FPGA-Based Edge Detection with Subpixel Accuracy

Prof. Dr. Eng. Sergiu Nedevschi
Technical University of Cluj-Napoca
Department of Computer Science
Baritiu 28, RO-400391 Cluj-Napoca, Romania
Sergiu.Nedevschi@cs.utcluj.ro

Stefan Mathe
Technical University of Cluj-Napoca
Baritiu 28, RO-400391 Cluj-Napoca, Romania
mathestefan@freemail.utcluj.ro

***Abstract*— Edge detection is ubiquitous in image processing. Methods that provide high quality results are available, but are computationally expensive. A solution using FPGA technology for such a method having subpixel accuracy is presented. The method requires computing the gradient, the second directional derivative along the direction of the gradient and the histogram of the gradient. An efficient pipeline design that performs this task is proposed. The performance of the implemented solution on a VirtexE600 FPGA device is analyzed and compared with that of today's personal computers.**

I. Introduction

Entailed by most image processing applications, edge detection techniques have represented an active field of research since the beginnings of image processing. Several such techniques exist, each having different characteristics (accuracy, noise sensitivity, processing cost, etc.). In most applications, edge detection is performed in the early stages of processing and all the operations that follow are entirely dependant of the quality of the results provided by this stage. It is thus essential to choose an edge detection technique that has good functional characteristics (like noise sensitivity and accuracy). Unfortunately, there is a tradeoff between the functional and non-functional characteristics of an edge detector (like processing speed and memory requirements). Especially in the case of real-time applications, meeting the nonfunctional requirements becomes a critical issue.

In this work, an edge detection technique having subpixel accuracy is considered. This technique provides high quality edges but also implies high processing costs. The alternative of a hardware solution using FPGA technology is presented, in order to reduce the processing time and make the technique usable for real-time applications.

II. The Edge Detection Algorithm

The method ([1],[2],[3]) implies two stages. The first stage requires computing the second directional derivative along the direction of the gradient. This is, according to [4], the 2D equivalent of the derivative of the one-dimensional Gaussian.

Let us denote by $G(x, y)$ the Gaussian kernel, given by:

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

Let I be the image under consideration and n be the direction of the gradient ($*$ denotes the convolution operator):

$$n = \frac{\nabla(G * I)}{|G * I|} \tag{2}$$

The first derivative along the direction of the gradient, G_n is given by the formula:

$$G_n = \frac{\partial G}{\partial n} = n \cdot \nabla G \tag{3}$$

The second derivative along the direction of the gradient is thus computed as:

$$\frac{\partial^2}{\partial n^2} G * I = \frac{\frac{\partial^2 I}{\partial x^2} \cdot \left(\frac{\partial I}{\partial x}\right)^2 + 2 \cdot \frac{\partial I}{\partial x} \cdot \frac{\partial I}{\partial y} \cdot \frac{\partial^2 I}{\partial x \partial y} + \frac{\partial^2 I}{\partial y^2} \cdot \left(\frac{\partial I}{\partial y}\right)^2}{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2} \tag{4}$$

There are several solutions that can be used to compute the derivative terms that appear in equation (4). Some of the operators mentioned in the literature are Roberts, Sobel and Prewitt. Due to its low noise sensitivity, the Sobel operator has been preferred for our application. It consists of the following 3x3 convolution kernels, one for the x and the other for the y axis:

$$\frac{\partial}{\partial x} = \frac{1}{8} \cdot \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \tag{5}$$

$$\frac{\partial}{\partial y} = \frac{1}{8} \cdot \begin{pmatrix} -1 & -2 & 1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \tag{6}$$

Having computed the second directional derivative, process of edge extraction at subpixel accuracy can start. This second stage of the edge detection process will not be presented in detail in this work, since it is beyond its scope.

The process consists of two inter-related processes:

- A scanning process over the entire image that identifies the starting points for contour following and extraction

- A contour tracing process, which is triggered by the scanning process each time a valid contour starting point is found. At startup, this process receives the coordinates of the starting point and start tracing the contour and extracting the edge until the edge is closed or falls below a given gradient intensity threshold.

The contour tracing process uses the gradient and the second directional derivative to reconstruct the edges. Both processes also require a gradient intensity threshold parameter to discriminate the useful edges from noise. In practice, instead of using a fixed value for this parameter, an adaptive thresholding technique is preferred. In this approach, a histogram of the magnitude of the gradient is used to choose the threshold such that there is a fixed number of pixel having gradient intensities less than this threshold. This method provides good results even when the contrast of the original image was not previously known.

The rest of this paper will focus on accelerating the first stage of the process using FPGA Technology.

III. Using FPGA Technology for Acceleration

As it can be seen in the previous section, the edge extraction process requires the following pieces of information in order to produce the list of edges:

- The gradient of the image (along the X and Y axes)
- The second directional derivative along the direction of the gradient
- The histogram of the magnitude of the gradient

A solution that computes all this information using FPGA resources is presented in the following sections.

A. Gradient Computation

The derivatives along the X and Y axes are computed using the Sobel kernel. This implies perfoming a convolution between the original image and the 3x3 kernels shown in equations (5) and (6). For this, a 3x3 pixel neighborhood of the current pixel has to be obtained. Issuing 9 read operations for each pixel would imply a large time penalty. Instead, a pipeline approach is used.

Assume that the input image has a horizontal resolution of *hres* pixels. In this case, let ϕ be the phase (ordinal number) of the current pixel. Then, the neighbourhood is given by the following phase matrix:

$$\begin{pmatrix} \phi - hres - 1 & \phi - hres & \phi - hres + 1 \\ \phi - 1 & \phi & \phi + 1 \\ \phi + hres - 1 & \phi + hres & \phi + hres + 1 \end{pmatrix} \qquad (7)$$

As it can seen, the phase spans over a $2 \cdot hres + 3$ length interval, and thus this would be the length required by a pipeline implementing this kernel (see Figure 1). Each of the FIFO memories in this figure has a capacity of $hres - 3$, which, in the case of a 1024 horizontal resolution, means a pipeline of 1021 registers for just one line. The cost is obviously too high, thus an implementation based on

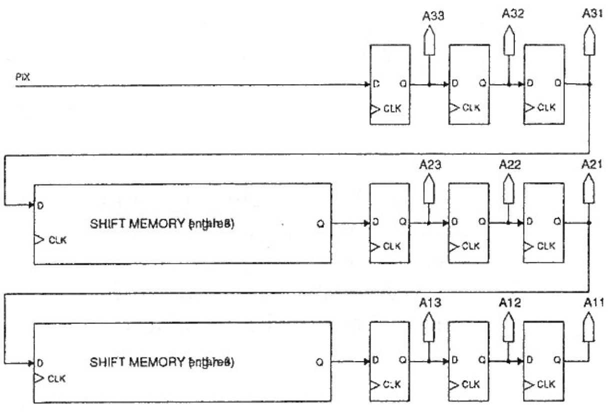

Fig. 1. Generating a 3x3 pixel neighbourhood

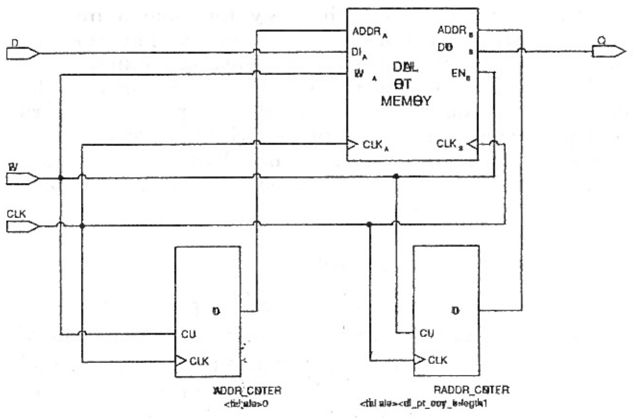

Fig. 2. A shift memory of a given depth using a dual-port memory

dual-port memory modules is needed. Such an implementation is presented in Figure 2. The initial values of the address counters are computed in such a way that their phase difference is always equal to the length of the shift memory. The two counters can be viewed as two pointers moving around in a circular fashion, progressing one step at each write enable signal. After the neighborhood has been obtained, the application of the Sobel kernel is a straightforward. The particular nature of this kernel allows for a much cheaper implementation than using a generic convolution engine (for such an engine, please consult [5]). First of all, it involves only multiplication by 2 and division by 8, which can be performed using shifting operations. Also, 3 of the coefficients are null, thus resulting in a waste of resources in the case of a generic convolution approach. The computation will be achieved using 4 stages, as it can be seen in figure 3.

B. Second Directional Derivative Computation

Assume that all the first and second order derivative terms have been already computed using the Sobel

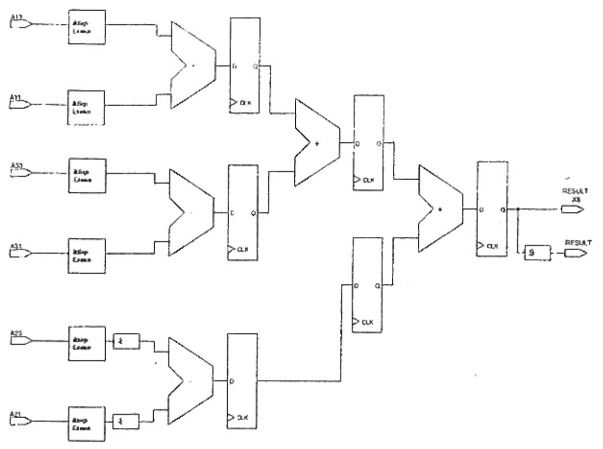

Fig. 3. Pipeline implementation of the convolution with Sobel kernel

derivative kernel presented before. Let us denote them by **dx**,**dy**, **dxx**,**dxy** and **dyy**. In order to compute the fraction appearing in equation (4), one has to perform 6 multiplications and 1 final division, as it is outlined by algorithm 1:

```
Stage 1;
$A \leftarrow dx \cdot dx$;
$B \leftarrow dy \cdot dy$;
$C \leftarrow dx \cdot dy$;
Stage 2;
$D \leftarrow A \cdot dxx$;
$E \leftarrow B \cdot dyy$;
$F \leftarrow C \cdot dxy$;
Stage 3;
$M \leftarrow D + E$;
$N \leftarrow F << 1$;
$Q \leftarrow A + B$;
Stage 4;
$P \leftarrow M + N$;
Stage 5;
$R \leftarrow \frac{P}{Q}$;
```

Algorithm 1: Second order directional derivative computation

Unfortunately, multiplication and division methods for signed integers, that is, represented in two's complement, are not well suited for pipeline implementation and are generally expensive. The multiplier and divider that are used in this work are unsigned, and thus take as input only the magnitude of the arguments (signs are dealt with separately). This involves certain conversions between two's complement and sign-and-magnitude, resulting in the insertion of 4 intermediary stages. The resulting pipeline for

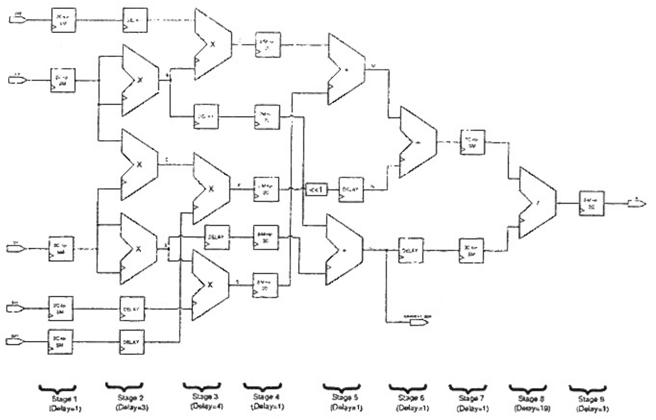

Fig. 4. The pipeline for computing the second order directional derivative

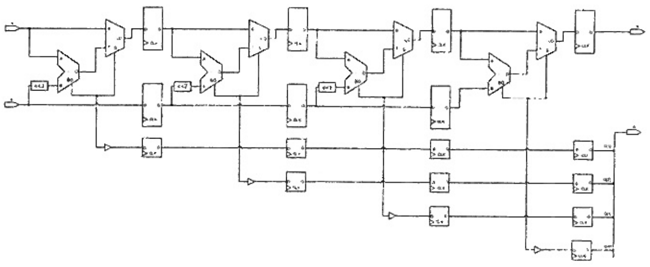

Fig. 5. Pipeline divider for a 4-bit quotient

second directional derivative computation is presented in figure 4. The multiplication modules used in this design contain binary trees of adders with output registers that add up the partial products. The multiplication component is configured by two generics, **width_a** and **width_b** giving the size of the two operands. The total time for performing a multiplication is $\log_2 width\_b$ clock cycles.

Implementation issue. As it can be seen, the structure and size of the tree are dependent on the width of the second operand (given by **width_b**). In order to generate such a tree, the GENERATE capability of VHDL has been extensively used. The tree has been represented as a vector of values, the parents of node i being stored at locations $2 \cdot i$ and $2 \cdot i + 1$. The resulting code is synthesizeable with the latest version of XST (Xilinx Synthesis Tool, see [6]).

For division, the array divider proposed in [7] has been used. It is based on repeated subtraction of the divider B from the dividend A. If, at a subtraction, the result is negative, rather than being restored by addition, it is multiplexed with the original value. Figure 5 presents such a divider for a 4-bit quotient.

C. Computing the Magnitude of the Gradient

The only item that remains to be computed is the histogram of the magnitude of the gradient, i.e, the expression $\sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2}$. As it can be noticed from figure 4, in stage 6, the expression $\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2$ has already been computed (it has been output onto the pin **GRADIENT_SQRT**). It follows that all that has to be done is to take the square root of this value in order to obtain the final result.

Unfortunately, the expression under the square root has a magnitude of $2 \cdot 128^2 = 32,768$, thus requiring 15 bits to represent. Extracting the square root of such a big number cannot be achieved by a simple LUT. Such a LUT would have 32Kbytes and could not be accommodated by the resources found in a VirtexE device.

Fortunately, there are other solutions to extracting the square root of a number. One of them, using Newton's method, is based on an approach similar to that of the divider, but a pipeline implementation would still be very costly due to the large number of stages. Another method for computing the square root has been used, one that is based on LUTs, but uses some tricks to drastically minimize the LUT space needed.

First, it should be noted that it is the integer part of the square root that has to be computed (more exactly, the largest integer smaller that the square root). Let us take the derivative of the square root function:

$$(\sqrt{x})' = \frac{1}{2 \cdot \sqrt{x}} \tag{8}$$

One can see that the derivative of the square root function is a uniformly decreasing function, approaching 0 as x grows to infinity:

$$\lim_{x \to \infty} \frac{1}{2 \cdot \sqrt{x}} = 0 \tag{9}$$

Let us consider $x = 256$. By replacing into equation 8, one gets:

$$(\sqrt{x})'(256) = \frac{1}{32} \tag{10}$$

And since the derivative of the square root is a decreasing function, we can extend this to:

$$(\sqrt{x})' \leq \frac{1}{32}, \forall x > 256 \tag{11}$$

But since it is the integer part of the square root we are interested in, it is obvious that, as the slope of this curve decreases, more and more values are mapped onto the same output value. In fact, the condition $\sqrt{x} \leq \frac{1}{k}$ guarantees that for any interval $[a, a+k-1]$, there will be at most one "step" in the integer part of the function (since the function can only increase with 1 in this interval). In particular, it is known that that:

$$\forall a \geq 256, \forall x \in [a, a+31], \sqrt{x} - \sqrt{a} \leq 1, \tag{12}$$

Let us now divide the interval [256,8192] into smaller intervals of length 32, starting from 256 onwards. Consider that we have a precomputed LUT that maps each value $\lfloor \frac{x}{32} \rfloor$ to $\lfloor \sqrt{x} \rfloor$. Then, by taking $a = 32 \cdot \lfloor \frac{x}{32} \rfloor$ and observing that $x \in [a, a+32]$, it follows from equation 12 that the maximum error of the result is 1.

It has already been stated that there is at most one "step" in any 32-length interval for all $x > 256$. Consider that one also has a precomputed LUT that memorizes the displacement of this value (that is, it stores the smallest value of the expression $x \ mod \ 32$ for which the error in the result becomes 1). In this case, a correction would be possible by incrementing the value of the result and finally, we would end up with the exact value of the square root.

A similar approach can be taken for x greater than 8192. The final algorithm is presented below:

```
if $x < 256$ then
    $r \leftarrow LUTV_1(x)$;
else
    if $x < 8192$ then
        $r \leftarrow LUTV_2(x/32)$;
        if $x \ mod \ 32 \geq LUTC_2(x/32)$ then
            $x \leftarrow x + 1$;
        end
    else
        $r \leftarrow LUTV_3(x/128)$;
        if $x \ mod \ 128 \geq LUTC_3(x/128)$ then
            $x \leftarrow x + 1$;
        end
    end
end
```

Algorithm 2: The square root extraction algorithm

$LUTV_1$, $LUTV_2$ and $LUTV_3$ represent the value lookup tables, each mapping $x, \lfloor \frac{x}{32} \rfloor$ and $\lfloor \frac{x}{128} \rfloor$ respectively to their square root values. $LUTC_2$ and $LUTC_3$ are the correction look-up tables, each mapping the values of $\lfloor \frac{x}{32} \rfloor$ and $\lfloor \frac{x}{128} \rfloor$ to their error correction offset (see figure 6 for a graphical representation). These tables are generated by software using the following formulas:

$$LUTV_2[x] = \lfloor \sqrt{32 \cdot x} \rfloor \tag{13}$$

$$LUTC_2[x] = (LUTV_2[x] + 1)^2 - 32 \cdot x \tag{14}$$

$$LUTV_3[x] = \lfloor \sqrt{128 \cdot x} \rfloor \tag{15}$$

$$LUTC_3[x] = (LUTV_3[x] + 1)^2 - 128 \cdot x \tag{16}$$

The **if** statements in the presented algorithm can be processed in parallel, resulting in a 3-clock pipeline for calculating the square root. The design is presented in Figure 7.

D. Histogram Computation

Histogram computation involves three major steps, for each pixel p arriving at the input:

1) $cnt \leftarrow hist[p]$
2) $cnt \leftarrow cnt + 1$
3) $hist[p] \leftarrow cnt$

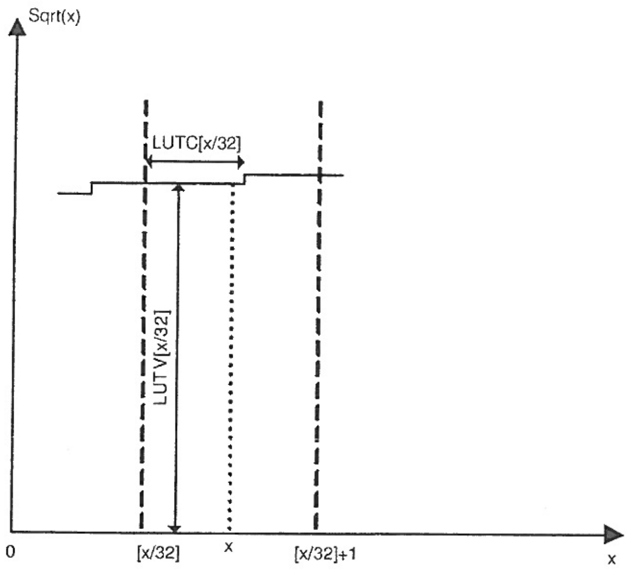

Fig. 6. A graphical representation of LUTC and LUTV.

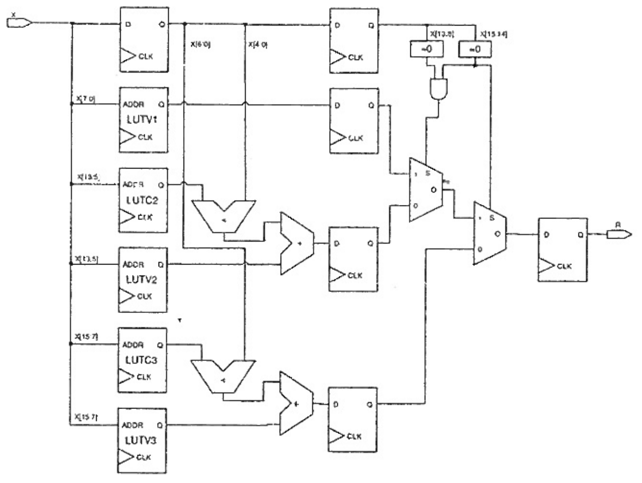

Fig. 7. Square Root Computation Pipeline

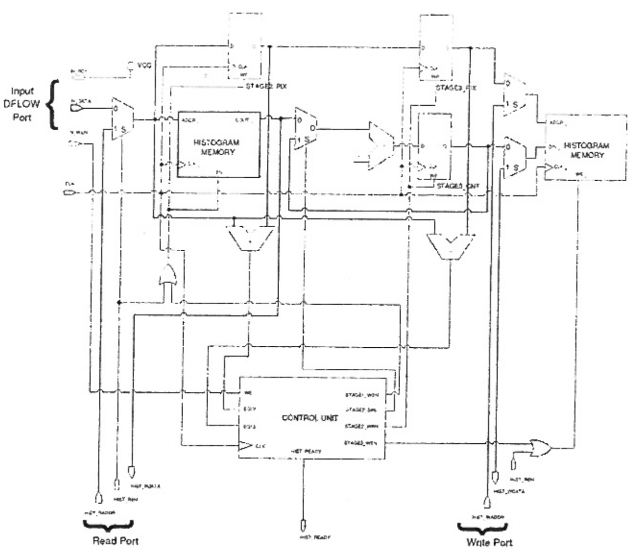

Fig. 8. The histogram computation pipeline

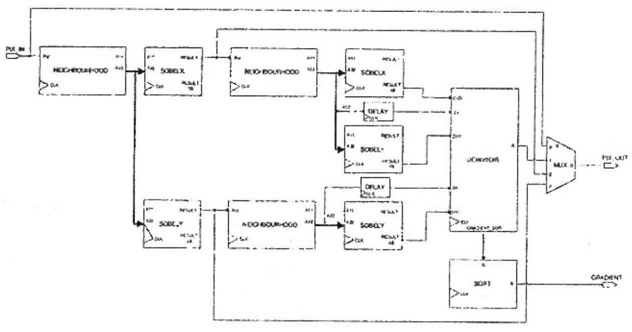

Fig. 9. Datapath needed to control the arithmetic pipeline

Since the rest of the design has a 1 clock cycle/pixel performance, a pipeline approach must be undertaken for implementing this module as well. This leads to a read and a write operation in each clock cycle, requiring a dual-port memory for storing the histogram. Two read after write (RAW) conflicts can appear in the pipeline, for the pixel patterns [**x x**] and [**x y x**]. The first conflict is handled using a forwarding technique and the second using bypassing. Conflict detection is performed by the two comparators (see figure 8).

E. Top-Level Design

Using the components that have been designed in the previous sections, the overall structure of the design has been presented in figure 9. Several delay elements are required to assure that all data is present at the right place at the right time. A control unit (not shown in the figure) is also present in order to assure proper loading/flushing of the pipeline.

IV. Experimental results

The design that has been presented above has been implemented and tested on a VirtexE600 device ([8]). The observed performance at an 80Mhz frequency is 3.2ms/frame. Unfortunately, due to the high communication needs, this computational speed remains unused, since the PCI interface imposed an 80Mb/s communication bottleneck. A Pentium 4 computer operating at 2.66Ghz requires 20ms/frame to perform the same computation. It should also be noted that the Virtex device that has been used has been produced in the year 1999, and FPGA technology has significantly evolved since then.

V. Conclusions and Future Work

A hardware solution that accelerates the edge detection process using a subpixel level method has been presented. Even though the performance of the design is much higher than than that of a general purpose computer, the high amount of data that must be transferred to the software environment for the second stage of edge detection drastically limits the framerate of the system. It is thus desirable to investigate a method to move the second part of processing into the hardware design, such that this transfer penalty is no longer incurred. This will be the subject of future research.

References

[1] A. Huertas and G. Medioni, "Detection of intensity changes with subpixel accuracy using laplacian-gaussian masks," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1986.

[2] P. Grattoni and A. Guiducci, "Contour coding for image description," *Pattern Recognition Letters*, vol. 11, no. 2, pp. 95–105, 1990.

[3] V. Torre and T. A. Poggio, "On edge detection, ieee transactions on pattern analysis and machine intelligence," vol. 8, no. 2, pp. 147–163, 1986.

[4] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679–698, 1986.

[5] S. Nedevschi, P. Samways, M. Marian, and T. Hall, "Real-time, single-chip, generalized convolution device for image processing," *A.C.A.M. Sci. Journal*, vol. 5, no. 1, pp. 11–22, 1996.

[6] *Xilinx ISE 6 Software Manuals*, Xilinx Inc., 2003.

[7] B. Z. Francisc, *Structure of Computer Systems.* U.T. Pres Cluj-Napoca, 2002.

[8] *Virtex-E 1.8 VField Programmable Gate Arrays, Product Specification*, Xilinx Inc., 2001.