Evaluation of High Level Synthesis for the implementation of Marker Detection on FPGA

October 2011

Author:
Tim Vriends

Supervisor:
Prof. Dr. H. Corporaal

Tutor:
Ir. Y. He

Committee:
Prof. Dr. H. Corporaal
Prof. Dr. Ir. P. Jonker
Ir. Y. He
Abstract

More and more real-time vision applications become available nowadays. Vision pipelines of such applications are computational intensive and have tight constraints on throughput and latency. FPGAs are often used to make sure that those vision pipelines are executed within their time budget. However, programming FPGAs is a time consuming and error prone job. In our Embedded Vision Architectures (EVA) project, a toolchain will be developed which supports the porting from sequential C (algorithms are usually written in sequential C for a general purpose processor) to HDL code which can run on an FPGA. The last stage of this toolchain will probably make use of a High Level Synthesis (HLS) tool. There are mainly two case studies in the EVA project. In this work, one of them, Augmented Reality (AR) will be used to evaluate the capabilities of HLS tools and compare the performance (throughput, latency, used resources on FPGAs) with a handwritten and highly tuned implementation of the same vision processing kernels.

The ARToolkit is one the most popular algorithms to create AR. It uses artificial markers to position and rotate the augmented objects onto the view of the real world. Augmented reality, and therefore ARToolkit, has tight constraints on latency and accuracy when detecting these artificial markers. The vision pipeline of ARToolkit consists of several different sub-algorithms (kernels). We start with a software implementation on both a Microblaze processor (soft-core in FPGAs), which achieves 5 fps, and an ARM processor, which achieves 20 fps. This initial Microblaze implementation can be accelerated using hardware accelerators. By accelerating with hardware IPs, the frame rate is increased to 20 fps, comparable with the ARM frame rate. After removing or omitting some of the Microblaze limitations a frame rate of 36 fps can be expected.

Writing these accelerators by hand is a very time consuming job. With the use of a High Level Synthesis tool, the development time can be reduced. Two of these tools are evaluated in this thesis. The first evaluated one, ROCCC, was not suitable for creating accelerators which can communicate with a Microblaze directly through a FIFO interface. The second one, AutoESL, was more suitable for this job. The generated accelerators reached a performance in the range of 6 to 100 percent slower compared to their handwritten counterparts. The design time of the accelerators was approximately a factor of 5 lower and during this design more architectures are explored.

The use of one of these HLS tools as final stage (HDL generation) in the EVA project is debatable after this case study. One of the strong points of an HLS tool, reducing the design time of a specific kernel by fast exploring many different designs, is not a main issue in the EVA project. If the guidelines of the code generation are available, this advantage is lost. So far, HLS tools only create accelerators which use more resources, have a lower throughput and higher latency, as their handwritten counterparts.
# Contents

1 Introduction .................................................. 1
   1.1 Case study: Augmented Reality .......................... 1
   1.2 Contributions and overview .............................. 3

2 Related work .................................................. 5
   2.1 Marker detection ......................................... 5
   2.2 Markerless detection .................................... 7
      2.2.1 Natural Marker Detection ............................ 7
      2.2.2 Natural Feature Detection ........................... 7
   2.3 Implementation of the ARToolkit on other platforms .... 8
   2.4 High Level Synthesis Tools ............................. 9

3 The marker detection algorithm ............................... 10
   3.1 Overview of the processing pipeline ..................... 10
   3.2 Pipeline stages explained in detail ...................... 11
      3.2.1 Binarization ....................................... 11
      3.2.2 Labeling ........................................... 12
      3.2.3 Contour detection .................................. 13
      3.2.4 Check square ....................................... 13
      3.2.5 Remove components in components .................. 14
      3.2.6 Pattern extraction .................................. 15
      3.2.7 Template matching .................................. 16
      3.2.8 Line contour estimation ............................. 17
      3.2.9 Sub-pixel corner detection ......................... 17

4 Requirements and Testing ..................................... 18
   4.1 Accuracy definition of the output variables .......... 18
   4.2 Frame rate .............................................. 20
   4.3 Platform definition ..................................... 20
   4.4 Testing ................................................ 21

5 ARM implementation .......................................... 23
   5.1 Initial implementation and algorithmic changes ...... 23
   5.2 Optimizations .......................................... 23
   5.3 Evaluation and improvements ............................ 24
6 Manual FPGA implementation

6.1 Floating Point or Fixed Point ........................................... 26
6.2 Initial Implementation .................................................. 27
6.3 Hardware accelerators .................................................. 28
  6.3.1 Binarization .......................................................... 29
  6.3.2 Labeling .............................................................. 31
  6.3.3 Pattern extraction ................................................... 34
  6.3.4 Evaluation ............................................................ 36
6.4 Combining Hardware accelerators ...................................... 38
6.5 Hardware accelerators - revisited .................................... 39
  6.5.1 Improvement 1: Dedicated camera link or DMA ................. 39
  6.5.2 Improvement 2: Use more resources for the pattern extraction .. 40
  6.5.3 Improvement 3: Line contour estimation accelerator ............ 40
  6.5.4 Overview of the improvements .................................... 41

7 FPGA implementation using High Level Synthesis tools ............... 42

7.1 ROCCC ......................................................................... 42
  7.1.1 Modules and Systems ................................................ 42
  7.1.2 Compiler directives ................................................... 43
  7.1.3 Binarization accelerator ............................................ 43
  7.1.4 No other accelerators ............................................... 44
7.2 AutoESL ....................................................................... 44
  7.2.1 Testing ................................................................... 44
  7.2.2 Compiler directives ................................................... 44
  7.2.3 Binarization ............................................................ 45
  7.2.4 Labeling .............................................................. 45
  7.2.5 Pattern Extraction .................................................... 47
7.3 Evaluation of the two HLS tools ......................................... 49
  7.3.1 ROCCC ................................................................. 49
  7.3.2 AutoESL ............................................................... 49

8 Conclusions and Future Work ............................................. 51

8.1 Conclusions .................................................................. 51
8.2 Future Work .................................................................. 52

A Pseudocode of the sub-algorithms ........................................... 55
  A.1 Labeling ................................................................... 55
  A.2 Contour following ....................................................... 56

B Fixed point variables overview ............................................ 57
Chapter 1

Introduction

Nowadays more and more real-time vision applications are used. Vision applications are not only used for control purposes in industry, but educational and entertainment applications become more mainstream nowadays. Many of those vision applications are very computational intensive. A common way to make sure that the application will meet its real-time requirements is the use of FPGAs as accelerators. An FPGA can perform computational intensive tasks and therefore mitigate the constraints for the other parts of the system design. However, FPGAs must be programmed in a Hardware Description Language (HDL) and generating HDL code by hand is a time consuming and error prone job. Besides generating this HDL code by hand, a second method becomes more available nowadays. Generating HDL code from sequential C can be done using a High Level Synthesis (HLS) tool. Using High Level Synthesis will reduce the implementation time and improve the quality of the HDL code (human mistakes in the HDL code will be excluded).

At the University of Technology in Eindhoven research is performed regarding HLS tools and their quality. The EVA project aims at generating a generic architecture for vision applications on FPGA. This generation will be done using a tool flow where the last part covers the generation of HDL code from C code (or an extended version of C). The generation of this HDL code can be done using predefined templates in HDL or using an HLS tool, but can an HLS tool create HDL code which has the same performance (latency and throughput) and uses the same amount of resources as a handwritten implementation? Which other advantages and disadvantages can be discovered when using an HLS tool and are there any tradeoffs (area/performance/design-time) which have to be made when using HLS? This is verified in a case study.

The rest of this chapter is organized as follows. Section 1.1 describes the case study which is mentioned above. The section thereafter, Section 1.2, gives an overview of structure of the thesis and lists the contributions of the author.

1.1 Case study: Augmented Reality

Augmented Reality (AR) can be described as the view of a physical real-world environment whose elements are augmented. This live view, direct or indirect, displays elements that are computer-generated via sensory input. The computer modifies the view of reality and enhances one’s current perception of reality. AR is not a complete virtual world, but consists of virtual objects or information superimposed on the real world in real-time (where real-time
means; the placement is fast enough so it is not annoying for the user). By contrast, virtual reality replaces the real-world with a simulated one.

To provide the user at the right time with the right information, the positioning of the virtual objects must be accurate and fast. For extracting and delivering this information, virtual markers can be used. One of the first introduced algorithms to create AR is the ATRoolkit (introduced in 1999, [14]). ATRoolkit is still one of the most popular algorithms.

An example of the ATRoolkit is given in Figure 1.1(a). A video stream from a camera is captured, analyzed and augmented. The marker below the blue block (not visible anymore) is extracted and the artificial blue block projected onto the marker. The combined (augmented) video stream is displayed on a screen.

AR can be used in many different domains. The first AR applications were used in industry. A nice example is given in [18]. This application helps workers in a factory with assembling a car door. Every instruction during the assembly is guided and explained in detail. This information is displayed on two small screens directly in front of the eyes, called a Head Mounted Display (HMD). Especially for new workers this is really handy and increases the productivity from the start. Another example from industry is Cylicon (see Figure 1.1(b)). This application is introduced in [4] and is meant for service applications. The augmented reality provides instructions which parts must be checked and eventually replaced.

![AR Toolkit and Cylicon](image)

(a) ATRoolkit  
(b) Cylicon

Figure 1.1: Examples of Augmented Reality Applications

Augmented reality becomes more important in the consumer applications as well as in games. Layar is a nice example of a consumer AR application ([2]). This application for Smartphone’s, augments the world with information about interesting objects. The real world is seen through the camera and augmented with virtual objects displayed on the screen. Information like shops, restaurants and train stations can be displayed. The information is gathered via Internet and the position is determined with the built-in GPS and compass modules. A few AR games are developed for entertainment. Invisible train is an example. For this game, a wooden track must be placed on the floor. The track is viewed via a handheld device and is augmented displayed on the screen. It is possible to change the virtual track switches to direct the train onto another track. An example of the Invisible train is given in Figure 1.2(a).

A fairly recent project from the Human Interface Technology Laboratory in New Zealand
is the Magic Book. This book is used to teach people the history of New Zealand. This book must be read/viewed via a HMD. The HMD has a camera mounted in front of it. The text in the book is augmented with moving scenes and dialogs between characters, during reading the book. An example of the magic book can be seen in Figure 1.2(b).

An AR application consists in general of two main parts. The first part is a vision pipeline which extracts the information from the captured image of the video stream. The second part augments the image with virtual objects based on the extracted information of the first part. The focus of the EVA project is on vision pipelines and therefore only the first part of an AR application is taken as the case study. The second part consist usually of well know visualization kernels, like OpenGL, and is therefore not so interesting for the EVA project. To evaluate the differences between handwritten code and HLS generated code an implementation with both methods must be made. So, the vision part of an AR application is implemented twice. The first implementation is a manual implementation, all the HDL code is written by hand. The second implementation is done using an HLS tool, the HDL code is generated from sequential C code using an HLS tool. Both methods are evaluated and compared. The main question which is answered in this thesis is: Can an HLS tool generate HDL code which is as efficient as its handwritten counterpart and are there tradeoffs which must be made. Efficiency, and also the tradeoffs, can be measured in performance (latency and throughput), used resources (area on the FPGA) and design time (time that is needed by the designer to achieve a certain performance).

![Invisible train game](image1.png) ![Magic Book](image2.png)

Figure 1.2: Examples of augmented reality applications

1.2 Contributions and overview

The two methods mentioned in the previous section (handwritten and HLS implementation of a vision pipeline) are evaluated in this thesis. But before the actual implementations were started, the requirements were defined. A vision pipeline has certain outputs. These output variables must have a certain accuracy to guarantee correct rendering of the objects in the next functions. A definition of the accuracy of the output variables is necessary. A test sequence is developed to check the produced outputs. The time budget (which implies the frame rate) must be defined as well. To enable a fair comparison between an FPGA
and another embedded platform, the vision pipeline is first ported to an ARM processor board. After that, an implementation is made by hand. The second FPGA implementation is implementation using an HLS tool. These two FPGA implementations are evaluated and some conclusions are made over these implementation.

This thesis is organized as follows. Chapter 2 describes the related work. The algorithm which has to be ported is explained in Chapter 3. Chapter 4 defines the requirements of the algorithm and the different platforms. The first implementation on the ARM processor board is described in Chapter 5. Implementation on FPGA is done in two ways, by hand (Chapter 6) and using High Level Synthesis tools (Chapter 7). Conclusions and Future work are given in Chapter 8.
Chapter 2

Related work

To create augmented reality from a video sequence, different steps have to be preformed. This is illustrated in Figure 2.1. An input image must be analyzed and the information must be extracted (gray block) from the image. The extracted information is used to render the virtual object(s) in the image (green block). The analysis/information extraction can be done in different ways. As indicated in the gray block this can globally divided in two sections, marker detection and markerless detection. Marker detection is based on specially placed artificial markers and is discussed in Section 2.1. The second way, markerless detection, is discussed in Section 2.2. As explained before, the visualization and rendering of the virtual objects is not taken into account in this research. Common techniques and toolkits, like OpenGL, can be used for this. The EVA project aims at vision processing pipelines and therefore the rendering part is not taken into account.

The choice for ARToolkit is motivated in the first two sections of this chapter. After choosing an algorithm, some research to different ports of AR algorithms is preformed. To the best of my knowledge, we are the first ones who implement the ARToolkit Marker detection algorithm on FPGA. However, the ARToolkit has already implementations on other platforms and there are some attempts to use an FPGA to create Augmented Reality. This is described in Section 2.3.

The final part of the described related work is about High Level Synthesis Tools. The related work regarding HLS is given in Section 2.4. Some sub-algorithm specific literature is studied as well. This specific literature is discussed in subsection 6.3.4 where the sub-algorithm implementation is analyzed. A certain foreknowledge is required to give a small overview of this literature.

2.1 Marker detection

ARToolkit is the first toolkit which can construct Augmented Reality. It is introduced in [14] and an example is given in Figure 1.1(a). ARToolkit remains one of the most popular toolkits to construct AR. ARToolkit relies on artificial placed markers. However, ARToolkit is not the only marker detection algorithm. Another algorithm proposed at the beginning of Augmented Reality is Arloc ([25]). Improvements of the ARToolkit algorithm are also proposed. ARTag ([8]) and ARToolkitplus ([22]) are examples of the improvements. ARToolkit is the only AR algorithm which has an open source marker detection algorithm. Most other algorithms are free for non-commercial use, but the detection algorithm is closed and cannot be viewed.
Almost all algorithms use the same kind of markers (with a solid black square as border) but other options are possible, like circle markers or QR codes. An advantage of the square markers is the ease of extraction of information (like absolute 3D position and orientation) that is required for AR applications. To distinguish different markers, they contain unique features inside the black square. The detection of these inner parts of the markers is the part where the ARToolkit and its improvements differ the most. Some comparison studies between different marker detection algorithms have been made. [24] compares ARToolkit and ArLoc. In terms of execution time the ARToolkit is better at single and at multiple markers per frame. The exact position extraction is better with ArLoc. The matching algorithm from ARToolkit is better on small markers and the matching on large markers is better with Arloc. In the introduction paper of ARTag a comparison is made between ARTag and ARToolkit. The ARTag uses as basis the ARToolkit, but has an improved matching algorithm and does not use binarization. Together this leads to a lower false positive rate (positive identification of a maker which is none or another marker) and less inter-marker confusion (identification of a certain marker where it is actually an other marker). In terms of speed there is almost no difference between ARToolkit and ARTag. The other improvement on ARToolkit is ARToolkitplus. One of the main differences here is: the binarization threshold is determined at runtime and this is done each iteration. Another difference is the use of binary code markers instead of template markers. Binary code markers do not have a human recognizable pattern inside (like a character, digit or drawing) but contain a 2D binary code. Like the markers in Figure 1.2(a). The third difference is the use of vignetting compensation. Vignetting compensation is sometimes necessary for mobile cameras which have a non uniform light distribution over the image (darker borders and a brighter center). A drawback of these improvements is the increased computational workload. A comparison between ARToolkit and ARToolkitplus could not be found, but there is one between ARToolkitplus and ARTag [9]. The previous mentioned paper describes the differences in false positive rate and inter-marker confusion. ARTag gets the best results on these tests.

The most suitable algorithm for porting to FPGA is the ARToolkit. It is an open source algorithm, has a lower execution time than ARLoc and can detect smaller markers better than ARLoc. The improvements of ARToolkit lead to better performance in terms of detection, but
not in terms of execution time. The computation intensity increases significant when using one of those improvements of ARToolkit. The improvements are not open source either.

2.2 Markerless detection

Next to AR via marker detection also markerless AR exists. Markerless feature detection can be separated in two categories. The first category is natural marker detection; the second one is natural feature detection. Natural marker detection traces a natural maker (not a placed artificial one) and extracts the information from that marker. Natural feature detection tracks all features in a scene and extracts the information from those features. In most natural feature detection algorithms, a sequence of images from the scene is needed to extract all information. Advantages of these kinds of detection are that no adaption of the environment is required upfront.

2.2.1 Natural Marker Detection

An example of Natural Marker Detection for creating AR is described in [17]. This method uses the human hand as natural marker (see Figure 2.2(a)). A downside of this method is that the marker is fixed and cannot be adapted. The marker (hand) must have a human skin color and the 5 fingertips must be visible to display the result correctly. It is not possible to implement another marker without major changes in the algorithm; the whole algorithm is created around this marker. In the case of the ARToolkit, it is relatively easy to add other markers to the algorithm. Natural Marker Detection can be seen as object detection with a fixed type of object, with information extracted from that object.

Although this kind of algorithms has the advantage of no adaption of the environment, the number of different markers which can be detected is limited. A marker detection algorithm can usually detect more than one different marker which makes the number of possible applications higher. Therefore a marker detection algorithm is chosen to be ported.

2.2.2 Natural Feature Detection

Another alternative is Natural Feature Detection (NFD). NFD does not use markers at all. NFD uses natural viewable differences (edges, corners). Most natural feature detection algorithms consist of two parts. The first part detects the natural features, commonly done via SIFT or SURF. An example of Natural Feature Detection with detected features is given in Figure 2.2(b). The second part of the algorithm maps the detected features to the discovered features of previous images. This is the tracking part of the algorithm. A well know and often used algorithm for NFD is Parallel Tracking And Mapping (PTAM, [16]). A drawback of this method is that the scale and absolute 3D pose information is lacking, which is necessary to create realistic augmented reality. This can be solved using stereo vision or moving the camera around and capturing the scene from different angles. Natural feature detection can be used for AR, but it is not so extensively used as marker detection nowadays. In the introduction paper of PTAM a small AR example is given, see Figure 2.2(c).

Natural Feature Detection algorithms are not chosen to be ported to the FPGA due to the requirement of keeping large data structures with features inside the FPGA. Such high memory requirements can lead to resource problems in the FPGA. The algorithm is also
more computational intensive than the marker detection and therefore ARToolkit is chosen as algorithm in this case study.

![Image](image.png)

(a) Natural Marker Detection  (b) Natural Feature Detection  (c) AR via Natural Feature detection

Figure 2.2: Non-artificial-marker Augmented Reality

### 2.3 Implementation of the ARToolkit on other platforms

Several attempts have been made to port the ARToolkit to other platforms. The first one, described in [21], ports the ARToolkit to a PocketPC. As described in the paper, the initial port was relatively straight forward. The original code is in pure C and was cross compiled to Windows CE. But the performance was really low (around 1 fps). An improvement was made through switching from emulated floating point to fixed point. This increased the performance to 5 fps. Similar porting problems occurred in [12], where a port of the ARToolkit to a mobile phone (running a Symbian OS) is described. A mobile phone can also use the emulated floating point, but this has a bad influence on the performance. With the step to fixed point there was a significant increase in the frame rate. Both ports are ports to a sequential processing platform and therefore no big changes had to be made to the algorithm itself.

To the best of my knowledge, the ARToolkit has never been ported to an FPGA. There are attempts to use an FPGA for AR, like [15], but the FPGA is only used as accelerator for a specific image processing function in this paper. The goal of this research is to create a complete vision pipeline in the FPGA. A complete vision pipeline plus limited visualization is implemented in [10] and the according case study in [19]. This framework, called ARCam, and the case study, a game called MarkerMatch, supports images up to 240 times 240 pixels and can apply some basic image processing functions, like edge detection, binarization and binary decoded marker recognition on those images. Though this framework looks promising at the first sight it has some downsides: The low image resolution will look nice on older hardware but current screens have at least a VGA size or even HD-ready or full HD size. Next to that is the current ARCam setup is not capable to really augment pictures with 3D objects or even extract the information to do this. The markers in the captured image can be recognized, and eventually a 2D overlay can be projected. The 2D overlay must be of one color, and is not rotated but just a fill of the marker with the color. The hard to extract information, like position, rotation and exact fitting lines, which is needed to augment an image with a 3d object is not extracted.

Within the EVA project, other vision algorithms have been ported to FPGA. In [11], a
Fast Focus on Structure (FFOS) application is mapped to an FPGA. The focus of the FFOS project was to achieve a frame rate of 1000 fps with a vision application and identify the bottlenecks around that application. Next to that, the vision pipeline was required to have a deterministic processing time because this application was used for control purposes. From the FFOS paper can be concluded that the exposure time and the image read out time are the bottlenecks in an ultra high frame rate visual servoing application. Expected is that the problems described in [11] will not appear in the porting of the marker detection algorithm. AR is used for humans and if they perceive a non-flickering video sequence of a reasonable quality, it is good enough. They do not have much benefit from the high frame rate as in the previous case study.

2.4 High Level Synthesis Tools

Although there are a lot of discussions in the High Level Synthesis domain, there are not many independent comparisons between HLS tools. Last year, Berkeley Design Technology Inc (BDTi) did an independent evaluation of two tools. In [5], AutoESL’s AutoPilot and Synopsys Synphony C Compiler are analyzed. Both tools are evaluated using two case studies. In terms of resource usage, the AutoESL tool performs somewhat better. According to the paper, AutoESL can create code which uses fewer resources than handwritten HDL code. The Synopsys tool is a bit worse; the produced code uses a bit more resources than handwritten code. In usability and documentation, both tools are comparable. In [6], an implementation of a sphere detector is made with AutoESL. Comparable results with the previously mentioned Berkeley paper are achieved. A noticeable conclusion from [6] is: The tool does not generate code which performs better than handwritten code. But it is easy to explore many different designs, so when enough time is spend on exploring different designs; a better design will be created. An HLS tool which is not certified by Berkeley, but also seems very promising and open source tool is the one introduced in [20]. In this introduction paper of the ROCCC compiler and toolchain, VHDL code is generated which has a higher clock frequency and a higher throughput than its handwritten counterparts. Resource usage is not discussed in this paper. Expected is that the higher clock frequency and throughput also have a downside in terms of extended resource usage.

Nowadays, more HLS tools are available. This research is limited to two tools. ROCCC is chosen because it is an open source tool and the results in its introduction paper are promising. In a later stage of the EVA project adaptations to this tool can be made, if necessary, because it is an open source tool. AutoESL is chosen because it is reviewed by BDTi as one of the best tools available at the moment. Another reason to choose this tool is the close integration with the Xilinx tool chain and AutoESL was available at the TUe. Some of the other HLS tools will evaluated in the Embedded Systems group on the Faculty of Electrical Engineering at the University of Technology in Eindhoven. A recent global evaluation of the available HLS tools at the moment, done in the Embedded Systems group, indicates that AutoESL seems to be one of the best tools.
Chapter 3

The marker detection algorithm

The ARToolkit consists of multiple algorithms. The focus of this project is on the marker detection part. The extraction of markers is done via a chain of sub-algorithms, called the vision pipeline. These sub-algorithms are responsible for specific parts of the algorithm. Before porting to a specific platform, it is important to understand the algorithm and analyze the complexity of this algorithm. After analysis, already an indication is gained which sub-algorithms are the most computational intensive and where optimization effort must be spend. To gain this information, the algorithm is tested and analyzed. This analysis is performed based on the code and on the documentation on the internet [1].

Section 3.1 contains the global overview of the whole vision pipeline. A detailed overview of every sub-algorithm is given in Section 3.2.

3.1 Overview of the processing pipeline

As mentioned before, there is some documentation provided together with the algorithm. This documentation consists of one flowchart of the algorithm, and a very brief indication of the working of the sub-algorithms. After analysis of the code a new flowchart is made. The new flowchart gives a clearer overview of the sub-algorithms and the data which has to be communicated between the sub-algorithms. In an FPGA, data communication is often an issue. A good indication of the amount of data, which has to be communicated, and the data-dependencies of the sub-algorithms is given in the new-flowchart. The flowchart also gives an indication of the task level parallelism which is available in the algorithm. The new flowchart is given in Figure 3.1. The vision pipeline is discussed in detail in the next section.

There are four input variables in this vision pipeline. One variable is the image; this image can vary every iteration of the pipeline. The rest of the input variables are loaded once at the start of the program. The camera parameters, the template patterns and the binarization threshold are defined upfront; if they must be changed, the program must be restarted.

At the bottom of Figure 3.1 can be seen that the functions after the marker detection algorithm receive different variables. Lines and vertices describe the rotation and exact position of the marker. The confidence value, the ID of the marker and the direction describe which marker it is. The position and area give a global overview of the size and the position inside the detected frame. The Number of Markers indicates how many markers are discovered, and thus, how many items the previous mentioned variables contain.
3.2 Pipeline stages explained in detail

The different pipeline stages from the vision algorithm (Figure 3.1) are described in this section. This is done per pipeline stage. The algorithm can be globally divided into two parts. The first part extracts information about where the marker(s) is (are) located in the image. It is discussed in subsection 3.2.1 (binarization) to subsection 3.2.5 (remove components in components). The second part extracts the real information of the marker from the original image. This is done using the information of the first part. The second part is discussed in subsection 3.2.6 (pattern extraction) to subsection 3.2.9 (sub-pixel corner detection). Both parts are illustrated with colors in Figure 3.1. Blue indicates the first part and red indicates the second part of the vision pipeline.

3.2.1 Binarization

The first function of the algorithm, the binarization (illustrated in Figure 3.2(b)), is fairly simple. This is a pixel to pixel operation in which the value of the pixel is compared with a fixed threshold. Binarization is applied on the whole image. The pixels are saved in 8 bit RGB format. To compare them, the red, green and blue value are summed together and
compared with the threshold. The exact mathematical description of this sub-algorithm is given below.

\[
\text{Binarized\_value} = \begin{cases} 
1 & \text{if } (R + G + B) \leq \text{threshold} \\
0 & \text{otherwise}
\end{cases}
\]

### 3.2.2 Labeling

The second function in the algorithm is labeling. The purpose of this function is the creation of an output image in which connected components are labeled with the same number. Connected components (CC) are pixels which are binarized with a 1 and lie next to each other in the image (8-way connected). The result of this function is displayed in Figure 3.2(c), where every color implies one component. The labeling function determines to which connected component a pixel belongs. The algorithm works as follows: if the pixel has a neighbor which already belongs to a connected component, the pixel belongs to the same connected component. If not, a new connected component is created. The whole image is processed from left to right and from top to bottom. It is possible that two connected components, which first seem to be two separate ones, appear to be connected. If a merge between two components is discovered this information is stored in an auxiliary array, called the list of connected components (ListofCC). This ListofCC is not a list with pairs but an one dimensional array. A value at a certain index corresponds to the label of the pixel. E.g. A pixel is labeled with a 5 in the labeled image. If a function needs the label of that pixel, it first checks in the ListofCC the actual label. If the number on index 5 is another number, this other number is the label of the pixel. Component 5 is merged with another one during the labeling function and the ListofCC was updated. The original sequential implementation consist of two runs through the image, but here is no second run through the image. Because there is no second run through the image, a non perfect labeled image is produced. E.g. a connected component can be partially labeled with one label and partially with another label. Although this image is not perfect, together with the produced ListofCC enough information is available to proceed with the next functions. Not only the components which are connected are stored in an auxiliary array, also the area of a component, the position of the center, the minimum and maximum \(x\) and \(y\) values and the total number of different connected components is stored. The pseudo code of this algorithm is given in Appendix A, Section A.1

![Figure 3.2: First sub-algorithms of the vision pipeline](image-url)
3.2.3 Contour detection

The next step is contour detection. For all connected components, the contours are detected. The result of this function is visualized in Figure 3.4(a). During the previous function (labeling), the minimal $y$ and minimal $x$ value of each component is stored. This information is used at the start of the contour detection algorithm. The contour detection algorithm starts following the contour at the most upper left pixel of the contour (extracted from the minimal $y$ and minimal $x$ value) and proceeds over the contour in a clockwise direction. Pixels which belong to the contour are stored in an array and the algorithm terminates if the starting point is reached again. If the detection is finished, the detected coordinates are reordered. The point which is at the furthest Euclidian distance from the starting point will be listed first. The rest of the contour coordinates follows in clockwise order. Every connected component gets a separate list with its contour coordinates. To reorder the coordinates, an extra iteration through the whole contour list is performed. The pseudo code of this algorithm is given in Appendix A, Section A.2

3.2.4 Check square

Check square removes all objects which are not squares or rectangles. This can be seen in Figure 3.4(b). Depending on the viewing angle, squares and rectangles could be projected in another form on the image (e.g. a trapezoid from). This deformation of squares and rectangles is taken into account in this algorithm. Next to elimination of objects, the vertices of the discovered squares and rectangles are produced (visible as the red circles in Figure 3.4(b)). The algorithm takes the first point (start point) from the contour and determines the point on the contour which is the furthest from the start point (Euclidian distance). This point is called the endpoint. A straight line is calculated between the start point and the endpoint. The algorithm analyzes if the contour does not deviate too much from the calculated line. If it deviates too much, the point which is at the largest Euclidian distance from the calculated line becomes the new endpoint. If the deviation is below a certain threshold, the current endpoint becomes a start point and the next endpoint is processed by the algorithm. The threshold is depending on the area of the component. This area is saved during the labeling function and used here.

An example is given in Figure 3.3. The black lines indicate the discovered contour in the previous section (consisting of many points) and three points are highlighted in this contour, indicated with the numbers. The first iteration of the check square algorithm is given in Figure 3.3(a). Dot 1 is the start point and dot 3 is discovered as endpoint in the first iteration over de contour. The horizontal red dashed line is the estimated/calculated line between start point and endpoint. The algorithm goes over the black contour and discovers that dot 2 is at the furthest Euclidian distance from the calculated line and its distance is above the threshold. Dot 2 becomes the new endpoint and a new line is calculated (visible as the red line in Figure 3.3(b)). In the second iteration (Figure 3.3(b)), the furthest point from calculated line is below the threshold and the algorithm continues with the next iteration. In the third iteration, the old endpoint from iteration two becomes the start point and the next (already discovered in iteration one) endpoint becomes the current endpoint. The same algorithm is applied again and is shown in Figure 3.3(c).

When the original endpoint (discovered in iteration one) is reached, the algorithm continues with the second part of the contour array. In the example from Figure 3.3 this means 3
is the start point and 1 is the endpoint and the algorithm will start processing the gray part of the contour. The procedure is repeated and vertexes (start and endpoints) are discovered. If the check square algorithm produces one extra vertex between the first half (original start point - original endpoint) and one extra vertex in the second half of the contour array (from the original endpoint - original start point), the connected component is a square or rectangle. If two extra vertices are needed between the two original points on one side of the contour and no extra vertex is needed in the other half, the object has also the correct shape. Any other variations are removed because they have the wrong shape.

![Figure 3.3: 3 iterations of the check square algorithm](image)

3.2.5 Remove components in components

A marker can contain squares in it as identifiers (like the i in the example in Figure 3.2(a)). These inner squares are still in the frame, but must be removed. This is done in the remove components in components step. For every combination of two possible components a simple equation is performed. The difference in $x$ plus the difference in $y$ (both squared) between the centers of the components is calculated. If this difference is smaller than $\frac{1}{4}$ of the area of the largest component, the small component is removed. If the result is larger than $\frac{1}{4}$ of the area of the largest component, both components remain. The result is visible in Figure 3.5(a).
The factor of $\frac{1}{4}$ is based on the assumption which is illustrated in Figure 3.5(b). Markers are always constructed like the maker in Figure 3.5(b). The maximum distance between two centers of components is illustrated with the line $A$ in the figure. The maximum length of $A$ is $(\frac{N}{4})^2 + (\frac{N}{4})^2 = \frac{N^2}{8}$. All components where the center lies in the range below $\frac{N^2}{8}$ from the center of the large component will be removed. This means all components inside the white part of the marker will be removed. The factor is a bit to large, this is to make sure that the algorithm will work for rotated markers as well (in all 3 dimensions).

(a) Result of the remove components in components algorithm

(b) Illustration of the ratio between border and inner part of the marker. $A$ indicates the maximal distance between the center of the marker and one of its components inside

Figure 3.5: Remove components in components algorithm

After remove components in components every remaining component is treated as marker. This marker is described in a contour list, has a position and a certain area and a global estimation of the corners (vertices). In the following sub-algorithms the pattern of every marker is extracted and matched with the template patterns. This pattern extraction is done on the original image. Next to the processing on the inner part of the marker, the exact fitting lines and the corners from the border of the marker are detected.

3.2.6 Pattern extraction

Pattern extraction extracts, as the name describes, the patterns inside the markers. Pattern extraction is done on the original image, using the information extracted with the previous functions. Before extraction, a transformation (rotation) matrix must be made. When this transformation matrix is applied on the marker, the marker will be rotated in such a way that it is viewed directly from the front aligned with the borders of the image. An example given in Figure 3.6(a). However, the marker can be rotated at 90, 180 or 270 degrees. This transformation matrix is calculated via a Direct Linear Transform algorithm.

After calculation of the transformation matrix the real pattern extraction starts. The area between the black borders of the marker is divided into 16 by 16 squares (bins). The
average of the pixels inside these squares is calculated. A maximum of 16 (4 × 4) pixels per bin is taken to calculate the average and those 16 pixels are taken equally distanced over the bin. The 256 average values together make the extracted pattern. The division into squares is illustrated in Figure 3.6(b). The pattern is extracted for all three colors, so three extracted patterns are created. If black and white markers are used, the patterns per color have not that much deviation. When using colored images, the different patterns per color will deviate much more.

(a) Transformation matrix applied on the whole marker  
(b) Division into bins from the inner part of the marker  

Figure 3.6: Pattern extraction

### 3.2.7 Template matching

The extracted pattern from the previous function is matched/compared with a reference pattern. The reference patterns are loaded at the start of the program. They are not only stored in the three different colors, but also in all four possible rotations (every time 90 degrees rotated). That makes twelve patterns per marker. The extracted pattern from the previous function is matched with the reference ones (per color, so four times three matches per marker). To make the matching more light invariant, this is done in the following way. First the average from all extracted pattern bins is calculated. Then the values of the bins are recalculated with the difference between them and the average (can be positive or negative). The reference patterns are also stored in this normalized way. If the contrast is maintained under changing lighting conditions, the patterns will still match. Every bin from the extracted pattern is now multiplied with the same bin in the reference image, and they are summed together. The total sum is divided by the total sum of all bins in the reference image and then divided again by the total sum in the extracted image. This leads to a (confidence) value between 1 and −1. The closer the confidence value is to 1, the better the match between the reference and the extracted pattern is. The reference pattern which has the highest confidence value (and is above 0, 5) is chosen as output. Next to the confidence value, the direction and the pattern number are outputted to the following function.
3.2.8 Line contour estimation

Line contour estimation performs an estimation on the outer contours of the marker. As input it uses the contour coordinates and the vertices. The contour coordinates are recalculated (corrected) with the camera distortion values. Between every two consecutive vertices a line is estimated. This is done by calculating the mean and eigenvector of the recalculated contour pixels. From these calculated parameters the formula for the line is constructed (in the format $Ax + By + C = 0$). The Eigenvector determines the $A$ and $B$(negated) in previous mentioned function. $C$ is constructed by multiplying the mean of the $x$ coordinates with the previous constructed $A$. This is summed with the multiplication of the mean of the $y$ coordinates with the previous constructed $B$ value. Line contour estimation function looks a lot like the line estimation in check square, but this one is executed on the corrected contour values. This functions is more computational intensive than the line estimation in check square algorithm, therefore it is only executed for the final markers.

3.2.9 Sub-pixel corner detection

From the lines, calculated in the previous function, the vertices are estimated again. This is done in a few, relatively simple calculations which produce the intersections between the lines.
Chapter 4

Requirements and Testing

With an in-depth understanding of the algorithm, the actual porting can be started. But before that, there are some other parameters which have to be determined. This chapter defines the output accuracy, the frame rate, the system onto which the algorithm has to be ported and the test sequence which checks the produced outputs for correctness and accuracy. Section 4.1 contains the definition of the output variables. The next section, Section 4.2 describes the definition of the frame rate. The platforms are described in Section 4.3 and finally the test sequence in given is Section 4.4.

4.1 Accuracy definition of the output variables

The marker detection algorithm outputs floating point variables to the next functions in the ARToolkit. Floating points are used internal in the marker detection algorithm as well. For expected performance improvement in further implementations, double precision floating point variables of the algorithm should be converted to single precision floating point variables or even into fixed point variables as far as possible. Fixed point variables consist of an integer part and a fraction part. To determine the integer part (number of bits in front of the radix point) the range of the floating point variable should be known. Also the sign of the variable must be known. The fraction part (number of bits after the radix point), determines the precision of the fixed point number. For single precision floating point variables this is somewhat different. A single precision floating point number has a fixed number of bits (32). 23 bits determine the accuracy and is called the significant. This significant is scaled with an 8 bits exponent (one of the 8 bits is for the sign of the exponent). The final bit is to determine the sign of a floating point number. Depending on the range of the integer value and the precision of the significant, it can be said a single precision floating point is accurate enough.

The visualization functions, which come after the marker detection, in the ARToolkit are not in the scope of this research, therefore an approximation of the accuracy of the output variables of the marker detection algorithm is made. The number of required bits is displayed in Table 4.1. The accuracy of output variables is based on the following assumption. Assume this port is meant to work with a Head Mounted Display (HMD). A HMD is the most common way to display augmented reality information nowadays. Most HMDs have a camera attached to it and have a see-through screen in front of both eyes which make them very suitable for this application. The viewer can see the real world through the screens; the augmented
information is projected on the screen and covers parts of the real world. The screens are fixed at a position of 3 cm from the eyes (like normal glasses) and have a size of 4 by 6 cm. This leads to a viewing angle of approximately 68° by 90°. According to chapter 3 from [7], the observer would not have any quality loss if the resolution is 1943 by 2571 pixels, taking into account the distance and the viewing angle.

Although these kinds of screens are currently (2011) not available, this size is assumed to determine the accuracy. The accuracy of the lines output variable is defined as follows; assume a line has to be displayed and this line covers the whole length of the screen. The line is displayed with a rotation of one pixel (with respect to perfect alignment with the borders of the screen) and given in the format $Ax + By + C = 0$. In the worst case, the direction has to be determined by one variable ($A$ or $B$), the other two variables are assumed to be zero or one. To display the line correctly (the jump to the next pixel must be exactly in the middle), the minimal accuracy must be $\frac{1}{2571}$ and $\frac{1}{1943}$ to display this correctly. So for these variables, we need an accuracy of 12 bits (the largest one is taken). The maximum and minimum in values of these parameters (which determine the integer parts) are the maximum dimensions of the screen (also 12 bits).

The position variable is pixel values. So an accuracy of a pixel is enough in this case. This means no fraction bits are required here. The maximum and minimum dimensions of the screen determine the integer part of the position variables.

The accuracy of vertices is somewhat more difficult to compute. In principle this is also a pixel value. But since the follow up functions use the vertices when drawing objects on the screen and little deviations could already lead to faulty rendering, therefore a higher accuracy is required for this variable. To guarantee a good quality the same accuracy is assumed as with the lines. With this accuracy, no further errors are introduced in the follow up functions.

For the confidence value the following approximation is done. The inner part of the maker is sampled into 256 bins during execution of the algorithm. Each bin contains the average of 16 samples. Assume the samples can be black or white, the bin can have maximum of 16 different values (order of the samples doesn’t matter). This makes a maximum of 4096 different possibilities (order of the bins does not matter, due to the rotated matching). If the translation from sampled pattern and template pattern to confidence value was perfectly linear, 12 bits (which imply an accuracy of $\frac{1}{4096}$) was enough to guarantee a correct confidence value. But the transfer from patterns to confidence value is not perfectly linear and more bits are required to achieve this. On the other hand, usually markers differ more than one sample in one bin. So, the accuracy of twelve bits is assumed to be sufficient.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Signed bit</th>
<th>Integer bits</th>
<th>Fraction bits</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidence</td>
<td>yes</td>
<td>1</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>Position</td>
<td>no</td>
<td>12</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>Vertex</td>
<td>no</td>
<td>12</td>
<td>12</td>
<td>24</td>
</tr>
<tr>
<td>Lines</td>
<td>yes</td>
<td>12</td>
<td>12</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 4.1: Required bits for fixed point outputs

The accuracies mentioned in Table 4.1 cannot always be covered with single precision floating point variables. Some variables need more than 23 bits in the significant (Lines and Vertex). But for this setup, floats are used. The only case when a floating point is not
sufficient is when the screen resolution of the HMD is above 2048 pixels (per dimension) which is above a full-HD resolution. HMDs above this resolution are currently not available, so for this test setup, floats are assumed to be sufficient. The exponent can easily be covered within the 8 bits which are available (maximal movement is 12 places which can be covered in 4 bits).

4.2 Frame rate

Another variable which has to be defined is the frame rate. The frame rate basically defines the time budget of the algorithm. For this project, the frame rate is defined at 30Hz. Although only frame rates above 50Hz are flicker free, this is not chosen as requirement. The execution time for the algorithm at 50Hz will be so limited; the expectation is that this will not be reachable in this hardware setup. This expectation is based on the execution time of the marker detection algorithm on CPU. A modern Intel Core 2 Duo CPU (T9300) can achieve a frame rate of approximately 128 fps on the marker detection algorithm. The processor runs at 2.5GHz with only one thread. Assume there is an embedded platform which has the same architecture/properties as the Intel CPU (a bit unrealistic, but this is an approximation) and this embedded platform has a frequency between 100MHz and 700MHz (which is common for an embedded system). The frame rate will be in the range of 6 fps to 35 fps. Embedded platforms are usually not so sophisticated as an Intel processor, so the frame rate is determined somewhat lower than 35fps. A good thing to keep in mind is that this port is meant for a HMD. A human head with a HMD attached to it will not move at a very high speed, so a lower frame rate will hardly be noticeable. Therefore this requirement is set at 30Hz.

4.3 Platform definition

As mentioned in Chapter 1 the focus of the EVA project is on FPGAs as platform for vision applications. This focus defines the platform for this project. A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by the customer or designer after manufacturing. The FPGA configuration is specified using a hardware description language, which can be VHDL or Verilog. FPGAs can be used to implement all kinds of logical functions. FPGAs contain programmable logic components called slices, and a hierarchy of reconfigurable interconnects that allow the blocks to be wired together. Slices can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the slices also include memory elements (FlipFlops). The available FPGA at the TUe is a VIRTEX 6 (XC6VLX240T version) from Xilinx. The VIRTEX 6 contains, next to the before mentioned basic elements, some more advanced Digital Signal Processors (DSPs) and Block Random Access Memories (BRAMs). DSPs can be used to execute more complex computations. This can also be done using basic slices, but a lot of slices must be used for this (so more area is used) and this will probably cost more cycles. BRAMs can be used to store large data structures, which is also very costly in slices. This VIRTEX 6 is mounted on an evaluation board. The evaluation board (ML605) is shown in Figure 4.1. In Figure 4.1 peripherals are connected to the FPGA, e.g. DDR3 RAM, UART, Leds, Push buttons, LCD screen, a Compact Flash card and a PCI bridge are attached. Not all peripherals are needed for this algorithm; the ones who are needed and attached are discussed in the following chapters.
If, in a later stadium, the system will be used in a real application, this VIRTEX 6 board is a very expensive solution. A smaller FPGA, like a SPARTAN 6 on a custom board with the necessary peripherals, is a more likely choice to be used as embedded system. This will have impact the coming solutions described in this report. Resources in a SPARTAN are scarcer than in a VIRTEX and more effort should be spend on optimizing when using such smaller system.

Figure 4.1: ML605 evaluation board with VIRTEX 6 and attached peripherals.

Next to the FPGA platform, another platform is defined. The FPGA is a completely different architecture compared to the CPU where the algorithm is currently running. To enable a fair comparison, a processor must be found which has a comparable power usage with the FPGA but has the same sort of architecture as a CPU. A suitable and available low power device at the TUe is an IGEPv2 board. The board contains an OMAP3530 chip from Texas Instruments. The OMAP3530 contains an ARM A8, a DSP and a small GPU. The ARM processor runs at 720MHz, the other two processors run at 520MHz. The ARM core has two issue slots, can use NEON (SIMD) instructions and can perform out of order execution. This embedded board is comparable with an FPGA in terms of power usage (4W for the ARM, 6W for an implemented FPGA design). The used IGEPv2 board is visible in Figure 4.2. The algorithm is also ported to this device and this is used as reference in the remaining part of the report.

4.4 Testing

During porting double precision floating point variables will be replaced as much as possible with single precision floating points or even with fixed point variables (mentioned in Section 4.1). The accuracy of output variables was also defined in Section 4.1. To make sure
that the replacement of the internal double precision variables does not influence the output variables too much (more than the accuracy definition allows) a test method is developed. For this test method, a large set of test images is created. Examples images of this test set are given in Figure 4.3. The test set consists of 159 images with one marker viewed from different angles under varying lighting conditions. Next to that, a few test images with multiple markers are added to the test set. For every image, a reference file is generated. This reference file contains the output values in double precision floating point format. If an internal variable is changed in the algorithm, the output values are generated with the changed algorithm and compared to the reference outputs (comparison is done in double precision format). If the result of the comparison stays within the defined accuracy (see for the definition of the output variables Section 4.1 of this chapter), the replacement is valid. Although this test sequence is not complete (some viewing angles can be missed, or other cases can be overlooked), it gives a strong indication of correct functioning of the algorithm.

Figure 4.3: A few examples from the set of test images.
Chapter 5

ARM implementation

The first implementation of the algorithm in this project is on the reference ARM board. Chosen is to start with this reference implementation to get a feeling about what to expect and have a guideline to work to. The implementation is done directly on the board (OS and compiler are running on the arm processor) using a host computer.

This chapter consist of three sections. Section 5.1 describes the initial implementation of the marker detection algorithm on the ARM processor and the changes which have been made on the algorithm. The next section, Section 5.2, describes the optimizations which are done to improve the performance of the algorithm. The last section, Section 5.3, evaluates the current performance and gives some directives to improve this performance.

5.1 Initial implementation and algorithmic changes

The initial implementation was a relatively easy implementation. The original code is taken as starting point. Internal double precision floating point variables are replaced as much as possible by single precision floating point and the results are tested using the described test sequence in Section 4.4. A few double precision floating point variables remain; they are located in the pattern extraction and in the line contour estimation. Calculations on these floating point variables are done via software emulation (default of the compiler). Software emulated floating point calculations are very costly (in terms of execution time) because software emulation produces a lot of instructions which have to be executed.

Next to the double precision floating point conversion some other algorithmic changes are made as well. In the check square function, a recursive function is changed into a loop. The recursive function determines very precisely how many vertices where needed for a fitting curve, where results above two were discarded. The replacement loop has a fixed number of iterations and is faster than the original one on round shapes. The new algorithm is equally fast on squared objects. Some other minor changes in the labeling function are performed as well, some loops are merged to get a better performance and smaller buffers.

The performance on the ARM after the algorithmic changes is given in Table 5.1.

5.2 Optimizations

Further optimizations of the algorithm are done by the compiler. The ARM processor has support for SIMD (Single Instruction Multiple Data) instructions. These SIMD instructions
Table 5.1: Absolute execution time of the marker detection algorithm on the ARM processor

<table>
<thead>
<tr>
<th>Step</th>
<th>Initial implementation (us)</th>
<th>Final implementation (us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binarization</td>
<td>13703</td>
<td>13702</td>
</tr>
<tr>
<td>Labeling</td>
<td>26050</td>
<td>26031</td>
</tr>
<tr>
<td>Contour detection</td>
<td>1556</td>
<td>1526</td>
</tr>
<tr>
<td>Check square</td>
<td>732</td>
<td>702</td>
</tr>
<tr>
<td>Remove components</td>
<td>153</td>
<td>92</td>
</tr>
<tr>
<td>Pattern extraction</td>
<td>8789</td>
<td>5127</td>
</tr>
<tr>
<td>Template matching</td>
<td>519</td>
<td>275</td>
</tr>
<tr>
<td>Line contour estimation</td>
<td>10864</td>
<td>3540</td>
</tr>
<tr>
<td>Sub-pixel-corner detection</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td><strong>Total (us)</strong></td>
<td><strong>62397</strong></td>
<td><strong>51026</strong></td>
</tr>
<tr>
<td><strong>Fps (approx)</strong></td>
<td><strong>16.0</strong></td>
<td><strong>19.6</strong></td>
</tr>
</tbody>
</table>

There are two different floating point units in the ARM processor. One is a non pipelined IEEE-754 compliant floating point accelerator (called the lite-VFPU, Vector Floating Point Unit) and the other one is the floating point unit in the NEON accelerator. This NEON floating point unit is a IEEE-754 compatible unit and can preform multiply-add instructions. The rest of the floating point arithmetic is done via software emulation. The difference between fully compliant IEEE-754 processing cores and compatible IEEE-754 processing cores are mostly found in the border cases (rounding, wrap around) and exceptions (like division by 0). Although those cases probably will not appear in this algorithm, the test sequence is performed every time to check the result on correctness. Using the first option (the lite-VFPU, enabled via a compiler directive) did not improve the execution time. This is probably due to the lack of pipelining in the execution unit. The second option, using the NEON floating point unit, resulted in a speedup. The compiler flag which enables these instructions is set in the compiler. A check in the produced assembly code indicates that multiply-add instructions on the NEON are used. Double precision floating point arithmetic and instructions which are not multiply-adds will be executed via software emulation.

Other compiler optimization are tried as well, e.g. automatic loop unrolling, but give no performance increase. The results of the final implementation on the ARM core are given in Table 5.1. A visualization of the division of the total execution time is given Figure 5.1.

### 5.3 Evaluation and improvements

A better compiler or manual insertions of the SIMD instructions can decrease the execution time. Though, the expected influence of these instructions is limited. The opportunities
Figure 5.1: Relative execution time of the final implementation of marker detection algorithm on the ARM processor listed per sub-algorithm.

to insert SIMD instructions can only be found in the first functions because this is these functions contain parallel operations without dependencies between data entries. Assume the binarization function can be improved with a factor of 4 and the labeling function also has some opportunities to perform parallel updating the auxiliary arrays (assume a speedup factor of 1.2). The resulting frame rate will be around 27 fps.

The performance from the marker detection algorithm on an ARM core is not a main goal of this project, so no further steps are performed to optimize this code further. The performance given in Figure 5.1 is used as a global reference through the rest of the thesis.
Chapter 6

Manual FPGA implementation

The first attempt to run the ARToolkit marker detection on an FPGA is described in this chapter. The porting is done by hand, everything is written in VHDL or C and common IP blocks from Xilinx are used. Before starting porting, the floating point variables in the vision pipeline are analyzed. This is done in Section 6.1. The real porting is started in Section 6.2, which contains the initial port from the algorithm to FPGA. Section 6.3 describes the different accelerators which decrease the execution time of the initial port. Some of these accelerators can be combined (Section 6.4) and improved (Section 6.5).

6.1 Floating Point or Fixed Point

A common practice during mapping algorithms to FPGA is changing the floating point variables into fixed point variables. This is often done to improve the performance of the implementation. Floating point cores are expensive in used number of cycles and expensive in used resources of the FPGA. Originally, this change was also planned during the design of the FPGA implementation. Before the replacement, the integer and fraction part of all floating point variables must be determined. Analysis of the integer and fractional part for the fixed point variables is done with a Matlab simulation using the fixed point toolbox. The whole algorithm is implemented in Matlab and all internal floating point variables are replaced by fixed point variables. The number of integer bits and fraction bits is reduced one by one. Every time a reduction is done, the whole test set is simulated to verify the impact of the reduction on the output. A long list with all replaced fixed point variables is produced, this list is given in Appendix B. There are some notable results in this list. Some of the variables need more bits than the significant of the original floating point implementation. These variables (especially in the pattern extraction and the line contour estimation) make use of the dynamic range of the floating point variables. E.g. in the pattern extraction, values appear which lie in the range from $10^{23}$ to $10^{-6}$. This can be covered by a single precision floating point but also by a fixed point of 92 bits. If this variable is covered with a single precision variable and it has a higher integer value, it will lose some precision, but apparently this is not a problem in this algorithm. When looking at the exponent of the variables which have to be replaced, the absolute value of all variables would fit in the exponent of a single precision floating point variable. But the significant of a single precision floating point has not always enough bits to guarantee the output precision. So if a larger significant is needed, a double precision variable is selected.
A multiplication of a double precision variable costs on average 9 DSP units of the FPGA, a multiplication of two 46 bit fixed point numbers costs on average the same number of DSP units. The LUT and FlipFlop resource requirements are somewhat higher at the floating point core, but it differs not that much. This is verified using two cores generated with the Core Generator from Xilinx. The number of cycles the cores use does not deviate that much either, although the initiation interval (when a new data sample can be provided to the inputs) is shorter (in general, depending on implementation) in the fixed point multiplier.

Due to the large number of floating point variables in the algorithm (almost half the number of variables is floating point), the large number of required bits per variable for fixed point variables (which implies very large arithmetic units) and the limited extra amount of used resources and used clock cycles for floating point cores, the floating points are left in the implementation. Highly optimized cores are used to process these floating point variables.

6.2 Initial Implementation

The first implementation of the ARToolkit Marker detection algorithm on FPGA is done in a Microblaze. A Microblaze is a soft-core processor developed by Xilinx and can be configured/tuned in the Xilinx software. Different peripherals can be attached to the soft-core. For this implementation the DDR3-RAM, a UART peripheral, and the compact flash card are attached to the Microblaze. The compact flash card is a persistent storage medium which contains the test images. In future work a camera can be attached to the board. In that case, the processor should not read from the flash card but use images from the video stream. This is not done in the current implementation because this is not the focus of this project. The attached UART is for control and feedback purposes. The Microblaze can be configured with a certain amount of on-chip memory. The memory needed for for this algorithm is at least 600kB. The configured maximum amount of this on-chip memory (256kB) is not enough, therefore the DDR3 memory is used to extend this. The DDR3 memory is only used for the data storage, the instructions fit into the on-chip memory of the Microblaze itself. To improve the performance of the Microblaze, the cache between the processor and the DDR3 is enabled. The processor can be tuned as well. Support for floating point arithmetic and integer divisions is enabled. Also a barrel-shifter is enabled, this shifter is often used to speed up type conversions (e.g. from float to double) or software emulated (double precision) floating point arithmetic.

The same code as used for the ARM core is used for the Microblaze. Some extra modifications are done here, there are C functions which are not supported or inefficiently implemented in the libraries of the Microblaze and are changed therefore. E.g. printf is replaced by Xilinxs’ own xil_printf function. The testing is done with the in Section 4.4 described test sequence. Although no floating point variables are changed, the testing still remains important. The floating point core in the Microblaze is also a compatible IEEE-754 core, like the NEON core in the ARM processor.

The obtained results are given in Table 6.1. The Microblaze implementation is slower than the same algorithm on the ARM board. Although the frequency differs more than a factor of 7 (720 MHz for the ARM, 100 MHz for the Microblaze) and the ARM has two issue slots (which makes a theoretical difference of 14 times), the Microblaze implementation is only a factor of 4 slower approximately. There are two reasons for this. The first one is the difference in floating point cores. The Microblaze has a dedicated floating point unit which executes
all floating point arithmetic. The ARM uses partially software emulation and partially the NEON core for calculating floating points. A Microblaze configured without a floating point core (so it must use software emulation) resulted in a difference of approximately 60 to 90 times in the pattern extraction and line contour estimation function compared to the ARM implementation. The other functions had approximately the same difference factor. This is more than the expected 14 times, the use of the NEON unit is an advantage of the ARM here. A Microblaze configured without limited floating point core support (add, subtract, multiply and division) resulted in a difference factor of 11 for the pattern extraction and line contour estimation. This is somewhat less than the expected 14 times. The floating point core in the Microblaze supports more functions that the NEON floating point core, this is the cause of the smaller difference factor. The second reason is the difference in cache size. The Microblaze has a larger cache (64kB) than the ARM processor (16kB). A test with a Microblaze configured with a 16kB cache resulted 22% slower execution compared to the Microblaze with a 64kB cache, where the labeling suffers the most from the smaller cache. Compared to the ARM processor, the difference factor is approximately 5 times instead of 4 times.

In Figure 6.1 and Figure 5.1 can be seen that the binarization and the labeling function are the most time consuming functions in both implementations. The pattern extraction and line contour estimation follow thereafter. The division between the different architectures is not exactly the same, but it can be said that most time is spend on the same functions.

The Microblaze uses a lot of resources, shown in Table 6.4. Although these numbers look pretty large, the resource usage of the Microblaze stay within 4 percent of the VIRTEX 6 FPGA (except the BRAMs, 21 percent of them is used).

<table>
<thead>
<tr>
<th>Function</th>
<th>ARM (us)</th>
<th>MB (us)</th>
<th>Difference factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binarization</td>
<td>13702</td>
<td>61819</td>
<td>4.51</td>
</tr>
<tr>
<td>Labeling</td>
<td>26031</td>
<td>72647</td>
<td>2.79</td>
</tr>
<tr>
<td>Contour detection</td>
<td>1526</td>
<td>6124</td>
<td>4.01</td>
</tr>
<tr>
<td>Check square</td>
<td>702</td>
<td>5871</td>
<td>8.36</td>
</tr>
<tr>
<td>Remove components</td>
<td>92</td>
<td>277</td>
<td>3.01</td>
</tr>
<tr>
<td>Pattern extraction</td>
<td>5127</td>
<td>28482</td>
<td>5.56</td>
</tr>
<tr>
<td>Template matching</td>
<td>275</td>
<td>997</td>
<td>3.63</td>
</tr>
<tr>
<td>Line contour estimation</td>
<td>3540</td>
<td>15525</td>
<td>4.39</td>
</tr>
<tr>
<td>Sub-pixel-corner detection</td>
<td>31</td>
<td>10</td>
<td>0.32</td>
</tr>
<tr>
<td><strong>Total (us)</strong></td>
<td><strong>51026</strong></td>
<td><strong>191752</strong></td>
<td><strong>3.76</strong></td>
</tr>
<tr>
<td><strong>Fps (approx)</strong></td>
<td><strong>19.6</strong></td>
<td><strong>5.2</strong></td>
<td><strong>3.76</strong></td>
</tr>
</tbody>
</table>

Table 6.1: Comparison between the Arm implementation and the Microblaze version

### 6.3 Hardware accelerators

The initial implementation is tuned and optimized as much as possible, but it can never reach the same performance as the ARM processor in its current form. To reach a higher frame rate, hardware accelerators are created. These accelerators are written in VHDL and connected to
Figure 6.1: Relative execution time of the initial Microblaze implementation listed per sub-algorithm

the Microblaze via Fast Simplex Links (FSLs). This link contains a First In First Out (FIFO) buffer and some control logic. The link operates at 100 MHz, can transfer 4 bytes per cycle and has a latency of 1 cycle. This gives a theoretical bandwidth of 400 MB/s. There are two links per accelerator. One link from the Microblaze to the accelerator for putting the data in the accelerator, and one link from the accelerator to the Microblaze to send the data back. This is illustrated in Figure 6.2.

Accelerators can be made in any form. It is possible to perform multiple operations in parallel per clock cycle. Sometimes multiple operations can be performed in series in one clock cycle. So there is a large degree of freedom while designing these accelerators. Different levels of parallelism should be exploited in accelerators, due to the relative low clock frequency of the FPGA. If the algorithm allows (i.e. no data dependencies are violated in the algorithm), it is possible to create a large number of parallel units to perform the computations. Of course this has a certain cost in area, so a tradeoff must be made between execution time (inherent to the exploited parallelism) and used resources on the FPGA.

6.3.1 Binarization

One of the most time consuming functions is binarization. The first hardware accelerator is created to decrease the execution time of this function. As described in subsection 3.2.1, the binarization of a pixel consists of multiple steps. The red, green and blue values of a pixel are summed together and the resulting value is compared with a fixed threshold. The result is one bit (equal/below or above the threshold). Since the FSL link is a 32 bit wide link, the RGB values (8 bit per value) of the pixels are packed so that the link is fully utilized. This means that 3 words, which are send over the FSL link, contain 4 pixel RGB values (illustrated in Figure 6.3(a)). This packaging is done when reading the image from the Compact Flash (CF) card. In future implementation the CF card will probably be replaced by a camera link, but also a camera link can be wired in such a way that it can produce packaged pixels.
The inner working of the accelerator is illustrated in Figure 6.3(b). The accelerator must buffer some values and sometimes it can calculate two binarization values at the same time. For the return link the same trick as with the input is applied. 32 1-bit values are grouped into one word and are send over the FSL link.

The VHDL implementation of this accelerator contains a finite state machine (FSM). This state machine consists of 25 states. The first 24 states receive an input from the FSL link and process this input immediately. Some of these 24 binarize two pixels in one cycle (as mentioned before). The last state outputs the 32 binarized values and returns to the first state. If an input is available, the output state will take this input as well and the very first state will be skipped.

This implementation has only one cycle latency (processing 24 inputs takes 25 cycles), plus the cycles which are lost in the transfer via the FSL (one cycle for both directions). If the Microblaze provides the data in the right order (e.g. first 4 times 24 inputs), the latency of the accelerator and the latency of the return FSL link can be hidden. The throughput of this accelerator can be equal to the throughput of the FSL link. The Microblaze cannot provide the data fast enough to fully utilize the FSL link. This is because the Microblaze first has to fetch a pixel value from cache or DDR3 and then copy it to the FSL link. So the
Microblaze can provide an element to the FSL link at most every 2 cycles which implies that the link is only half, or less than half, utilized.

Although this implementation is not optimal in resource usage (some of the states in the FSM could be reused so others can be removed), the resource costs of this accelerator is relatively low. The accelerator uses 183 LUTs and 188 FlipFlops (FF) and is comparable with a simple $12 \times 12$ bit multiplier optimized for area (180 LUTs and 158 FlipFlops). The result of this accelerator is given in Table 6.3. An improvement of 5 times is achieved with this accelerator.

As mentioned before, the accelerator is not optimal in resource usage. In future work this can be improved. In terms of execution time and latency, this is an optimal solution in the current setup with the Microblaze and an FSL communication.

### 6.3.2 Labeling

Another very time consuming kernel of the algorithm is the sub-algorithm directly after the binarization, the labeling. Labeling labels binarized pixels together which are connected (8-way) with the same label. Next to that, the algorithm keeps track of the total area, the central $x$ and $y$ coordinate and the bounding box (called ‘clip’ in the algorithm) of the component. A group of pixels labeled with the same label is called a component. To label all the components, the following approach is applied. The algorithm goes though the image from top left to bottom right, row by row. If an input pixel is 0, which means this pixel does not belong to an object, a 0 will be written in the labeled image. Every time an input pixel is 1, which means the pixel belongs to an object, the algorithm looks into the labeled image if there is already one or more labeled neighbor(s). If this is the case, the lowest label is chosen to be the label for this pixel and it is written into the labeled image. If there is no neighbor with a label, a new component is created and written in the labeled image. There is a certain order in which the neighbor label is selected. This selection method is illustrated in Figure 6.4. If the pixel ($p$ in Figure 6.4) must be labeled and the neighbor directly above the selected pixel has a label (number 1 in Figure 6.4(a)), larger than 0, this label will be copied. The next option is right above (number 2), followed by left above (number 3) and left at the same row (number 4) of the pixel. If right above is selected, the algorithm checks if two component must be merged. This is illustrated (Figure 6.4(b)) in two steps, after the right above check (2.0), first 2.1 checked on a label, after that 2.2 is checked. If a merge takes place, the list of connected components is updated (the list of connected components is explained in the next paragraph). Every time a non-zero label is written in the labeled image the auxiliary arrays (like area and position) are updated.

This is a standard sequential approach to label components, only the second pass of the original sequential algorithm is discarded. Skipping the second pass of the algorithm leads to a non-perfect labeled image. An example of this imperfection appears when two components merge; the connection between the two components is saved in a list. The list is called the List of Connected Components (ListofCC). Originally the component with the highest number would be relabeled to the lowest label number in the second pass of the original algorithm. Skipping this relabeling saves one iteration over the image, but has a drawback. When the label of a pixel is needed in a follow up function, the label must be checked first in the list of connected components, to make sure that the label is correct. Although the labeled image is not perfect, the labeled image together with the produced list of connected components are sufficient to continue with the next steps of the algorithm.
Due to the order of selection in the neighbor labels, the mid above label is selected more often than the rest. This is verified using the test images set. An average picture from the test set contains around 40 percent ones after binarization. During labeling, more than 90 percent of the ones is labeled using the mid above neighbor selection. This means the accelerator must have a very short execution time for mid above labeling and zero labeling to establish a short execution time. In Table 6.2 the average percentage for each selection method, of the labeled pixels of the test set is shown.

<table>
<thead>
<tr>
<th>Selection method</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>No labeling (0 in binarization)</td>
<td>60</td>
</tr>
<tr>
<td>Mid above</td>
<td>36</td>
</tr>
<tr>
<td>Right above</td>
<td>1</td>
</tr>
<tr>
<td>Left above</td>
<td>0</td>
</tr>
<tr>
<td>Left at the same row</td>
<td>0</td>
</tr>
<tr>
<td>New component</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.2: Preference of the labeling selection algorithm (average from all test images)

During the execution of the algorithm, a lot of data must be kept in the accelerator. To establish this, a few BRAMs are instantiated in the accelerator. These BRAMs are Xilinx’s own dual ported memory blocks which are suitable to keep, read and write data. The BRAMs are placed in hardware on the FPGA. The accelerator has 3 BRAMs to keep auxiliary values (position, area and clip), one BRAM that serves as history buffer and one BRAM that contains the list of connected components. The accelerator first receives a whole row of binarized values (20 words times 32 bits) and then start processing. During the processing, the labeled pixels are send to the Microblaze. In this algorithm it was possible to use the same trick as with the binarization: put more outputs in one FSL word, but this is not done. Test indicates that the gained time due to reduced number of FSL transfers is lost in the Microblaze. The Microblaze uses too many cycles to extract/separate the outputs. Also the same problem as with the binarization (subsection 6.3.1), occurs here. The Microblaze cannot provide the data fast enough to fully utilize the labeling accelerator. Both problems (or actually this is one problem) could be solved using a Direct Memory Access (DMA) controller. The Microblaze could issue the DMA to copy the labeled pixels back to the DDR3, while the Microblaze itself could do other processing or copy actions. A DMA implementation is not as efficient in resource usage as the current one and will invoke some time to create.
Figure 6.5(a) gives an overview of the functionality of the accelerator and in Figure 6.5(b) an overview of the hardware setup is given. A detailed overview of the processing block is given in Figure 6.6. The different colors in this figure indicate the different execution paths as previously mentioned. This figure shows that the zero and mid above selection have the shortest execution paths (indicated by the red and yellow blocks). The other paths are longer and the execution time is data depended. A common practice during design of such accelerators is the creation of a pipeline. To create a pipeline, all execution paths must be of almost equal length (if this is not the case, the pipeline must have a stall option) and no data dependencies may be violated. To create equal length execution paths was hard, due to the data dependency of some of the cases (orange paths in Figure 6.6). The information from previous iteration is needed in the current iteration, which is also a difficulty in a fully pipelined implementation. The current implementation is more efficient than a fully pipelined implementation in resource usage and no data dependencies are violated.

![Diagram](image)

(a) Functional overview  (b) Hardware overview

Figure 6.5: Functional and hardware overview of the labeling accelerator

The whole accelerator is implemented as a FSM (like the binarization accelerator). In this way it is easy to switch between receiving inputs and outputs. The different options described in Figure 6.6 can easily be implemented in the FSM. This accelerator consists of 15 states, including the states which output the auxiliary arrays at the end. The reuse of logic and states is very high in this accelerator. The total used resources are 1168 LUTs, 1702 FlipFlops and 6 BRAMs. This accelerator occupies more resources than the binarization accelerator, but with such a complex function and relative large data structures the used resources stay within 1 percent of the total resources of the VIRTEX 6.

The results of the new created accelerator are given in Table 6.3. The execution time is reduced almost 7 times and this implementation is faster than the labeling function on the ARM. Most of the speedup is due to the parallel fetching and updating of the auxiliary arrays, but also intelligent prefetching of history values and partial pipelining reduce the execution time.

This implementation is not the fastest one. This is due to the use of the BRAMs. If all arrays are kept in FlipFlops inside the logic slices, the design will explode in resource usage but it can omit the fetch and store cycles from the BRAM and perform all paths (except labeling a 0) which are indicated in Figure 6.6, one or more cycle(s) faster. However, the use
of BRAMs is required otherwise the implementation is not realistic.

With the BRAM requirement, this implementation is the fastest one achievable at the processing part, but not the total execution time could be reduced. Reducing the total execution time could be done by decoupling the input receiving part from this FSM and place it into another auxiliary FSM. With these two FSMs, it is possible to receive the next inputs at the same time as processing the current inputs. This will speed up the accelerators, but probably it will not result in a shorter execution time. This is due to the Microblaze; it cannot receive and send data at the same time because it is a single issue machine. The separated FSM setup will probably result in a higher resource usage due to the extra buffer between the two FSMs and some extra state decoding from the second FSM.

### 6.3.3 Pattern extraction

The third accelerator is created for the pattern extraction, which is the third most time consuming function on the Microblaze. As indicated in subsection 3.2.6, the pattern extraction consists of two parts. The first one is the calculation of the rotation/transformation matrix, the second one is the real pattern extraction. The second part consists of an address cal-

Figure 6.6: Processing block of the labeling accelerator; different colors indicate the labeling selection methods as described in Table 6.2
Table 6.3: ARM implementation, MB version, HW accelerated MB version and the speedup due to the accelerators (compared with the MB version)

calculation and an addition from the value of the pixels located at those addresses (the added pixels together make the extracted pattern). The address calculation uses the aforementioned rotation matrix to calculate the \(x\) and \(y\) coordinate of the pixel which must be added to a certain cell/bin of the extracted pattern. This is done a fixed number of times for every marker. Both parts of the algorithm include a lot of floating point calculations.

In this accelerator, floating point calculations are performed using IP blocks generated with the CORE GENerator from Xilinx. Using the Coregen, it is possible to generate a floating point unit specific for a certain function (like add/subtract or multiplication) for a certain floating point type (single or double precision). Some parameters of the floating point core can be tuned; these parameters influence the execution time, used resources and maximum possible frequency.

The implementation of the first part of the algorithm was relatively straightforward. The steps which are executed in the C code are converted into stages of the FSM. If a floating point computation must be performed, the correct IP is created and connected to the FSM. The parameters of the IP are tuned in such a way that the computation is performed in as few clock cycles as possible, but can still be executed at 100 MHz. The second part of the algorithm is a bit more difficult. The required data (image) is still in the Microblaze. To fetch a pixel, the accelerator has to send the coordinates to the Microblaze (via FSL); the Microblaze fetches the pixel from DDR3 or cache and put it back to the accelerator using the FSL link. To enable a fast execution of the second part, a second FSM is created. This is visualized in Figure 6.7. In this way, it is possible to compute the next \(x\) and \(y\) coordinates while waiting for the current pixel value. In Figure 6.7 4 FSL links are visible, but in hardware only two are present. The links are not used at the same time, so they can be merged into two FSL links, one from and one to the Microblaze.

The results are given in Table 6.3. A speedup of more than 7 times is achieved. Most of the speedup is enabled by parallel computations of \(x\) and \(y\) coordinates and parallel computation/fetching as explained before. This accelerator uses more resources than the ones created before. 10611 LUTs, 12023 FlipFlops, 5 BRAMs and 31 DSP blocks are used. This high
resource usage is mainly due to the use of the floating point cores, they are very expensive in resources. Due to the previous mentioned parallel execution paths the latency is reduced to the minimum. Due to the parallel FSMs, the throughput is increased. As indicated in Table 6.4, this accelerator is larger than the whole Microblaze in terms of Slice LUTs, Slice registers and DSPs.

<table>
<thead>
<tr>
<th>Function</th>
<th>LUTs</th>
<th>FlipFlops</th>
<th>BRAMs</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microblaze</td>
<td>6691 (4%)</td>
<td>9179 (3%)</td>
<td>87 (21%)</td>
<td>6 (1%)</td>
</tr>
<tr>
<td>Binarization</td>
<td>183 (0%)</td>
<td>188 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Labeling</td>
<td>1168 (1%)</td>
<td>1702 (1%)</td>
<td>6 (1%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Pattern Extraction</td>
<td>10611 (7%)</td>
<td>12023 (4%)</td>
<td>5 (1%)</td>
<td>31 (4%)</td>
</tr>
</tbody>
</table>

Table 6.4: Resource usage on the FPGA per accelerator and from the Microblaze. The percentage of used resources from the VIRTEX 6 is between brackets.

6.3.4 Evaluation

The three most time consuming functions are mapped to hardware in this implementation. With these three handwritten accelerators, almost the same frame rate as the ARM processor could be achieved. Creating these accelerators was a time consuming job. The simplest function, the binarization function, resulted in the smallest accelerator. Most time is spend on creating the labeling function. Especially the partial pipeline creation was time consuming, there were a lot of cases which should be taken care of. The very optimized code which implies a small area and a short execution time is a good result of the work done. The creation of the
last accelerator took the shortest time. Although some experience was already gained during the previous accelerator creations, the use of the IP-cores from Xilinx reduced the creation time as well. The pattern extraction is the largest accelerator; it is also the one who gains the most speedup compared to the initial Microblaze implementation. The labeling has the largest speedup compared to the ARM implementation.

There are opportunities to explore more parallelism in the binarization accelerator (there is no dependence between different pixels), but this is only limited exploited. Already with this limited use of this parallelism, the accelerator is faster than the Microblaze can provide data. It was not necessary to exploit this extra parallelism, which often implies a higher resource usage, to get a shorter execution time. In the pattern extraction, there is some parallelism which is not exploited. Most of the computations are done in parallel (where possible) but the address generation from the pixels which has to be fetched, is not done in parallel. The addresses of these pixels could be calculated completely independent from each other and could be speedup the implementation. This is, of course, at a certain cost of resources. The Microblaze which has to serve these address/pixel requests can become a limiting factor then.

During the creation of the labeling accelerator a small literature study is performed to gain some knowledge about this algorithm and how it could be implemented. There are some implementations from the labeling function on an FPGA and GPU. But all of them are, or sequential implementations, or certain knowledge is assumed about the image which has to be processed. The best and most suitable two papers are listed here. In [23] a VLSI architecture is purposed which can label an image in two scans. This implementation basically does the sequential labeling approach as described in subsection 3.2.2 but on two pixels at the same time instead of one. The second scan relabels the merged components. Another approach is the one in [13]. The paper describes connected component labeling on a GPU. This is done in parallel, but in a sort of hierarchal way. First the whole image is labeled with one label per positive binarized pixel. After that, the neighbors are examined and the labels with the lowest numbers are propagated. This step is repeated a number of times, until the whole image is correctly labeled.

The first approach could have been a possibility for this accelerator. After implementing the sequential variant, with one core, and discovering that not the accelerator but the Microblaze is the bottleneck this path is discarded. If the possibilities are there to speed up the FSL connection and the current accelerator becomes the bottleneck, this paper describes a good way to speed up the implementation. The resource usage described in the paper is 5 times as high as the current resource usage and in the paper are auxiliary arrays not taken into account (auxiliary arrays are an extension on the original labeling function) so a tradeoff must be made if the speedup is worth the required resources. A small downside of this implementation is that it could not immediate replace the current one. Because two pixels are labeled at the same moment, both labels have to be send back to the Microblaze. This cost two cycles or the labels must be multiplexed into one word (which cost more cycles to de-multiplex in the Microblaze). In the approach described in the second paper the whole image must be kept in the accelerator. For a GPU, as used in the paper, this is not a problem but for an FPGA this could be a problem due to the limited memory resources. Next to that, the streaming efficiency will be partially lost in this implementation. Because there are multiple runs over the image and in those runs the whole image must be compared with a previous one, the sequential outputting of the image can only start if all comparisons are done.

According to Amdahl’s law ([3]) it is not wise to spend more time on pattern extraction
because it is not the most time consuming function anymore. If the pattern extraction function is further accelerated with e.g. a factor of 10, the frame rate is improved to 19.4 fps. This only 6 percent faster. Due to the relative larger influence of the other sub-algorithms on the execution time it is wise to spend time on optimizing them. Implementing other functions, but also optimizing labeling and binarization will have a positive influence on the execution time because they consume the most time at the moment.

6.4 Combining Hardware accelerators

In subsection 6.3.1 and 6.3.2 it is mentioned that the Microblaze/FSL link is often the limiting factor for the accelerators. The accelerators are faster than the Microblaze. The data providing/receiving speed of the Microblaze is too low. Copying data to the FSL costs the Microblaze at least two cycles, one for fetching it from memory (if it is in the cache, otherwise it could easily need 20 cycles) and one for copying it to the FSL link. Receiving costs always two cycles, one for fetching it from the FSL link and one for storing the data. So the FSL links are underutilized. This can be solved in two ways, utilize the link more or decrease the number of transfers. The first option is difficult to achieve in the current setup, so the number of transfers via FSL must be reduced to enable a speedup. Speedup is not always guaranteed when reducing the number of transfers over the FSL link and cropping the data into less words, see third paragraph of subsection 6.3.2. The binarization and labeling are executed after each other, both in accelerators. The unnecessary copying of data from the binarization accelerator to the Microblaze and back from the Microblaze to the labeling accelerator can easily be omitted. This is illustrated in Figure 6.8. The results of this merging are shown in Table 6.5. To give a complete overview, the binarization and labeling are also merged in the ARM implementation and in the initial Microblaze implementation. The execution time of binarization and labeling in the ARM implementation is roughly the same as the two separated execution times added together.

![Figure 6.8: Overview of the hardware setup of the combined accelerators](image)

In Table 6.5 it can be seen that combining the accelerators decreases the execution time with 19 percent. This is a remarkable result because the number of transactions over the FSL link is only decreased with 4 percent. The large difference can be explained by the size of the intermediate arrays. In the original implementation, the complete binarized image must be received before the labeling was started. The binarized image has a size of approximately 80kB. This will not fit in the cache, so the Microblaze should often wait more than 1 cycle before it can send the data over the FSL. This is the cause of the large speedup of the
combined accelerator. The resource usage of this accelerator is 1173 LUTs, 1174 FlipFlops and 6 BRAMs. The number of BRAMs is the same as in the labeling function; this is because not more information is stored. The number of FlipFlops is reduced. This is mainly because the input buffer from the labeling (640 byte) is replaced by a 2 byte buffer. The LUT usage is roughly the same as in the labeling function. Somewhat more because the binarization logics are added, but somewhat less than the two separated ones added together because half of the FSL link logics are now lacking.

### 6.5 Hardware accelerators - revisited

The created accelerators are not optimal in performance. Is it possible to optimize them more? Which surrounding logic/connections must be changed to get more performance out of the accelerators? The next paragraphs give an indication of the logic which could be changed to improve the implementation and what the expected performance will be after the change.

#### 6.5.1 Improvement 1: Dedicated camera link or DMA

At the start of the program, the image is loaded from flash card into the DDR3 memory and sections of the program have to fetch the data from DDR3 or from the cache. This causes slowdowns, which are already often mentioned before. But if a camera is connected to the FPGA, the camera will stream the data in on request. It could be possible to connect the binarization directly to the camera link, or connect the camera link to the Microblaze. In case of the second option, the Microblaze will be able to fetch a pixel in a clock cycle and put it to the FSL link in another cycle. Both options are mimicked in the Microblaze. In case of the first option, a dummy input is provided to the FSL link every clock cycle. The execution time of the binarization accelerator is reduced to 3347us in this case. In case of the second option, the result is 5653 us. The same experiment is applied on the combined accelerator. This was a bit more difficult, because the dummy input leads to corrupt labeling (for example: all zeros will go through the shortest labeling path), but based on the average number of different label execution paths (Table 6.2) a good estimation can be made. The result of this estimation is 6364us for the first option and 8668us for the second one.
Another option to speed up the binarization is the use of a Direct Memory Access (DMA) controller. A DMA can reduce the copy workload of the Microblaze. When initialized, a DMA will automatically copy a block of memory to a specified place; this can be a FSL link or another memory structure. Although DMA is a Xilinx IP-core, it is not just plug and play to get it working. Next to the DMA, an extra PLB bus with corresponding master and slaves must be created. It is hard to estimate what the speedup of this improvement will be, but expected is that it is in the same range as with the Camera Link.

6.5.2 Improvement 2: Use more resources for the pattern extraction

In the pattern extraction 90 percent of the time is spent on the second part of the algorithm. In those 3395us, 4096 iterations are performed. One iteration consists of an address calculation in parallel with a request for pixel and an update of the extracted pattern. The address calculation takes the most time (approx 800ns). Fetching pixels from the memory takes at most 250ns.

The different iterations of the second part of the pattern extraction are completely independent and can be executed next to each other. If the number of floating point units which generate the addresses is extended, so 4 addresses can be calculated in parallel, the total execution time can be reduced to 849us for the pattern extraction part. If the execution of the different floating point units is pipelined, it is possible for the Microblaze to serve all the floating point units after each other. Due to this sequential service of the Microblaze, the pixels can be put in one pattern. The total execution time for this function (including the first rotation matrix calculation) will be 1226us. This extension will invoke more floating point cores (global estimation: 3 extra single precision floating point adders, 3 extra single precision dividers and 3 extra single precision multipliers). No more memory blocks will be the needed.

In subsection 6.3.4 is mentioned that it is not wise to spend more time on optimizing this function because it is not one of the most time consuming functions anymore. This is true, though this improvement is mentioned here. The improvement will be minor on the total frame rate, but time to create the improvement will be minor as well. The created accelerator is suitable for multiple address calculation units and expected is that this can be implemented in a limited amount of time.

6.5.3 Improvement 3: Line contour estimation accelerator

The most time consuming function at this moment is the line contour estimation function. If the implementation on FPGA must be improved more, this is the function to look into. The line contour estimation consists of multiple sub functions. First, the coordinates must be corrected with the camera parameters. After that, the eigenvalue, eigenvector and mean must be calculated. Parts of these algorithms can be implemented in parallel (especially the pixel correction with camera parameters) and parts must be implemented sequential. There are a lot of opportunities to improve this implementation and expected is that the speedup is in the same range as the previous accelerators, around 7 times faster. There can be some issues which are overlooked at the moment, so the speedup factor is conservatively determined at 4 times, the total execution time will be 3882us.
6.5.4 Overview of the improvements

In the previous paragraphs some directions are given to speed up the implementation. The results of these improvements are given in Table 6.6. It can be seen that the expected execution time of this implementation is faster than the ARM implementation and reaches the required 30 fps.

<table>
<thead>
<tr>
<th></th>
<th>ARM (us) 720 MHz</th>
<th>MB (us) 100 MHz</th>
<th>MB + HW acc (us) 100 MHz</th>
<th>Expected (us) 100 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binarization and Labeling</td>
<td>39702</td>
<td>117691</td>
<td>18226</td>
<td>8668</td>
</tr>
<tr>
<td>Contour detection</td>
<td>1526</td>
<td>6124</td>
<td>6124</td>
<td>6124</td>
</tr>
<tr>
<td>Check square</td>
<td>702</td>
<td>5871</td>
<td>5871</td>
<td>5871</td>
</tr>
<tr>
<td>Remove components</td>
<td>92</td>
<td>277</td>
<td>277</td>
<td>277</td>
</tr>
<tr>
<td>Pattern extraction</td>
<td>5127</td>
<td>28482</td>
<td>3772</td>
<td>1226</td>
</tr>
<tr>
<td>Template matching</td>
<td>275</td>
<td>997</td>
<td>997</td>
<td>997</td>
</tr>
<tr>
<td>Line contour estimation</td>
<td>3540</td>
<td>15525</td>
<td>15525</td>
<td>3882</td>
</tr>
<tr>
<td>Sub-pixel-corner detection</td>
<td>31</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td><strong>Total (us)</strong></td>
<td><strong>50995</strong></td>
<td><strong>174977</strong></td>
<td><strong>50802</strong></td>
<td><strong>27055</strong></td>
</tr>
<tr>
<td><strong>Fps (approx)</strong></td>
<td><strong>19.610</strong></td>
<td><strong>5.715</strong></td>
<td><strong>19.684</strong></td>
<td><strong>36.962</strong></td>
</tr>
</tbody>
</table>

Table 6.6: ARM implementation code, initial MB version, Combined HW accelerated MB version and the expected performance
Chapter 7

FPGA implementation using High Level Synthesis tools

The second implementation of the marker detection algorithm on the FPGA in described in this chapter. The initial Microblaze implementation is used as basis. The previous, hand-written, accelerators are now created with a High Level Synthesis tool. Most HLS tools use C or C++ code as basis and output VHDL or Verilog code after compiling. The HDL code can be imported into the Xilinx project and compiled together with the rest of the code. This chapter describes the use of two tools, ROCCC (Section 7.1) and AUTOESL (Section 7.2), and evaluates the achieved performance and the used resources on the FPGA of the produced code. The ease of usability is also taken into account during this evaluation. The motivation for this two tools is already given in Section 2.4.

7.1 ROCCC

The first tool which is tried is the Riverside Optimizing Compiler for Configurable Computing (ROCCC) from Jacquard computing. ROCCC is an open source tool which consists of a compiler (which converts the C-code into VHDL-code) and a GUI. The GUI is a plug-in for the well know eclipse environment. The installation of this tool is easy; a good tutorial is given on the website. The tool is only available for a Linux based operating system. The Xilinx toolchain for programming FPGAs has a Linux variant as well so that will not cause problems.

7.1.1 Modules and Systems

ROCCC distinguishes two different components in its tooling. These components are called modules and systems. It is important to make a good separation between these two components. A module is a computational data path and must be written as a computational function. Both input and output must be scalar values and modules may not have arrays as input or output variables. Internal variables may be created but are not visible to the outside of the module. Loops may be created in a module, but they are automatically fully unrolled during compilation. Therefore loops must have static bounds.

Systems can perform computations on streams of data and can produce streams of data as output. Scalars may also be used as inputs and outputs, but scalar inputs are only read
once at the beginning of the program and the output scalars are generated once at the end of the computation. Systems generate BRAMs at their inputs and outputs to read data from and store the data streams. The size of the BRAMs is depending on the window size (part of the data stream where the system performs its operations) which is needed inside the system. This means the BRAMs will contain as less data as possible, depending on the needs of the systems. Modules and other systems can be inlined in systems and other modules. Loops are not automatically unrolled, but can be partially unrolled using compiler directives.

### 7.1.2 Compiler directives

One compiler directive is already mentioned, it is possible to (partially) unroll loops in systems. There are some other directives. There are not discussed in detail here, but can be found on the website of ROCCC. Most of the directives have to do with loops (partial unroll, interchanging, fusion) or with resource usage (common sub expression elimination). Directives which influence the performance can also be found. Most of these directive try to replace multiplies and divisions or inline modules into systems. The last directive gives the number of pipeline stages the hardware may have. This has a large influence on the frequency of the design. The number of pipeline stages can be set per module/system. If a low number of pipeline stages is allowed, generally the frequency will drop. Compiler directives are specified per module/system. They cannot be specified per inner loop or sub function. E.g. it is not possible to unroll one loop with another factor as another loop in the same system.

### 7.1.3 Binarization accelerator

The first accelerator which was created with ROCCC was the binarization accelerator. The functionality of this accelerator was explained in subsection 3.2.1. It was easy to code this accelerator in C, although already one downside of ROCCC immediately was discovered. When specifying the input and output ports of a module, it is not allowed to use unsigned integers. Only (single precision) floats and signed integers are supported. The packaging which is done in subsection 6.3.1 cannot be used in this accelerator. Another problem which is discovered is the interface with the Microblaze. Originally it was planned to use the FSL link, the same link as in the handwritten accelerator, but there is no support for this link in ROCCC. The only supported interface is the PLB bus. The PLB interface may only be used in combination with modules in ROCCC, which is also a downside of this tool. The created accelerator has no support for streams; it is basically a fast/extended function unit of the processor. This means, every bunch of pixels, one or multiple outputs are generated and every iteration must be totally completed before another one can be started. The positive influence of streaming as in the handwritten accelerator is totally lost in this setup. The independence between successive pixels is only partly exploited (only the pixels in one function call are processed in parallel). Although many downsides are mentioned so far, the ease of generating this accelerator was positive. Within a day the accelerator was generated and tested. Although all different optimization and compiler directives are tested, it was not possible to achieve a better execution time than the initial Microblaze implementation. The final and best implementation reads 24 pixels, processes them and writes the corresponding outputs. This is done in only 4 cycles, which is relatively fast (faster than the handwritten accelerator). Most of the time is lost in the Microblaze. The gathering of data and putting it at the PLB bus costs too much time (at least more than streaming it via the FSL).
used resources for this implementation are 4783 LUTs and 2915 FlipFlops. No BRAMs and DSPs where used in this implementation. The resource usage is very large for a binarization accelerator. A small note must be made here, the PLB interface is included into the resource usage results, but this is a minor part of the used resources.

7.1.4 No other accelerators

Due to the bad performance of the generated binarization accelerator and the limited number of interfacing options of ROCCC, no more accelerators are generated with ROCCC. Although it is expected that the tool is better than it has shown in the binarization accelerator, it is simply not usable for the creation of hardware accelerators. Probably when whole designs are created with this tool, and no interfacing with a Microblaze is involved in the algorithm, better results can be achieved.

7.2 AutoESL

The second tool which is used to generate the accelerators is AutoESL. AutoESL is a tool from Xilinx which is not part of the default toolchain to program FGPA's. The tool can be downloaded and installed separately from the other toolchain. The AutoESL tool is closely integrated with the other toolchain from Xilinx. AutoESL is a tool where creation, compilation, simulation and implementation are done in the same program.

7.2.1 Testing

One large advantage of the tool is already discovered when doing the first tutorials. If an accelerator is generated, this accelerator can be tested using a C test bench. The accelerator is called like a normal C function, but simulated in system-C. This system-C is generated by the tool as intermediate step and will be converted to Verilog or VHDL in a later stadium. The produced outputs can be checked in the C test file. In this way, it is easy to specify a large test bench and the produced outputs can be easily checked. Another advantage with regard to the ROCCC tool is the possibility of using all kinds of data types. Fixed point, floating point (double and single precision), unsigned and signed integers of different lengths and so on, all is possible with this tool.

7.2.2 Compiler directives

Like in the ROCCC tool, it is possible to use directives in AutoESL. In AutoESL, directives are specified in a directives file, or directly in the source code. The examples given in this section contain directives specified in the source code. A lot of directives are available, both local and global, and the user has the job to specify the correct ones at the right place. Often this is a non trivial job, because the designer must know which hardware is generated in a certain case and which directive solves the bottleneck which is encountered. Most global directives influence the compiler with general statements like: optimize for speed or try to reuse hardware at a high level. Local directives can influence loops, functions, interfaces and memories like: Inline this function in the main function, map this array into two memories in a cyclic way or pipeline this loop with an initiation interval of four cycles. Finally, the tool
contains a visualization tab, which visualizes the generated schematic that can be used for debugging and solving bottlenecks.

### 7.2.3 Binarization

The first accelerator which is implemented using AutoESL is the binarization accelerator. The same functionality and interfaces are specified as in the handwritten one. The specification of the FSL link is relatively easy in this tool, although it was not documented or given in the provided examples. In the function call, both input and output must be mentioned as an array. The FSL directive must be given to this array and the tool will automatically generate the correct interface. The user must take care of the order in which the elements in the array (which is a FIFO link) are addressed. Fetching elements must be in order; otherwise the execution in hardware will fail. The order can be checked with a test bench. The specification of the FSL link of the binarization accelerator is given in Code Example 7.1. Every link gets two directives. One specifies the interface, which is FIFO interface in this case. The other directive specifies the adapter. In this case the adapter is an FSL link. The

```
void binarization ( unsigned int pixel_in[24], unsigned int* output ){
  #pragma AP resource variable=output core=FSL
  #pragma AP interface ap_fifo port=output
  #pragma AP resource variable=pixel_in core=FSL
  #pragma AP interface ap_fifo port=pixel_in

  // rest of the function body
}
```

Code Example 7.1: Specification of a FSL interface in AutoESL

rest of the binarization kernel is written in normal C, like the code in the Microblaze. The global directives for extensive search for hardware reuse are enabled. Other directives are not used because they were not useful or not applicable in this algorithm.

In total, the porting of this accelerator is done within 1.5 days. The first half of this time is spend on creating the code, specifying the test bench and optimizing for performance. The last $\frac{3}{4}$ day is spent on reducing the area of the generated code. The reached performance and the used resources are given in Table 7.1. The increase in area and execution time between the handwritten code and the AutoESL generated code is also given in previous mentioned table. The achieved performance is only 6 percent slower than the handwritten one, which is a nice result. The use of LUTs is a lot higher and an in depth look into the generated reports indicate that most of the extra LUTs are used in multiplexing the data path. Some effort is spend trying to reduce this, but a better result is not achieved. The number of FlipFlops is lower than in the handwritten implementation. Probably some of the temporary results are saved inside the LUTs and a tradeoff can be made between saving temporary LUTs and FlipFlops.

### 7.2.4 Labeling

The second accelerator is the labeling accelerator. This accelerator is implemented with the same interfaces as the binarization accelerator. The creation of the interfaces was easy and
the creation of the rest of the accelerator was not that difficult either. The real problems appeared when starting to optimize the accelerator. The initial performance was about 4 times as slow as the handwritten one. This was mainly because the tool tried to equalize the execution time of all label selection paths in the main loop of the accelerator (see Figure 6.6 for the different selection paths). Due to the optimization of the most occurring selection paths in the handwritten accelerator the high performance is achieved. To establish this in the AutoESL tool, the code had to be rewritten. The original code consists of a number of else-if statements to distinguish the different paths. To enable the different execution path lengths, all cases are written in if/else statements where the shortest paths are in the first if statements. This decreased the execution time with a factor of 1.5 times. The rewriting of the code is illustrated in Code Example 7.2 and Code Example 7.3.

An intelligent prefetch mechanism is developed in the handwritten implementation and the code is partially pipelined. This is hard to mimic in the AutoESL tool. The prefetching can be established, but gives a very limited speedup. This speedup was achieved after partitioning the history buffer into two equal pieces in a cyclic way. In this way, more items can be fetched in one cycle. Originally 2 items could be fetched (history buffer was implemented in one dual port BRAM), with this partitioning 4 items can be fetched (2 dual port BRAMs are used). The partitioning of the array is illustrated in Figure 7.1, the used directive is given in Code Example 7.4. The partitioning could only be done in a cyclic way due to the sequential order fetching from the labeling algorithm.

Partial pipelining cannot be established, only full pipelining can be created by the tool. The partial and full pipelines are illustrated in Figure 7.2. Full pipelining cannot be applied to this function because for a full pipeline, all execution paths must be of equal length. This can be established, but the cycle count of the most taken paths will be so large (above 5 cycles per iteration) this has no profit anymore. Due to the data dependencies inside the algorithm, the initiation interval of the pipeline must be 4 cycles. Establishing a pipeline will introduce more resource usage, but will not speed up the implementation.

In Table 7.2 can be seen that the AutoESL generated accelerator is approximately a factor of 2 slower. This is faster than the initial, software only, implementation but still the performance is a bit disappointing. An iteration of labeling a pixel with a 0 or using the mid above label takes one cycle more than in the handwritten implementation. This extra cycle accounts for large part of the increased execution time. Table 7.2 displays also the used resources. It can be seen that the tool has chosen to use two extra BRAMs instead of implementing an array in a number of FlipFlops. The same effect appears as with the binarization accelerator, the used number of LUTs is more than 400 percent higher than in the handwritten case. Also here, the data path multiplexing is the most resource demanding.

<table>
<thead>
<tr>
<th></th>
<th>Handwritten</th>
<th>AutoESL</th>
<th>Increase (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time (us)</td>
<td>11654</td>
<td>12318</td>
<td>6</td>
</tr>
<tr>
<td>LUTS</td>
<td>183</td>
<td>986</td>
<td>439</td>
</tr>
<tr>
<td>FlipFlops</td>
<td>188</td>
<td>77</td>
<td>-59</td>
</tr>
</tbody>
</table>

Table 7.1: Comparison of the handwritten and AutoESL generated binarization accelerator
if ( binarized_pixel==0 ){
  output_label=0;
}
else{
  if( mid_above_label>0 ){
    output_label=mid_above_label;
  }
  else if( right_above_label>0 ){
    if( left_above_label>0 ){
      output_label=left_above_label;
      // merge two components
    }
    else if( previous_label>0 ){
      output_label= previous_label;
      // merge two components
    }
    else{
      output_label= right_above_label;
    }
  }
  else if( left_above_label>0 ){
    output_label= left_above_label;
    // create a new component
    output_label= new_component_number;
  }
}

if ( binarized_pixel==0 ){
  output_label=0;
}
else{
  if( mid_above_label>0 ){
    output_label=mid_above_label;
  }
  else if( right_above_label>0 ){
    if( left_above_label>0 ){
      output_label=left_above_label;
      // merge two components
    }
    else{
      output_label= right_above_label;
    }
  }
  else if( previous_label>0 ){
    output_label= previous_label;
  }
  else{
    // create a new component
    output_label= new_component_number;
  }
}

Code Example 7.2: Original labeling code

Code Example 7.3: Part of the rewritten labeling code

<table>
<thead>
<tr>
<th></th>
<th>Handwritten</th>
<th>AutoESL</th>
<th>Increase (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time (us)</td>
<td>10692</td>
<td>22039</td>
<td>106</td>
</tr>
<tr>
<td>LUTS</td>
<td>1168</td>
<td>5687</td>
<td>387</td>
</tr>
<tr>
<td>FlipFlops</td>
<td>1702</td>
<td>600</td>
<td>−65</td>
</tr>
<tr>
<td>BRAM</td>
<td>6</td>
<td>8</td>
<td>33</td>
</tr>
</tbody>
</table>

Table 7.2: Comparison of the handwritten and AutoESL generated labeling accelerator

### 7.2.5 Pattern Extraction

The third and final accelerator implemented in AutoESL is the pattern extraction. The same procedure is used to generate the interfaces. Specifying arrays and give the correct directives
defines the interface. Almost the same code as the original pattern extraction for the Microblaze is used to create this accelerator. Only some renaming and moving of the input and output interfaces is done. This function consists of many loops and different sub-functions, so lots of opportunities to optimize and give directives to enable a good performance. The execution time of the generated accelerator comes within 24 percent of the handwritten accelerator. This is pretty good, but also in this case a lot more resources are used. The number
of extra consumed resources is not as large as in the previous accelerators (in percentage). An explanation for this is that a lot of standard IP-cores are generated with the CoreGen from Xilinx; this is also done in the handwritten implementation.

Partitioning intermediate arrays is tried here as well. Partitioning caused a small speedup (< 1%), but an enormous increase in used resources (> 30%). Due to the partitioning of the work arrays, a lot more parallelism is enabled. This parallelism is exploited, so more computations are performed in parallel, but for this exploration much more floating point units are needed which caused the increase in resources. The extra compaction units where are utilized in the first part of the algorithm (rotation matrix calculation), which caused the limited speedup.

<table>
<thead>
<tr>
<th></th>
<th>Handwritten</th>
<th>AutoESL</th>
<th>Increase (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time (us)</td>
<td>3772</td>
<td>4666</td>
<td>24</td>
</tr>
<tr>
<td>LUTS</td>
<td>10611</td>
<td>16002</td>
<td>51</td>
</tr>
<tr>
<td>FlipFlops</td>
<td>12023</td>
<td>8835</td>
<td>-27</td>
</tr>
<tr>
<td>BRAM</td>
<td>5</td>
<td>7</td>
<td>40</td>
</tr>
<tr>
<td>DSPs</td>
<td>31</td>
<td>41</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 7.3: Comparison of the handwritten and AutoESL generated pattern extraction accelerator

7.3 Evaluation of the two HLS tools

Previous described research covers the evaluation of two HLS tools. A short and general evaluation of these tools is given in the sections below.

7.3.1 ROCCC

The first evaluated tool is the ROCCC HLS tool. This tool was not suitable to create accelerators which could speed up the implementation. The support for Xilinx interfaces was limited, only the PLB interface within the Xilinx 12.x toolchain is supported. Although there are papers and evaluations over this tool, ROCCC was not usable in this case. In [20] VHDL code is generated which has higher throughput and lower latency than its handwritten counterpart. Probably in larger designs, which do have no or limited communication with the Microblaze the tool can perform better. Communications with other parts in the FPGA are not discussed in the ROCCC paper. A possibility to use this tool which is not explored is the generation of a generic wrapper. This wrapper must be placed between the FSL link and the generated accelerator. Wrapper generation is not done during this research because the focus was on exploring the HLS tool, and not on get this one working at all cost.

7.3.2 AutoESL

AutoESL performs better than the ROCC tool. Generating different interfaces which can communicate with the Microblaze is not difficult. Porting and simulating code is not that difficult either, although simulations can take a long time for larger test benches. In general, the generated code uses more resources and it has a preference to use BRAMs instead
of FlipFlops. So far, the use of more BRAMs is not a problem. In [6] a sphere detector is implemented using AutoESL. The number of used resources of this implementation is approximately the same as the handwritten implementation. This result could not be achieved in this thesis. In the before mentioned research paper is approximately the same amount of time spend on optimizing the code for AutoESL than on writing VHDL code in the handwritten implementation. In this research the time spend on implementing and optimizing AutoESL code is far less than the time spend on writing VHDL code (differs a factor of 5 to 6). So maybe improvements can be achieved when optimizing this code further. In general it can be concluded that code which can be relatively easy run in parallel (such as the binarization and the pattern extraction) achieves the best performance when using AutoESL.

The AutoESL program has a user-friendly GUI. The schematic viewer provides a lot of feedback to optimize the code further. The rest of the feedback to the user is limited. Identifying the bottlenecks must be done by hand by the designer. To get a good performance with this tool, the designer must have some knowledge about hardware and how it is generated. Due to this required knowledge it can be said that this tool is more an auxiliary tool for hardware designers instead of creating a hardware designer from every software designer.
Chapter 8

Conclusions and Future Work

This chapter contains an overview of this work and the conclusions which can be made from gathered results. The conclusions are given in Section 8.1. After that the future work is discussed in Section 8.2.

8.1 Conclusions

The marker detection algorithm of ARToolkit can be implemented on an FPGA. This is achieved after optimizing and accelerating with hardware accelerators. The implementation will meet the requirements defined in advance. Different implementations were made before the requirements are met. The initial implementation consists of a Microblaze (soft-core processor in the FPGA) with the required peripherals attached. This initial implementation is 4 times as slow as the same algorithm on an embedded ARM processor, which is dual-issue and 7 times faster. Due to the dedicated floating point unit in the Microblaze, the performance gap is less than the expected 14 times. By optimizing the initial implementation and adding handwritten hardware accelerators to it a frame rate of almost 20 fps can be achieved. This frame rate is comparable with the frame rate on the ARM processor. Limiting factor in this Microblaze design is often the disability of the Microblaze to move and copy data from and to the accelerators. Moving data from the Microblaze to the accelerators (and vice versa) is slower than the accelerator itself and thus the Microblaze is the limiting factor in this design. Improvements with a camera link or with a DMA can be made, the line contour estimation function can be put in hardware (in an accelerator) and the pattern extraction can be speed up when more resources are added. With these improvements a frame rate of 36 fps can be achieved. This frame rate is above the required 30 fps.

A High Level Synthesis tool cannot approach the efficiency of a highly optimized handwritten accelerator until so far. Both in area (used resources of the FPGA) and in execution time, the handwritten accelerators perform better than the HLS generated ones. In this thesis, two HLS tools are evaluated: ROCCC and AutoESL. Between these two HLS tools, large differences occur. The first evaluated ROCCC tool was not suitable for the purpose of interfacing with a Microblaze. Interfacing was a requirement for designing accelerators and therefore this ROCCC tool is not further evaluated. Xilinx’s own HLS tool, AutoESL, is better for this type of interfacing. Specifying accelerators is easy in both HLS tools. Generating the first unoptimized implementation of an accelerator was also relatively easy and could be done in half a day. Optimizing these accelerators was more difficult and knowledge of (generated)
hardware is a requirement. Although some of time is spend on optimizing the accelerators in AutoESL, the minimum possible area is probably not achieved. The tutorials and reference designs recommend optimization for throughput and latency first, so the requirements in that domain are met. Later optimization for area should be performed until the requirements (or the handwritten resource usage) are met. This final step is not completely performed.

At this point, after this research, it is debatable if these HLS tools can be used in the EVA tool flow. The high performance of a handwritten accelerator cannot be mimicked until so far. Probably not all accelerators, which need to be generated with the EVA tool flow, do have the same interfaces as the specified ones in this case study. Expected is that the interfaces are not the limiting factor in this design. Maybe the high performance is not achieved due to the unfulfilled design directory. But most of the performance optimizations are tried and still the performance of the handwritten is not achieved. The requirements of the EVA project, e.g. low latency and high throughput, often imply cycle accurate design. It is hard to describe cycle accurate behavior in an HLS tool. The discovered advantages, short design time and fast exploring of different architectures are not a large advantage in the EVA project. When the templates for the EVA project are generated once, the design time will not be an issue. No different architectures have to be explored because the correct one is already chosen up front in the EVA project.

8.2 Future Work

In future work, different directions can be taken. We can spend more time on optimizing the current AutoESL implementation and look if it is possible to reach the same performance and/or area as the handwritten implementation. Another research path is to investigate other tools. There are many other HLS tools which are not evaluated in this research and maybe there are more suitable, user friendly or better performing tools available. Some other tools are already evaluated at the Embedded Systems group of the TUe. Finally, the non-HLS path can also be taken. The manual implementation can be improved and the different purposed improvements can be implemented and tested. Eventually a real-time demonstration can be made using the camera link and a small visualization kernel for displaying objects. Regarding the EVA tool flow, if a non-HLS path is taken, the designed handwritten templates can be generalized and used in the tool flow.
Bibliography


Appendix A

Pseudocode of the sub-algorithms

This appendix lists the pseudocode of the labeling algorithm and the contour following algorithm. Both algorithms are part of the marker detection algorithm of ARToolkit. Only a very brief overview is given in this pseudo code. This is done to give an indication of the working of the algorithm.

A.1 Labeling

Input: binarized_image[height][width]
Output: labeled_image[height][width], List_of_connected_components[max_nr_of_components],
area[max_nr_of_components], position[max_nr_of_components], clip[max_nr_of_components],
number_of_detected_components

for i = 0; i < height; i + + do
  for j = 0; j < width; j + + do
    if binarize_image[i][j] == 0 then
      labeled_image[i][j] = 0
    else
      if labeled_image[i - 1][j] < 0 then
        labeled_image[i][j] = labeled_image[i - 1][j]
        Update area, clip, position
      else if labeled_image[i - 1][j + 1] < 0 then
        if labeled_image[i - 1][j - 1] < 0 then
          Merge two components. Determine lowest label and update the necessary entries
          in List_of_connected_components
          labeled_image[i][j] = lowest label
        else if labeled_image[i][j - 1] < 0 then
          Merge two components. Determine lowest label and update the necessary entries
          in List_of_connected_components
          labeled_image[i][j] = lowest label
        else
          No merge, only right above has a label
          labeled_image[i][j] = labeled_image[i - 1][j + 1]
    end if
  end for
end for
Update area, clip, position
else if labeled_image[i-1][j-1] < 0 then
  labeled_image[i][j]=labeled_image[i-1][j-1]
  Update area, clip, position
else if labeled_image[i][j-1] < 0 then
  labeled_image[i][j]=labeled_image[i][j-1]
  Update area, clip, position
else
  Create a new component, increase number_of_decteded_components, initialize items
  in List_of_connected_components, area, clip and position
end if
end if
end for
end for

A.2 Contour following

Input: labeled_image[height][width], List_of_connected_components[max_nr_of_components],
clip[max_nr_of_components], number_of_decteded_components
Output: Number_of_contour_coordinates[max_nr_of_components],
Contour_coordinates[max_nr_of_components][2]
for all detected components do
  define startpoint as the minimal x and minimal y position (extracted from clip).
  while startpoint does not belong the the component do
    move startpoint one pixel in the positive x direction
  end while
  while startpoint is not reached again do
    Search for the next pixel of the contour. Make sure the pixel is a border pixel and the
    traveling over the contour goes in clockwise direction.
    Save the x and y coordinates of the discovered pixel in Contour_coordinates.
    Increase the Number_of_contour_coordinates.
    Discovered pixel becomes current pixel.
  end while
for All contour pixels do
  Discover the pixel at the largest Euclidian distance from the startpoint. This pixel is
called the New_startpoint
end for
for All contour pixels do
  Reorder the pixels from the New_startpoint in clockwise direction in the Contour_coordinates
  array.
end for
Appendix B

Fixed point variables overview

This appendix gives an overview of the required bits for the fixed point variables in the marker detection algorithm. These numbers are simulated with a Matlab implementation using the defined test sequence. The initial Matlab implementation contained over dimensioned fixed point variables, they are reduced one by one, bit by bit until the accuracy of the output variables is not met anymore. The results are given in Table B.1.
<table>
<thead>
<tr>
<th>Function</th>
<th>Variable</th>
<th>(Un)Signed</th>
<th>Int</th>
<th>Frac</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marker detection</td>
<td>Position</td>
<td>U</td>
<td>12</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Lines</td>
<td>S</td>
<td>12</td>
<td>12</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Vertex</td>
<td>U</td>
<td>12</td>
<td>12</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Confidence</td>
<td>S</td>
<td>1</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>Binarization</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Labeling</td>
<td>Pos</td>
<td>U</td>
<td>12</td>
<td>12</td>
<td>24</td>
</tr>
<tr>
<td>Contour detection</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check Square</td>
<td>Threshold</td>
<td>U</td>
<td>34</td>
<td>6</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>Condition</td>
<td>U</td>
<td>39</td>
<td>6</td>
<td>45</td>
</tr>
<tr>
<td>Remove components</td>
<td>D (intern)</td>
<td>S</td>
<td>18</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>D (output)</td>
<td>S</td>
<td>18</td>
<td>2</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Testcondition</td>
<td>U</td>
<td>18</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>Line contour estimation</td>
<td>Camera distortion</td>
<td>S</td>
<td>10</td>
<td>40</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>Mean</td>
<td>S</td>
<td>12</td>
<td>34</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>Eigenvector</td>
<td>S</td>
<td>1</td>
<td>28</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>px, py, p, z, q, z0</td>
<td>S</td>
<td>23</td>
<td>52</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>outputx, outputy</td>
<td>S</td>
<td>24</td>
<td>53</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>input</td>
<td>S</td>
<td>14</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Input (next sub function)</td>
<td>S</td>
<td>12</td>
<td>16</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>Srow, mean, work, work2</td>
<td>S</td>
<td>29</td>
<td>51</td>
<td>81</td>
</tr>
<tr>
<td></td>
<td>evvec, ev, sum</td>
<td>S</td>
<td>29</td>
<td>52</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>Srow, sum (division)</td>
<td>S</td>
<td>22</td>
<td>19</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>Work, mean,</td>
<td>S</td>
<td>25</td>
<td>18</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>Row</td>
<td>S</td>
<td>32</td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>Input, output</td>
<td>S</td>
<td>21</td>
<td>22</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>ev, dv, w, y, x</td>
<td>S</td>
<td>22</td>
<td>21</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>u, s, t, c</td>
<td>S</td>
<td>26</td>
<td>55</td>
<td>82</td>
</tr>
<tr>
<td>Sub-pixel-corner</td>
<td>lines</td>
<td>S</td>
<td>18</td>
<td>23</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>W1</td>
<td>S</td>
<td>9</td>
<td>32</td>
<td>42</td>
</tr>
<tr>
<td>Pattern extraction</td>
<td>d, xw, yw,</td>
<td>S</td>
<td>22</td>
<td>32</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>xdiv2_reciprocal</td>
<td>S</td>
<td>22</td>
<td>32</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>ydiv2_reciprocal</td>
<td>S</td>
<td>22</td>
<td>32</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>world,local,para</td>
<td>S</td>
<td>21</td>
<td>68</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>aout</td>
<td>S</td>
<td>21</td>
<td>70</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>work</td>
<td>S</td>
<td>22</td>
<td>32</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>ap, p, pbuf</td>
<td>S</td>
<td>21</td>
<td>70</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>a(array), b(array), c(array)</td>
<td>S</td>
<td>21</td>
<td>70</td>
<td>92</td>
</tr>
<tr>
<td>Template matching</td>
<td>patt_pow, data_pow</td>
<td>S</td>
<td>12</td>
<td>15</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>sum2, max</td>
<td>S</td>
<td>24</td>
<td>15</td>
<td>40</td>
</tr>
</tbody>
</table>

Table B.1: Overview of the required bits for fixed point variables in the marker detection algorithm