Hardware acceleration for image processing

Semester project
January 2008, EPFL, I&C, BIRG

Author Lukas Benda
Supervisors Pierre-André Mudry
Prof. Auke Ijspeert

Contents

1 Introduction **1**

2 Methods **3**
2.1 Image processing . . . 3
2.2 Hardware . . . 5
2.2.1 Material used . . . 6
2.3 Sliding window architecture . . . 8
2.3.1 Three images lines FIFO . . . 9
2.3.2 Image border copy . . . 9
2.3.3 5x5 processing window . . . 9
2.4 Communication and memory management . . . 10
2.4.1 Register transfers . . . 11
2.4.2 DMA transfers . . . 11
2.4.3 Parameters transfer . . . 12
2.5 Processing primitives . . . 13
2.5.1 Implemented primitives . . . 13
2.5.2 Gaussian filter . . . 15
2.5.3 Gradient image . . . 18
2.5.4 Median/Min/Max filter . . . 21
2.5.5 Canny edge detector . . . 24
2.6 C Program . . . 29
2.6.1 Communication with the FPGA . . . 29
2.6.2 Image processing . . . 29
2.6.3 OpenCV . . . 29

3 Results **31**
3.1 Global comparison . . . 32
3.2 Relative comparison . . . 34
3.3 Hardware time analysis . . . 35

4 Conclusion **37**
4.1 Future work . . . 37
4.1.1 External RAM memory . . . 37
4.1.2 Larger processing window . . . 37
4.1.3 Processing immediately after reception . . . 38

5 Acknowledgments **41**

References **43**

List of Figures

2.1 Image representations 4
2.2 Image processed by a sliding window. The convolution is given here as an example of processing primitive. 5
2.3 Global architecture of the system 6
2.4 A Virtex II FPGA similar to the one used in this project 6
2.5 A Logitech QuickCam Web 7
2.6 Hardware architecture of the sliding window 8
2.7 Image border copy for different sizes of processing window 10
2.8 FPGA usage for the sliding window 10
2.9 Architecture including a block RAM memory 11
2.10 Software image division 12
2.11 Operators design 13
2.12 FPGA usage for an Operator unit 14
2.13 FPGA usage for Operator unit (containing four operator unit) . . 14
2.14 Gaussian distribution with mean (0,0) and $\sigma = 1$ 15
2.15 Gaussian discrete kernel 15
2.16 Gaussian filter convolution 16
2.17 Gaussian filter example 16
2.18 FPGA usage for the Gaussian filter 17
2.19 Gradient function approximation 18
2.20 Gaussian discrete kernel 18
2.21 Gaussian filter convolution 19
2.22 Gradient image example 19
2.23 FPGA usage for the Gradient filter 20
2.24 Architecture for sorting 3 pixels 21
2.25 System global architecture 22
2.26 Median filter example 22
2.27 FPGA usage for C_2 (two pixel comparator) 23
2.28 FPGA usage for C_3 (three pixel comparator) 23
2.29 FPGA usage for median filter 23
2.30 Canny edge detector algorithm 24
2.31 Possible directions for the gradient phase 24
2.32 Simplified phase calculations 25
2.33 Hardware schema computing the phase 25
2.34 FPGA usage for gradient Magnitude and Phase 26
2.35 Example of non-maxima elimination for a particular pixel 26
2.36 Hardware schema for non-maxima elimination 27
2.37 FPGA usage for Non-maxima elimination 27
2.38 Canny edge detector steps 28
2.39 Cone and rodes sensitivities 30

3.1 openCV functions processing time 32
3.2 Hardware processing time 32
3.3 Processing times 33
3.4 Processing times relatively to a hardware monopass processing time 34
3.5 Subdivision of hardware processing time 35
3.6 Subdivision of hardware processing time 35
3.7 FPGA usage for the whole system 36

4.1 Processing times 38

Chapter 1

Introduction

Today images are more and more part of our everyday life. We use cell phones, webcams and high resolution cameras and we can modify our photos on our computer. In industry this trend is visible as well, very often combined with some sort of image processing. We can see security systems identifying cars' licence plates, character recognition systems to recognize addresses on letters as well as medical imagery we can find in hospitals.

The goal of this semester project is to obtain a hardware acceleration system for image processing. There are essentially two reasons to implement this kind of processing in a hardware way : speed and implementation in an embedded system. Speed up image processing can be very useful for real-time processing images coming form a camera. This way we avoid heavy processing with a generic microprocessor. Embedding this system can be useful in many fields, like vision in robotics or a monitoring system.

The idea here is mainly to obtain a system faster than software image processing. The final user should be able to process a stream of images coming from a video camera with a simple call to a set of C functions. The link with hardware has to stay invisible to the user. The framework stays limited to grayscale images with constant size is going to implement several kinds of filtering (Gaussian blur, median/min/max filter, gradient image and a Canny edge detector). These operators are based on a 3 by 3 pixels wide sliding window similarly to convolution.

The system is going to run on a Xilinx FPGA communicating with the computer through the PCI port. In order to test and compare the results with software image processing, the whole process is going to be compared to openCV functions [5].

Chapter 2

Methods

2.1 Image processing

There are different kinds of images and many different processing methods we can apply to an image. We can distinguish three kind of images.

Binary images

Binary images, where a pixel is black or white. These images are mostly used for morphology processing, the main operators are dilatation and erosion. "The game of life" is a well known example of this kind of processing.

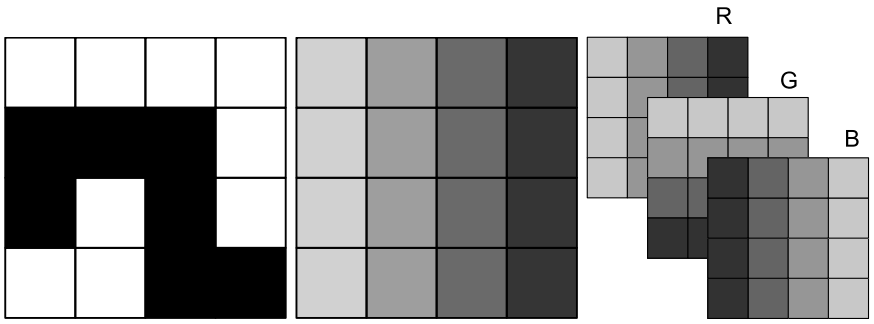
Greyscale images

Greyscale images where a pixel is defined by an intensity level between black and white (included). Theses images are used if we are interested in the shape of objects present in the image. There are many operators that can be applied on these images, like a simple Gaussian blur, an edge detector or a sophisticated face recognition system.

Color images

The last kind of image used in image processing are color images, where a pixel is defined as a triplet of intensities of red, green and blue. These images can be seen as a group of three grey images. The processing applied on these images concerns essentially colors, like white balancing, color region classification, etc ...

In this project we choose to work only with grey images which is a good complexity compromise between binary and color images. We also needed to choose the kind or family of operators that we want to apply on our images. Some operators, like histogram equalization, need global informations about the image, like the min/max pixel of the whole image. Others operations need only partial or local information about the image, for example convolution only needs to know the currently processed pixel's neighborhood. Operators can also be more complex and need several computational steps, for example edge detection discussed later. We choose to use only operators that use a windowing system, which means that we only need to know the currently processed pixel and it's neighborhood. Convolution is a window operation, but there are window operations that are not



(a) Binary image (b) Grey image (c) Color image

Figure 2.1: Image representations

convolutions. There is the complete list of the operators choose to be implemented in our project :

- Identity operator
- Gaussian blur
- Gradient image
- Median filter
- Canny edge detector

All these operators will be discussed later in detail.

The approach of image processing using a processing window is described in figure 2.2 :

The processing window (PW) slides on the image and a specific processing is applied to every pixel. The resulting pixel depends on the operator, the processed pixel and its neighborhood. Of course we need to choose the size of our processing window. It can be between 0 and the size of our image. We choose a 3 by 3 pixels PW which is simple to handle. Of course a bigger PW is more powerful, but requires more computation, or more precisely a longer delay as our primitives are made in a combinatorial way.

We mentioned the neighborhood of the currently processed pixel. We realize that this neighborhood is not defined for pixels on the corners or edges of the processed image, because the neighborhood is "outside" of the image. Different approaches are used to solve this problem. We can simply set the undefined pixels to a constant value, for example black. Another way is to imagine the image as a torus, the top edge linked to the bottom edge and the left side linked to the right side. The adopted solution was to set the undefined neighbor pixel to the same gray value as the currently processed pixel.

Convolution is a widely used operation based on a PW and we give it in figure 2.2 as an example. Each pixel of the current PW is multiplied by the corresponding "pixel" of the convolution kernel, which is of the same size as the the PW. The results of these multiplications are summed up to obtain the resulting

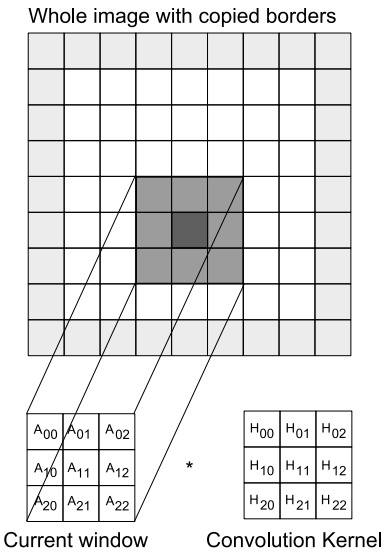


Figure 2.2: Image processed by a sliding window. The convolution is given here as an example of processing primitive.

pixel. Once the operation is done the PW "slides" on the next pixel. Convolution can be described by the following formula :

$$X(m,n) = I(m,n) \times H(m,n) = \sum_{j=0}^{J-1} \sum_{k=0}^{K-1} H(j,k) I(m-j, n-k) \qquad j,k \in \{0,1,2\}$$

In the figure above, the current pixel is A_{11}, its neighborhood are the pixels A_{xy} and the convolution kernel is H_{xy}.

2.2 Hardware

In the previous section we described the theoretic approach to process an image. Now we want to implement this system on hardware keeping in mind that our goal is speed gain. A lot of design choices had to be done. There is also a lot of limitations linked to hardware that are not present in software image processing. That is what we are going to discuss now.

The general architecture of the whole system is shown in figure 2.2.

In order to show our results on a real-time application we choose to use as input a continuous flow of images provided by a webcam. Here we had the choice to connect the webcam directly to the FPGA that contains the image processing system and sends directly the result to the computer or to connect the camera to the computer, store the image in the computer memory and only then to send it to the FPGA. We choose the second approach because we want to make performance comparisons between hardware and software computational speed. The whole system is controlled through a C program. This way we can create an interface that hides the hardware part to the programmer.

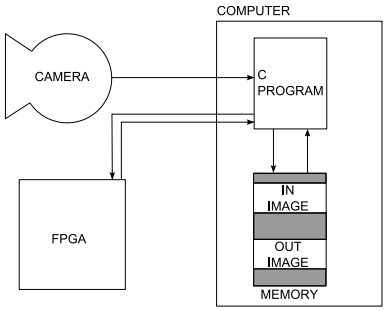


Figure 2.3: Global architecture of the system

2.2.1 Material used

Here we describe the specifications of the equipment used in this project.

Computer

The computer used was a Pentium 4 with 1GB of RAM using Linux 2.6.9-55.0.6.ELsmp as operating system.

To program and test the FPGA, we used Xilinx ISE 9.1i and ModelSim SE 6.3a software.

Fpga

It was important to take into account the limitations of the FPGA used since the beginning of the project to avoid lack of memory or logical units at the end. The used FPGA was a Xilinx Virtex II 3000-4ff1152 which is similar to the one on in figure 2.4.

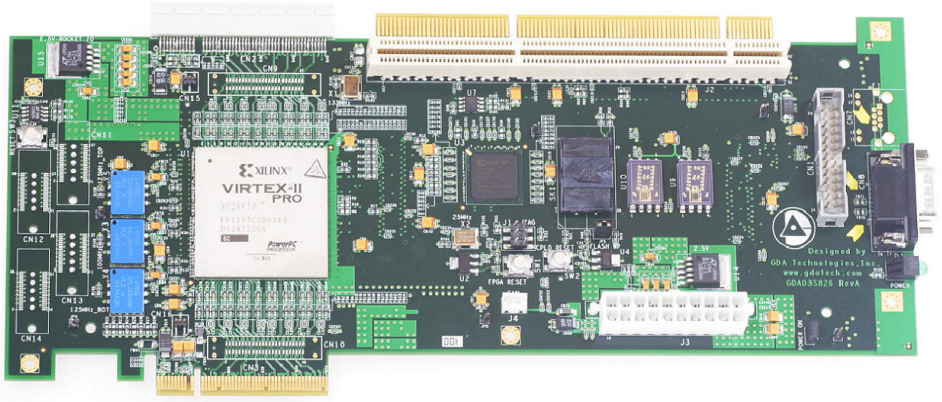


Figure 2.4: A Virtex II FPGA similar to the one used in this project

Refer to [4] for complete specifications.

Camera

We used the Logitech QuickCam Web. It provides color images of size 356 by 292 pixels.

Figure 2.5: A Logitech QuickCam Web

Comparative software

Anthony Edward Nelson [1] has done a project similar to ours, the results were about 100 times slower, but the work was done in 2000 and the architecture used less hardware. In his work the comparison between hardware and software results was done using Matlab. Unfortunately, Matlab is not designed for fast real-time image processing. We choose to use the openCV [5] library. OpenCV not only allows very fast real-time image processing, but is also written in C, which allows us to run our processing functions and openCV processing functions in the same C program and make more precise comparisons.

OpenCV also provides very useful input/output functions that we need in any cases. We use openCV functions to acquire images from the camera. Plotting functions are not necessary, but are very useful for tests.

2.3 Sliding window architecture

The figure 2.2 shows the theoretical approach of image processing using a sliding processing window. Now we want to apply this theoretical model to a hardware architecture.

As we will see later, the communication between the computer and the FPGA is the speed bottleneck of the whole system. For this reason we want to transfer the minimum amount of data. In [1] only the nine pixels of the PW were stored in the FPGA. Using this approach, after each "slide" of the PW, 3 new pixels were red from the image and three pixels were forgotten. This means that after the processing of the whole image each pixel was read three times and it represents a lot or redundant transfers, but the register usage was small. If a 5 by 5 pixels windows was used each pixel would be read five times.

To avoid these redundant transfers we chose to permanently store three lines of the image. The PW slides from left to right on the central line. When it comes at the right end, a new line is red and the oldest line is forgotten. These forgotten pixels will newer be used again and each pixel of the image is red only once. Of course this approach implies much higher amount of registers on the FPGA. Figure 2.6 illustrates this approach :

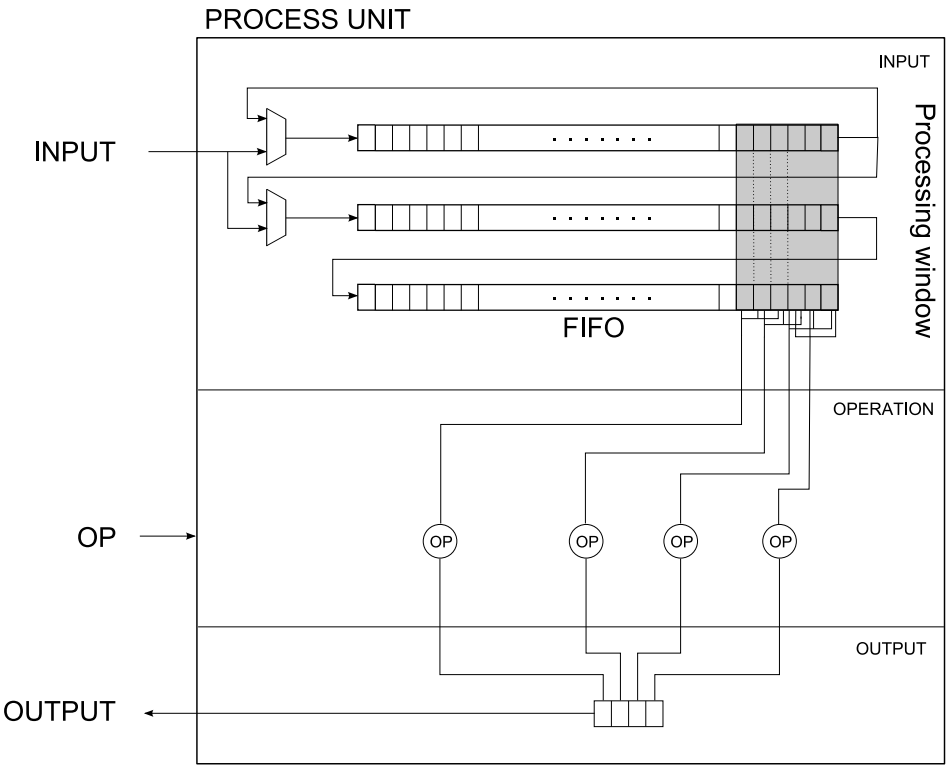


Figure 2.6: Hardware architecture of the sliding window

In order to use a standard word length for the PCI transfers, we wanted to handle pixels using 32 bits long words. As each pixel has a value between 0 and 255 and is stored on 8 bits, we process pixels by packets of four. Due to this fact we process always four pixels paralellry. It is why we have four identical processing

primitives on the previous schema. Due to this fact, we create a constraint on the size of our input image which has to have a length which is a multiple of four. There is no constraint on the image height.

2.3.1 Three images lines FIFO

Unlike in the theoretical version, the hardware PW doesn't slide on the pixel lines, but the pixels are shifted under the PW. After several versions, the simplest and fastest way to implement the three lines system was to use three large interconnected FIFOs. The input pixels are pushed in the first FIFO and the popped pixels are the ones processed by the PW. A state machine separates the processing into three stages.

First stage : fill the FIFOs

First we need to wait until all the FIFOs are full. At this stage there is only input. No output and no processing.

Second stage : main processing

Second we can begin to process the image. At this stage we read one input pixel unit, we process it and we get an output as result. The whole is done in one clock cycle. This is another constraint on our processing primitives, each operation has to be done in one clock cycle. We will see the impact of this constraint later.

Last stage : empty the FIFOs

In the third and last stage, most of the image was already processed and there is no more input to read, because the whole image was red, but the FIFOs are still full. In this stage we process and output pixel units until the FIFOs are empty.

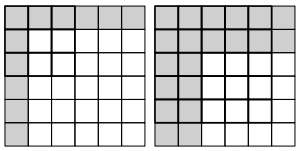
2.3.2 Image border copy

As we told above the approach used to manage the undefined neighborhood, was to copy the borders. Here we separate the problem into horizontal and vertical borders. The corners are taken into account in the horizontal part. To manage the borders, we use a vertical and a horizontal counter to know which part of the image we are processing. The horizontal copy is performed if we are processing the first or the last line of the image. In this case we use multiplexers to copy the first or the last line twice as we can see in figure 2.6.

This way the neighborhood pixels of the first and the last line are explicitly inside the FIFO. Concerning the vertical borders, those are not inside the FIFO, but they are implicitly computed by the PW depending on the vertical counter. This is the best approach we found to reduce complexity and memory usage.

2.3.3 5x5 processing window

Even if this functionality wasn't implemented, it is interesting to consider the changes necessary to use a 5 by 5 PW to show the scalability of our system. We need three modifications. First we need five lines of the image instead of three, which requires more registers, but is still feasible using our FPGA. The second



(a) 3x3 window (b) 5x5 window

Figure 2.7: Image border copy for different sizes of processing window

thing to do is the copy of a border of two pixels instead of a border of one pixel as we can see in figure 2.7.

The third modification concerns the processing primitives, because we have an input of 25 pixels instead of 9. All these modifications are not difficult to implement, but the third one is quite long.

11-bit adder	2
1-bit register 11-bit register 2848-bit register 8-bit register	1 2 3 1
48-bit latch	3
2848-bit 4-to-1 multiplexer	1
Number of 4 input LUTs Number used as logic Number used as Shift registers	1035 out of 28672(3%) 204 831
Number of GCLKs	2 out of 16 (12%)
Delay	6.370ns (Levels of Logic = 4)
Clock period	6.370ns (frequency: 156.986MHz)

Figure 2.8: FPGA usage for the sliding window

Table 2.8 shows the FPGA elements needed for the sliding winfow architecture.

2.4 Communication and memory management

Once we defined our sliding window system as close to the theoretical one as possible, we want to integrate it as an entity in a complete system. To do this we need a way to communicate between the FPGA end the computer. Depending on the type of communication chosen, we need or not to use RAM memory on the FPGA.

The communication part is done using the PCI bus. It is the most sensible part of the architecture, because it represents the speed bottleneck of the whole system. It is much more time demanding than the processing itself as we will see in the obtained results.

We have the choice between using simple register transfers or DMA burst transfers. Here we implemented both approches to allow comparisons.

2.4.1 Register transfers

This type of transfer is much easier to implement than the DMA transfer, so it was implemented first to allow tests on the whole system. The problem here is that we can transfer only one 32 bit word at the same time. Each transfer of this kind needs to open the PCI bus, perform the transfer and then close the PCI bus. All this has to be done once per a pixel unit, which is 4 pixels. This approach is very slow because of the numerous openings/closings of the PCI port. On the other hand, using using this technique, we don't need to use any additional memory. We only fill the three lines and the resulting output pixels are directly send the result back to the computer. We don't need to modify the schema, because the input and output pixels are directly connected to the PCI bus. Once these transfers were working we could finish the processing part of the project and test the whole system very fast, but the speed results were clearly unsatisfying and we had to modify our approach.

2.4.2 DMA transfers

As the goal of this project was speed, DMA transfers became necessary. This way the PCI bus was opened and closed only once per image, and the speed of transfer became one 32 bit word by clock cycle. The implementation was based on an example code given by Alpha-Data [3]. Using this approach and unlike the register transfer, we needed to store the whole image in the FPGA memory before processing. The memory management is shown in figure 2.9.

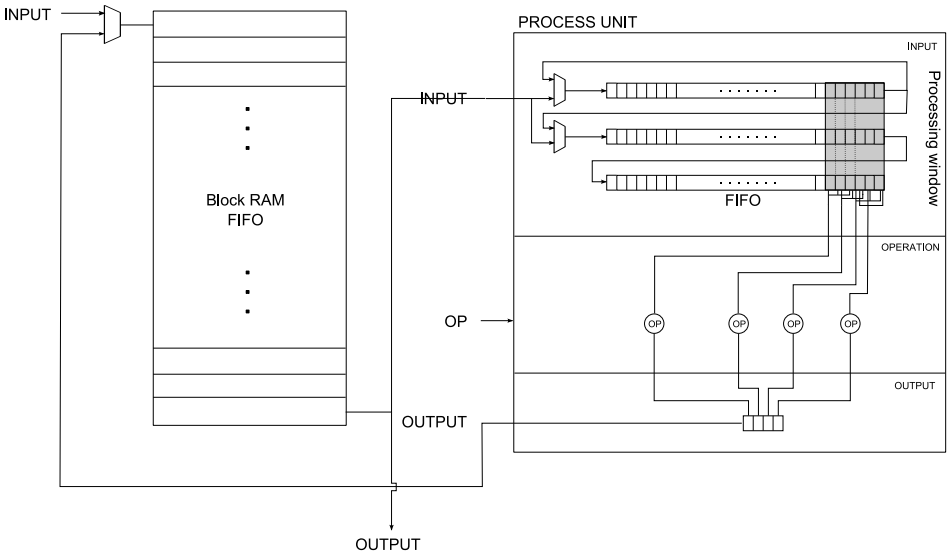


Figure 2.9: Architecture including a block RAM memory

Here the memory used to store the image is again a simple FIFO. Using this architecture we need three stages. First, during the incoming DMA transfer, the

FIFO is filled up with the complete input image. There is no processing at this stage. Once the image is inside the FIFO the second stage begins. We pop pixels inside the processing unit, perform processing and we push them back inside the same FIFO. The multiplexer chooses to push input pixels in the FIFO if we are in the first stage and result pixels if we are in the second stage. Once the whole image is processed, which means that the FIFO contains only processed pixels comes the third stage where the content of the FIFO is transferred back to the computer.

Unfortunately, here we reach the limits of our FPGA. Even if the amount of RAM memory is just sufficient to contain a 356 by 292 pixels wide image, we are not allowed to generate a single FIFO large enough for the whole image. A first approach to solve this problem would be to generate a large FIFO by connecting two smaller ones. We suppose this would be the fastest way to solve the problem, but here we reach memory space limits of our FPGA. To solve this problem and at the same time show the scalability of our system, we choose to "cut" the input image using software and we send two half of the image and process it as two separated images. If we use a bigger image it's easy to extend this approach to more image divisions. Once the two images are processed and send back to the computer, we need to connect them to provide the real output image. To do this we have to manage borders and process the middle line of the image twice as is shown in figure 2.10.

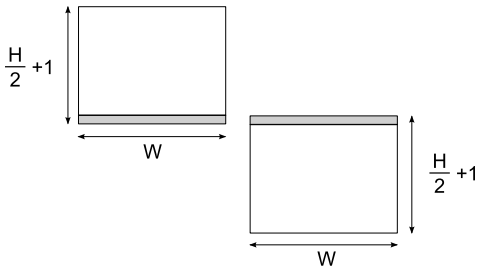


Figure 2.10: Software image division

As we said if we use a bigger image, we can divide it in more than two parts, but this approach slows the system down.

2.4.3 Parameters transfer

We don't only need to transfer the image to be processed. The hardware also needs to know which kind of processing we want to apply (Gaussian, median, ...). To do this we had the choice to use register transfer for the arguments and DMA transfer for the image. This solution would avoid to attach a header to each image at the software level. At the other hand if we need to send a lot of arguments, it would slow down the system. The chosen approach was to attach a header in a software way, perform a single DMA transfer and then separate the image from the header inside the FPGA.

2.5 Processing primitives

This part concerns the image processing itself and describe in detail all the processing primitives. First we give elements of image processing theory and then we explain their hardware implementation. Each primitive is followed by an output image example and the number of necessary logical units used by the FPGA to implement the primitive.

2.5.1 Implemented primitives

Identity	Returns the input image, used for tests
Gaussian filter	Smooths the image
Median/Min/Max Filter	Removes 'Salt and pepper' noise preserving edges
Gradient filter	Returns the gradient image
Canny edge detector	Returns thin edges of the image

The whole computational unit "Operators" is compsed of four "Operator" units. This way we can process four pixels in a parallel way. A single "Operator" unit contains each of the primitives described above. This is shown in Figure 2.11.

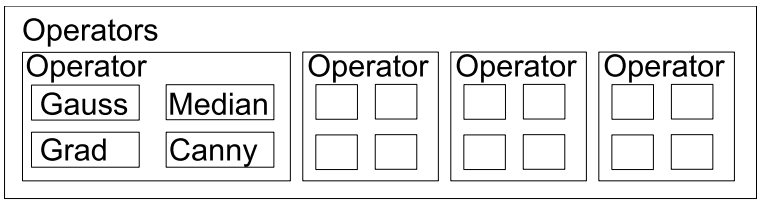


Figure 2.11: Operators design

Tables 2.12 and 2.13 describe the FPGA usage of the "Operator" and "Operators" modules.

10-bit adder	11
11-bit adder	8
11-bit subtractor	4
12-bit adder	1
13-bit adder	2
9-bit adder	10
2-bit latch	1
13-bit comparator greater	2
6-bit comparator greatequal	1
6-bit comparator greater	8
8-bit comparator less	18
2-bit 3-to-1 multiplexer	1
Number of 4 input LUTs	806 out of 28672(2%)
Delay	38.088ns (Levels of Logic = 63)

Figure 2.12: FPGA usage for an Operator unit

10-bit adder	44
11-bit adder	32
11-bit subtractor	16
12-bit adder	4
13-bit adder	8
9-bit adder	40
2-bit latch	4
13-bit comparator greater	8
6-bit comparator greatequal	4
6-bit comparator greater	32
8-bit comparator less	72
2-bit 3-to-1 multiplexer	4
Number of 4 input LUTs	3206 out of 28672(11%)
Delay	38.409ns (Levels of Logic = 64)

Figure 2.13: FPGA usage for Operator unit (containing four operator unit)

2.5.2 Gaussian filter

The Gaussian filter is a convolution operator which is used to blur images and remove noise. In the continuous domain, we can find it for a certain σ through :

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

which looks like :

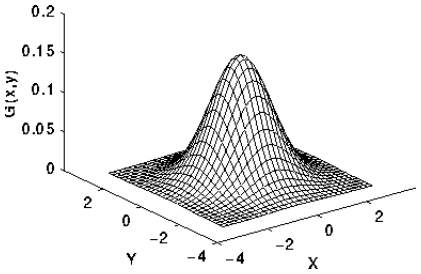


Figure 2.14: Gaussian distribution with mean (0,0) and $\sigma = 1$

For our purpose use we will use a discrete version of this kernel with a 3x3 window :

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} * \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Figure 2.15: Gaussian discrete kernel

Figure 2.16 shows the hardware implementation of the Gaussian filter we designed:

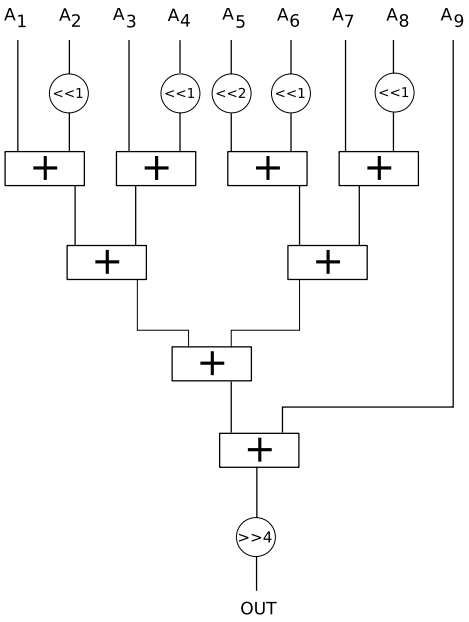
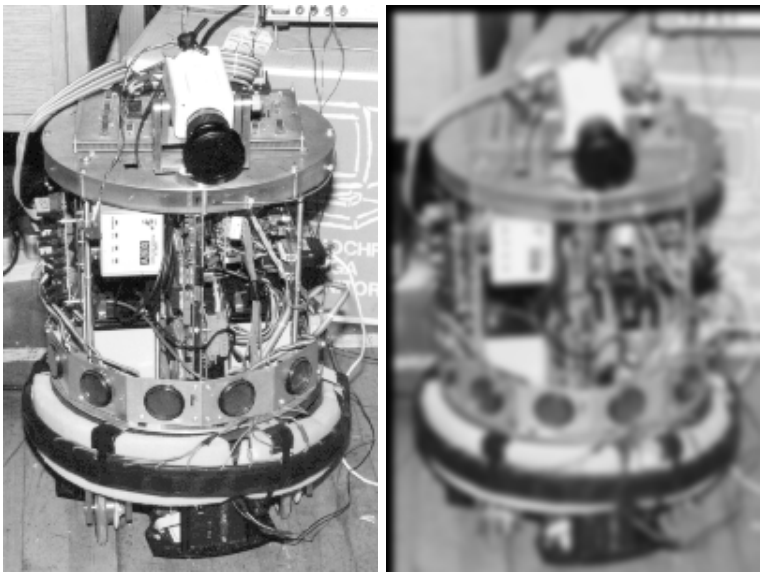


Figure 2.16: Gaussian filter convolution

Exemple



(a) Input image (b) blurred image with $\sigma = 2$

Figure 2.17: Gaussian filter example

FPGA usage

10-bit adder 11-bit adder 12-bit adder 9-bit adder	3 2 1 2
Number of 4 input LUTs	64 out of 28672(∼0%)
Delay	19.867ns (Levels of Logic = 20)

Figure 2.18: FPGA usage for the Gaussian filter

2.5.3 Gradient image

The gradient of an image is essentially used for edge detection in image processing. The idea is to first find the horizontal and vertical derivatives of an image, this means regions in the image where the difference of intensity of close pixels. For this reason the filter is very sensitive to noise and we usually apply a Gaussian blur before using this operator. The filter is defined by the following operation :

$$\bigtriangledown A = \frac{\delta I}{\delta x} + \frac{\delta I}{\delta y} = (H_x \times A) + (H_y \times A)$$

Then we can obtain the gradient magnitude :

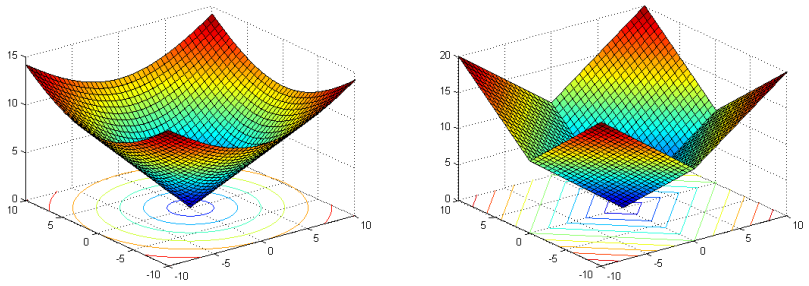
$$|\nabla A| = \sqrt{(H_x \times A)^2 + (H_y \times A)^2}$$

And the gradient direction :

$$\theta(\nabla A) = arctan\left(\frac{H_x \times A}{H_y \times A}\right)$$

The gradient magnitude is often approximated by :

$$|\nabla A| = |H_x \times A| + |H_y \times A|$$



(a) real function (b) Approximated function

Figure 2.19: Gradient function approximation

We will also use this approximation in our hardware implementation in order to reduce complexity. The convolution kernel used is the Sobel operator :

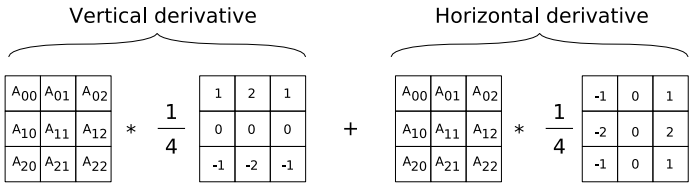


Figure 2.20: Gaussian discrete kernel

The following schema shows the hardware implementation of the gradient image we designed:

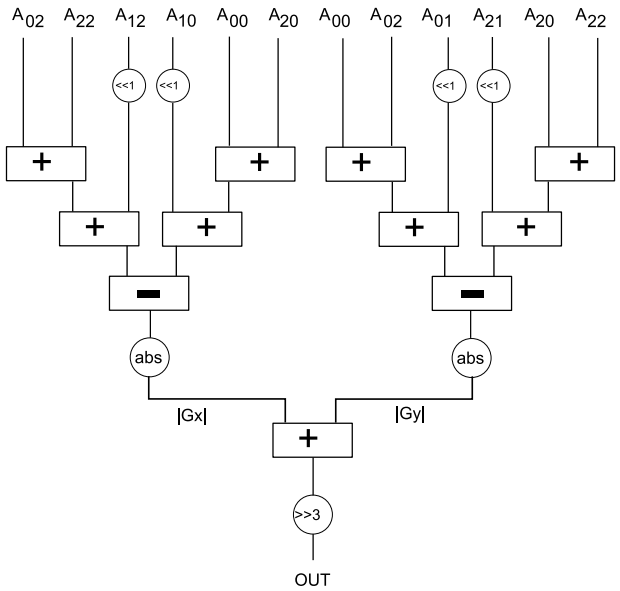


Figure 2.21: Gaussian filter convolution

Exemple

(a) Input image (b) Gradient image

Figure 2.22: Gradient image example

FPGA usage

10-bit adder 11-bit adder 12-bit subtractor 9-bit adder	4 3 2 4
Number of 4 input LUTs	113 out of 28672(∼0%)
Delay	20.565ns (Levels of Logic = 16)

Figure 2.23: FPGA usage for the Gradient filter

2.5.4 Median/Min/Max filter

This operator is a bit different from Gaussian filter and gradient image because it is not a convolution. Here the idea is to sort the pixels of the current window. There are three possibilities of filtering with this approach. We can always take the median value, or always take the minimum value or always take the maximum value. Depending on this choice, the output image will be the following :

Min	Erosion of the features of the image
Max	Dilatation of the features of the image
Median	Flattening of the image
	'Salt and pepper' noise removal

For our purpose we don't need to sort completely the nine pixels of our window. All we need to know is the min, max and median pixel. It is relatively easy to sort n^2 pixels in a hardware fashion, but here we have 9 pixels. To solve this problem, we first define a comparator C3 that sorts 3 pixels and is assembled of three binary comparators C2. This architecture is shown in the next figure :

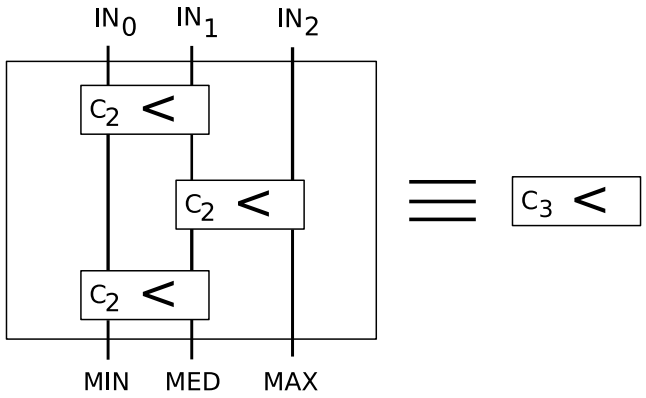


Figure 2.24: Architecture for sorting 3 pixels

Using only six of these comparators, and a multiplexer to select the min/max/median value, we are not able to implement a 9 pixel sorter, but it is sufficient for our needs, because we find the min/max and median value. The 'X' represent the remaining unsorted pixels.

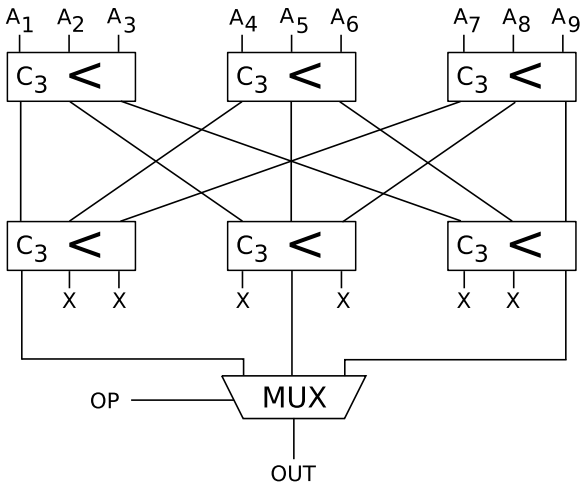


Figure 2.25: System global architecture

Example

(a) Input image (with noise)

(b) Median filter 3x3

(c) Median filter 5x5

Figure 2.26: Median filter example

FPGA usage

8-bit comparator less	1
Number of 4 input LUTs	24 out of 28672(∼0%)
Delay	10.434ns (Levels of Logic = 12)

Figure 2.27: FPGA usage for C_2 (two pixel comparator)

8-bit comparator less	3
Number of 4 input LUTs	72 out of 28672(∼0%)
Delay	19.960ns (Levels of Logic = 31)

Figure 2.28: FPGA usage for C_3 (three pixel comparator)

8-bit comparator less	18
Number of 4 input LUTs	368 out of 28672(∼1%)
Delay	35.179ns (Levels of Logic = 61)

Figure 2.29: FPGA usage for median filter

2.5.5 Canny edge detector

This operator is here to show the usage of a multi-pass filter. An detailed explanation of this algorithm can be found in [2]. The popular Canny edge detector uses the following steps to find contours presents in the image.

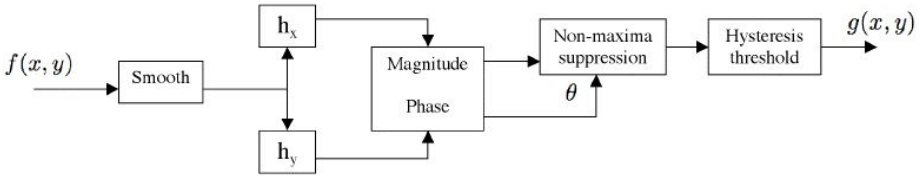


Figure 2.30: Canny edge detector algorithm

The first stage is achieved using our Gaussian smoothing. The resulting image is send to the PC that sends it back to the gradient filter, but here we modified our gradient filter a bit because this time we don't only need the gradient magnitude that is given by our previous operator, but we need separately G_x and G_y. We also need the phase or orientation of our gradient which is obtained using the following formula :

$$\theta = arctan\left(\frac{G_y}{G_x}\right)$$

As we can see this equation contain an arctan and a division. These operators are very difficult to implement using hardware. We also don't need a high precision. The final θ has to give only one of the four following possible directions, as we can see on Figure 2.31. The fourth direction is the horizontal direction with zero degrees, not indicated on the figure.

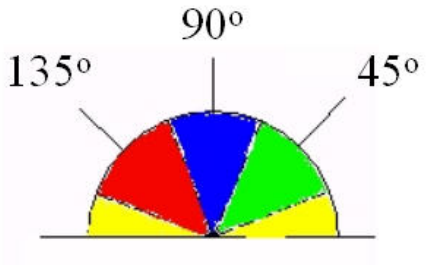


Figure 2.31: Possible directions for the gradient phase

Arctan and the division can be eliminated by simply comparing G_x and G_y values. If they are of similar length, we will obtain a diagonal direction, if one is at last 2.5 times longer than the other, we will obtain a horizontal or vertical direction. The figure 2.32 shows this idea.

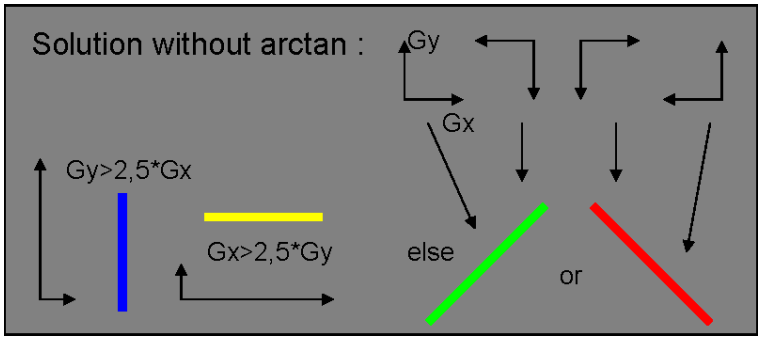


Figure 2.32: Simplified phase calculations

Here we can see a new problem. For the next stage, non-maxima elimination, we need two informations, the magnitude and the phase at the same time. Our system is designed to process only one image at a time. The solution used here was to store both informations on the same result image. Each possible 8bit pixel stores one of the four possible directions on two bits and the magnitude on the 6 remaining bits. The price to pay is the lost of two bits of information of the magnitude. The figure 2.33 shows the corresponding hardware schema, Table 2.34 presents its FPGA usage.

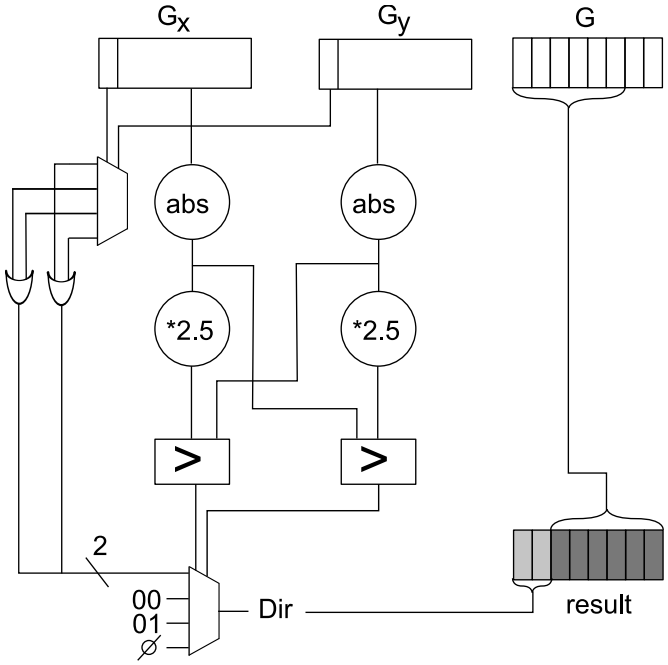


Figure 2.33: Hardware schema computing the phase

10-bit adder	4
11-bit adder	3
11-bit subtractor	2
13-bit adder	2
9-bit adder	4
2-bit latch	1
13-bit comparator greater	2
2-bit 3-to-1 multiplexer	1
Number of 4 input LUTs	168 out of 28672(∼0%)
Delay	20.707ns (Levels of Logic = 16)

Figure 2.34: FPGA usage for gradient Magnitude and Phase

Non-maxima elimination

The non-maxima elimination filter is here to eliminate pixels that are not part of a continuous line, like isolated pixels even if they have a high gradient magnitude. This technique allows to get ride of a lot of noise.

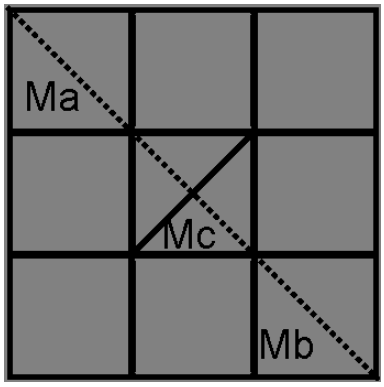


Figure 2.35: Example of non-maxima elimination for a particular pixel

The figure 2.35 shows this principle. The M_x is the magnitude of the current pixel. M_a and M_b are the magnitudes of the neighbor pixels perpendicular to the current pixel's phase.

If $(M_x > M_a)$ AND $(M_x > M_b)$ we eliminate the current pixel, else we keep the pixel. The figure 2.36 shows the hardware implementation.

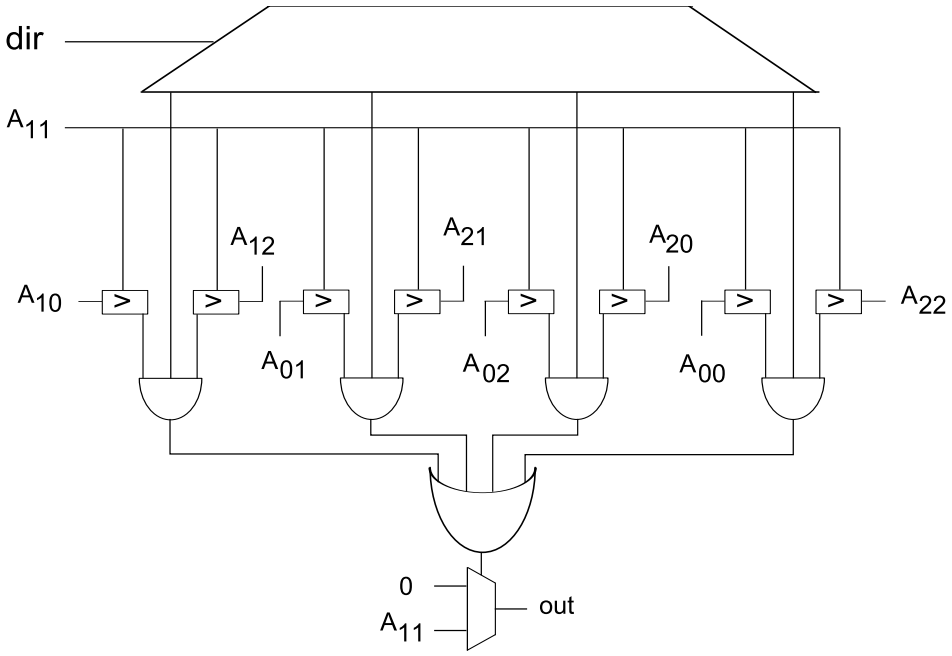


Figure 2.36: Hardware schema for non-maxima elimination

For the last part, the edge detector uses only one threshold instead of the usual two. But this is not adapted to hardware implementation.

6-bit comparator greatequal 6-bit comparator greater	1 8
Number of 4 input LUTs	59 out of 28672(∼0%)
Delay	11.845ns (Levels of Logic = 12)

Figure 2.37: FPGA usage for Non-maxima elimination

Example

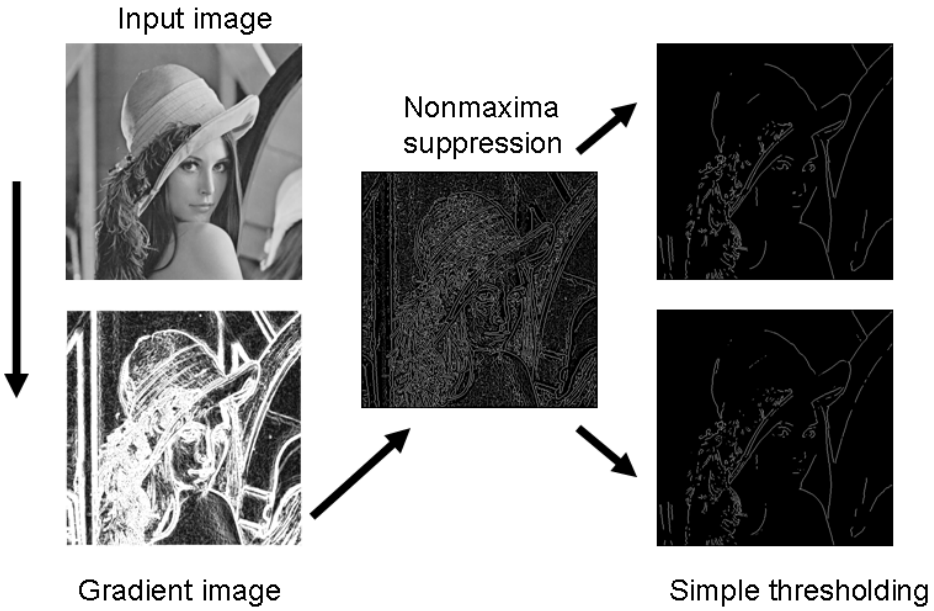


Figure 2.38: Canny edge detector steps

2.6 C Program

Three sets of C functions were used through this work. First the functions used for communication with the FPGA. Theses functions are not supposed to be used by the final user. The image processing functions that are given to the user and perform calls to the hidden communication functions. Finally we used openCV functions for speed comparisons.

2.6.1 Communication with the FPGA

The initFPGA() is a modified and modular version of the example program given by Alpha-Data. It is called only once in the main() function. It allocates block of memory that will be used to communicate. The closeFPGA() frees the allocated space. Again it has to be called only once at the end of the main() function. Unfortunately these two functions are necessary and because of them the hardware system is not as hidden as we wanted.

2.6.2 Image processing

The process_image(iin, iout, oper) function receives pointers to input/output images and operator type as arguments. We need to copy the input image in the space allocated by initFPGA(). This is unfortunately a necessary step because we can transfer only the block of memory allocated above. To transfer the operator type (identity, Gaussian, ...) and its parameters (min, max or median for the median filter and threshold level for the canny edge detector)we first copy this operator as a header of the image at the beginning of the allocated memory and then we copy the image itself.

The received resulting image don't need any header when it is transferred back to the computer.

2.6.3 OpenCV

First tests were not done on real images. We used small still images generated by hand. This way we could easily plot the resulting integer value of each pixel. In order to get an input image from the camera, we used cvCaptureFromCAM() and cvQueryFrame(). These functions five us a color image that has to be transformed into a grey image. This conversion is done by simply taking the green value of the RGB image. Green is used because human sensitivity to green is very close to the human sensitivity to light intensity as shows figure 2.39 found in [7].

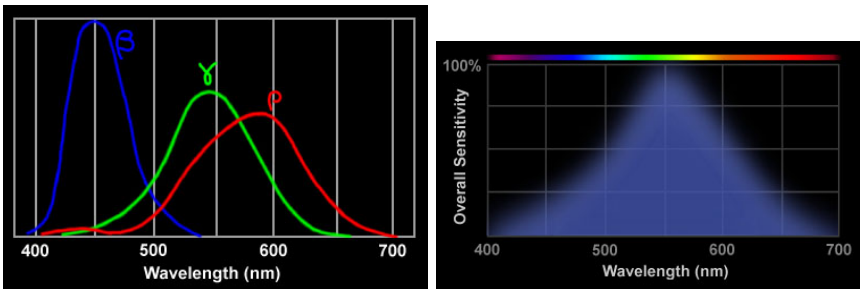
This is again a necessary software processing that has to be done once per acquisition. Fortunately for multipass primitives as Canny edge detector we don't apply this operation between different passes, but only before the first one, because after the transformation we use the same grey image format.

To obtain an image plot as output we used cvShowImage(...) to test visually large images.

openCV image processing

We also used the following image processing functions to compare the processing speed between hardware and software :

- cvSmooth(input, output, CV_GAUSSIAN, windowSize);



(a) Sensitivity of cones (color sensitivity)

(b) Sensitivity of rhodes (luminescence sensitivity)

Figure 2.39: Cone and rodes sensitivities

- cvSmooth(input, output, CV_MEDIAN, windowSize);
- cvCanny(...);

A software equivalent of our hardware Identity function makes no sense and the there is no equivalent openCV function for a Gradient image.

Chapter 3

Results

It is interesting to note that different openCV functions need more or less processing time depending on their complexity. For example median filtering is more complex and longer than gradient filtering. With the FPGA this is not the case. The processing time is equivalent independently of the type of processing. This is due to our vhdl implementation constraint which says that any operator has to process a block of four pixels in one clock cycle. Knowing this we decided to show processing time for each software operator. For the hardware version we give the time needed for different parts of the processing: DMA transfer length, processing length, etc ...

An interesting thing to note is that, if available, the Intel Integrated Performance Primitives(IPP) is used for lower-level operations for OpenCV [6]. In the case tis option was activated, we were not comparing our system to pure software processing but to a hardware accelerated software. Unfortunately did not manage to verify if the openCV hardware acceleration was available or activated.

3.1 Global comparison

In Figure 3.1 we present processing times of the software implementation of our primitives. The name of the primitive is followed by NxN which is the size of the processing window. These times are shown in blue in the bar plot of Figure 3.3 and are preceded by the mention "soft_XXX". On the same graph, the resuls are of our hardware implementation using DMA transfers are shown in green and preceded by the mention "hard_XXX". The exact times are shown in Figure 3.2.

hard_complete	time needed for the whole processing
hard_in	time needed for incoming DMA transfer
hard_out	time needed for out DMA transfer
hard_prcessing	time needed for image processing

The red bar on the right, "hard_register" is the performance of our hardware version implementing register transfers.

Every given time lengths are given in milliseconds[ms].

median 7x7	median 5x5	median 3x3	gauss 7x7	gauss 5x5	gauss 3x3	canny
0.051606	0.040798	0.013219	0.007561	0.004694	0.003877	0.004051

Figure 3.1: openCV functions processing time

processing	transfer IN	transfer OUT	complete using DMA transfer	complete using register transfer
0.000746	0.001206	0.001302	0.003725	0.054165

Figure 3.2: Hardware processing time

The figure 3.3 gives a global comparison between processing using openCV software, hardware with DMA transfers and hardware with register transfers.

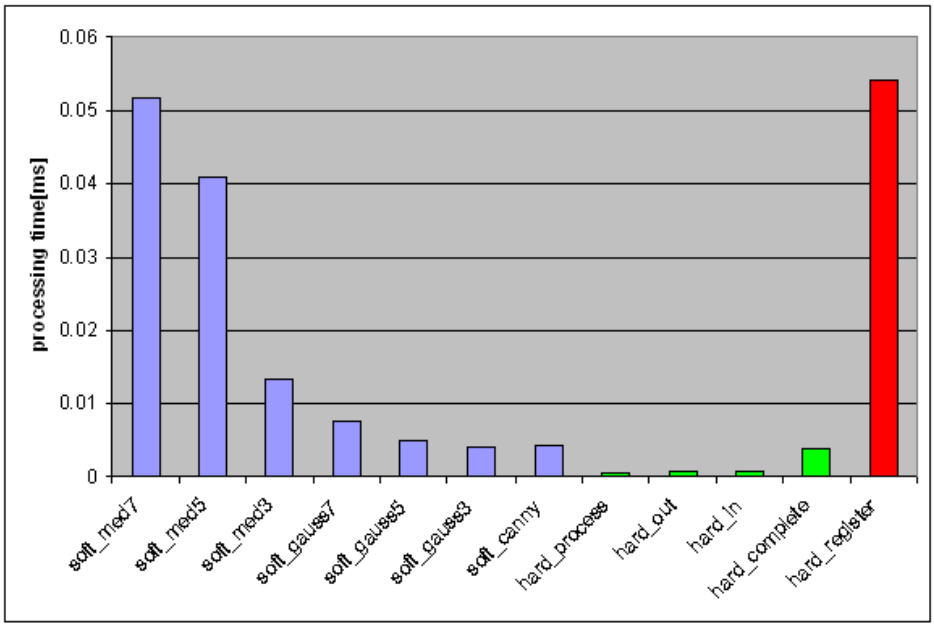


Figure 3.3: Processing times

3.2 Relative comparison

The figure 3.4 shows the software and register transfer results of Figure 3.3. This time the vertical axis unit is the time needed for a "hard_complete" processing. For example a median filter using a 7x7 wide processing window takes about 14 times more processing time than a hardware primitive using DMA transfers.

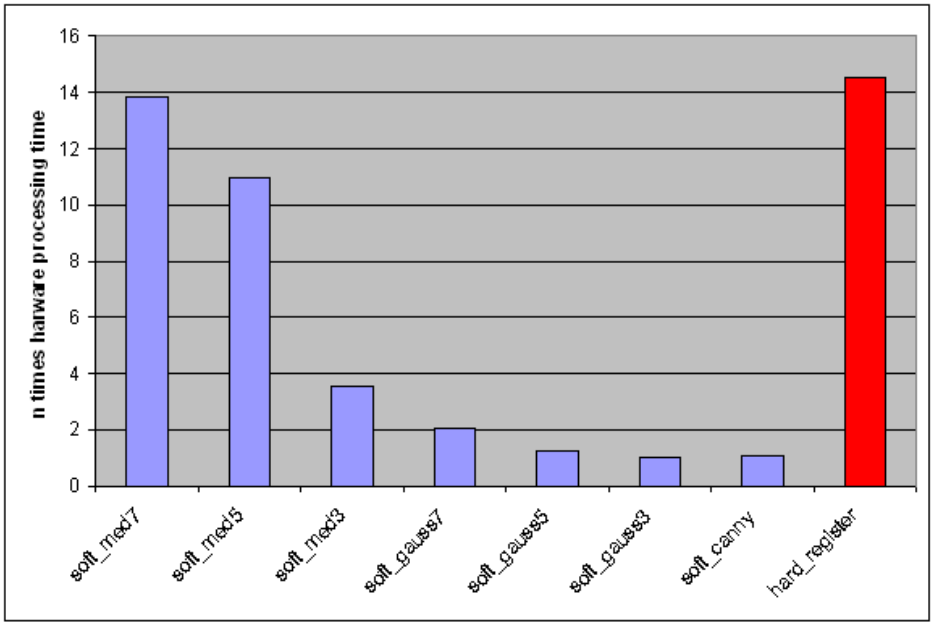


Figure 3.4: Processing times relatively to a hardware monopass processing time

3.3 Hardware time analysis

In order to analyze more deeply the hardware version using DMA transfers, figure ?? gives time percentage for a full monopass processing. We can see that the speed bottleneck of our system is the transfer of data. The "software part" shown here is here because we want to take into account all the software processing needed by our hardware system. This processing includes the arrays copies discussed above, image header formatting, image division, etc ...

processing	transfer IN	transfer OUT	software
0.000746	0.001206	0.001302	0.000471

Figure 3.5: Subdivision of hardware processing time

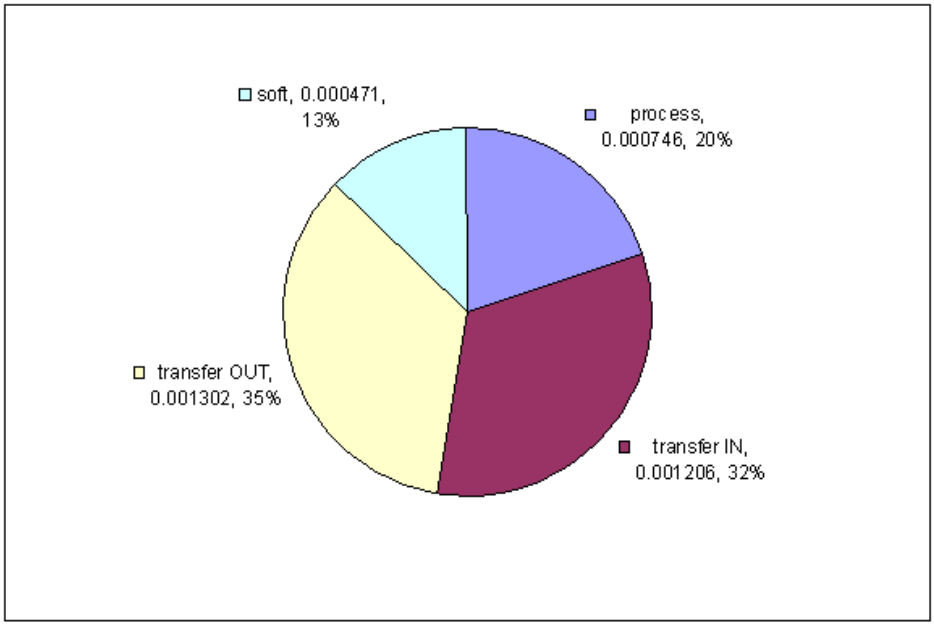


Figure 3.6: Subdivision of hardware processing time

Figure 3.7 shows the final usage of the FPGA needed for the whole project.

10-bit adder	44
11-bit adder	34
11-bit subtractor	16
12-bit adder	4
13-bit adder	8
14-bit adder	1
15-bit addsub	1
30-bit subtractor	2
32-bit adder	1
9-bit adder	40
14-bit up counter	3
32-bit up counter	2
1-bit register	89
11-bit register	2
128-bit register	1
15-bit register	1
2-bit register	1
22-bit register	1
2848-bit register	3
3-bit register	2
32-bit register	3
4-bit register	1
8-bit register	1
2-bit latch	4
48-bit latch	3
13-bit comparator greater	8
17-bit comparator less	1
31-bit comparator greatequal	1
31-bit comparator greater	1
31-bit comparator less	1
31-bit comparator lessequal	1
32-bit comparator less	3
6-bit comparator greatequal	4
6-bit comparator greater	32
8-bit comparator less	72
2-bit 3-to-1 multiplexer	4
2848-bit 4-to-1 multiplexer	1
32-bit 4-to-1 multiplexer	1
1-bit tristate buffer	2
32-bit tristate buffer	1
Number of 4 input LUTs	4489 out of 28672(15%)
Number used as logic	3913
Number used as Shift registers	576
Number of BRAMs	32 out of 96 (33%)
Number of GCLKs	2 out of 16 (12%)
Clock period	9.216ns (frequency: 108.502MHz)
Delay	9.216ns (Levels of Logic = 13)

Figure 3.7: FPGA usage for the whole system

Chapter 4

Conclusion

For monopass image processing primitives, the results obtained using hardware with DMA transfer are faster than their software implementation. For mutipass processing this is not always the case, because of numerous transfers that remain the bottleneck of our system. As the processing is really fast, it would be interesting to use much more complex processing primitives so that the transfers would be just a small percentage of the whole process.

4.1 Future work

The results obtained using DMA transfers were satisfying because faster than the software version. Anyway there are several possible ways to get a higher speed up.

4.1.1 External RAM memory

As we saw above, the speed bottleneck of the project are image transfers. When we use a multipass primitive like the canny edge detector, the image is transferred three times from the computer to the FPGA and back again. To reduce the number of transfers, we could use external RAM memory present on the board. This way we could store intermediate images directly on the FPGA instead of transferring large amount of data several times between the computer and the FPGA. Here we suppose memory transfers are faster than DMA transfers, an assumption that should be tested.

4.1.2 Larger processing window

For this project we choose 3 by 3 pixel wide processing window. We can extend this to 5 by 5 or 7 by 7 pixels wide window. Doing this we can apply more complex processing to our images, but the main advantage is again indirectly the speed up. As we have seen, the hardware processing time stays constant, this seems to be true (but should be tested) even if we use a larger window. On the software side, it is much longer to process a larger window as we saw in the results chapter.

4.1.3 Processing immediately after reception

Another idea that could be interesting to test is to process pixels immediately after their reception. This way we would not wait until the FIFO is filled with the incoming image before the processing as the actual implementation does. Doing this we could slightly reduce the processing time. Figure 4.1 illustrates this process.

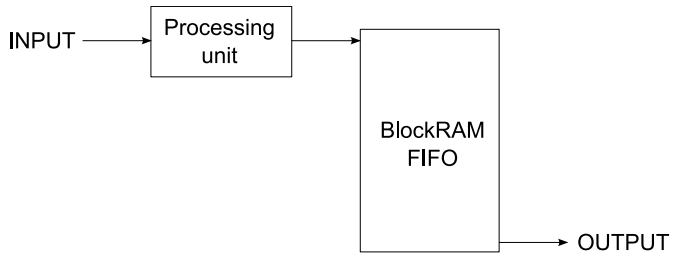


Figure 4.1: Processing times

Chapter 5

Acknowledgments

I would like to thank Pierre-André Mudry, the supervisor of this project as well as Michel Ganguin, Julien Ruffin and Alessandro Crespi for their great help on this project.

Bibliography

[1] Anthony Edward Nelson, 2000, *Implementation of image processing algorithms on FPGA hardware*, Vanderbilt University.

[2] Dmitrij Csetverikov, *Basic Algorithms for Digital Image Analysis: a course*, Eötvös Loröand University, Budapest, Hungary.

[3] *ADM-XRC SDK 4.8.0 User Guide (Win32).*

[4] Xilinx, 2005, *Virtex-II Platform FPGAs: Complete Data Sheet.*

[5] Vadim Pisarevsky, 2000, *Introduction to OpenCV*, Intel Corporation, Software and Solutions Group.

[6] Changsong Shen, Sidney S. Fels and James J. Little, *OpenVL: Towards A Novel Software Architecture for Computer Vision*, University of British Columbia.

[7] *Human color vision :*
http://photo.net/photo/edscott/vis00010.htm

Appendix

The attached files contain the vhdl and C code of the project as well as this file.

Vhdl files

comp2.vhd
comp3.vhd
comp9.vhd
gauss3x3
grad3x3.vhd
gradPM.vhd
nonmax.vhd
operator.vhd
operators.vhd
input1.vhd
ddma-xrc.vhd
image_pack.vhd

C files

fpga.h
fpga.c
opencv.h
opencv.c
simple.c
Makefile