TECHNICAL REPORT: DESIGN AND SIMULATION OF REAL-TIME IMAGE PROCESSING ALGORITHMS

ABSTRACT
Image processing algorithms, such as average filtering, median filtering, convolution and edge detection are used frequently in many different imaging applications. Implementation of these algorithms in software has been widely studied. However, many imaging systems are required to operate in real-time which is difficult to achieve using software. Hardware solutions, such as FPGAs and ASICs, offer real-time implementations of complex imaging systems. Real-time performance in hardware is achieved by re-designing algorithms to take advantage of parallelism and pipelining. This paper describes the process of adapting common image processing algorithms for hardware implementation (resulting in a hardware image processing library). Comparisons of the algorithms using software implementations and hardware simulations are performed.

KEY WORDS
Parallel Processing, Modelling and Simulation, Digital Image Processing Algorithms.

1 Introduction

Digital image processing is prevalent throughout society and used in a wide range of scientific fields (*e.g.*, medicine, security, telecommunications, manufacturing, military, *etc.*). It is often necessary for image processing applications to operate in real-time (*e.g.*, video surveillance, target recognition, image enhancement, *etc.*).

Using a general purpose Personal Computer (PC) to implement real-time image processing algorithms poses a number of problems: (*i*) PCs are relatively costly, (*ii*) they have a high power consumption and (*iii*) they are large and heavy (making them impractical for some applications) [1]. Also, the inherent sequential nature of a PC means that algorithms are implemented in a serial fashion (which is considerably slower than a parallel implementation). On the other hand, hardware solutions (such as ASICs and FPGAs) do not suffer the same limitations as PCs when performing real-time image processing.

1.1 ASICs

An alternative to a PC would be to use an Application Specific Integrated Circuit (ASIC); hardware chips designed for a specific application [2]. ASICs are cheap to mass produce, they have low power consumption and they are small in size. Algorithms designed for ASICs are implemented in parallel and thus run extremely quickly. The drawbacks of ASICs are: (*i*) their initial development costs are high and (*ii*) their designs cannot be altered after fabrication (if a design is to be altered, the ASIC would need to be manufactured again).

1.2 FPGAs

Field Programmable Gate Arrays (FPGA) represent a compromise between PCs and ASICs. Like ASICs, FPGAs are a hardware solution. However, unlike ASICs, FPGAs consist of an array of routing resources that can be *reprogrammed* after manufacture [3]. FPGAs have a number of advantages over ASICs: (*i*) they have a shorter time to market, (*ii*) they have lower development costs and (*iii*) they have the ability to be reprogrammed after manufacture [4]. However, FPGAs tend to draw more power and run slightly slower than ASICs. This gives FPGAs the flexibility of a PC software solution (reprogramming) whilst maintaining the performance advantage of an ASIC solution. For these reasons, as well as their relatively low cost, real-time image processing with FPGAs is an active research field [5, 6, 7].

Unfortunately, both ASICs and FPGAs have a low-level programming model [8]. Thus, designing an image processing application for the first time using an ASIC or a FPGA is a time consuming and difficult task. Having a pre-written hardware library of common image processing algorithms would dramatically reduce both the time needed and the difficulty of creating an image processing application in hardware.

1.3 Hardware image processing library

This paper presents a library of common image processing algorithms written in the widely used Verilog Hardware Description Language [9] and tested in a ModelSim [10] hardware simulator. This hardware library has been designed to be **efficient** and **reusable**. The algorithms descibed here are not unique to Verilog and could easily be ported to a different hardware description language.

The **efficiency** of the image processing algorithms comes from exploiting parallelism and pipelining. Parallelism indicates that many non-conflicting operations of an algorithm are performed at the same time (in a single clock cycle). Pipelining indicates that multiple operations are overlapped in execution; after an initial startup delay, an output is produced for every clock cycle.

The **reusability** of the library is achieved by including image processing algorithms that are used in many different image processing applications. For example, consider the Gaussian smoothing filter. This filter is widely used (for noise removal) as a pre-processing step in many imaging applications (*e.g.* Canny edge detector [11]). Also, the code in the hardware library should be reusable to a large extent. For example, by slightly altering the internal design of the Gaussian smoothing filter, it can easily be converted to a different convolution algorithm (*e.g.* Laplacian sharpening).

The hardware image processing library described in this paper consists of algorithms (like the Gaussian smoothing filter) that are widely used and reusable. The library has been designed to exploit parallelism and pipelining in order to implement algorithms efficiently in a hardware environment. The algorithms have been written in Verilog and verified using ModelSim. Using vendor tools, it would be possible to synthesize the Verilog code into a bitstream for implementation in an FPGA or even target the creation of ASIC.

2 Image processing algorithms

Here, four common image processing algorithms are described: (*i*) average filter, (*ii*) median filter, (*iii*) convolution and (*iv*) Canny edge detector.

These use kernel operations, where a window of a certain size (*e.g.*, 3×3, 5×5, *etc.*) is slid over an image. The centre pixel value of the sliding window is replaced with a new value based on the values of its neighbours as well as its own value. In software, a sliding window operation is commonly implemented via a nested *for* loop as shown in the following pseudo C-code. Figure 1 shows an example of a 3×3 sliding window.

```
// pseudo C-code for nested for loop
for (r=0; r<rows; r++){
    for (c=0; c<cols; c++){
        kernelArray = pixels around image(r,c);
        image(r,c) = average(kernelArray);
    }
}
```

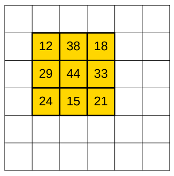

Figure 1. Sliding 3×3 window, centre pixel value of 44

2.1 Average filter

The average filter [11] computes the average value of the pixels in a sliding window. In Figure 1, the original central pixel value of 44 would get replaced by $(12 + 38 + 18 + 29 + 44 + 33 + 24 + 15 + 21)/9 = 234/9 = 26$.

2.2 Median filter

The median filter [11] ranks (sorts) the values in a sliding window according to their brightness. This type of filter is good at removing *salt and pepper* noise. In Figure 1, the values in the window would get sorted: 12 15 18 21 <u>24</u> 29 33 38 44. The median value of this sorted list is 24. Thus, the median filter would replace the original central pixel value of 44 with 24.

2.3 Convolution

Convolution [11] calculates the weighted sum of pixels in a sliding window. A convolution mask, such as a Gaussian smoothing mask shown in Figure 2, can be used for this process. At corresponding locations, convolution multiplies a value in the sliding window (Figure 1) with a value in the convolution mask (Figure 2). In the example given: $[(12 \times 21) + (38 \times 31) + (18 \times 21) + (29 \times 31) + (44 \times 48) + (33 \times 31) + (24 \times 21) + (15 \times 31) + (21 \times 21)]/256 = 7252/256 = 28.33$. Thus, convolution would replace the original central pixel value of 44 with 28.33. Convolution is an extremely versatile operation; by changing the weights of a convolution mask, it is possible to create completely different filters (*e.g.*, Laplacian sharpening filter, Sobel edge masks, *etc.*). It is also common for a convolution mask to consist of signed numbers.

$$\frac{1}{256}\begin{array}{|c|c|c|}\hline 21 & 31 & 21 \\ \hline 31 & 48 & 31 \\ \hline 21 & 31 & 21 \\ \hline\end{array}$$

Figure 2. Gaussian 3×3 smoothing convolution mask

2.4 Canny edge detection

The Canny edge detection algorithm [11, 12] is a widely used algorithm in imaging applications as it gives good edge detection, good localisation and produces binary images with thin edges. There are four main stages to the Canny algorithm; these are shown in Figure 3. Initially,

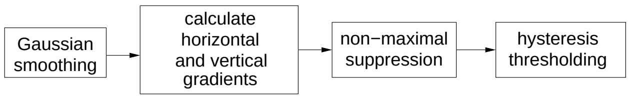

Figure 3. Design-flow of Canny edge detection algorithm

a Gaussian smoothing operation is performed. Next, horizontal (*dx*) and vertical (*dy*) gradients are calculated for each pixel by differentiating the smoothed image in the *dx* and *dy* directions.

Next, directional non-maximal suppression is performed. The hypotenuse and arctangent of the *dx* and *dy* gradients are used to calculate the magnitudes and phases of each gradient. The phases are quantized into one of four orientations. Once the orientation of the gradient is known, the magnitudes of the pixels in the neighbourhood of the pixel under examination are interpolated. If the magnitude of the pixel under examination is less than either of the interpolated values, it is eliminated as a non-maximum gradient.

Finally, hysteresis thresholding is performed to eliminate broken edges and single edge points. A low and a high threshold are chosen by the user and a 3×3 sliding window is applied recursively to find definite edges.

3 Adaptation of algorithms for hardware

The following sections explain how the algorithms in the image processing library were designed for hardware implementation (taking advantage of parallelism and pipling-ing). The algorithms were coded in Verilog [9] and tested using ModelSim [10]. Initially, a testbench module was written that read a greyscale image into memory. The testbench then passed this image, in rasta order one pixel at a time, to the sliding window module (described in Section 3.1).

3.1 Sliding window operation

One of the most important aspects of implementing image processing algorithms in hardware is the design of a sliding window module. As timing is critical in hardware, it is prudent to update the values of a sliding window on every clock cycle (*i.e.*, a pipelined design).

Figure 4 shows the design of a sliding 3×3 window that generates a new set of window values on every clock cycle (after an initial start up latency). Two FIFOs (First

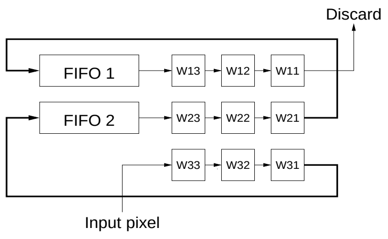

Figure 4. Hardware design of a sliding 3×3 window

In First Out) are required for this operation. A FIFO is a type of memory that is used to buffer data; Verilog code for different types of FIFOs can be found in [13]. In Figure 4, the widths of both FIFOs are $\mathit{ImageWidth} - 3$.

In Figure 4, an image pixel is inserted into register W33. It is then propagated through registers W32 and W31 and then into FIFO2. From FIFO2, it is propagated through registers W23, W22 and W21 and then into FIFO1. From FIFO1, it is propagated through registers W13, W12 and W11 and then it is no longer needed. Once W11 has received its first image pixel, the startup period is over and the values in the registers W11, W12, W13, W21, W22, W23, W31, W32 and W33 represent the sliding window; these values can then be transferred to other modules for further processing (*e.g.*, filtering, convolution, *etc.*). Figure 5 shows a similar design for implementing a 5×5 sliding window; the widths of all four FIFOs are $\mathit{ImageWidth} - 5$.

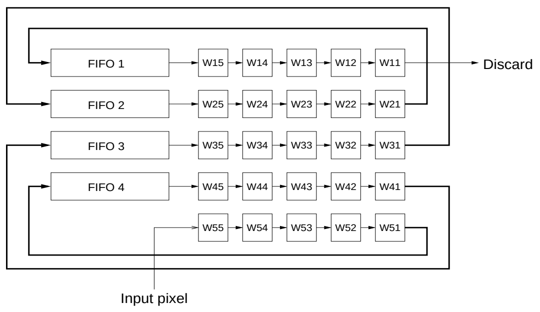

Figure 5. Hardware design of a sliding 5×5 window

3.2 Hardware average filter

An average filter module was designed to read the values of the nine sliding window registers from Section 3.1. In hardware, it is not prudent to simply add all these values together and then divide by nine. This is due to the longest-path delay. The hardware clock speed will *at most* be equal to the path with the longest delay, so hardware designers should attempt to reduce this delay (this can be achieved by reducing complexity). An example of this is given below for an implementation of an average filter using pseudo Verilog code. This code finds a total by adding all the sliding window values together (from a 3×3 window) and then divides this total by eight to compute an estimate of the average; it does this in five distinct steps.

```
// pseudo Verilog code for an average filter
output reg [7:0] average;           // 8 bits
reg [8:0] r10, r11, r12, r13, r14; // 9 bits
reg [9:0] r20, r21, r22;            // 10 bits
reg [10:0] r30, r31;                // 11 bits
reg [11:0] total;                   // 12 bits

// w11, w12 ... w33 are values from the sliding window
r10 <= w11 + w12;        // 1st clock cycle
r11 <= w13 + w21;        // 1st clock cycle
r12 <= w22 + w23;        // 1st clock cycle
r13 <= w31 + w32;        // 1st clock cycle
r14 <= w33;              // 1st clock cycle

r20 <= r10 + r11;        // 2nd clock cycle
r21 <= r12 + r13;        // 2nd clock cycle
r22 <= r14;              // 2nd clock cycle

r30 <= r20 + r21;        // 3rd clock cycle
r31 <= r22;              // 3rd clock cycle

total <= r30 + r31;      // 4th clock cycle

average <= (total >> 3); // 5th clock cycle
```

It can be seen from the pseudo Verilog code that there are no long and complex expressions (thus minimizing the longest-path delay). Initially, registers *r10, r11, r12, r13* and *r14* are all assigned (at the same time) on the first clock cycle. This is achieved using Verilog's non-blocking assignment (<=) which synchronizes assignments so they occur **at the same time** on a clock edge (*i.e.*, operate in parallel). Registers *r20, r21* and *r22* are all assigned (at the same time) on the second clock cycle, registers *r30* and *r31* are both assigned (at the same time) on the third clock cycle. The summation of all the sliding window values, *total*, is assigned on the fourth clock cycle and their *average* is assigned on the fifth clock cycle. In this example, the average value will be valid five clock cycles after it has received the sliding window values. Flags are used to indicate when an output from a module is valid. This final *average* value is calculated via right-shifting *total* by 3 (divide by 8). Although this should be a divide by 9, division is an expensive operation in hardware so a compromise is made via right-shifting by 3 instead[1] (shifting is an inexpensive operation in hardware). A programmer should always consider adapting an algorithm to use more efficient computations when designing for hardware.

The average filter design is pipelined; after an initial latency of five clock cycles, an average value is produced on **every clock cycle**. The registers are different widths to account for possible carry bits during addition (the values from the sliding window registers are eight bits wide).

[1]the full Verilog code has checks to make sure values greater than 255 do not occur for *average*

3.3 Hardware median filter

A median filter module was designed to read the values of the nine sliding window registers from Section 3.1. An odd-even transposition sort algorithm [14] was implemented using two Verilog modules (an odd sort module and an even sort module). The odd-even transposition sort algorithm essentially runs many bubble sort [15] computations in parallel; it can sort a 1-D array of *N* values after *N* applications. An example of the odd-even transposition sorting algorithm is given in Figure 6. In this example, the

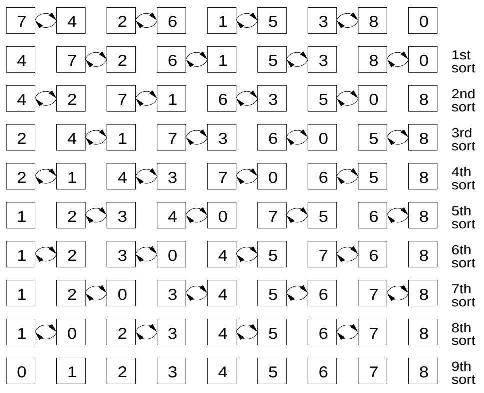

Figure 6. Odd-even transposition sorting algorithm

un-ordered array of nine values (7 4 2 6 1 5 3 8 0) is sorted into ascending order after nine applications. Comparators, indicated by circular arrows in Figure 6, are used in parallel to compare two values; the lower of these values is moved to the left and the higher is moved to the right. This can be achieved via the following pseudo Verilog code:

```
// pseudo Verilog code for a comparator
if regLevelONE_1 < regLevelONE_2 begin
    regLevelTWO_1 <= regLevelONE_1;
    regLevelTWO_2 <= regLevelONE_2;
end else begin
    regLevelTWO_1 <= regLevelONE_2;
    regLevelTWO_2 <= regLevelONE_1;
end
```

The design in Figure 6 is pipelined. For example, the values from a sliding window (*A*) are received and undergo their first sort resulting in $A^{1^{st}}_{sort}$. On the next clock cycle, the values in $A^{1^{st}}_{sort}$ are sorted again (resulting in $A^{2^{nd}}_{sort}$) while **at the same time** the values from the next sliding window (*A+1*) are received and undergo their first sort. Once the initial start-up latency of nine clock cycles has passed, a sorted listed will be produced on **every clock cycle**. The median of this sorted list is selected and is returned as the new pixel value. Morphological operations of greyscale erosion and dilation can also be computed using sorted lists. Grayscale erosion returns the minimum value from a sorted list and greyscale dilation returns the maximum value from a sorted list. Thus, with a small alteration, the code for the median filter can be altered to implement greyscale erosion and dilation.

3.4 Hardware convolution

A convolution module was designed to read the values of the nine sliding window registers from Section 3.1. Unlike the average and median filters, it is necessary for the convolution module to handle signed numbers (as many convolution masks have negative values). Compared to average and median filtering, convolution is a complex operation involving the use of two's complement arithmetic to represent signed numbers. Convolution is performed by multiplying the values in a sliding window (w_{xx}) by a constant value (k_x) from a convolution mask. The results of these multiplications are summed together and the final summed value is divided (using a shift-right calculation) to give a final convolution value to be returned. This process is shown diagramatically in Figure 7. Pseudo Verilog code is shown

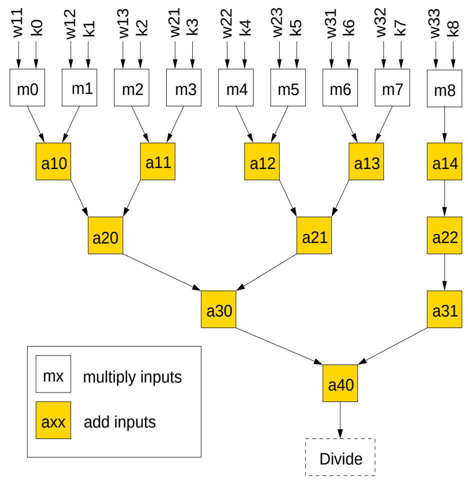

Figure 7. Convolution algorithm

for implementing hardware convolution. The pseudo Verilog code operates on 8 bit unsigned sliding window registers (w_{xx}). The convolution mask registers (k_x) are 9 bits wide (8 bits for the value, 1 bit for the sign). The multiplication registers (m_x) are 17 bits wide (16 bits to store the result of the multiplication and 1 bit for the sign). The addition registers (a_{xx}) are 18, 19, 20 and 21 bits wide; these registers are wider because as addition continues, larger values have to be stored.

Multiplication of the 8 bit unsigned sliding window registers (w_{xx}) with the convolution mask registers (k_x) initially alters the w_{xx} registers to be 9 bit signed (positive) values. It does this using the Verilog concatenation operator `{1'b0,w11}` to prefix a binary value of 0 to the value of w_{xx}. Signed multiplation of the w_{xx} and k_x registers can then be performed using the Verilog `$signed()` operator.

```
// Pseudo Verilog code for convolution
// convolution mask values 9 bits (signed)
reg signed [8:0] k0, k1, k2, k3, k4, k5, k6, k7, k8;

// multiplication results 17 bits (signed)
reg signed [16:0] m0, m1, m2, m3, m4, m5, m6, m7, m8;

// addition 18, 19, 20 and 21 bits (signed)
reg signed [16+1:0] a10, a11, a12, a13, a14;  // 18 bits
reg signed [16+2:0] a20, a21, a22;            // 19 bits
reg signed [16+3:0] a30, a31;                 // 20 bits
reg signed [16+4:0] a40;                      // 21 bits

m0 <= $signed({1'b0,w11}) * $signed(k0);      // 1st clk
m1 <= $signed({1'b0,w12}) * $signed(k1);      // 1st clk
m2 <= $signed({1'b0,w13}) * $signed(k2);      // 1st clk
m3 <= $signed({1'b0,w21}) * $signed(k3);      // 1st clk
m4 <= $signed({1'b0,w22}) * $signed(k4);      // 1st clk
m5 <= $signed({1'b0,w23}) * $signed(k5);      // 1st clk
m6 <= $signed({1'b0,w31}) * $signed(k6);      // 1st clk
m7 <= $signed({1'b0,w32}) * $signed(k7);      // 1st clk
m8 <= $signed({1'b0,w33}) * $signed(k8);      // 1st clk

a10 <= $signed({m0[16],m0}) + $signed(m1);    // 2nd clk
a11 <= $signed({m2[16],m2}) + $signed(m3);    // 2nd clk
a12 <= $signed({m4[16],m4}) + $signed(m5);    // 2nd clk
a13 <= $signed({m6[16],m6}) + $signed(m7);    // 2nd clk
a14 <= $signed({m8[16],m8});                  // 2nd clk

a20 <= $signed({a10[17],a10}) + $signed(a11); // 3rd clk
a21 <= $signed({a12[17],a12}) + $signed(a13); // 3rd clk
a22 <= $signed({a14[17],a14});                // 3rd clk

a30 <= $signed({a20[18],a20}) + $signed(a21); // 4th clk
a31 <= $signed({a22[18],a22});                // 4th clk

a40 <= $signed({a30[19],a30}) + $signed(a31); // 5th clk

convolutionResult <= $unsigned(a40 >> 8);     // 6th clk
```

The addition calculations also use the concatenation operator. For example, `a14 <= $signed({m8[16],m8})` concatenates the most significant bit of m_8 to itself[2] (on the left-hand side). This ensures that the 18 bit a_{14} register has the correct sign value (0 for positive and 1 for negative) when it is assigned the value of the 17 bit m_8 register. This concatenation is required otherwise the most significant bit of a_{14} would automatically get set to zero by Verilog (correct for positive values, incorrect for negative values). The addition continues so that, after a latency of six clock cycles, a result for the convolution operation is obtained on **every clock cycle** (*i.e.*, pipelined). Parallelism is achieved by performing numerous simple calculations at the same time (calculations are kept simple to minimize the longest-path delay and thus maximize the hardware clock speed).

3.5 Hardware Canny edge detection

The Canny edge detector is much more complex than the average filter, median filter and convolution; an in-depth

[2]Verilog has a syntax similar to the C programming language, thus arrays start counting from 0

explaination [3, 12, 16] of its implementation in hardware is beyond the scope of this paper. Instead, a general overview of how it was implemented in hardware is given here. Its implementation utilises the hardware techniques of pipelining, parallelism and minimizing the longest-path delay (used previously in Sections 3.2, 3.3 and 3.4).

The first stage of the Canny edge detector is to implement the convolution operation of **Gaussian smoothing** upon the input image. The next stage is to **calculate the gradient in the *x* and *y* directions**. These two operations can be performed simultaneously [3] by using an 11×11 sliding window to perform Gaussian derivative smoothing; this smooths the image and at the same time calculates the gradients (of the smoothed image) in the *x* and *y* directions.

Next, **non-maximal suppression** is performed by first calculating the magnitude and phase (orientation) of each pixel in the *x* and *y* directions. Calculating the magnitude of the pixels in the *x* and *y* directions is straightforward, however, calculating the phase of the pixels requires use of the *arctan* function. This function is difficult to implement in hardware, so an effective but simplier calculation is used based upon: (*i*) the magnitudes of the *x* and *y* gradients and (*ii*) the sign of the *x* and *y* gradients. This produces a quantized orientation[3] in one of four directions. Once the orientation of the gradient is known for a pixel, the magnitudes of its immediate neighbours *along that orientation* are interpolated (this is performed using a 3×3 sliding window). If the magnitude of the pixel under consideration is less than either of its interpolated neighbours, it is suppressed. Otherwise, the pixel under consideration is marked as a local maximum gradient.

The final stage of the Canny edge detection algorithm is **hysteresis thresholding**. The edge image produced from the previous non-maximal suppression step may consist of single edge points and broken edges. These can be eliminated by hysteresis thresholding. Hysteresis thresholding can be implemented easily in hardware via a 3×3 sliding window and two user defined thresholds (th_{low} and th_{high}). The 3×3 window is slid over the gradient magnitude image (from the non-maximal suppression step); using the pixels in the window and the two user defined thresholds, a comparison is made that decides whether a possible edge is kept or removed. This produces an image with very sharp and thin edges [16]. By applying hysteresis thresholding numerous times (recursively), it is possible to *grow* the edges and thus improve the quality of the edge image.

As with the average filter, median filer and convolution, the four stages of the Canny algorithm were each designed with a high level of parallelism and pipelining. Care was taken to break complex calculations into numerous simple calculations in order to minimize the longest-path delay. Much code from Sections 3.2, 3.3 and 3.4 was reused in the design of the Canny edge detector.

[3]orientations based on compass points: 1: NNW-SSE, 2: WNW-ESE, 3: NNE-SSW and 4: WSW-ENE

4 Results

The algorithms (average filter, median filter, convolution and Canny edge detector) were written in software (C-code) and in hardware (Verilog). ModelSim [10] was used to verify the correct performance of the Verilog code via simulation. The C-code was run on a Pentium 4 PC with a clock speed of 3000MHz and 1000MB of RAM. The times taken to execute C and Verilog implementations of each algorithm (using greyscale Lena images of 256×256 and 512×512) are shown in Table 1. Note that the simulated results for the Verilog code in ModelSim used a clock period of 20 nano-seconds; this equates to a hardware clock running at 50MHz in a FPGA. Faster times are possible by reducing this clock period, but this would put more emphasis on minimizing the longest-path delay.

Lena 256×256	
Algorithm	Time to process
average 3×3 (C-code)	0.008818371 seconds
median 3×3 (C-code)	0.036465949 seconds
convolution 3×3 (C-code)	0.018986322 seconds
Canny 11×11 (C-code)	0.041033778 seconds
average 3×3 (Verilog)	0.001310830 seconds
median 3×3 (Verilog)	0.001311050 seconds
convolution 3×3 (Verilog)	0.001310870 seconds
Canny 11×11 (Verilog)	0.001382990 seconds

Lena 512×512	
Algorithm	Time to process
average 5×5 (C-code)	0.146352561 seconds
median 5×5 (C-code)	0.401161394 seconds
convolution 5×5 (C-code)	0.165119984 seconds
Canny 11×11 (C-code)	0.217648294 seconds
average 5×5 (Verilog)	0.005243070 seconds
median 5×5 (Verilog)	0.005243470 seconds
convolution 5×5 (Verilog)	0.005243090 seconds
Canny 11×11 (Verilog)	0.005386770 seconds

Table 1. Results for Lena 256×256 and 512×512

The simulated Verilog hardware results are clearly much faster than the results for the software C-code (for both the 256×256 and 512×512 Lena images). For example, the Canny algorithm is over 29 times faster in hardware than it is in software (Lena 256×256). Similarly, the median 5×5 algorithm is over 76 times faster in hardware than it is in software (Lena 512×512). Figure 8 shows the outputs from the ModelSim hardware simulation (erosion and dilation outputs were obtained using the median filter returning the minimum (erosion) and maximum (dilation) values instead of the median value).

5 Conclusion

A hardware image processing library has been described and produced consisting of an average filter, a median fil-

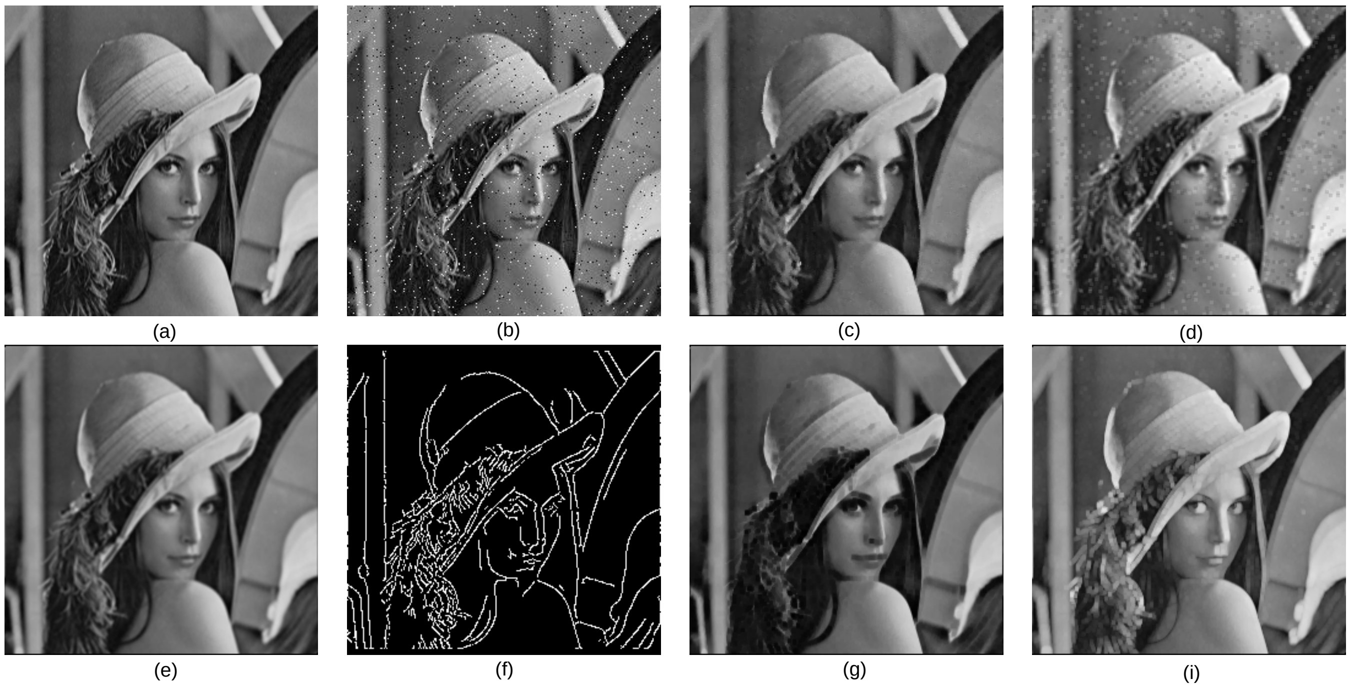

Figure 8. Images processed in ModelSim using Verilog: **(a)** original Lena 256×256, **(b)** Lena with salt and pepper noise, **(c)** median 3×3 filter applied to Lena with salt and pepper noise, **(d)** average 3×3 filter applied to Lena with salt and pepper noise, **(e)** Gaussian 3×3 smoothing (convolution) applied to original Lena, **(f)** Canny algorithm applied to original Lena, **(g)** Greyscale erosion (minimum) 3×3 applied to original Lena, **(h)** Greyscale dilation (maximum) 3×3 applied to original Lena

ter, a minimum filter (greyscale erosion), a maximum filter (greyscale dilation), a convolution algorithm and a Canny edge detector algorithm. These algorithms were designed to be highly parallel, pipelined and with low complexity (minimizing longest-path delay); such a design facilitates very quick execution speeds in hardware. The algorithms in this library are commonly used in image processing applications and as such it is foreseen that they can be frequently re-used. Simulation results show that these hardware algorithms significantly outperform their software counterparts.

6 Addendum: Histogram equalisation

6.1 Histogram equalisation background

Histogram equalisation [11, 17, 18] increases the global contrast of an image by spreading out the most frequently occuring intensity values. This process counts the occurance of each pixel value thus forming an image histogram. The cumulative distribution function (CDF) of this histogram is calculated and then normalised (resulting in histogram equalisation). As an example, consider the 4×4 sub-image in Table 2. The histogram for this sub-image is obtained by counting the occurrence of each pixel value; Table 3 shows the histogram (note that pixel values with a zero count have been excluded for brevity). The CDF is

127	94	127	94
165	199	127	112
80	127	199	165
94	165	112	210

Table 2. Example sub-image, 4×4

obtained by keeping a *running total* of the *count* values in the histogram (shown in Table 4). The CDF shows that the minimum value in the sub-image is 80 and the maximum value is 210. The CDF must be normalised to the range [0,255]. The histogram equalisation (he) formula is:

$$he(i) = round\left(\frac{cdf(i) - cdf_{min}}{(M \times N) - cdf_{min}} \times (L-1)\right) \quad (1)$$

where $M \times N$ are the image dimensions, cdf_{min} is the minimum value in the CDF and L is the number of grey levels. For example, in Table 4 the CDF of 165 is 13, using equation 1 it is possible to calculate what value 165 maps to after histogram equalisation:

$$he(165) = round\left(\frac{13-1}{16-1} \times (256-1)\right) = 204 \quad (2)$$

Pixel value	Count
80	1
94	3
112	2
127	4
165	3
199	2
210	1

Table 3. The histogram

Pixel value	Count
80	1
94	4
112	6
127	10
165	13
199	15
210	16

Table 4. The cumulative distribution function

153	51	153	51
204	238	153	85
0	153	238	204
51	204	85	255

Table 5. Histogram equalised sub-image

6.2 Histogram equalisation in hardware

An issue in implementing this algorithm in hardware is the use of floating point numbers. The fractional part of equation 1 is bounded to the range [0,1]. This normalised range is multiplied by $L-1$ (where L is 256) to obtain the histogram equalised range of [0,255]. It is possible to do floating point calculations using integer values by using *fixed point arithmetic* [19, 20].

As an example, consider dividing the 16-bit unsigned integer a by the 16-bit unsigned integer b using fixed point arithmetic. It can be said that a and b have a *M.N* notation of 16.0 (M is the number of integer bits and N is the number of fractional bits). To do this division [20], multiply a by 2^N and then divide the result by b. An example of how to do this is given in the pseudo Verilog code below. Figure 9 shows an example of histogram equalisation obtained from a ModelSim simulation.

```
reg [15:0] a;           // M.N = 16.0 (16-bit)
reg [15:0] b;           // M.N = 16.0 (16-bit)
reg [15:-16] aShift;    // M.N = 16.16 (32-bit)
reg [15:-16] temp;      // M.N = 16.16 (32-bit)
reg [15:0] result;      // 16-bit result (rounded)

aShift[15:0]<=a;        // a<<N, ie: multiply by 2^{N}

temp<=(aShift/b)*255;   // do the calculation

if (temp[-1]==1'b1)     // is fractional part >= 0.5?
  result<=temp[15:0]+1;  // round up
else if (temp[-1]==1'b0) // is fractional part < 0.5
  result<=temp[15:0];    // round down
```

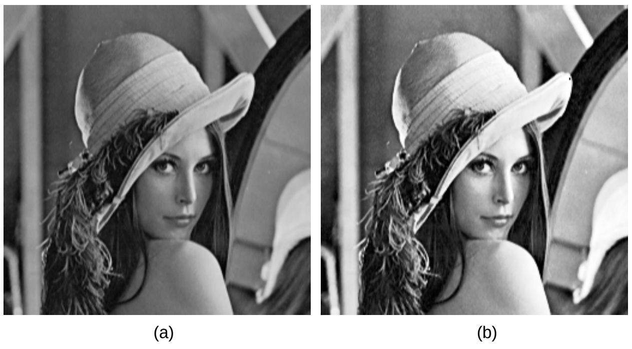
Figure 9. (a) Original Lena and (b) histogram equalised Lena

7 Addendum: Canny edge detection

An overview of the Canny edge detection algorithm was given in Sections 2.4 and 3.5. Here, a more detailed description of its hardware implementation is given.

7.1 Smoothing

An 11×11 sliding window is used to perform the smoothing part of the Canny algorithm. A Gaussian derivative smoothing function is created using the convolution module described in Section 3.4. The left-hand values of the Gaussian derivative mask (standard deviation of 1.25) are given in equation 3; the right-hand values of this mask are given in equation 4 (symmetrical but negative).

$$G(1.25)_{LHV} = \frac{1}{2.48}(+0.17 \ \ +0.38 \ \ +0.64 \ \ +0.76 \ \ +0.54) \tag{3}$$

$$G(1.25)_{RHV} = \frac{1}{2.48}(-0.54 \ \ -0.76 \ \ -0.64 \ \ -0.38 \ \ -0.17) \tag{4}$$

As shown in equations 5 and 6, equations 3 and 4 can be approximated using integers rather than floating point numbers.

$$G(1.25)_{LHV} = \frac{1}{27}(+2 \ \ +4 \ \ +7 \ \ +8 \ \ +6\ ) \tag{5}$$

$$G(1.25)_{RHV} = \frac{1}{27}(-6 \ \ -8 \ \ -7 \ \ -4 \ \ -2\ ) \tag{6}$$

A graphical representation of these equations is given in Figure 10. A further optimisation can be obtained by replacing the divide by 27 operation with a divide by 16 operation (shift-right). An adjustment in the Canny thresholds can compensate for this alteration.

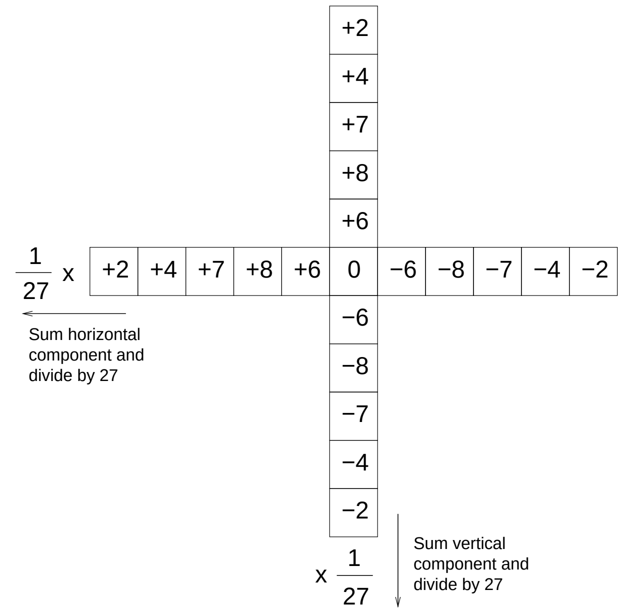
+2
+4
+7
+8
+6
$\frac{1}{27}$ x
+2
+4
+7
+8
+6
0
−6
−8
−7
−4
−2
Sum horizontal component and divide by 27
−6
−8
−7
−4
−2
x $\frac{1}{27}$
Sum vertical component and divide by 27

Figure 10. Mask for Gaussian derivative smoothing

It can be seen from Figure 10 that the Gaussian derivative mask is applied in both the vertical and horizontal directions. Equation 7 shows how these vertical and horizontal Gaussian derivatives can be combined.

$$G(r,c) = \sqrt{G_v(r,c)^2 + G_h(r,c)^2} \tag{7}$$

Equation 7 is expensive in terms of hardware resources thus a simplified version (equation 8) can be used instead.

$$\begin{aligned} G(r,c) = max(|G_v(r,c)|, |G_h(r,c)|) + \\ (min(|G_v(r,c)|, |G_h(r,c)|))/4 \end{aligned} \tag{8}$$

Figure 11 shows an example of applying Gaussian derivative smoothing (output from equation 8) upon the Lena image.

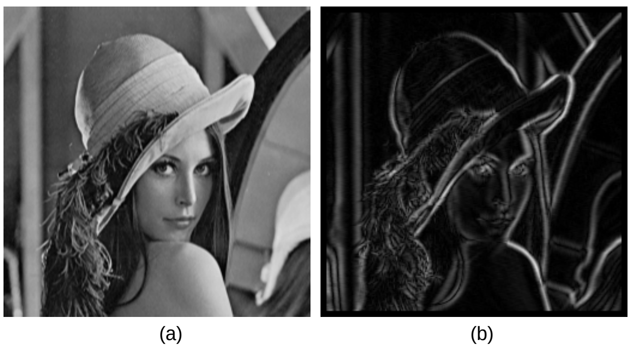

Figure 11. (a) Original Lena and (b) Gaussian derivative smoothing applied to Lena

7.2 Non-maximal suppression

This is an edge thinning step that reduces thick edges (Figure 11(b)) into thin edges that are one-pixel-wide. Non-maximal suppression is performed by first calculating the magnitude and phase of each pixel in the veritical and horizontal directions. The Gaussian derivative image in Figure 11(b) gives the magnitude of each pixel (as described in Sections 3.5 and 7.1). Calculating the phase of each pixel requires use of the *arctan* function ($direction = arctan\left(\frac{dy}{dx}\right)$). The *arctan* function is complex and requires use of floating point numbers (making it difficult to implement in hardware).

A simplier alternative to the *arctan* function is to use the magnitude and sign of the vertical and horizontal components of each pixel. From the mask (Figure 10) used to apply Gaussian derivative smoothing the dy and dx magnitudes of each pixel can be obtained. The phases of each pixel are quantised into one of four orientations based upon the compass rose: (1) NNW-SSE, (2) WNW-ESE, (3) NNE-SSW and (4) WSW-ENE. The pseudo Verilog code below shows how this is done.

```
dx_times_dy <= dx*dy;
abs_dx <= (dx < 0) ? -dx : dx;
abs_dy <= (dy < 0) ? -dy : dy;
if (dx_times_dy > 0) begin
    if (abs_dx < abs_dy)
        pixelPhaseQuant <= 1;
    else
        pixelPhaseQuant <= 2;
end
else begin
    if (abs_dx < abs_dy)
        pixelPhaseQuant <= 3;
    else
        pixelPhaseQuant <= 4;
end
```

This process is shown diagramatically in Figure 12. Note that the x-axis in this figure runs in the opposite direction to normal due to the positive values being on the left hand side in the Gaussian derivative smoothing mask (Figure 10). Now that the magnitude (equation 8, Figure 11(b)) and the phase (Figure 12) of each pixel is known, non-maximal suppression can be applied to reduce the thickness of the edge map.

Let the pixel under consideration be C. The pixels around C along the direction of the gradient are used to determine if C is a local maximum gradient (a 3×3 window is used for this). If C is a local maximum gradient, it is kept, otherwise, it is suppressed. Figure 13 shows which four neighbours are used based upon the gradient (phase) of the central pixel C. Consider Figure 13(a) where C has a phase quant of one. Pixel 1 (p1) and pixel 2 (p2) are interpolated by finding their average magnitude value $\overline{mag_{p_{12}}} = (p1_{mag} + p2_{mag})/2$. The magnitude of pixels 3 and 4 are also interpolated: $\overline{mag_{p_{34}}} = (p3_{mag} + p4_{mag})/2$. Finally, if the magnitude of C is less than either $\overline{mag_{p_{12}}}$ or

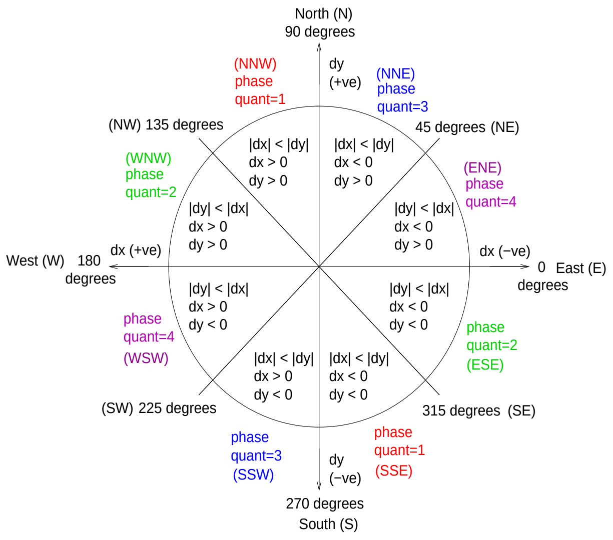

Figure 12. Obtaining quantised phases based upon the magnitude and sign of the dx and dy components of pixels

$\overline{mag_{p_{34}}}$, it is suppressed, otherwise, it is kept. The output from non-maximal suppression is shown in Figure 14(a).

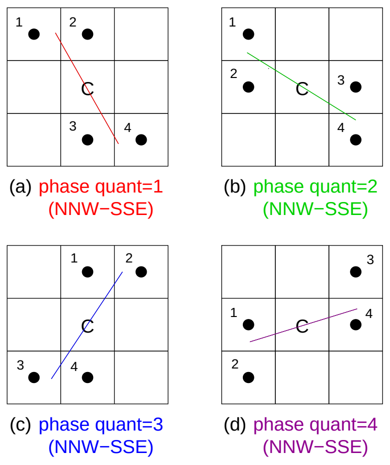
1
2
C
3
4
(a) phase quant=1 (NNW−SSE)
1
2
C
3
4
(b) phase quant=2 (NNW−SSE)
1
2
C
3
4
(c) phase quant=3 (NNW−SSE)
3
1
C
4
2
(d) phase quant=4 (NNW−SSE)

Figure 13. Interpolated pixels used for computing non-maximal suppression based upon quantised phases

7.3 Hysteresis thresholding

The image produced after non-maximal suppression may consist of single-edge points and broken edges which contribute to noise [16]. To suppress this, hysteresis thresholding applies two thresholds to the non-maximally supressed image, th_{low} and th_{high}. Applying two thresholds is better than applying one because if an edge fluctuates above and below a single threshold, it will appear broken (*streaking*). If the magnitude of a pixel is greater than th_{high}, it is accepted as a *definite edge*. If the magnitude of a pixel is less

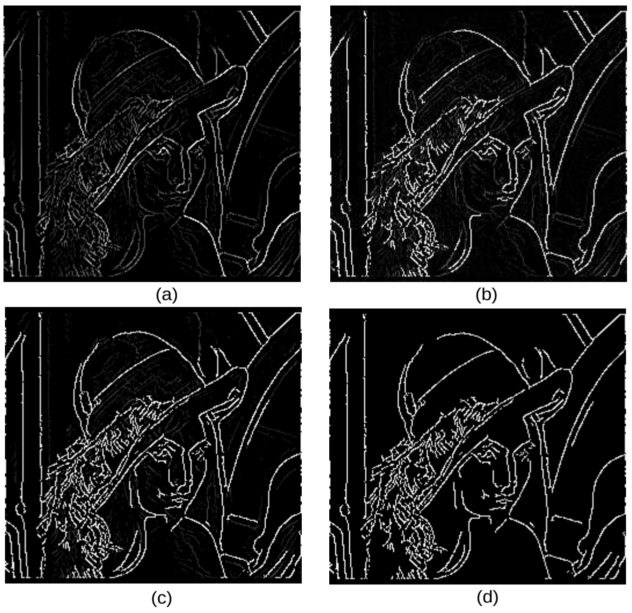

Figure 14. (a) Non-maximal suppression, (b) hysteresis thresholding applied once, (c) hysteresis thresholding applied seventeen times and (d) binarised version of hysteresis thresholding applied seventeen times

than th_{low}, it is rejected and set to zero. If the pixel magnitude lies between th_{low} and th_{high}, it may be an edge so it is labelled a *possible edge*. If there is a path from the *possible edge* to a pixel with magnitude $\geq th_{high}$, the *possible edge* is changed to a *definite edge*. If there is no such path, the *possible edge* is set to zero.

The paths between *definite edges* and *possible edges* are computed using 3×3 windows. If the centre pixel is a *definite edge* (Figure 15(a)), any of its neighbours that are *possible edges* are converted to *definite edges* (Figure 15(b)). If the centre pixel is a *possible edge* (Figure 15(c)) and if any its neighbours is a *definite edge*, the centre pixel is changed to a *definite edge* (Figure 15(d)).

This process was applied once to the non-maximally suppressed image (Figure 14(a)) resulting in Figure 14(b). In order to *grow* the lines in the edge map, hysteresis thresholding is applied recursively. For example, Figure 14(c) shows the edge map obtained after recursively applying hysteresis thresholding (seventeen times) upon the non-maximally suppressed image. It can clearly be seen that the edge map has grown compared to Figure 14(b). Figure 14(d) shows a binarised version of Figure 14(c); all pixel values of 255 were kept, all pixel values below 255 were set to zero.

References

[1] D. Johnson, K. Gribbon, D. Bailey, and S. Demidenko. Implementing signal processing algorithms in FPGAs: Digital spectral warp-

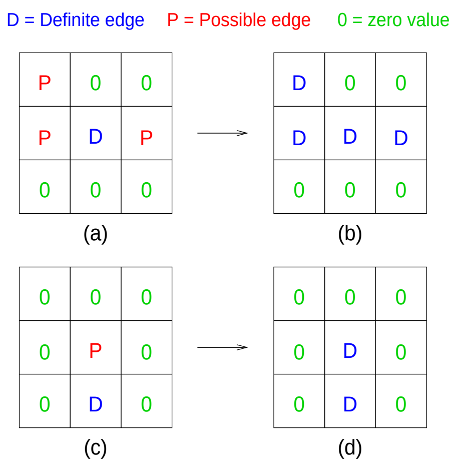

Figure 15. Hysteresis thresholding example

ing. In *Proc. of the ninth Electronics New Zealand Conference, ENZCon'02*, pages 72–74, Dunedin, New Zealand, November 2002.

[2] A. Bouridane, D. Crookes, P. Donachy, K. Alotaibi, and K. Benkrid. A high level FPGA-based abstract machine for image processing. *Journal of Systems Architecture*, 45:809–824, 1999.

[3] M. S. Prieto. *A system for embedded machine vision using FPGAs and neural networks*. PhD thesis, Department of Engineering, University of Aberdeen, Scotland, UK, December 2005.

[4] D. Crookes. Architectures for high performance image processing: The future. *Journal of Systems Architecture*, 45:739–748, 1999.

[5] J. Batlle, J. Martí, P. Ridao, and J. Amat. A new FPGA/DSP-based parallel architecture for real-time image processing. *Real-Time Imaging*, 8:345–356, 2002.

[6] L. Kessal, N. Abel, and D. Demigny. Real-time image processing with dynamically reconfigurable architecture. *Real-Time Imaging*, 9:297–313, 2003.

[7] S. McBader and P. Lee. An FPGA implementation of a flexible, parallel image processing architecture suitable for embedded vision systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'03)*, Nice, France, April 2003.

[8] K. Benkrid, D. Crookes, J. Smith, and A. Benkrid. High level programming for real time FPGA based video processing. In *Intl. Conf. on Acoustics, Speech, and Signal Processing, ICASSP'2000*, volume 6, pages 3227–3230, Istanbul, Turkey, June 2000.

[9] D. E. Thomas and P. R. Moorby. *The Verilog Hardware Description Language (Fifth Edition)*. Kluwer Academic Publishers, 2002.

[10] ModelSim 6.2. Mentor Graphics Corporation. http://www.model.com (date accessed: 9th Dec. 2008).

[11] J. C. Russ. *The Image Processing Handbook (Third Edition)*. CRC Press, 1998.

[12] J. Canny. A computational approach to edge detection. *IEEE Trans. on Pattern Recognition and Machine Intelligence*, 8(6):679–698, 1986.

[13] B. Zeidman. *Verilog Designer's Library*. Prentice Hall, 1999.

[14] D. Bitton, D. J. DeWitt, D. K. Hsaio, and J. Menon. A taxonomy of parallel sorting. *ACM Computing Surveys*, 16(3):287–318, 1984.

[15] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching (Third Edition)*. Addison-Wesley, 1997.

[16] V. R. Daggu and M. Venkatesan. Design and implementation of an efficient reconfigurable architecture for image processing algorithms using Handel-C. Master's thesis, Department of Electrical and Computer Engineering, University of Nevada, Las Vegas, USA, 2003.

[17] J. A. Stark and W. J. Fitzgerald. An alternative algorithm for adaptive histogram equalisation. *Graphical Models and Image Processing*, 58(2):180–185, 1996.

[18] Wikipedia. Histogram equalization. Website. http://en.wikipedia.org/Histogram_equalization (date accessed: 9th Dec. 2008).

[19] R. Yates. Fixed point arithmetic: An introduction. Website. http://www.digitalsignallabs.com/fp.pdf (date accessed: 9th Dec. 2008).

[20] J. Lauha. The neglected art of fixed point arithmetic. Website. http://jet.ro/files/The_neglected_art_of_Fixed_Point_arithmetic_20060913.pdf (date accessed: 9th Dec. 2008).