Master of Science Thesis in Electrical Engineering
Department of Electrical Engineering, Linköping University, 2016

A Parallel FPGA Implementation of Image Convolution

Henrik Ström

Master of Science Thesis in Electrical Engineering

A Parallel FPGA Implementation of Image Convolution

Henrik Ström

LiTH-ISY-EX--16/4931--SE

Supervisor: **Andreas Ehliar**
ISY, Linköpings universitet
Åsa Detterfelt
MindRoad AB

Examiner: **Oscar Gustafsson**
ISY, Linköpings universitet

Division of Computer Engineering
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

To SM5EEP...

Abstract

Image convolution is a common algorithm that can be found in most graphics editors. It is used to filter images by multiplying and adding pixel values with coefficients in a filter kernel. Previous research work have implemented this algorithm on different platforms, such as FPGAs, CUDA, C etc. The performance of these implementations have then been compared against each other. When the algorithm has been implemented on an FPGA it has almost always been with a single convolution. The goal of this thesis was to investigate and in the end present one possible way to implement the algorithm with 16 parallel convolutions on a Xilinx Spartan 6 LX9 FPGA and then compare the performance with results from previous work. The final system performs better than multi-threaded implementations on both a GPU and CPU.

Acknowledgments

First, I would like to thank MindRoad for giving me the opportunity to work with this thesis. A special thanks goes to my supervisor Åsa Detterfelt at MindRoad for guiding me through this thesis project.

I would also like to thank my examiner Oscar Gustafsson at Linköping University for his valuable input and feedback.

Finally, thanks to my family and friends for their support throughout these years.

Linköping, April 2016
Henrik Ström

Contents

List of Figures

List of Tables

Notation

Acronyms

Acronyms	Meaning
2D	Two-dimensional
BRAM	Block RAM
CLB	Configurable Logic Block
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DDR	Double Data Rate
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
LPDDR	Low Power DDR
LSB	Least significant bit(s)
MSB	Most significant bit(s)
RAM	Random Access Memory
SDRAM	Synchronous Dynamic RAM
TCM	Two Convolution Module (TC Module)

1 Introduction

This thesis is the final examining part of a master's degree in Computer Science and Engineering at Linköping University. It was conducted at Mindroad in Linköping.

1.1 Background

This thesis work is based on a previous bachelor's thesis [8], where the authors sought to find two different algorithms that are commonly implemented in a single-thread sequential manner. The authors then created their own multi-threaded versions of these algorithms and compared the performance with the single-thread counterpart. The chosen algorithms were *merge-sort* and *image convolution*. Merge-sort did not turn out to be a suitable algorithm in the end, even if it can be highly parallelized in the beginning, the later part of the sorting is more or less sequential when the complete list has to be sorted. The image convolution algorithm turned out to be a suitable algorithm to parallelize.

Image convolution is a common algorithm used to filter images. It can be found in most graphics editors, such as Adobe Photoshop and GNU Image Manipulation Program. Typical filters are blur, sharpen and edge detection.

Previous studies of image convolution concludes that the algorithm is suitable to parallelize. The authors of [5] suggests that the performance of the algorithm can be improved by dividing the input image into smaller images and run multiple convolutions in parallel. By doing this the performance will probably be improved by a factor proportional to the number of convolutions that run in parallel. Another similar method to improve the performance of the algorithm is presented by the authors in [9].

1.2 Mindroad AB

Mindroad is an engineering company with focus on software, both embedded software and application development. The headquarter is located in Linköping, Sweden. Mindroad has customers spread out across Sweden. The company has four business areas:

- *Academy* - Offers business training in software development with focus on new technologies and new tools.
- *Application* - Focuses on development of application software and complex software systems.
- *Embedded* - Develops software for embedded systems.
- *Sourcing* - Focuses on international business.

1.3 Problem Statements

This thesis deals with the following problem statements:

- *What is the performance of the parallel image convolution compared to a CPU and a GPU implementation?*

The test results from [8] are compared with the performance results of the FPGA implementations. It was also of interest to compare the FPGA implementation results with the results from [5]. For smaller images the FPGA was better, in terms of number clock cycles, than an implementation in CUDA. However, their FPGA implementation did not utilize any kind of parallelism.

- *Does the performance scale proportionally to the number of convolutions running in parallel?*

Reference [5] concludes that the performance can be increased by using multiple convolution modules in parallel, if the hardware allows it. However, this solution would consume more die area of a chip. This is a trade-off that the manufacturer has to take into consideration.

- *What are the limiting factors when implementing the parallel image convolution on an FPGA?*

As mentioned earlier, a parallel implementation of image convolution on an FPGA is very dependent on the hardware. By introducing more convolution modules one must take into account that the memory bandwidth and speed might become a bottle neck in the final design.

1.4 Scope and Delimitations

The thesis work was limited to 20 weeks. In order to limit the scope of this project a few delimitations were needed.

Mindroad requested that an *Avnet Xilinx Spartan-6 FPGA LX9 Microboard* should be used to start with and scale up if needed.

- The focus of this thesis work was to increase the number of convolutions running in parallel.
- The filter kernel was limited to 3×3.
- The image resolution was limited to $480p$, $720p$, $1080p$ and $1440p$.
- The input and output images were 24 bit RGB .ppm files.

1.5 Thesis Outline

The outline of this thesis is as follows:

- Chapter 2 presents the theory behind the image convolution algorithm and previous research work that is related to this thesis.
- Chapter 3 presents the different systems and how they were implemented.
- Chapter 4 presents the results from the tests that were conducted.
- Chapter 5 discusses the implementations, alternative solutions and the test results.
- Chapter 6 presents the conclusions and what can be done in the future.

2

Theory and Related Work

This chapter discusses the image convolution algorithm and previous articles related to the topic of this thesis.

2.1 Image Convolution

The image convolution algorithm uses discrete convolution for two dimensions, the definition is shown in equation 2.1 below.

$$O(x,y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(x-i, y-j) \cdot H(i,j) \tag{2.1}$$

Where $O(x,y)$ is the pixel at position (x,y) in the output image, $I(x,y)$ is the corresponding pixel in the input image and $H(i,j)$ is the filter kernel.

Both the input and output images consist of W(width) $\times$ H(height) pixels. The pixel that is currently being calculated, $O(x,y)$, is the sum of all the pixels, $I(x,y)$ in the input image multiplied by the coefficients in the filter kernel, H.

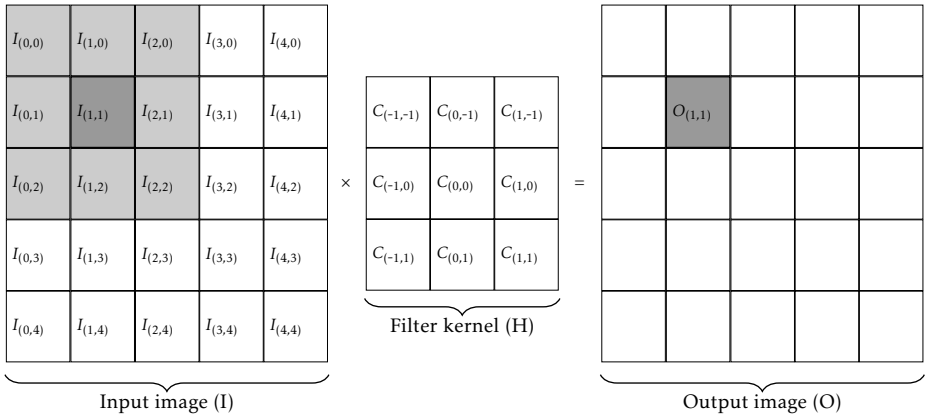


Figure 2.1: *Image convolution with an input image of size* 5×5 *and a filter kernel of size* 3×3*.*

Figure 2.1 above illustrates a simple example of image convolution. An input image, I, with 5×5 pixels and a 3×3 filter kernel, H. The resulting output, O, will be a 5×5 image, where every pixel has been processed.
Equation 2.2 below shows the calculations needed to calculate the value of $O_{(1,1)}$ in the example above.

$$\begin{aligned} O_{(1,1)} = (I_{(0,0)} \cdot C_{(-1,-1)}) + (I_{(1,0)} \cdot C_{(0,-1)}) \; + \; (I_{(2,0)} \cdot C_{(1,-1)}) + \\ (I_{(0,1)} \cdot C_{(-1,0)}) + (I_{(1,1)} \cdot C_{(0,0)}) \; + \; (I_{(2,1)} \cdot C_{(1,0)}) + \\ (I_{(0,2)} \cdot C_{(-1,1)}) + (I_{(1,2)} \cdot C_{(0,1)}) \; + \; (I_{(2,2)} \cdot C_{(1,1)}) \end{aligned} \tag{2.2}$$

Depending on the coefficients in the filter kernel the output image will differ. A more detailed explanation of the filter kernel and actual illustrations can be found in section 2.1.1 below.

2.1.1 Filter Kernel

In order to explain the effect of different coefficients in the filter kernels, figures 2.2, 2.3 and 2.4 illustrate three different filter kernels that are commonly used.

Sharpen Filter

This filter, illustrated in figure 2.2, makes the image slightly sharper compared to the input image. If we look at the coefficients in the filter kernel, the center pixel is amplified by a factor of 5 and then the neighboring pixels in the x and y directions are subtracted. This means that edges are more prominent and thus making the image look sharper.

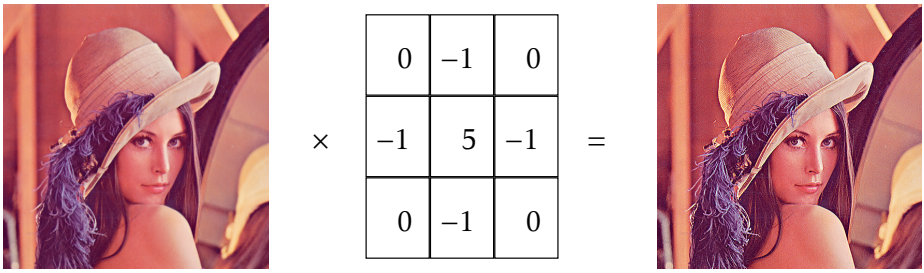


Figure 2.2: *Illustration of sharpen filter.*

Gaussian Blur Filter

This filter, illustrated in figure 2.3, adds a blurring effect to the image. Looking at the coefficients we notice that the output pixel is the sum of all neighboring pixels weighted differently. This makes the edges smoother and the image looks blurred. Another thing that one can notice is that the coefficients are all divided by 16, this is the normalizing constant which is explained more in detail below.

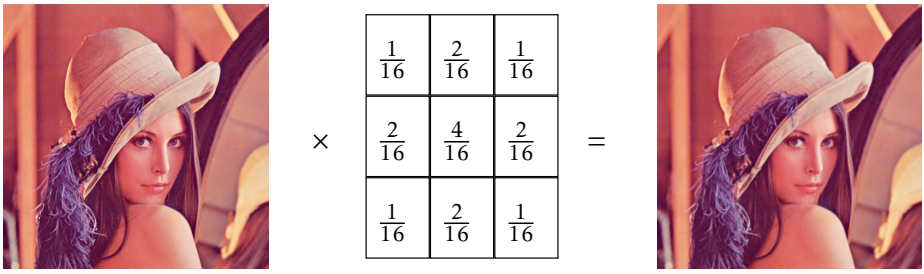
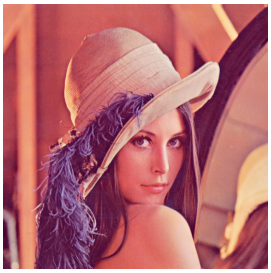


Figure 2.3: *Illustration of Gaussian blur filter.*

Edge Detection Filter

The edge detection filter, illustrated in figure 2.4, is a filter kernel that is commonly used to find edges in an image. Similar to the sharpen filter the center pixel is amplified a lot. In this case the edges are exposed, which means mostly the contours around objects are shown in the output image.



$\times$

−1	−1	−1
−1	8	−1
−1	−1	−1

$=$

Figure 2.4: *Illustration of edge detection filter.*

Normalizing Constant

This constant is used to normalize the filter kernel. All coefficients in the filter kernel are divided by the sum of all coefficients. This reduces the risk of the output pixel from ending up outside a set interval, e.g. in this project $0 \leq O_{(x,y)} \leq 255$. If the sum of the ouput pixel is still outside this interval there are ways to solve this problem, which is explained in section 2.1.2 below.

2.1.2 Saturation

Saturation is used to limit the value within a fixed interval, in this project the interval is 0 to 255. Psudocode for the saturation module can be found in algorithm 1, inspired by [3].

Algorithm 1 Pseudocode for saturation.

```
if value > max_val then
    O_{x,y} ← max_val
else if value < min_val then
    O_{x,y} ← min_val
else
    O_{x,y} ← value
end if
```

Where $value$ is the value after the multiplications and additions are done and $O_{x,y}$ is the output pixel.

2.1.3 Handling the Border Pixels

A problem will occur when the filter kernel is processing pixels on the border around the image. All pixels within the radius of the filter kernel are needed in the multiplication and addition. In figure 2.1 the border pixels are found in the top and bottom rows and in the columns on each sides of the image. There are several common methods to deal with this problem, more details of these methods can be found in [2]. All methods have their advantages and disadvantages.

Cropping

The border pixels are excluded from the image convolution, the output image is smaller than the input image. Figure 2.5 below illustrates an example of cropping.

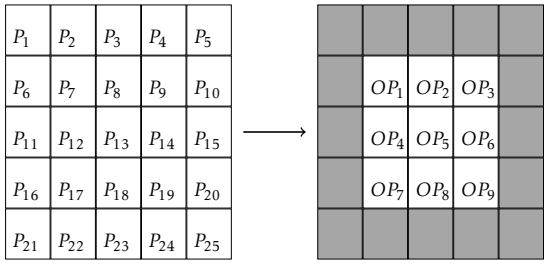


Figure 2.5: *Illustration of cropping.*

The convolution is not performed on the border pixels because some of the values does not exist. The convolution will only be performed when the kernel has values for all the coefficients.

Extended Border

The border pixels are extended to create a second border layer. Figure 2.6 illustrates how the input image seen by the convolution module.

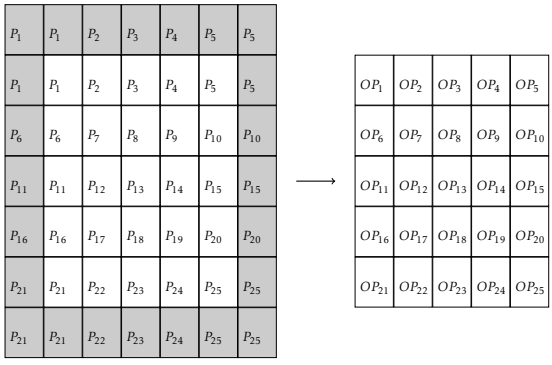


Figure 2.6: *Illustration of an extended border.*

As shown in figure 2.6, the corner pixels in the image are copied to three pixels in the additional border layer. This solution requires more logic to create the second border layer.

Zero-padding

Zero-padding is a simpler version of extended border, all pixels outside the image are treated as zeroes. Figure 2.7 illustrates how zero-padding works.

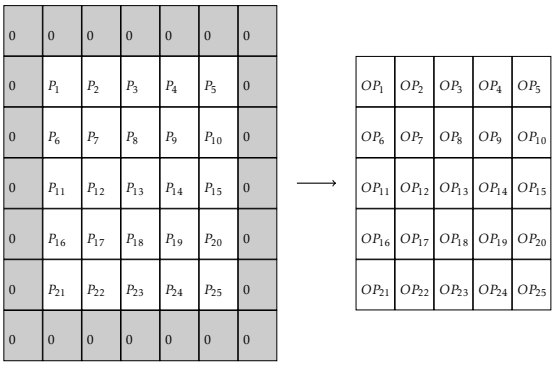


Figure 2.7: *Illustration of a zero-padded border.*

This is the simplest method to implement, if the kernel can not retrieve a pixel value it will be set to zero.

Wrap-around

When a pixel located on the border is being processed, the pixel values from outside the border are taken from the next or previous row. A simple example is illustrated in figure 2.8.

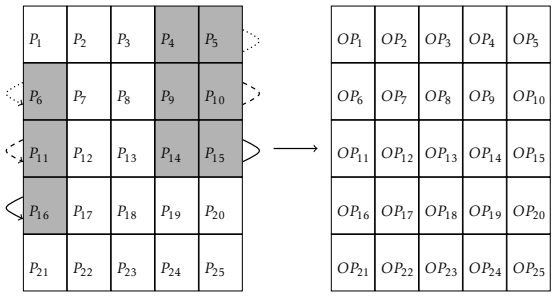


Figure 2.8: *Illustration of a wrap-around border.*

When OP_{10} is processed, the following calculations are done:

$$\begin{aligned} OP_{10} = (P_4 \cdot C_{(-1,-1)}) + (P_5 \cdot C_{(0,-1)}) \ + \ (P_6 \cdot C_{(1,-1)}) + \\ (P_9 \cdot C_{(-1,0)}) + (P_{10} \cdot C_{(0,0)}) \ + \ (P_{11} \cdot C_{(1,0)}) + \\ (P_{14} \cdot C_{(-1,1)}) + (P_{15} \cdot C_{(0,1)}) \ + \ (P_{16} \cdot C_{(1,1)}) \end{aligned}$$

The wrap-around method does have its disadvantages. The top and bottom rows have to use pixels from the opposite side of the image. This results in a more complex addressing method. An easier alternative is to use zero-padding or border extension on the top and bottom row.

2.1.4 Image Partitioning

When parallel convolutions are introduced, one has to consider how the convolutions are going to traverse the image. References [4] and [12] discusses how to partition the image in different ways and distribute the sub-images between a number of workstations. Each workstation will then perform the image convolution algorithm on its own sub-image.

Reference [4] proposes three methods: *row partitioning*, *cross partitioning* and *heuristic partitioning*. The advantage of row partition is that the workload is spread evenly between the workstations, the disadvantage is that the total number of border pixels of the sub-images is relatively high.

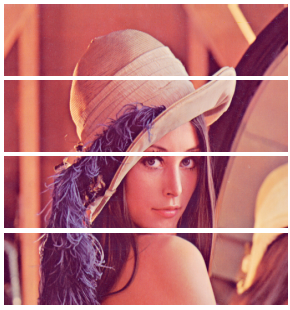


Figure 2.9: *Example of row partition with four sub-images.*

The cross partition method splits the image evenly in both the vertical and horizontal direction, creating n^2 sub-images. Just like the horizontal partition this method distributes the workload evenly between all workstations, but the total number of border pixels is lower than horizontal partition. The disadvantage is that it requires n^2 number of workstations, compared to n in row partitioning.

The last method, heuristic partition, proposed by the authors of [4]. This method is based on the cross partition but it can divide the image into any number of sub-images. See [4] for a more detailed explanation.

2.2 Related Work

As mentioned earlier in this chapter, image convolution is a very common algorithm used in digital image processing. Implementations of the algorithm on FPGAs have been discussed in multiple articles and reports, e.g. this topic was

Figure 2.10: *Example of cross partition with four sub-images, $n = 2$.*

discussed in [1] and [7] back in the 1990s. This section will discuss and compare the different approaches the authors of these articles have made.

2.2.1 2D Convolution vs. Separable Convolution

The convolution algorithm discussed in section 2.1 was a two-dimensional (2D) convolution. The 2D convolution performs all multiplications simultaneously and then all results are added together. Separable convolution performs two one-dimensional convolutions. The first convolution is done either horizontally and then vertically or vice versa. It does however require that the kernel is separable.

$$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix} = \frac{1}{4}\begin{bmatrix}1\\ 2\\ 1\end{bmatrix} \times \frac{1}{4}\begin{bmatrix}1 & 2 & 1\end{bmatrix}$$

Figure 2.11: *Illustration of 2D convolution kernel and separable convolution kernel.*

As illustrated in figure 2.11, the kernel to the left is the 2D convolution kernel (matrix). This kernel can be separated into two vectors and the normalizing constant is also split between the vectors.
Both the separable and 2D convolution can process one pixel per clock cycle but the hardware complexity differs.

	2D convolution	Separable convolution
Multipliers	m^2	$2m$
Adders	$m^2 - 1$	$2m - 2$

Table 2.1: *Hardware requirement for an $m \times m$ filter kernel.*

Reference [5] discusses the advantages and disadvantages of using separable or 2D convolution. Even though the separable convolution reduces the total number of operations it still requires the data to be processed twice.

2.2.2 Pixel Buffer

Reference [1] proposes a delay line shift register to limit the filter kernel from reading the same pixel value multiple times from an external memory, this solution reads a pixel value from the external memory. Figure 2.11 illustrates a 10×10 image and a 3×3 filter kernel.

P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
P_{11}	P_{12}	P_{13}	P_{14}	P_{15}	P_{16}	P_{17}	P_{18}	P_{19}	P_{20}
P_{21}	P_{22}	P_{23}	P_{24}	P_{25}	P_{26}	P_{27}	P_{28}	P_{29}	P_{30}
P_{31}	P_{32}	P_{33}	P_{34}	P_{35}	P_{36}	P_{37}	P_{38}	P_{39}	P_{40}
P_{41}	P_{42}	P_{43}	P_{44}	P_{45}	P_{46}	P_{47}	P_{48}	P_{49}	P_{50}
P_{51}	P_{52}	P_{53}	P_{54}	P_{55}	P_{56}	P_{57}	P_{58}	P_{59}	P_{60}
P_{61}	P_{62}	P_{63}	P_{64}	P_{65}	P_{66}	P_{67}	P_{68}	P_{69}	P_{70}
P_{71}	P_{72}	P_{73}	P_{74}	P_{75}	P_{76}	P_{77}	P_{78}	P_{79}	P_{80}
P_{81}	P_{82}	P_{83}	P_{84}	P_{85}	P_{86}	P_{87}	P_{88}	P_{89}	P_{90}
P_{91}	P_{92}	P_{93}	P_{94}	P_{95}	P_{96}	P_{97}	P_{98}	P_{99}	P_{100}

Figure 2.12: *Illustration of a pixel buffer (bright gray) and a filter kernel (dark gray).*

The filter kernel consists of the pixels in the dark gray area and the bright gray area is the pixel buffer. A new pixel value is read every clock cycle, it enters the first shift register (bottom row of the filter kernel). After m clock cycles this pixel value is shifted into the pixel buffer. The pixel buffer is $W - m$ cells wide. In figure 2.12 the image width is 10 pixels ($W = 10$) and the width of the filter kernel is 3 pixels ($m = 3$). When the pixel has passed through the pixel buffer it enters the next shift register, this process is repeated until the filter kernel is filled with pixel values.

3
Method

This chapter describes the hardware that was used and the different parts of the practical work involved in this thesis work. The project was divided into two phases: single convolution, section 3.2, and parallel convolutions, section 3.3. Each phase consisted of system design, system implementation and lastly system evaluation. The test results are presented in chapter 4.

3.1 Hardware

This section describes the hardware that was used during the thesis work.

3.1.1 Computer

During this thesis a regular consumer PC was used for development and simulations. The specifications of the computer is listed in table 3.1 below.

Component	Name
Processor	Intel Core i7-4790 @ 3.6GHz
GPU	Intel HD Graphics 4600
Memory	16 GB
Operating system	Windows 7.1 64-bit

Table 3.1: *Computer specifications.*

3.1.2 FPGA

As mentioned in section 1.4, it was requested that an Avnet Xilinx Spartan-6 FPGA LX9 Microboard (XC6SLX9) would be used. This is a low-cost entry level FPGA suited for people who wants to explore the capabilities of FPGAs.

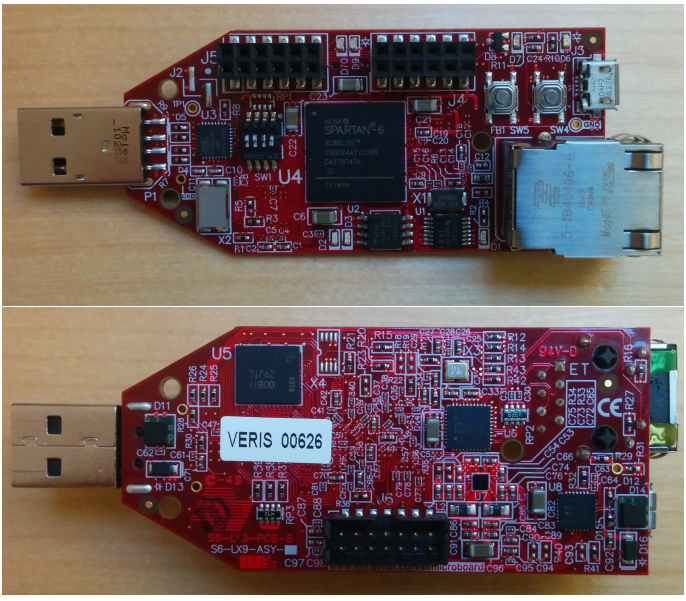


Figure 3.1: *The front and back of an Avnet Xilinx Spartan 6 LX9 Microboard.*

A summarized list of features of the FPGA is listed in table 3.2 below, more details can be found in datasheets [10, 11].

Logic cells		9,152
CLBs	Slices Flip-flops Max Distributed RAM	1,430 11,440 90
DSP48A1 Slices		16
Block RAM blocks	18Kb Total (Kb)	32 576
Max User I/O		200
Micron LPDDR Mobile SDRAM		128 Mb

Table 3.2: *Avnet Xilinx Spartan 6 LX9 Microboard feature summary.*

3.2 Single Convolution

A system that ran only a single convolution module was implemented during the first part of the development phase of the project. The single convolution implementation can be compared with a single thread implementation on a CPU.

3.2.1 Implemented System

The system consists of four modules: Clock divider, Filter Selector, Buffer and Convolution. Figure 3.2 below illustrates the implemented system and the modules are described in detail in their respective section. Table 3.3 below lists all signals and their direction.

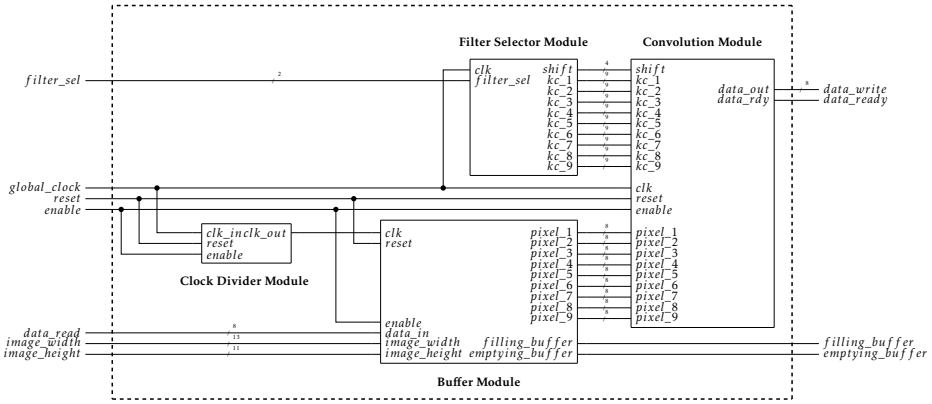


Figure 3.2: *Single convolution system.*

Signal	Description	Direction
global_clock	Global clock running at 66.7 MHz.	In
filter_sel	Selects what filter to be used.	In
reset	Reset signal.	In
enable	Enables the modules.	In
data_read	Pixel value from the PC.	In
image_width	Specifies the width of the input image.	In
image_height	Specifies the height of the input image.	In
data_out	Pixel value to be sent back to the PC.	Out
data_ready	Signals that data is ready.	Out
filling_buffer	Signals that the buffer is filling up.	Out
emptying_buffer	Signals that the buffer is emptying.	Out

Table 3.3: *List of I/O signals.*

3.2.2 Convolution Module

An illustration of the convolution module can be seen in figure 3.3 below.

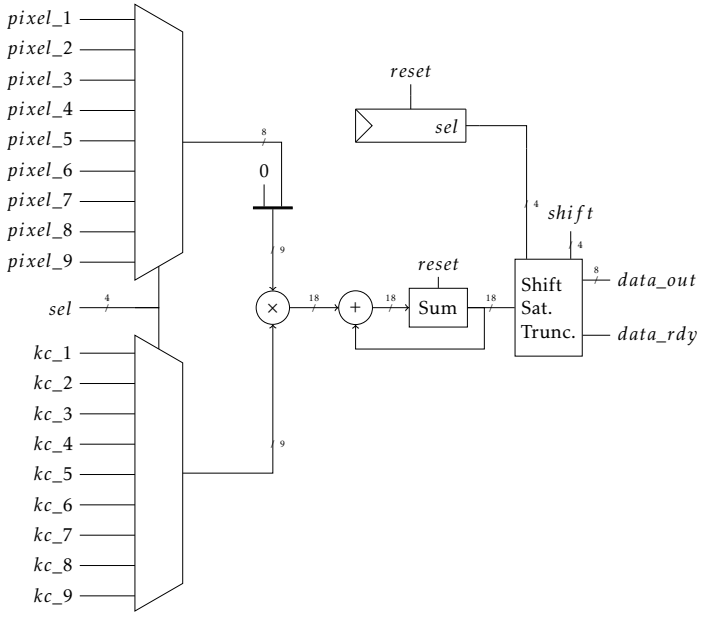


Figure 3.3: *Illustration of the convolution module.*

Due to the limited number of DSP48A1 slices, the convolution module was designed with only one multiplier. This would allow more parallel convolution modules to be implemented in the later part of this thesis work. Instead of multiplying all pixel values with their corresponding kernel coefficient at the same time, which would require nine multipliers, the incoming pixel values and kernel coefficients pass through multiplexers and then they are multiplied. By doing this, the single multiplier is reused nine times. The multiplexers are controlled by a counter with the output signal *sel*. The counter is incremented on every positive edge of the clock, starting on 1 and counts to 9 and then resets back to 1.

Before the multiplication can be done, the pixel values are converted from eight bit unsigned values to nine bit two's complement values. This is done by concatenating a '0' as a sign bit, since all pixel values in the input image can only have a positive value between 0 and 255. The kernel coefficients on the other hand can be both positive and negative, depending on the filter kernel that is being used. The coefficients are represented as nine bit two's complement values.

The result of the multiplications is accumulated in a sum register, Sum in figure

3.3. When the last multiplication is done the sum is shifted, saturated and truncated. The new value is then assigned to *data_out*, and *data_rdy* signals that the data is ready to be read.

3.2.3 Buffer Module

The pixel buffer method discussed in section 2.2.2 was slightly modified. Instead of using the delay line shift register proposed in [1], the delay line was implemented as a circular buffer using Block RAMs. Figure 3.4 below illustrates how the buffer module is constructed. A new pixel value is shifted into the bottom shift register every clock cycle, when a pixel value has been shifted through the whole shift register it is written to the pixel buffer. This process is repeated for the middle and top shift registers. Each shift register feeds three pixel values to the convolution module.

As discussed in section 2.1.3, the system has to handle the border pixels in one way or another. The implemented buffer module combines the border extension and wrap-around methods. If the filter kernel is on the top or bottom row the pixels outside the image are assigned the value of the closest pixel inside the image. When the filter kernel is on the left or right border it uses the wrap-around method.

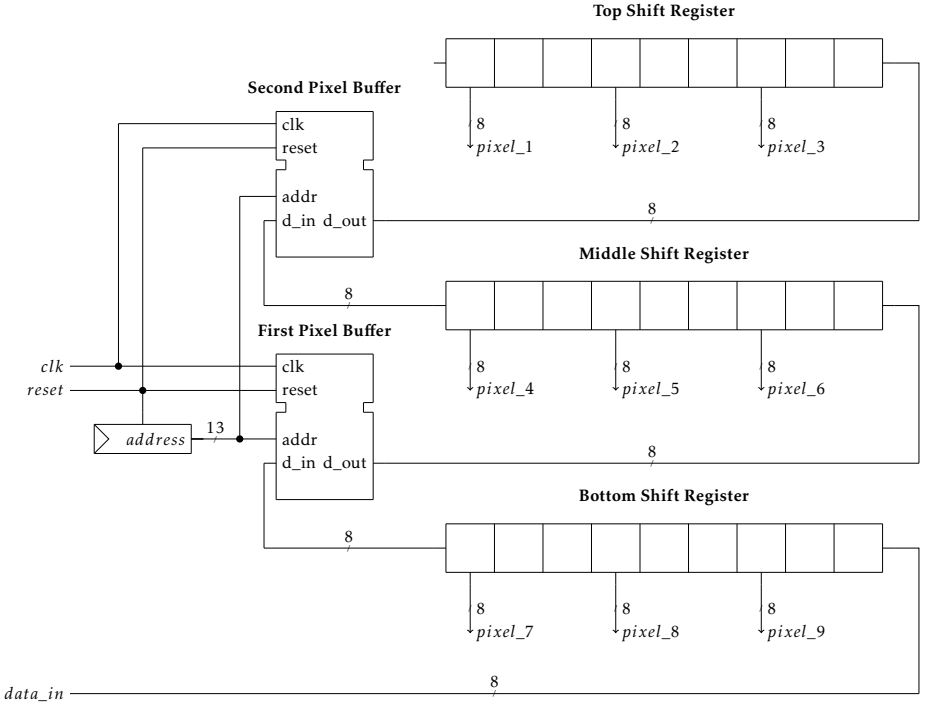


Figure 3.4: *Illustration of the buffer module.*

The circular buffers are designed to fit images that are 2560 pixels wide, if a smaller image is used the address counter's maximum value is lowered to accomodate the smaller images. This means the circular buffer will only address as many addresses as needed. The signal $image\_width$ specifies the width of the image and it sets the maximum value for the address counter. $image\_height$ specifies the height of the image.

The buffer module starts in a filling state, during this state $filling\_buffer = 1$. When it has read the first row and the first six pixel values on the next row it leaves this state, setting $filling\_buffer = 0$. When the buffer module has read the last row of the image it enters an emptying state, setting $emptying\_buffer = 1$. During this state the buffer module is emptying the buffer to let the convolution module process the last row.

3.2.4 Filter Selector Module

This module allows the user to select what filter kernel to be used when running the convolution. All filter kernels are predefined and they are shown in table 3.4 below. The filter kernels listed in section 2.1.1 are implemented in this module. More filter kernels can easily be implemented in the future.

filter_sel	Filter	shift
00	Identity	$0x0$
01	Edge detection	$0x0$
10	Sharpen	$0x0$
11	Gaussian blur	$0x4$

Table 3.4: *Predefined filters and their respective control signal.*

The module assigns the kernel coefficient (KC) signals $kc\_[1:9]$ to the correct values. The shift signal controls the normalizing constant by shifting the result. One right shift is equal to dividing by 2.

3.2.5 Clock Divider Module

The global clock in the system is running on a clock frequency of $f_{clk,g} = 66.7MHz$, this clock frequency is used by most modules in the system. The buffer module is running on a clock frequency of $f_{clk,buf} = 6.67MHz$. The clock divider module is used to divide the incoming clock signal by a factor of 10.

3.3 Parallel Convolutions

The single convolution system forms the core of a two convolution module (TC module), more details about this module can be found in section 3.3.2. The final system that was implemented consists of eight TC modules, which is able to run 16 convolutions in parallel. Figure 3.5 below illustrates a block diagram of the final system.

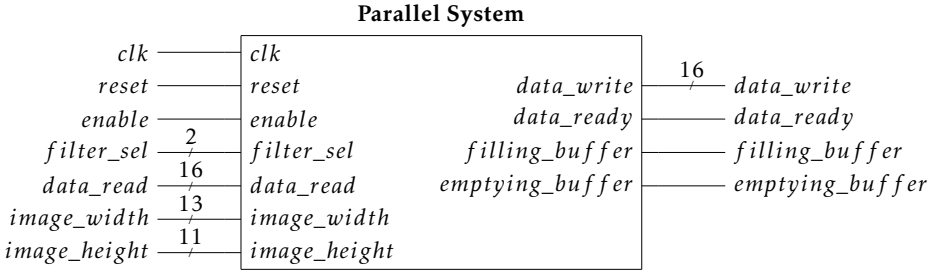


Figure 3.5: *Block diagram of the final system.*

The incoming data is distributed to a TC module that is free, it processes the data and then the result is sent back to the PC. Table 3.5 below lists all the signals and their respective direction.

Signal	Description	Direction
global_clock	Global clock running at 66.7 MHz.	In
filter_sel	Selects what filter to be used.	In
reset	Reset signal.	In
enable	Enables the modules.	In
data_read	Pixel values from the PC.	In
image_width	Specifies the width of the input image.	In
image_height	Specifies the height of the input image.	In
data_out	Pixel values to be sent back to the PC.	Out
data_ready	Signals that data is ready.	Out
filling_buffer	Signals that the buffer is filling up.	Out
emptying_buffer	Signals that the buffer is emptying.	Out

Table 3.5: *List of I/O signals.*

3.3.1 Image Partition

The input image is split into vertical sub-images and each sub-image is treated as an individual images, figure 3.6 below illustrates how the image is split for two parallel convolutions.

Figure 3.6: *Illustration of how the input image is split into two sub-images.*

The advantage of this method is that it is easy to scale up the number of parallel convolutions, each convolution receives its own sub-image of the input image and processes it independently of the other convolutions.

3.3.2 Two Convolution Module

The first method to increase the number of parallel convolutions was to increase the data width from 8 bits to 16 bits. The width of the memory bus in the FPGA is 16 bits and thus it is a good way to motivate this simple modification to increase the number of parallel convolutions. This module is called *Two Convolution Module* (TC module) and is illustrated in figure 3.7 below.

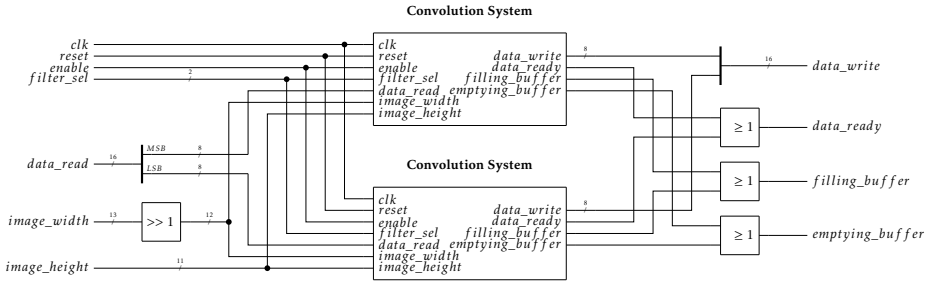


Figure 3.7: *Illustration of the Two Convolution Module (TC module).*

The incoming data is split into two eight bit signals, $MSB = data\_read[15:8]$ and $LSB = data\_read[7:0]$. These signals are sent to two different single convolution systems. The signal $image\_width$ is shifted right once since the image is split between the two systems. The remaining incoming signals are connected to the single convolutions systems without any modifications.

data_write from both systems are merged back into a 16 bit signal. *data_ready*, *filling_buffer* and *emptying_buffer* from both systems are connected to OR gates.

3.3.3 Implemented System

The final implementation of the parallel system is illustrated in figure 3.8 on page 27. As mentioned earlier in this chapter, the system consists of eight TC modules. Due to the limited number of DSP48A1 slices in the FPGA, it can only support 16 simultaneous multiplications. This means that all 16 DSP48A1 slices are utilized by the system.

Since one single convolution requires ten clock cycles to complete, the initial system was idling during nine clock cycles. It was decided to let the parallel system start the TC modules at different times, i.e. $TCM1$ starts at $t = 0$, $TCM2$ starts at $t = 1$ and so on. $TCM1$ is ready to both read and deliver data at $t = 10$.

3.3.4 Pre- and Post-processing

The pre-process is done by a Matlab script. The script uses the border extension policy, it extends the outer border by one pixel. If the desired number of convolutions is two or higher the script will also rearrange the pixel values so that the data can be read sequentially by a PC.

The pixel values are normally ordered after each other in a sequential manner, i.e. P_n is followed by P_{n+1} and so on. With the rearranged order one pixel value from a sub-image is followed by the corresponding pixel value in the next sub-image, i.e. a two parallel convolution system would arrange the pixels as follows: $P_{A,1}$ is followed by $P_{B,1}$ and then $P_{A,2}$ and so on, where A and B are the different sub images.

Table 3.6 below shows an example of how the first row of an image with $W = 8$ would arrange the pixel values in a normal and a rearranged order for two parallel convolutions.

Address	Pixel		Address	Pixel
0x0000	P_1		0x0000	P_1
02	P_2		02	P_5
04	P_3		04	P_2
06	P_4		06	P_6
08	P_5		08	P_3
0A	P_6		0A	P_7
0C	P_7		0C	P_4
0E	P_8		0E	P_8

Table 3.6: *Example of pre-processing. Left: Normal order. Right: Rearranged order.*

Figure 3.9 below illustrates a pre-processed version of the original image. This specific example is made to accomodate two parallel convolutions, as shown in table 3.6, one pixel value from the left sub-image is followed by a pixel value from the right sub-image. Due to this, one can still distinguish the orignal image, but as soon as the desired number of parallel convolutions increases it gets harder to distinguish the original image.

The pre-processed image is sent to the FPGA, the image is convolved and then the result is sent back to the PC.

When the output image is finished it has to be post-processed in order to make it look normal again. The post-process is also a Matlab script that does the following tasks. First it rearranges all the pixel values back to the original order and then it removes the extra layer of border pixels

Figure 3.9: *Illustration of a pre-processed image.*

3.3.5 Border Handling

Another thing to take into consideration is how to handle the new border that is introduced when the image is split into sub-images. If no extra border handling is added one will notice a vertical line that is a result of the wrap-around method used for the borders around the original image. Figure 3.10 below illustrates an example of this effect when no extra border handling is added.

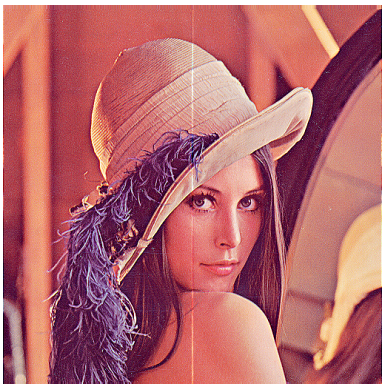


Figure 3.10: *Example of the vertical border between two sub-images.*

This problem was corrected by letting the pre-process extend both the external and the new internal border by one pixel. This means that an image that is m pixels wide and running x number of parallel convolutions will after the pre-process be $m + 2x$ pixels wide.

This solution lets the image convolution to be done correctly on all borders, both internal and external. The wrap-around effect will not be noticable in the final image since the post-process removes the extra pixels that are not needed.

3.3.6 Hardware Utilization

Design Summary

	Used	Available	
Slice Logic Utilization:			
Number of Slice Registers:	2599	11440	22%
Number used as Flip Flops:	2471		
Number used as Latches:	128		
Number of Slice LUTs:	3234	5720	56%
Number used as logic:	2626	5720	45%
Number used as Memory:	576	1440	40%
Number used as Shift Register:	576		
Slice Logic Distribution:			
Number of occupied Slices:	1039	1430	72%
Number of MUXCYs used:	1056	2860	36%
Number of LUT Flip Flop pairs used:	3585		
Number with an unused Flip Flop:	1595	3585	44%
Number with an unused LUT:	351	3585	9%
Number of fully used LUT-FF pairs:	1639	3585	45%
IO Utilization:			
Number of bonded IOBs:	71	200	35%
Specific Feature Utilization:			
Number of RAMB16BWERs:	0	32	0%
Number of RAMB8BWERs:	32	64	50%
Number of BUFG/BUFGMUXs:	9	16	56%
Number of DSP48A1s:	16	16	100%

Table 3.7: *Synthesis results for hardware utilization on Spartan 6 LX9 FPGA.*

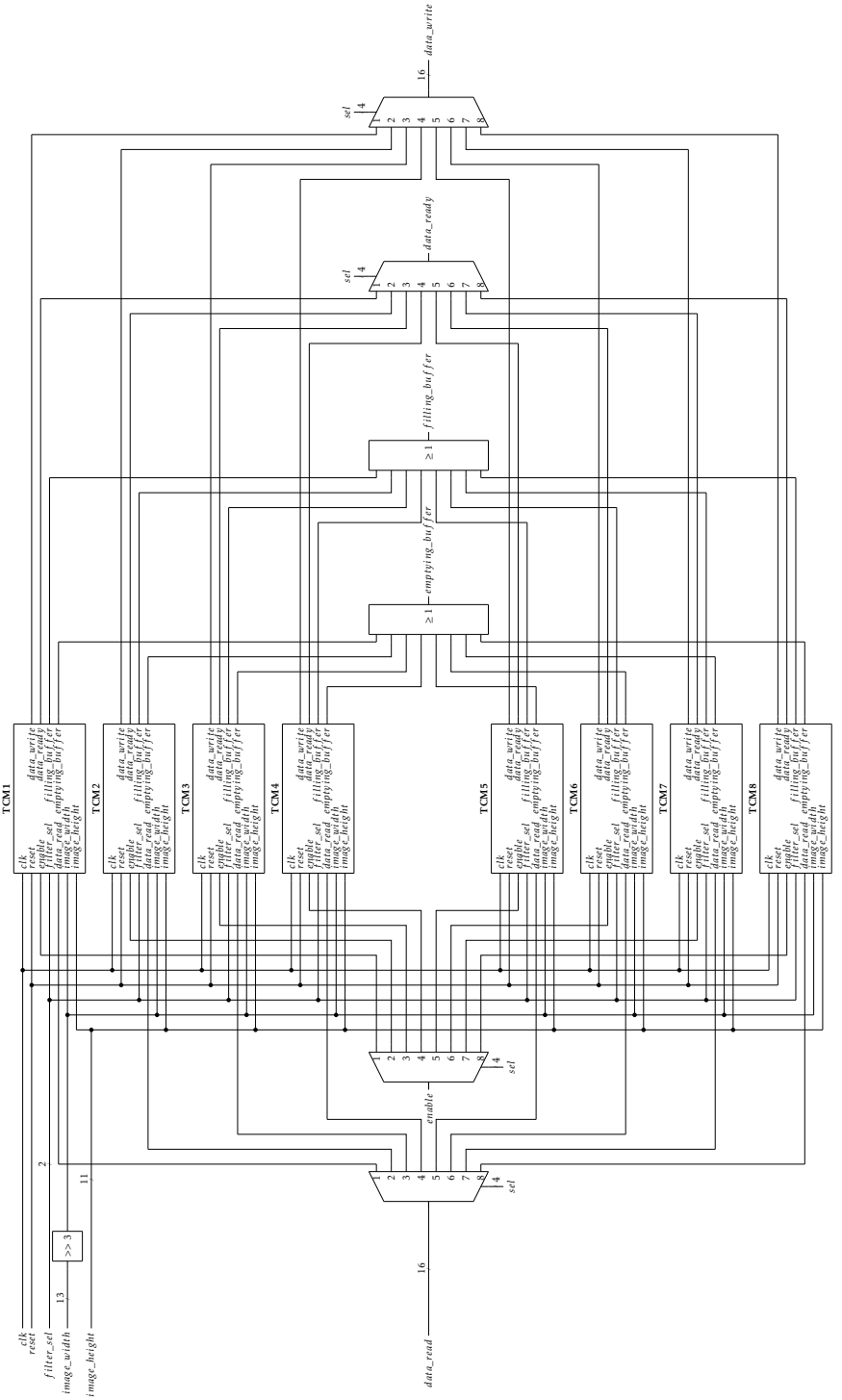


Figure 3.8: *Illustration of the final system with 16 parallel convolutions.*

4
Results

This chapter presents the test results and compare them to test results from previous work.

4.1 Testing

The tests were done by using Xilinx's Integrated Synthesis Environment (ISE) to simulate the implementations. Four images with sizes 852×480 (480p), 1280×720 (720p), 1920×1080 (1080p) and 2560×1440 (1440p) were used. The images were pre-processed and post-processed as discussed in sections 2.1.4 and 3.3.4.

The test bench read the pixels values, represented with ASCII characters, from a .ppm file and assigned them to $data\_read$. When the system signaled that data was ready to be delivered, the test bench read the data on $data\_write$ and wrote the value in another .ppm file. The images were post-processed and then opened in a graphics editor to inspect and verify that the result was correct.

Each filter was also tested to verify that they would not impact the test results. This assumption was correct, since the only difference between the filters are the coefficients.

4.2 Single Convolution

Table 4.1 below presents the results from the single convolution implementation and the single-threaded CPU implementation from [8]. Figure 4.1 illustrates the results.

	480p	720p	1080p	1440p
CPU	0.138s	0.370s	0.828s	1.720s
FPGA	0.185s	0.416s	0.935s	1.661s

Table 4.1: *Single convolution implementation results.*

The FPGA used a clock frequency of $f_{clk,g} = 66.7MHz$. The CPU was an i7-4790 running at 3.6GHz.

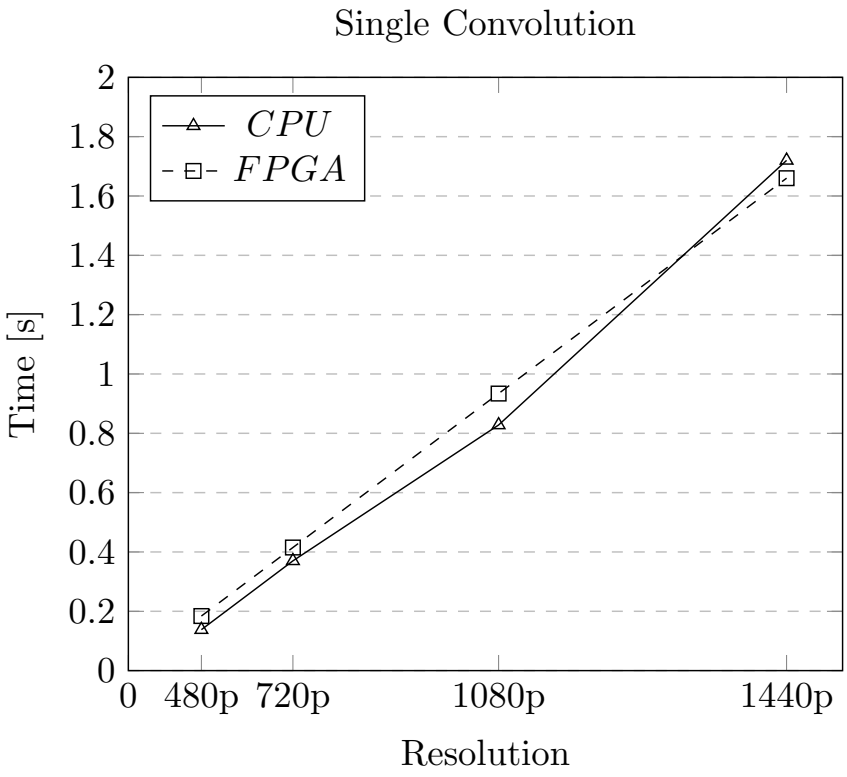


Figure 4.1: *Single convolution test results.*

4.3 Parallel Convolutions

Table 4.2 below presents the results from the parallel convolutions implementation, the multi-threaded CPU and GPU implementations from [8]. Figure 4.2 illustrates the results.

	480p	720p	1080p	1440p
CPU	0.030s	0.061s	0.125s	0.221s
CPU (Mod.)	0.025s	0.048s	0.110s	0.191s
GPU	0.050s	0.092s	0.198s	0.347s
GPU (Mod.)	0.065s	0.121s	0.247s	0.449s
FPGA				
2 Conv.	0.093s	0.208s	0.468s	0.831s
4 Conv.	0.047s	0.104s	0.234s	0.416s
8 Conv.		0.053s	0.118s	0.209s
16 Conv.		0.027s	0.059s	0.105s

Table 4.2: *Parallel convolution implementation results.*

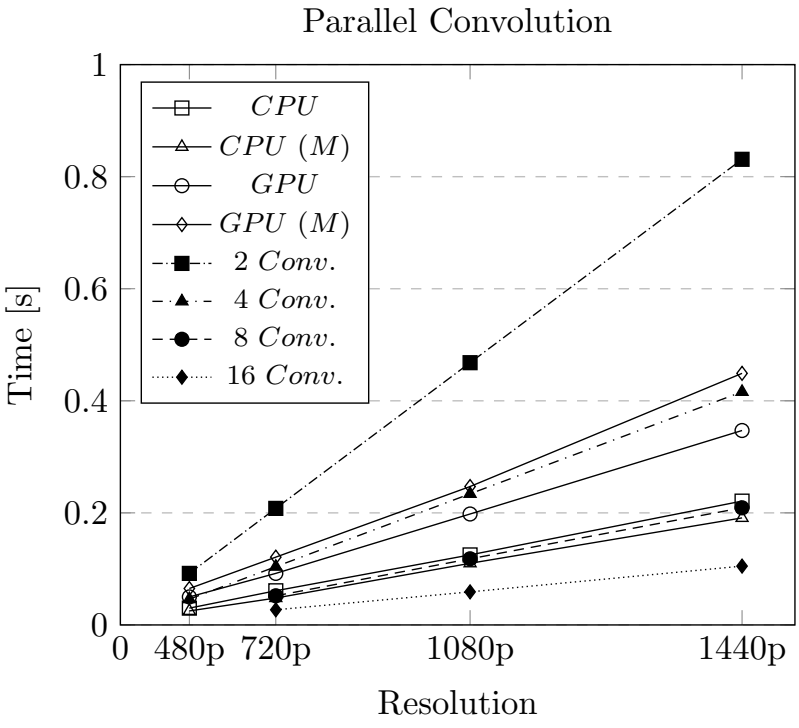


Figure 4.2: *Test results from parallel convolution implementations on FPGA, CPU and GPU.*

The FPGA used a global clock frequency of $f_{clk,g} = 66.7MHz$. The CPU was an i7-4790 running at 3.6GHz. The GPU was an Intel HD Graphics 4600 running at 1.2GHz.
(M) denotes the modified algorithm implemented in [8]. The modified algorithm used zero-copying to avoid unnecessary overhead.
No tests were conducted on a 480p image with eight or 16 parallel convolutions. The implementation requires all sub-images to have the same width.

4.4 Scaling

Table 4.3 presents and figure 4.3 illustrates the results for each image resolution when scaling up from one to 16 parallel convolutions on the FPGA.

#Convolutions	480p	720p	1080p	1440p
1	0.185s	0.416s	0.935s	1.661s
2	0.093s	0.208s	0.468s	0.831s
4	0.047s	0.104s	0.234s	0.416s
8		0.053s	0.118s	0.209s
16		0.027s	0.059s	0.105s

Table 4.3: *Processing time for parallel convolutions.*

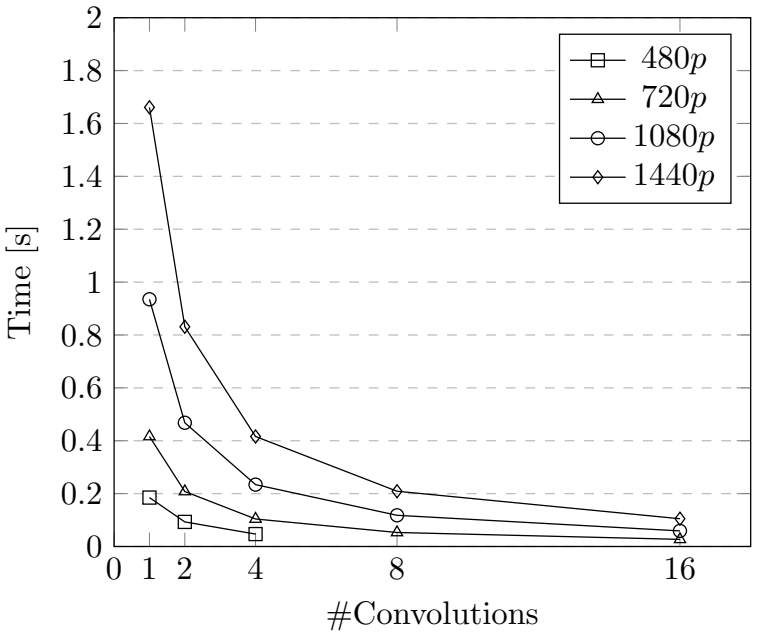


Figure 4.3: *Time vs. number of convolutions.*

Table 4.4 and figure 4.4 below presents the test results for the FPGA in clock cycles.

#Convolutions	480p	720p	1080p	1440p
1	1232328	2772972	6233052	11075532
2	617610	1388652	3119772	5542092
4	310252	696492	1563132	2775372
8		350412	784812	1392012
16		177372	395652	700332

Table 4.4: *Number of clock cycles for parallel convolutions.*

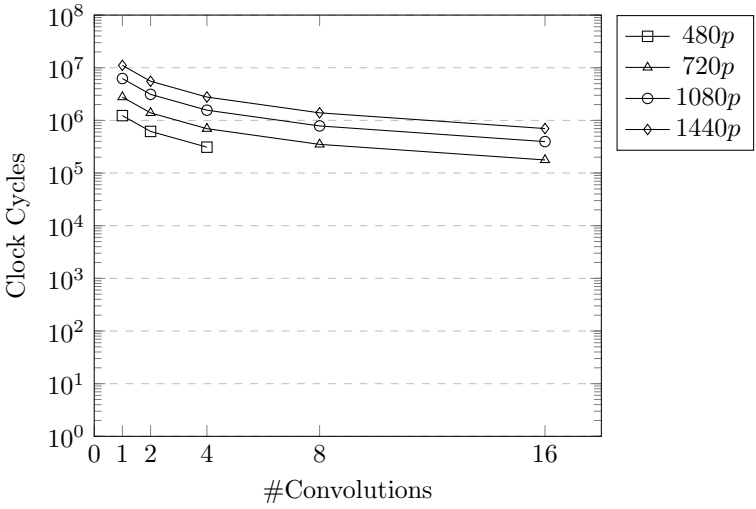


Figure 4.4: *Clock cycles vs. number of convolutions.*

5
Discussion

This chapter discusses the design choices that were made during implementation, the results from the previous chapter, suggestions on how to improve the implemented system and what can be done in the future.

5.1 Method

In order to keep the project moving forward when a problem was encountered a couple of design choices had to be made. These choices were discussed with the supervisor before a final decision was taken.

5.1.1 Vertical vs. Horizontal Sub-images

Section 2.1.4 discussed the method of splitting the original image into smaller sub-images. It was decided to split the input image into vertical sub-images, the main reason to split it into vertical rather than horizontal slices was to keep the total size of the pixel buffers constant. If horizontal slices would be used the total size of the pixel buffers would increase linearly, i.e. with two parallel convolutions it would require twice as much BRAM. The pixel buffers have to store two rows of the input image.

5.1.2 Pre- and Post-processing

The pre- and post-processing scripts were created to make it easier to test the implementation. The read from file function in VHDL can only read the first line in a text file. Due to this, the input images had to be modified in order to be

able to read the input files. As explained in section 3.3.4, the pixel values were rearranged so that the pixel values could be read after each other.

The write to file function is also very primitive, it can only write on the last line in a text file. When the image had been processed by the FPGA, all the pixel values had to be arranged back to their correct positions.

If the implementation would be used in a real scenario, the pre- and post-process scripts could be removed. The user would then have to let the PC send the correct pixel values to the FPGA and afterwards the pixel values would have to be arranged back to their correct position in the text file.

5.1.3 Clock Frequency

The FPGA has three pre-programmed clock frequencies: 40MHz, 66.7MHz and 100MHz. The initial plan was to use $f_{clk,g} = 100MHz$. The single convolution implementation was able to fulfill the timing constraints for $100MHz$. When the number of parallel convolutions increased, the timing constraints could not be fulfilled. The convolution module was not completely sequential and the data had to propagate through five levels of logic. This meant that either the timing constraints had to be relaxed or the convolution module had to be redesigned. Due to the limited amount of time it was decided to go with the former alternative. The new clock frequency was set to 66.7MHz, $f_{clk,g} = 66.7MHz$.

5.1.4 Convolution Module

As mentioned in section 1.4, the focus of this thesis work was to run as many convolutions in parallel as possible. The FPGA had 16 DSP48A1 slices that could be used as multipliers and thus a maximum of 16 convolution modules could be implemented.
The convolution module could have been designed with two or more multipliers instead of one. With two multipliers the processing time for one pixel would have been halved, five clock cycles instead of ten. This solution would have been slightly faster than the implemented solution if all timing constraints are fulfilled. However, only eight parallel convolutions could have been implemented.

5.1.5 Improvements

This section discusses suggestions on how to improve the performance of the implementation.

Convolution Module

The convolution module needs redesign to allow the system to run on a higher clock frequency. The current implementation requires only nine clock cycles to process the pixel values in the filter kernel, this means the data has to be read and processed on the first clock cycle. By allowing the process to take ten clock cycles, which would not impact the current system, the data can be read and

written during one clock cycle and then the pixel values can be processed during the remaining nine clock cycles.

Buffer Module

One alternative that was brought up during the project was the possibility to use two filter kernels next to each other. To stop the two filter kernels from processing the same pixels twice, they have to move two pixels every clock cycle. This improvement would be rather easy to implement in the current implementation, it requires a new design of the buffer module. Instead of splitting the incoming data in the Two Convolution Module, it would sent directly to the new buffer module. Figure 5.1 below illustrates the new buffer module. The pre- and post-processing have to be done in a different way, the color channels have to be separated. This means the red channel is processed first, followed by the green channel and then the blue channel. The image partitioning is still required to allow more parallel convolutions.

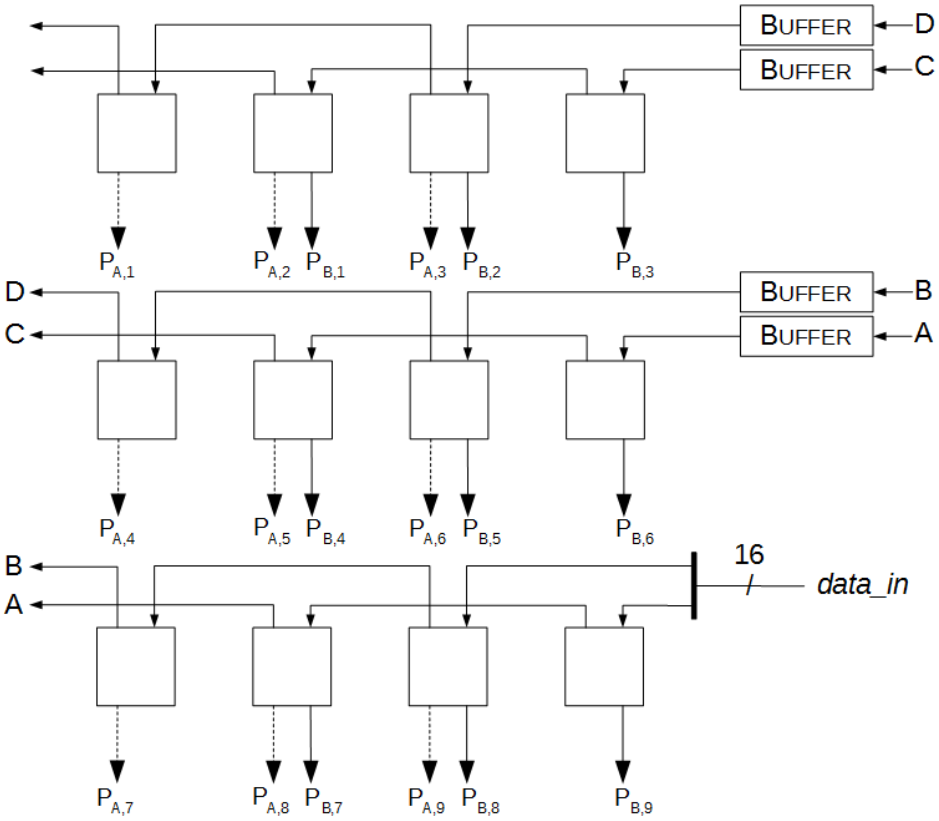


Figure 5.1: *Illustration of the new buffer module.*

Each level of the new buffer module consists of two shift registers, as illustrated

in figure 5.2 below. The pixel values, denoted $P_{A,X}$ and $P_{B,X}$ in the illustration, are sent to two different convolution modules. Each circular buffer in the original implementation is split into two circular buffers to accommodate the data.

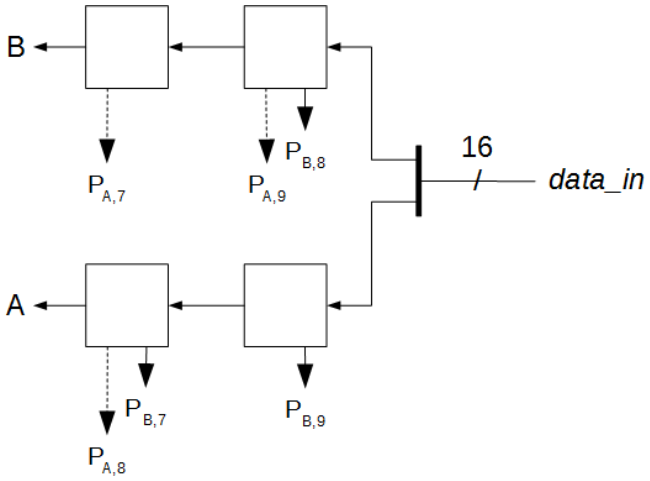


Figure 5.2: *Illustration of a rearranged bottom shift register.*

Every clock cycle, 16 bits of data enters the new buffer module. The data is split into two eight bit signals and then fed to the different shift registers.

This alternative would reduce the required hardware a bit. The number of circular buffers would increase, but they would still hold the same amount of data as the current implementation. Another advantage of this alternative is that the image partitioning would only require the input image to be split into eight sub-images, compared to 16 in the current implementation.

Another advantage is that the total number of pixels in the input image would be fewer than the current implementation, this is due to the extension of the sub-images by one pixel on each side. E.g. a 1080p (RGB) image in the current solution contains $1952 \cdot 1080 \cdot 3 = 6324480$ pixels. The new solution would only consist of $1936 \cdot 1080 \cdot 3 = 6272640$ pixels.

A similar solution to this implementation is presented by the authors of reference [13]. Their proposed idea, called Multiwindow Partial Buffer (MWPB), is to reuse data that is already in the internal buffers.

5.2 Results

Overall, the results were not too different from what was expected. The initial plan was to implement single convolution system that could process one pixel per clock cycle. This system would then be duplicated to utilize all DSP48A1

slices in the FPGA. The lower clock frequency had a large impact on the final results.

5.2.1 Single Convolution

The results for the single convolution system were in line with what was to be expected. The system was designed to process one pixel per clock cycle. For an image with resolution 1920×1080 pixels the theoretical time would be:

$$\frac{1920 \cdot 1080 \cdot 3}{6.67MHz} = 0.933s \tag{5.1}$$

Compared with the actual result, 0.935s, the implemented system is coming close to the theoretical time. The system has to process an image that is two pixels wider, due to the extended border added by the pre-processing, and it also has to spend a couple of clock cycles to fill the pixel buffers.

Compared with the single-threaded CPU implementation the FPGA implementation does match it fairly well. Looking at figure 4.1 one can see that the FPGA implementation is linear and the CPU implementation varies depending on the image resolution. It would be interesting to see the results for 3840×2160 (4K) and 7680×4320 (8K) images. The theoretical times would be around $3.73s$ for 4K and $14.93s$ for 8K. The test data from [8] reveals that the CPU implementation took $3.99s$ for 4K and $15.97s$ for 8K. This would mean that the FPGA images is more efficient for larger images.

5.2.2 Parallel Convolutions

The first thing one has to take into consideration is that the GPU used in [8] was the integrated GPU on the CPU. This GPU is far from being a high performance graphics card, but it is still interesting to compare it against the FPGA implementation. On the other hand, the Intel i7 4790 CPU can be considered to be in the higher range when it comes to performance.

Looking at the results in table 4.2 and figure 4.2, the 16 parallel convolution implementation could process the image almost twice as fast as the optimized CPU implementation. The interesting result is the eight convolution implementation that manages to place itself between the two CPU implementations.

In the discussion following the first question in 1.3 it is mentioned that it was of interest to compare the final FPGA implementation against the implementation in [5]. Their implementation manages to process one pixel per clock cycle. The final implementation in this thesis work manages to process 16 pixels per clock cycle, one pixel per convolution module.

It would be interesting to compare the power consumption required to process the same image.

5.2.3 Scaling

One of the research questions stated in section 1.3 was to investigate if the performance would scale proportionally to the number of convolutions running in parallel. Figure 4.3 and table 4.3 can be used to answer this question.

Looking at the 1440p results and compare the times for one convolution and two convolutions. For one convolution it takes $1.661s$ and for two convolutions it takes $0.831s$. By calculating $1.661 \div 0.831 \approx 1.999$, which means that when the hardware is doubled the processing time is halved. Doing the same calculation for one and sixteen convolutions, $1.661 \div 0.105 \approx 15.82$. The deviation is most likely caused by the border extension.

6
Conclusions

This chapter presents the answers to the problem statements and possible ways to expand the work in the future.

6.1 Answers to Problem Statements

- *What is the performance of the parallel image convolution compared to a CPU and a GPU implementation?*

The performance of the single convolution system did match the single threaded CPU implementation. In this case the CPU implementation was faster than the FPGA for smaller images. For images larger than 1080p the FPGA implementation was faster.

The parallel convolution implementations on the FPGA required four parallel convolutions to be able to compete with the GPU implementations in [8]. To ensure better performance than the GPU it requires at least eight parallel convolutions. A high performance GPU would most likely outperform the 16 convolution implementation, in terms of processing time. In [5] the algorithm was implemented with CUDA running on a GTX295 graphics card. The CUDA implementation is roughly 25 times faster than the final implementation in this thesis.

At least eight parallel convolutions were required to outperform one of the CPU implementations in [8] and 16 to beat both implementations. The CPU was a high performance CPU, thus making it a good candidate to compare against.

- *Does the performance scale proportionally to the number of convolutions running in parallel?*

This question was answered in 5.2.3. The initial assumption was that it would scale proportionally to the number of convolutions. The conclusion is: if the hardware is doubled, the processing time is halved.

- *What are the limiting factors when implementing the parallel image convolution on an FPGA?*

The limiting factor that impacted the implemented system the most was the combinational logic. As discussed in 5.1.5, this would have not been a problem if the convolution module was implemented sequentially. Another limiting factor was the DSP48A1 slices, this is easily solved by using an FPGA with more slices.

6.2 Future Work

The improvements listed in section 5.1.5 would be the first thing to implement. One thing that would be interesting to investigate is how a larger filter kernel can be implemented. The current implementation would require the convolution to be done in 25 clock cycles for a 5×5 filter kernel. This would force the user to either drastically increase the global clock frequency or use more multipliers in the convolution module.

Another thought is to combine this work with the work presented in [6]. The authors are running a MicroBlaze soft processor implemented on a Virtex 4 FPGA. They use this to recognize different traffic signs in a video feed. The video feed is limited to 640×480 pixels with a 60Hz frame rate. In the conclusion they mention that the image convolution algorithm in their work could be implemented as a subsystem to increase the performance.

Bibliography

[1] B. Bosi, G. Bois, and Y. Savaria. Reconfigurable pipelined 2-D convolvers for fast digital signal processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(3):299–308, September 1999. ISSN 1063-8210. doi: 10.1109/92.784091. Cited on pages 12, 13, and 19.

[2] Wilhelm Burger and Mark James Burge. *Digital Image Processing - An Algorithmic Introduction Using Java*. Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA, 1st edition, 2008. ISBN 978-1-84628-379-6. doi: 10.1007/978-1-84628-968-2. Cited on page 8.

[3] A. Ehliar. TSEA26 - Lecture 2, Numerical Representations, 2015. URL `http://www.isy.liu.se/en/edu/kurs/TSEA26/lectures/02-Numerical.pdf`. Cited on page 8.

[4] Chi kin Lee and Mounir Hamdi. Parallel image processing applications on a network of workstations. *Parallel Computing*, 21 (1):137–160, January 1995. ISSN 0167-8191. doi: 10.1016/0167-8191(94)00068-L. URL `http://www.sciencedirect.com/science/article/pii/016781919400068L`. Cited on page 11.

[5] L. M. Russo, E. C. Pedrino, E. Kato, and V. O. Roda. Image convolution processing: A GPU versus FPGA comparison. In *Proc. Southern Conf. Programmable Logic*, pages 1–6, March 2012. doi: 10.1109/SPL.2012.6211783. Cited on pages 1, 2, 13, 39, and 41.

[6] Peter Ševčik. Traffic sign recognition system based on the FPGA device utilization. *Proceedings in ITS 2013-Inteligent Transportation Systems*, (1), 2013. Cited on page 42.

[7] R. G. Shoup. Parameterized convolution filtering in a field programmable gate array interval. Technical report, 1993. Cited on page 12.

[8] A. Söderholm and J. Sörman. GPU-acceleration of image rendering and sorting algorithms with the OpenCL framework, 2015. Bachelor's thesis. Cited on pages 1, 2, 30, 31, 32, 39, and 41.

[9] F. Talbi, F. Alim, S. Seddiki, I. Mezzah, and B. Hachemi. Separable convolution Gaussian smoothing filters on a Xilinx FPGA platform. In *Proc. Int. Conf. Innovative Computing Technology*, pages 112–117, May 2015. doi: 10.1109/INTECH.2015.7173372. Cited on page 1.

[10] Xilinx Inc. *Spartan-6 Family Overview, v2.0*, March 2011. URL `http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf`. Cited on page 16.

[11] Xilinx Inc. *Xilinx Spartan-6 FPGA LX9 MicroBoard - User Guide, Rev. C*, August 2011. URL `http://www.em.avnet.com/Support%20And%20Downloads/Xilinx_Spartan-6_LX9_MicroBoard_Rev_B2_Hardware_User_Guide.pdf`. Cited on page 16.

[12] Shuling Yu, Mark Clement, Quinn Snell, and Bryan Morse. Parallel algorithms for image convolution. In *Proc. Int. Conf. Parallel and Distributed Techniques and Applications*, 1998. Cited on page 11.

[13] Hui Zhang, Mingxin Xia, and Guangshu Hu. A multiwindow partial buffering scheme for FPGA-based 2-D convolvers. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 54(2):200–204, February 2007. ISSN 1549-7747. doi: 10.1109/tcsii.2006.886898. Cited on page 38.