Aproximación a la Implementación de Aplicaciones Aero-espaciales en Hardware

Víctor Diges Teijeira

GRADO EN INGENIERÍA DE COMPUTADORES. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

CURSO 2014-2015

Trabajo Fin de Grado en Ingeniería de Computadores

29/06/2015

Directores:

Marcos Sánchez-Elez Martin

Inmaculada Pardines Lence

Autorización de difusión

Víctor Diges Teijeira

29/06/2015

El/la abajo firmante, matriculado/a en el Grado en Ingenría de Computadores de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado: Aproximación a la Implementación de Aplicaciones Aero-espaciales en Hardware, realizado durante el curso académico 2014-2015 bajo la dirección de Marcos Sánchez-Elez Martín e Inmaculada Pardines Lence en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen

En la actualidad, los sistemas de hardware reconfigurable están muy extendidos en el entorno aeroespacial. Esto es debido a su gran potencia computacional comparada con la de soluciones software. En este proyecto se analizan las múltiples aplicaciones de las FPGAs en esta industria, y cómo la investigación espacial y de defensa han liderado el desarrollo de sistemas aeronáuticos electrónicos. Estos están presentes hoy en día en la mayoría de aeronaves comerciales. Además, se introducirán brevemente los efectos de la radiación del espacio en los elementos electrónicos, explicando cómo afectan a las memorias y los circuitos y cómo pueden ser detectados y corregidos por la FPGA.

Se ha escogido el procesado de video en tiempo real como aplicación objetivo de este trabajo, mostrando las características de las FPGAs que pueden ser explotadas para obtener resultados con muy poca latencia. Se ha utilizado la herramienta Matlab para simular los algoritmos, indicando al mismo tiempo cómo podrían ser segmentados en una FPGA. Para mejorar los resultados se ha desarrollado un procedimiento para la detección de objetos para una vista de cámara móvil, basado en la eliminación de ruido de la imagen y en humbralado.

Palabras clave

FPGA, Aeroespacial, Visión por Computadora, Procesamiento de Video, Reconocimiento de Objetos.

Abstract

Nowadays, reconfigurable hardware systems are widely used in the airspace environment. This is thanks to their great computing power in comparison to software solutions. In this project we analyze the multiple applications of FPGAs in this industry, and also how the space and defense research have lead the development of electronic aeronautic systems. They are now present in most of the commercial aircrafts. In addition, we briefly introduce the effects of space radiation in electronic, explaining how they affect memories and circuits and how they can be detected and corrected by an FPGA.

Real time video processing was chosen as a target application of this work, showing how FPGA's characteristics can be exploited to obtain results with very low latency. Matlab tool was used to simulate the algorithms, providing insight on how they could be pipelined in an FPGA. In order to improve the results an object detection procedure for a moving camera view was developed, based on noise removal and thresholding.

Keywords

FPGA, Aerospace, Computer Vision, Video Processing, Object Recognition.

Contents

Index i

1 FPGA Applications in avionics 1

1.1 Flight Control . . . 2
1.2 Navigation and Monitoring . . . 3
1.3 Augmented Vision . . . 4
1.3.1 Real Time analysis . . . 4
1.3.2 Intelligent Transport . . . 5
1.3.3 Immersive Displays . . . 5
1.3.4 Image Stabilization . . . 6
1.4 Multispectral Image Processing . . . 6

2 FPGA Applications in space 9

2.1 Single Event Effects . . . 9
2.2 SEU detection and mitigation . . . 10
2.3 SEU Mitigation Approaches . . . 11
2.3.1 On-chip detection . . . 11
2.3.2 External detection . . . 11
2.3.3 Watchdogs . . . 11
2.3.4 Dual and Triple Module Redundancy . . . 12
2.3.5 Lockstep Architecture . . . 12

3 Video Analysis: Tracking objects 13

3.1 Image Digitization . . . 14
3.2 Pre-processing . . . 16

3.2.1 Kinds of operations 16
Punctual 16
Local 16
Global 18
3.2.2 Grayscale Conversion 18
3.2.3 Histogram Equalization 20
3.2.4 Noise reduction 22
Median filter 22
Mean filter 23
Gaussian blur 23
3.3 Segmentation 23
3.3.1 Border Detection 24
Sobel Operator 24
Laplacian operator 25
3.3.2 Object Recognition 26
3.4 Feature Extraction 27
3.4.1 Connected-component labeling 27
3.5 Representation 29
3.6 Final result 30

Conclussions **33**

Conclusiones **35**

Bibliography **37**

Appendix: Source Code **39**

Chapter 1

FPGA Applications in avionics

Computing requirements in aircraft integrated systems are specially high. For example, a militar aeroplane can require more than 10^{10} operations per second. Furthermore, most of these systems are critical to the safety and control of the aircraft which makes its reliability of particular relevance.
Field Programmable Gate Arrays (FPGAs) are able to provide this processing power and they benefit from other features such as reconfigurability, lower cost design, lower power consumption and smaller form factor compared to other computing solutions.

Manufacturers current tendency is to integrate a hard ARM core in their FPGA chips. This allows to combine software based control and analysis with hardware based real time processing, reducing costs and design complexity. Top products in this category are Xilinx's Zynq-7000 and Altera's Stratix 10 SoCs[1], that combine a dual-core ARM Cortex with a rich peripheral set which is complimented with memory mapped programmable logic. This allows to design new peripherals and hardware accelerators.
In order to help developers, FPGA manufacturers usually provide a full range of pre-designed modules referred as IP cores. These modules use to be highly configurable, however their source code is not usually disclosed.

[1] System on a Chip.

FPGAs are used in many different areas for Aerospace and Defense environments. The main processing unit in civil aircrafts is the avionic system[2]. This system includes navigation, communication, monitoring and control, along with safety systems like the collision avoidance system (CAS).

1.1 Flight Control

Most modern aircrafts include a fly-by-wire control system[1], which replaces the conventional manual flight controls with an electronic interface. Pilot's input is converted to electronic signals transmitted to the aircraft computers which determine how to move the control surfaces. By giving a direct input through electrical signals, commands are much more precise. Furthermore, the control system monitors pilot commands to ensure the aircraft is kept within the flight protection envelope. This improves flight safety and reduces crew's workload during critical phases of the flight such as takeoff and landing. The use of electrical wires in stead of mechanical cables and pulleys also reduce drastically the aircraft weight and makes maintenance easier.

The Flight Management Computer (FMC) also provides autopilot features. Autopilots do not replace human pilots, but assist them in controlling the aircraft. Early autopilots were only capable of attitude holding but they have evolved to perform automated landings under the supervision of the pilot. Modern autopilot also provides great route-following precision in current tight airspaces, being able to follow a pre-stablished path of waypoints in stead of early point-to-point navigation and compensating path deviations due to wind.

[2]Electronic systems used on aircraft, artificial satellites, and spacecraft.

1.2 Navigation and Monitoring

Modern digital electronics have also been very important in flight instruments and navigation. Around 1970 the increasing complexity of aircraft systems needed a great amount of gauges in the cockpit. The average cockpit in a mid-1970s transport aircraft had more than one hundred cockpit instruments and controls. The growing number of cockpit elements were competing for cockpit space and pilot attention. This raised awareness of the need of displays that could process the raw aircraft system and flight data into an integrated, easily understood picture of the aircraft situation, position and progress.

In response, NASA led a research with key industry partners to develop and test electronic flight displays concepts. The result was the development of a glass cockpit[3] system [2]with an autopilot that increased safety by reducing pilot workload at peak times. The success of the NASA-led glass cockpit work is reflected in the total acceptance of electronic flight displays in commercial and military aircraft. This technology was later ported to spaceships, beginning with the Atlantis Shuttle which first mission with its renewed cockpit took place in May 2000.

This new displays, along with the FMC[4], provide a great advance in navigation information. Current navigation systems show the aircraft's position on moving map displays. In-flight computers can also store high resolution maps, flight charts and other important information previously carried on paper. Early glass cockpits behave more like enhanced flight instruments but current systems are proper computers, with windows and data that can be manipulated by point-and-click devices. Other information that can be shown in this displays is weather information, terrain and a realistic 3D depiction of the outside world (like in a flight simulator).

[3]Aircraft cockpit that features electronic (digital) flight instrument displays, typically large LCD screens, rather than the traditional style of analog, electro-mechanical dials and gauges

[4]Flight Management Computer

1.3 Augmented Vision

One of the main advantages that FPGAs have in high performance environments is their ability to process massive amounts of data in real time. This is useful in signal processing and specifically for video processing and analysis. The combination of multiple video inputs along with sensor data provides pilots with easy access to useful information of the surroundings, usually referred as Situational Awareness. There are three main areas in which FPGA's are used:

1.3.1 Real Time analysis

Artificial vision has always been a major subject in computer science. One of the most useful features is object recognition and identification.
The first step in this process is image enhancement, which goal is to preprocess the image to improve the interpretability or perception of information about the objects. Software approaches are slow and/or require very high computing power, making them inefficient for such environments. Hardware solutions take great advantage from parallelization and data processing pipelining, specially in pixel-wise operations, making them the preferred solution. Next steps in the analysis can be executed by common processors with OpenCV libraries, making hybrid CPU/FPGA solutions a very interesting approach.

While animals can easily identify objects even if they are partially obstructed from view, this is not an easy task for computers. Therefore, Object tracking and identification is a very useful feature specially in Unmanned Air Vehicles (UAVs) and Space applications. It helps in predicting object trajectories and avoiding collisions. It is specially important in target locking systems for military aircraft. However, object recognition is not only useful for moving items: it can be used to recognize obstacles or ground equipment, for example a landing strip in low visibility operations, and show it in the Heads Up Display (HUD) to the pilot.

1.3.2 Intelligent Transport

High-quality video represent a huge amount of data. In order to save and transmit this data, raw video needs to be encoded and compressed. FPGAs allow for this compression to be executed with low latency and high noise immunity, delivering broadcast quality video over IP networks. Apart from compression, encryption is also available in case the application requires it, for example a live video feed from military aircrafts or drones.

1.3.3 Immersive Displays

With the development of glass cockpits, more and more information was being shown in the dashboard. This information needs to be displayed in an accessible, graphic manner and so GPUs and FPGAs began to be used for this purpose.

In the beginning only analog instruments were rendered to graphics, but FMCs soon started to show navigation information plotted on a virtual map as well as route prediction. Modern glass cockpits might include Synthetic Vision Systems, developed by NASA and the U.S. Air Force in the late 1970s, which provides situational awareness to the pilots, displaying a realistic 3D depiction of the outside world based on a database of terrain and geophysical features, in conjunction with the attitude and position information gathered form the aircraft sensors and navigational systems.

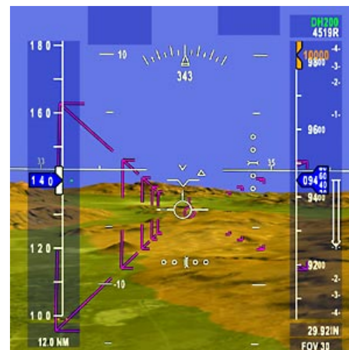

Figure 1.1: NASA Synthetic Vision Display

Enhanced Vision Systems incorporate information from the aircraft sensors, such as infra-red cameras or radar, to provide vision in limited visibility environments[3]. Night vision systems have been available to military pilots for

many years, however enhanced vision systems have been certified for civil aviation less than 15 years ago.

1.3.4 Image Stabilization

Cameras mounted in moving objects or vehicles receive part of this movement mainly as vibration. Even though physical solutions such as shock absorbers provide a good noise reduction, some video processing is needed to obtain a stable image. This is usually performed by FPGA and ASIC solutions. The processing unit needs to find the movement vector from the frames of the video stream and adjust each frame consequently[4][5].

Aircrafts can move freely on the 3 dimensional space, unlike land and sea vehicles. This adds another not desired movement in the output stream: the roll rotation. Gimbals are often used to provide aircraft cameras freedom of movement but usually only offer pan and tilt axis of freedom.

Video stabilization can be obtained in two manners: obtaining the rotational vector from the frames or using the aircraft inertial navigation system (INS) to obtain the current rotation and adjust the image to it.

1.4 Multispectral Image Processing

A multispectral image is one that captures image data at specific frequencies across the electromagnetic spectrum. It provides information not available by only exploiting the visible region of the electromagnetic spectrum[6]. Used mainly for topographic study, it's become very important in defense and scientific observation.

Non-visible spectrum imaging was first used during the Vietnam War, when color infrared imagery was used to distinguish artificial features, like camouflage, from vegetation. This technology led to the development of sensors to exploit a wider range in the electromagnetic spectrum which were later

equipped in the Landsat satellites. Landsat program became the longest running enterprise for acquisition of satellite imagery of Earth and has launched eight satellites so far (Landsat 6 failed to reach orbit) and nowadays Landsat 7 and 8 are still active.

Multispectral sensors produce a large set of images which have to be transmitted. Images need to be encoded and compressed in order to achieve good transfer rates, and for this purpose FPGAs prove to be useful thanks to their Single Instruction Multiple Data set (SIMD) computation capabilities[7].

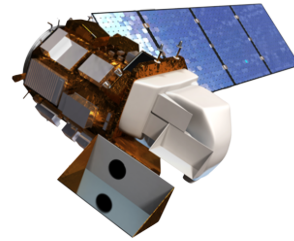

Figure 1.2: Landsat8

Chapter 2

FPGA Applications in space

Apart from high computational performance and other advantages already discussed, FPGA reconfigurability permits upgrades after launch that makes them highly suitable for space applications. Furthermore, this flexibility helps solving mishaps during runtime which can be more difficult to solve using ASICs or CPUs.

2.1 Single Event Effects

Electronics main problem in space applications is the radiation. When a high-energy particle passes through the silicon substrate of a device, charged particles are created as the result of sub-atomic particle collisions. These particles are generated by an ionization trail along the path of the incoming particle and can interfere with the normal behavior of an electronic circuit[8]. The disruptions provoked by radiation are known as Single Event Effects. Depending of the severity, two kind of errors can be identified: Hard and Soft Errors.

Soft errors are recoverable, but the system may need to be power-cycled or initiate a recovery procedure. Three types of soft errors can be identified[9]:

Single-event transients (SETs) Occurs when a high-energy particle impacts a combinatorial path of a device and can induce a voltage/current

spike. It can propagate to the rest of the circuit.

Single-event upsets (SEUs) These are the result of high-energy particles causing a change in the state of a memory element (SRAM, flip-flop, or latch). These are the most common SEE in avionics applications.

Single-event function interrupts (SEFIs) They are disruptions to normal device operation. These types of effects alter the functionality of the circuit and typically require reconfiguration/reset or power cycling for recovery.

SEE can also cause very rare hard errors, which can be permanent. FPGAs are mainly susceptible to Single Event Burnout where the voltage/current spike produced by a high energy particle can provoke physical damage to the components of the circuit.

2.2 SEU detection and mitigation

FPGAs use memory both in user logic (bulk memory and registers) and in Configuration Random Access Memory (CRAM) and these memories, along with flip-flops, can be upset by radiation. CRAM configures all logic and routing in the FPGA; if an SEU strikes a CRAM bit, the effect can be severe if it affects critical logic internal signal routing (such as a lookup table bit)[10].

Block RAMs can be protected with error-correcting code (ECC) and parity schemes. FPGA configuration memory, however, cannot be directly protected in the same manner as block memory via ECC or parity checks. SEU detection techniques that monitor device configuration memories are recommended. Some space-grade FPGAs include built-in configuration memory error detection, using ECC, but more recently, the use of flash memories has been proposed as an alternative, since they need a much higher charge to discharge the floating gate and switch state.

2.3 SEU Mitigation Approaches

Manufacturers develop radiation hardened space-grade FPGAs to counter measure SEUs adding physical protection. In addition, other design measures can be taken:

2.3.1 On-chip detection

FPGA manufacturers offer IP cores to mitigate SEUs, monitoring and repairing FPGAs configuration memory. On-chip processing is autonomous: the FPGA device determines whether it is affected by an SEU, without needing external logic. These cores have the ability to detect the sensitivity of the failure and assert an error signal so the system can provide an appropriate response according to its severity.

2.3.2 External detection

SEU processing IP cores can be configured to use an external processor. An external CPU receives an interrupt request signal when the FPGA detects a SEU. The CPU then reads the Error Message Register and performs the sensitivity lookup. This system frees up FPGA memory space and configurable logic.

2.3.3 Watchdogs

Watchdogs are special-purpose hardware modules attached to the processor. They have limited impact on the performance of the system but they may require special development efforts at the software level in its management. These timers monitor the control-flow execution, the data accesses patterns and perform consistency checks, while letting the software running on the processor mostly untouched.

2.3.4 Dual and Triple Module Redundancy

These solutions are very useful when system downtime is critical. In Triple Module Redundancy (TMR) three identical instances of the hardware are run and their outputs are processed by a voting system to produce a single output. If any of the three instances fails due to an SEU, the other two can correct and mask the error.
This approach implies a penalty in both area and power consumption, but it is by far the best available alternative to protect the programable logic against SEU.

Since in Dual Module Redundancy a discrepancy is not easily solved by voting, these systems usually work in a master-slave configuration. The slave works as a hot-standby to the master. In case the master fails, the slave is ready to continue execution as a backup.

2.3.5 Lockstep Architecture

In a Lockstep Architecture, two systems are synchronized to start from the same state and both receive the same inputs. Therefore the state of both systems should be equal at every clock cycle unless an abnormal condition occurs[11]. Consistency checks are performed periodically, either by time or when the execution reaches a milestone.

When the states differ due to an error, the execution must be halted and restarted. Since a restart from the beginning is highly expensive, checkpoints are used to keep a copy of an error-free state in safe storage. Whenever a consistency check shows that the response of the two systems are the same, their context[1] is saved as a checkpoint in error-protected memory. If the consistency check fails, a rollback operation must be performed, restoring the previous context of a checkpoint and the execution is resumed.

[1]All information needed to restore the system to a valid state

Chapter 3

Video Analysis: Tracking objects

Among all the FPGA applications for modern civil and military aviation, as well as for space equipment, artificial vision was chosen to expand on in this project. It benefits enormously from FPGA features like parallelism and high computing power to provide real time video analysis.

In this project, we will focus on an application to identify objects in a video taken from a moving camera. The video had to be downscaled and trimmed due to computer performance restrains.

Modern serial processors are able to perform some of the processing required for this application with low latency, but they are not very efficient and have trouble keeping an affordable latency when introducing high quality video like current standard 1080p[1] at 60 frames per second. With current tendency to increase vertical resolution up to 4K pixels, serial processors are not a good solution for video processing and analysis. FPGAs on the other hand can be programmed to perform parallel, pixel-wise operations over a frame buffer which latency is of barely a few clock cycles, allowing also pipelining.

Artificial vision is composed of many stages:

- Image Digitalization

[1] 1920 pixels in width and 1080 pixels in height frames, with progressive scan

- Pre-processing
- Segmentation
- Representation
- Feature extraction
- Classification and identification

In this project, the pre-processing stage will be explained in depth, along with some algorithms used in it, since it is the phase that most benefit from FPGAs capabilities. Other stages will be analyzed as well in a more theoretical approach.

3.1 Image Digitization

An image is an optical representation of one or many objects illuminated by one or more sources of radiation (visible light, x-ray, ultrasonic, infra-red, etc.). Part of this radiation is absorbed by the object and the rest is reflected, depending of the physical-chemical characteristics of the object.

There is no image capturing system as precise as the human (or animal) visual system. From an image in the real world, the first step in digitizing is an optical system, usually formed by a set of lenses, which acts as a low-pass filter removing high frequencies (details).

Next step is capturing the image itself using a sensor. Common sensors nowadays use CMOS (Complementary Metal-Oxide-Semiconductor) and CCD (Charge-Coupled Devices) technology. They are composed by an array of photo-sensitive elements which transform light intensity to an electric signal. The quantity of these elements per surface unit defines the resolution of the images.

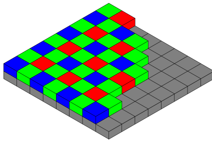

Figure 3.1: Bayer Pattern

The photo-sensitive elements can only perceive the intensity of the light, which would generate a gray-scale picture. In order to separate colors, a filter must be applied before the sensor so that it only reads the intensity of that color in particular. The most used approach nowadays is a Bayer filter mosaic. It is a color filter array for arranging red, green and blue color filters on a square grid of photosensors.

A last layer of hardware is needed to convert these analog signals to a digital value. This analog to digital converter performs quantization of the input, hence introducing some error in the conversion. The resolution of the converter indicates the number of discrete values it can produce over the range of analog values which in this case represents the intensity of the color.

As a result of this conversion, an $N \times M$ matrix of integer or real numbers is obtained. Each element of this matrix is called a pixel. If the image is color separated it becomes a 3-dimensional array, with one layer for each color (usually three).
It's easy to calculate the required space to store an image from here. The general formula is:

$$Size_{bytes} = Width \times Height \times ColorDeepness \times NColors/8$$

For a raw 1080p image with 10 bit color deepness the size is $1290 \times 1080 \times 10 \times 3/8 = 7776000$ bytes or 7.77 Mb. At a rate of 60 frames per second, each second 466.56 Mb of data are generated. Huge bandwidth is needed in order to transport such amount of data, hence compression and encoding are usually performed. Software solutions are widespread to general public but encoding performance in a serial processor is rather inefficient. On the other hand, commercial IP cores for FPGAs and ASICs[2] have a very good performance, with

[2] Application-Specific Integrated Circuit

almost real-time conversion (around 2ms of latency) but they usually are very expensive.

For video real-time analysis, however, a grayscale depiction of a frame will be used unless color recognition is important for the application.

3.2 Pre-processing

In this step, the video is enhanced to make it suitable for future processing and analysis. This includes, amongst others:

- Contrast improvement
- Borders improvement
- Noise attenuation or removal

3.2.1 Kinds of operations

Three kinds of operations can be performed on an image:

Punctual

The result of the operation depends only on the value of the input pixel for a (x, y) position. That is $g(x, y) = T[f(x, y)]$ being $f(x, y)$ the input pixel with coordinates (x, y), $g(x, y)$ the output pixel with coordinates (x,y) and T a given transformation. If the input data is a stream of pixels, this operation is performed in a FIFO order, as every pixel is treated as it enters the system.

Local

The result of the operation depends on the value of the input pixel as well as the values of the pixels in its vicinity. That is $g(x_c, y_c) = T\{f(x_c, y_c)\}$ with i = 1..c..n and j = 1..c..m where n and m is the size of the vicinity around the

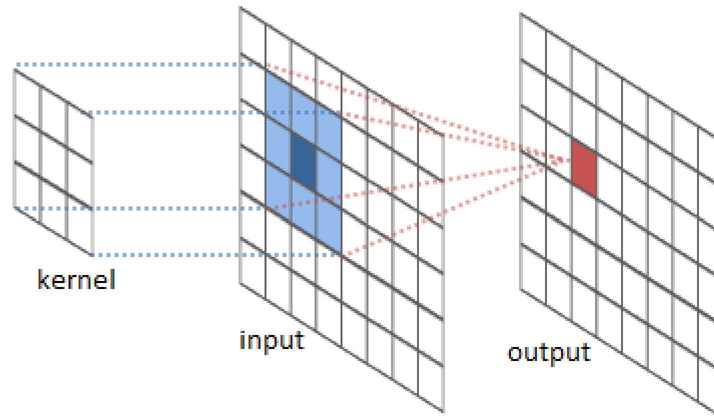

Figure 3.2: Local operation.

point x_c, y_c.
In general any vicinity can be used, although most common are square and sized 3×3, 5×5, 7×7 and so on. The operator used is usually called filter or kernel and is the same size as the vicinity used.
These operations are usually performed by two-dimensional convolution. The operation is executed on every pixel, so the straightforward approach is to store the image in a frame buffer, retrieve the vicinity and perform the window operation as in figure 3.2. This involves accessing each pixel $Width \times Height$ times each frame and it is highly inefficient. Hence, another approach more adapted to FPGA features, using latches to temporary store a vicinity and shift registers as row buffers. In stead of sliding the kernel across the image, the image is fed through the window as can be seen in figure 3.3.

When using 2-dimensional convolution, the border of the images can't be computed. This occurs due to the need to fill the whole vicinity around the pixel. In the cached algorithm, this is easily solved by calculating where to start and end each row and hence trimming the size of the resulting image. In the pipelined version, the solution is more complex. Incorrect values will be yield if this is not addressed since the previous frame will be computed in the next one. One approach could be to implement mechanisms to add to the row

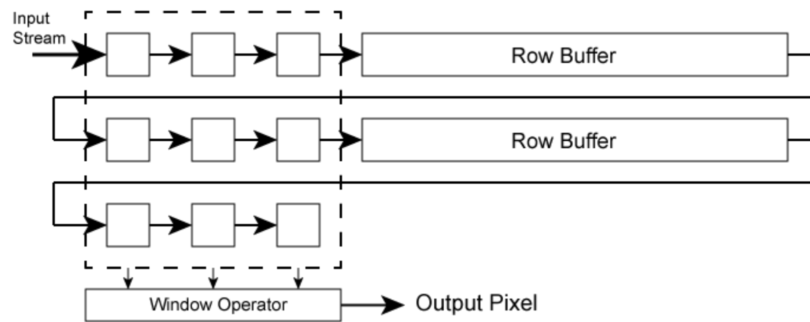

Figure 3.3: Pipelined convolution operation

buffers and to flush the pipeline. However, this implies stalling the pipeline while a flushing operation is taking place. A better option is to replicate the edge pixels of the closest border.

Global

The result of the operation in a pixel of the output image depends on most of the input image. This operations are, amongst others, Fourier transform or Wash-Hadamard transform.

The ability to perform these operations in real-time, and usually pipelined, is the great advantage for hardware implementations. MATLAB will be used to emulate the operations and hardware implementations will be discussed. Although MATLAB offers many functions to process images, loops and algorithms will be used to offer a better vision of how it could be implemented with VHDL.

3.2.2 Grayscale Conversion

Unless the application requires color recognition, most of the processing is better performed on a grayscale depiction of the video. This way the memory space and bandwidth needed are reduced. RGB often uses 256 levels of each color (8 bits), although high quality video already uses 10 bits color resolution.

Figure 3.4: Example of an original RGB frame from the video and the converted output

Having each red, green and blue color separated on a channel or $N \times M$ matrix, the most extended formula to calculate a grayscale representation is the weighted sum of these channels:

$$Grayscale = (R \times 0.2989) + (G \times 0.5870) + (B \times 0.1140)$$

However, VHDL can't handle floating point division unless a FP unit is used. To avoid using this unit, which at the same time adds much latency, division by powers of 2 can be easily performed by bit shifting the number, providing we are working with unsigned integers. Hence, this approximation can be used to good avail:

$$Grayscale = (R \times 0.25) + (G \times 0.5) + (B \times 0.125)$$

The simulation in Matlab loops over the frames of the video and performs the weighted sum of the matrix for each color. Note that the sum of the multipliers of each color is not equal to 1, hence a dynamic range compression is performed and the output image will lack of higher intensities. The source code can be consulted in the appendix as listing 1 and the result of applying it can be seen in figure 3.4.

The design shown in the figure 3.5 could be easily implemented in an FPGA.

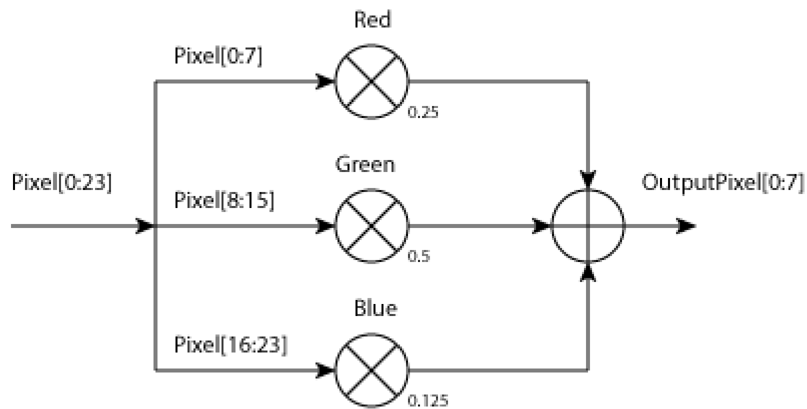

Figure 3.5: Diagram for RGB to Grayscale conversion

3.2.3 Histogram Equalization

The histogram of an image is the plot of the number of times (frequency) each level of gray is repeated. The horizontal axis represents the different gray levels (usually 256) and the vertical axis represents the number of pixels of these levels. A dark image would have most of its pixels in the leftmost part of the histogram, while a bright one will have them in the rightmost part. The histogram of an image with equally distributed gray levels along all its dynamic range (0-256) indicates it has good contrast.

The figure 3.6 shows that the image has a lot of bright pixels as well as some dark areas, but low frequency in median levels. In order to improve its contrast, one of the most used punctual operations is histogram equalization. Its goal is to transform the original histogram in another with uniform distribution of its gray levels[12]. However, since we are working with discrete values and not continuous, this is not possible in practice.

The cumulative image histogram can be defined as a mapping that counts the cumulative number of pixels in all of the gray levels up to the specified

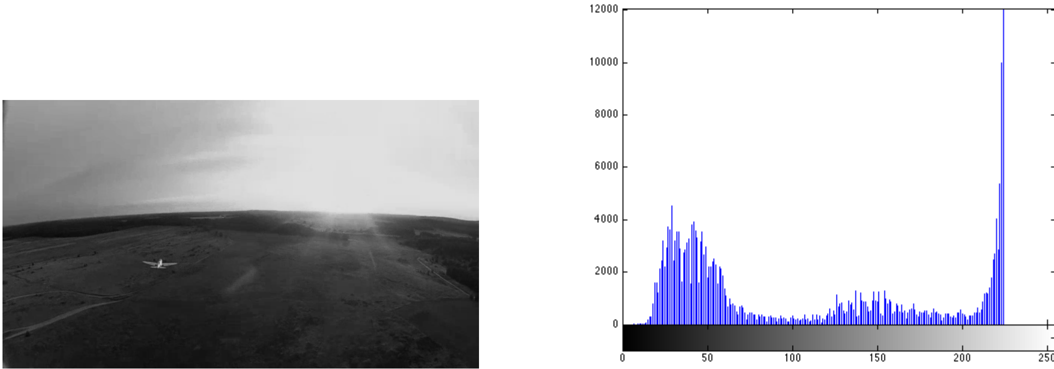

Figure 3.6: Histogram of a frame in grayscale

level. That is:

$$M_i = \sum_{j=1}^{i} m_i$$

This is a histogram equalization algorithm proposed by Milan Sonka[13]:

1. For a $N \times M$ image of G gray levels (usually 256), create an array H of G elements initialized to 0.

2. Form the image histogram: Scan every pixel and increment the relevant number of H — if pixel p has intensity g_p, perform:

$$H[g_p] = H[g_p] + 1$$

3. Form the cumulative image histogram H_c:

$$H_c[0] = 0$$
$$H_c[p] = H_c[p-1] + H[p], \quad p = 1, 2, ..., G-1$$

4. Set

$$T[p] = round\left(\frac{G-1}{NM} H_c[p]\right)$$

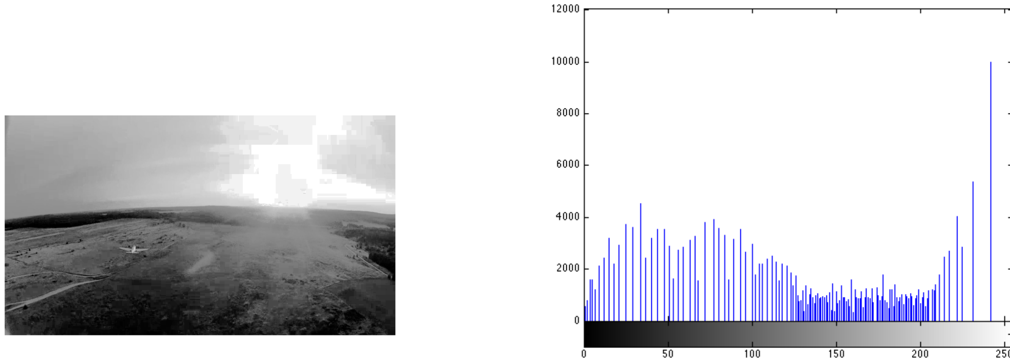

Figure 3.7: Frame from figure 3.6 after histogram equalization

5. Rescan the image and write an output image with gray levels g_q, writing:

$$g_q = T[g_p]$$

The result of histogram equalization over the frame from figure 3.6 can be seen in the figure 3.7. The gray levels have been spread along the dynamic range. Being a discrete environment, this is the best result obtainable. The Matlab code can be consulted in the listing 2. Histogram equalization is a global operation that requires reading the whole image and store it in a frame buffer to operate over it. This breaks the pipeline and adds a latency of 1 frame per second to the process.

3.2.4 Noise reduction

Usually, images have white noise, with zero mean and finite variance. Noise reduction helps improving the results of the later image processing, such as border recognition. The approach to remove this noise is by low-pass filters[14]. The most used are:

Median filter

Although the same windowing process is used as in a convolution, median filter doesn't perform any sum of the neighbor pixels. In stead, the median is the number separating the higher half of the neighbor elements from the lower

half. The median filter is a nonlinear digital filtering technique.
It needs a lot of computational effort since the median of the window of each element of the image has to be calculated. For small to moderate levels of noise, the median filter is demonstrably better than other filters at reducing noise and preserving borders.

Mean filter

This filter computes the average of all the pixels of a window centered on the output pixel. It is easily implemented as a convolution using kernels such as in figure 3.8. However, this filter is very aggressive with the borders despite its gain in performance.

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Figure 3.8: Mean filter kernel

Both median and mean kernels don't need to be square, although they are the most used. The size is usually 3×3, 5×5, etc. But it's been demonstrated that if the size is very big, that is 7×7 or 9×9, the results are not good. In stead, if the image is very noisy, it's better to apply smaller kernels in cascade, that is, apply the same filter to the result of a previous application.

Gaussian blur

This filter uses a Gaussian function, with a bell shape, that is less aggressive to the borders, since the center of the kernel will attenuate more the noise than the peripheral elements. However, the use of exponentials, and complex divisions in the gaussian function, makes it unsuitable for testing with simple FPGAs.

3.3 Segmentation

The goal of image segmentation is to separate the interesting objects from the rest, considered as background. Segmentation is usually considered as a process of classifying objects in an image. The level of segmentation depends

on the application.

In this case, we are analyzing videos taken from a moving camera (another plane) and hence background identification and subtraction is not an easy task. In stead, the frames must be carefully segmented and thresholded. An advantage of using FPGAs operating in real time is that the algorithms can be readjusted with feedback so thresholds and depth of segmentation can be adapted.

3.3.1 Border Detection

A border can be defined as a significant change in the intensity of the pixels of a region of an image, or as a high gradient on a point. Border detection consists of identify which pixels can be considered part of a border. The knowledge of this points allows the construction of the borders and hence the bounding of the frontiers of the regions on an image.

Border detection can yield information about the direction of the border, it's intensity (the difference in contrast) or it's direction (which part is brighter than the other). In addition, as image formation is not a perfect process, border detection has to deal with noise or fragmentation (some parts of the contour are lost).

In order to help the detector, a noise reduction process should be applied previously. A median filter will be applied before the border detection algorithm for the examples shown in the next sections.

Sobel Operator

It is one of the classic techniques in border detection based in computing a discrete approximation of the gradient of the grayscale image. It creates an image that emphasizes edges and transitions. At each pixel, the value of the output is an approximation of the norm of the gradient vector of the input pixel. The Sobel operator is based on convolving the image with a

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

Figure 3.9: Vertical and horizontal kernels for the Sobel operator

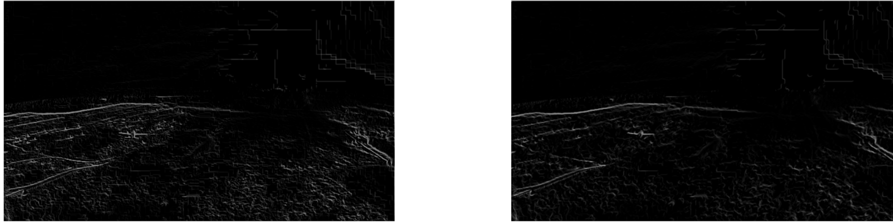

Figure 3.10: Left, border detection with Sobel operator over the frame from figure 3.6; right with median filter applied before border detection.

small and integer valued filter in horizontal and vertical direction which can be implemented in an FPGA with the design from the figure 3.3.

There are two kernels for the Sobel operator, one for vertical borders and other for horizontal as shown in figure 3.9. Additionally, other filters can be created to identify diagonal lines by rotating these filters. Both filters must be applied to the original image, generating two frames, so the real output must be the mean of the two. Ideally, the root mean square should be used, but in in an FPGA environment is easier to calculate the mean of two values in stead of calculate a square root.

Laplacian operator

Another classic approach for border recognition is by using the second derivative. The Laplacian operator is widely used to accentuate a border without taking into account its direction. The discrete version of this filter can be seen in figure 3.11. However, the second derivative is very sensitive to noise, and so usually it is used together with a Gaussian low-pass filter. Since the convolution operation is associative, the Laplacian and Gaussian filters can be convolved beforehand and then convolve this hybrid filter with the image to

0	1	0
1	-4	1
0	1	0

0.5	1	0.5
1	-6	1
0.5	1	0.5

Figure 3.11: Two versions of Laplacian filters. The one in the right includes diagonal lines

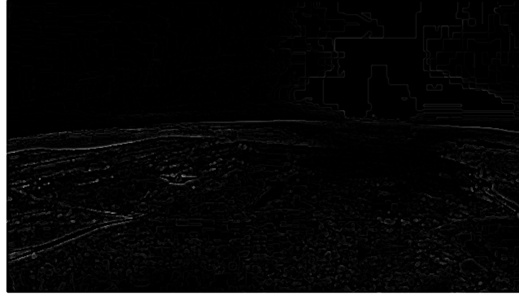

Figure 3.12: Laplacian operator over the frame from figure 3.6

obtain the borders.

During the tests performed with Matlab the Laplacian operator showed good results identifying corners (figure 3.12), but not so good in the integrity of the border where Sobel had better results. Hence the chosen operator was Sobel with a previous (double) median filter smoothing. Its Matlab code can be consulted at listing 3.

3.3.2 Object Recognition

An advantage of using Sobel operators to detect borders is that they return the intensity of the border, allowing thresholding to discard low-gradient transitions that can be noise or background while enhancing objects. The thresholding operation can be easily implemented in a pipeline since it is a punctual operation. A threshold T is the value applied to a transformation of input

frame $f(x, y)$ into output $g(x, y)$ as in the formula:

$$g(x,y) = \begin{cases} 1 & \text{if } f(x,y) \geq T \\ 0 & \text{if } f(x,y) < T \end{cases}$$

FPGAs allow this threshold to be dynamically calculated and adjusted in real time so it meets the requirements of the application. Matlab simulation can be found in listing 4.

The frame is now a cloud of border points. A high density of these points in a given area can be assumed as a separated object, hence applying a mean filter with a big enough window can yield the density of border points in the area surrounding each pixel as a fraction. Applying a threshold to the resulting frame with the percentage value that could be considered an object. An special Matlab function (listing 5) was created to perform these two operations and yield a binary image that suits the next steps.

3.4 Feature Extraction

3.4.1 Connected-component labeling

Connected-component labeling is an algorithm that labels subsets of connected components based on a given heuristic in a binary image. This serves to identify and separate the clouds of borders generated in the previous step.

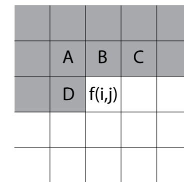

Figure 3.13: Already processed neighbors (8-connection)

The classic algorithm is performed in two passes. The algorithm examines the already processed neighbors of each pixel shown in figure 3.13 and determines which label to assign according to the procedure defined by Milan Sonka[13] as follows:

1. First pass: Search the entire image R by rows and assign a non-zero

value v to each non-zero pixel $R(i, j)$. The value v is chosen according to the labels of the pixel's neighbors.

- If all the neighbors are background pixels (with pixel value zero), $R(i, j)$ is assigned a new (and as yet) unused label.
- If there is just one neighboring pixel with a non-zero label, assign this label to the pixel $R(i, j)$.
- If there is more than one non-zero pixel among the neighbors, assign the label of any one to the labeled pixel. If the labels of any of the neighbors differ (*label collision*), store the label pair as being equivalent. Equivalence pairs are stored in a separate data structure— an equivalence table.

2. Second pass: All of the region pixels were labeled during the first pass, but some regions have pixels with different labels (due to label collisions). The whole image is scanned again, and pixels are re-Iabeled using the equivalence table information (for example, with the lowest value in an equivalence class).

Label collisions are pretty common. They are produced when the blob has an U shape. In the first pass the algorithm will detect each top part of the U as different labels and when it reaches the bottom there are 2 different labels in the neighbors. This conflicts are solved during the second pass. Hence, this algorithm is highly inefficient since it needs to perform two passes over each frame to correctly detect all objects and needs to use an intermediate frame buffer.

Some investigators have proposed implementations for this algorithm in real-time using FPGAs, exploiting their capabilities in parallelization and pipelining. Donald G. Bailey and Christopher T. Johnston first approach [15] avoided the need for buffering the image between passes by performing just one pass but had problems with worst case scenarios where the number of labels generated prior to merge was huge. In the next iteration[16], this problem was solved by reusing the labels. One last approach implemented the

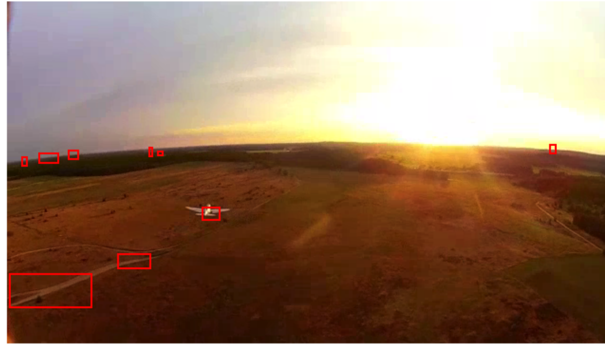

Figure 3.14: Object recognition plotted over original video

gathering of feature data for each region in parallel with labeling, enabling a single streamed pipeline implementation[17].

This algorithm relies on lookup tables for merging, and so adapting the algorithm to Matlab code is not an easy task. However, Matlab offers a function that labels the regions of a binary image, and was used in listing 6.

3.5 Representation

Once the frames have all their important objects labelled, a good way to display this data is by plotting it over the original video, since the processed one no longer holds the visual features of the original.

The labeled pixels of each frame have to be search to separate their regions and obtain their information such as maximum and minimum on both axis and/or their centroid. This data is later used to generate rectangles that will show the identified objects of the frame.

While the information gathering can be performed in the pipeline, the graphics generation can't. Taking into account the fact that they are applied over the original frame, the plotting doesn't need to wait for the frame to be

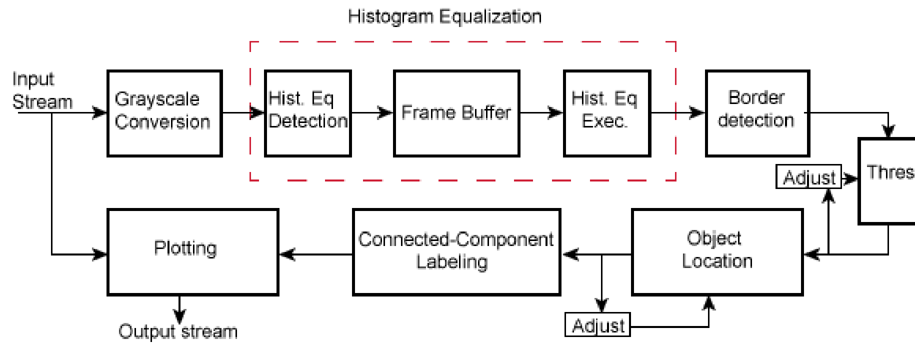

Figure 3.15: Sequence of steps

completely out of the pipeline.

In the Matlab function in listing 7, the Visual library is used to generate the graphics, for simplification sake. The final result can be seen in figure 3.14. The process is able to recognize objects, but it needs readjustment in real time to capture most of the important data.

3.6 Final result

The sequence of steps in Matlab can be found in Listing 8. Executing the procedure on a computer takes a couple of minutes while the delay on an FPGA should be of about 1-2 frames. Hence, the latency in serial computers increases linearly with the length of the video, making the process unaffordable in a stream, while the pipelined algorithm on an FPGA has a fixed latency.

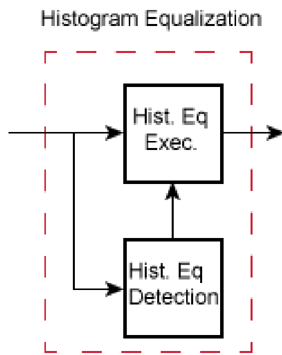

Figure 3.16: Alternative for Histogram Equalization

Parallel calibration can be introduced in the thresholding and object location steps when implementing the algorithm in the FPGA as can be seen in figure 3.15. The major delay is produced in the histogram equalization step because it needs a frame buffer to operate, hence stalling the pipeline. However, the detection step can be parallelized, reading all input pixels in the stream and calculating the lookup table. The grey value correction can then be executed with a latency of one frame, delaying only this step in stead of the whole pipeline. Luminosity doesn't change suddenly in open areas so the delay is affordable.

The output video generated in this work can be seen in `https://youtu.be/nqsTOEPAwyo`

Conclussions

FPGAs are a very powerful resource for aerospace applications. Their combination of hardware-powered computing with reconfiguration makes them very suitable for systems implemented in space probes, satellites and other equipment not physically accesible after launch. FPGAs also have proper tools to detect and recover from radiation upsets in comparison to electronic circuits and ASICs.

The implementation of real time video processing and computer vision algorithms in serial processors is not very suitable because it requires very low latency and hence a high computing power. Hardware implementations can pipeline the process and parallelize tasks to obtain a high throughput. The procedure developed in this work was created using low-level algorithms in order to be easily adaptable to hardware definition languages and pipelined entirely in an FPGA. Using the FPGA as a standalone processor rather than to provide acceleration allows for a more effective use of the resources in the FPGA.

The results obtained are positive. Through the different stages of the process the image is cleaned of most of the noise and background. However, results can be improved by exploiting the parallel execution of feedback gathering and readjustment of the thresholds and filters thanks to parallelization in hardware. Using multi-spectral video can also help differentiate objects from the background by their footprint in specific spectrums. Another possible approach is to execute parallel image stabilization and frame matching to perform background subtraction, acting as a low-pass filter.

Conclusiones

Las FPGAs son un recurso muy potente para aplicaciones aeroespaciales. La combinación de computación en hardware con reconfigurabilidad las hace muy adecuadas para sistemas implementados en sondas espaciales, satélites y demás equipamiento no accesible físicamente tras el despliegue. Las FPGAs además tienen herramientas apropiadas para detectar errores provocador por la radiación y recuperarse de ellos, al contrario que los circuitos electrónicos y ASICs.
La implementación de procesamiento de video en tiempo real y algoritmos de visión por computadora en procesadores convencionales no es muy efectivo ya que es necesaria muy poca latencia y por tanto gran potencia computacional. Las implementaciones en hardware pueden segmentar el proceso y paralelizar tareas para obtener un gran ancho de banda. El procedimiento desarrollado en este trabajo fue creado utilizando algoritmos de bajo nivel para ser fácilmente adaptable a lenguajes de definición hardware y ser segmentado enteramente en una FPGA. Utilizar la FPGA como procesador independiente en vez de servir como acelerador permite un uso más efectivo de los recursos de la FPGA.

Los resultados obtenidos han sido positivos. A través de las etapas del proceso la imagen es limpiada de la mayoría del ruido y del fondo. Sin embargo, los resultados pueden ser mejorados explotando la ejecución paralela de evaluación y reajuste de los filtros y humbrales gracias a la paralelización en hardware. Utilizar imágenes multi.espectrales también podría ayudar a separar objetos del fondo por su huella en espectros específicos. Otra posible aproximación consiste en ejecutar estabilización de video en paralelo, ajustar el marco y sustraer el fondo entre frames, consiguiendo un filtro paso-bajo.

Bibliography

[1] *Fly-By-Wire*. 2015. URL: http://www.airbus.com/innovation/proven-concepts/in-design/fly-by-wire/ (visited on 27/04/2015).

[2] NASA. *The Glass Cockpit*. URL: http://www.nasa.gov/centers/langley/news/factsheets/Glasscockpit.html (visited on 21/04/2015).

[3] NASA. *Synthetic Vision*. URL: http://www.nasa.gov/centers/langley/news/factsheets/SynthVision.html (visited on 24/04/2015).

[4] Eddy Vermeulen. *Real-time Video Stabilization For Moving Platforms*. 2007.

[5] Li Gang. *FPGA Implementation of Real-time Digital Image Stabilization*. 2014.

[6] The Air University. *Mulstispectral Imagery*. 2002.

[7] M. Fleury, R. P. Self and A. C. Downton. *Multi-spectral Satellite Image Processing on a Platform FPGA Engine*.

[8] Dagan White. *Considerations Surrounding Single Event Effects in FPGAs, ASICs, and Processors*. Ed. by Xilinx. 2012.

[9] Altera, ed. *Introduction to Single-Event Upsets*. 2013.

[10] Altera, ed. *Mitigating Single Event Upsets*. 2014.

[11] F. Abate et al. *New Techniques for Improving the Performance of the Lockstep Architecture for SEEs Mitigation in FPGA Embedded Processors*. 2009.

[12] Stephanie Parker and J. Kemi Ladeji-Osias. *Implementing a Histogram Equalization Algorithm in Reconfigurable Hardware*.

[13] Milan Sonka. *Image Processing, Analysis, and Machine Vision*. 2008.

[14] Roberto Rodriguez Morales and Juan Humberto Sossa Azuela. *Procesamiento y Análisis Digital de Imágenes*. 2011.

[15] Donald G. Bailey and Christopher T. Johnston. “Single Pass Connected Components Analysis”. In: *Proceedings of Image and Vision Computing* (2007).

[16] Ni Ma, Donald G. Bailey and Christopher T. Johnston. “Optimised Single Pass Connected Components Analysis”. In: *ICECE Technology, 2008. FPT 2008. International Conference on* (2008).

[17] Donald G. Bailey, Christopher T. Johnston and Ni Ma. “Connected components analysis of streamed images”. In: *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on* (2008).

Appendix: Source Code

Listing 1: Grayscale Conversion

```
function [ vidgray ] = vid2grayscale( vid )
    vidgray = uint8(zeros(size(vid,1),size(vid,2), size(
        vid,4)));
    for i=1:size(vid,4)
        vidgray(:,:,i) = vid(:,:,1,i)*0.25 + vid(:,:,2,i
            )*0.5 + vid(:,:,3,i)*0.125;
    end;
    vidgray=squeeze(vidgray);
end
```

Listing 2: Histogram Equalization

```
function [ videq ] = histEqvid( vid )
    videq=uint8(zeros(size(vid)));

    [M, N, T] = size(vid);
    G = 256;
    for t=1:T
        %Scan every pixel and increment the relevant
            member of H-- if pixel p has
        %intensity gp, perform
        H = uint32(zeros(1, G));
        for i = 1:M
            for j = 1:N
```

```
12                     gp = vid(i,j,t);
13                     Temp = H(gp+1)+1; % Adjust gp to go from
                           1 to G, not 0 to G-1
14                     H(gp+1) = Temp;
15                 end
16             end
17
18             %Form the cumulative image histogram Hc and
                 create the lookup table for step 4:
19             Hc = uint32(zeros(1,G));
20             T = uint32(zeros(1,G));
21             Hc(1) = H(1);
22             T(1) = round(((G-1)/(N*M)).*H(1));
23             for p = 2:G
24                 A = Hc(p - 1) + H(p);
25                 Hc(p) = A;
26                 B = round(((G-1)/(N*M)).*Hc(p));
27                 T(p) = B;
28             end
29
30             %Rescan the image and write an output image with
                   gray-levels gq
31             for i = 1:M
32                 for j = 1:N
33                     gp = vid(i,j,t);
34                     Temp = T(gp+1); % Adjust gp to go from 1
                           to G, not 0 to G-1
35                     videq(i,j,t) = Temp;
36                 end
37             end
38         end
39     end
```

Listing 3: Border detection with Sobel operator, smoothed with double median filter

```
function [ bVid ] = vidBorders( vid )
    %Creation of the two Sobel filters
    hSobel= [-1 -2 -1; 0 0 0; 1 2 1];
    vSobel= [-1 0 1; -2 0 2; -1 0 1];

    %Memory allocation for the buffers
    bVidBuff = zeros(size(vid(:,:,1)));
    bVidH = zeros(size(bVidBuff));
    bVidV = zeros(size(bVidBuff));

    for i=1:size(vid,3)
        %Double application of the median filter to
            smooth the frame
        bVidBuff(:,:,i) = uint8(medfilt2(vid(:,:,i)));
        bVidBuff(:,:,i) = uint8(medfilt2(bVidBuff(:,:,i)
            ));
        %Convoluting of both kernels with the input
            frame.
        %Matlab requires conv2 parameters to be of type
            single.
        %The 'same' parameter establishes that the
            output frame is the same size as the input
        bVidV(:,:,i) = uint8(conv2(single(bVidBuff(:,:,i
            )),vSobel, 'same'));
        bVidH(:,:,i) = uint8(conv2(single(bVidBuff(:,:,i
            )),hSobel, 'same'));
    end;
    bVid=uint8((bVidH+bVidV)./2);
end
```

Listing 4: Thresholding

```
function [ vidThres ] = vidThreshold( vid , threshold)
    %Memory allocation for the resulting frame
    vidThres = logical(zeros(size(vid)));

    for i = 1:size(vid,3)
        vidThres(:,:,i)=(im2bw(vid(:,:,i), threshold));
    end;
    %Return matrix is converted to uint8 to serve as
       input for next steps
    vidThres = uint8(vidThres.*255);
end
```

Listing 5: Object detection

```
function [ vidO ] = locateObjects( vidI , threshold)

    mean = (1/121)*(ones(11,11));
    vidBuff = uint8(zeros(size(vidI)));
    vidO = logical(zeros(size(vidBuff)));

    for i=1:size(vidI,3)
        vidBuff(:,:,i) = uint8(conv2(single(vidI(:,:,i))
            , mean, 'same'));
        vidO(:,:,i)=(im2bw(vidBuff(:,:,i), threshold));
    end
end
```

Listing 6: Connected-Component Labeling

```
function [ vidO, nObjects ] = cclVid( vidI )
    nObjects = uint8(zeros(1,size(vidI,3)));
    vidO = uint8(zeros(size(vidI)));
    for i=1:size(vidI,3)
        %The Connected Component Labeling algorithm is
```

```
                very unefficient to implement in Matlab due
                to the fact that Matlab is not prepared to
                work with complex data structures like lookup
                 tables, stacks, etc.
6           %However, Matlab offers a function that performs
                 labeling on binary images
7           [vidO(:,:,i),nObjects(i)] = bwlabel(vidI(:,:,i))
                ;
8       end
9   end
```

Listing 7: Objects representation on video

```
1   function [ vidO ] = plotVideo( vidRGB, vidLabeled,
       nObjects )
2       %In order to plot figures in Matlab, an
           shapeInserter object, included in the vision
           library, is needed
3       shapeInserter = vision.ShapeInserter('LineWidth',2,'
           BorderColor', 'Custom', 'CustomBorderColor', [255
            0 0]);
4       vidO=uint8(zeros(size(vidRGB)));
5
6       for t=1:size(vidLabeled,3)
7           %Initialize the output with the original, color
                video
8           vidO(:,:,:,t) = vidRGB(:,:,:,t);
9           [y, x]=ndgrid(1:size(vidLabeled,1),1:size(
                vidLabeled,2));
10          %Iterate over the detected "blobs" of each frame
11          for k=1:nObjects(t)
12              %calculate the max and min dimensions to
                    plot rectangles
13              maxx=max(y(vidLabeled(:,:,t)==k));
```

```
14                minx=min(y(vidLabeled(:,:,t)==k));
15                maxy=max(x(vidLabeled(:,:,t)==k));
16                miny=min(x(vidLabeled(:,:,t)==k));
17                %Define the initial points and dimension of
                     the rectangle
18                rectangle = int32([miny, minx, maxy-miny,
                     maxx-minx]);
19                %Draw the rectangle
20                vidO(:,:,:,t) = step(shapeInserter, vidO
                     (:,:,:,t), rectangle);
21            end
22        end
23    end
```

Listing 8: Sequence of stages of video processing

```
1 load('mov');
2 gray = vid2grayscale(mov);
3 eq = histEqvid(gray);
4 borders = vidBorders(eq);
5 bordersThres = vidThreshold(borders, 0.35);
6 obj = locateObjects(bordersThres, 0.15);
7 [labeled, nObjects] = cclVid(obj);
8 output = plotVideo(mov, labeled, nObjects);
9 implay(output);
```