POLITECNICO DI TORINO

Corso di Laurea in Nanotechnologies for ICTs

Tesi di Laurea Magistrale

Implementation of neural networks systems on FPGA for feature detection

Relatore
prof. Maurizio Zamboni

Correlatore:
prof. Carlo Ricciardi

Studente
Andrea Petrini
matricola: s251476

Supervisore aziendale CSEM
Dr. Petar Jokic

Anno accademico 2018 – 2019

Abstract

Nowadays Neural Networks are fast developing for a large number of applications. In particular, the detection of features inside images is an extensively investigated task for its many purposes. Inside each image the network can recognize patterns, features and events that may be used to automatically monitor the field of view, as well as taking decisions without the intervention of an external human operator. For fast detection purposes, high speed cameras are usually employed, which can generate a huge stream of data in a very short time. The information stored inside these bits may exceed the transmission capabilities of state-of-the-art systems. In order to apply the benefits that a Neural Network can supply to a camera system without having to compress the information, FPGA-based circuits have been developed, which can faster emulate the behavior of neural algorithms requiring less space and power.

While the solution to integrate such features on FPGA already exists, each camera system presents specific hardware implementation which needs to be combined with the existing characteristics of the network. Moreover, the network response to incoming data can be used inside the device to develop new features. The work of this thesis consists in extending the existing FPGA framework in order to include new functionalities related to the Neural Network embedded block to the system. A new technique to feed data to the accelerator has been proposed. With the incoming data, a new block has been designed inside the framework to state whether a significant change has occurred inside the field of view, with the possibility of using this signal to trigger data acquisition.

During the work, a review of the existing sensor interface has resulted necessary in order to proceed further with the hardware on disposal. This part consisted in adapting the project design to a different model of FPGA in use, which has been successfully included inside the latest version. As far as it concerns the data processing, much care was spent to respect the preexistent characteristics of the framework, which presented an already functioning apparatus able to capture images using the VIA1M CMOS sensor developed inside CSEM company. In order to extend the functionalities, the thesis has also focused on understanding the previous work done, trying to implemented the new features as compatible as possible with the already included framework.

Acknowledgements

I would like to thank my thesis supervisors at Politecnico di Torino, Prof. Maurizio Zamboni and Prof. Carlo Ricciardi for their help and constant advice during the thesis. Furthermore, I thank them for being such good teachers and for having provided me with the best knowledge possible on the topic I am discussing in this document. A special thank to my supervisor in CSEM, Dr. Petar Jokic, who gave me the possibility to work in CSEM during my internship and assigned me a very interesting problem to solve during my thesis work. The months spent in Zurich, in every sense, represented for me a very significant change, from which I learnt a lot, both personally and professionally.

Finally, I would like to thank my parents, who supported me even during my worst crises, in these five years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This work presents the result of the implementation of a Neural Network system on a high speed camera namely the Fasteye model, developed inside the CSEM (Centre Suisse d'Électronique et Microtechnique) company. Most of the activity has focused on developing new blocks on the FPGA (Field Programmable Gate Array) chip on the board to add new functionalities to the framework. An embedded Neural Network was synthesized on the FPGA using the solution in [1], which was extended to match the new characteristics of the device.

The aspects studied in this thesis have great practical relevance, considering the numerous applications for high speed cameras in such fields as automotive, transportation, industrial manufacturing, consumer electronics, food recognition, entertainment, media, and sports. This is confirmed by the trend of the market of fast cameras, which is expcted to dramatically increase worldwide in the next few years [2].

Below, we first provide a short introduction to Neural Networks, then we discuss the innovation brought by high speed cameras and the technology beyond them. Then we provide an overview of the project, which has been the focus of this thesis work, its main objectives, and the methodologies that have been adopted to address the main challenges imposed. Finally, we detail the contents of the thesis.

1.1 Neural Networks

In this section, some background information on neural networks and on their application within a camera system are provided. Neural Networks (NN) for feature detection in images have been extensively developed for digit recognition [3]: Arabic characters displayed in a low-resolution picture get analyzed by means of a layered structure, which is intended to discover specific patterns for identification. Other works have designed networks capable of detecting more elaborate features, even human faces and expressions [4][5]. Industrial applications have been examined in [6] [7], specifically in the fields of automation for the purpose of fast defect assessment. Such application could possibly save hundreds of thousands of dollars. The computational architecture used in these works is based on the concept of Convolutional Neural Network (CNN), a multilevel, complex structure [8]. Each layer is composed by a sequence of single cells, which can state with a certain accuracy degree if a certain feature (presence of straight lines, edges, dots, ...) is present in the picture. The larger the number of layers, the more complex will be the features identified by the network, since at each step simple patterns get mixed with others, creating a complex structure. The output will always show a certain approximation, but with very extensive training and wide enough networks it is possible to obtain basically perfect results from the classification process. This process can be extended to any kind of feature present in the picture, since by changing the connections and the values inside the network, this can be applied to any purpose. The basic structure of a common NN is displayed in Figure 1.1.

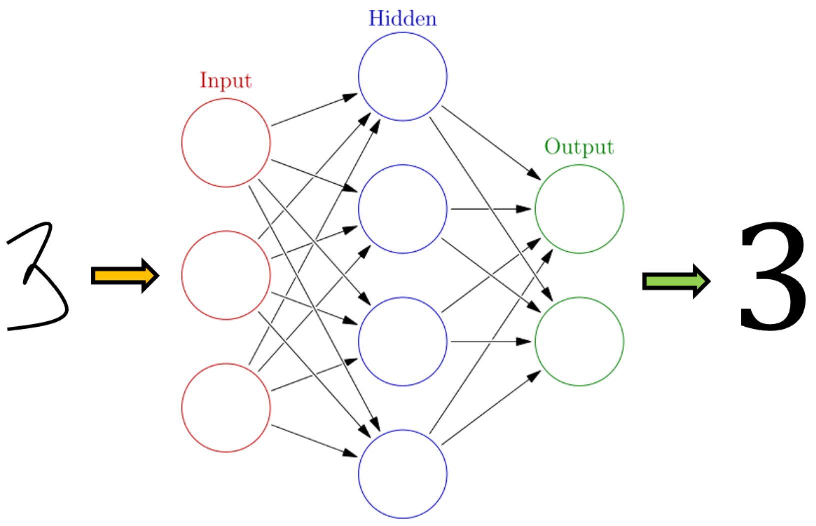


Figure 1.1. The basis structure of a NN, with an input layer, several hidden layers to add depth to the computation and a final output layer to extract the result of the calculation.

Nevertheless, the procedure requires sizable resources, and gets more and more complicated as the image size increases. GPUs are usually employed for this purpose, this implies that data must be continuously transmitted to the computational part in order to obtain the result, which is not always possible. For large streaming of data, the acquisition process may be too fast for the system to keep up, resulting in a loss of important information. Many efforts have been spent to solve the issue, either by increasing the transfer speed or simplifying the network, but the most promising solution relies on edge computing. Processing information at the source outputting only meaningful information instead of raw data is the ultimate strategy to maximize the throughput of the process without losing any advantage. Clearly this method requires further optimization of the embedded system, since in this case the camera (or a generic sensor for image acquisition) must be provided with an integrated system for feature recognition.

The challenge presented is to match the characteristics of edge processing with the limitations of the integrated system, which present only a limited amount of space available and power consumption to instantiate all the required assets for computation. In these systems, the usual approach is to use an FPGA chip to create the netlist (nodes and electronic components) to process and transmit data from the sensor, which could then be exploited as well for inserting the computational system of the neural network. Many efforts have been spent through the last years to identify a feasible way of inserting all the functionalities of the NN into a configurable chip, and a viable solution has been lately proposed [1].

A new popular method consists in using a BNN (Binarized Neural Network) as FPGA accelerator to overcome the issue related to the limitation of the framework. Usually the computation exploits floating point data to increase the accuracy of the result, but it is possible to show that, even with binary values, the final outcome can be quite reliable. Several assumptions have to be taken into account, but when the obtained output can still be considered accurate when checked against expected results. The idea of substituting weights and activation functions with much simpler expressions must be matched with the framework requirements. The architecture is generated through a Python code which takes as inputs the number of layers and the number of cells per layer and produces as output a synthesizable implementation on RTL.

Given these conditions, the previous work has been extended inside CSEM project in order to obtain a more general instantiation of the components. The system which has been integrated inside the camera can, in the latest implementation, process floating point numbers with the same architecture given in [1], which improves the flexibility and accuracy of the system. The framework created is so peculiar for its flexibility and space required, but this must be included inside the camera system, in order to correctly process the incoming data. This means that an adequate interface between the preexistent design and the newly added feature must be as instantiated looking to the current state of the framework, with the following requirements:

- NN parameters (weight, biases) must be transferred to the FINN accelerator from outside: since the network is only computational, it cannot be trained and therefore must be fed with the correct values obtained with an external trial.

- Sensor data from the actual system must be processed in order to correctly transfer those values to the accelerator unit in the standard format, specified during the creation of the synthesizable code.

- The result must be wisely acquired and processed, since it is application-based and must be used inside the framework to trigger the acquisition process or another output which can be designed.

1.2 High speed cameras

The human eye is unable to distinguish the transition between two consecutive images in a stream when the two are displayed in fast succession. At the beginning of the cinematographic era, the frame rate (or fps, frame per second) for movies was set to just 24 fps. This frame rate was high enough for the images to appear as a video to the human eye. As far as it concerns the maximum required time lapse not to perceive the transition between different pictures, it seems that 50 ms is an accurate esteem.

For many purposes, recording videos or streams of images at a higher rate than 120 fps seems useless for what concerns the human vision capability. Nevertheless, many phenomena are indeed observable only at very high speed, faster than the eye can manage to perceive. In order to "see" these kind of events, electronic systems have been developed. Eyes take advantage of biologic structures to sample the light intensity and frequency, using this information to reconstruct an image of the surrounding world. On the other hand, an electronic chip exploits the property of materials like silicon to retrieve the same information using photogenerated charges. Certainly, no electronic chip is currently able to emulate the totality of the functionalities of the human eye, but at the same time it can surpass some of its limitations.

Surely, the restraints posed by the human eye can be overcome, but the camera cannot state the presence of specific features inside images or detect a certain pattern as an observer would. With such a continuous and fast image streaming, real time adjustments of camera settings seem unachievable, giving only a partial control on the system during the data acquisition. The use of special computational architectures, such as Neural Networks, which can be trained to emulate the human response for specific applications, seems a viable option. Many algorithms have been developed to autonomously analyze the content of pictures, and the results are automatically used for further applications. With this solution, the devices may take independently decisions without communicating with other entities, which is particularly advantageous for saving resources. With machine learnings techniques, it seems that the separation between human and electronic devices in terms of image recognition could be further reduced.

Nevertheless, even with the best algorithms available, the camera payload might result too abundant for the rest of the system to handle. Modern devices can provide 10's of Gb/s of data [9] that, at the current state of the transmission technology, cannot be sent with the same rate to an external receiver. It is still possible to provide some additional memory to the boards, but this can only store a limited number of pictures before getting full. At the same time, due to matters like privacy rights, collecting pictures from camera in specific area may result in the violation of norms. The solution to these issues consists in keeping the Neural Network on the camera itself, outputting only the result of the computation. Exploiting this method, there is no necessity to bring all the data outside the camera itself, given that the same application performed on the images with an external hardware can be implemented on the camera.

Common imaging sensors exploit two different technologies for the chip architecture: CCD and CMOS, which will then be presented in more detail.

1.2.1 CCD cameras

These devices can provide information on the intensity of a light source by measuring the photogenerated charge inside a doped semiconductor material. On the surface of the chip, MOS capacitors are designed in triplets, which are intended to create regions inside the semiconductor where the charge can effectively be stored for short periods of time (depending mainly on the semiconductor characteristics and the capacitor leakage current). During exposure time, during which the objective shutter remains open, only one of the three capacitors can effectively store the charge, being the terminal voltage opportunely set. Once the sampling has been completed, the charge is then moved through the pixel rows using multiple clock signals, synchronized so as to output the intensity result. The charge is shifted from the pixel position to the collection border of the chip, where it is sampled using digital-to-analog converters (DACs) and sent to the application board for data processing. According to the characteristics of the device, different wavelengths can be investigated changing the geometry and the material properties. Moreover, it is possible to add color filters on the surface of the chip in order to let only a certain range of frequencies being sampled by a single pixel. In this way, colored images can be obtained without massively changing the device.

1.2.2 CMOS cameras

The latter category of imaging sensors is still realized with silicon technology, but it fully exploits the potentialities of the CMOS architecture. The photoactive element is a photodiode which collects the photogenerated carriers. This changes the value of the voltage through the diode, which is then amplified using a source-follower configuration. This means that normally more transistors are required for a single pixel. Moreover, each pixel is provided with a reset transistor, which connects the diode directly to the power source node and a row and column selectors, for the readout. Many configurations are possible for the CMOS standard cell, but usually these all comprehend the photoactive element, an amplifier and a reset transistor.

1.2.3 Comparison between camera technologies

When comparing the two technologies, it appears that the CMOS sensor requires more area than the CCD one for the same number of pixels. Also, the insertion of several transistors and an amplifier inevitably increases the noise of the output signal. Nevertheless, the readout mechanism of CMOS architecture lets a single DAC converter to be put in each pixel, which greatly fastens the readout of the image. This is the main reason why for fast cameras, the CMOS technology is the predominant one. Nowadays most of the disadvantages with respect to CCD sensors have been overcome [10], a difference between image quality is no more noticeable as in previous sensor models.

State-of-the-art systems with the ability of recording more than 1 thousand kiloframes per second (more than 1kpfs) have become common use. These can be exploited for many different applications, such as high-speed imaging in fluids [11] or biological events detection [12], but these may as well be used for industrial purposes [13].

1.2.4 Fast camera electronics

To be considered a fast camera, one system must display at least the following characteristics:

- A frame rate greater than 250 fps;
- an exposure time lower than 1 ms.

To match these requirements, it is mandatory to run the system with high speed, MHz clocks, in order to process the whole image in a short fraction of second. This is the reason why processing architecture for image data manipulation needs to perform many parallel, pipelined computations, in order to increase the elaboration throughput to match the requirements. The number of pixels a single chip may possibly contain can reach millions of units. All of these must be processed and eventually stored in a fraction of seconds, which means that a fast readout mechanism of the sensor is needed. Furthermore, the interface between the sensor and the rest of the application board must be able to consider and avoid possible lags in the sensor output. Usually, in order to achieve larger outputs, the data from the chip are sampled twice per clock cycle, with a method called Double Data Rate sampling (DDR). In this way the throughput of the chip is doubled as well, but the rest of the system must then be able to handle more data than normally. This can be achieved, as mentioned previously, by parallelizing the dataflow. A further trick to simplify the system is to divide the framework into different clock domains, each controlled with a clock signal with a specific frequency. The sensor clock will be assigned the higher valued, while for the rest of the system a slower clock will let the computational parts on the application board more flexible. Nevertheless, when crossing the border of two clock domains, some of the data might end being mistaken due to the oversampling of the signals. Specific synchronization blocks must be added to ensure that whenever a new meaningful data cross the border, the next clock domain receives the correct information.

1.3 Project description

The overall camera system, on which the thesis work has focused, is depicted in Figure 1.2. This is the Fasteye model, designed from CSEM company. The image (or video) captured by the sensor is transferred to the FPGA board which processes the data and feeds them into the external, main memory. From the latter the data are then moved through the USB interface toward the USB connector, and eventually to a peripheral device (e.g., a laptop). The main aim was to create a reliable, generic interface between the sensor interface in Figure 1.2 (hereinafter also referred to as application) and the FINN accelerator block within the Datapath, adding new features. The addition of a neural network to the camera is motivated by the possibility to create an extremely fast feature recognition tool for images and video recordings, which allows for interesting applications.. The aforementioned interface is intended to feed the FINN with meaningful data for furhter elaboration.

As an additional goal, the work aimed at making the existing k325-mercury enclustra-based framework compatible with a simpler FPGA system (k160-mercury board), adding new features and without losing the preexistent functionalities. A version for the k160 board had already been developed, but with less features. The final purpose of the project has not been fully achieved though, since some components still need to be included and properly tested inside the framework.

The camera itself, which had been developed by the CSEM company during the previous years, was characterized by a good percantage of unused FPGA instances. This represented a major opportunity to add new functionalities to the system. With the introduction of the FINN accelerator for fast detection of peculiar features or events during the acquisition

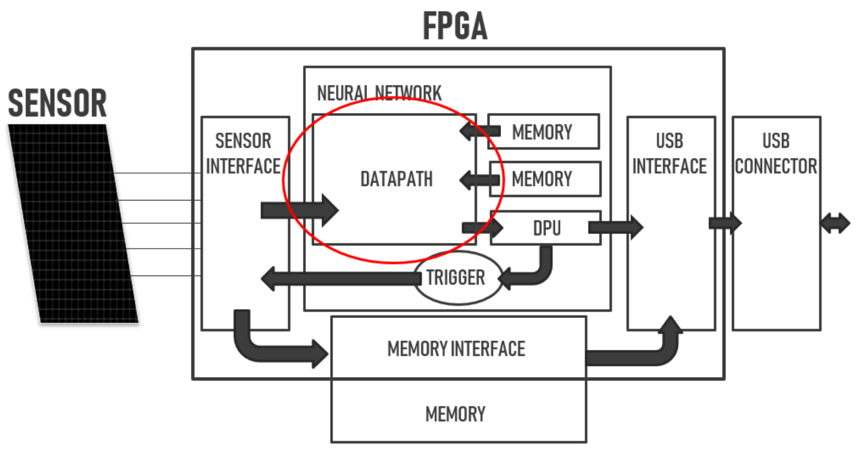


Figure 1.2. Camera system: the red circle highlights the part of the system on which the work focused.

of images or video recordings, now it is possible to execute on camera specific recognition algorithms. The content of chapters 2 and 3 will describe in detail the following steps of the work:

- adaptation of the k325 model to the k160 model was performed;
- the difficulties found in the above migration and how they have been overcome;
- the new average sampling functionality that has been added to the accelerator block in order to make the system more flexible and powerful;
- the introduction of a PWM generator so that an external peripheral device can be controlled by the accelerator output;
- the setup for a possible demonstration, with the aim to show the operation of the PWM generator;
- the testing of the developed solutions and the obtained results;
- the future work that would be needed in order to complete the aforementioned tasks.

1.4 Objectives of the thesis

The objective of the project consists in developing an FPGA-based Neural Network on a high speed camera. The purpose of this innovation is to have a system that:

- Processes thousands of images per second.
- Generates an output based on the application for which it was trained.

- Fits inside the preexistent model of camera, using the space left unused on the FPGA.
- Can be easily reconfigured for different applications as soon as the parameters of the system are modified.

In order to achieve these objectives, the following challenges have to be faced:

- The architecture must consider the large amount of data and the limited amount of time to perform the required computations.
- Load inside the internal memory of the chip the weights and biases obtained with a training consistent with the application the system will tackle.
- Use an appropriate model of FPGA able to contain the necessary circuitry.

1.5 Methodology

The first step was to acquire the necessary knowledge on the characteristics of the framework available at CSEM, which was the object of this thesis. To this end, the initial scheme of the system has been analyzed in detail. To access the schematics, we used HDL designer tool from Mentor Graphics, which helped visualizing the dataflow of the device. A preliminary phase to test what was already implemented resulted necessary to fully analyze the characteristics of the device. During this phase, the work leveraged ModelSim (v. 9.1) to run the scripts referring to the VHDL implementation of the project. Next, in order to fully understand how the camera worked, the existing hardware functionality (e.g., data acquisition and sensor configuration) have been validated, using a Matlab (v. 2019) script to initialize the parameters of the device. The code already implemented at each step has been properly modified whenever needed, to test different device configurations.

We then we had to develop some functions in order to create the interface between the sensor and the accelerator, namely, the function of averaging of the input to the accelerator and the trigger to acquire a stream of images. For each of the functions, we adopted the following approach for their implementation. First, an initial version of the solution was drafted, considering different possible implementations, their pros and cons, and choosing the optimal one, based on the characteristics of the device. The selected solution was then implemented using VHDL and other programming languages, which were particularly suitable for the purpose.

As far as the implementation of the interface as embedded blocks on FPGA is concerned, the task consisted mainly in the transposition of the algorithms chosen to realize the averaging and trigger functions in a hardware description, which could be synthesized by the tool. After the realization of the embedded part, a first validation was performed using an opportune testbench written only to verify the accurate transposition of the algorithm in the hardware description. This step was particularly important, because it allowed the detection of possible bugs or overlooks in the algorithm. A specific simulation was run on ModelSim to check the characteristics and timing of each signal, with appropriate test inputs intended to emulate possible data coming from inside the design.

Once verified that the embedded block performed as intended, the next step consisted in introducing this inside the design. Due to the particular requirements in terms of parallelization, data acquisition and timing, new constraints for the algorithm needed to be considered. Again, during this phase, many different parts of the project needed to be adjusted to introduce the new part and/or functionality. Particular attention was paid to assess possible conflicts between the implemented algorithm and the existing design. In order to facilitate the insertion of new blocks, the HDL designer tool showed to be particularly handy, as it allowed us to handle basic blocks inside the schematics. By generating automatically higher hierarchy components, it was easier to conclude the task without inadvertently modifying other parts of the design.

A second validation phase of the new framework was performed after the inclusion of the new functionalities. This step regarded the generation of the code describing the new hardware and the modifications to the simulation script in order to take into account the new parts. Inside ModelSim environment meaningful signals were checked to verify the intended behavior and timing, with respect to the existing part of the circuit.

After having validated the system through simulation, the circuit was then synthesized on the FPGA chip mounted inside the camera. During this step, some errors due to an incorrect timing assessment report and resources utilization rates must be addressed, in order to match the new design with the characteristics of the device. In some cases, this led to a reconsideration of the implemented algorithm itself, which could not be properly synthesized on the board.

The last step was to load the synthetized design on the device. Using a MATLAB script, the chip was configured as intended and the system started acquiring data, according to the configuration settings. During this phase, the MATLAB script had sometimes to be changed to include the new functionalities adopted in the framework. These were then tested to verify their behavior on camera. If case of any error occurred during acquisition, debugging using MATLAB commands was performed to point out the origin of the flaw.

1.6 Organization of the thesis

The thesis content is organized as follows:

- Chapter 1: Introduction
  A state-of-the-art description of the current technologies in the field of high speed cameras and Neural Networks is presented. The objectives of the project is presented, as well as the methodology adopted to obtain the final results

- Chapter 2: Design and Implementation
  This chapter introduces the main work, what has been implemented on the camera and how. First, issues with matching the current FPGA model with the synthesized project are shown and how they have been solved. Then in the implementation part, the algorithm to elaborate a change trigger embedded block and a compression averaging system for the Neural Network are described. The algorithm of each component is represented with a flow chart. These components have been inserted inside an acceleration block, which also comprehends the Neural Network instance. Eventually, a servo controller system to interface the Neural Network output with a servo motor is described, which is intended to move accordingly to the result of the computation of the accelerator.

- Chapter 3: Evaluation and Testing
  The third chapter shows the results of the implementation work on the camera. The averaging feature is first investigated, showing the input data buffer content of the accelerator. The image is scaled down to the size accepted from the accelerator with different modes for different purposes. The results from the FPGA synthetization are then reported to highlight the differences in terms of timing and utilized resources produced by the new features introduced. Last the servo controller functionality is shown, describing the work to connect the embedded block to the peripheral device.

- Chapter 4: Conclusions and Future Work
  A short recap of the objectives and the achievements of the project are presented. Each step of the work is analyzed in terms of what has been done and what has still to be fully developed. A final description of possible future steps to improve the introduced functionalities and complete the work is then presented.

Chapter 2

Design and Implementation

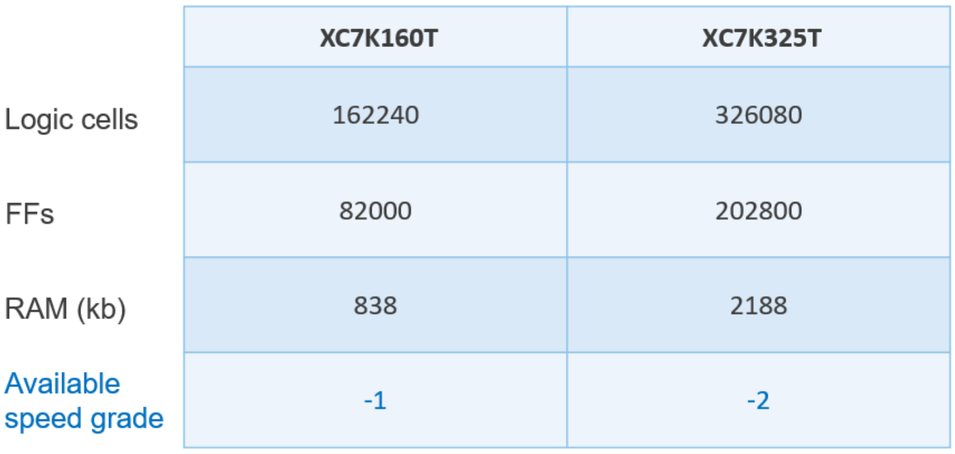
In this section, the characteristics of the FastEye system are first introduced (Section 2.1), highlighting the differences between the k325 and the k160 boards. It is also described the analysis of the internal memory interface block and the application one, aimed at verifying whether the integration issues are related to this part of the system. Then, in Section 2.2, the accelerator unit is presented, along with its sub-components, namely, data preprocessor, change trigger, and eventually the servo controller in Section 2.3.

2.1 The FastEye system

The Fasteye camera is essentially composed by the image sensor VIA1M developed in CSEM, which is capable of producing 1Mpx images with a 2kfps (2000 frames per second) rate, a Xilinx XC7K325T (or XC7K160T) FPGA for data processing and an external memory, namely a DDR3 SDRAM, which is intended to store images during fast mode acquisition, being thedata streaming channel too narrow for real time acquisition through the USB connector applied to the camera. Inside the framework it has been possible to implement the acquisition of single images, streams of images or short videos, according to the capacity of the external memory. The camera itself can be furthermore controlled through a MATLAB script which is used to set those parameters required to specify image dimensions, acquisition rate, etc.

Data from the sensor is acquired through a DDR sampling procedure, to be then further processed for memory storage. Many stages are intended to sort, prepare and count the bits which are sent to the GTM part, which contains the memory chip and interface. To the existent framework, the FINN accelerator for detection of specific features inside the images directly on camera has been added, which occupies the remaining space on the FPGA board. This structure, which is generated through a python script, can instantiate automatically a convolutional neural network for processing the images, analyzing internal patterns to figure out whether certain features are present inside the raw data fed to this. The new feature which has been added is the possibility to pass from a software-triggered system to an automatic triggering produced when the neural network part detects something along the image, which will then be recorded. This is quite important, since the memory only has space for about a second fast-speed video, so for long acquisition and sudden events, only certain portions of time, in which something meaningful happens, should be recorded.

Two different versions of the system exist, the original one has been developed for the k160 FPGA model, which is slower and smaller, and the k325, which contains more cells and therefore is more indicated to host the FINN accelerator, being that the large network requires larger spaces to be efficiently synthetized by the tool provided on the board. Part of the project has as well been spent for eliminating the incompatibilities between these two different systems, with the creation of a single script able to efficiently switch between the models without the need of changing part of the code.



	XC7K160T	XC7K325T
Logic cells	162240	326080
FFs	82000	202800
RAM (kb)	838	2188
Available speed grade	-1	-2

Figure 2.1. Comparison between main features of the two FPGA models, k160 and k325; the number of logic elements is presented, as well as the number of memory instances and the available speed grade of the silicon chips.

2.1.1 Main differences between k160 and k325 models

The two boards on which the FPGA chips are mounted present many similarities, they are both provided with a USB connector and an I2C communication block, but what really changes is the overall performance. The k325 contains twice the logic cells and flipflops that are inside the k160 model, plus it is also characterized with a larger RAM and BRAM. Moreover, the speed grade of the two FPGA's was different, marked with a -2 for the k325 model and a -1 for the k160 one. Such differences between the two boards are summarized in Figure 2.1. The aforementioned factors lead to two conclusions:

- large FINN networks which can be synthetized on the k325 board may not be feasible on the k160 due to the limitation of area available for the system;
- certain processes, which are correctly timed on k325 may result in creating a negative slack in the k160 model, which is slower and therefore subject to increased timing issues.

The latter of these two consequences represents the reason why, at first, the latest k325 project was not able to be run onto the k160 framework. When executing the MATLAB script for acquiring images with the camera, after a few captures the system abruptly stopped, ceasing the process and forcing the user to restart the whole apparatus. The reasons and the explanation on how this issue has been solved will be presented in the next part. For more information on the two FPGA models, it is possible to consult the reference datasheets [14].

2.1.2 The analysis of the memory interface

Given that the system execution stopped when trying to read out the content of the external memory, it was at first hypothesized that the problem was related to the memory interface, which was used to transfer data from the application part to the DDR3 RAM chip, or during the USB read out. Several tests have been performed using the MATLAB script to check different functionalities and debugging options, but none could find the cause of the system flaw.

After evaluating some possible options, instead of continuing trying to adapt the k325 version to the k160 board, it was decided to proceed with an opposite strategy: taking the old version of the code and updating the different parts to restore it to the newest version. During this phase, memory interfaces and control blocks have been substituted, looking for specific problems that could generate the issue. Some of the control registers had been modified, and so were some of the control signals which were related to specific functions.

The most important feature to check was represented by a small adaptation added to the memory buffer: the interfaces of the two systems were instantiated automatically from a predefined hardware schematics, which was provided from the manufacturer. Inside HDL designer, instances for both versions (k160 and k325) were inserted with a specific command that was intended to only generate one of the two components, since the dimension of the address connected to the memory buffer was not equal. Originally, the address input size was not generically defined and each time the project was run on a different hardware, its length needed to be adjusted accordingly. It was foreseen that this small inconvenience would have been easily fixed by adding the necessary library and reference to the signal in order to automatically switch between different sizes whenever required and adapt automatically. In theory, this simple procedure would not create any issue to the framework, but when the block was generated through the HDL designer tool, an additional error prevented the software from completing correctly the instantiation of the block.

The problem seemed to lay on the fact that the new framework, with the generic parameter included, was checking the matching of the input address signal with both the instances, which of course were characterized by a different length. Since one of those inevitably mismatched the actual address size, the generation process stopped abruptly, and no HDL description was created for the component.

In order to extend the framework to both versions, a small component was developed to instantiate the component as intended. Furthermore, this mismatch characterized both input and output signals, so a double solution was studied: since the two pieces required different inputs, in order not to add more signals to the higher hierarchy design, a mux-like structure was generated, which would have automatically taken into account the difference in sizes. From a single input, the component generates a pair of outputs with different length, each compatible with the desired value. The same idea was applied to the output, but in this case two different inputs were then routed to the same output. At this point, whenever the required address size changed, the system automatically instantiated a new component without displaying any error and without requiring additional modifications of the code. Figure 2.2 shows the schematics of the newly instantiated component.

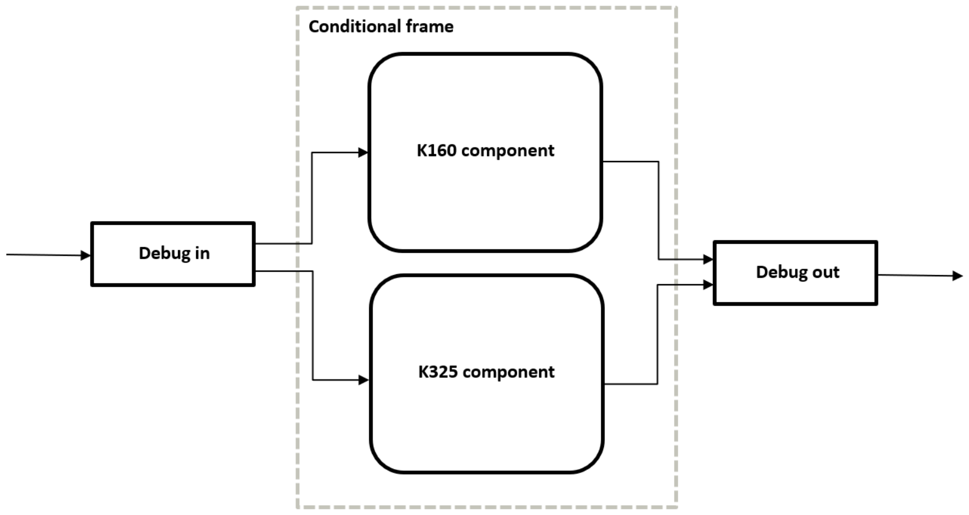


Figure 2.2. Final appearance of the memory interface related block: the system is characterized by a conditional frame, which is used to instantiate only one of the internal components according to the framework requests. The two small blocks on left and right side, called respectively debug in and debug out, implement the added features described in Paragraph 2.1.2.

This configuration was compatible with the k160 version, on which the testing process was correctly carried out, but when testing the components on the k325 version, it was not possible to further proceed with the generation, due to a bad assignment of the input/outputs sizes connected to this part. Unfortunately, due to lack of time, this part could not be investigated more and currently requires completion. However, the idea was still adopted for the k160 hardware, in which the implementation was inserted, simulated and tested, showing the same results as before.

After updating all the instances, when checking both the simulation and the actual implementation of the system, the old k160 project with updated memory-related parts was still perfectly working as expected. Since no other possible inference could have derived from this part, it was decided to move on to the rest of the framework, in order to check the remaining possible causes.

2.1.3 The application analysis

The remaining portion of the system consisted in the part assigned to the organization of the data flow from the sensor to the memory. Information was sorted and arranged to be correctly stored, passing inside different clock domains.

One of the first tries consisted in reducing the clock speed related to the sensor part: longer periods between clock edges would have given signals more time to settle, obtaining a better worst slack parameter. While the k325 could be overclocked at higher frequencies showing a small negative slack, the k160 version was run at lower speed, with the same effect. Since the clock frequency should not have been reduced further, at the minimum acceptable value the slack was still negative, which could have compromised the operation of the camera.

This factor however was in the first place overlooked, since more promising causes were examined first (since the fastest hardware could run even when showing a negative slack). Parts were updated to the latest version to check if any change with respect to the old project could be the cause of the failing mechanism due to a desynchronization in the data path. The piece which regulated the acquisition of data from the sensor was examined more in details, since the actual implementation was characterized by a series of embedded components which had been inserted just to adjust the flow of information from the VIA1M chip.

When tested these changes eventually presented the same error which characterized the 325k model, since in this case the image acquisition abruptly stopped. After some trials though it was evident that the reason why this happened depended on desynchronization of the sensor data, a separate reason with respect to the cause that was searched. At last, the timing report of the synthetization was analyzed and the real problem was discovered.

2.1.4 The frame_mem block

Inside the sorting path that connects the sensor to the memory controller block, a specific part of the design called frame_mem is used to sort out the data packets which arrive from the previous portion of the elaboration system. Its behavior can be summarized as a line 1024-pixel wide buffer which stores the incoming data through a specific algorithm, able to rearrange the pixels order to be better inserted and processed by the memory. When analyzing the code from the two different systems, it was noticed that these only matched in part. In the k325 case, which could not be synthetized on the k160 model, the sorting process was clocked, while in the other case its behavior was related to some assessment signals, which were intended to synchronize the whole dataflow.

Since the algorithm basically store a pixel value in the buffer according to its position in the packet and in the packet number (which is the number assigned to the 160-pixel wide input of the block), a very large multiplexing entity is required to correctly move the pixels into their correct position. The first attempt consisted in changing the synthetization strategy of the tool. In fact, VIVADO lets the user choose some settings to generate the synthesized design from the HDL code. For all the previous runs, the standard option was activated, which could have been the fastest, but also a possible optimization route. When analyzing worst path characteristics, the it was discovered that most of the time from one clock edge to the next was mostly due to the interconnections between instantiated components. Being the time required to transfer a signal proportional to the fanout of the line, this number was checked for the longest paths and it was found it would normally exceed the value of 300. Being so high, limiting this value would have inevitably led to a certain time save, which would have helped fixing the timing issue. The new implementation strategy with a maximum fanout limit of 50 was then included during the synthetization process to verify this hypothesis. The results indeed showed a decrement of the worst slack of several ps, passing from 140 ps to 70ps, still when tested the framework did not behave as expected. The implementation step, which translates the synthesized design into the final netlist, was already optimized through a retiming option, so modifying it would not possibly lead to somewhere.

Returning to the code, when the clocked procedure was exploited, only those buffer pixels which are interested by the specific packet number value will be processed, while all the others remain the same. In the other process instead, all buffer flipflops are evaluated, so that the ones referring to a packet number lower than the input one are kept the same, those referring to the specific value are changed while the others are set to '0' value. At first glance these two methods would not seem to present many differences, but when coming to synthetizing only one presents a worst slack compatible with the k160 speed grade. When looking more in detail into the question, we can assume that while the two processes have the same logic output for the system, they will basically produce two different outcomes as netlists. The k325, which is the most complex one, will produce a single multiplexing entity, with a large number of inputs and outputs. This intricated system will cause data to travel for a longer time, violating the setting time and leading the framework to an undetermined state, which explains why the execution of the acquisition procedure fails and remains in a blank state. On the other hand, the other script will first look to the value assumed by the packet number, and only after that it will update the correct values. This might implicate that, rather than a large multiplexer, the system contains smaller entities which are intended to compare the value of the packet number with respect to single definite values. When hitting the right value for the specific box, the inputs of the box will be the outputs that fill in the sorting buffer, while for all the other cases the values will remain either the same or '0'. When looking to this configuration, we see that only smaller, independent blocks are required, which might explain why the synthetization of these entities completely avoids having bad timing constraints.

The final solution was to update the k325 for the frame_mem block with the process used for the k160 one, which was unclocked. In this way, the final result is a framework which can be run on both models of FPGA's, without the requirement of changing the script of the block (for what concerns the frame_mem block at least). Though this solution works finely for the k160 version, it still presents some incompatibilities with the k325, which cannot be synthesized with the new blocks in the design.

As it is possible to see notice from Figure 2.3, in the first case (the slowest) the system makes use of a large mux-like structure to sort the data into the chosen buffer input configuration, while in the fast design the synthetization tool is supposed to arrange different evaluation blocks which update some buffer positions according to the packet number.

2.2 The accelerator_unit block

The scheme of the accelerator unit is the following (the red parts have been added or modified by this project):

- Parameter feed (untouched): this component is just required to fill the accelerator with data from memory for result computation, but it is also needed to read out the results;

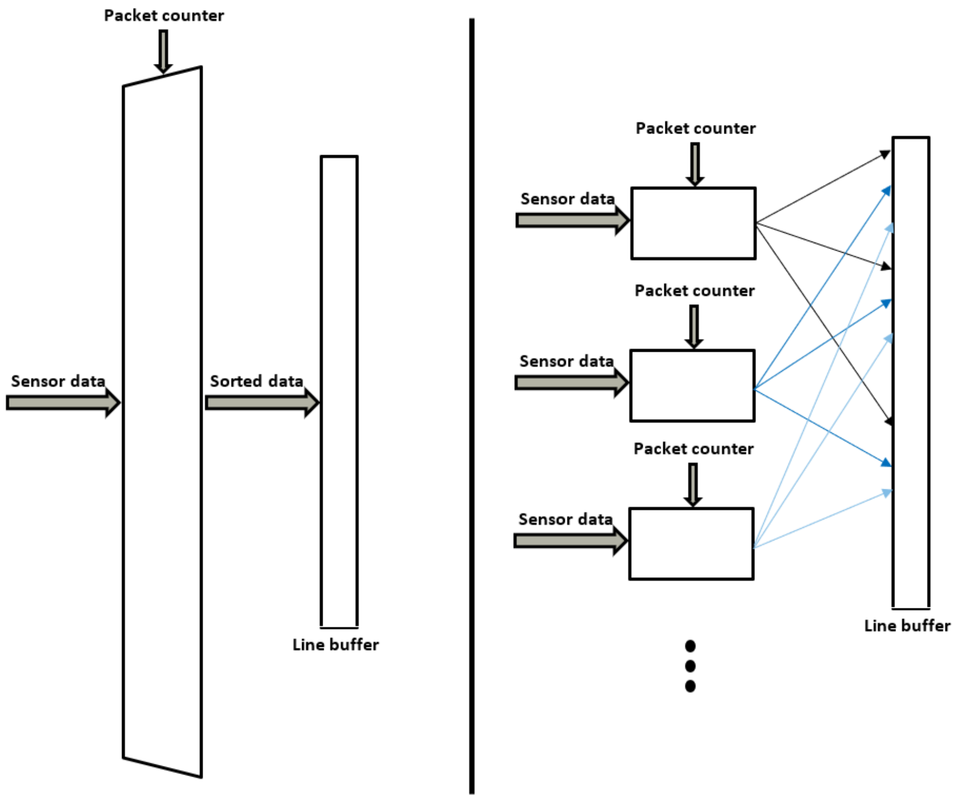


Figure 2.3. An explicative sketch of the single multiplexer entity on the left with the distributed structure on the right.

- Data preprocessor (modified): the component is the core of the interface, its purpose is to receive the pixels values and to produce a reduced-size image that can be successfully read by the accelerator;
- FINN accelerator (untouched): this block contains the netlist which implements the neural network structure created with the algorithm to satisfy the performance required;
- Change trigger (added): the component is intended to state when a certain object has entered the field of view, generating automatically an output which can then be read by the system (as described in the following chapters, this has not been tested inside the framework);
- Trigger multiplexer (added): in the latest version of the project, the accelerator unit block could create an assessment signal which could trigger the acquisition of data from the memory. This functionality has been integrated with the possibility of acquiring the trigger generated by the change between two images, which can then be sent to the trigger control part to extend the framework possibilities.

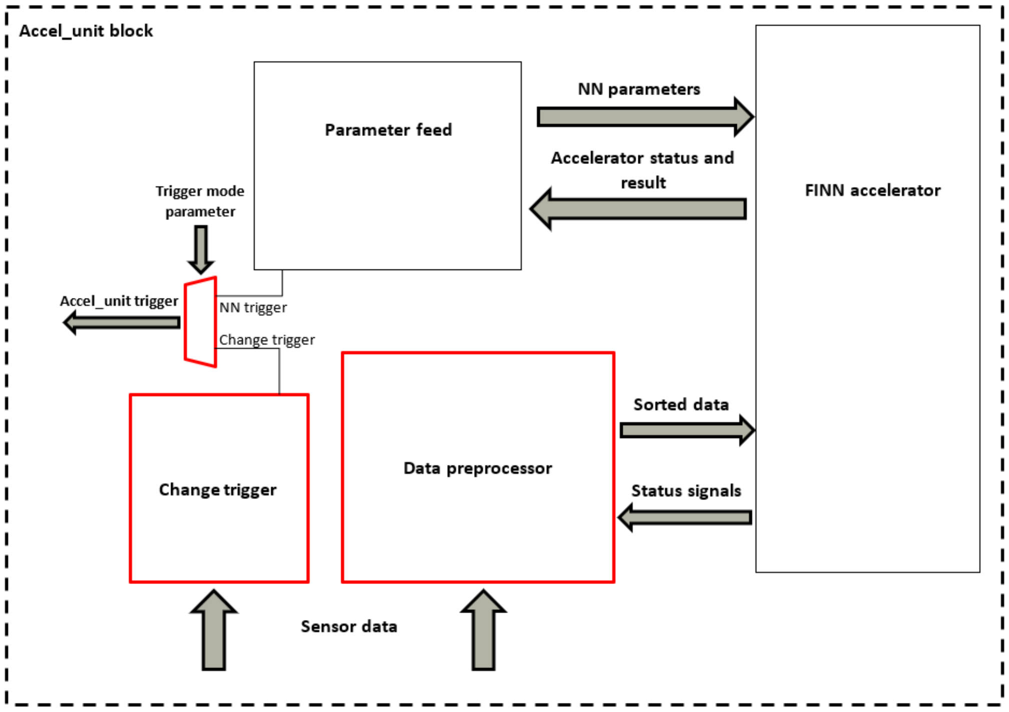


Figure 2.4. Structure of the accelator_unit block.

Figure 2.4 depicts the structure of the accelerator_unit block. The next sections will explain more in detail the characteristics of each part, which have been modified, in order to show its behavior and capabilities.

2.2.1 Data preprocessor

The purpose of the data preprocessor is to create an NxN array of pixels, where N is a configuration parameter for the accelerator; such array of pixel can then be evaluated by the neural network instance. Since this value is arbitrary, some expedients have resulted necessary to allow N to take any positive value. As for as the content of the array is concerned, two different modes for reducing the size of the input have been developed. The first performs a subsample of the image, the second divides the picture into tiles and reduces each of them into a single value by calculating the averaging of pixels inside the tile. My contribution to this section is limited to the design and implantation of the averaging mechanism. I also added a short description of the other features present in the block to better clarify its behavior for further discussion. The subsequent parts describe in more details the main functionalities of the data preprocessor.

Image subsampling
The 1Mpx image (smaller if using the Region of Interest - ROI - option) must be reduced in dimensions in order to fit the required size for the accelerator input data. A first process just takes some pixels from a few rows. In this way the output may not be able to correctly represent certain features present in the original image. This holds especially in the case

where such features are related only to a small portion of it, but also may have some glitches due to the fact that the change of a single pixel may produce a noticeable effect on the final output. Nevertheless, this procedure is still capable of providing useful data, with a certain accuracy, and is much easier to implement on hardware, with lower computational cost and fewer required resources. This feature was already implemented in the previous version.

Output addressing

The data preprocessor receives at the same time data from bottom and top rows of the captured picture. In order for the accelerator to correctly organize the image, each pixel is assigned with an address, which represents its position inside the image, sent along with the data values. This process will not be explained in detail, since it has only been slightly modified with respect to its original version, to include the possibility of sending data obtained from the averaging process, rather than only those acquired through subsampling. This feature, as the previous one, was also already present in the previous version. The next parts will describe the averaging system, which has been implemented along with the subsampling, with the aim to extend the functionalities of the block.

Image averaging

The input row size and number are two variables which are defined through generic parameters inside the code, as well as the required output image size. Usually, the number of rows and columns are not the same; furthermore, these values are not required to be a multiple of the output size. Thus, before starting the averaging calculation, some input preprocessing is needed in order to simplify the next steps, described below. A flow chart description of the image averaging procedure is represented in Figure **??**.

- The input image is squared, since in the averaging process the number of pixels per tile must be constant. In order to make the original image squared, the minimum between the number of columns and rows is calculated (usually inside the system the former is greater than the latter). Then the input size, which represents an entire row of the incoming image, gets modified so that it takes a portion whose size equals the one of the minimum obtained so far.

- The column/row number of the obtained image is a number which rarely can be perfectly divided by the required output size. The solution is to further reduce the dimensions of the image by cutting out the rows from the bottom and the top which represent the remainder of the division, as well as part of the columns to keep the square shape. For example, if the total number of columns and rows after the first step is 1024, and the final image side must be 26 pixels, this means that each averaged tile would contain $1024/26 = 39.38$ pixels in each direction, which is not easy to handle as a result. Thus, the solution exploited consisted in calculating the integer part of the division (in this case 39) and then in multiplying the value by the tile size in order to get a multiple value. Then by subtracting this to the number of rows and columns, the remainder of the integer division is found and its value stored for further use. In fact, this number will represent the starting line for the averaging process, cutting off a portion of the image edges. Furthermore, this value will be exploited to further reduce the input size excluding the most left and right pixels from the computation. The final result is exemplified in Figure 2.6

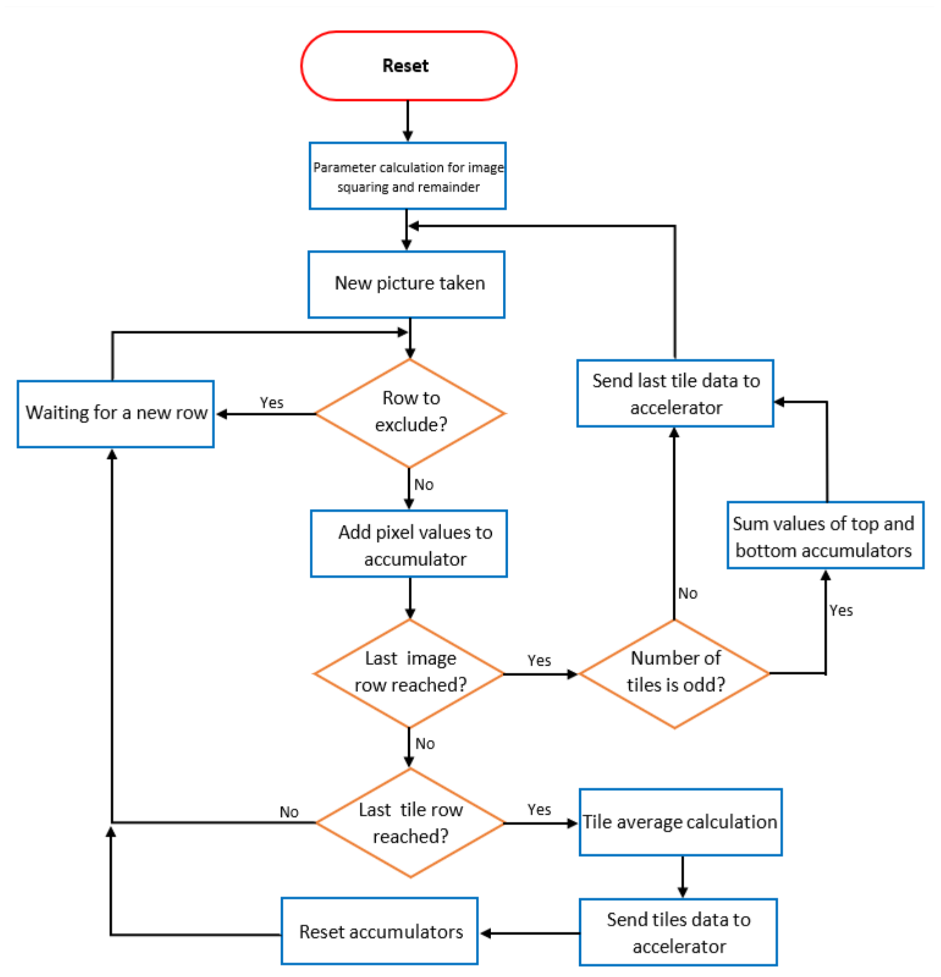

Reset
Parameter calculation for image squaring and remainder
New picture taken
Row to exclude?
Yes
Waiting for a new row
No
Add pixel values to accumulator
Last image row reached?
Yes
Number of tiles is odd?
No
Send last tile data to accelerator
Yes
Sum values of top and bottom accumulators
No
Last tile row reached?
No
Yes
Tile average calculation
Send tiles data to accelerator
Reset accumulators

Figure 2.5. Flow chart of the averaging process.

- Now that the image is correctly and easily processable, we need to consider a further aspect of this process. The number of rows and columns so far calculated will always be an even number, due to the characteristics of the system. Instead, the number of averaged tiles can also be an odd number. The reason why this feature is problematic lays on the fact that the block is receiving data from the bottom and upper rows of the image. With an even number of divisions, the two parts are handed parallelly without any interaction. However, in the case of an odd value for the output image dimensions, this is no further acceptable. The code must consider also this case, which requires to calculate the last row of tiles separately from the others. The solution consists in splitting the last tiles into the bottom and top section of the image, and when the tile counter inside the averaging process reaches the last row, the values of the top and bottom portions of the tiles are summed together and divided by the number of pixels per tile (see the example in Figure 2.7). This solution allows us to handle easily even odd values without changing the previous part of the code

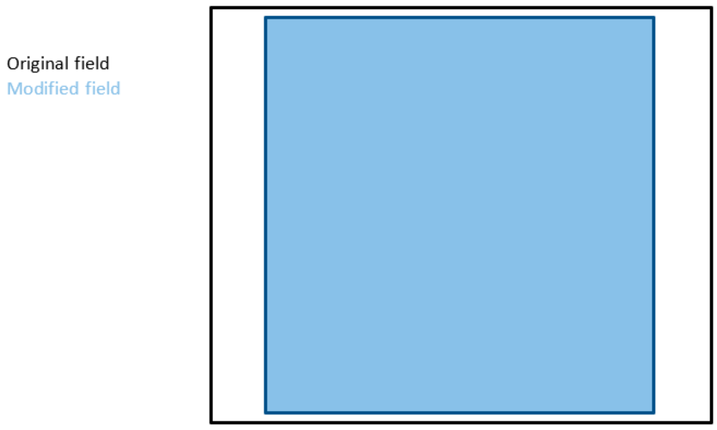


Figure 2.6. The original field, which is processed from the FPGA application, gets reduced into a smaller square, which will be then analyzed.

(preprocessing).

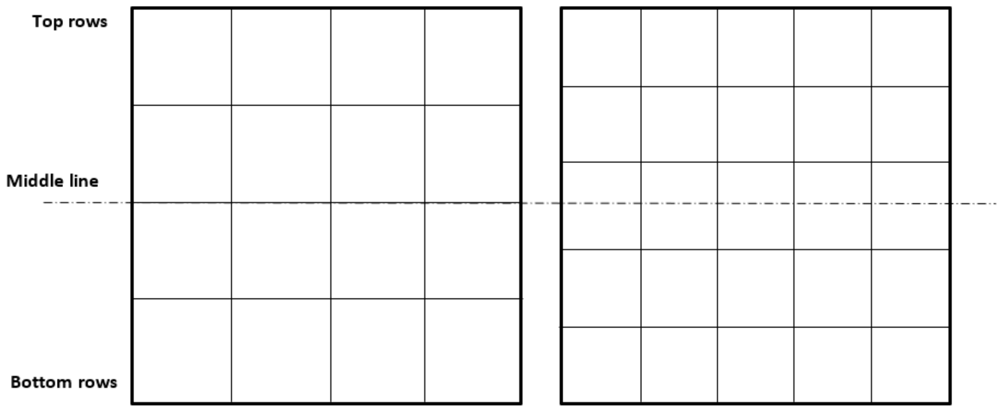


Figure 2.7. On the left side, an even tile number (N = 4) divided into top and bottom parts, while on the right one, the middle line of tiles gets crossed by the central row. In this case, since the tile number is odd (N = 5), top and bottom rows must be considered together to calculate the average of this portion of the image.

- The actual averaging process can now be introduced: the input of the block is constituted by an entire pixel row, which changes at each clock cycle. Given the previous steps, each row can be divided into a number of sections equal to the output size, with no remainder. Inside each portion, the most meaningful bits (chosen through another arbitrary parameter defined at the beginning of the code) of every pixel is summed up in an accumulator, one for every row section, every clock cycle. At the same time, a counter keeps track of the row number that is considered at each time, which is useful to determine when the end of a tile is reached. The averaging process can only start when the counter value is greater than the remainder one, since we are due to exclude the bottom and top lines from the computation for the reasons explained before. Every time the counter reaches a multiple of the number of rows per tile plus the value of the remainder, the accumulator value is divided by the number of pixels per tile and sent to the accelerator block, along with its peculiar address. The accumulator is then reset, ready to compute the averaging of the next tile.

  One of the most critical issues encountered during this step consisted in matching the timing of this circuit with the actual capabilities of the FPGA. During the first implementation part, the way to find the final row of each tile was to use the mod function, which calculates the remainder of a division in HDL. Since this function is expensive in terms of processing time, the final slack of this portion of the component resulted negative when implemented on FPGA. This produced weird effects on the reduced image, which are explained in more details in Section 3.1.

  In order to decrease the required time for exerting the operation, a new solution was found, which exploited a different approach. Instead of calculating the remainder, a small counter has been implemented to keep track of the number of rows which have already been considered in a single tile. When the value of the counter reached the total number of rows per tile (given by the result of the number of the rows in the reduced field and the number of tiles), this asserts a signal to state that the end of a tile is reached, and at the next clock cycle the counter is reset. In this way, the most expensive operation in terms of computational time was avoided, resulting in a better, positive slack with respect to the previous coding.

In this way, the output will always represent accurately the input image. However, due to persisting issues with timing, the image reconstruction of the accelerator buffer still presents some glitches, which must be fixed in future. In the design of the data preprocessor has also been included the possibility of switching between averaging and subsampling mode by modifying one assigned bit inside the control registers, adding flexibility to the framework.

2.2.2 Change trigger

The camera system contains an on-chip memory which can record a maximum of about 1 second streaming at full speed. This means that once activated manually, each event which happens after acquisition end will be lost. Also, some peculiar phenomena (very fast, but that can happen along a long time period) would be missed or would require multiple trials to be recorded. This is the reason why the need of a trigger which could automatically start the memory acquisition as soon as something in the field of view changes seemed a valuable feature to add to the design. The first phase required to define some specifications of the block, which are the following:

- Reliability: the block should assert the trigger only when recording a real change, not triggered by external reasons;
- Robustness: the system would overcome those little changes which may appear during acquisition (pixels LSB's, light changes, wind effects);
- Integration easiness: must take advantage of the existing features of the system to be easily inserted in the framework.

The idea underlying the component that was instantiated was structured this way (see Figure 2.8): a small memory stores a subsampling of the image inside the block, which is then compared with the incoming one to establish if any significant change occurred in the field of view; two adders, one used to compress the incoming data and the other to calculate the rate of change between the two images; a comparator, in order to assess the changes between the images; a simple control trigger, which enables the activation of the change assessment signal only after the first image has been acquired. The flow chart of the algorithm is represented in Figure 2.9.

Compression technique
The input for compression adder is constituted by the sensor data coming from the sensor interface. Therefore, the compression adder only takes portions of the single rows of pixels of an image. The pixel in a portion are summed up to compress the image data and save up space on the FPGA, however this method also increases the robustness of the framework: doing so, the effect of small fluctuations between adjacent pixels (like the movement of leaves on a tree, for example) is greatly mitigated. Only events that change considerably the subject of the image will be detected, indeed the background influence in this case is much less relevant. Moreover, it is possible to choose the number of bits to consider during this step, since it is possible to only take the most significant bits of the pixel values. This feature can result particularly useful as tradeoff between accuracy and required space: adjusting the quantity of resources destined to the component, it is possible to synthesize larger networks in the framework. The threshold, which will be discussed in the next paragraphs, would need to be adjusted as well, decreasing its value based on the number of bits per pixel and, consequently, the accuracy of the change detection algorithm.

After being compressed, the data are then stored in a small memory which collects the results of the summation. In order to further decrease the required resources of the block, it is possible to select, changing the relative parameter in the HDL code, the number of rows to sample. The algorithm skips a number of rows equal to N-1, compressing and storing only the following row. With this method the image can be further compressed, in order to reduce the space required from the component. Each time a new image is acquired

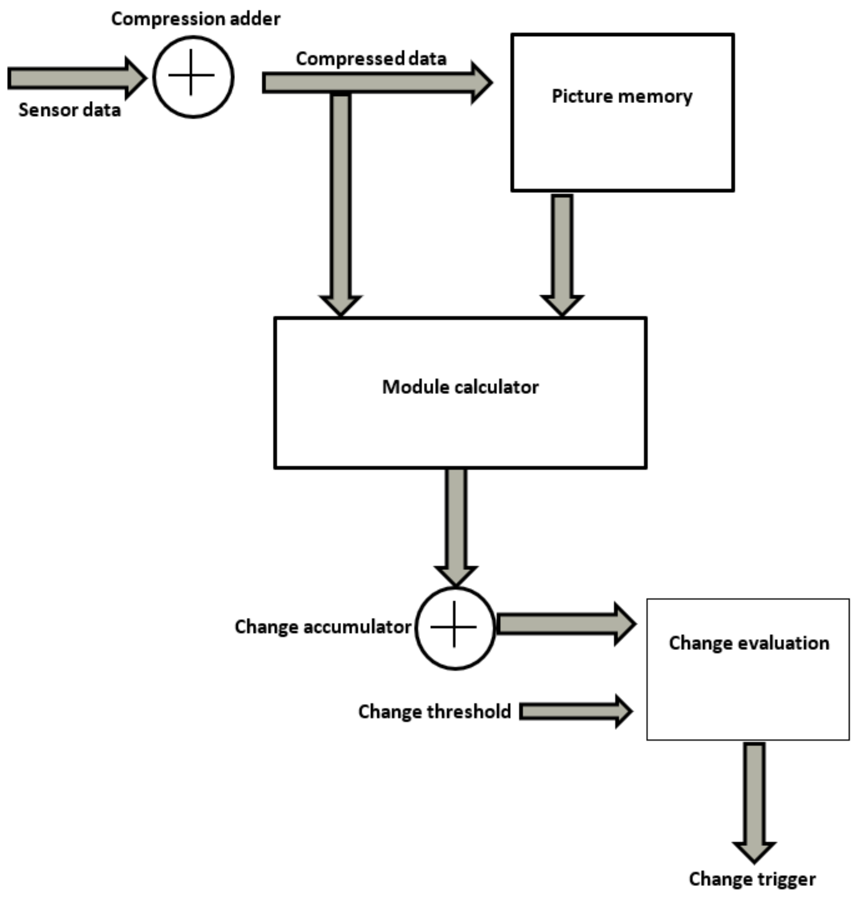


Figure 2.8. Block schematics of the change trigger.

(and evaluated), the compressed values substitute the ones stored in the memory, so as to only detect relevant changes between consecutive images.

Change degree evaluation
In order to numerically evaluate when a certain part of the image has significantly changed, the algorithm exploits a comparison mechanism between the previous picture and the newly acquired one. As explained before, each captured image gets compressed and stored in a small memory. The evaluation step consists in calculating the difference between the incoming and the last image compressed values, identified with a numerical value, finding the module of the difference between correspondent sections of the images. The first data in memory will be compared to the first incoming packet, the second to the following one and so on. At each step, the algorithm sums up the modules so far obtained in an accumulator, which stores during the acquisition process the cumulative change of degree, here identifying the total sum of all the modules. Since only positive values can be computed, being the change calculated with an absolute value, there will be no aliasing due to the

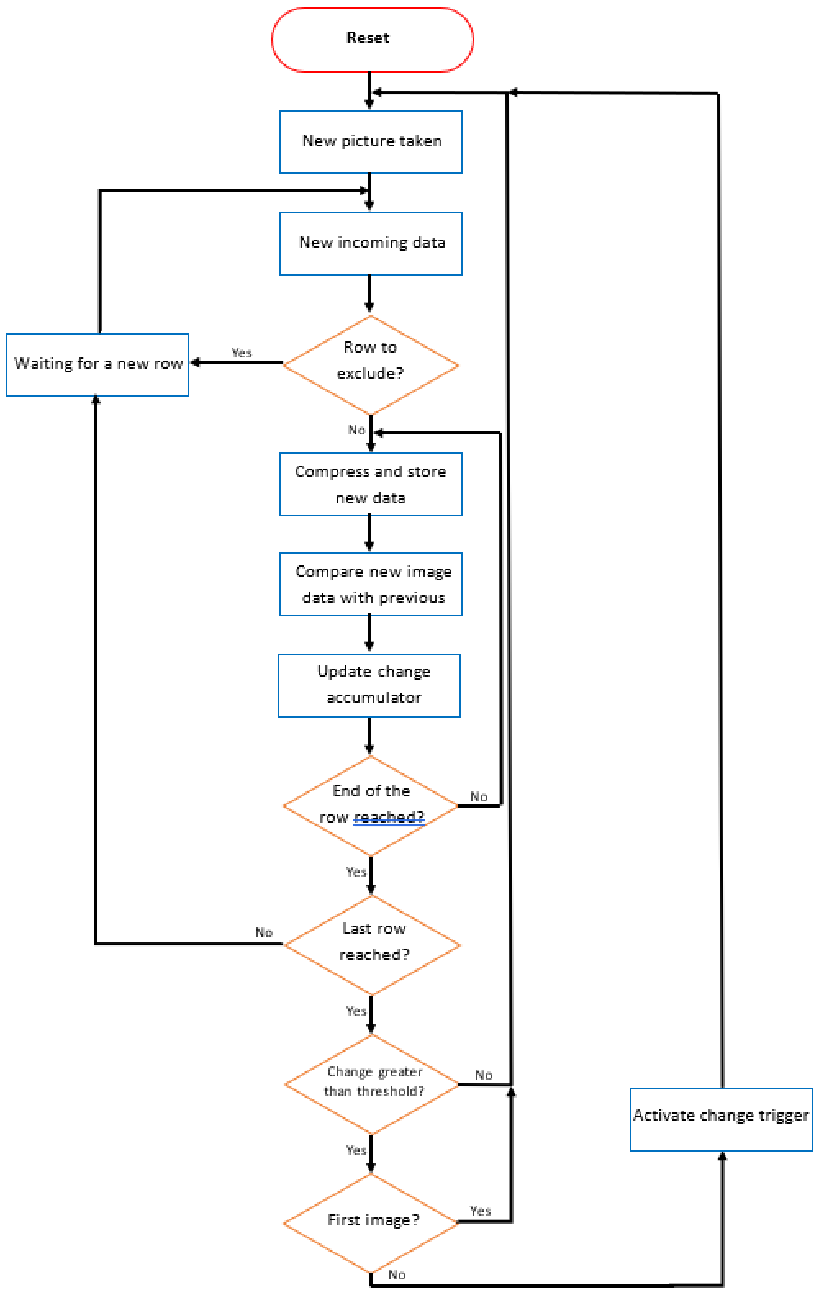


Figure 2.9. Flow chart of the algorithm to activate the change trigger.

opposite contributions of different sections (for example, an object moving inside the field of view).

Change threshold
After the difference in pixel values between images has been fully evaluated, the algorithm

checks if the value stored in the change accumulator is greater than a given threshold. The value of the threshold is controllable by setting one of the control registers present in the design, which is modifiable through MATLAB. This feature is particularly important, since it is possible to adjust this value without having to re-synthesize the whole design. In this way, the same framework can be set to work with different phenomena, faster or slower ones, and even to capture different events according to when the trigger is set. For the implementation of this feature, it was necessary to modify parts of the system related to assessing the control signal, included in the memory interface, and also in the MATLAB part.

When the first image is stored, it will be inevitably checked against the reset state of the memory whether a change has occurred or not. Since the content of the first picture taken is unpredictable, the accumulator may assume a value greater than the threshold. To avoid this event happening, a new feature was inserted. The trigger can only be activated if the related enable signal is high; during the first acquisition this is always '0', which means that is not possible to trigger the system during this time. After the picture has been stored inside the memory, the enable signal is asserted high, which activates the trigger functionality for the next image on.

Trigger mode selection
Another feature that has been added to the acceleration_unit, which is external but directly connected to the change trigger, is the possibility of switching between the trigger produced by this block and the one asserted from the NN part, which was already present in the design. Selection between these two modes is made by changing a single bit inside the control registers of the system, to which this function has been assigned. If the accelerator_unit trigger mode is activated, the system would wait for the signals from the NN or the change trigger block to start recording a stream of images at the intended time.

Further work required
This component has been integrated in the system and has been validated with a specific simulation on ModelSim, but on camera test still needs to be tried in order to assess its very functionality inside the framework. Moreover, it would be interesting to include as functionality the possibility of storing the incoming image when the trigger has been activated, in order to continuously monitor the events that may happen after the first detection.

2.3 Servo controller

In order to connect the accelerator output to a servo motor, another block, namely, a server controller, was inserted inside the design with this specific purpose. The servo motor must to be controlled through a PWM signal, which has to be generated from the accelerator result. The main idea to realize the servo controller was to use two different sub-components for this purpose with the aim to make the servo controller more flexible and able to cope with future changes in the framework. The two sub-components that have been implemented are the interface with the accelerator and a PWM generator.

Servo control interface
The servo control interface (or shortly *servo interface*) is just used to match the output of the accelerator with the size of the input required by the PWM generator. Being the output of the accelerator adjustable according to the user needs, the interface should be able to correctly compute inputs of different size.
Since the accelerator output format was not coded yet as a settable parameter, only the 1-bit output case was contemplated. However, in the future it will be possible to extend the framework in order to work with the desired parameter settings. As far as this specific implementation is concerned, the 1-bit output of the accelerator gets resized into a 1-byte output for the PWM generator. In other words, the '1' or '0' accelerator result is translated into an 8-bits sequence of ones or zeros, respectively, which correspond to 255 and 0 in decimal base (from binary unsigned representation).

PWM generator
Simply called *servo_ctrl*, this part generates a PWM signal with characteristics that match the servo specifications. The component has been designed not only to command the servo to move on a few positions, but it is also theoretically possible to move its arm to any desired angle.

Below, the servo specifications are first analyzed: the motor requires a 20 ms square signal with a duty cycle comprised between 5% and 10% (these values correspond to the limit positions of the arm, which will be called, respectively, position 1 and 2). These features have been implemented by designing a counter that increases at each clock cycle. The counter is reset after reaching a pre-computed value (called reset value), in order to create a periodic square wave. Moreover, the output signal is switched when the counter reaches a second value (called switching value), to obtain the correct duty cycles. The formula which sets the reset value is the following:

$$R = \frac{T_{SW}}{T_{CLK}}$$

where $\mathrm{T_{SW}}$ is the period of the square wave expressed in ns, and $\mathrm{T_{CLK}}$ is the clock period expressed in ns. As an example, with a 100 MHz clock (10 ns), and with a waveform period of 20 ms, R = 2 x 106, which requires at least a 21-bits counter. The formula to calculate the switching value is instead:

$$S = \frac{R}{20}\left(1 + \frac{I}{255}\right)$$

where R is the reset value, while I represents the 1-byte input value expressed in decimal base. The S value goes from R/20 when I = 0 to R/10 when I = 1, which correspond to the PWM signals that move the arm to position 1 and 2, respectively. Note that, at reset, S is always set to the minimum value, since in the case the servo should always move in a predetermined position (in the specific case, 1).

Figure 2.10 shows the schematics of the final implementation.
More information on servo characteristics and specifications are available at [15].

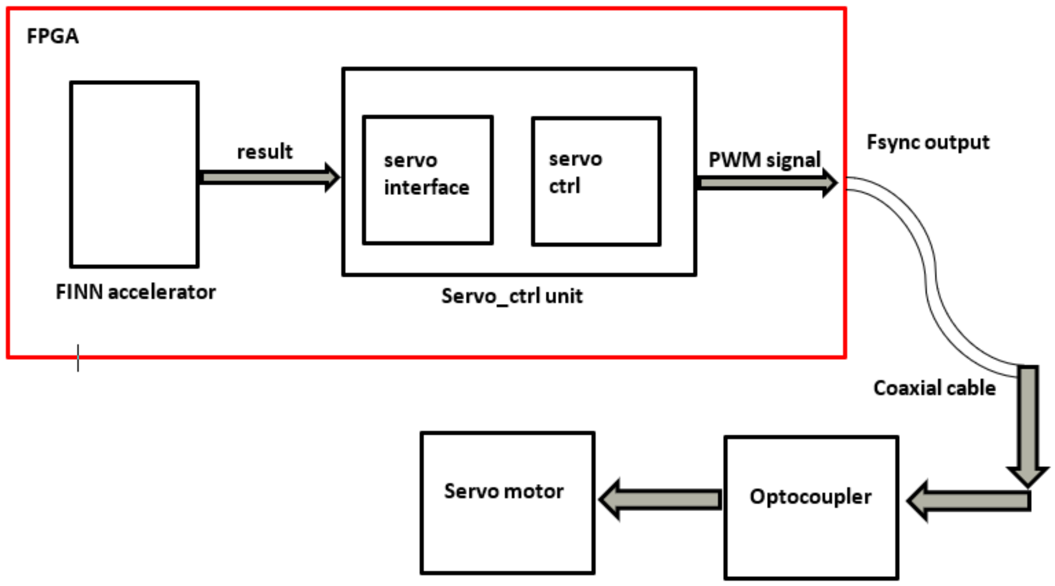


Figure 2.10. Final implementation of the servo controller.

Chapter 3

Evaluation and Testing

At the end of of the implementation phase, only the accelerator pre-processor could be effectively put inside the complete design and tested. The change trigger, even if theoretically completed and present inside the final accelerator unit version, still had to be completely evaluated. The final version of the servo controller on the other hand, even if it was not connected to the output of the accelerator, could still be tested, in order to check whether it would correctly respond to a series of synthetic inputs. For what regards the k160 adaptation obtained by changing the frame_mem block inside the design, the final testing showed that both hardware would correctly operate within the desired period of time, with all the previous functionalities preserved.

The next sections will be dedicated to explain in more detail which parts have reached the required completeness and which still needs further work to develop the expected outcomes.

3.1 Averaging feature

The block which was first added to the complete framework was the final version of the pre-processor entity, to which the averaging part was added. In order to correctly work, the MATLAB code was modified so as to read the buffer inside the accelerator, which stores the image after the reduction process from the interface. In this way, both functionalities of the block have been tested: the subsampling option and the averaging mode. To this end, the control bit that was chosen to select the type of compression was introduced inside the configuration file of the MATLAB project; in this way, it could be set it using the proper assignment function. The images taken from this analysis, which was intended to demonstrate the differences between the acquisition methods and possible advantages of both, will be shown in the next paragraphs.

Figure 3.1 shows the acquisition of a single image done from the camera system: this is the only modality in which the reduction mechanism could be tested, since in order to read out the accelerator internal buffer, single pixels get extracted, resulting in a large required time to acquire the stored picture. The main subject chosen for the photo is a painted glass, with shard-like shapes on it. This pattern will result particularly interesting, since the representation given by the two reduction modes will show a greater, much more

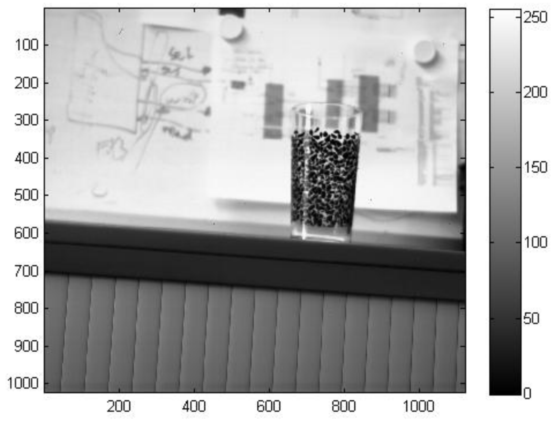


Figure 3.1. The original 1Mpx image obtained from the camera.

noticeable difference. In the same image, the piece of furniture below the glass shows some oblique stripes, which represent another feature that can be highlighted using the different acquisition methods.

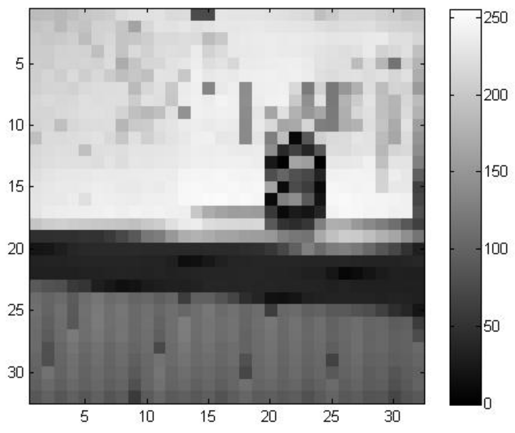


Figure 3.2. Subsampled 32x32 image.

From the results obtained in Figure 3.2, we can clearly notice that the glass has been sampled taking only some of the pixels; as a result, its appearance is characterized by a sequence of dark and bright tiles. This will help distinguish its particular pattern, which is characterized by a sequence of black shapes, but it also causes the glass shape to become blurred. In absence of a white board on the background, it would be even more difficult to distinguish where the shape ends, which could complicate the detection of specific features from the accelerator that receives those data. Furthermore, when looking at the piece of furniture, it is still possible to distinguish the stripes that characterize its surface, but their orientation is much more difficult to predict, since some shading effects present on the image modify some of the pixels that become much darker and more influent than before. A recognition system would produce a more uncertain output when shown this image, since some stripes display irregularities.

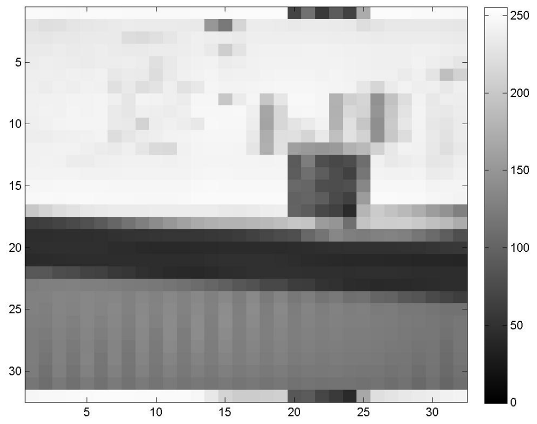


Figure 3.3. Averaged 32x32 image.

Switching the selection bit, it is now possible to analyze instead the averaging functionality, which is represented in Figure 3.3. The first thing that can be appreciated is the fact that two lines from the center of the picture have been moved to the top and bottom rows. The reason for this effect is that some mismatch between the calculation and the transfer mechanism produces the incorrect arrangement of the pixels inside the buffer. Many images have been taken from the same subject, resulting in a different disposition of the incorrect rows each time. This means that the final implementation of the feature is characterized by an incorrect timing, with aleatory of pixels's position inside the image.

Even when using the maximum permitted clock period, the images still show an incorrect pattern. Investigating the issue, it appeared that, inside the accelerator pre-processor block, during the transfer of data from two consecutive registers, the slack measured was negative and this would not let the tiles be set in the correct position during the transfer time. When this was discovered, a new approach which could exemplify the average calculation was developed, in order to speed up the process, as described in the Design and Implementation Chapter (Chapter 2). Nevertheless, this method reduced strongly the slack, making it positive, but due to the lack of time to dedicate to this part, the new modified feature remained untested, though from basic simulations it seems to work as intended. Future work would need to assess the correctness of the algorithm developed, showing on camera the equivalency of the two behaviors, with the exclusion of image defects which characterized previous work.

Nevertheless, reconstructing the image, as displaced in Figure 3.4, it is still possible to distinguish the main object (the glass), which is now characterized by a more uniform tone: the dark and bright pixels have completely disappeared, resulting in a much worse

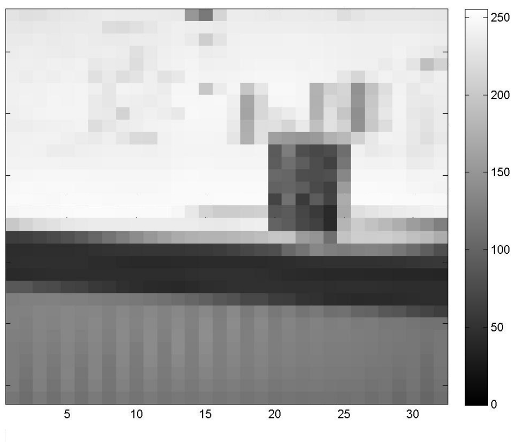


Figure 3.4. Reconstructed 32x32 averaged image.

detection of the surface alternating coloration pattern. On the other hand, now the glass shape can be better assessed, since it will appear more clearly in front of the whitish background. The accelerator would much more easily recognize the rectangular shape, with respect to the previous result. Moreover, the stripes of the piece of furniture below are now all well detectable, since their pattern is more regular, with an absence of darker shades. This effect is caused by the fact that during averaging all the pixels in each tile are counted up to extract the final value, resulting in a much more regular overall outlook. The consequence is that, especially when examined with a regular background, right shapes (as rectangles, squares, stripes, ...) will resemble much more their original aspect with the averaging mode.

The conclusions expressed so far might result useful for future exploitation of the two functions: depending on the purpose, the accelerator might find more convenient to work with one of the modes with respect to the other. For example, if the user request consists in finding out if something relevant is moving inside the field of view, they may opt for the subsampling option, since it is still more reliable when detecting large objects that are easily distinguishable from the background. Instead, if the object is small, it would be preferable to use the averaging mode, since it would continuously display a trajectory, yet a feeble one, of the object instead of a discontinued, unreliable one. The important result is that, according to its requests, the user can freely adjust the camera settings to move from a simpler to a more accurate representation of the image to feed to the accelerator, increasing the flexibility and the potentialities of the analysis.

3.2 Implementation results

At the end of the six-month internship, two different versions of the project have been produced: one with the accelerator unit inserted in the camera system and the k160 FPGA, the other with the latest version of the k325 adapted to the k160. Due to the lack of time for merging the two projects together, both of them lack of some of the features described so far, but they can both be synthesized and loaded on the camera system.

In more detail, the first version of the project, hereinafter referred to as Accelerator_Unit, comprehends the accelerator unit block inside, with all the features depicted in the previous sections of this document, but it still runs with an old 160k version, without the latest improvements coming from the 325k. It is important to mention that this version does not contain the accelerator FINN core. The other one, hereinafter referred to as v2_fasteye, was created to solve the incompatibilities between the latest version of the k325 board and the k160 board. Due to the fact that some features were developed alongside the first design and implementation steps, at a certain point it was not possible to work on one version only, so both of them have been carried on to the end.

Figure 3.5 shows the results obtained from the utilization report of the Accelerator_Unit and of the v2_fasteye project, so as to check and compare the values obtained. Essentially, the v2_fasteye project requires overall less resources to be synthesized with respect to the other one. By performing such a comparison, we can assess the effect of the modified accelerator unit on the design, estimating the resources required with respect to the project without this component. The implementation of the modified data pre-processor and the change trigger increases both the usage of LUTs (Look Up Tables) for logic computations and flipflops, especially for what regards the averaging calculations. Moreover, the BRAM (Block Random Access Memory) usage increases as well, probably due to the fact that the compressed image obtained from the change trigger is potentially stored in this part of the FPGA. Also, the number of used BUFGs (global buffers) increases, basically due to the higher complexity of the design.

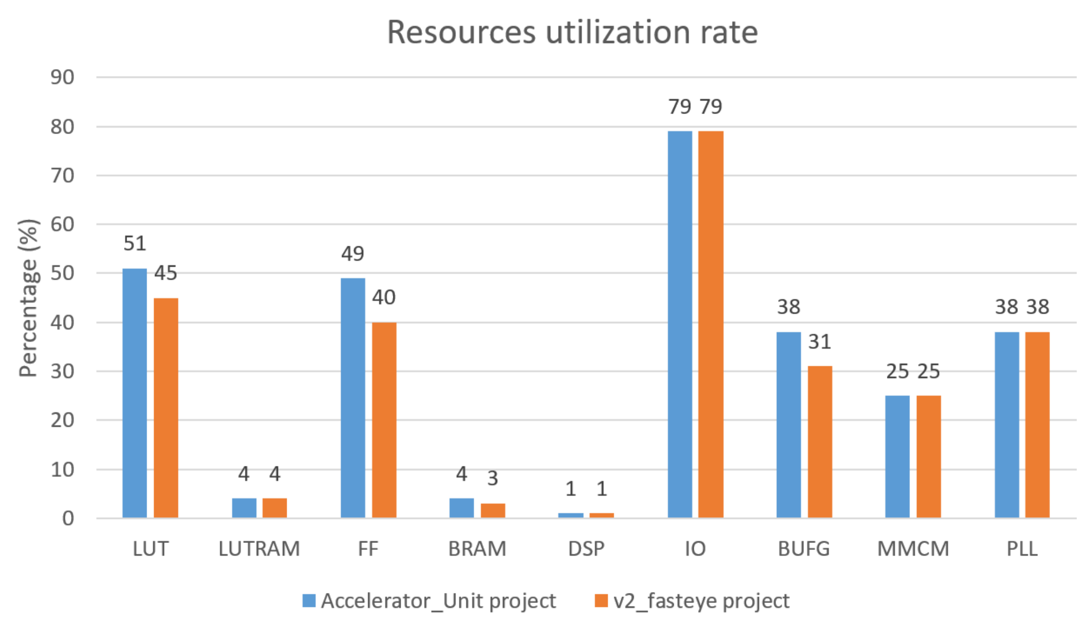


Figure 3.5. Utilization rate of the resources inside the k160 FPGA derived from the implementation of the Accelerator_Unit and v2_fasteye projects.

As mentioned before, both projects were synthesized and loaded on the FPGA, despite being incomplete in some parts (as the accelerator FINN core). Due to the large size of the existing accelerator introduced inside the design, this part was not synthesizable on the k160 model. The results from the utilization report generated by the VIVADO software showed that the total required flipflops largely exceeded the available quantity inside the chip. This is the reason why the accelerator has been substituted with a simpler embedded block, which simulated the value of the outputs of the component.

It is worthwhile to remark that, as an alternative solution, one could have substituted the pre-existent accelerator structure with a simpler one, since, thanks to the FINN characteristics [1], the framework is highly scalable and adjustable to the requests of the user and the space available. Nevertheless, since the planned timeline was not followed as supposed to and since the presence of the NN core was not mandatory, such an alternative solution was not adopted. This method of substituting the NN part with the embedded block should be fixed in future work, to properly examine the utilization rate of the camera system when a real computational structure is inserted.

Table 3.1 summarizes the final timing and power consumption characteristics of the two projects.

Table 3.1. Power consumption and worst slack of the Accellerator_Unit and v2_fasteye projects.

Project version	Power consumption (W)	Worst slack (ns)
Accelerator_unit	3.062	0.068
v2_fasteye	3.062	0.323

As mentioned before, both the projects were run using a 160 MHz clock for the sensor part, which helped to obtain a positive result for the slack. Adding the new features to the accelerator unit, the required power needed to run the application on board increases as expected. Still, this value does not represent the final power required by the system, since the accelerator part, which requires the larger computational power, is missing from this evaluation.

After some optimization process, a positive value for the worst slack statistics has been reached for both projects. Table 3.1 shows how the substitution of the frame_mem block inside the v2_fasteye project strongly improves this parameter, suggesting that even higher frequencies could be exploited in order to work with a higher frame rate. Nevertheless, for what regards the accelerator unit, the time required for the new functionalities is very small (barely below the maximum allowed time for the signal to be correctly sampled), which means that the system should still be further optimized in order to reach even better performances in the future.

3.3 Servo controller functionalities

As mentioned in Chapter 2, the servo controller was tested to verify its actual behavior when attached to the servo motor, since the component was specifically designed to match the peripheral device specifications. A first validation step consisted in verifying the actual output of FPGA fsync pin, which should have output the correct 3.3 V high square wave. In order to check this, the output of the servo controller was assigned to pin P26 on the FPGA, which was then connected to the desired output. This procedure required to analyze the datasheets of the system schematics, in order to properly connect the FPGA and the board output pin, resulting in an easy soldering of part of the board. P26 pin was connected to the output of the servo controller by modifying the pin assignment constraint file inside the project. *fsync* board output was mainly chosen for two reasons:

- It was easily accessible from outside the camera through a coaxial cable connection, which was used to connect the output to the oscilloscope and the evaluation board;
- Pin P26 was inside a 1.3 V-bank within the FPGA, so its voltage level had to be raised. In the electronic design, before *fsync* a voltage level shifter buffer was present, which would have increased the output value to the required value of 3.3 V.

After this preliminary phase, the PWM signal generated by the component could be finally checked, looking in particular at whether the desired frequency (50 Hz) was obtained and the duty cycle was correctly assessed. Since the accelerator result was not available at the time, having the servo controller been developed during a preliminary phase, simulated inputs have been provided to the component in order to verify its behavior. An embedded block was inserted for this reason in the design, whose output could be easily modified changing a single line of code. Connecting the block to the servo controller and the fsync output to the oscilloscope, it was indeed found that, when asserting '0' and '1' inputs, the square wave showed the intended period; moreover, the duty cycles were 5% and 10% as intended. This first validation process was useful to verify the correctness of the algorithm described through HDL, and that the correct pin assignment had been made between the board and the FPGA.

After this first step, the actual evaluation board, comprised of the servo motor, was developed, as shown in Figure 3.6. The final setup was defined, accounting also for the fact that the actual voltage level of the PWM signal had to be raised again according to servo specifications (between 4.8 V and 6 V). The PWM signal generated from the camera does not directly control the servo motor, but the fsynch pin is connected to an optocoupler, which would then work as an intermediate inverter stage. The optocoupler also isolates electrically the board and the servo, in order to protect the former from possible tension fluctuation. After connecting two resistances to the optocoupler ports and a voltage supply (set to 6V), the setup was completed. The resistance values (see Figure 3.6) have been chosen in order to limit the value of the current flowing inside the light emitting device (about 5 mA) and the transistor. At last, it is worth mentioning that, since the new component would negate the PWM signal, the obtained duty cycle was not 5-10% anymore, but 90-95%. To fix this, the servo controller code was simply adjusted to produce a negated square wave with respect to the original one, which would have responded correctly to the new specifications of the setup.

The behavior of the servo motor was as expected: when a '0' result from accelerator was simulated, the servo arm moved correctly to position 1, while the '1' result would let the arm move to position 2.

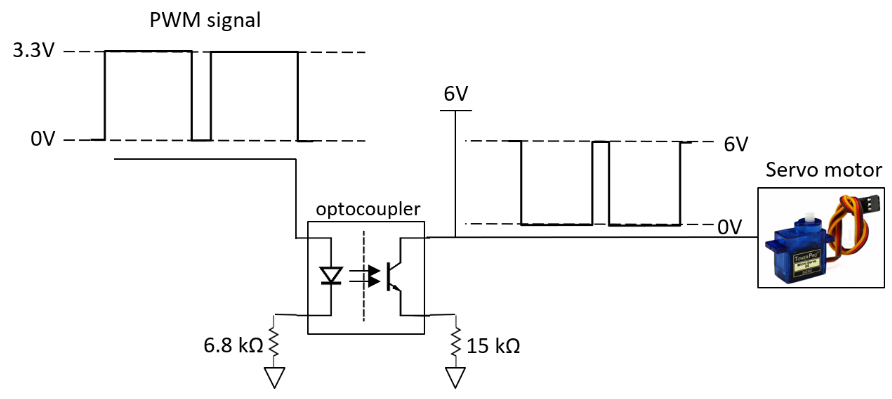


Figure 3.6. Final implementation of the setup that controls the servo motor (picture of the servo motor taken from [15]).

Chapter 4

Conclusions and Future Work

The main aim of the internship was to create a framework that could analyze the images obtained with the camera through the implemented neural network. The image would have been processed to be fed to the accelerator using one of the two modes, subsampling or averaging, the latter implemented during the internship. Also, a new trigger mode was meant to be inserted, which would start the acquisition of images as soon as a significant change in the field of view occurred. These functionalities were intended to be included in a single block, a reliable generic interface between the application part of the camera and the accelerator on FPGA. Additionally, the work aimed at identifying the issue that caused the mismatch between the k160 and k325 projects, in order to solve them.

The first task was divided in two subtasks: the creation of a change trigger functionality inside the design and the addition of an averaging feature to the data preprocessor block for the accelerator. For both subtasks, an initial design was performed in order to define appropriate algorithms for their realization. After validating the behavior of the HDL codes implementing the algorithms through a specific testbench, the new version of the data preprocessor was inserted within the camera system to check its functionalities. Also, the change trigger component was inserted along, creating a unified accelerator interface component.

A servo controller embedded block has been designed for a possible future demonstration. The accelerator output was simulated during testing since this was not included at the time. It was designed a simple setup to connect the camera system to the servo motor with the aim to fulfill the motor specification. The interaction between the camera and the motor was tested, synthesizing the framework for right and left positions. The correct behavior of the developed functionality was assessed.

The second task related to the k160 and k325 projects was solved by changing the frame_mem block inside the camera system by merging the coding from the k160 and k325 projects. Before finding the cause of the mismatch, other options have been investigated regarding the memory interface part and other components of the system. In order to assess where the system was actually failing, the VIVADO reports have shown to be particularly useful. Such reports pointed out that the timing of certain portions of the design were not compatible with the k160 FPGA characteristics.

The new system can now sample images for the accelerator NN with a new method, more complex if compared with the previous one, which can provide more useful information for data elaboration. The mismatching between the two versions has been solved, since now it is possible to run the project framework on both available boards, even if the characteristics and performances of the two are different (see paragraph 2.1.1), also at some extent the new functionalities are compatible. The change trigger block, designed but not yet inserted in the system, works as supposed to and with the flexibility required from the specifications. Still, it has not been completely tested, so the achievement related to this part has been fulfilled only partially. Once those final modifications will be provided, the camera is already configured to be used in a real environment through the output functionality implemented by means of the servo controller block, which is now part of the framework. With respect to the previous project status, the camera can now control an external device for practical testing in the real world.

In order to complete the work, some parts still need to be properly tested and/or inserted within the framework. In particular,

- the averaging system must be perfected to eliminate row glitches, which are still present and are mainly due to the aforementioned timing issue. A possible solution could be to use the k325 FPGA, but fixing the timing issues is still important to adapt the averaging process to the limited k160 model. The optimization of the averaging algorithm could meet the required performance, but it still needs to be tested and, possibly, further refined;
- according to pre-testing (verification through simple test vectors, not adapted to the real case of the camera), the change trigger block behaves as intended, but it still needs to be integrated inside the latest version of the framework. Also, further testing on the camera system requires to be performed, after properly routing the change trigger output toward the trigger control block to verify its correct assessment. Furthermore, testing the final framework with different thresholds might provide more information on the appropriate value to which the threshold parameter should be set under different conditions;
- the servo controller has been tested with the use of a small breadboard, and it appears to efficiently control the servo motor using the PWM signal. Future work could focus on extending the capabilities of the servo interface component in order to make it able to process accelerator outputs of different length.

These are the main tasks which were required to be performed during the internship that are still missing.

In terms of future directions of the work, which could be in general pursued, several possible evolutions of the activity could be investigated and performed:

- Further work on the change trigger might be done for optimizing the algorithm, in order to find better coding to match the capabilities of the framework. Especially the compression mechanism could be improved, so that it takes less space and becomes faster.
- An accelerator, which can be synthesized on k160 FPGA model, can be inserted inside the design, in order to properly test the final framework completed as intended.

- With the accelerator output available and a functional set of biases and weights appropriate for a specific purpose, the camera can be fully tested. The servo controller can be connected to the accelerator output, in order to check the actual final implementation.
- A final demonstration can be performed: using the breadboard designed and the camera system, it will be possible to show the actual operation of the new features on a real case. In particular, it would be interesting to show the performance of (i) the switching between subsampling and averaging modes for the accelerator input data, and (ii) the capability of the change trigger to correctly start the acquisition of a predetermined stream of images inside the memory.

The final framework could be useful for many, appealing applications, such as defect checking, scientific experiments recording, or fast face detection.

Bibliography

[1] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers, “FINN: A Framework for Fast, Scalable Binarized Neural Network Inference”, *2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA ’17)*, New York, NY, USA, pp. 65-74, 2017.

[2] Fremont Weekly Magazine, "Global High Speed Camera Market 2019 to 2025 With Top Countries Data : Market Growth, Share, Size, Consumption and Growth Rate by Application, Types, Key Players and Competitive Regions", fremontweekly.com

[3] Saleh Ali K. Al-Omari, Putra Sumari, Sadik A. Al-Taweel and Anas J.A. Husain, “Digital Recognition using Neural Network”, *Journal of Computer Science*, vol. 5, no. 6, pp. 427–434, 2009.

[4] O. M. Parkhi, A. Vedaldi, A. Zisserman, “Deep face recognition”, in *British Machine Vidion Conference (BMVC)*, vol. 1, p. 6, 2015.

[5] L. Nwosu, H. Wang, J. Lu, I. Unwala, X. Yang, and T. Zhang, “Deep Convolutional Neural Network for Facial Expression Recognition Using Facial Parts”, in *2017 IEEE 15th International Conference on Dependable, Autonomic and Secure Computing,* Orlando, FL, USA, pp. 1318-–1321, 2017.

[6] J. Heikkonen, J. Lampinen, “Building industrial applications with neural networks”, *European Symposium on Intelligent Techniques*, Orthodox Academy of Crete, Chania, Greece, June 3-–4, 1999.

[7] X. Gibert, V. M. Patel, R. Chellappa, “Deep Multitask Learning for Railway Rrack Inspection”, *IEEE Transactions on Intelligent Transport Systestems*, vol. 18, pp. 153–164, 2017.

[8] R. Yamashita, M. Nishio, R.K.G. Do, K. Togashi, “Convolutional Neural Networks: An Overview and Application in Radiology”, *Insights Imaging*, 2018.

[9] P. Jokic, S. Emery, L. Benini, “BinaryEye: A 20 kfps Streaming Camera System on FPGA with Real-Time On-Device Image Recognition Using Binary Neural Networks”, *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, Graz, Austria, 6–8 June 2018, pp. 1—7.

[10] D. Lilwiller, "CCD vs. CMOS: Facts and Fiction," DALSA Technology with Vision, Accessed in October 2019.

[11] M. Versluis, “High-speed imaging in fluids", *Experiments in Fluids*, 2013, 54. 10.1007/s00348-013-1458-x.

[12] B. Fu, M. C. Pitter, N. A. Russell, “A Reconfigurable Real-Time Compressive-Sampling Camera for Biological Applications", *PLoS One.* 2011; 6(10)

[13] "High-speed cameras enhance automation and measurement applications," https://www.vision-systems.com/factory/article/16736123/highspeed-cameras-enhance-automation-and-measurement-applications, Accessed in October 2019.

[14] https://www.xilinx.com/support/documentation/data_sheets/ds182_Kintex_7_Data_Sheet.pdf, Accessed in October 2019.

[15] http://www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf, Accessed in October 2019.