PROGRAMMING OF PROCESSING ALGORITHM OF THE IMAGE ON GPU USING CUDA TECHNOLOGY

Final Degree Project /

Trabajo Final de Grado

Degree in Video Game Design and Development

Author: David Insa Moreno

UJI Supervisors: Sven Casteleyn & Carlos Granell Canut

June 2019

ABSTRACT

The reason for the TFG is related to the need that the company AUTIS has, to find a solution to reduce the use of the CPU during the taking and the processing of images, using LabVIEW software, in the project of dynamic inspection of the defects in the car bodies. That is why this work is based on research and the creation of tools that allow you to perform the maximum number of possible functions on the graphic card. In order to achieve this, it has been learned LabVIEW and CUDA. The first is the work environment that is used in the project and in the company. And the second, because it is an architecture to work with the graphics card directly. Once it has been learned, it has had to move some of the common algorithms in the process of images made in the CPU to the graphic card by CUDA doing them through parallel programming, arriving to create a DLL and integrating it in LabVIEW.

UNIVERSITAT JAUME I

CONTENTS

Contents

1

INTRODUCTION

Contents

AUTIS, a company dedicated to the industrial automation sector, offered me the possibility to introduce myself as a programmer in one part of one of its most important projects. The mentioned part consists in the reduction of CPU consumption so as not to overload the systems of inspection of body surfaces that they possess. To do this, it was thought to perform the greatest possibility of actions that are carried out, in said inspection, with the graphic card (which from now on I will refer to it as a GPU or Graphic Processing Unit), since the actions that are carried out they are much more suitable and would suppose a liberation of work of the CPU.

So, my training stage began in a new programming and performance realm that would allow algorithms to process images in GPU.

1.1 Work Motivation

The simple idea of learning a new way of programming as it is parallel programming and also be able to apply it optimally in the processing of images, creates a great curiosity about the extent to which the processing algorithms can become parallelizable and also be the more efficient.

1.2 Objectives

The objectives set by the company are:

- Analysis of the functions that are currently used in their systems for surface inspection.
- Approach of these functions for its execution on GPU.
- Valuation of existing bookstores in the market, such as OpenCV CUDA, NPP NVIDIA and others that can be assessed during the course of the project.
- Programming of a set of algorithms for surface inspection with CUDA technology.

The personal objectives are:

- Learn as much as possible.
- Be able to be useful with everything learned.
- Have enough dexterity to transform / adapt sequential algorithms to a parallelizable format.

1.3 Envioronment and initial size

At the time of starting the project, there was no idea of the LabVIEW environment or parallel programming. And a learning has been started from 0 of many concepts. For example, as a preview, LabVIEW is a very different kind of programming that requires a lot of knowledge of all the tools it has, so as not to make programs that are already done and lose time that way. The environment when learning and receiving training has always been positive, an aspect that greatly assimilates.

1.4 Related subjects

After all the subjects given during the degree, I opted to do the TFG oriented to programming subjects. For the most part, the subjects most related to all the work have been Programming I and Programming II, because they teach the basic concepts; Algorithms and Data Structures, since in it C ++ is learned and ways to use optimal recursiveness and Operating Systems, you learn about aspects related to parallel programming.

2

PLANNING AND RESOURCES EVALUATION

Contents

2.1 Planning

As can be seen in the diagram, the project was developed in five phases, where three of which were learning and the other two of analysis and research.

Since one of the most used tools by the company is the LabVIEW software, there was a learning stage with it. During this stage, it has been learned the basics to start doing their own projects. In addition, it has been observed and moved with their own architecture that AUTIS uses as the basis of your projects in this software.

Once the first phase was finished, the operation of five methods widely used in the project had to be analyzed. This analysis consisted in carrying out tests with these methods on images of bodies to contrast what results were obtained when applying one or the other.

Subsequently, it was necessary to investigate how these methods were being implemented, currently in the project, as a basis for their internal functioning and thus be able to carry them out in the future in parallel.

In the fourth phase, it was an introduction in the world of parallel programming. In it, it was possible to learn a number of concepts and forms of programming, which in previous circumstances were unknown. It was learned more about CUDA, since the company opted to introduce all its concepts in all defect inspection projects on body surfaces, because it represents a quite remarkable software improvement for the equipment they use. But apart from CUDA, it has been possible to introduce other parallel programming concepts such as multiprocessor programming (OpenMP), programming of computer clusters (MPI), new models of parallel programming such as OpenACC, ...

Finally, it has been necessary to apply all the concepts studied during the previous phases to get to create a palette (is the term used to call the set of data and functions in LabVIEW) of image processing. In this phase, a lot of time has been invested in realizing algorithms with Visual Studio, implementation of these algorithms in LabVIEW through the creation of DLLs and debugging errors that have been appearing when combining LabVIEW with CUDA.

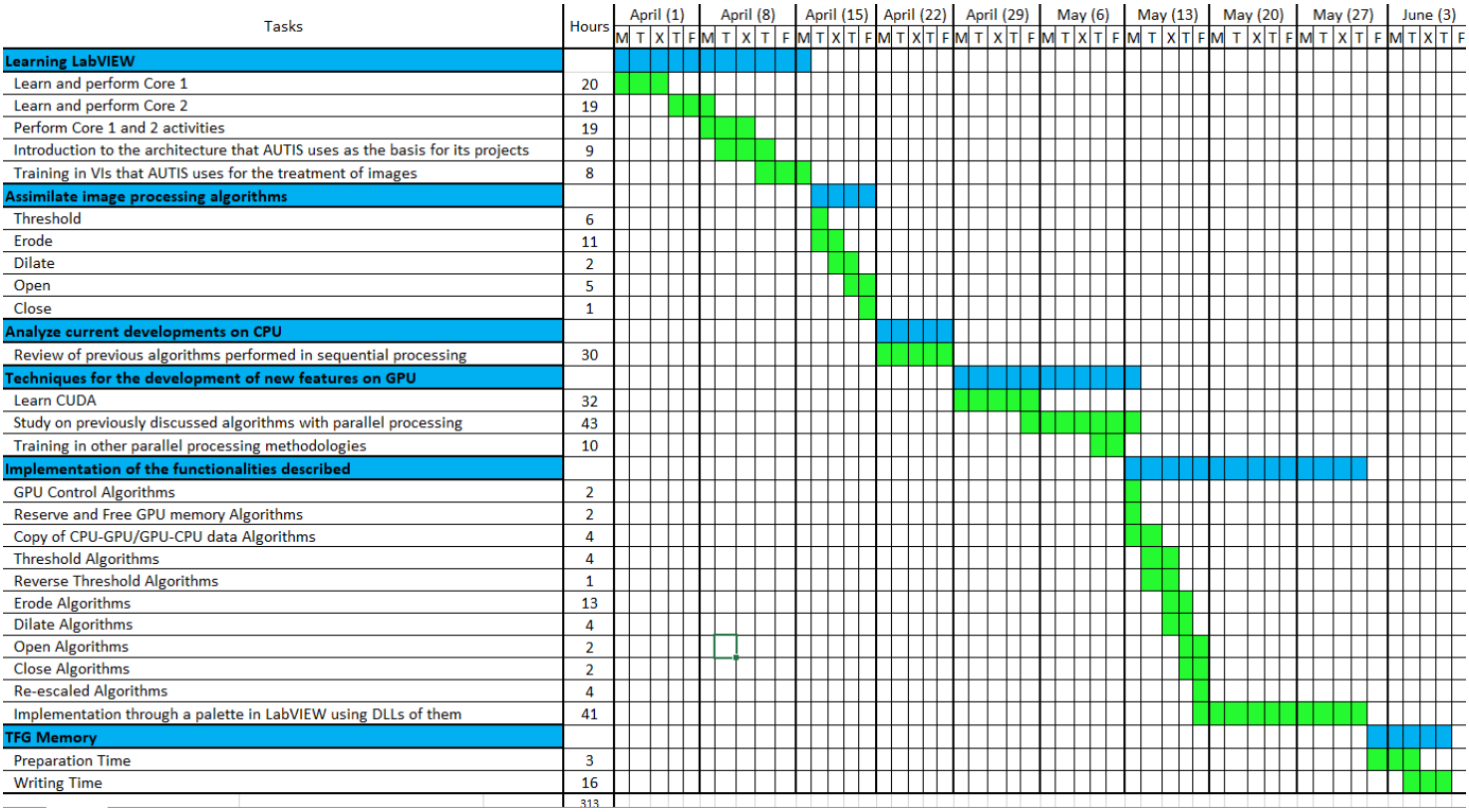


Figure 2.1: Gant chart about the realized tasks and dedicated hours.

2.2 Resource evaluation

Regarding the amount of human resources invested during the learning stage, it can vary. This is because all the concepts of this work can be learned autonomously without the need for professionals, since there are material resources such as guides or books about each of the environments (LabVIEW, Visual Studio and CUDA). The downside of not using professionals for learning is that the time spent can increase. Regarding my situation, enough were used for LabVIEW and less for the other two, due to external work circumstances.

The amount of economic resources is quite high, since the licenses of LabVIEW and internal libraries (those of IMAQ). By searching the internet you can get books that explain the whole functioning of CUDA, which as a basis to have basic concepts are fine. But they don't help to obtain the skill of parallel programming. In my circumstance, it has been had an online training course certified and made by NVIDIA. In addition to receiving tutorials and information about CUDA from professionals in the field outside the company.

The time spent can also vary according to the amount of previous resources that have been decided to invest. It is also influenced by the type of person who does it, since they are not completely clear concepts and it is necessary to understand very well.

So that, the economic estimate of human resources would be around 3800 euros, counting all the hours spent in conducting training classes. The economic estimate of material resources, would be around 3600 euros. This counting that the license of LabVIEW is the cheapest (400 euros per year), in my case the licenses used are the professionals (around 5000 euros per year). The remaining amount is based on CUDA courses, library licenses in LabVIEW and a team with the necessary. Therefore, the total estimate would be about 7400 euros.

3

SYSTEM ANALYSIS AND DESIGN

Contents

3.1 Requirements analysis

In this section, it can be seen the schemes about the palette.

3.1.1 Functional requirements

The user needs to perform certain functions that allow an image to be treated with the GPU. For this purpose, operations that prepare the image for processing, send and receive operations, and memory reserve and release operations are required.

Input:	Minimum and Maximum Numbers & Black / White Image
Output:	Binarized Image
Description:	The user will apply a threshold to the image. Which is based on binarizar the image. It puts 1 in the pixels of the image whose value is between the minimum and the maximum of the numbers entered. And to 0 those that are not inside

Table 3.1.1: Functional requirement << Algortithm Threshold >>

Input:	Minimum and Maximum Numbers & Black / White Image
Output:	Binarized Image
Description:	The user will apply a reverse threshold to the image. Which is based on binarizar the image. It puts 0 in the pixels of the image whose value is between the minimum and the maximum of the numbers entered. And to 1 those that are not inside

Table 3.1.2: Functional requirement << Algortithm Reverse Threshold >>

Input:	Radio Number & Binarized Image
Output:	Processed Image
Description:	The user enters a binarized image and applies an erode algorithm. This algorithm gives each a pixel the minimum of the sum of its neighbors. The number of neighbors depends on the radio.

Table 3.1.3: Functional requirement << Algortithm Erode>>

Input:	Radio Number & Binarized Image
Output:	Processed Image
Description:	The user enters a binarized image and applies an dilate algorithm. This algorithm gives each a pixel the maximum of the sum of its neighbors. The number of neighbors depends on the radio.

Table 3.1.4: Functional requirement << Algortithm Dilate>>

Input:	Radio Number & Binarized Image
Output:	Processed Image
Description:	The user enters a binarized image and applies an open algorithm. This algorithm is base don using firstly a erode and secondly a dilate. It is used to eliminate noise in the picture.

Table 3.1.5: Functional requirement << Algortithm Open>>

Input:	Radio Number & Binarized Image
Output:	Processed Image
Description:	The user enters a binarized image and applies an close algorithm. This algorithm is base don using firstly a dilate and secondly a erode. It is used to fill holes in areas with 1 in their pixels.

Table 3.1.6: Functional requirement << Algortithm Close>>

Input:	Width and Height of the Image
Output:	Nothing
Description:	The user will reserve memory in GPU for the image with which he will try.

Table 3.1.7: Functional requirement << Algortithm to Reserve GPU memory>>

Input:	Nothing
Output:	Nothing
Description:	The user will free memory in GPU.

Table 3.1.8: Functional requirement << Algortithm to Free GPU memory>>

Input:	Image
Output:	Nothing
Description:	The user needs to copy the image from GPU to CPU

Table 3.1.9: Functional requirement << Algortithm to Copy data in GPU>>

Input:	Space for a Processed Image
Output:	Processed Image
Description:	The user needs to copy the image from CPU to GPU

Table 3.1.10: Functional requirement << Algortithm to Copy data in CPU>>

3.1.1 Non-functional requirements

The user must use some extra functions to be able to perform the image processing. As will be explained later, in the GPU it is necessary to reserve memory to perform operations, pass the data, which will be used, from CPU to GPU and vice versa and free the used memory. Therefore, two or four (depending on the implementation of the program) functions that allow error-free processing will be used. In addition, everything must be connected sequentially.

Algorithms that process images work with arrays of numbers, instead of image files. So that the user can work without problem. It will be provided with methods that transform the image file into arrays. In addition to functions to be able to visualize the original and the result.

So that everything is clear, each implemented function will have a detailed description of what it does, where it should be located and what input and output parameters it uses. In addition, it will have a physical manual and PDF with all the methods used.

3.2 System design

CASE USE DIAGRAM

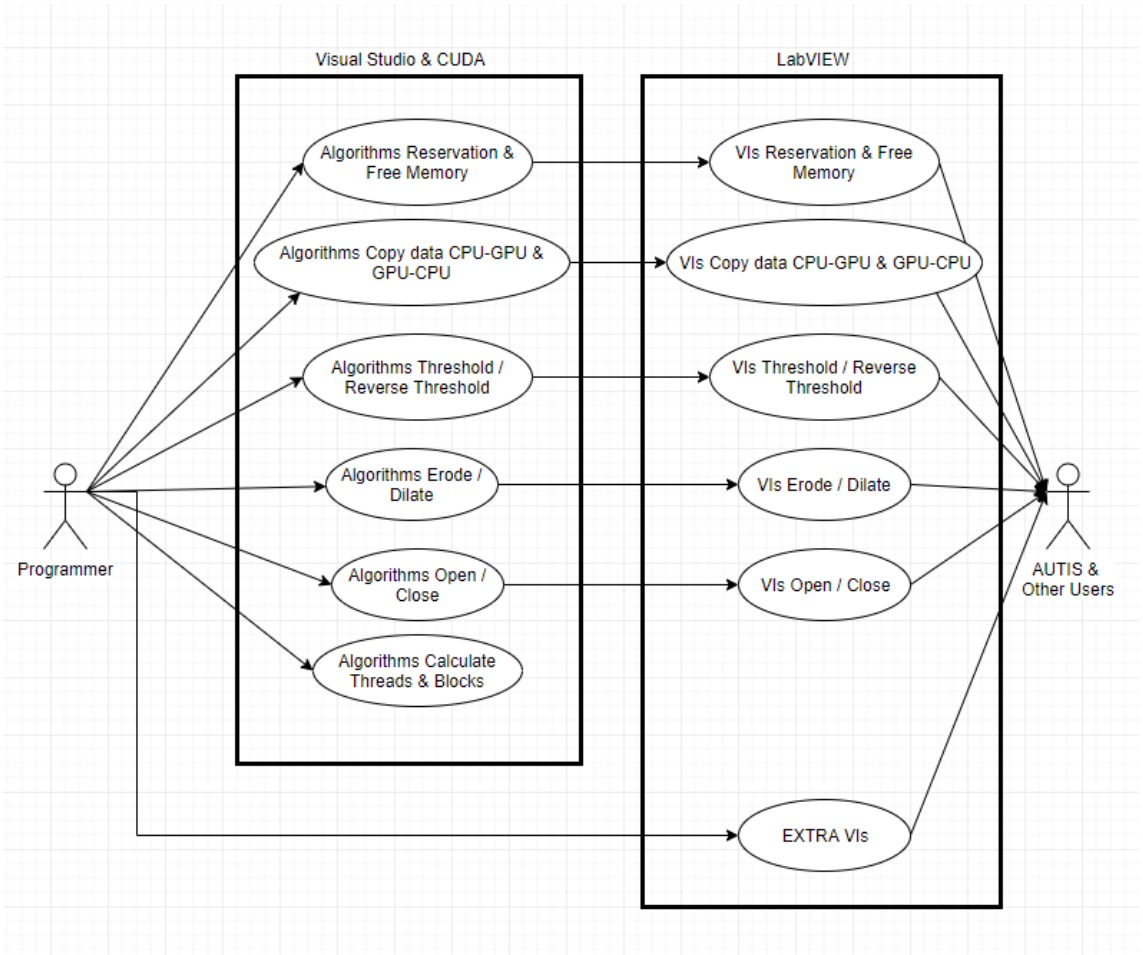


Figure 3.2.1: Case use diagram

The programmer is responsible for creating functions that allow operations with the GPU through Visual Studio. Users work only with LabVIEW. That's why the programmer passes those functions in Visual Studio to LabVIEW and also creates extra functions so users can connect all of them without problems.

CLASS DIAGRAM

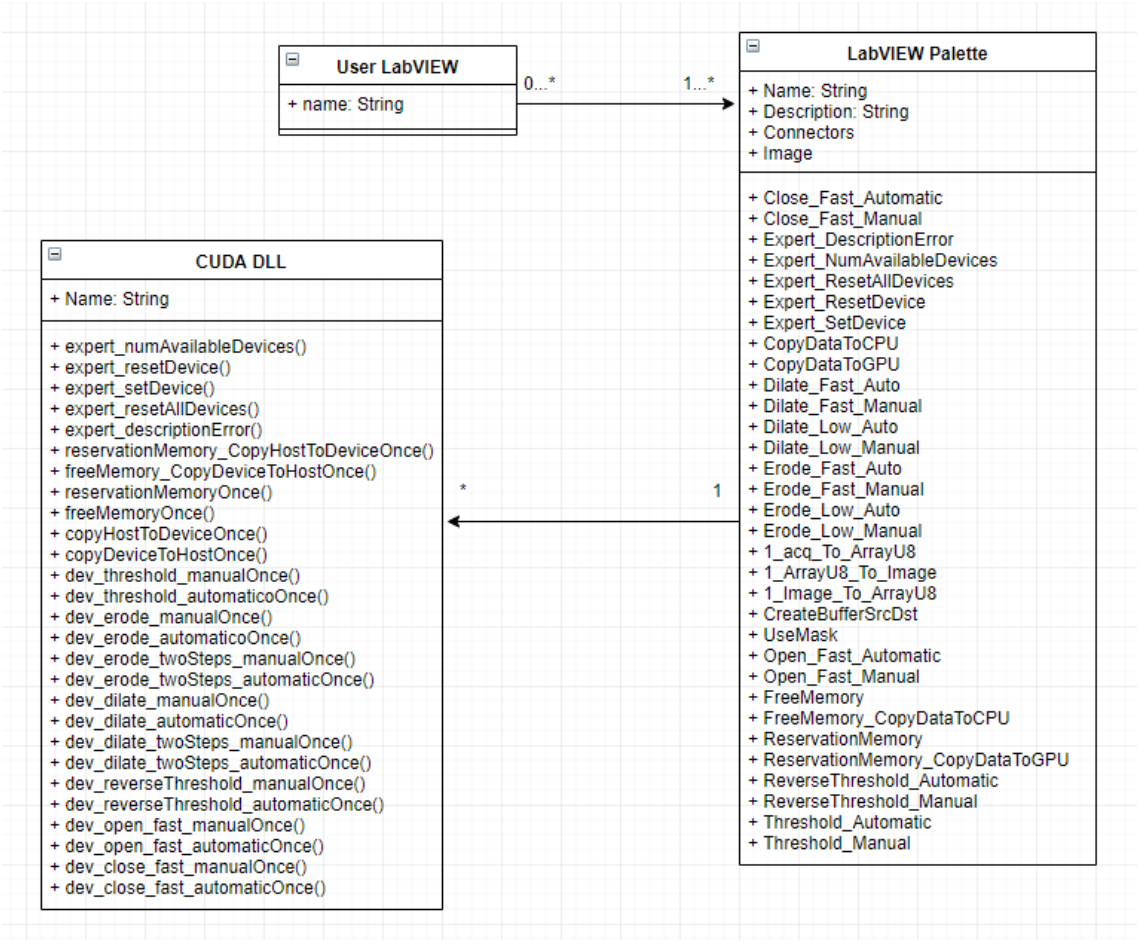


Figure 3.2.2: Class diagram

The user of LabVIEW creates a project or applies in another the set of functions created for the treatment of the image. These functions call those created in Visual Studio. Where each Visual function can be used only by a LabVIEW function and each user can use the LabVIEW feature set between 1 and infinite times.

ACTIVITIES DIAGRAM

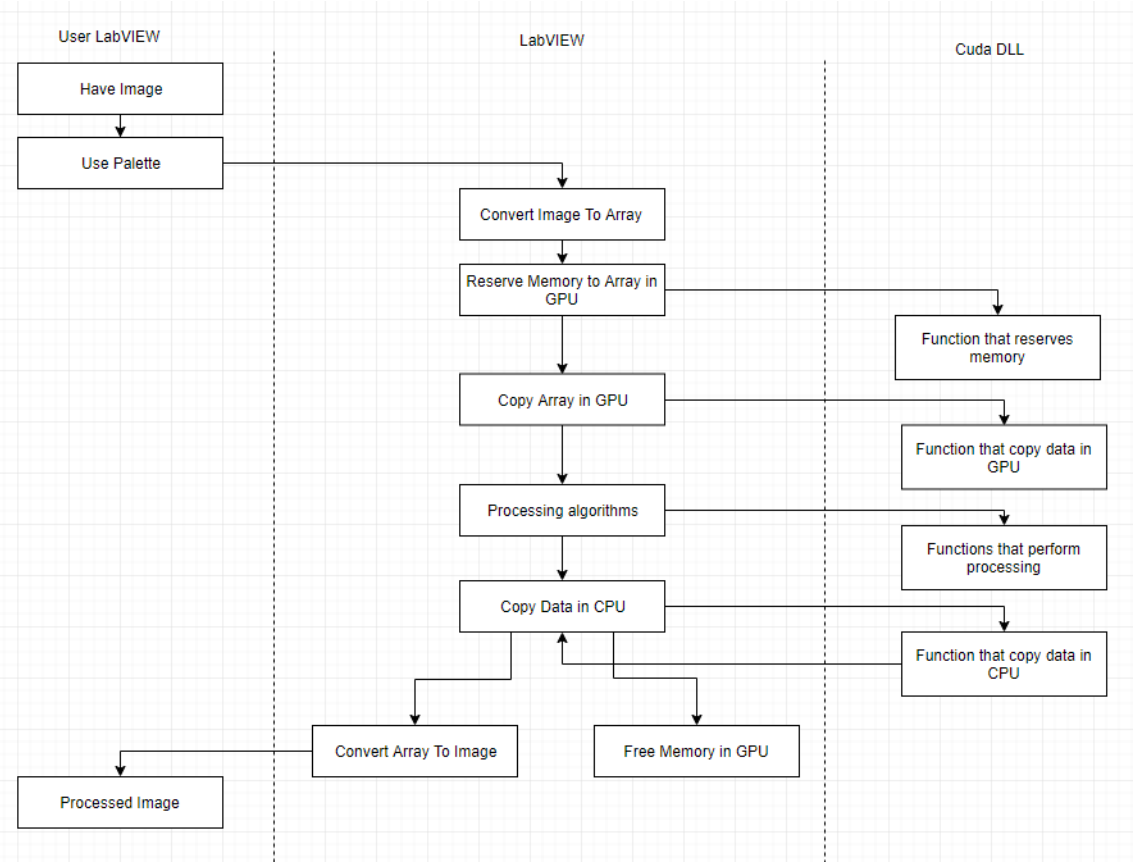


Figure 3.2.3: Activities diagram

The user who uses the image processing functions, gives an image to LabVIEW. This set of functions is responsible for performing all operations to send, process and receive the image to the GPU. Until the image acquisition operations are performed, it remains in GPU and can not be used by LabVIEW.

INTERACTION DIAGRAM

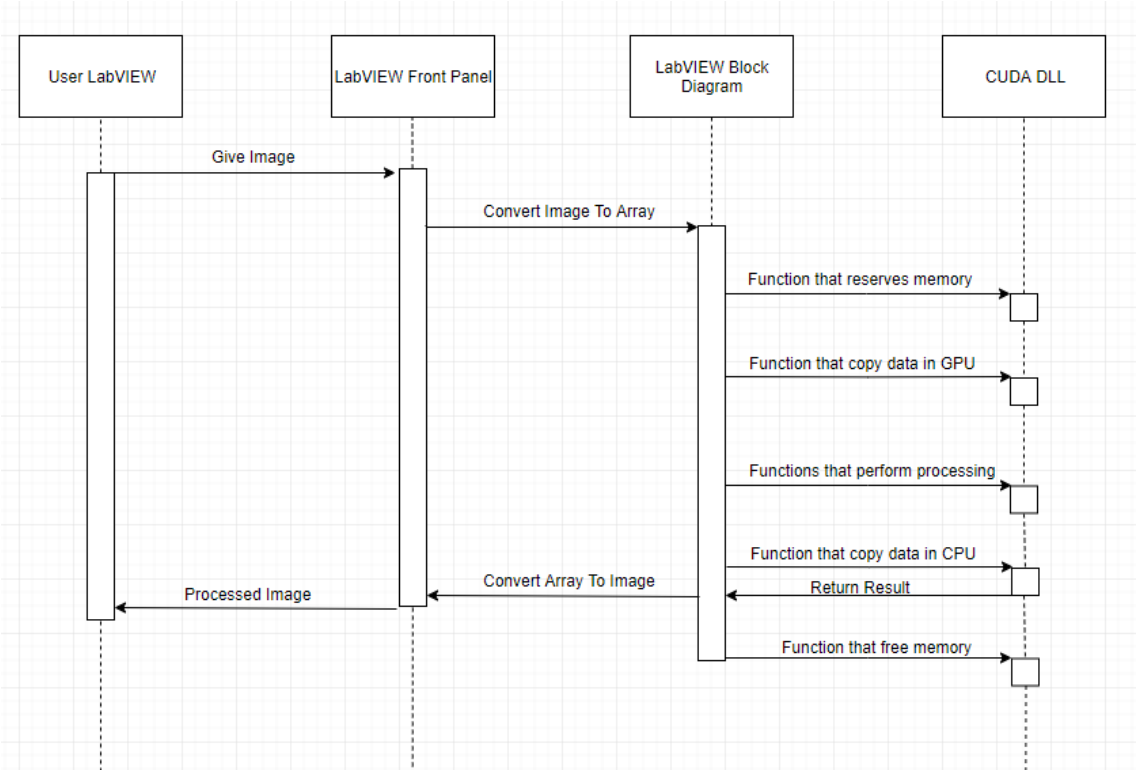


Figure 3.2.4: Interaction diagram

The user gives an image to LabVIEW and it is displayed in the user interface, LabVIEW internally performs the processing operations. And in turn, these functions call the library that allows us to send, process and receive the image in GPU. Once the processing is finished, the final result of the image is returned.

3.3 System architecture

To be able to carry out this project, at the software level you need to have licenses from LabVIEW, IMAQ, Visual Studio and CUDA. At hardware level, it is mainly required to have a graphic card compatible with CUDA.

The processor of the computer is an important factor when it comes to obtaining a greater number of processed images, but it is not vital if what is required is not the quantity but the times. Therefore, if you want an optimal image processing, we recommend devices with a good RAM, graphics card and processors.

The equipment used for the project is:

- Processor: Intel® Core™ i7-8700K CPU 3.70 GHz
- RAM: 16 GB
- Graphic Card: GeForce GTX 1050 Ti (4GB)
- Hard Disk: 500 GB
- Monitor: HP VH27

3.4 Interface design

The interface of this project does not exist as it can be that of an application or video game. A palette of VIs (or Virtual Instruments, are the program on wich LabVIEW is based) has been designed and these are distributed by folders. The only design part that exists is the visual and documented part of each VI.

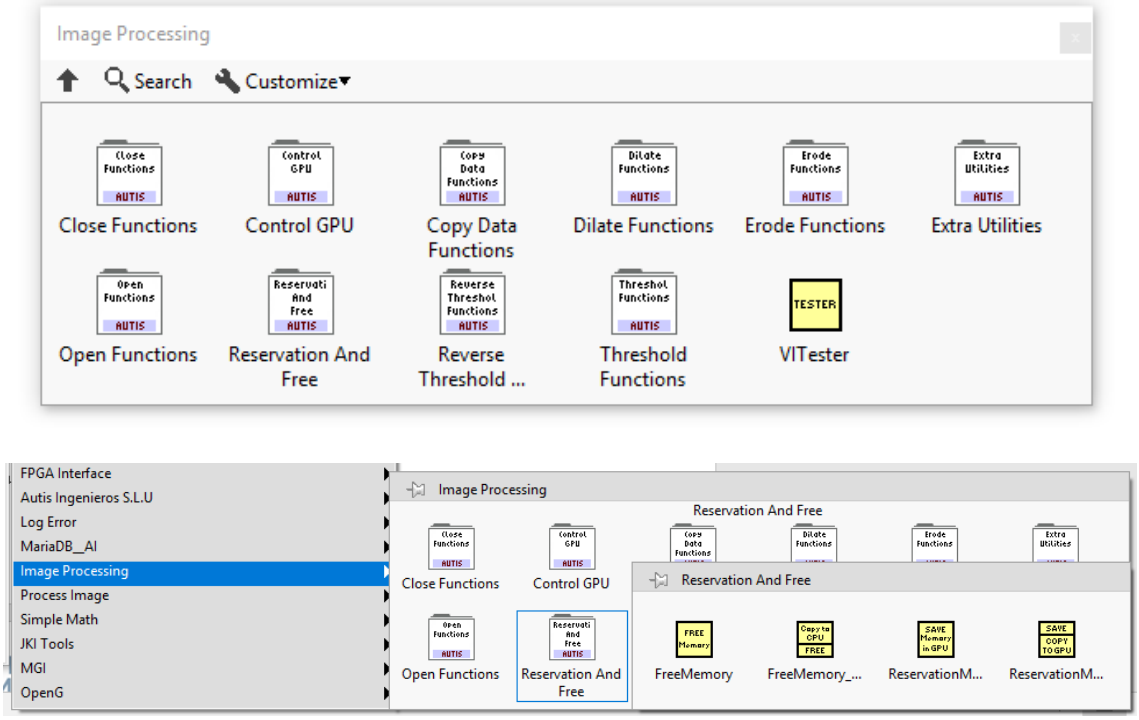


Figure 3.4.1: Final result of the palette.

It has been tried that each VI has a representative icon, an internal order and a small description of its use and utility.

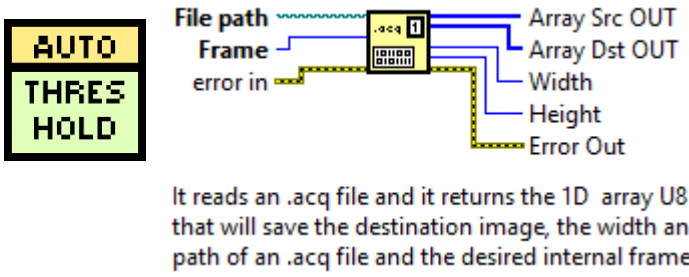


Figure 3.4.2: Example of VI icon, inputs and outputs parameters and a VI description.

Likewise, a VI testet has been carried out with everything implemented, but with a very basic design. In it, we can compare times of each type of VI implemented in the palette, allowing the modification of the input parameters and adding own test files.

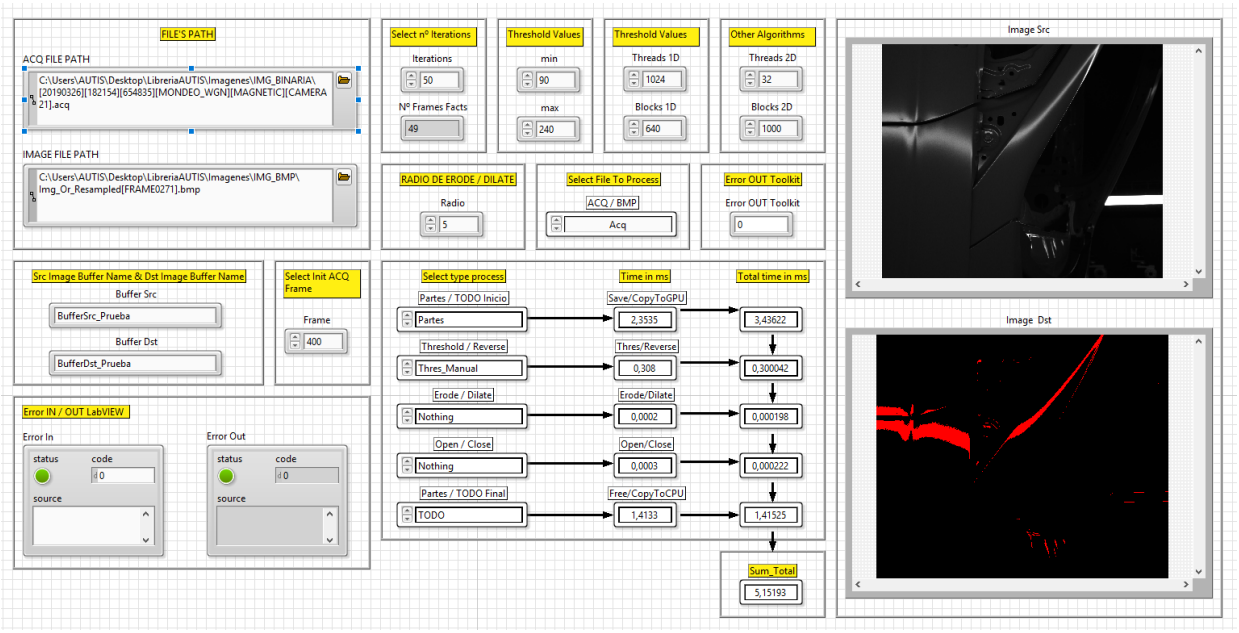


Figure 3.4.3: Front Panel Tester.

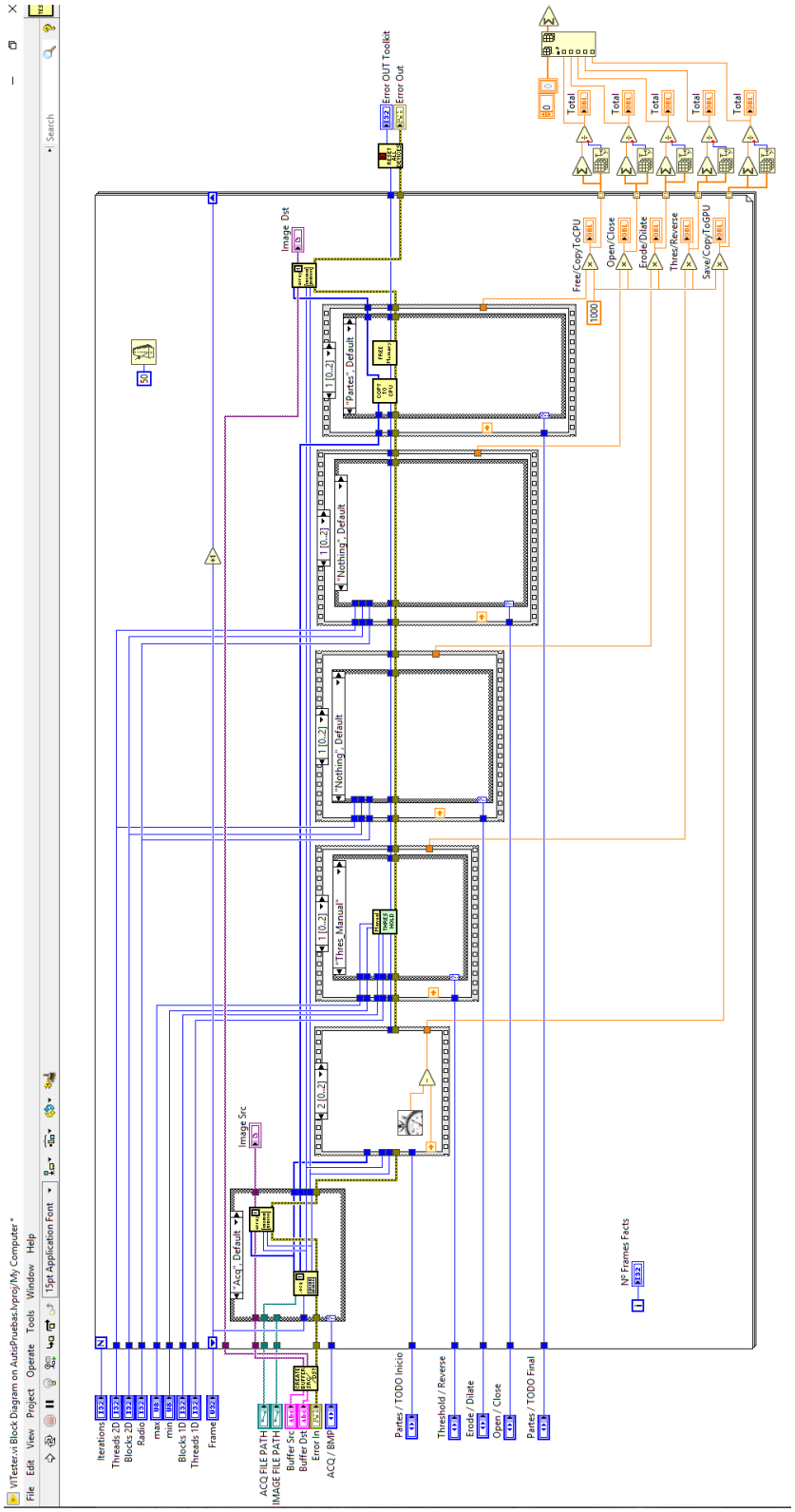


Figure 3.4.4: Block Diagram Tester.

4

WORK DEVELOPMENT AND RESULTS

Contents

4.1 Work development

4.1.1 Learning in LabVIEW

During the first two weeks, they introduced me to the new graphic programming environment, LabVIEW, which I would use as the basis of my project. The formation of this new language is distributed in two blocks, Core 1 and 2.

LabVIEW is based on the creation of programs called VI or Virtual Instruments, each VI consists of three parts: Front Panel, is the user interface; Block Diagram, contains the source code; and Icon / Connector Panel, represents the VI and allows connect with others.

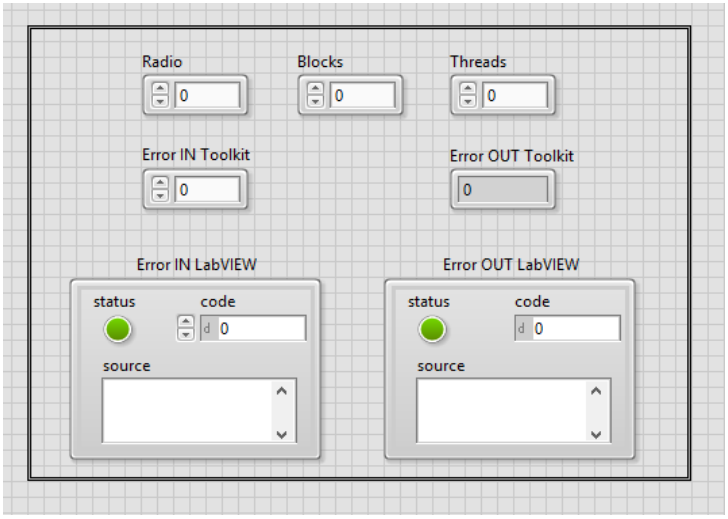


Figure 4.1.1.1: Front Panel Example.

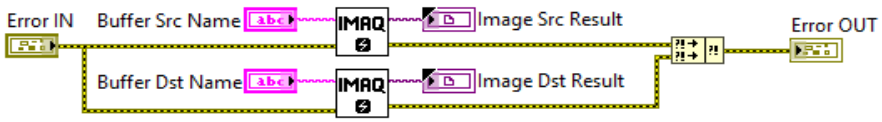


Figure 4.1.1.2: Block Diagram Example.

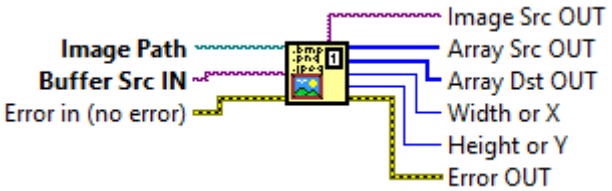


Figure 4.1.1.3: VI Example with its identification Icon and all the connectors it has.

CORE 1 is distributed as follows:

- LabVIEW and distribution.
- A first application.
- Problem solving and debugging of VIs.
- Loops.
- Data structure.
- Decision making structure.
- Modularity.
- Access to files.

LabVIEW and distribution: This part presents the software, how it is formed and how a project is created. That is a little what it has been had previously introduced.

A first application: Here is explained the flow of data that manages, which is based on the transfer of data through connections between the different elements within a Block Diagram and always goes from left to right and from top to bottom. A large number of factors must be taken into account in order to carry out operations correctly and that the data flow does not collapse. The elements of LabVIEW can have connections of entrance, of exit or both. Those who don't have input will always run first and those who have input must wait for the flow to reach them.

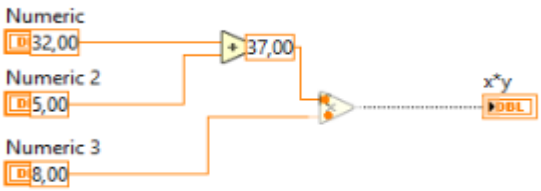


Figure 4.1.1.4: Data flow Example. In this case, by the connections, it will first make the addition and then the multiplication.

In addition, the type of data it uses is revealed: numeric data of 8, 16, 32 and 64 bits; singles, doubles, booleans, strings, etc.

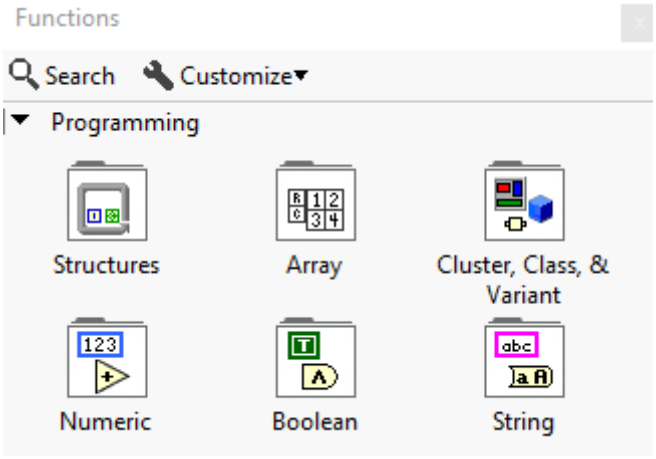


Figure 4.1.1.5: A part of general palette interface

Problem solving and debugging of VIs: In this section is exposed how to prevent all kinds of errors that can occur and how to solve them to avoid unwanted effects. LabVIEW has a large amount of aid for debugging projects, a manual on all errors that may appear and for dealing with errors.

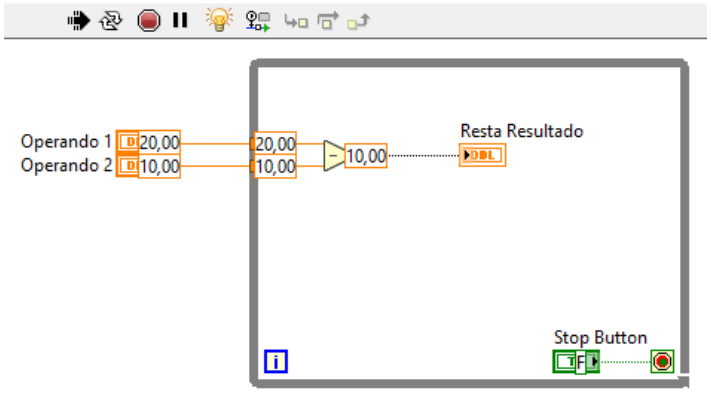


Figure 4.1.1.6: Debug mode example. It is activated with the menu light

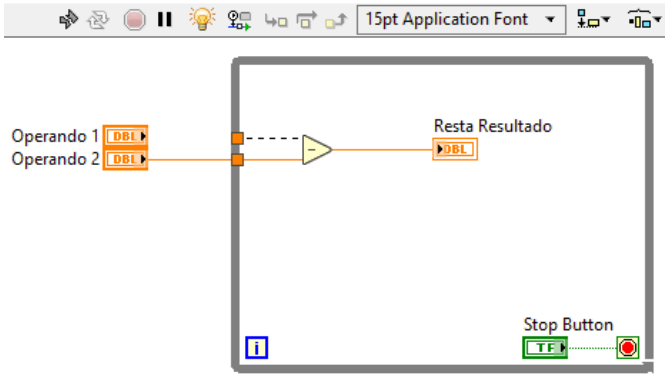


Figure 4.1.1.7: VI with error example. It is indicated with the broken arrow.

Loops: We talk about the "for" and "while" loops and introduce a new concept that is the temporalization of loops, this is because, if we don't apply a runtime between each loop cycle, LabVIEW consumes a lot of CPU resources. With timers, we better distribute CPU usage for processes.

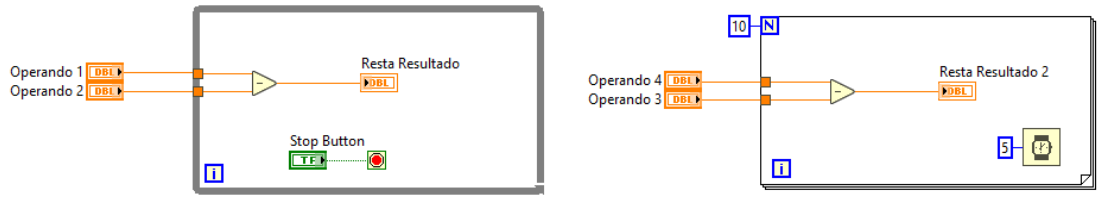
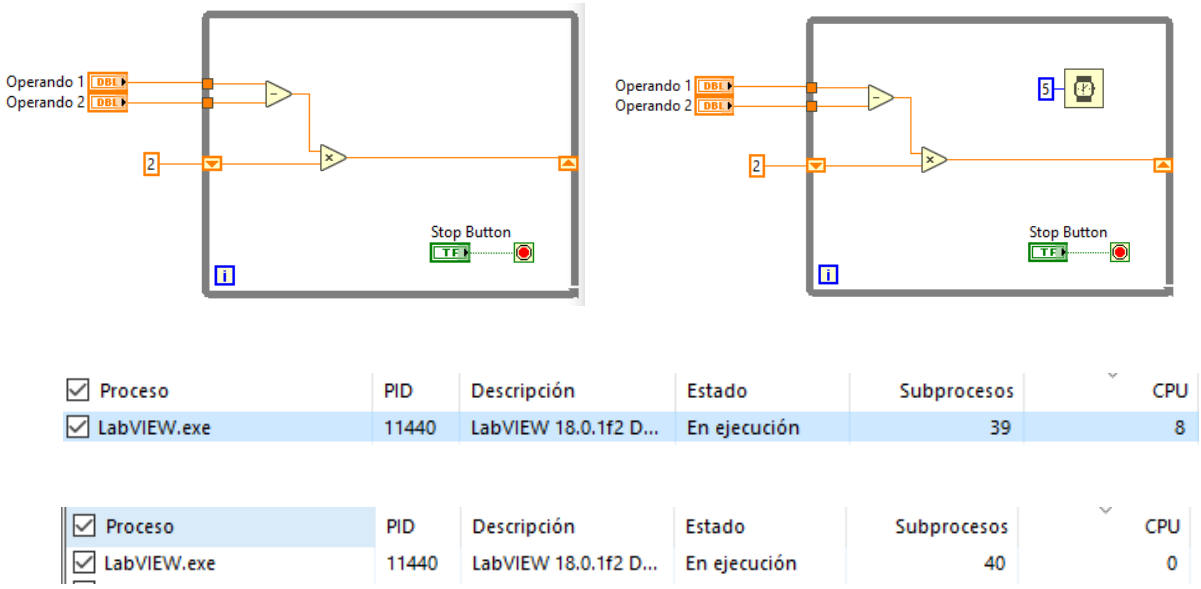


Figure 4.1.1.7: While loop (Left) and For loop (Right) examples.



Proceso	PID	Descripción	Estado	Subprocesos	CPU
LabVIEW.exe	11440	LabVIEW 18.0.1f2 D...	En ejecución	39	8

Proceso	PID	Descripción	Estado	Subprocesos	CPU
LabVIEW.exe	11440	LabVIEW 18.0.1f2 D...	En ejecución	40	0

Figure 4.1.1.8: Example CPU usage by two loops example (one without and the other with a timer), it can be seen that the left loop consumes 8 of CPU and the timer is 0.

Data Structure: It talks about arrays of one and two dimensions. It is noted that arrays can only have elements of the same type and that clusters are used to use structures with several types of elements. It explains the functions that can be performed with them, polymorphism, which is based on the concept of input of different types of data in one element and that are transformed into another, and the autoindexed, which are several qualities that arrays have to get your data in the loops.

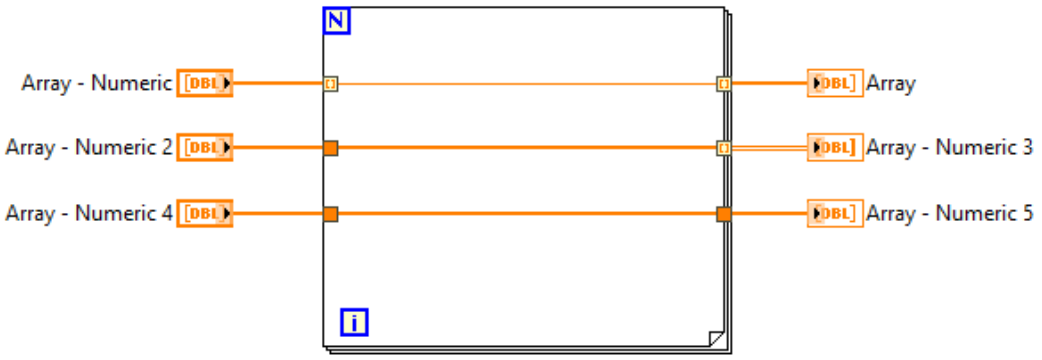


Figure 4.1.1.9: Three ways to treat an array in a loop (the first one, it is autoindexed when enter in a loop, that is, only the parameters of array enter. When exit, all the elements ara grouped again in an array. The second one, the complete array enters and an array of arrays comes out. The third one, the array enters and the array comes out).

Decision making structure: The type of structures they use for this are similar to the "if" and "switch" used in conventional programming.

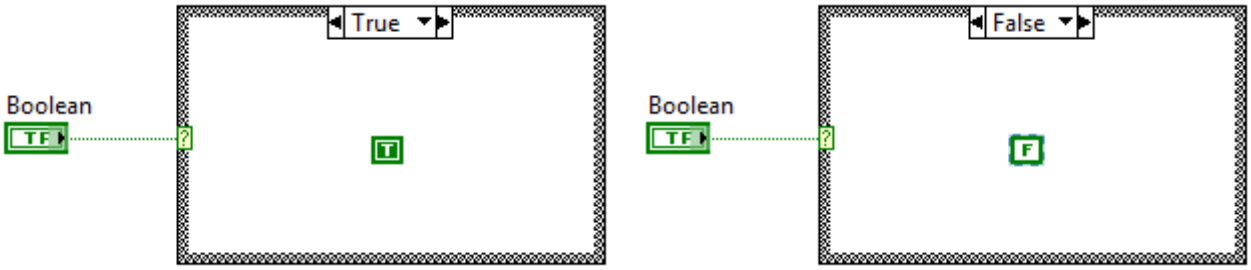


Figure 4.1.1.10: Case structure as a "if"

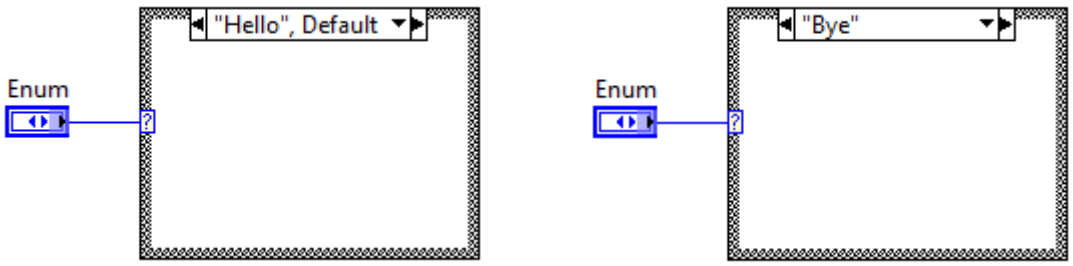


Figure 4.1.1.11: Cases structure as a "switch"

Modularity: This concept is related to the benefit of reusing code by creating VIS that are known to be used more than once, so that they don't have to recreate everything, they are known as subVIs.

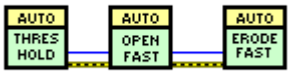


Figure 4.1.1.12: VIs used in other VI

Acces to file: LabVIEW can open, read, write and close all types of files and this section deals with everything related to it.

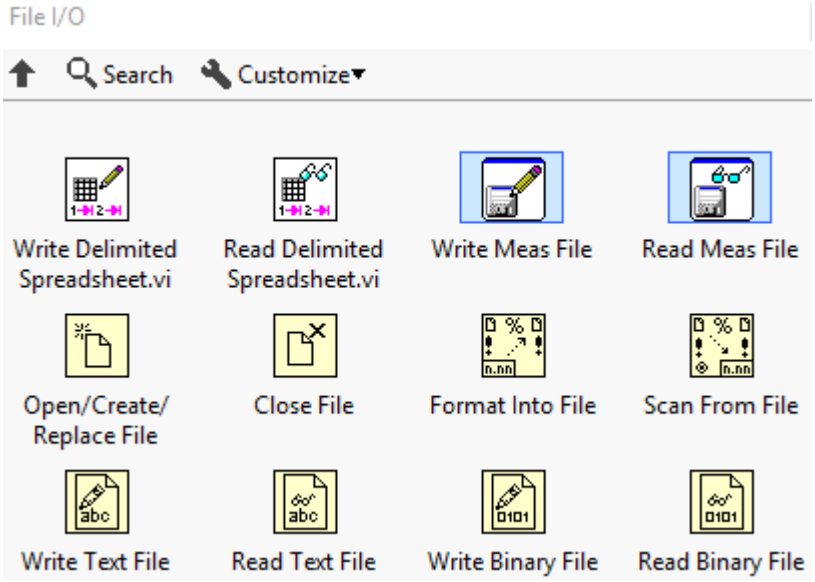


Figure 4.1.1.13: File I/O Palette

And CORE 2 is based on the following topics:

- Use of variables.
- Communication between parallel loops.
- Design patterns.
- Control of the user interface.
- File I / O techniques.
- Improvement of an existing VI.
- Creation and distribution of an installer.

Use of variables: Here it talks about local and global variables, which pass information between locations in the application that can not connect with cables. The differences between the two are explained, at what moment to use each one and the possible career conditions that can be produced if they handle properly.

Figure 4.1.1.14: Up (element to create a local variable) and down (element to create a global variable)

Communication between parallel loops: In this lesson you learn to develop code that synchronizes the data between parallel loops and explains which communication method is the most appropriate in different scenes.

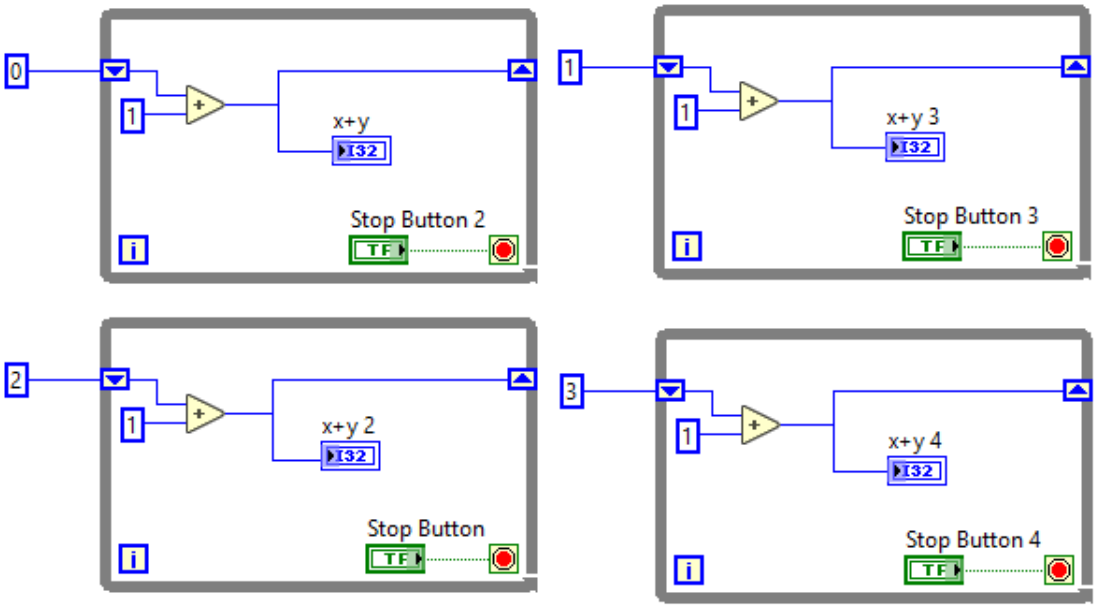


Figure 4.1.1.15: 4 loops in parallel

Design patterns: It shows how to perform different types of patterns (simple, general, state machine and state machine based on events). The simple VI pattern consists of a single VI that performs the operations (such as addition, subtraction, concatenation of strings,...). The general VI pattern consists of three phases: Start-up, all previous procedures are initialized; Main application, performs all VI operations and is usually composed of at least one loop; and Closing, in charge of the finalization. The pattern of the state machine, this consists of a loop with a case structure in its interior. The state machine pattern based on events combines the interaction of the user with a state machine.

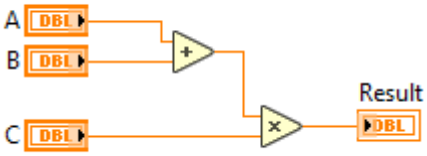


Figure 4.1.1.16: Simple pattern

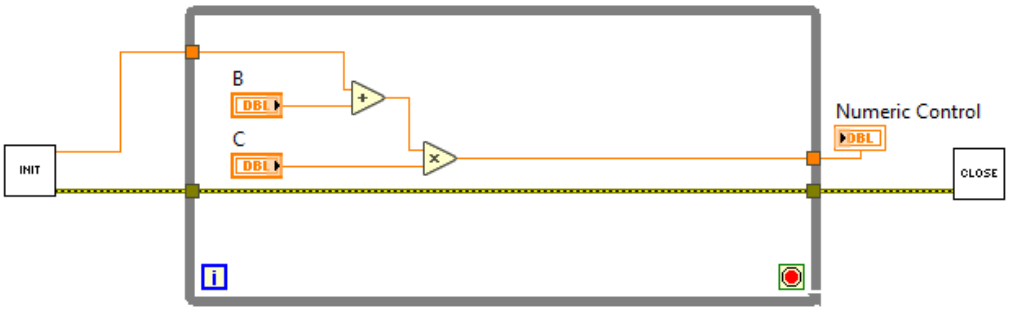


Figure 4.1.1.17: General pattern

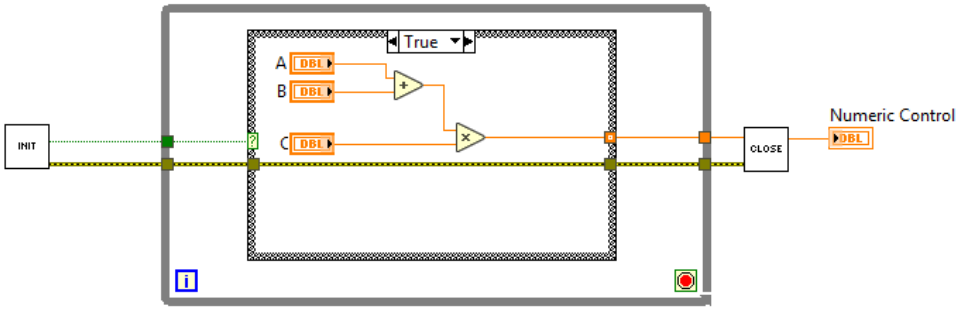


Figure 4.1.1.18: State machine pattern

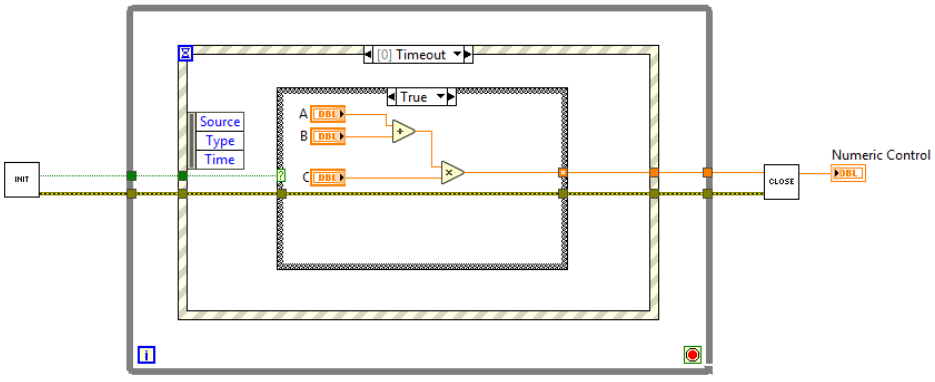


Figure 4.1.1.19: State machine with events pattern

Control of the user interface: In this part of the lesson you learn to use the elements in an appropriate way, to invoke them and control their references to control objects of the Front Panel through programming.

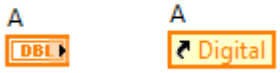


Figure 4.1.1.20: A double variable with its reference

File I/O techniques: The topic of dealing with external files in LabVIEW is explained in more depth.

Improvement of an existing VI: Based on code optimization, factoring and correction of elements to have a sustainable, readable and easy to understand VI for all.

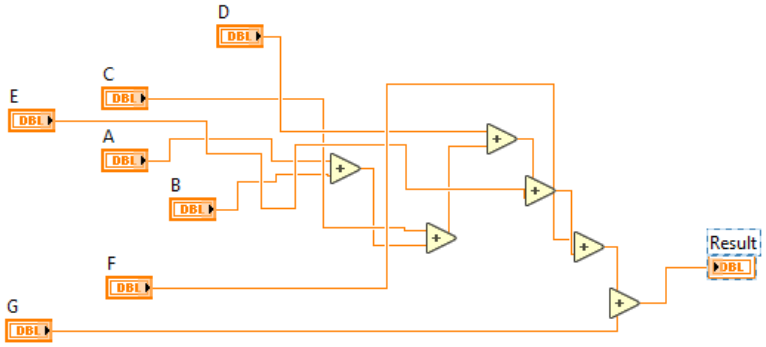


Figure 4.1.1.21: A sequential sum with a bad order.

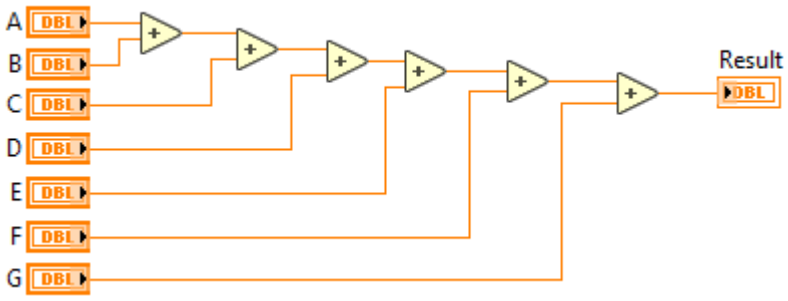


Figure 4.1.1.22: The same sum with a correct order.

Creation and distribution of an installer: It exposes all the steps for the creation of ejectuables of the VI.

After the basic training in LabVIEW, it has been learned a bit about the Block Diagram architecture that AUTIS uses for most projects, which is based in a state machine with events custom pattern, to make it more sustainable and with a lot of modularity. With it all the VIS that we insert in the right place will be controlled by the architecture making it become a much more stable project.

And finally, the LabVIEW training course, the libraries that the company uses for the whole issue of processing and imaging of IMAQ were released. Which has a good set of elements for it, such as memory reserve for images, determination of the size of images, conversions of binary arrays to images, color images to black and white, image processing algorithms (erode, threshold,...), determine histograms, etc.

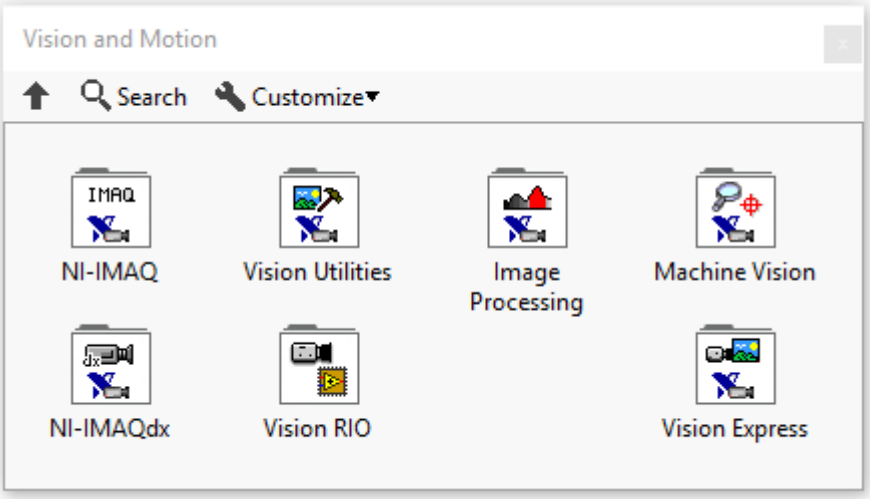


Figure 4.1.1.23: IMAQ palette

4.1.2 Assimilate image processing algorithms

During this period it was necessary to investigate the most common algorithms that the company applies to the images it takes of the surfaces of the bodies of the vehicles. Five were investigated: Threshold, Erode, Dilate, Open and Close. For this, it had to be understood that each of them was used, as they were applied to LabVIEW and how they were performed in CPU.

The mechanism that the company uses to detect defects in the surfaces consists of a tunnel where the bodywork is introduced and with a beam of light reproject a specular reflection on the surface and all this is captured by frame by frame by the cameras. That reflection is the area that will be analyzed later to determine if there are defects. But to be able to treat that area, the previous algorithms and others must be applied.

Explanations of each algorithm:

Threshold: Method that allows to binarize the image according to two input parameters (a minimum and a maximum). Analyze each pixel of the image and if it falls within the range between the minimum and maximum, the value of that pixel is replaced by 1. If it is left out it replaces it with 0. It is useful to discard from the analysis of the image areas where the reflex does not appear or is not valid. This method is introduced in LabVIEW through a VI that calls a DLL (which it will be explained later).

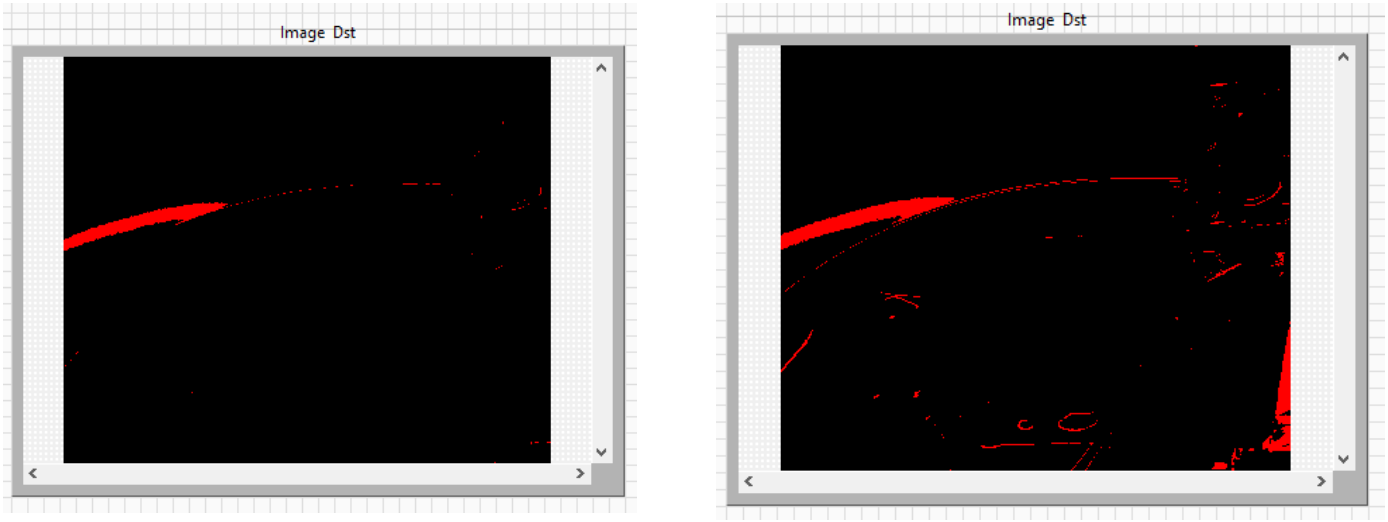


Figure 4.1.2.1: To the left, a threshold with minium 70 and maxium 240. And to the right, minium 10 and maxium 240.

Erode: Method that reduces the limits of a region. This process is done through a variable size / radius kernel. What is done is to apply said kernel on a pixel, determining the minimum of its neighbors and applying on said pixel. It is used to reduce the contours of the reflection taken.

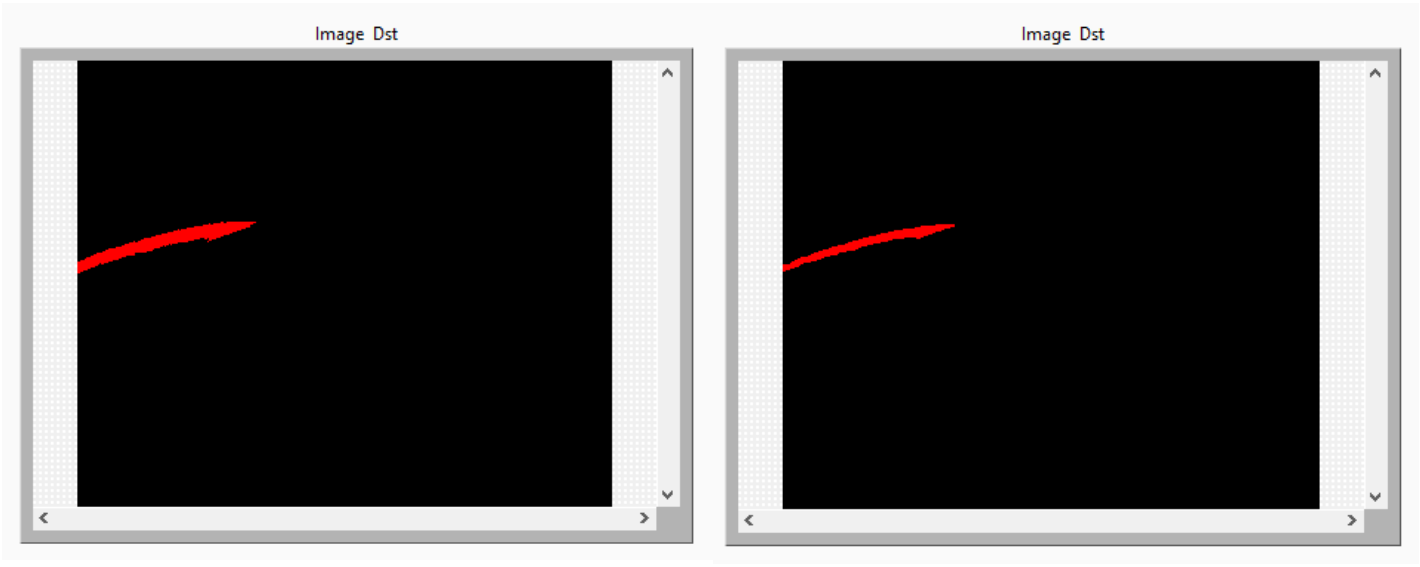


Figure 4.1.2.2: To the left, an erode with radius 1. And to the right, with radius 5.

Dilate: Brother method of the Erode. Perform the opposite operation, calculation of the maximum of its neighbors.

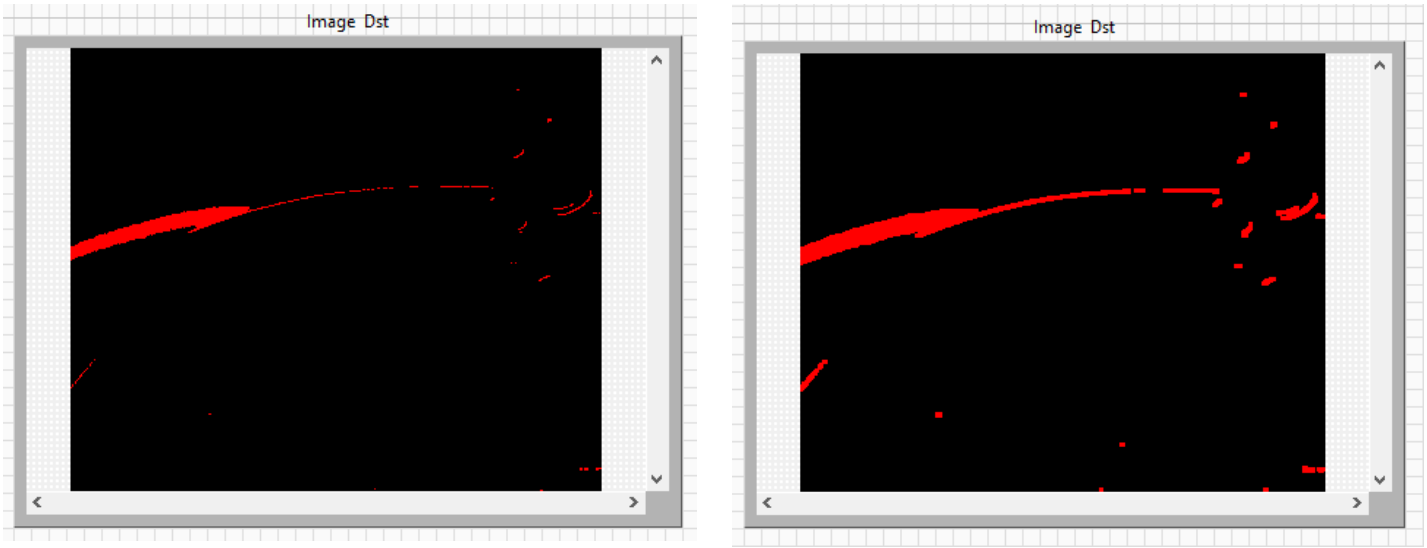


Figure 4.1.2.3: To the left, a dilate with radius 1. And to the right, with radius 5.

Open: Method that first applies an Erode and then a Dilate.

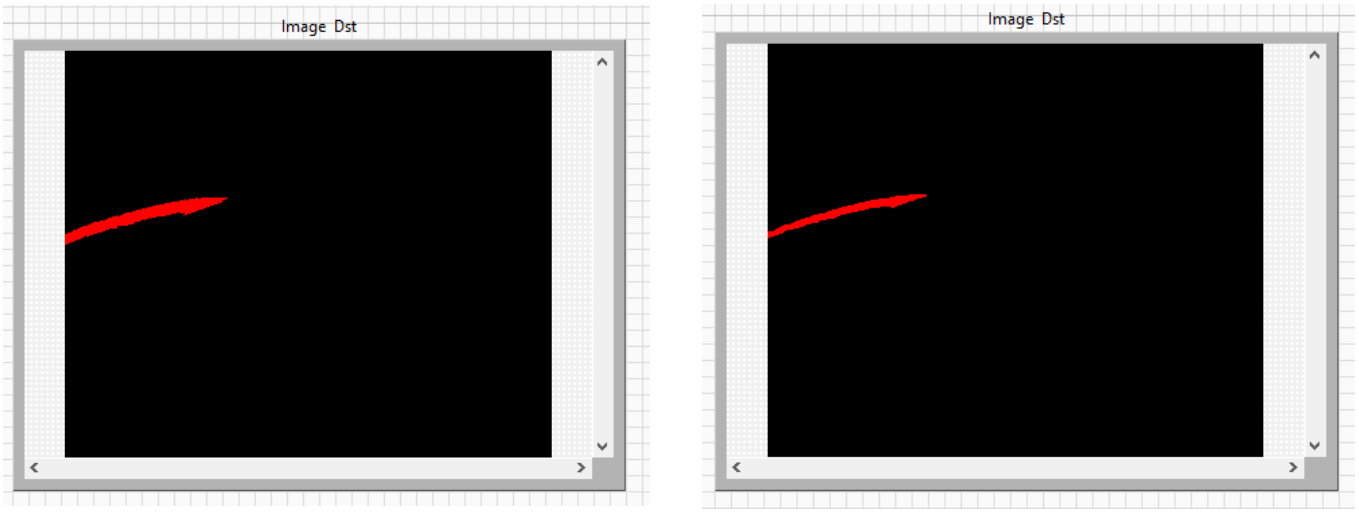


Figure 4.1.2.4: To the left, an open with radius 1. And to the right, with radius 5.

Close: Method that first applies a Dilate and then an Erode.

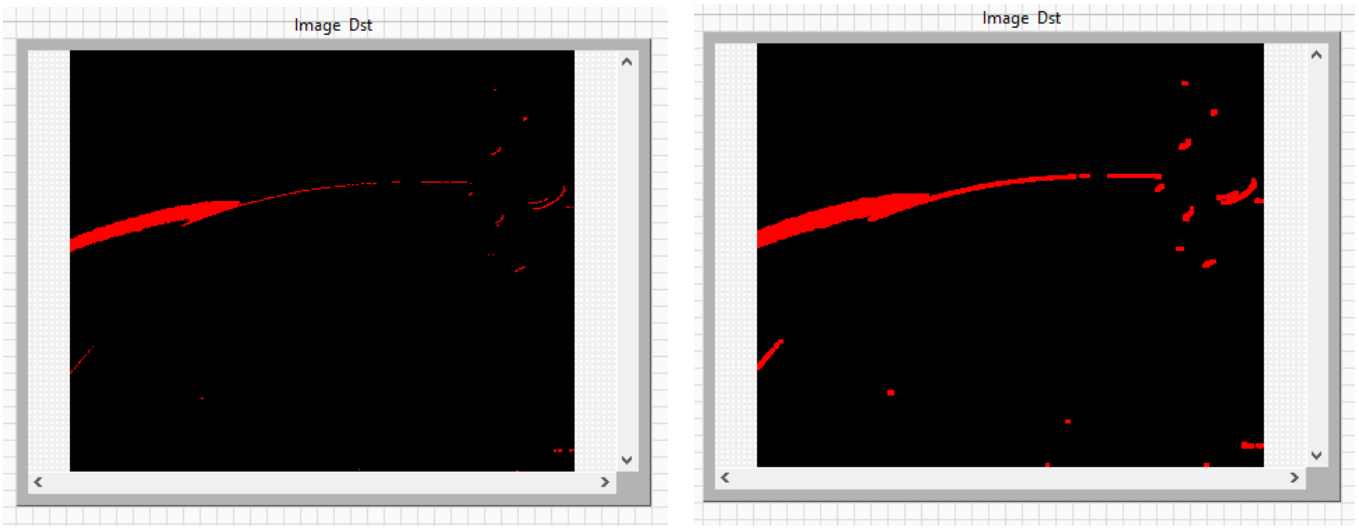


Figure 4.1.2.4: To the left, a close with radius 1. And to the right, with radius 5.

4.1.3 Analyze current developments on CPU

The algorithms described in the previous section can be carried out sequentially or in parallel in the CPU. Both ways don't fit the needs of the company, either by execution times or by continuing to load many tasks to the CPU. Therefore, the company chooses to look for alternatives with the GPU. To be able to perform the largest number of operations with the GPU and thus achieve two things: better processing times and reduce the computational load of the CPU to be able to perform more tasks with it.

4.1.4 Techniques for the development of new features on GPU

As a result of wanting to perform the maximum possible operations in GPU, it is decided to use the functionalities of the CUDA architecture. It is a parallel computing platform that allows to use a variation of the programming language in C to encode algorithms in GPU created by NVDIA.

During this period, a CUDA training is carried out through a course on fundamentals of computational acceleration that the NVDIA itself imparts online, through the tutoring of professionals in this field and through its own research.

In addition, we opted for a 10-hour course introducing us in the world of parallel programming in both CPU and GPU, where programming concepts were seen as OpenMP, OpenACC, CUDA,...

4.1.4.1 CUDA architecture

CUDA is born from the search of a unification of the two internal processors (the one of vertices and fragments) of a GPU. This is done because previously there was a problem of load imbalance between the two processors and the difference between instruction repertoires. With this new architecture we are allowed to create a program where a function or a complete program is called, known as a kernel. This runs parallel to the GPU as a set of "threads" that are organized in a hierarchy that can be grouped into "blocks" (which can also be grouped together in a grid). Blocks and grids can have one, two or three

dimensions. In order to specify the quantity of all these, it must be specified when making the calls.

Already within the GPU each thread is assigned an identifier inside its block and the same for the blocks within the grid. So, this allows each thread to decide what data it should work with. Finally, it should be noted that threads are grouped in groups of 32, and that the maximum number of threads per block is 1024 and that the number of threads can vary but it is around 65553.

4.1.5 Implementation of the functionalities described

The objective with all this learning is to build a set of VIs in LabVIEW that allow to perform image processing operations with CUDA. Put both concepts together using DLLs (Dynamic Link Libraries).

To create a DLL it has been used the Visual Studio 2017 development environment, which allows to incorporate CUDA into its C / C ++ language. When creating a new project, you must specify that the project will not be .exe but .dll and in the header files you must specify a condition for each function, which you want to export. Therefore, the functions are created in C / C ++ language and the CUDA functionalities are incorporated.

Once we have the created DLL file, already in LabVIEW an element is used, "Call Library Fuction Node", that allows to call the desired DLL, passing the necessary input and output parameters.

After carrying out all the previous processes, we will have one or several VIs that contain the desired functionalities and therefore a palette can be created. A palette is nothing more than a grouping of VIs introduced in LabVIEW in a general way, used according to the required needs and without the need to have the source project (the one containing the VIs of said palette) open.

In order for the image processing palette to work correctly, a strict order must be followed in the execution of the VIs placed in the Block Diagram. This order will be: GPU memory reserve> Copy of CPU to GPU data> Processing Functions> Copy of GPU data to CPU> GPU memory release. And all of them must be linked by the same variable, "Error IN / OUT Toolkit", which must be received from the VI previously executed and extracted by the next VI, thus completing the marked order.

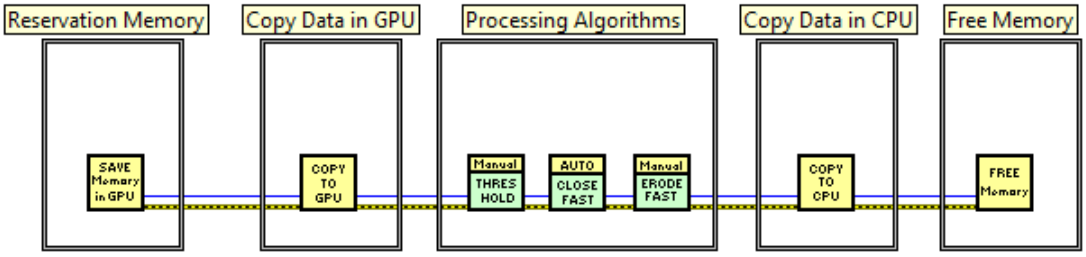


Figure 4.1.5: Order of use of the palette

4.1.5.1 GPU Control Algorithms

CUDA has a large number of default functions to handle the entire GPU environment. After reading about it, it was decided to implement five functions in my palette that avoid interaction errors with LabVIEW, because in the long run there could be errors in the use of the GPU, if the indicated path was not followed when working with it.

The first function implemented indicates how many graphics processors compatible with CUDA the device has. The following restart a certain or all of the GPUs, freeing up the memory space that the processor uses. The third one establishes which graphic card to use. And finally, although for reasons of time it has not been possible to make it work correctly, a function that explains what is the error that could have occurred in the DLL.

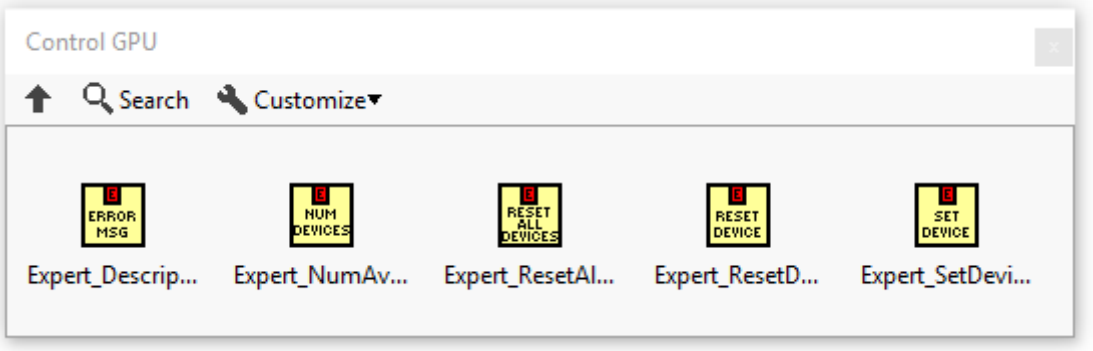


Figure 4.1.5.1: GPU Control VIs

4.1.5.2 Reserve and Free GPU memory Algorithms

One of the concepts to take into account when working with CUDA is that it is essential to reserve GPU memory (similar to the process that is done with the CPU) to perform operations with it. The release is also essential, to avoid the storage of unnecessary data. So, whenever you want to work with it, we must use the functions of reservation and release of that memory.

Due to the purpose of creating a palette for image processing, these two functions are implemented as two VIs. The first, the reserve, you pass the sizes of the image. To the second, only the variable "Error IN / OUT Toolkit" commented in a previous section.

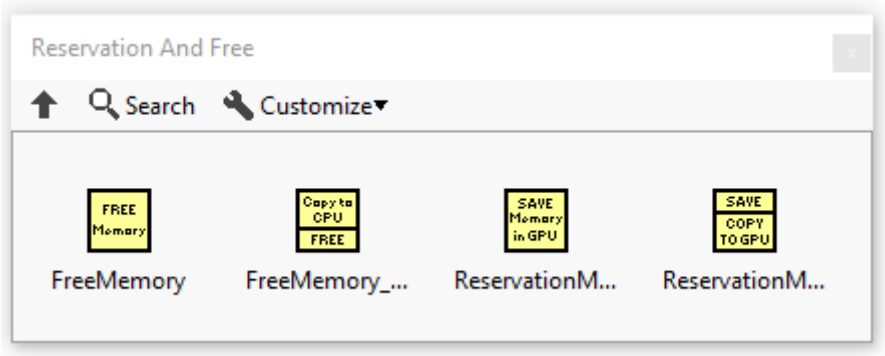


Figure 4.1.5.2: Reserve and free GPU memory VIs

4.1.5.3 Copy CPU-GPU and GPU-CPU data Algorithms

Once you have reserved a memory area, you must send the necessary data to the GPU with which to perform the operations. And to get the data, they must be sent back to the CPU. In order to implement these requirements, CUDA has several very practical functions in this regard.

It has been chosen to make two VIs. The one that makes a copy of the CPU to GPU, is passed the array of the image (which it will be explained in more depth in a later section) that you want to send to the GPU. The other VI receives an empty array and returns it with the result of the processing.

To speed up and avoid errors, it has been put together the functions of Memory Reserve and Copy of CPU-GPU data into one, thus creating a single VI to deal with the initialization of work with GPU. In the same way, it has been did

it with the Memory Release and GPU-CPU Data Copy, to treat the completion as a single VI. This is possible because nothing can be added between these unions.

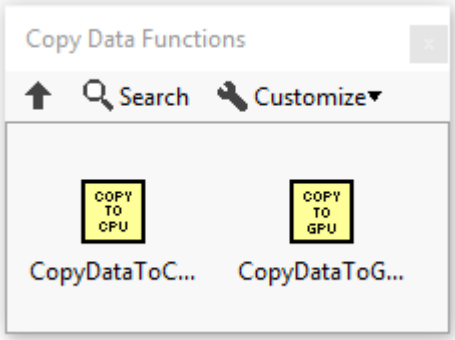


Figure 4.1.5.3: Copy data VIs

4.1.5.4 Threshold and Reverse Threshold Algorithms

In the following algorithms, it is taken into account if the user that will use these processing functions, understands the internal management of the "threads" and "blocks" that the GPU uses. Because VIs are made that control these parameters automatically or manually.

The threshold realized takes advantage of the parallel programming in CUDA, since as it is about the replacement of pixels only, independently of its neighbors, it is enough to apply a number of threads and blocks of 1D, that is to say there will only be in x, needed to change all the pixels.

Reverse threshold is the function contrary to threshold, it substitutes 0 the pixels inside the range and 1 the ones outside.

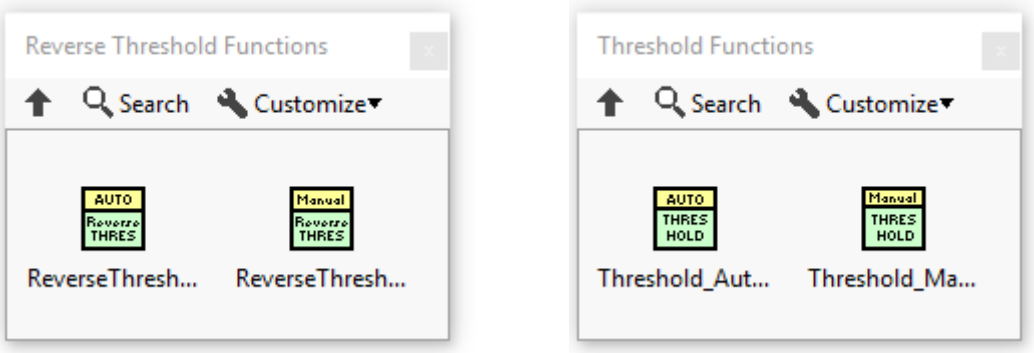


Figure 4.1.5.4: Threshold and Reverse Threshold VIs

4.1.5.5 Erode and Dilate Algorithms

It was also taken into account to perform these algorithms automatically and manually. After analyzing and investigating how an Erode worked, a quite reliable solution was obtained, not quite optimal, but which performed image processing in a very acceptable time. It is based on obtaining the minimum and maximum of the neighbors, but in a parallel way. To be able to do it is based on the use of the two dimensions of the "threads" and "blocks", to speed up much more.

After implementing this function, the Dilate was performed by changing some parameters. By performing tests, a code was obtained that performed these algorithms with a higher speed, but with a disadvantage. Much greater speed is achieved if the number of images is very large. In order to arrive at this solution, what is done is to divide the algorithm made in CUDA of the first Erode / Dilate in two steps. First it is done with the threads of the y axis and in the other step with those of the x. The only downside of this new code is that we use another auxiliary array, with which we must reserve memory and release it. But all this is done internally and this function receives and passes the data as the other VIs of these algorithms.

In this way you have four VIs of each: 2 of the fastest version (when you have a large number of images) and 2 of the normal version.

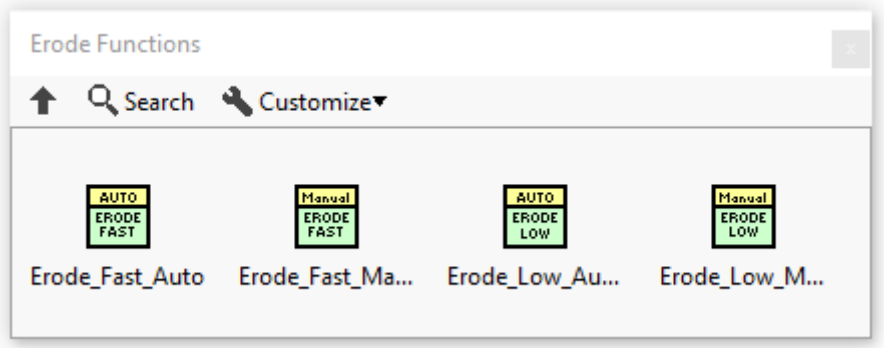


Figure 4.1.5.5.1: Erode VIs

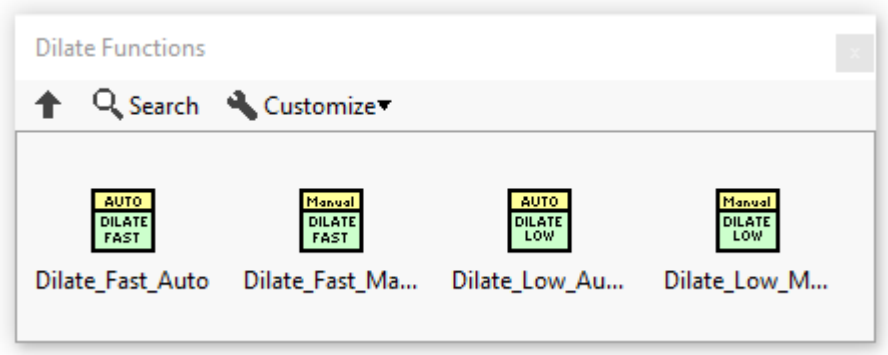


Figure 4.1.5.5.2: Dilate VIs

4.1.5.6 Open and Close Algorithms

These are also made for automatic and manual format. The versions that make Erode and Dilate are taken much faster. The reason for this is that these VIs will process large amounts of images and in the long run would obtain better results.

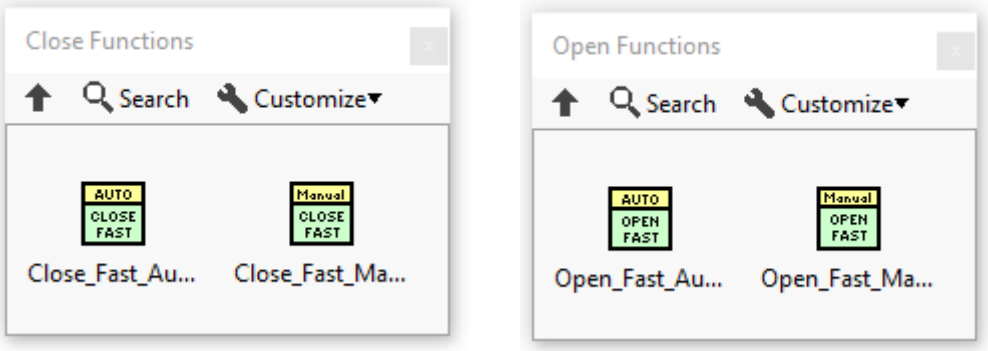


Figure 4.1.5.6: Open and Close VIs

4.1.5.7 Internal Algorithms

Since we have tried to implement this palette for professional and non-professional users in this field, we have always sought the best optimization, the best possible results and the greatest ease of use. By the way, knowing that when working with the GPU there is not a total of always efficient "threads" and "blocks", a function has been created in Visual Studio that calculates all this automatically to avoid the user having to know nothing of all handled by the GPU.

With this function according to the sizes of the image to be processed, quantities of these are established.

Also for those more expert users, we have a manual version of all the algorithms to determine the numbers of "threads" and "blocks" to use GPU. For the moment, in this format, if you use algorithms that require two dimensions of these, the quantities will be the same for x as for y.

In LabVIEW, VIs have been created that facilitate all the previous and final travel to obtain a processed image. For this, the following VIs have been implemented:

- 1_acq_To_ArrayU8: Because the company stores a set of images taken in the ".acq" file, this VI has been created to be able to read one or all of the desired images and transform each image into an array of Unsigned 8, which is the file format that receives the first necessary VI.

- 1_ArrayU8_To_Image: After completing the entire processing path, the image is obtained in an array of Unsigned 8, so we must do a conversion. From this step, this VI is commissioned.

- 1_Image_To_ArrayU8: In the same way that you can convert an ". acq" file, in this one you can do with all these types of images: BMP, TIFF, JPEG, JPEG2000, PNG and AIPD.

- CreateBufferSrcDst: In many cases you want to display both the initial result and the end of the desired image, for this to occur it is necessary to reserve memory. In this VI, making use of IMAQ libraries is responsible for making a memory reservation for two images.

- UseMask: This function at the moment has no use. It is based on receiving a mask of the desired image, transform it to binary and from binary to array Unsigned 8. If no mask is received, create an entire mask full of 1s. The reason for this function is that in the future, it can be used to streamline other algorithms by removing image areas that will not be necessary to analyze.

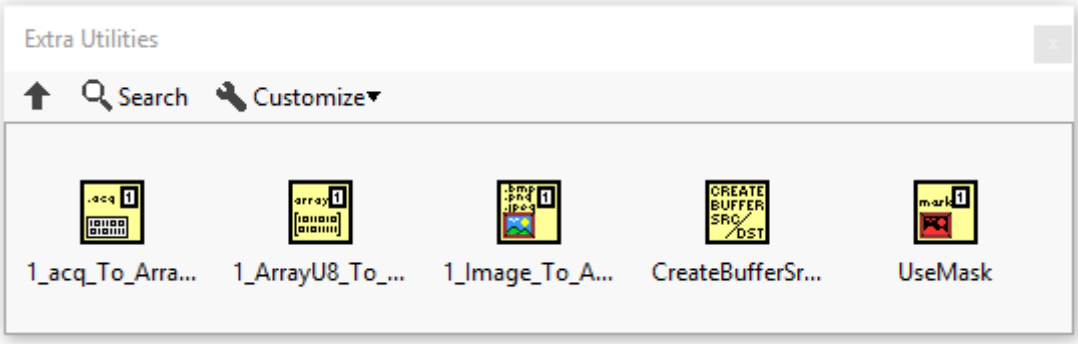


Figure 4.1.5.7: Extra VIs

4.2 Results

After all the achievements have been advanced throughout the project, it can be affirmed that the objectives set have been achieved. In this way, we achieve an operative and optimal palette in LabVIEW that can be used in a professional field such as the inspection of body surfaces. In turn, freeing up workload to the CPU and using the GPU as much as possible.

The times obtained in each algorithm don't surpass the 2 milliseconds, arriving to process images of sizes 2464 x 2056 pixels in less than 4 milliseconds and of 1232 x 1028 pixels in less than 3.

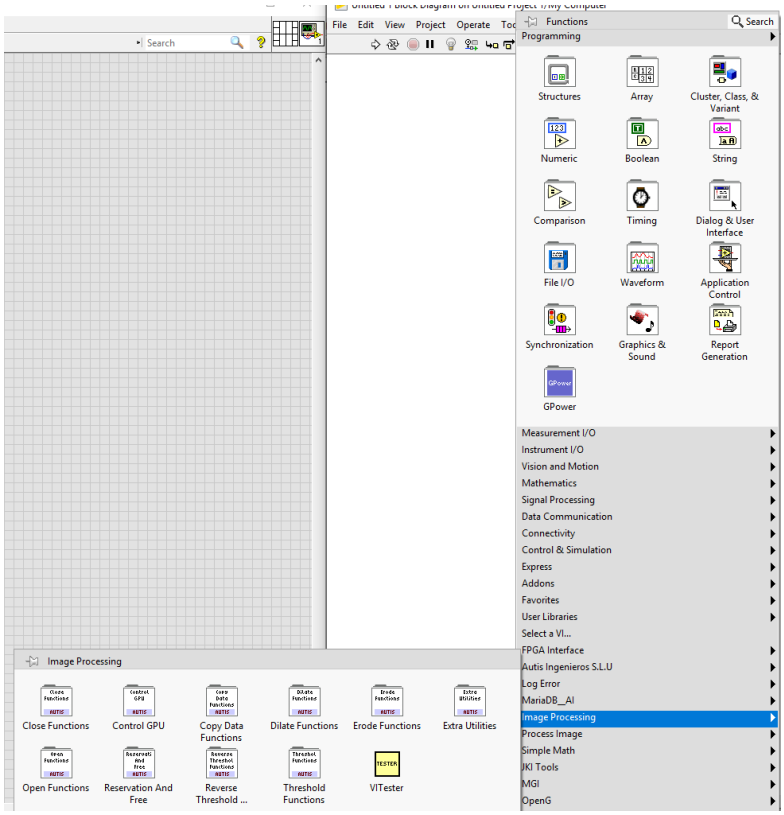


Figure 4.2: Final palette introduced in LabVIEW

5

CONCLUSIONS AND FUTURE WORK

Contents

5.1 Conclusions

After all the work done I am satisfied with the result obtained. Learning new forms of programming and new work environments give much greater skill and possibility of creating codes, although as seen in some cases it is not always possible to parallelize certain algorithms.

Working with the LabVIEW environment has been a great experience, since it is based on a different programming than the one seen during the degree, highly recommended when debugging code and much more visual.

I think the only fault is that I have had is that spent a lot of time fixing errors between LabVIEW and CUDA, because I was trying to perform actions that couldn't be done. One of them was to try to handle from LabVIEW the images that were in GPU memory, which can' t be done. And the other problem, which has been rather a limitation, is that it has been tried that this palette could be used several times, that is, to be able to use for example two Erodes in a parallel way in LabVIEW. It couldn't be achieved due to lack of time and inexperience,

since he was investing a lot of time in a factor that he didn't know for sure if he could. After finishing the whole project and researching it, I have several possibilities to achieve it.

One thing I would like to add is that all the functions of processing the image in a Visual document are performed. The reason for this is that it allows you to make changes more accurately and quickly than creating news documents for each function. In addition only few functions have been realized because the learning of LabVIEW and CUDA is done from zero, so the time to create and test new algorithms was not very high.

Finally, in my opinion, doing the TFG in a company brings great experience to observe how the real world really works. Since until it is entered, it isn't known. I must also add that the company where I have done it isn't related to videogames, but it's related to many of the subjects I have given programming. Externally to the TFG, I have used 3DS Max and Blender, for company activities.

5.2 Future work

I would like to continue working in this area and parallelize more complicated algorithms, so that I can really appreciate all the knowledge acquired.

Also, I would like to improve the functionality of the palette so that it would be possible to use it more than once in the same VI, which, as I said earlier, is still not possible.

A

OTHER CONSIDERATIONS

Contents

A.1 Bibliography

Information about online courses offered by NVIDIA:
https://www.nvidia.com/en-us/deep-learning-ai/education/#ac-online

Basic information about CUDA:
https://es.wikipedia.org/wiki/CUDA
https://sg.com.mx/revista/programando-para-el-gpu-cuda
https://devblogs.nvidia.com/even-easier-introduction-cuda/

Book of the University of Burgos about CUDA:
http://riubu.ubu.es/bitstream/10259/3933/1/Programacion_en_CUDA.pdf

CUDA Toolkit Documentation:
https://docs.nvidia.com/cuda/index.html#

Operation on the NVIDIA GPU:
https://hardzone.es/2018/05/06/nucleos-cuda-tarjetas-graficas-nvidia/

Basic CUDA programming with C ++:
https://eslinux.com/programacion-gpu-cuda/

First steps with CUDA:
http://fisica.cab.cnea.gov.ar/gpgpu/images/clases/clase_1_cuda.pdf

Creating a C ++ DLL for CUDA:
https://www.youtube.com/watch?v=eAWXktWXOo4

Creating a C ++ DLL for Visual Basic:
http://alonso_m.tripod.com/visualc/creardll.htm

Creating a C ++ DLL:
http://codecrab.blogspot.com/2010/05/utilizando-librerias-dll-desde-c.html

Treatment of images in C:
https://poesiabinaria.net/2011/07/salvando-archivos-de-imagen-bmp-en-c/

Information about CUDA bookstores:
https://docs.nvidia.com/cuda/cuda-samples/index.html#box-filter-with-npp

LabVIEW Core 1 and 2:
http://sine.ni.com/tacs/app/overview/p/ap/of/lang/es/ol/es/oc/gt/pg/1/sn/n24:12725/id/1582/

Information about morphological transformations:
https://docs.opencv.org/trunk/d9/d61/tutorial_py_morphological_ops.html
https://docs.opencv.org/2.4/doc/tutorials/imgproc/erosion_dilatation/erosion_dilatation.html
https://en.wikipedia.org/wiki/Erosion_(morphology)
https://homepages.inf.ed.ac.uk/rbf/HIPR2/open.htm
https://courses.cs.washington.edu/courses/cse576/06sp/notes/Basics1.pdf

UPV documents about "Introduction to parallel processing"

B

SOURCE CODE

Repository in GitHub of Visual Studio Project:
https://github.com/david32crack/DLLCUDA

Repository in GitHub of LabVIEW Project:
https://github.com/david32crack/PaletteLabVIEWProject

Repository in GitHub of Palette:
https://github.com/david32crack/PaletteLabVIEWPackage

Algotithms in Visual Studio:

Threshold Algorithm with GPU:

```
__global__ void threshold(byte* src, byte* dst, byte min, byte max, int stride, int size)
{
    for (int pos = blockIdx.x * blockDim.x + threadIdx.x; pos < size; pos += stride)
        dst[pos] = (src[pos] >= min && src[pos] <= max) ? 1 : 0;
}
```

First version Erode Algorithm with GPU:

```
__global__ void erode(byte* src, byte* dst, int w, int h, int radio)
{
    int posx = threadIdx.x + blockIdx.x * blockDim.x;
    int posy = threadIdx.y + blockIdx.y * blockDim.y;

    if (posx >= w || posy >= h)
        return;

    unsigned int start_y = max(posy - radio, 0);
    unsigned int end_y = min(h - 1, posy + radio);
    unsigned int start_x = max(posx - radio, 0);
    unsigned int end_x = min(w - 1, posx + radio);

    int _min = 255;

    for (int y = start_y; y <= end_y; y++)
        for (int x = start_x; x <= end_x; x++)
            _min = min(_min, src[y * w + x]);

    dst[posy * w + posx] = _min;
}
```

Second version Erode Algorithm with GPU:

```
__global__ void erode_separable_step2(byte* src, byte* dst, int w, int h, int radio) {

    int posx = threadIdx.x + blockIdx.x * blockDim.x;
    int posy = threadIdx.y + blockIdx.y * blockDim.y;

    if (posx >= w || posy >= h)
        return;

    unsigned int start_y = max(posy - radio, 0);
    unsigned int end_y = min(h - 1, posy + radio);

    int _min = 255;
    for (int y = start_y; y <= end_y; y++) {
        _min = min(_min, src[y * w + posx]);
    }
    dst[posy * w + posx] = _min;
}

__global__ void erode_separable_step1(byte* src, byte* dst, int w, int h, int radio) {

    int posx = threadIdx.x + blockIdx.x * blockDim.x;
    int posy = threadIdx.y + blockIdx.y * blockDim.y;

    if (posx >= w || posy >= h)
        return;

    unsigned int start_x = max(posx - radio, 0);
    unsigned int end_x = min(w - 1, posx + radio);

    int _min = 255;
    for (int x = start_x; x <= end_x; x++) {
        _min = min(_min, src[posy * w + x]);
    }
    dst[posy * w + posx] = _min;

}
```

First version Dilate Algorithm with GPU:

```
__global__ void dilate(byte * src, byte *dst, int w, int h, int radio)
{
    int posx = threadIdx.x + blockIdx.x * blockDim.x;
    int posy = threadIdx.y + blockIdx.y * blockDim.y;

    if (posx >= w || posy >= h)
        return;

    unsigned int start_y = max(posy - radio, 0);
    unsigned int end_y = min(h - 1, posy + radio);
    unsigned int start_x = max(posx - radio, 0);
    unsigned int end_x = min(w - 1, posx + radio);

    int _max = 0;

    for (int y = start_y; y <= end_y; y++)
        for (int x = start_x; x <= end_x; x++)
            _max = max(_max, src[y * w + x]);

    dst[posy * w + posx] = _max;
}
```

Second version Dilate Algorithm with GPU:

```
__global__ void dilate_separable_step2(byte* src, byte* dst, int w, int h, int radio) {

    int posx = threadIdx.x + blockIdx.x * blockDim.x;
    int posy = threadIdx.y + blockIdx.y * blockDim.y;

    if (posx >= w || posy >= h)
        return;

    unsigned int start_y = max(posy - radio, 0);
    unsigned int end_y = min(h - 1, posy + radio);

    int _max = 0;
    for (int y = start_y; y <= end_y; y++) {
        _max = max(_max, src[y * w + posx]);
    }
    dst[posy * w + posx] = _max;
}

__global__ void dilate_separable_step1(byte* src, byte* dst, int w, int h, int radio) {

    int posx = threadIdx.x + blockIdx.x * blockDim.x;
    int posy = threadIdx.y + blockIdx.y * blockDim.y;

    if (posx >= w || posy >= h)
        return;

    unsigned int start_x = max(posx - radio, 0);
    unsigned int end_x = min(w - 1, posx + radio);

    int _max = 0;
    for (int x = start_x; x <= end_x; x++) {
        _max = max(_max, src[posy * w + x]);
    }
    dst[posy * w + posx] = _max;

}
```

Algorithm to calculate Threads and Blocks:

```
void setNumberThreads2D(int threadsX, int threadsY, int blocksX, int blocksY, bool automatic) {

    if (automatic) {

        if (width > 3000) {
            threadsInX = 16;
            blocksInX = 500;
        }
        else if (width > 1600) {
            threadsInX = 8;
            blocksInX = 400;
        }
        else {
            threadsInX = 8;
            blocksInX = 240;
        }

        if (height > 3000){
            threadsInY = 16;
            blocksInY = 500;
        }

        else if (width > 1600) {
            threadsInY = 8;
            blocksInY = 400;
        }

        else {
            threadsInY = 8;
            blocksInY = 240;
        }

    }
```

```
else {

    if (threadsX == 0 || threadsX > 512)
        threadsX = 512;
    if (blocksX == 0 || blocksX > 65535)
        blocksX = 65535;

    threadsInX = threadsInY = threadsX;
    blocksInX = blocksInY = blocksX;

}
```