot FSMs implemented as ring counters can be used. One counter will keep track of the columns, the other of the rows. The one-hot enconding will be the following, for both the state machines:

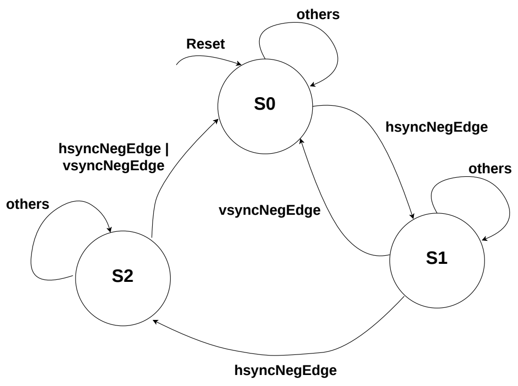
- Row/Column = 0 → state: S0 = 001
- Row/Column = 1 → state: S1 = 010
- Row/Column = 2 → state: S2= 100

This one-hot encoding allows to greatly simplify the logic in the datapath, because everything will just depend on the value of 1 bit of the state.

The state diagrams of the state machines are shown in Figure 5. The control signals (inputs) are *vsyncNegEdge*, *hsyncNegEdge* and *validCamera*. *vsyncNegEdge* and *hsyncNegEdge* are the signals that respectively denote the beginning of a new frame and a new line. *validCamera* indicates when the camera is providing the pixels of a frame. The circuit that implements the state machine is shown in Figure 6. It is made up of three flip-flops plus some simple combinational input logic.

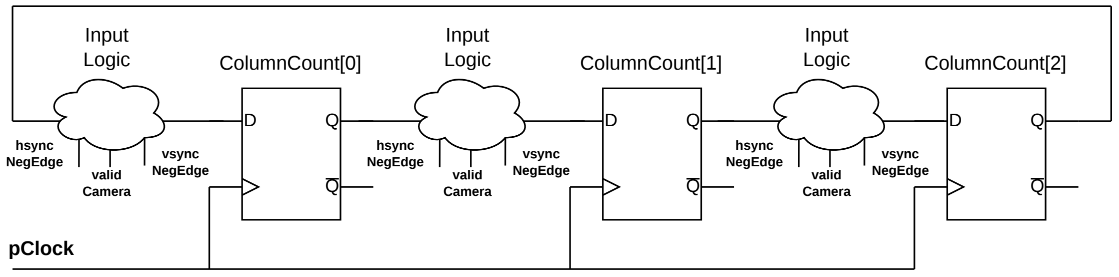


(a) *Column Counter State Diagram*

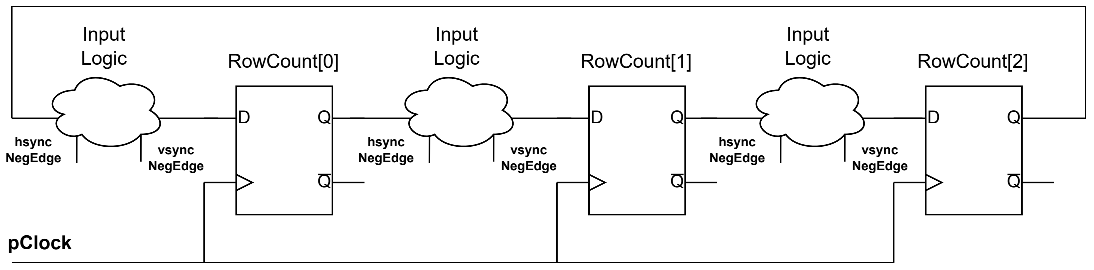


(b) *Row Counter State Diagram*

Figure 5: *State Diagrams of the column and row ring-counters.*



(a) *Column Counter*



(b) *Row Counter*

Figure 6: *Ring Counters that implement the State Machines.*

Memories and Addresses:
A total of 18 memories are used to store intermediate values, 9 for X and 9 for Y gradient. The memories instantiated are dual port SRAMs with 1kB memory space. To avoid overlapping between reading and writing, one port is active on the positive edge of clock and the other on the negative edge. The memory is depicted in Figure 8.
Addresses of the memories are managed as explained in the previous section, using equations 4(a) and 4(b) with a slight modification. As a register is present everything is delayed by one clock cycle. To manage synchronization, the address of a generic memory n is updated when the column counter is $((n \ mod \ 3) - 1) \ mod \ 3$ so that at the next cycle, when the column counter is indeed $n \ mod \ 3$, the address is already updated. The result is that the addresses of memories (0,3,6) are updated when the column counter is 2, (1,4,7) when it is 0 and (2,5,8) when it is 1. Now, to guarantee that at the first triplet of columns all addresses are equal to zero at the right moment, at each new row one needs to reset the addresses of memories (0,3,6) to 0 and the others to 255.
In hardware, the concept of triplet comes from the periodicity of the ring-counters. Each time the state is 001, a new triplet of rows or columns begins.
To implement that, the circuit built is a simple accumulator as the one shown in Figure 7 with a register to store the current values of the address and an adder to compute the next value. In the image it was written an arbitrary $columnCount[i]$. In reality, the bit of the ring counter to use depends on the memory the address refers to.

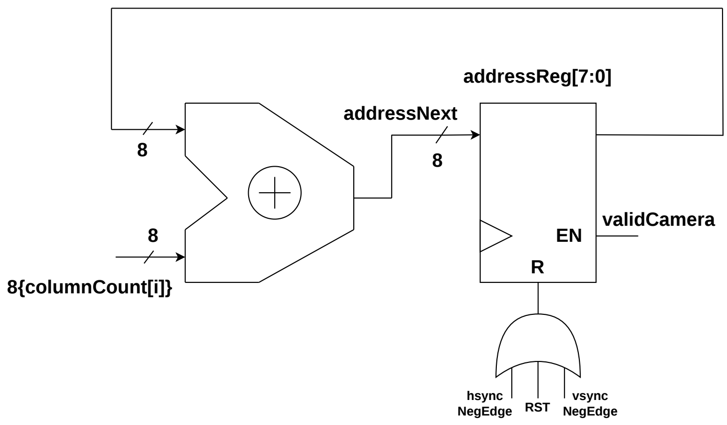


Figure 7: *Address Accumulator. 8-bit adder and 8-bit register with enable signal.*

Accumulation:
The flow of data in the system is described by equation 2.
However, some hardware optimizations have to be done. First of all, a multiplier in series with an adder is detrimental in terms of both timing and area. To resolve this issue, one can notice that the Sobel convolution contains only multiplications by 1,-1, 2 and -2. If so, bitwise operations can be used and the operations become as follow:

- $pixel \cdot (-1) \quad \rightarrow \quad \sim pixel + 1$
- $pixel \cdot 2 \quad \rightarrow \quad pixe1 << 1$
- $pixel \cdot (-2) \quad \rightarrow \quad \sim (pixe1 << 1) + 1$

Notice how, given the ease of the arithmetic, to deal with negative numbers, instead of using signed numbers the two's complement has been implemented by hand.

As a further step, to exploit parallelism as much as possible, one could think of precomputing all the possible filtered values of the incoming pixel in parallel, and then, for each memory, choose the correct result. After that according to the overwrite condition defined in the previous section, one either overwrites the pixel in memory or accumulates it to the partial gradient contained in memory. This is easily done with a 2-to-1 multiplexer. The circuit is shown in Figure 8. Selection signals of the pixel multiplexers are the row and column counters, i.e. the row and column of the current pixel. The logic to choose among the input values is the same of equations 7 (a) and (b), however here one does not choose directly the weight but the pixel already multiplied by the weight.

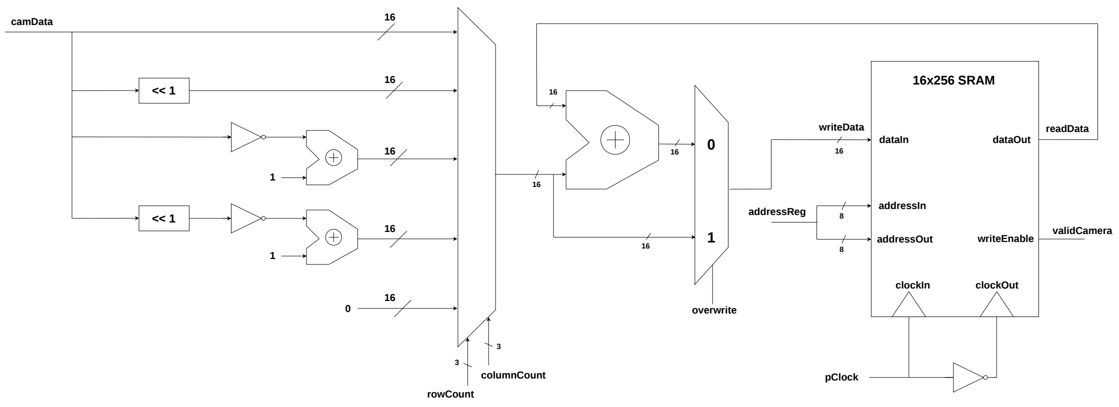


Figure 8: *DSP circuit for the Sobel convolution. Precompute the filtered values of camData. Choose the correct one and either accumulate with the previous partial gradient stored in the memory or overwrite that memory address.*

Complete Gradient and Thresholding:
One of the advantages of the algorithm proposed is that, after the first triplet of rows, at each new row the gradient of a new set of pixels is ready, as the covolution is computed on the fly.
Therefore, it is enough to design a circuit that implements the logic that, for a given pixel, outputs the window completed by that pixel, following equations 8a and 8b.
The easiest solution is to construct a combined AND-OR tree, as shown in Figure 9. Each result is in AND with the condition that verifies that its window is ready, denoted in Figure 9 as a generic selection signal. In this way, if the condition is verified, there will be an AND with 1's, otherwise an AND with 0's that gives 0. The output of the ANDs are then ORed all together such that, as only the result to be chosen has a value different from 0, the output of the OR gate will be the needed result.
This logic operation occurs for both X and Y gradient. Subsequently, the absolute value of them is taken. Again, this operation is done bitwise as:

- $|gradient| \quad \rightarrow \quad gradient[15] \oplus gradient + gradient[15]$

As a two's complement is used for negative numbers, if a number is negative its MSB is 1. The operation above ensures that if a number is negative its sign is changed, otherwise the number stays the same.
The final step is to add together the two X and Y gradients and send the result to a comparator that sets the pixel (now compressed to one bit) to 1 if the whole gradient is larger than the threshold and to 0 otherwise. Also this steps are shown in Figure 9.
As the image illustrates, to cut the critical path, pipelining has been exploited with registers after the OR tree.

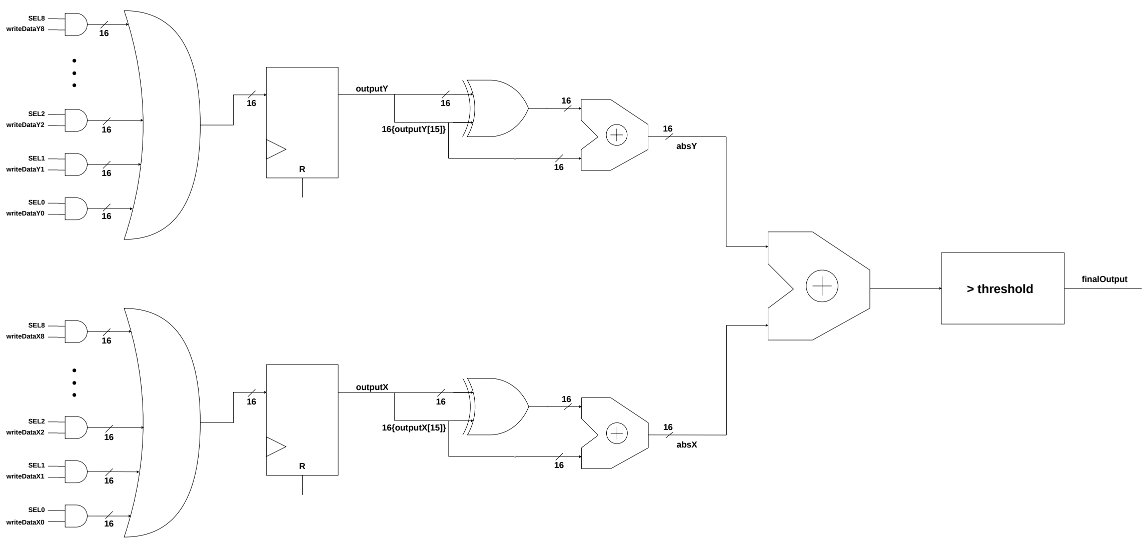


Figure 9: *Circuit to choose the writeData of the completed window and perform the absolute value and thresholding. Computation for X and Y is done in parallel.*

Line buffer:

The pixels are streamed to the bus line by line, so a intermediate line buffer has been introduced. This buffer will store an entire line of pixels. When the line is completed, it is sent to the DRAM through the bus using a DMA embedded in the accelerator and a new line can be stored in buffer.
As now, each pixel is just represented by 1 bit, then each line will be represented by 20 words of 32 bits each. So, the buffer is instantiated as a dual port SRAM of 32x32 (32 words of 32 bits).
To manage this transfer to the buffer, a serial to parallel converter can be used. The converter is implemented using a standard shift register. For 32 iterations it collects sequentially the 1-bit pixels, then it writes them as a word in the buffer.
A counter is used to manage when and where to write. The counter is reset at every new line or new frame and is incremented using the $validCamera$ signal.
A single register of 10 bits is used to manage everything: the least significant 5 bits will be dedicated to count 32 pixels, while the most significant 5 will be the write address. In this way, the most significant 5 toggle once the least significant 5 have reached their full value, that is after 32 iterations.
The reading address is instead controlled by the DMA.
The buffer, being a dual port memory, behaves also as a boundary between the camera clock and the system clock domains. Indeed, having the buffer two clock inputs, it is sufficient to write in the buffer at the camera clock rate and read from it at the system clock rate.
For other signals that come from the camera and have to control the DMA, i.e. the $hsyncNegEdge$ and $vsyncNegEdge$, the $synchroFlop$ module provided is used.

Direct Memory Access:

The DMA is embedded in the accelerator module and has been designed as a state machine. The states of the machine and their encoding are:

- IDLE $\rightarrow$ 000
- REQUEST $\rightarrow$ 001
- INIT $\rightarrow$ 010
- CLOSE $\rightarrow$ 011
- WRITE $\rightarrow$ 100

At reset, IDLE state is entered and it is maintained until a transaction is requested by the CPU. If that occurs, the machine enters the REQUEST state and a bus request is sent to the arbiter. REQUEST state is maintained until the bus access is granted. At this point the INIT state is entered. There, the DMA sends the configuration signals to the receiver within one clock period. After that, the transaction of data can begin inside the WRITE state. The machine stays there until either the transaction is completed or an error occured. In both cases, the CLOSE state is entered and, at the next clock cycle, it goes back to IDLE to start again the process for a new line. At the end of the frame, the done signal is raised for one clock cycle and the machine remains IDLE, waiting for another request from the CPU.

The DMA has been specifically designed for this module, thus only needed states and signals where defined to spare some logic.
As the accelerator does not perform any reading from the bus, no state for that purpose has been designed.
Similarly, as a line is made up of 20 32-bits words, the burst size has been hard-coded to 20 and the *byteEnables* to 1111.
All the other signals are managed following the protocol of the system.

3.4 Movement Detection

The last phase to be optimized is represented by the movement detection task. As shown by the data initially collected, movement detection accounts for around 10% of the total computation. This task has been shown to be particularly difficult to optimize due to the long dependency distance of each pixel.

The proposed design includes two different optimizations and approaches that are orthogonal. The first approach consists of an algorithmic optimization. In the naive software implementation, every pixel requires three memory accesses and a comparison. The final version, despite being still implemented in software, tries to reduce the number of memory accesses. The edge detection phase encodes each pixel in 1 bit, thus compressing the image by a factor of 16 compared to the RGB encoding. Storing the image as a uint32_t array allows it to load 32 pixels per memory access. The proposed code performs bit-wise operations to extract and compare each bit of the two consecutive images instead of accessing an element per pixel. The data reuse of each memory access enables the computation to be reduced by more than half compared to the original software implementation.

The second approach attempts to bypass the demanding CPU execution for the pixel image comparison. The main idea is the possibility of creating a signature for each image and then only comparing the two signatures belonging to consecutive images to detect movement. The movement detection phase is now performed as the comparison of two signatures instead of the comparison of each pixel, a more demanding computation. In order to achieve this goal, it is necessary to define two aspects with the first one being the algorithm used to create a signature for each image and the second one consisting of the choice of the appropriate metric to compare two signatures.

The algorithm chosen for this task is known as dHash which has several applications in the anti-piracy field and it has been shown as one of the most computationally efficient and accurate. It is, indeed, commonly used to detect copyright violations when posting pictures online. It belongs to the category of perceptual hashing functions or locality-sensitive hashing functions. These types of functions are similar to classic hash functions but with a fundamental difference. In general, a hash function is a specific function that accepts an arbitrarily long input and produces a fixed-size output. While a classic hash function tries to maximize the entropy meaning that even a small change in the input results in a completely different output, LSH functions, which dHash belongs to, try to minimize entropy. In this way, two inputs that are similar will produce similar outputs, signatures. It is possible to employ LSH functions to compare the similarity between two images. The image represents the input of the perceptual hash function while a 128-bit signature represents the output. By comparing the two signatures, it is possible to detect differences indicating movement.

The metric chosen to calculate the difference between the two signatures is the renown Hamming Distance. Given two sequences of zeros and ones, the Hamming Distance consists of the minimum number of substitutions required to change one sequence into the other. One interesting perk offered by the choice of Hamming Distance is the ability to define a threshold for the Hamming Distance. If the distance is higher than the threshold it means that the two signatures are different enough to be considered as movement in the streaming video. The threshold allows to tune the system based on the characteristics of the camera and makes it resistant to noise.

In the proposed design the Sobel accelerator is enhanced with a dedicated module to compute the signature during the image acquisition. The next step is performed in software by the CPU that accesses the signature using a custom instruction and calculates the Hamming Distance. The proposed design implements a map table to efficiently calculate the distance. To determine the Hamming Distance between two signatures it is possible to do bit-wise xor and then for each byte of the result access the table and sum together the resulting values.

In order to reduce the burden on the CPU, the signature calculation is offloaded to a dedicated hashing module connected to the camera and to the Sobel accelerator. In the next sections, an overview of the dHash algorithm and its implementation in hardware is given.

3.4.1 Algorithm

Algorithm 2 dHash Algorithm for Image Comparison

```
 1: Input: Two grayscale images to be compared
 2: Output: Boolean value indicating whether the images are the same

 3: procedure NN-Downscale($image, target\_width, target\_height$)
 4:     Initialize: $src\_width \leftarrow original\_width$
 5:     $src\_height \leftarrow original\_height$
 6:     $scale\_x \leftarrow \frac{src\_width}{target\_width}$
 7:     $scale\_y \leftarrow \frac{src\_height}{target\_height}$
 8:     $downscaled\_image \leftarrow new\_image\ (target\_width, target\_height)$
 9:     for $y \leftarrow 1$ to $target\_height$ do
10:         for $x \leftarrow 1$ to $target\_width$ do
11:             $src\_x \leftarrow \lfloor x \cdot scale\_x \rfloor$
12:             $src\_y \leftarrow \lfloor y \cdot scale\_y \rfloor$
13:             $downscaled\_image(x, y) \leftarrow image(src\_x, src\_y)$
14:         end for
15:     end for
16:     return $downscaled\_image$
17: end procedure

18: procedure dHash($image1, image2$)
19:     $image1 \leftarrow$ NN-Downscale($image1, 14, 10$)
20:     $image2 \leftarrow$ NN-Downscale($image2, 14, 10$)
21:     for each row in the grayscale image do
22:         for each pixel $p_i$ in the row, $i \in \{0, 2, \dots, 10\}$ do
23:             $difference \leftarrow p_{i+1} - p_i$
24:             if $difference > 0$ then
25:                 $signature[i] = 1$
26:             else
27:                 $signature[i] = 0$
28:             end if
29:         end for
30:     end for
31:     return $signature$
32: end procedure
```

The dHash algorithm starts with a grayscale image. The first step consists in applying nearest-neighbour sampling to downscale the image to 14x10 pixels with the second one being the calculation of the differences between consecutive sampled pixels based on their grayscale value. The final step is represented by the signature generation that is performed by placing a one every time the difference is positive and a zero every time the difference is negative.

As it can be noticed, this algorithm is suitable to be adapted to a streaming paradigm. The nearest-neighbour downscaling consists in periodically sampling the image and taking pixels as representatives of the whole image. This step can be achieved in combination with the image acquisition by simply using counters that indicate when to consider the pixel for the signature.

The difference between these sampled pixels and the subsequent sign extraction can be combined into a simple comparison. It is, indeed, possible to pipeline the pixel processing and compare every sampled pixel to its predecessor. If the new pixel has a higher grayscale value, the signature will have a one otherwise it will have a zero. The only aspect to consider is that the first sampled pixel per line should not considered for the comparison because of the absence of a predecessor. In this way, 14x10 sampled pixels turn into a 130-bit signature that is then sliced to a 128-bit value to make it suitable for an efficient hardware implementation.

3.4.2 Hardware Implementation

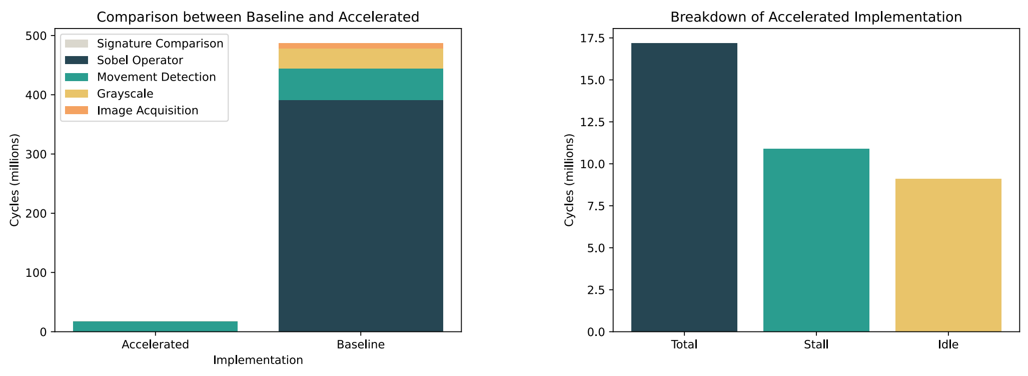
The hardware implementation is based on multiple counters that are in charge of sampling the grayscale image coming from the camera. The sampled pixels are equally distributed across the image as required by the Nearest-neighbour algorithm. In addition, there is a basic mechanism that ensures that the first pixel of each line in the 14x10 final image is not compared to a predecessor.

Based on the comparison of every sampled pixel, the corresponding bit is set in a shift register called *currentSignatureReg* while the new sampled pixel is stored to be used in the next comparison. Every 32 sampled bits the shadow register is saved in a small buffer used to store the whole signature. The process is repeated four times per image creating a 128-bit signature.

In order to synchronize the signature generation with the image processed by the Sobel Accelerator, the *takeSignature* signal, coming from the accelerator, acts as an enabler for the creation of a new signature. In this way, the dedicated module is guaranteed to be generating a new signature every time the CPU requests the Sobel module to acquire a new image and to never overwrite it until no longer necessary. The main idea is that the image shown by the camera will be always associated with the signature behind considered.

To handle the double clock, a dual port memory module is employed. It allows to write the values using the camera clock and read them using the system clock. The whole module is implemented as a custom instruction and the CPU can request to read the signature in blocks of 32-bit using as operand the index of the requested slice.

4 Results



(a) (b)

Figure 10: *Comparison of final design and baseline version.*

As the final step of this work, the proposed system has been evaluated using performance counters. The results show that offloading the Sobel operator and grayscale conversion in combination with algorithmic optimizations to the movement detection phase, have achieved a 28x improvement compared to the baseline. In addition, the use of a rotating buffer to store the compressed images has allowed to completely overlap the data movement with the CPU processing, further reducing the cycle count.

The data in figure 10 shows that there is still unnecessary data movement to perform the comparison in software and cycles wasted to perform the pixel-by-pixel comparison. The movement detection phase is now the main bottleneck of the design being still implemented in software and performed by the CPU. Despite being further optimizable by moving this task completely in the hardware accelerator, this solution would require a dedicated memory to store the previous image for the comparison or to significantly modify the structure and concept behind the already present components.

From a subsequent comparison, the use of a Locality-Sensitive Hash function to detect the movement proves to be a valid alternative thanks to the simplicity of the hardware implementation and the robustness of the algorithm that intrinsically provides noise cancellation from the image. It exhibits insignificant overhead over the software-optimized version. The use of the additional custom instruction combined with the fast Hamming Distance calculation does not impact the total number of cycles when compared to the pixel-by-pixel comparison. The qualitative tests show that the movement detection performed using the dedicated hash module is well aligned with the original version in terms of accuracy and can be further tuned to be sensitive only to real movements.

To conclude, the design has reached the maximum achievable performance while still utilizing the CPU. Despite the possibility of additional improvements, any further optimization would require the use of specialized hardware to perform the movement detection.