1-1-2003

Efficient design and implementation of image processing algorithms on reconfigurable hardware using Handel-C

Venkateshwar Rao Daggu
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
Daggu, Venkateshwar Rao, "Efficient design and implementation of image processing algorithms on reconfigurable hardware using Handel-C" (2003). UNLV Retrospective Theses & Dissertations. 1586.
https://digitalscholarship.unlv.edu/rtds/1586

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
EFFICIENT DESIGN AND IMPLEMENTATION OF IMAGE PROCESSING

ALGORITHMS ON RECONFIGURABLE HARDWARE

USING HANDEL-C

by

Venkateshwar Rao Daggu

Bachelor of Technology
Kakatiya University, Warangal, India
2000

A thesis submitted in partial fulfillment
of the requirement for the

Master of Science Degree in Computer Engineering
Department of Electrical and Computer Engineering
Howard R Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
December 2003

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The Thesis prepared by

Venkateshwar R. Daggu

Entitled

An Efficient Design and Implementation of Image-Processing Algorithms on Reconfigurable Hardware Using Handel-C

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering

Examination Committee Chair

Dean of the Graduate College

Graduate College Faculty Representative
ABSTRACT

Efficient Design and Implementation of Image Processing Algorithms on Reconfigurable Hardware using Handel-C

by

Venkateshwar Rao Daggu

Dr. Muthukumar Venkatesan, Examination Committee Chair
Professor of Electrical and Computer Engineering
University of Nevada, Las Vegas

Computer manipulation of images is generally defined as Digital image processing (DIP). DIP is used in variety of applications, including video surveillance, target recognition, and image enhancement. These applications are usually implemented in software but may use special purpose hardware for speed. With advances in the VLSI technology hardware implementation has become an attractive alternative. Assigning complex computation tasks to hardware and exploiting the parallelism and pipelining in algorithms yield significant speedup in running times. In this thesis the image processing algorithms like median filter, basic morphological operators, convolution and edge detection algorithms are implemented on FPGA. A pipelined architecture of these algorithms is presented. The proposed architectures are capable of producing one output on every clock cycle. The hardware modeling was accomplished using Handel-C (DK2 environment). The algorithm was tested on standard image processing benchmarks and the results are compared with that obtained on software.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>viii</td>
</tr>
<tr>
<td>CHAPTER 1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>CHAPTER 2 LITERATURE REVIEW</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Field Programmable Gate Arrays (FPGAs)</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Virtex™-E FPGA</td>
<td>6</td>
</tr>
<tr>
<td>2.2.1 Configurable Logic Block (CLB)</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2 Block SelectRAM (BRAMs)</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Handel-C</td>
<td>10</td>
</tr>
<tr>
<td>2.3.1 Parallel Hardware Generation</td>
<td>10</td>
</tr>
<tr>
<td>2.3.2 Efficient FPGA Resources Usage</td>
<td>12</td>
</tr>
<tr>
<td>2.3.3 Bit Level Operators</td>
<td>12</td>
</tr>
<tr>
<td>2.3.4 Channel communications</td>
<td>13</td>
</tr>
<tr>
<td>2.3.6 External Communication</td>
<td>13</td>
</tr>
<tr>
<td>2.3.5 Memory</td>
<td>14</td>
</tr>
<tr>
<td>2.3.7 Targets Supported by Handel-C</td>
<td>15</td>
</tr>
<tr>
<td>2.4 RC1000 PCI Board</td>
<td>16</td>
</tr>
<tr>
<td>2.5 Optimization Techniques</td>
<td>16</td>
</tr>
<tr>
<td>2.5.1 Parallelism</td>
<td>17</td>
</tr>
<tr>
<td>2.5.2 Longest Path Delay</td>
<td>18</td>
</tr>
<tr>
<td>2.5.3 Pipelining</td>
<td>19</td>
</tr>
<tr>
<td>2.6 Prior Related Work</td>
<td>19</td>
</tr>
<tr>
<td>CHAPTER 3 IMAGE PROCESSING ALGORITHMS</td>
<td>21</td>
</tr>
<tr>
<td>3.1 Median Filtering</td>
<td>21</td>
</tr>
<tr>
<td>3.2 Morphological Operators</td>
<td>24</td>
</tr>
<tr>
<td>3.3 Convolution</td>
<td>27</td>
</tr>
<tr>
<td>3.3.1 1D Convolution</td>
<td>27</td>
</tr>
<tr>
<td>3.3.2 2D-Convolution</td>
<td>28</td>
</tr>
<tr>
<td>3.4 Edge Detection</td>
<td>30</td>
</tr>
<tr>
<td>3.4.1 Smoothing</td>
<td>32</td>
</tr>
<tr>
<td>3.4.2 Gradient Calculation</td>
<td>33</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.3 Magnitude and Phase</td>
<td>34</td>
</tr>
<tr>
<td>3.4.4 Non-Maximum Suppression</td>
<td>35</td>
</tr>
<tr>
<td>3.4.5 Threshold</td>
<td>35</td>
</tr>
<tr>
<td>CHAPTER 4 HARDWARE IMPLEMENTATION</td>
<td>37</td>
</tr>
<tr>
<td>4.1 Moving Window Operator</td>
<td>37</td>
</tr>
<tr>
<td>4.2 Median Filter</td>
<td>40</td>
</tr>
<tr>
<td>4.3 Basic Morphological Operators</td>
<td>43</td>
</tr>
<tr>
<td>4.3 Convolution</td>
<td>44</td>
</tr>
<tr>
<td>4.4 Edge Detection</td>
<td>48</td>
</tr>
<tr>
<td>4.4.1 Image Smoothing</td>
<td>49</td>
</tr>
<tr>
<td>4.4.2 Vertical and Horizontal Gradient Calculation</td>
<td>50</td>
</tr>
<tr>
<td>4.4.3 Directional Non Maximum Suppression</td>
<td>51</td>
</tr>
<tr>
<td>4.4.4 Threshold</td>
<td>53</td>
</tr>
<tr>
<td>CHAPTER 5 RESULTS AND FUTURE WORK</td>
<td>56</td>
</tr>
<tr>
<td>5.1 Results</td>
<td>57</td>
</tr>
<tr>
<td>5.2 Future Work</td>
<td>62</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>63</td>
</tr>
<tr>
<td>APPENDIX</td>
<td>65</td>
</tr>
<tr>
<td>VITA</td>
<td>97</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2.1</td>
<td>Summary of Four Commercial FPGAs</td>
<td>6</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Timing result of Median Filter on a 256 x 256 grayscale image</td>
<td>57</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>Timing Result of Median Filter on a 512 x 512 grayscale image</td>
<td>58</td>
</tr>
<tr>
<td>Table 5.3</td>
<td>Timing result of 5x5 Convolution on a 256 x 256 grayscale image</td>
<td>58</td>
</tr>
<tr>
<td>Table 5.4</td>
<td>Timing result of 5x5 convolution on a 512 x 512 grayscale image</td>
<td>59</td>
</tr>
<tr>
<td>Table 5.5</td>
<td>Timing Results of 3x3 Convolution on a 256 x 256 grayscale image</td>
<td>59</td>
</tr>
<tr>
<td>Table 5.6</td>
<td>Timing Results of 3x3 Convolution on a 512 x 512 grayscale image</td>
<td>60</td>
</tr>
<tr>
<td>Table 5.7</td>
<td>Timing Result of edge detection algorithm on 256 x 256 grayscale image</td>
<td>60</td>
</tr>
<tr>
<td>Table 5.8</td>
<td>Timing Result of edge detection algorithm on 512 x 512 grayscale image</td>
<td>60</td>
</tr>
<tr>
<td>Table 5.9</td>
<td>Implementation Cost on FPGA</td>
<td>61</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>FPGA Architecture</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Vertex^Tm-E Architecture Overview</td>
<td>7</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Vertex-E Local Routing</td>
<td>8</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>2-slice VirtexTm-E CLB</td>
<td>9</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>Parallel branch execution flow</td>
<td>11</td>
</tr>
<tr>
<td>Figure 2.6</td>
<td>Channel Communication</td>
<td>13</td>
</tr>
<tr>
<td>Figure 2.7</td>
<td>Translating code into hardware using Handel-C</td>
<td>15</td>
</tr>
<tr>
<td>Figure 2.8</td>
<td>RC1000 Block Diagram</td>
<td>17</td>
</tr>
<tr>
<td>Figure 2.9</td>
<td>Basic parallelism</td>
<td>18</td>
</tr>
<tr>
<td>Figure 2.10</td>
<td>Breaking Up complex Operations</td>
<td>19</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Median Filter</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Median Filter on an image</td>
<td>23</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Structuring element fitting and not fitting</td>
<td>25</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Erosion and Dilation of a Gray Scale Image</td>
<td>26</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Convolution</td>
<td>29</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>Convolution Mask</td>
<td>30</td>
</tr>
<tr>
<td>Figure 3.7</td>
<td>5x5 Gaussian Convolution of Lena Image</td>
<td>30</td>
</tr>
<tr>
<td>Figure 3.8</td>
<td>Schematic of canny edge detection</td>
<td>32</td>
</tr>
<tr>
<td>Figure 3.9</td>
<td>5x5 Gaussian convolution mask</td>
<td>33</td>
</tr>
<tr>
<td>Figure 3.10</td>
<td>Gradient of Image</td>
<td>34</td>
</tr>
<tr>
<td>Figure 3.11</td>
<td>Edge Detection of Gray Scale Image</td>
<td>36</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Architecture of 3x3 moving window</td>
<td>39</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Architecture of 5x5 moving window</td>
<td>39</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Compare Unit</td>
<td>41</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Hardware Design for Sorting Algorithm</td>
<td>42</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Pipelining Process</td>
<td>43</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>8 bit Constant Coefficient Multiplier (KCM)</td>
<td>45</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>Convolution Masks</td>
<td>46</td>
</tr>
<tr>
<td>Figure 4.8</td>
<td>Design Flow of Edge Detection</td>
<td>48</td>
</tr>
<tr>
<td>Figure 4.9</td>
<td>Pipelined Architecture</td>
<td>49</td>
</tr>
<tr>
<td>Figure 4.10</td>
<td>Gradient Convolution Kernels</td>
<td>50</td>
</tr>
<tr>
<td>Figure 4.11</td>
<td>Gradient Orientation</td>
<td>52</td>
</tr>
<tr>
<td>Figure 4.12</td>
<td>Pixel Interpolation</td>
<td>53</td>
</tr>
<tr>
<td>Figure 4.13</td>
<td>Lena image processed on FPGA</td>
<td>55</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Host FPGA Communication</td>
<td>56</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

Comments, reviews and support of many people have helped the development of this thesis. I would like to thank all of them who have been instrumental in this work. The first and foremost I would like to thank Dr. Venkatesan Muthukumar for his guidance on my research and study at University of Nevada, Las Vegas. Without his valuable advice and encouragement, I could not have reached this stage in my academic pursuits.

I wish to thank members of my Thesis committee Dr. Evangelos Yfantis, Dr. Henry Selvaraj, Dr. Emma Regentova and also in providing me some valuable tips which made this thesis a success. I would like to extend thanks to my colleagues at University of Nevada, Las Vegas for their unconditional support in helping me. Finally I would like to thank all those who were involved either directly or indirectly in the completion of this thesis. I am too grateful to say anything else.

Last but certainly not least, I am forever indebted to the love and care of my family, who made me capable to accomplish their expectations.
CHAPTER 1

INTRODUCTION

Digital image processing is an ever expanding and dynamic area with applications reaching out into our everyday life such as in medicine, space exploration, surveillance, authentication, automated industry inspection and in many more areas.

Applications such as these involve different processes like image enhancement, and object detection [14]. Implementing such applications on a generable purpose computer can be easier but not very efficient in terms of speed. The reason being the additional constraints put on memory and other peripheral device management. Application specific hardware offers much greater speed than a software implementation.

There are two types of technologies available for hardware design. Full custom hardware design also called as Application Specific Integrated Circuits (ASIC) and semi custom hardware device, which are programmable devices like Digital signal processors (DSP’s) and Field Programmable Gate Arrays (FPGA’s).

Full custom ASIC design offers highest performance, but the complexity and the cost associated with the design is very high. The ASIC design cannot be changed; time taken to design the hardware is also very high. ASIC designs are used in high volume commercial applications. In addition, if an error exist in the hardware design, once the design is fabricated, the product goes useless.
DSP’s are a class of hardware devices that fall somewhere between an ASIC and a PC in terms of the performance and the design complexity. DSP’s are specialized microprocessors, typically programmed in C, perhaps with assembly code for performance. It is well suited to extremely complex math intensive tasks such as image processing. Hardware design knowledge is still required, but the learning curve is much lower than some other design choices [3].

Field Programmable Gate Arrays are programmable devices [2]. They are also called reconfigurable devices. Reconfigurable devices are processors which can be programmed with a design, and the design can be by reprogramming the devices. Hardware design techniques such as parallelism and pipelining techniques can be developed on a FPGA [7], which is not possible in dedicated DSP designs. So FPGAs are ideal choice for implementation of real time image processing algorithms.

FPGAs have traditionally been configured by hardware engineers using a Hardware Design Language (HDL). The two principal languages being used are Verilog and VHDL. Verilog and VHDL are specialized design techniques that are not immediately accessible to software engineers, who have often been trained using imperative programming languages. Consequently, over the last few years there have been several attempts at translating algorithmic oriented programming languages directly into hardware descriptions. A new C like hardware description language called Handel-C introduced by Celoxica [5], allows the designer to focus more on the specification of an algorithm rather than adopting a structural approach to coding. For these reasons the Handel-C is used for implementation of image processing algorithms on FPGA.
The goal of this thesis is to implement image processing algorithms like Median filter, convolution and canny edge detection on FPGA using Handel-C and compare against the performance of software implementation on a general purpose computer.

Chapter two provides information on FPGA’s, Handel-C, RC1000 and prior related work. Chapter three describes the image processing algorithms like Median Filter, convolution and canny edge detection. Chapter four provides the details on the implementation of the image processing algorithms on a Xilinx Vertex-E FPGA for a 256 x 256 gray scale image. Chapter five summaries the results and future work.
CHAPTER 2

LITERATURE REVIEW

This chapter provides information on the basic concepts of field programmable gate arrays (FPGAs) including a description of the Xilinx Vertex-E FPGA, Handel-C language and RC1000 board which are used in this thesis followed by prior related work.

2.1 Field Programmable Gate Arrays (FPGAs)

A Field Programmable Gate Array (FPGA) [2] as name suggests is a programmable device in which the final logic structure can be directly configured by the end user for a variety of applications. In its simplest form an FPGA consists of an array of uncommitted elements that can be programmed or interconnected according to a user’s specification. The ability to reprogram these devices over and over again of the flexibility of interconnection resources makes FPGAs an ideal device for implementing & testing ASIC prototypes. The Figure 2.1 portrays the architecture of a conceptual FPGA. The most important components in an FPGA are configurable logic blocks (CLB’s), input-output blocks and programmable switches.

The architecture has a two dimensional array of CLBs that are by general interconnection resources. These CLBs can be as simple as 2-input NAND gates or it can have a complex structure such as multiplexers or look-up tables. Most logic blocks also contain some type of flip-flop, to aid in the implementation of sequential circuits.
The interconnect consists of segments of wire, where the segments may be of various lengths. These interconnects are made up programmable switches that serve to connect the CLBs to the wire segments, or one wire segment to another. The wire segments along with programmable switches are together wired as routing architecture. Similar to the logic blocks, these switches can be designed in many ways. Some FPGAs offer a large number of simple connections between blocks where as others provide fewer, but complex routes. The programmable switches can be constructed in several ways including: pan-transistors controlled by static RAM cells, antifuses, EPROM transistors and EEPROM transistors.
Commerically FPGAs have been classified into four-major categories based on their interconnection. The interconnection can be symmetrical array, row based, hierarchical, or sea of gates. Table 2.1 shows commercially available FPGA’s [2].

2.2 Virtex™-E FPGA

Virtex™-E FPGA [6] produced by Xilinx, Inc. is used for our implementation. The Virtex™-E FPGA architecture has two major configurable elements: configurable logic blocks (CLBs) and Input/output blocks (IOBs). CLBs provide the functional elements for constructing logic. IOBs provide the interface between the package pins and the CLBs. The Virtex™-E FPGA also has dedicated block memories called Block SelectRAM™ memories (BRAMs). The Virtex™-E belongs to the Virtex™ family of FPGAs which features regular arrays of CLBs arranged in columns surrounded on all sides by IOBs as shown in Figure 2.2 [6].

Table 2.1 Summary of Four Commercial FPGAs

<table>
<thead>
<tr>
<th>Company</th>
<th>Architecture</th>
<th>Logic Block Type</th>
<th>Programming Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actel</td>
<td>Row-Based</td>
<td>Multiplexer-Based</td>
<td>Anti-fuse</td>
</tr>
<tr>
<td>Altera</td>
<td>Hierarchical-PLD</td>
<td>PLD Block</td>
<td>EPROM</td>
</tr>
<tr>
<td>QuickLogic</td>
<td>Symmetrical Array</td>
<td>Multiplexer-Based</td>
<td>Anti-fuse</td>
</tr>
<tr>
<td>Xilinx</td>
<td>Symmetrical Array</td>
<td>Look-up Table</td>
<td>Static RAM</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The VersaRing I/O interface provides additional routing resources around the periphery of the device. This routing improves I/O routability and facilitates pin locking. The interconnection within them is very versatile as the wire segments are of varying lengths and the programmable switches are fast and placed in locations that allow them to efficiently connect these wire segments. Interconnection of CLBs is through a general routing matrix (GRM) as shown in Figure 2.3. The GRM contains routing switches that connect the vertical and horizontal routing channels. Each CLB nests into a VersaBlock™ that connects the CLBs to the GRM. Virtex™ FPGAs are SRAM-based. A design is implemented by loading configuration data into their internal memory cells. The values stored in static memory cells control the configurable logic elements and the interconnect.

![Figure 2.2 Vertex™-E Architecture Overview](image)
resources. These values load into the memory cells on power-up and can reload if necessary to change the function of the device.

2.2.1 Configurable Logic Block (CLB)

The basic building block of the vertex-E CLB is the logic cell (LC). A Virtex™-E CLB contains four logic cells (LC). An LC contains a four-input function generator, carry logic, and a storage element. The entire Vertex CLB is made of two CLB slices, each containing two LCs. Figure 2.4 illustrates the various components of the Virtex™-E CLB. The output from the function generator in each LC drives both the CLB output and the D flip-flop. Four input look-up-tables (LUTs) with 16 locations in each LUT implement function generators. A function is implemented in a LC by loading data into the LUT. The input into the LC is an address into the LUT. The value stored at that address is the output of the LC. Two LUT within a slice can be combined to create a 16 x 2-bit or 32 x1-bit synchronous RAM, or a 16x1 dual-port synchronous RAM.
can also work as 16-bit shift register, which can be used to store data in high speed applications such as Digital Image Processing.

2.2.2 Block SelectRAM (BRAMs)

Block SelectRAM (BRAMs) are dedicated blocks of memory that can store large amounts of data. Each memory block is four CLBs high and is organized into memory columns stretching the entire height of the chip. There is one such memory column between every twelve CLB columns. The block SelectRAM also includes dedicated routing to provide an efficient interface with both CLBs and other block SelectRAMs. Each Block SelectRAM is a fully synchronous dual-ported and can store 4096 bits. Each port has independent control signals so that the two ports can be configured independently. The width of each addressable location can vary from 1 to 16 bits. For example, if each location is 16-bits wide, then there will be 256 such locations within one
Block SelectRAM memory. The dual-port Block SelectRAM is used in our implementation to store image data.

2.3 Handel-C

Handel-C is essentially an extended subset of the standard ANSI-C language, specifically designed for use in a hardware environment. Unlike other C to FPGA tools which rely on going via several intermediate stages, Handel-C allows hardware to be directly targeted from software, allowing a more efficient implementation to be created. The language is designed around a simple timing model that makes it very accessible to system architects and software engineers.

The Handel-C compiler comes packaged with the Celoxica DK1 development environment. DK1 does not provide synthesis, and the suite must be used in conjunction with one of any number of synthesis tools available to complete the design flow from idea to hardware.

This section is not intended to be a full description of the tool and language, but it does describe the most important features, especially those that influence the design decisions described later in this thesis. For full details of the language and development environment the reader is referred to the user guides and reference material from the manufactures.

2.3.1 Parallel Hardware Generation

One of the advantages of using hardware is the ability to exploit parallelism directly. Handel-C has additional constructs to support the parallelization of code using the par statement. When instructed to execute two instructions in parallel, those two instructions
will be executed at exactly the same instant in time by two separate pieces of hardware. When a parallel block is encountered, execution flow splits at the start of the parallel block and each branch of the block executes simultaneously. Execution flow then re-joins at the end of the block when all branches have completed. Any branches that complete nearly are forced to wait for the slowest branch before continuing as shown in Figure 2.5. For example, the block

```
par {
    a=10;
    b=20;
}
```

Figure 2.5 Parallel branch execution flow

generates hardware to assign the value 10 to a and 20 to b in a single clock cycle. Using this statement, large blocks of functionality can be generated that execute in parallel. It should be noted that variable cannot be written multiple times in same clock cycle.
Par{
    a=10;
    a=20;
}

// this is not allowable.

Hardware can be replicated using the construct

par (i=0;i<10;i++)
{
    a[i] = b[i];
}

which results in 10 parallel assignment operations.

2.3.2 Efficient FPGA Resources Usage

For efficient use of hardware, Handel-C provides the flexibility of use of user defined data types of variable sizes. int n x; This defines a variable x of type int and size of n bits. For example, int 10 count; is a signed integer that is 10 bits wide.

2.3.3 Bit Level Operators

Handel-C provides a number of bit manipulation operators. The following bit operators are provided:

- $y = x \downarrow n$ Drops the $n$ least significant bits from $x$
- $y = x \leftarrow n$ Takes the $n$ least significant bits from $x$
- $y = x @ z$ Concatenates the bit patterns that represent $x$ and $z$
- $y = x[3:1]$ Selects bits 1, 2 and 3 from $x$
2.3.4 Channel communications

Channels provide a link between parallel branches. One parallel branch outputs data onto the channel and the other branch reads data from the channel. Channels also provide synchronization between parallel branches because the data transfer can only complete when both parties are ready for it. If the transmitter is not ready for the communication then the receiver must wait for it to become ready and vice versa. In Figure 2.6, the channel is shown transferring data from the left branch to the right branch. If the left branch reaches point a before the right branch reaches point b, the left branch waits at point a until the right branch reaches point b.

![Figure 2.6 Channel Communication](image)

2.3.6 External Communication

Communication between the hardware and the outside world is performed using interfaces. These may be specified as input or output, and, as with assignment, a write-to or a read-from an interface will take one clock cycle. The language allows the designer to target particular hardware, assign input and output pins, specify the timing of signals, and generally control the low level hardware interfacing details. Macros are available to help target particular devices.
2.3.5 Memory

RAMs and ROMs can be implemented directly using the "ram" and "rom" keyword. Specifying the "block" parameter in conjunction with the "ram" keyword can identify Block RAMs. Normal variables are implemented as flip-flops. The Handel-C code for RAM declaration is as follows.

A 16x8 bit RAM declaration

```
Ram 8 RAM[16];
```

A 4x8 bit memory in block RAM declaration

```
ram 8 blockRAM[16] = {11, 22, 33, 44} with {block = 1};
```

A 34x6 bit memory in distributed RAM declaration

```
ram unsigned 6 distRAM[34];
```

2.3.6 Some Restrictions When Using Handel-C and FPGAs

One problem with Handel-C is that it is not designed as a HDL, rather a high-level language with a hardware output, and as such does not always completely support the full utilisation of the underlying hardware.

Since Handel-C targets hardware, it imposes some programming restrictions when compared to a traditional C compiler. These need to be taken into consideration when designing code that can be compiled by Handel-C. Some of these restrictions particularly affect the implementation of algorithms. Firstly, there is no stack available, so recursive functions cannot be directly supported by the language. cannot be implemented without some modification. Secondly, the size of memory that can be implemented using standard logic cells on an FPGA is limited, because implementing memory is an inefficient use of FPGA resources. However, some FPGAs have internal RAM that can
be used by Handel-C. A limitation of using RAM or ROM is that it cannot be accessed more than once per clock cycle. This restricts the potential for parallel execution of code that accesses RAM or ROM.

2.3.7 Targets Supported by Handel-C

Handel-C supports two targets. The first is a simulator target that allows development and testing of code without the need to use any hardware. This is supported by a debugger and other tools. The second target is the synthesis of a netlist for input to place and route tools. Place and route is the process of translating a netlist into a hardware layout. This allows the design to be translated into configuration data for particular chips. An overview of the process is shown in Figure 2.7. When compiling the design for a

![Figure 2.7: Translating code into hardware using Handel-C](image-url)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
hardware target, Handel-C emits the design in *Electronic Design Interchange Format* (EDIF) format. A cycle count is available from the simulator, and an estimate of gate count is generated by the Handel-C compiler. To get definitive timing information and actual hardware usage, the place and route tools need to be invoked.

### 2.4 RC1000 PCI Board

Figure 2.8 is a block diagram of the full length RC1000 PCI board is used in this thesis. The RC1000 is a PCI bus plug-in card for PC's. It has one large Xilinx Virtex E FPGA, four 2MB banks of memory for data processing operations, a programmable clock and 50 auxiliary I/Os.

All four memory banks are accessible by both the FPGA and any device on the PCI bus. A FPGA has two of its pins connected to clocks. One pin is connected to either a programmable clock or an external clock. The programmable clocks are programmed by the host PC, and have a frequency range of 400kHz to 100MHz. The RC1000 FPGA can be programmed from the host PC over the PCI bus.

### 2.5 Optimization Techniques

Handel-C is a C based hardware description language. In Handel-C registers are implemented using flip-flops and all other circuitry is made up of logic gates. Each of the logic gates in the circuit has delay associated with it as the inputs propagates through the outputs. Optimization [20] is the main part while modeling hardware to reduce the propagation delay and to exploit parallelism and pipelining. The following techniques are followed in this work for hardware implementation of the image processing algorithms.
2.5.1 Parallelism

Exploiting the potential parallelism of the program and then run different non-conflicting operations at the same clock cycle to acquire speed up. On FPGA's by designing specific hardware many operations can be run in parallel, significant speed up can be obtain. This is the main reason why application on FPGA can sometimes run faster than the software version even though the FPGA hardware run at much slower clock speed. Figure 2.9 shows the basic parallelism.
2.5.2 Longest Path Delay

Reducing the longest path delay is important because the hardware clock speed will at most be the same as the path with longest delay. By reducing the delay, it can make sure

```
{ 
  a = b * c; cycle 1
  k = l * m; cycle 2
  y = x *z; cycle 3
  a = a * k; cycle 4
  y = x + y; cycle 5
}
```

The above operations take 5 cycles.

```
Par{
  a=b*c; cycle 1
  k=l*m;
  y=x*z;
}
par{
  a = a *k;
  y = x+y;
}
```

Using parallelism it takes 2 cycles.

Figure 2.9  Basic parallelism

that parallel optimization will be optimal in the later stages. The delay of a path can be defined as

\[
T_{\text{delay}} = T_{\text{logic}} + T_{\text{routing}}
\]

Where \( T_{\text{delay}} \) is the total delay of the path.

\( T_{\text{logic}} \) is the delay due to logic.

\( T_{\text{routing}} \) is the delay due to routing

Therefore, reducing the delay is done by reducing one of the \( T_{\text{logic}} \) or \( T_{\text{routing}} \) or both.

In hardware a complex operation require longer clock to complete. Longest path delay can be reduced by breaking up complex operations into several simpler operations. This step effectively reduces the logic in each operation thus reduce the \( T_{\text{logic}} \). Figure 10 shows an example of breaking up a complex operation into simple operations at the cost
of extra hardware resources.

<table>
<thead>
<tr>
<th>Complex Operation</th>
<th>Simpler Operation</th>
</tr>
</thead>
</table>
| \( \text{sum} = a \times b + c \times d + e \times f \); | \( T_1 = a \times b; \)  
| | \( T_2 = c \times d; \)  
| | \( T_3 = e \times f; \)  
| | \( \text{Sum} = T_1 + T_2 + T_3 \) |

Figure 2.10  Breaking Up complex Operations

2.5.3 Pipelining

Pipelining is an implementation technique whereby multiple tasks are overlapped in execution. Ideally next task is started after every clock cycle. When the pipeline is full, the throughput will be a task per cycle in regardless of how many cycles it takes for the task to finish.

2.6 Prior Related Work

Richard G.S [8] discusses the idea of parameterized program generation of convolution filters in an FPGA. A 2-D filter is assembled from a set of multipliers and adders, which are in turn generated from a canonical serial-parallel multiplier stage. Atmel application notes [9] discuss 3x3 convolver with run-time reconfigurable vector multiplier in Atmel FPGA. Ernest and Wiatr [22] discussed a method for development of an automated tool for generating convolution in FPGAs. Lorca, Kessal and Demigny [12] proposed a new organization of filter at 2D and 1D levels, which reduces the memory size and the computation cost by a factor of two for both software and hardware implementations. Fahad Alzahrani and Tom Chen [13] present high performance edge
detection VLSI architecture for real time image processing applications, the architecture is fully pipelined. It is capable of producing one edge-pixel every clock cycle at a clock rate of 10 MHz, the architecture can process 30 frames per second. V.Gemignani, M. Demi, M Paterni, M Giannoni and A Benassi [10] presents the real time implementation of two mathematical operators which are commonly used to detect edges: (i) gradient of Gaussian,(ii) b operator a new operator. The algorithms are implemented on digital signal processor.
CHAPTER 3

IMAGE PROCESSING ALGORITHMS

This thesis is focused on developing hardware implementation of image processing algorithms on FPGA using Handel-c. This chapter discusses the following basic image processing algorithms which include:

i) Median Filter,

ii) Basic Morphological Operators,

iii) Convolution,

iv) Edge Detection.

3.1 Median Filtering

A Median filter is a non-linear digital filter which is able to preserve sharp signal changes and is very effective in removing impulse noise (or salt and pepper noise)[1]. An impulse noise has a gray level with higher low that is different from the neighborhood point. Linear filters have no ability to remove this type of noise without affecting the distinguishing characteristics of the signal; median filters have remarkable advantages over linear filters for this particular type of noise. Therefore median filter is very widely used in digital signal and image/video processing applications.

A standard median operation is implemented by sliding a window of odd size (e.g. 3x3 window) over an image. At each window position the sampled values of signal or
image are sorted, and the median value of the samples is taken as the output that replaces the sample in the center of the window as shown in Figure 3.1.

The main problem of the median filter is its high computational cost (for sorting N pixels, the time complexity is $O(N \log N)$, even with the most efficient sorting algorithms). When the median filter is carried out in real time, the software implementation in general-purpose processors does not usually give good results. For this reason, FPGAs are used in the real-time implementation of a median filter.

The initial version of the median filter is programmed using VC++ on PC, so that its operation could be verified and its results could be compared to the hardware version. The following lines represent the pseudo-code of median filter.

For $x = \text{number of rows}$

For $y = \text{number of columns}$.
Window_Array = Array consisting of current window pixels

Output_image(x,y) = Median(window_Array);

End

End

Software implementation works by using for loops to simulate a moving window of pixel neighborhoods. For every shift of the window, the algorithm creates a sorted list of the pixel values in ascending order or descending order and a middle value is picked from the sorted list. The output of the program is an image consisting of the median values of the moving window on an image. Since a 3x3 window is used in median filter implementation, the output is dependent on the pixels from neighboring rows. The result of this is that some edge effects occur in the output image, meaning that there is always

(a) Original Image

(b) Filtered Image

Figure 3.2 Median Filter on an image
an invalid pixel along the borders of the output image. This is true for all algorithms using windowing approach in image processing. The Figure 3.2 shows noisy gray scale lena image(a) image filtered by a median filter (b).

3.2 Morphological Operators

The term morphological image processing [17] refers to a class of algorithms that is interested in the geometric structure of an image. Morphology can be used on binary and gray scale images, and is useful in many areas of image processing, such as skeletonization, edge detection, restoration and texture analysis.

A morphological operator uses a structuring element to process an image as shown in Figure 3.3. The structuring element is a window scanning over an image, which is similar to the pixel window used in the median filter. The structuring element can be of any size, but 3x3 and 5x5 sizes are common. When the structuring element scans over an element in the image, either the structuring element fits or does not fit. Figure 3.3 demonstrates the concept of a structuring element fitting and not fitting inside an image object.

The most basic building blocks for many morphological operators are erosion and dilation [20]. Erosion as the name suggests is shrinking or eroding an object in an image. Dilation on the other hand grows the image object. Both of these objects depend on the structuring element and how it fits within the object. For example, if erosion is applied to an binary image, the resultant image is one where there is a foreground pixel for every center pixel where its structuring element fit within an image. If dilation is applied, the output will be a foreground pixel for every point in the structuring element.
Important operations like opening and closing of an image can be derived by performing erosion and dilation in different order. If the erosion is followed by dilation, the resulting operation is called an opening. Closing operation is dilation followed by erosion. These two secondary morphological operations can be useful in image restoration, and their iterative use can yield further interesting results such as; skeletonization of an input image.

While morphological operations usually are performed on binary images, some processing techniques also apply to grayscale images. These operations are for the most part limited to erosion and dilation. Grayscale erosions and dilations produce results identical to the nonlinear minimum and maximum filters.

In a minimum filter, the center pixel in the moving window is replaced by the smallest pixel value. This has the effect of causing the bright areas of an image to shrink,
or erode. Similarly, grayscale dilation is performed by using the maximum operator to select the greatest value in a window.

Figure 3.4 Erosion and Dilation of a Gray Scale Image
The basic morphological operators are programmed using VC++, so that its operations could be verified and its results could be compared to the hardware version. Since grayscale erosion and dilation are minimum and maximum filters respectively, the similar algorithm as median filtering is used. Instead of selecting the middle value from a sorted list of window pixels, a minimum value is selected for minimum filter, that is, erosion and a maximum value is selected for dilation. Figure 3.4 show the output of an erosion and a dilation applied on gray scale image.

3.3 Convolution

Convolution is a simple mathematical operation which is fundamental to many common image processing operators. Convolution is a way of multiplying together two arrays of numbers of different sizes to produce a third array of numbers. In image processing the convolution is used to implement operators whose output pixel values are simple linear combination of certain input pixels values of the image. Convolution belongs to a class of algorithms called spatial filters. Spatial filters use a wide variety of masks, also known as kernels, to calculate different results, depending on the desired function.

3.3.1 D Convolution

The convolution operation is a mathematical operation which takes two functions \( f(x) \) and \( g(x) \) and produces a third function \( h(x) \). Mathematically, convolution is defined as:

\[
h(x) = f(x) * g(x) = \int_{-\infty}^{\infty} f(\tau)g(x-\tau)d\tau
\]

where \( g(x) \) is referred to as the filter.
3.3.2 2D-Convolution

2D-Convolution, is most important to modern image processing. The basic idea is that a window of some finite size and shape is scanned over an image. The output pixel value is the weighted sum of the input pixels within the window where the weights are the values of the filter assigned to every pixel of the window. The window with its weights is called the convolution mask. Mathematically, convolution on image can be represented by the following equation.

\[
y(m,n) = \sum_{i=0}^{\text{width of image}} \sum_{j=0}^{\text{height of image}} h(i, j) x(m-i, n-j),
\]

where \( x \) is the input image, \( h \) is the filter and \( y \) is the image.

An important aspect of convolution algorithm is that it supports a virtually infinite variety of masks, each with its own feature. This flexibility allows many powerful applications. 3x3 convolution masks are most commonly used. For example the derivative operators which are mostly used in edge detection use 3x3 window kernels. They operate only a pixel and its directly adjacent neighbors. Figure 3.5 shows a 3x3 convolution mask operated on an image. The center pixel is replaced with the output of the algorithm; this is carried for the entire image. Similarly larger size convolution masks can be operated on an image.

To illustrate the convolution algorithm, Gaussian convolution filters are chosen. It is used to blur images. The Gaussian distribution in 1-D has the form:

\[
G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}
\]

where \( \sigma \) is the standard deviation of the distribution.
In 2-D, a circularly symmetric Gaussian has the form

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

(3.4)

The idea of Gaussian convolution is to use this 2-D distribution as a point spread function, and this is achieved by convolution. Since the image is stored as a collection of discrete pixels. A discrete approximation to the Gaussian function is required to perform the convolution. In theory, the Gaussian distribution is non-zero everywhere, which would require an infinitely large convolution kernel, but in practice it is effectively zero more than about three standard deviations from the mean, and so convolution kernel is truncated as shown in Figure 3.6.

Software version of the 5x5 Gaussian convolution on an image is implemented in VC++. The results obtained in software version are used to compare the results obtained in hardware version. Figure 3.7 shows the images obtained after applying the 5x5 Gaussian convolution kernels.
3.4 Edge Detection

Edges are places in the image with strong intensity contrast. Edges often occur at image locations representing object boundaries; edge detection is extensively used in
image segmentation. Representing an image by its edges has the further advantage that the amount of data is reduced significantly while retaining most of the image information.

Edges can be detected by applying a high pass frequency filter in the Fourier domain or by convolving the image with an appropriate kernel in the spatial domain. In practice, edge detection is performed in the spatial domain, because it is computationally less expensive and often yields better results. Since edges correspond to strong illumination gradients, the derivatives of the image are used for calculating the edges.

The Canny edge detection algorithm [11] is considered a "standard method" and it is used by many researchers, because it was designed to be an optimal edge detector and thin edges. The Canny operator works in a multi-stage process. Canny edge detection uses linear filtering with a Gaussian kernel to smooth noise and then computes the edge strength and direction for each pixel of the smoothed image. This is done by differentiating the image in two orthogonal directions and computing the gradient magnitude as the root sum of squares of the derivatives. The gradient direction is computed using the arctangent of the ratio of the derivatives. Candidate edge pixels are identified as the pixels that survive a thinning process called non-maximal suppression. In this process, the edge strength of each candidate edge pixel is set to zero if its edge strength is not larger than the edge strength of the two adjacent pixels in the gradient direction. Thresholding is then done on the thinned edge magnitude image using hysteresis. In hysteresis, two edge strength thresholds are used. All candidate edge pixel values below the lower threshold are labeled as non-edges, and the pixels values above the high threshold are considered as definite edges. All pixels above low threshold that an
be connected to any pixel above the high threshold through a chain are labeled as edge pixels. The schematic of the canny edge detection is shown in Figure 3.8.

3.4.1 Smoothing

The Gaussian distribution in 1-D has the form:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{x^2}{2\sigma^2}}$$  \hspace{1cm} (3.5)

Where $\sigma$ is the standard deviation of the distribution.

In 2-D, a circularly symmetric Gaussian has the form

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$  \hspace{1cm} (3.6)

The idea of Gaussian convolution is to use this 2-D distribution as a point spread function, and this is achieved by convolution. Since the image is stored as a collection of discrete pixels. A discrete approximation to the Gaussian function is required to perform the convolution. In theory, the Gaussian distribution is non-zero everywhere, which would require an infinitely large convolution kernel, but in practice it is effectively zero more than about three standard deviations from the mean, and so convolution kernel is truncated. The convolution kernel of standard deviation($\sigma$) 1.4 is used for smoothing in this thesis as shown in Figure 3.9.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 3.9  5x5 Gaussian convolution mask

The effect of Gaussian convolution is to blur an image. The degree of smoothing is
determined by the standard deviation of the Gaussian.

3.4.2 Gradient Calculation

After smoothing the image and eliminating the noise, the next step is to find the edge
strength by taking the gradient of the image. Most edge detection methods work on the
assumption that an edge occurs where there is a discontinuity in the intensity function or
a very steep intensity gradient in the image as shown in Figure 3.10.

Most edge-detecting operators can be thought of as gradient-calculators. Because the
gradient is a continuous-function concept and images are discrete functions, we have to
approximate it. Since derivatives are linear and shift-invariant, gradient calculation is
most often done using convolution. Numerous kernels have been proposed for finding
edges, some of the kernels are: Roberts Kernel, Kirsch Compass Kernel, Prewitt Kernel,
Sobel Kernel, and many others.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The Prewitt kernels are based on the simple idea of the central difference between rows for horizontal gradient and difference between columns for vertical gradient.

\[
\frac{\partial I}{\partial x} = \frac{I(x+1,y) - I(x-1,y)}{2}, \quad \text{and} \quad \frac{\partial I}{\partial y} = \frac{I(x,y+1) - I(x,y-1)}{2} \tag{3.7}
\]

The following convolution masks are derived from equations.

<table>
<thead>
<tr>
<th>Horizontal Convolution</th>
<th>Vertical Convolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 -1 0</td>
</tr>
<tr>
<td>-1 0 1</td>
<td>0 0 0</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 1 0</td>
</tr>
</tbody>
</table>

These convolutions are used for calculating the horizontal and vertical gradients.

3.4.3 Magnitude and Phase

Convolution of the image with horizontal and vertical gradients produces horizontal gradient (dx) and vertical gradient (dy) respectively. The absolute gradient magnitude (|G|) is calculated by the mean square root of the horizontal (dx) and vertical (dy)
gradients. That is, \( |G| = \sqrt{dx^2 + dy^2} \). To reduce the computational cost of magnitude, it is often approximated with absolute sum of the horizontal and vertical gradients (\( |G| = |dx| + |dy| \)).

The direction of the gradient (\( \phi \)) is calculated by arctangent of the vertical gradient to the horizontal gradient. \( \theta = \arctan(dy/dx) \).

3.4.4 Non-Maximum Suppression

With the magnitude and direction obtained from previous stage one can apply the thresholding operation in the gradient-based method and end up with ridges of edge pixel. To get rid of ridges, the edge strength of each candidate edge pixel is set to zero if its edge strength is not larger than the edge strength of the two adjacent pixels in the gradient direction. This is called thinning process.

3.4.5 Threshold

The output image of non-maximum suppression stage may consist of broken edge contours, single edge points which contribute to noise. This can be eliminated by thresholding with hysteresis. Two thresholds are considered for hysteresis, one high threshold other low threshold. If any edge response is above a high threshold, those pixels constitute definite edge output of the detector for a particular scale. Individual weak responses usually correspond to noise, but if these points are connected to any of the pixels with high threshold, they are more likely to be actual edges in the image. Such connected pixels are treated as edge pixels if their response is above a low threshold.

To get thin edges two thresholds (high threshold (\( T_H \)) and low threshold(\( T_L \)) ) are used. If the gradient of the edge pixel is above the \( T_H \), it is considered as an edge pixel. If the gradient of the edge pixel is below \( T_L \) then it is unconditionally set to zero. If the
gradient is between these two, then it is set to zero unless there is a path from this pixel to a pixel with a gradient above $T_H$; the path must be entirely through pixels with gradients of at least $T_L$.

First a software model is designed and programmed in VC++, so that its operation could be verified and its results could be compared with software. Figure 3.11 (b) shows the canny edge detection of a 256 x 256 gray scale lena image.

![Figure 3.11 Edge Detection of Gray Scale Image](image-url)
CHAPTER 4

HARDWARE IMPLEMENTATION

This chapter explains in detail the reconfigurable hardware implementation of image processing algorithms discussed in (Chapter 3) on a Xilinx Virtex-E FPGA. The algorithms implemented are:

- Median Filtering,
- Basic Morphological Operations (Erosion and Dilation),
- Convolution,
- Edge Detection of an image.

First the implementation of the moving window operator which form the basis of these algorithms is explained.

4.1 Moving Window Operator

The algorithms implemented in this work use the moving window operator. The moving window operator usually process one pixel of the image at a time, changing its value by some function of a local region of pixels (covered by the window). The operator moves over the image to process all the pixels in the image. In this section a 3x3 moving window used for the median filtering, morphological and edge detection algorithms and a 5x5 moving window used in Gaussian smoothing filter operation are explained.
For the pipelined implementation of image processing algorithms all the pixels in the
moving window operator must be accessed at the same time for every clock. In order to
access all the pixels in a moving window system, a design was devised that took
advantage of certain features of FPGAs. The First In First Out (FIFO) buffers are used to
create the effect of moving an entire window of pixels through the memory for every
clock cycle. A FIFO consists of a block of memory and a controller that manages the
traffic of data to and from the FIFO. The FIFO’s are implemented using circular buffers
constructed from multi-port block RAM with an index keeping track of the front item in
the buffer. The availability of multi-port block RAM in the Xilinx Vertex-E FPGA helps
in achieving the read and write operations of the RAM in the same clock cycle. This
allows a throughput of one pixel per clock cycle. The same effect can be achieved using
double-width RAMs implemented in lookup tables on the FPGA. However, the use of
block RAMs is more efficient and has less associated logic for reading and writing.

For a 3x3 moving window two FIFO buffers are used. The size of the FIFO buffer is
given as W-3, where W is the width of the image. To access all the values of the window
for every clock cycle the two FIFO buffers must be full. Figure 4.1 shows the architecture
of the 3x3 moving window. For every clock cycle, a pixel is read from the RAM and
placed into the bottom left corner location of the window. The contents of the window are
shifted to the right, with the rightmost member being added to the tail of the FIFO. The
top right pixel is disposed after the computation on the pixels is completed, since it is not
used in future computation.

Similarly for a 5x5 window operation four FIFO buffers are used. Each FIFO size is
W-5, where W is width of the image. To access the values in the moving window in
every clock cycle the four FIFO buffers must be full. Figure 4.2 shows the architecture of 5x5 moving window. For every clock cycle, a pixel read from the RAM is placed into the bottom left corner location of the window. The contents of the window are shifted to the right, with the rightmost member being added to the tail of the FIFO. The top right pixel is disposed after the computation of the pixels is computed.
4.2 Median Filter

A median filter is implemented by sliding a window of odd size on an image. A 3x3 window size is chosen for implementation for median filter, because it is small enough to fit onto the target FPGA's and is considered large enough to be effective for most commonly used image sizes. The median filter uses the 3x3 window operation discussed in Section 4.1. The median filtering operation sorts the pixel values in a window in ascending order and picks up the middle value, the center pixel in the window is replaced by the middle value. The most efficient method of accomplishing this is with a system of hardware compare/sort units, which allows sorting a window of pixels into an ascending order.

The sorting can simply accomplished in Handel-c by the “if /else” statement.

```
If( wx1 < wx2)
{
  par{
    Cx1_L = wx1;
    Cx1_H = wx2;
  }
} else
{
  par{
    Cx1_L = wx2;
    Cx1_H = wx1;
  }
}
```

The hardware design of the median filter is shown in Figure 4.4. The Cxx represents a compare unit and Rxx represent a register to store the intermediate values. Functionality of the Compare unit is shown in Figure 4.3
The median filter algorithm used in this design is pipelined to produce one median value for every clock cycle. The output for a given input appears at the other end of the pipe after an initial latency. The median filtering algorithm uses a complex pipelining to accelerate the flow of data.

The pipelined architecture of a median filter of an image have four processes as shown in Figure 4.4. They are as follows:

- **Reading process**: Reads the image pixels from the external RAM.
- **Buffer Process**: Copies information from the reading process into the FIFO buffers and shifts data through the FIFO buffer at the desired time instant, so that valid data is placed at the input and collected at the output.
- **Filtering Process**: Processes the data at the desired time instant from the FIFO buffers and passes the filtered data to the writing process.
- **Writing process**: Reads output from the filtering process and writes it into the external RAM.

In pipelining each process generates the data that is passed to the subsequent process and that process is blocked until the data is available. Initially, the buffer process takes the data from the reading process. The first output of buffer process is blocked until the
Figure 4.4 Hardware Design for Sorting Algorithm

FIFO buffer is full, after which the filtering process takes data as input from the buffer process. The first output of the filtering process is subsequently blocked until the filtering is done. The writing process starts writing the data into the external RAM once it gets data from the filtering process. The pipelining process is shown in Figure 4.5.

Median filtering at the border of the image is handled by placing the zero value pixels around the borders. To perform this process, the two counters are used to track the borders.
The median filtering applied on a 256 x 256 Lena image was implemented using the pipeline design on Xilinx Vertex-E FPGA. Memory read has a latency of two cycles while memory writes are completed in the same cycle. So Reading process has a latency of two cycles. Filtering process to be ready it takes W+2 clock cycles, where W is the width of the image. The output of the filtering process is available after a latency of fourteen cycles. The writing process writes the filtered pixels into the RAM in one cycle. Since it is a design is pipelined the output is produced on every clock cycle. The median filter implementation with this pipelined architecture on FPGA requires far less cycles then the same algorithms implemented on general purpose computer.

Figure 4.5 Pipelining Process

4.3 Basic Morphological Operators

The basic morphological operators are erosion and dilation. The erosion and dilation of a grayscale image are called grayscale erosion or dilation. The grayscale erosion is performed by minimum filter, whereas the dilation is performed by maximum filter. In a 3x3 minimum filter, the center pixel is replaced by a minimum value of the pixels in the window. In a maximum filter, the center pixel is replaced by a maximum value of the pixels in the window. The implementation of minimum and maximum filters is similar to the median filters implementation.
4.3 Convolution

Convolution is one of the basic and common operations on images. It uses a window operator discussed in Section 4.1. The center pixel in the window is weighted sum of the input pixels within the window divided by the sum of the weights in the window, where the weights are the values of the filter assigned to every pixel of the window.

Convolution is a very complex operation, requires lot of computational power. To calculate a pixel for a given mask of size \( m \times n \), \( m \times n \) multiplications, \( m \times n - 1 \) additions and one division are required. So to perform a 3x3 convolution on a 256 x 256 gray scale image, 589824 multiplications, 393216 additions and one division are required.

Multiplication and division operators produce the deepest logic. A single cycle divide, or multiplication produces a large amount of hardware and long delays through deep logic. In order to improve the performance of the convolution operation, it is necessary to reduce the multiplication and division operators. Multiplication and division can be done using bit shifting, but this is only possible with the powers of 2's. Multiplier less multiplication can be employed to do multiplication of non power of 2's digits, where multiplication is done with only shifts and additions from the binary representation of the multiplicand. For example, A multiplied by B = 14 = 1110_2 \ (A \times B) can be implemented as \ (A \ll 1 + A \ll 2 + A \ll 3) \), where ‘\ll’ denotes a shift to the left.

The coefficient values in the convolution mask employed in most of the image processing applications remains constant for the entire processing. Constant Coefficient Multiplier (KCMs) can be employed. A KCM comprises of Look Up Tables (LUTs) and adders. The constant value \( k \) is multiplied by first 15 whole numbers and was stored in LUTs as shown in Figure 4.7. To get the output of a 8 bit number multiplication with a 8

44

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
bit constant, the 8 bit number is split into two 4 bit values, each addressing a different LUT to produce a 12 bit value. The 12 bit values are combined to produce a 16 bit output. The KCMs can be implemented on FPGA very efficiently, because of its LUT based architecture.

![Diagram of 8 bit Constant Coefficient Multiplier (KCM)](image)

**Figure 4.6** 8 bit Constant Coefficient Multiplier (KCM)

In this section a 3x3 and 5x5 Gaussian convolution Alter of standard deviation 1.4 (σ = 1.4) is shown in Figure 4.7 is implemented. In this smoothing window there are no negative numbers; convolution can be calculated using only the unsigned numbers.
To apply a 5x5 Gaussian convolution of an image, a 5x5 moving window operator is used. A pipelined implementation is carried out for Gaussian convolution of an image. In order to access all the pixels in the window in a single clock cycle, four 8 bit FIFO buffers are used. Since the convolution mask is fixed for the whole image a dedicated hardware can be designed. Some of the window coefficients are contains multiple of 2. The multiplication of these coefficients with the corresponding pixels in the window can be carried out using left shift operations and the non powers of 2 digits can be implemented using multiplier less multiplication. An important property of a symmetrical 5x5 window coefficients as shown in Figure 4.7 is that it allows pre-addition of certain input values before any multiplication takes place. This property is used in the implementation of convolution operation, which reduces the number of multiplication operations from 25 to 3 multiplications. Some of the multiplications are power of 2, they are done by left shift operation.
Division is also very expensive operation on FPGA, instead of using division operator it is much simple to use right shift operator, so a divide by 128 was implemented instead of divide by 115.

Longest path delay associated by the wide adders is large; delay associated with the carry ripple is more. Therefore short adders are employed, which can be done in parallel.

Convolution at the border of the image is handled by placing the zero value pixels around the borders. To perform this process, the two counters are used to track the borders.

The 3x3 convolution mask shown in Figure 4.7 does not have coefficient powers of 2. Direct multiplication is a complex operation on FPGA. Since the window coefficients are constant for entire image, a KCM based multiplication approach can be employed. The multiplication tables of 0 to 16 times of the window coefficients are stored in ROMs. When a 8 bit pixel is multiplied by a 8 bit constant. The 8 bit pixel is split to two 4-bit values to address the two ROMs, a 12 bit result is produced. The two 12 bit results are combined to produce a 16 bit output as shown in Figure 4.8. On FPGAs two locations of the ROM cannot be accessed in parallel. For efficient implementation, two ROMs are created to store the multiplication tables of a constant, so that the ROM can be accessed in parallel. In 3x3 Convolution

Convolution at the border of the image is handled by placing the zero value pixels around the borders. To perform this process, the two counters are used to track the borders.

The 3x3 convolution is implemented by using direct multiplication and look Up based multiplication and the results are analyzed.
4.4 Edge Detection

Hardware implementation of canny edge detection algorithm is discussed in this section. Canny edge detector operation consists of four stages:

- Image smoothing.
- Vertical and Horizontal Gradient Calculation.
- Directional Non Maximum Suppression.
- Threshold.

Classically, the image smoothing is implemented by applying the Gaussian convolution on the entire image. Furthermore, the smoothened image is used as an input to calculate the gradient at every pixel, and these gradient values are used to calculate the phase and the magnitude for each pixel, which is followed by non-maximum suppression.

To get rid of ridges, the edge strength of each candidate edge pixel is set to zero if its edge strength is not larger than the edge strength of the two adjacent pixels in the gradient direction. This is called non-maximum suppression. Two threshold (High threshold & Low threshold) values are used get the connected edge pixels. This is called hysteresis.

On a general purpose computer the four stages of the canny edge detection are performed sequentially on the entire image, one stage followed by other stage. This

Figure 4.8 Design Flow of Edge Detection
approach on FPGA require lot of hardware resources and design is slow. In order to efficiently use the hardware resources and increase the speed, hardware features like parallelism and pipelining are employed. A pipelined architecture shown in Figure 4.9 is designed.

![Pipelined Architecture](image)

Since output in each stage depend on the neighboring pixels, a moving window operator discussed in Section 4.1 is adopted.

4.4.1 Image Smoothing

Smoothing of the image is achieved by 5x5 Gaussian convolutions, as mentioned in Section 4.3. A 5x5 moving window operator is used, four FIFO buffers are employed to access all the pixels in the 5x5 window at the same time. Since the design is pipelined,
the Gaussian smoothing starts once the 2 FIFO buffers are full. That is, the output is produced after a latency of twice width of image plus two (2*width +2) cycles. The output of this stage is given as input to the horizontal and vertical gradient calculation stage.

4.4.2 Vertical and Horizontal Gradient Calculation

This stage calculates the vertical and horizontal gradients using 3x3 convolution kernels shown in Figure 4.10. An 8-bit pixel in row order of the image produced during every clock cycle in the image smoothing stage is used as the input in this stage. Since 3x3 convolution kernels are used to calculate the gradients, neighboring eight pixels are required to calculate the gradient of the center pixel and the output pixel produced in previous stage is a pixel in row order. In order to access eight neighboring pixels in a single clock cycle, two FIFO buffers are employed to store the output pixels of the previous stage.

The gradient calculation introduces negative numbers. In Handel-C, negative numbers can be handled easily by using signed data types. Signed data means that a negative number is interpreted as the 2’s complement of number. In this design, an extra bit is used for signed numbers as compared to unsigned 8 bit numbers i.e. 9 bits are used.

<table>
<thead>
<tr>
<th>Horizontal Convolution</th>
<th>Vertical Convolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 -1 0</td>
</tr>
<tr>
<td>-1 0 1</td>
<td>0 0 0</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 1 0</td>
</tr>
</tbody>
</table>

Figure 4.10 Gradient Convolution Kernels
to represent a gradient output instead of 8. Two gradient values are calculated for each pixel, one for vertical and other for horizontal. The 9 bits of vertical gradient and the 9 bits of the horizontal gradient are concatenated to produce 18 bits. Since the whole design is pipelined, an 18 bit number is generated during every clock cycle, which forms the input to the next stage.

4.4.3 Directional Non Maximum Suppression

Directional non maximum suppression works with the magnitude and orientation of the gradient of the pixel under consideration and creates edges of one pixel-width. The values of each component of the gradient obtained from the previous stage are used to get the magnitude and direction. The direction of the gradient is calculated mathematically as the arctangent of vertical gradient component over the horizontal gradient component (\(\text{direction} = \arctan\left(\frac{dy}{dx}\right)\)). Since arctangent is a very complex function and also requires floating point numbers, it is very difficult to implement such functions on FPGA. Instead, the value and sign of the components of the gradient is analyzed to calculate the direction of the gradient. If the current pixel is \(P_{x,y}\) and the values of the derivatives at that pixel are \(dx\) and \(dy\), the direction of the gradient at \(P\) can be approximated to one of the sectors shown in the Figure. 4.11.

Once the direction of the gradient is known, the values of the pixels found in the neighborhood of the pixel under analysis are interpolated. The pixel that has no local maximum gradient magnitude is eliminated. The comparison is made between the actual pixel and its neighbors, along the direction of the gradient.
or example, if the approximate direction of the gradient is between $0^\circ$ and $45^\circ$, the magnitude of the gradient at $P_{x,y}$ is compared with the magnitude of the gradient at adjacent points as shown in Figure 4.12. where $P_{x,y} = |dx_{x,y}| + |dy_{x,y}|$.

The values of the Gradient at the point $P_a$ and $P_b$ are defined as follows.

$$P_a = \frac{P_{x+y-1} + P_{x+y}}{2}, \text{where } P_{x+y-1} = |dx_{x+y-1}| + |dy_{x+y-1}| \text{ and } P_{x+y} = |dx_{x+y}| + |dy_{x+y}| \quad (4.1)$$

$$P_b = \frac{P_{x+y+1} + P_{x+y}}{2}, \text{where } P_{x+y+1} = |dx_{x+y+1}| + |dy_{x+y+1}| \text{ and } P_{x+y} = |dx_{x+y}| + |dy_{x+y}| \quad (4.2)$$
The center pixel $P_{x,y}$ is considered as an edge, if $p_{x,y} > p_a$ and $p_{x,y} > p_b$. If both conditions are not satisfied center pixel is eliminated.

Since the gradient calculation as explained depends on the direct neighboring pixels, a 3x3 window operator is used. The output of the previous stage is used as input in this stage. The output produced in the previous stage is a 18 bit number, first nine bits are horizontal gradient and other nine bits are vertical gradient. In order to access all the pixels in the 3x3 window at the same time two eighteen bit FIFO buffers of width of the image minus three array size are employed. To calculate the phase and magnitude at every pixel the horizontal and vertical gradient values derived from the eighteen bit number are used. The output produced in this stage is given as input to the threshold stage.

4.4.4 Threshold

The output image of non-maximum suppression stage may consist of broken edge contours, single edge points which contribute to noise. This can be eliminated by thresholding. To get thin edges two thresholds (high threshold ($T_{IH}$) and low threshold($T_{IL}$)) are used. If the gradient of the edge pixel is above the $T_{IH}$, it is considered as an edge.

![Figure 4.12 Pixel Interpolation](image-url)

The center pixel $P_{x,y}$ is considered as an edge, if $p_{x,y} > p_a$ and $p_{x,y} > p_b$. If both conditions are not satisfied center pixel is eliminated.
pixel, lets call it as *definite edge*. If the gradient of the edge pixel is below $T_L$ then it is unconditionally set to zero. If the gradient is between these two, then it may be an edge pixel, lets call it *maybe edge* pixel. It is set to zero unless there is a path from this pixel to a pixel with a gradient above $T_H$; the path must be entirely through pixels with gradients of at least $T_L$.

To get the connected path from the *maybe edge* pixel and the *definite edge* pixel, a 3x3 window operator is used. If the center pixel is an *definite edge* pixel and any of the neighbors is a *maybe edge* pixel, then maybe pixel is considered as a definite edge pixel. And also, if the center pixel is a *maybe* pixel and any of the neighbors is definite pixel, then maybe pixel is considered as a definite edge pixel. The resultant image is an image with thin sharp edges.

Since the design is pipelined each output pixel produced is written to RAM on every clock cycle.

The 256 x 256 gray scale Lena image was considered as a bench mark to implement the basic image processing algorithms like Median filter, basic morphological operators, 5x5 Gaussian convolution and canny edge detection processed on FPGA. In Figure 4.13 (b) shows the image obtained by processing (a) salt and pepper noise Lena image on FPGA. Figure 4.13 (c) and (d) shows the erosion and dilation of the original Lena image processed on FPGA respectively. Figure 4.13 (e) shows the 5x5 Gaussian convolution of image and (d) shows the edge detection of lena image implemented on FPGA.
Figure 4.13 Lena image processed on FPGA

(a) Lena (salt & pepper noise)  (b) Median filtered image
(c) Erosion of lena image  (d) Dilation of Lena Image
(e) 5x5 Gaussian Convolution  (f) Edge detection
CHAPTER 5

RESULTS AND FUTURE WORK

The image processing algorithms were simulated and synthesized using Handel-C hardware description language using the Celoxica DK2 environment. The Handel-C produces an Electronic Design Interchange Format (EDIF) output when compiling the design for hardware target. The Xilinx placement and routing tools are used to translate the EDIF format into hardware layout (bit format file).

A host side program on the PC was written using VC++ for FPGA configuration and communication with RC1000 board. The RC1000 device driver routines provided by the board vendor are employed to accomplish this task. Once the FPGA is configured, the host program requests the ownership of the SRAM memory bank on the RC1000 board. The image to be processed is loaded into SRAM memory bank and signals the FPGA to read the image from memory bank and start processing as shown in Figure 5.1.

![Figure 5.1 Host FPGA Communication](image)

Figure 5.1 Host FPGA Communication
When the processing of the image is completed by the FPGA, it signals the host program that the processed image is in the memory bank. The host PC reads the processed image from the memory bank into its main memory and displays.

5.1 Results

Timing is an important metric when comparing the hardware and software implementation. The timing results of image processing algorithms implemented on Xilinx Vertex-E FPGA and on Pentium III are tabulated below:

<table>
<thead>
<tr>
<th>System</th>
<th>Freq [MHz]</th>
<th>Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx Vertex-E FPGA</td>
<td>25.9</td>
<td>2.56</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1300</td>
<td>51</td>
</tr>
</tbody>
</table>

It may be observed from the Table 5.1 that the time taken to implement the median filter on a 256 x 256 grayscale image on a PC with Pentium III 1300 MHz is 51 ms. The pipelined implementation of median filter of a 256 x 256 gray scale Lena image on a Xilinx Vertex-E is 2.56 ms at a clock frequency of about 34 MHz. A close comparison of the results presented reveals that implementation of median filter on Xilinx Vertex-E FPGA is 26 times faster than that on PC.
Table 5.2  Timing Result of Median Filter on a 512 x 512 gray scale image

<table>
<thead>
<tr>
<th>System</th>
<th>Freq[MHz]</th>
<th>Time[ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx Vertex-E FPGA</td>
<td>23.06</td>
<td>11.3</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1300</td>
<td>235</td>
</tr>
</tbody>
</table>

The pipelined implementation of median filter on a 512 x 512 gray scale Lena image on a Xilinx Vertex-E is 11.3 ms at a clock frequency of about 23.06 MHz. A close comparison of the results presented reveals that time taken for pipelined median filter on Xilinx Vertex-E faster than that on PC.

Table 5.3  Timing result of 5x5 Convolution on a 256 x 256 gray scale image

<table>
<thead>
<tr>
<th>System</th>
<th>Freq[MHz]</th>
<th>Time[ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx Vertex-E FPGA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct division by 115</td>
<td>25.9</td>
<td>2.62</td>
</tr>
<tr>
<td>Division using right shift( &gt;&gt; 7)</td>
<td>42</td>
<td>1.57</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1300</td>
<td>31</td>
</tr>
</tbody>
</table>

Time taken for 5x5 Gaussian convolution on 256 x 256 gray scale image by employing division by 115 on Xilinx Vertex-E FPGA is 2.62 ms, where as time taken for 5x5 Gaussian convolution by employing shift operation ( >> 7) division is 1.57 ms. Time taken for 5x5 Gaussian smoothing convolution of a 256 x 256 gray scale image on PC with PentiumIII 1300 MHz is 31 ms.
Table 5.4  
Timing result of 5x5 convolution on a 512 x 512 gray scale image

<table>
<thead>
<tr>
<th>System</th>
<th>Freq[MHz]</th>
<th>Time[ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx Vertex-E FPGA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct division by 115</td>
<td>24.8</td>
<td>10.16</td>
</tr>
<tr>
<td>Division using right shift (&gt;&gt; 7)</td>
<td>40.43</td>
<td>7.03</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1300</td>
<td>125</td>
</tr>
</tbody>
</table>

Time taken for 5x5 Gaussian convolution on 512 x 512 gray scale image by employing division by 115 on Xilinx Vertex-E FPGA is 2.62 ms, whereas time taken for 5x5 Gaussian convolution by employing shift operation (>> 7) division is 7.03 ms. On a 1.3 GHz Pentium III PC the time taken for 5x5 Gaussian smoothing convolution of a 512 x 512 gray scale image is 125 ms.

Table 5.5  
Timing Results of 3x3 Convolution on a 256 x 256 gray scale image

<table>
<thead>
<tr>
<th>System</th>
<th>Freq[MHz]</th>
<th>Time[ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx Vertex-E FPGA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct Multiplication</td>
<td>42.03</td>
<td>1.58</td>
</tr>
<tr>
<td>LUT based Multiplication</td>
<td>50.99</td>
<td>1.31</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1300</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 5.5 shows the time taken for 3x3 Gaussian smoothing on 256 x 256 gray scale image by direct multiplication and Look Up Tables (LUTs) based multiplication on
Xilinx Vertex FPGA. It was observed that the convolution performed by using LUT based multiplication is faster than convolution performed by direct multiplication.

Table 5.6  Timing Results of 3x3 Convolution on a 512 x 512 gray scale image

<table>
<thead>
<tr>
<th>System</th>
<th>Freq[MHz]</th>
<th>Time[ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx Vertex-E FPGA</td>
<td>Direct Multiplication</td>
<td>42.03</td>
</tr>
<tr>
<td></td>
<td>LUT based Multiplication</td>
<td>50.99</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1300</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 5.6 shows the time taken for 3x3 Gaussian smoothing on 512 * 512 gray scale image by direct multiplication and Look Up Tables (LUTs) based multiplication on Xilinx Vertex FPGA.

Table 5.7  Timing Result of edge detection algorithm on 256 x 256 gray scale image

<table>
<thead>
<tr>
<th>System</th>
<th>Freq[MHz]</th>
<th>Time[ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx Vertex-E FPGA</td>
<td>16</td>
<td>4.2</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1300</td>
<td>47</td>
</tr>
</tbody>
</table>

Table 5.8  Timing Result of edge detection algorithm on 512 x 512 gray scale image

<table>
<thead>
<tr>
<th>System</th>
<th>Freq[MHz]</th>
<th>Time[ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx Vertex-E FPGA</td>
<td>15.8</td>
<td>16.7</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1300</td>
<td>171</td>
</tr>
</tbody>
</table>
From Table 5.7, 5.8, it can be seen that the time taken to implement the edge detection algorithm on hardware is ten times faster than time taken on PC.

The device usage is normally reported in terms of number of look up tables, block RAMs, CLB slices, Gate Count. The device usages for the algorithms implemented in this thesis are shown in Table 5.9.

From the tables, it was observed that implementation of the algorithms on FPGA by exploiting parallelism and pipelining is faster than on a general purpose computer.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>LookUp Tables (LUTs)</th>
<th>Flip-Flops</th>
<th>Block RAMs</th>
<th>Gate Count(%)</th>
<th>CLB Slices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median, Erosion &amp; Dilation</td>
<td>856</td>
<td>298</td>
<td>2</td>
<td>1.9</td>
<td>652</td>
</tr>
<tr>
<td>5x5 Convolution div by 115</td>
<td>947</td>
<td>926</td>
<td>4</td>
<td>2.08</td>
<td>1041</td>
</tr>
<tr>
<td>5x5 Convolution div by shift (&gt;&gt; 7)</td>
<td>597</td>
<td>802</td>
<td>4</td>
<td>3.3</td>
<td>791</td>
</tr>
<tr>
<td>3x3 Convolution direct multiplication</td>
<td>535</td>
<td>479</td>
<td>2</td>
<td>4.25</td>
<td>479</td>
</tr>
<tr>
<td>3x3 Convolution LUT based Multiplication</td>
<td>384</td>
<td>428</td>
<td>2</td>
<td>4.04</td>
<td>479</td>
</tr>
<tr>
<td>Edge Detection</td>
<td>945</td>
<td>807</td>
<td>4</td>
<td>4.85</td>
<td>1820</td>
</tr>
</tbody>
</table>
5.2 Future Work

The performance of the image processing algorithms in this work is achieved by implementing the algorithm on Field-Programmable Gate Arrays (FPGAs). Hardware implementation accelerates the designs by performing the operations concurrently. On the other hand, reprogrammability of the FPGAs allows for faster and cheaper design cycle of the system compared to Application Specific Integrated Circuit (ASIC) design.

Handel-C gives the flexibility of using the components of one design to be used in different designs. Because of this, the image processing algorithms implemented in this thesis can be used in many different applications.

This work can be further extended to real time implementation of object detection. The implementation of the whole design on FPGA is tedious and time consuming. With the advances in the software tools and growing functionality and capabilities of the FPGAs, hardware software solutions can be employed, that drastically reduces the design time for high speed applications.

One of the current shortcomings of the designs presented in this thesis is the resource utilization of FPGAs. This is mainly due to the FIFO units being used in the design. FPGA resource utilization can be greatly reduced by creating FIFO buffers on external RAM. A large part of the improvement possible in this design lies in the algorithms themselves. If the kernel for the convolution design were to be changed, the convolution algorithms would have increased functionality for changing the convolution kernels on the fly.
BIBLIOGRAHY


5. Handel-C Tutorial Celoxica Ltd.

6. Xilinx Vertex™-E Field Programmable Gate Arrays (V2.4) “July 17, 2002 Production Product specification”.


9. 3x3 Convolver with Run-Time Reconfigurable Vector Multiplier in Atmel AT6000 FPGAs. “AT6000 FPGAs Application Note 1997”.


12. F. G. Lorca, L Kessal and D. Demigny Efficient ASIC and FPGA implementation of IIR filters for Real time edge detection. “International Conference on image processing(ICIP-97) volume 2, oct 1997”.

63

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


23. Advanced RAM access from Handel-C. Celoxica Ltd.
APPENDIX

In this appendix the Handel-C codes that were developed to describe the hardware for the image processing algorithms on FPGA are dealt with.

/******************************************************************************/
Function: ReadImageFromRam:

Reads data from the RAM - bank 0, when the Host program signals the FPGA, it gets the ownerships of the RAM bank 0. The process reads a pixel for every clock cycle and outputs it down the channel OutData,

/******************************************************************************/
void ReadImageFromRam(chan unsigned 8 *OutData)
{
    unsigned 8 Pixel;           // 8 bit register to store the pixel value
    unsigned 17 addr;           // 17 bit address to access the RAM.
    static signal unsigned 1 NoDataSent = 0;
    macro expr PipeLatency = 4;
    unsigned 2 DelayCounter;    //counter to delay the output
    addr = 1;

    par // execution in parallel
    {
        while(1)
        {
            //if data has been sent on this clock cycle
            if (!NoDataSent)
            {
                par // execution in parallel.
                {
                    //Read one pixel from the Memory Bank
                    PP1000ReadBank0(Pixel,0@addr);
                    addr++;
                }
            }
            else     //otherwise do nothing
                delay;
    }
if( addr == 0 )
{
    // releases the ownership of the RAM bank
    PP1000ReleaseMemoryBank(0x1);
}

// delay until the pipeline is primed
do
{
    DelayCounter++;
} while(DelayCounter!=PipeLatency-1);

// send the data to the next process in the pipeline
while(1)
{
    prialt
    {
        case *OutData ! Pixel: break;
        default: NoDataSent = 1; break;
    }
}

}// end of ReadRAM

******************************************************************************
Function: WritePixelToRam:

Writes data to the RAM - bank 1, when the processed image is written to the RAM, it releases the ownership of the RAM BANK 1 and signals the Host program. The process reads a pixel from channel and writes one pixel per clock.

******************************************************************************/
void WritePixelToRam(chan unsigned 8 *PixelInChan)
{
    unsigned 8 Pixel; // Register to store the 8 bit pixel value.
    unsigned 17 addr; // 17 bit address for memory location
    static signal unsigned 1 NoDataSent = 0;
    macro expr PipeLatency = 4;
    // counter to delay the output of data
    unsigned 2 DelayCounter;

    // ... (Rest of the code follows)
addr = 1; // initialize the address location

*PixelInChan ? Pixel; // read pixel from the channel

par{ // parallel execution
  while(1)
  {
    // if data has been sent on this clock cycle
    if (!NoDataSent)
    {
      par
      {
        // to write the pixel at a memory location
        PP1000WriteBank1(0@addr,Pixel);
        addr++;
      }
    }
    else // otherwise do nothing
    {
      delay;
    }
    if( addr == 0)
    {
      // releases the ownership of Bank 0 and Bank 1
      PP1000ReleaseMemoryBank(0x2);
      // writes the control status to the host Program
      PP1000WriteStatus(0);
      // reads the control word from the host program
      PP1000ReadControl(Reg);
      // requests the memory bank,
      PP1000RequestMemoryBank(0x3);
      readsignal = 1;
      addr = 1;
    }
  }
}

{ // delay until the pipeline is primed
  do
  {
    DelayCounter++;
  } while(DelayCounter!=PipeLatency-1);

  // read the data from the next process when the data is available
  while(1)
case *PixelInChan ? Pixel : break;

default: NoDataSent = 1; break;
}
}
}

/**
 * Declaration of 8-bit First In First Out (FIFO) buffers.
 */

t _FIFO_PIXEL_8_
{
    // multiport ram for the fifo data

    mram

    { r om PIXEL_8 Read[FIFO_WIDTH]; // read only memory
      wom PIXEL_8 Write[FIFO_WIDTH]; // write only memory
    } ramBuffer with {block = 1};

    // pointer to the front of the fifo

    unsigned INDEX_WIDTH ramBufferFront;
};

The FIFO_PIXEL_8_ is an 8 bit dual-ported RAM. One port is to read only and
other to write only. The dual-port RAM allows the read and write in same clock cycle.
In FIFO from one end the data is read and other end the data is written in the same cycle.

/*
 * Macro Procedure:
 * ReadFIFO: To read a pixel from the head of the FIFO
 */

macro expr ReadFIFO(FIFO) = FIFO.ramBuffer.Read[FIFO.ramBufferFront
== FIFO_WIDTH-1 ? 0 : FIFO.ramBufferFront+1];

/*
 * Macro Procedure:
 */
WriteFIFO: To write a Pixel to the tail of the FIFO

macro proc WriteFIFO(FIFO, Pixel)
{
    FIFO.ramBuffer.Write[FIFO.ramBufferFront] = Pixel;
}

Macro procedure: IncrementFIFOPointer
The pointer to point to the head of the FIFO

macro proc IncrementFIFOPointer(FIFO)
{
    if (FIFO.ramBufferFront != FIFO_WIDTH-1)
    {
        FIFO.ramBufferFront++;
    }
    else
    {
        FIFO.ramBufferFront = 0;
    }
}

Function: ManageBuffers

This function generates a 5x5 window on every clock cycle. Since it is pipelined architecture, the output is produced once the FIFO buffers were filled. The first output occurs after 2*W+2 clock cycles, the channel output was delayed until the pipeline is primed. Once the FIFO are full 5x5 window pixels are produced on every clock cycle

void ManageBuffers5x5(chan PIXEL_8* InPixelChan, chan WINDOW_PIXEL_8* OutWindowChan)
{
    //5 by 5 window to be processed
    static WINDOW_PIXEL_8 Window;

    //4 FIFO buffers of pixels for the four rows of the image
    static FIFO_PIXEL_8 Buffers[4];
    //input pixel
    static PIXEL_8 Pixel;
    //signal indicating whether data has been sent on a clock cycle
    static signal unsigned 1 NoDataSent = 0;
}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
//delay in clock cycles from the input to the output
macro expr PipeLatency = (WIDTH << 1 + 2);

//counter to delay the output of data while the pipeline is primed
unsigned 10 DelayCounter;

//read in the first pixel (blocks until data is available)
*InPixelChan ? Pixel;
par // parallel execution of statements
{
    while(1)
    {
        //if data has been sent this clock cycle
        if (!NoDataSent)
        {
            par
            {
                //signal for data read from the buffers
                signal unsigned sReadBuffer[5];
                //read in a new pixel from the input channel
                *InPixelChan ? Pixel;
                //read the input data from SRAM
                sReadBuffer[4] = Pixel;
                //read the data from the head of the buffer into the signal
                par (y=0; y<4; y++)
                {
                    sReadBuffer[y] = ReadFIFO(Buffers[y<-2]);
                    IncrementFIFOPointer(Buffers[y<-2]);
                }
            /*
            The four elements of the window are moved to the FIFO buffers above
            from left bottom
            */
            par(y=1; y<5; y++)
            {
                WriteFIFO(Buffers[(y-1)<-2],ReadDataWindow(Window, y, 0));
            }
        }
    }
}

//construct the window from the registers and the buffers
par(y=0; y<5; y++)
{
    WriteDataWindow(Window, y, 0,ReadDataWindow(Window, y, 1));
    WriteDataWindow(Window, y, 1,ReadDataWindow(Window, y, 2));
    WriteDataWindow(Window, y, 2,ReadDataWindow(Window, y, 3));
}
WriteDataWindow(Window, y, 3, ReadDataWindow(Window, y, 4));
    WriteDataWindow(Window, y, 4, sReadBuffer[y]);
}
}
//otherwise pause the pipeline for one clock cycle
else
    delay;
}

//delay the output until the pipeline is primed
do
{
    DelayCounter++;  
}
while(DelayCounter != PipeLatency-1);

//Send the data to the next process in the pipeline.
while(1)
{
    prialt
    {
        case *OutWindowChan ! Window: break;

        default: NoDataSent = 1; break;
    }
}
} // end of Manage Buffers

/****************************
Function : ManageBuffers3x3

This function generates a 3x3 window on every clock cycle. Since it is pipelined
architecture, the output is produced once the FIFO buffers were filled. The first output
occurs after W+2 clock cycles, the channel output was delayed until the pipeline is
primed. Once the FIFO are full 3x3 window pixels are produced on every clock cycle

/*****************************/

void ManageBuffers3x3(chan PIXEL_8* InPixelChan, chan WINDOW_PIXEL_8* OutWindowChan)
{
    //3 by 3 window to be processed

71

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
static WINDOW_PIXEL_8 Window;

// 2 FIFO buffers of pixels for the two rows of the image
static FIFO_PIXEL_8 Buffers[2];
// input pixel
static PIXEL_8 Pixel;

// signal indicating whether data has been sent on a clock cycle
static signal unsigned 1 NoDataSent = 0;

// delay in clock cycles from the input to the output
macro expr PipeLatency = WIDTH;

// counter to delay the output of data while the pipeline is primed
unsigned 9 DelayCounter;

// read in the first pixel (blocks until data is available)
*InPixelChan ? Pixel;

par
{
    while(1)
    {
        // if data has been sent this clock cycle
        if (!NoDataSent)
        {
            par
            {
                // signal for data read from the buffers
                signal unsigned sReadBuffer[3];

                // read in a new pixel from the input channel
                *InPixelChan ? Pixel;

                // read the input data
                sReadBuffer[BOTTOM] = Pixel;

                // read the data from the head of the buffer into the signal
                par (y=0; y<2; y++)
                {
                    sReadBuffer[y] = ReadFIFO(Buffers[y<-1]);
                    IncrementFIFOPointer(Buffers[y<-1]);
                }
            }
        } /*

        DelayCounter--;
        if (DelayCounter==0)
        {
            NoDataSent = 1;
        }
    }
}
The four elements of the window are moved to the FIFO buffers above from left bottom

*/
/*
move the three elements of the kernel lines into the buffer for the line above
*/
par(y=1; y<3; y++)
{
    WriteFIFO(Buffers[(y-1)<-1],ReadDataWindow(Window, y, LEFT));
}

// construct the kernel from the registers and the buffers
par(y=0; y<3; y++)
{
    WriteDataWindow(Window, y, LEFT,ReadDataWindow(Window, y, CENTRE));
    WriteDataWindow(Window, y, CENTRE,ReadDataWindow(Window, y, RIGHT));
    WriteDataWindow(Window, y, RIGHT,sReadBuffer[y]);
}
}

// otherwise pause the pipeline for one clock cycle
else
    delay;
}

// delay the output until the pipeline is primed
do
{
    DelayCounter++;
}
while(DelayCounter!=PipeLatency-1);

/*
Send the data to the next process in the pipeline.
*/
while(1)
{
    prialt
    {
        case *OutWindowChan ! Window: break;
        default: NoDataSent = 1; break;
    }
Function: ProcessWindowMedian

Performs a Median Filter algorithm on the 3x3 window inputs. Since median filter is pipelined, the first output appears after a latency of 14 clock cycle.

```c
void ProcessWindowMedian(WINDOW_PIXEL_8 *W, unsigned 8 *Pixel)
{
    // registers to store all the outputs
    unsigned 8 out1,out2,out3,out4,out5,out6,out7,out8,out9;
    // signals for intermediate values
                          ,C33_L,C33_H,C34_L,C34_H,C41_L,C41_H,C42_L,C42_H,C43_L,C43_H,
                          ,C91_L,C91_H,C101_L,C101_H,C111_L,C111_H;

    signal unsigned int 8 R11,R21,R31,R41,R42,R43,R4A1,R4A2,R4A5,R4A4,R4A3;

    // comparison of the signals for median filtering
    par{// parallel execution
        if((*W).YC[0][0] <(*W).YC[0][1])
        {
            par{C11_L = (*W).YC[0][0];
                 C11_H = (*W).YC[0][1];
        }else
        {
            par{C11_L = (*W).YC[1][0];
                 C11_H = (*W).YC[0][0];
        }
    }
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
if(*(W).YC[0][2] < *(W).YC[1][0])
{ 
    par{
        C12_L = *(W).YC[0][2];
        C12_H = *(W).YC[1][0];
    }
} else 
{par{
        C12_L = *(W).YC[1][0];
        C12_H = *(W).YC[0][2];
    } }

if(*(W).YC[1][1] < *(W).YC[1][2])
{ par{
        C13_L = *(W).YC[1][1];
        C13_H = *(W).YC[1][2];
    } } else{
    par{
        C13_L = *(W).YC[1][2];
        C13_H = *(W).YC[1][1];
    } }

{ 
    par{
        C14_L = *(W).YC[2][0];
        C14_H = *(W).YC[2][1];
    } }
} else 
{ 
    par{
        C14_L = *(W).YC[2][1];
        C14_H = *(W).YC[2][0];
    } }

R11 = *(W).YC[2][2];
} //par

//level 2
par{

    if(C11_L < C12_L)
    {
        par{

        }
C21_L = C11_L;
C21_H = C12_L;
}
}
else
{
par{
C21_L = C12_L;
C21_H = C11_L;
}
}

if (C11_H < C12_H)
{
par{
C22_L = C11_H;
C22_H = C12_H;
}
}
else
{
par{
C22_L = C12_H;
C22_H = C11_H;
}
}

if (C13_L < C14_L)
{
par{
C23_L = C13_L;
C23_H = C14_L;
}
}
else
{
par{
C23_L = C14_L;
C23_H = C13_L;
}
}

if (C13_H < C14_H)
{
par{
C24_L = C13_H;

76

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
\[ C_{24\_H} = C_{14\_H}; \]

\[
\text{else}
\]

\[
\{ \\
\text{par}\{ \\\nC_{24\_L} = C_{14\_H}; \\
C_{24\_H} = C_{13\_H}; \\
\} \\
\}
\]

\[ R_{21} = R_{11}; \]

\[ \} \text{//par} \]

\[ // \text{level 3} \]

\[ \text{par}\{ \]

\[ \text{if} \ (C_{21\_L} < C_{23\_L}) \]

\[
\{ \\
\text{par}\{ \\
C_{31\_L} = C_{21\_L}; \\
C_{31\_H} = C_{23\_L}; \\
\} \\
\}
\]

\[ \text{else} \]

\[
\{ \\
\text{par}\{ \\
C_{31\_L} = C_{23\_L}; \\
C_{31\_H} = C_{21\_L}; \\
\} \\
\}
\]

\[ \text{if} \ (C_{21\_H} < C_{23\_H}) \]

\[
\{ \\
\text{par}\{ \\
C_{32\_L} = C_{21\_H}; \\
C_{32\_H} = C_{23\_H}; \\
\} \\
\}
\]

\[ \text{else} \]

\[
\{ \\
\text{par}\{ \\
C_{32\_L} = C_{23\_H}; \\
C_{32\_H} = C_{21\_H}; \\
\} \\
\}
\]
if (C22_L < C24_L)
{
    par{
        C33_L = C22_L;
        C33_H = C24_L;
    }
} else
{
    par{
        C33_L = C24_L;
        C33_H = C22_L;
    }
}

if (C22_H < C24_H)
{
    par{
        C34_L = C22_H;
        C34_H = C24_H;
    }
} else
{
    par{
        C34_L = C24_H;
        C34_H = C22_H;
    }
}

R31 = R21;
}//par
//-- level 4
par{
    R41 = C31_L;
    if (C31_H < C32_L)
    {
        par{
            C41_L = C31_H;
            C41_H = C32_L;
        }
    } else
    {
        par{
            C41_L = C31_H;
            C41_H = C32_L;
        }
    }
}
\begin{verbatim}
C41_L = C32_L;
C41_H = C31_H;

if (C32_H < C33_L)
{
  par{
    C42_L = C32_H;
    C42_H = C33_L;
  }
} else
{
  par{
    C42_L = C33_L;
    C42_H = C32_H;
  } 
}

if (C33_H < C34_L)
{
  par{
    C43_L = C33_H;
    C43_H = C34_L;
  }
} else
{
  par{
    C43_L = C34_L;
    C43_H = C33_H;
  }
}

R42 = C34_H;
R43 = R31;

//par
//-- level 4a
par{
  R4A1 = R41;
  if (C41_L < C42_H)
  {
    par{
      C4A1_L = C41_L;
      C4A1_H = C42_H;
    }
  }
}
\end{verbatim}
if (C41_H < C42_L)
{
    par{
        C4A2_L = C41_H;
        C4A2_H = C42_L;
    }
    else {
        par{
            C4A2_L = C42_L;
            C4A2_H = C41_H;
        }
        par{
            R4A2 = C43_L;
            R4A3 = C43_H;
            R4A4 = R42;
            R4A5 = R43;
        }
    }
}
//— level 4b
par{
    R4B1 = R4A1;
    if (C4A1_L < C4A2_L)
    {
        par{
            C4B0_L = C4A1_L;
            C4B0_H = C4A2_L;
        }
    }
    else {
        par{
            C4B0_L = C4A2_L;
            C4B0_H = C4A1_L;
        }
    }
if( C4A2_H < R4A2 )
{
par{
C4B1_L = C4A2_H;
C4B1_H = R4A2;
}
}
else
{
par{
C4B1_L = R4A2;
C4B1_H = C4A2_H;
}
}

if( C4A1_H < R4A3 )
{
par{
C4B2_L = C4A1_H;
C4B2_H = R4A3;
}
}
else
{
par{
C4B2_L = R4A3;
C4B2_H = C4A1_H;
}
}
par{
R4B4 = R4A4;
R4B5 = R4A5;
}
//par

//-- level 5
par{
if( R4B1 < R4B5 )
{
par{
C51_L = R4B1;
C51_H = R4B5;
}
}
else
{
par{
   C51_L = R4B5;
   C51_H = R4B1;
}
}
par{
R51 = C4B0_L;
R52 = C4B0_H;
R53 = C4B1_L;
R54 = C4B1_H;
R55 = C4B2_L;
R56 = C4B2_H;
R57 = R4B4;
}
} //par
//-- level 6

par{
if (R51 < C51_H )
{
{
par{
   C61_L = R51;
   C61_H = C51_H;
}
}
else
{
{
par{
   C61_L = C51_H;
   C61_H = R51;
}
}
par{
R61 = C51_L;
R62 = R52;
R63 = R53;
R64 = R54;
R65 = R55;
R66 = R56;
R67 = R57;
}
} //par
//-- level 7
par{
if (R62 < C61_H )
{
    par{
        C71_L = R62;
        C71_H = C61_H;
    }
}
else
{
    par{
        C71_L = C61_H;
        C71_H = R62;
    }
}
par{
    R71 = R61; // -- L
    R72 = C61_L; // -- 2L
    R73 = R63;
    R74 = R64;
    R75 = R65;
    R76 = R66;
    R77 = R67;
}
}//par
//-- level 8

par{
    if (R73 < C71_H )
    {
        par{
            C81_L = R73;
            C81_H = C71_H;
        }
    }
    else
    {
        par{
            C81_L = C71_H;
            C81_H = R73;
        }
    }
} par{
    R81 = R71;
    R82 = R72;
    R83 = C71_L;
    R84 = R74;
R85 = R75;
R86 = R76;
R87 = R77;
}
} //par
//-- level 9
par{
if(R84 < C81_H )
{
par{
C91_L = R84;
C91_H = C81_H;
}
}
else
{
par{
C91_L = C81_H;
C91_H = R84;
}
}
par{
R91 = R81;
R92 = R82;
R93 = R83;
R94 = C81_L;
R95 = R85;
R96 = R86;
R97 = R87;
}
} //par
//-- level 10
par{
if( R95 < C91_H )
{
par{
C101_L = R95;
C101_H = C91_H;
}
}
else
{
par{
C101_L = C91_H;
C101_H = R95;
}
R101 = R91;
R102 = R92;
R103 = R93;
R104 = R94;
R105 = C91_L;
R106 = R96;
R107 = R97;
}
//par
//-- level 11
par{
if(R106 < C101_H) {
par{
C111_L = R106;
C111_H = C101_H;
}
}
else {
par{
C111_L = C101_H;
C111_H = R106;
}
}
par{
R111 = R101;
R112 = R102;
R113 = R103;
R114 = R104;
R115 = R105;
R116 = C101_L;
R117 = R107;
}
//par
//-- level 12
par{
if(R117 < C111_H) {
par{
out8 = R117;
out9 = C111_H;
}
}
else
{
par{
    out8 = C111_H;
    out9 = R117;
}
}
par{
    out1 = R111;
    out2 = R112;
    out3 = R113;
    out4 = R114;
    out5 = R115;
    out6 = R116;
    out7 = C111_L;

    *Pixel = R111;
}
}//end par*
}
 }//end of Median filtering

/******************************************************************************
 * Function: processWindowConvolveKCM_3x3
 ******************************************************************************/

Convolution is done by 8 bit Look Up Tabled based multiplication. The multiplication
tables of above convolution mask are stored in ROMs. To access the same location in
ROM in parallel. For each mask value a multiplication table is stored in two different
ROMs. A 3x3 window pixels are accessed same time. First output is produced after a
latency of 7 cycles.

******************************************************************************

//ROM Declaration
rom unsigned int 16
MUL0[16]={0,21,42,63,84,105,126,147,168,189,210,231,252,273,294,315 };
rom unsigned int 16
MUL1[16]={0,31,62,93,124,155,186,217,248,279,310,341,372,403,434,465};
//Function process Window
void ProcessWindow(WINDOW_FDOLE_8 *W, unsigned 8 *Pixel)
{
    //registes to store the values
    unsigned 16 r01,r11,r21,r31,r41,r51,r61,r71,r81;
    unsigned 16 r02,r12,r22,r32,r42,r52,r62,r72,r82;
    unsigned 16 c01,c11,c21,c31,c41,c51,c61,c71,c81;
    unsigned 16 c02,c12,c22,c32,c42,c52,c62,c72,c82;
    unsigned 16 t0,t1,t2,t3,t4,t5,t6,t7,t8;
    unsigned 16 s1,s2,s3;
    unsigned 16 m1,t;

// 1st cycle

par{

// high order four bits of input to access the ROM
r01 = MUL0(*W).YC[0][0]/4;
rl = MUL1(*W).YC[0][1]/4;
r21 = MUL2(*W).YC[0][2]/4;
r31 = MUL3(*W).YC[1][0]/4;
r41 = MUL4(*W).YC[1][1]/4;
r51 = MUL5(*W).YC[1][2]/4;
r61 = MUL6(*W).YC[2][0]/4;
r71 = MUL7(*W).YC[2][1]/4;
r81 = MUL8(*W).YC[2][2]/4;

// low order four bits of input to access the ROM
r02 = MUL10(*W).YC[0][0]<-4;
rl2 = MUL11(*W).YC[0][1]<-4;
r22 = MUL12(*W).YC[0][2]<-4;
r32 = MUL13(*W).YC[1][0]<-4;
r42 = MUL14(*W).YC[1][1]<-4;
r52 = MUL15(*W).YC[1][2]<-4;
r62 = MUL16(*W).YC[2][0]<-4;
r72 = MUL17(*W).YC[2][1]<-4;
r82 = MUL18(*W).YC[2][2]<-4;

// 2nd cycle

// get first 12 bits
co1 = r01[11:0]@(unsigned 4)0;
c11 = rl1[11:0]@(unsigned 4)0;
c21 = r21[11:0]@(unsigned 4)0;
c31 = r31[11:0]@(unsigned 4)0;
c41 = r41[11:0]@(unsigned 4)0;
c51 = r51[11:0]@(unsigned 4)0;
c61 = r61[11:0]@(unsigned 4)0;
c71 = r71[11:0]@(unsigned 4)0;
c81 = r81[11:0]@(unsigned 4)0;

// get first 12 bits
co2 = 0@r02[11:0];
c12 = 0@r12[11:0];
c22 = 0@r22[11:0];
c32 = 0@r32[11:0];
c42 = 0@r42[11:0];
c52 = 0@r52[11:0];
c62 = 0@r62[11:0];
c72 = 0@r72[11:0];
c82 = 0@r82[11:0];
//3\textsuperscript{rd} cycle
//combine the 12 bits to produce 16 bit number
\begin{align*}
t_0 &= c_{01} + c_{02}; \\
t_1 &= c_{11} + c_{12}; \\
t_2 &= c_{21} + c_{22}; \\
t_3 &= c_{31} + c_{32}; \\
t_4 &= c_{41} + c_{42}; \\
t_5 &= c_{51} + c_{52}; \\
t_6 &= c_{61} + c_{62}; \\
t_7 &= c_{71} + c_{72}; \\
t_8 &= c_{81} + c_{82};
\end{align*}

//4\textsuperscript{th} cycle
\begin{align*}
s_1 &= t_0 + t_1 + t_2; \\
s_2 &= t_3 + t_4 + t_5; \\
s_3 &= t_6 + t_7 + t_8; \\
//5\textsuperscript{th} cycle
m_1 &= s_1 + s_2 + s_3;
//6\textsuperscript{th} cycle
\end{align*}
\begin{align*}
t &= (m_1)\ll 8; \\
\ast \text{Pixel} &= t\ll 8;
\end{align*}

//run the sobel edge detection on each of the three components
//Convolution((\ast \text{Window}).YC, \ast \text{Pixel});
This function processes the 5x5 gaussian convolution. Since the design is pipelined 5x5 window pixel are accessed at the same time. A single output is produced on every clock cycle. The first output is produced after a latency of 7 clock cycles.

```c
void ProcessWindowConvolution5x5(WINDOW_PIXEL_8 *W, unsigned 8 *Pixel)
{
    // 16 bit registers
    unsigned 16 t1,t2,t,m0,m1,m2,m3,m4,n0,n1,p0,p1,p2,p3,q2,q3;
    unsigned 16 m,n,p,q,temp;
    macro expr ext(n) = ((unsigned 8)0) @ n;
    unsigned 16 m11,m12,p11,p12,q11;

    par{
        // 1st clock cycle
        m0 = ext((*W).YC[0][1]) + ext((*W).YC[2][0]);
        m1 = ext((*W).YC[2][4])+ ext((*W).YC[4][2]);
        m2 = ext((*W).YC[1][1]) + ext((*W).YC[1][3]);
        m3 = ext((*W).YC[3][1]) + ext((*W).YC[3][3]);
        m4 = ext((*W).YC[2][2]);
        n0 = ext((*W).YC[0][0]) + ext((*W).YC[0][4]);
        p0 = ext((*W).YC[0][1]) + ext((*W).YC[0][3]);
        p1 = ext((*W).YC[1][0]) + ext((*W).YC[1][4]);
        p2 = ext((*W).YC[3][0]) + ext((*W).YC[3][4]);
        p3 = ext((*W).YC[4][1]) + ext((*W).YC[4][3]);
        q2 = ext((*W).YC[2][2]) + ext((*W).YC[2][1]);

        // 2nd clock cycle
        m11 = m0 + m1;
        m12 = m2 + m3;
        p11 = p0+p1;
        p12 = p2+p3;
        q11 = q2+q3;
    }
}
```
//3rd clock cycle
m = (m11 + m12 + m4);
n = (n0 + n1) << 1;
p = (p11 + m11 + p12 + q11) << 2;
q = (m12 + q11) << 3;

//4th clock cycle
t1 = (m+n);
t2 = (p+q);
t = (t1 + t2) / 115;

//5th clock cycle
temp = (t > 255) ? 255 : t;

//6th clock cycle
*Pixel = (unsigned)((temp) < -8);

Function: ProcessWindowGradient

Calculates the gradient using the following convolutions

\[
\begin{bmatrix}
-1 \\
0 \\
1
\end{bmatrix}
\text{ for horizontal gradient}
\begin{bmatrix}
0 \\
1 \\
\end{bmatrix}
\text{ for vertical gradient}
\]

3x3 pixels are given as input to this function and a 9 bit horizontal and 9 bit vertical gradient are combined to produce a 18 bit pixel on every clock cycle.

void ProcessWindowGradient(WINDOW_SM *W, int 18 *Pixel_dx_dy,unsigned 8 *count_row,unsigned 8 *count_col)
{
unsigned 8 YC;
macro expr ext(n) = ((unsigned 1)0) @ n;
int dy, dx;

if (*count_row < 4 || *count_row == 255) || (*count_col < 2 || *count_col == 255))
{
par{
dy = 0;
dx = 0;

```
Function: ProcessWindowNonMax

Calculate the directional non maximum gradient based on the sign and magnitude of the gradient. 18 bit 3x3 window pixels are given as input and produces a 8 bit output after an initial latency of 4 cycles.

#define HIGH_THRESHOLD 120
#define LOW_THRESHOLD 40

void ProcessWindowNonMax(WINDOW_DX_DY *Wdxdy, unsigned 8 *Pixel)
{
    int dx,dy,mag;

    //absoloute value
    macro expr abs(a) = (a<0 ? -a : a);

    //absoulute
    macro expr abs1(b) = (b[width(b)-1]? -b : b);
    //macro expr ext(n) = ((unsigned 8)0) @ n;
    int P1,P2,P3,P4,Pa,Pb;
    //1st cycle
    par
    //drop first nine bits
    dx = (int)(*Wdxdy).YC[1][1]\\9
    //take first nine bits

    *Pixel_dx_dy = 0@dx@dy;
    } //par

}
dy = (int)(*Wdxdy).YC[1][1]<-9

if(abs(dx) > abs(dy))
{
// between 0 & 45 degrees
if(dy == 0)
{
par{
P1 = (int)(abs1(*Wdxdy).YC[1][0]<-9) + abs1(*Wdxdy).YC[1][0]<-9);
P2 = (int)(abs1(*Wdxdy).YC[1][0]<-9) + abs1(*Wdxdy).YC[1][0]<-9);
}
}
else
if((dx > 0 && dy > 0) || (dx < 0 && dy < 0))
{
par{
P1 = (int)(abs1(*Wdxdy).YC[0][2]<-9) + abs1(*Wdxdy).YC[0][2]<-9);
P3 = (int)(abs1(*Wdxdy).YC[2][0]<-9) + abs1(*Wdxdy).YC[2][0]<-9);
P4 = (int)(abs1(*Wdxdy).YC[1][0]<-9) + abs1(*Wdxdy).YC[1][0]<-9);
}
} else
if(abs(dy) > abs(dx))
{
// vertical direction
if(dx == 0)
{
par{
P1 = (int)(abs1(*Wdxdy).YC[0][1]<-9) + abs1(*Wdxdy).YC[0][1]<-9);
P2 = (int)(abs1(*Wdxdy).YC[0][1]<-9) + abs1(*Wdxdy).YC[0][1]<-9);
}
P3 = (int)(abs1(*Wdxdy).YC[0][1]<<9) + abs1(*Wdxdy).YC[0][1]<-9;
}

} else // between 45 & 90 or 225 & 270
if( (dx > 0 && dy > 0) || (dx < 0 && dy < 0))
{
    par{
    P1 = (int)(abs1(*Wdxdy).YC[0][1]<<9) + abs1(*Wdxdy).YC[0][1]<-9;
P2 = (int)(abs1(*Wdxdy).YC[0][2]<<9) + abs1(*Wdxdy).YC[0][2]<-9;
    }
}

} else // between 90 & 135 & 270 & 315
if( (dx < 0 && dy > 0) || (dx > 0 && dy < 0))
{
    par{
    P1 = (int)(abs1(*Wdxdy).YC[0][0]<<9) + abs1(*Wdxdy).YC[0][0]<-9;
P2 = (int)(abs1(*Wdxdy).YC[0][1]<<9) + abs1(*Wdxdy).YC[0][1]<-9;
    }
}

} else // 45 & 135
if( abs(dx) == abs(dy) )
{

if( (dx > 0 && dy > 0) || (dx < 0 && dy < 0))
{
    par{
    P1 = (int)(abs1(*Wdxdy).YC[0][2]<<9) + abs1(*Wdxdy).YC[0][2]<-9;
P2 = (int)(abs1(*Wdxdy).YC[0][2]<<9) + abs1(*Wdxdy).YC[0][2]<-9;
    }
} else
if((dx < 0 && dy > 0) || (dx > 0 && dy < 0))
{
    par{
    P1 = (int)(abs1(*Wdxdy).YC[0][0]<<9) + abs1(*Wdxdy).YC[0][0]<-9;
P2 = (int)(abs1(*Wdxdy).YC[0][0]<<9) + abs1(*Wdxdy).YC[0][0]<-9;
    }
}
```c
} else
if(dy == 0)
{
par{
  P1 = (int)(abs1((*Wdxdy).YC[1][0]\9) + abs1(*Wdxdy).YC[1][0]<-9);
  P2 = (int)(abs1(*Wdxdy).YC[1][0]\9) + abs1(*Wdxdy).YC[1][0]<-9);
  P3 = (int)(abs1(*Wdxdy).YC[1][2]\9) + abs1(*Wdxdy).YC[1][2]<-9);
  P4 = (int)(abs1(*Wdxdy).YC[1][2]\9) + abs1(*Wdxdy).YC[1][2]<-9);
}
//2\textsuperscript{nd} cycle
Pa = (P1 + P2)\tt>>1;
Pb = (P3 + P4)\tt>>1;

//3\textsuperscript{rd} cycle
if( mag > Pa && mag > Pb )
{
  if(mag > HIGH_THRESHOLD)
    *Pixel = EDGE; // definite edge
  else
    if( mag < LOW_THRESHOLD)
      *Pixel = NOEDGE; // no edge
    else
      *Pixel = MAYBE_EDGE; // may be an edge
  else
    *Pixel = 0
}
//end par

}//end macro Nonmax
```
/*******************************************************/
/* Function : ProcessWindowHysteresis */

Description
Reads input 3 by 3 windows from *WindowIn, performs the hysteresis by two
threshold. Th and Tl (Th > Tl). if center pixel is maybe edge and if any of the
neighboring pixels is a definite edge, then that pixel is considered an edge. The first
output down the channel *OutChan will not occur until the pipeline has been primed with
valid data.

******************************************************************************/
#define EDGE 255
#define MAYBE_EDGE 200
#define EDGE 0

void ProcessWindowHysteresis(WINDOW_PIXEL_8 *W, unsigned 8 *Pixel)
{
    // check if the center pixel MAYBE_EDGE (i.e., pixel value between high threshold
    and low threshold).
    if((*W).YC[1][1] == MAYBE_EDGE)
    {
        if((*W).Y[0][0] == EDGE || (*W).Y[0][1] == EDGE || (*W).Y[0][2] == EDGE ||
        {
            *Pixel = EDGE; // center pixel is an edge.
        }
        else delay
    }
    else delay;
}
VITA

Graduate College
University of Nevada, Las Vegas

Venkateshwar Rao Daggu

Local Address:
1165, Maryland Circle #1
Las Vegas, NV 89119.

Degree:
Bachelor of Technology
Kakatiya University, India

Thesis Title:
Efficient Design and Implementation of Image Processing Algorithms on
Reconfigurable Hardware using Handel-C

Thesis Examination Committee:
Chairperson, Dr. Venkatesan Muthukumar, Ph. D.
Committee Member, Dr. Henry Selvaraj, Ph. D.
Committee Member, Dr. Emma Regentova, Ph. D.
Graduate faculty Representative, Dr. Evangelos Yfantis, Ph. D.