UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

TRABAJO DE FIN DE MÁSTER

FPGA prototype for wavefront reconstruction acceleration

Máster Universitario en Investigación e Innovación en TIC

Raúl Martín Lesma

Septiembre, 2016

FPGA prototype for wavefront reconstruction acceleration

AUTOR: Raúl Martín Lesma

TUTOR: Gustavo Sutter

High Performance Computing and Networking group

Dpto. de Ingeniería Telecomunicación

Escuela Politécnica Superior

Universidad Autónoma de Madrid

Septiembre 2016

Resumen

La óptica adaptativa es una tecnología usada para mejorar el rendimiento de diferentes tipos de sistemas ópticos. Lo hace corrigiendo las posibles aberraciones que son introducidas por la atmósfera. Para corregirlo se usa un sensor de frente de onda, habitualmente un Shack-Hartmann. En esta técnica, hasta centenas de centroides de imágenes deben ser calculados. De este modo el frente de onda original puede ser reconstruido.

La óptica adaptativa impone restricciones temporales muy acusadas, el proceso completo debe ser realizado en un tiempo del orden de un milisegundo. Debido a esta limitación los algoritmos usados para calcular cada centroide son rápidos, pero normalmente su precisión es baja.

En la misión Gaia de la ESA se ha desarrollado un algoritmo de cálculo de centroides de máxima verosimilitud, teniendo el mismo una precisión muy cercana al máximo teórico, la cota inferior de Crámer-Rao. Como este algoritmo es computacionalmente complejo, es normalmente demasiado lento para se usado en óptica adaptativa. Una versión simplificada que usa *look-up tables* fue desarrollada para estudiar si esta podría cumplir los requisitos temporales.

En un trabajo previo, una primera versión de un sistema basado en FPGA que implementa este algoritmo fue creada. Consiste en un sistema empotrado que usa un procesador *soft* Microblaze para controlar un sistema con un coprocesador. Este coprocesador fue creado usando herramientas de síntesis de alto nivel, lo que se probó adecuado para implementar algoritmos intensivos en el cálculo con datos.

Este prototipo tenía una funcionalidad reducida, y estaba seriamente limitado. El tamaño de la tabla usada era demasiado pequeño, y sólo realizaba una iteración del algoritmo de cálculo de centroides. En este proyecto se presenta una versión más completa de este prototipo, así como un estudio de la precisión alcanzada por tabla de diferentes tamaños y un estudio de la convergencia del algoritmo.

Además, se compara la precisión de algoritmo ya implementado con el mismo en una plataforma software. La aceleración del algoritmo ha sido medida y un estudio multinúcleo ha sido realizado.

Abstract

Adaptive optics is a technology used to improve the performance of different kinds of optical systems. It does so correcting the possible aberrations that are introduced by the atmosphere. To correct it, a wave front sensor is used, often a Shack-Hartmann. In this technique, up to hundreds of image centroids have to be determined. In this way the original wave front can be reconstructed.

Adaptive optics imposes a very restrictive time constraint, the whole process must be completed in a time of the order of one millisecond. Due to this time limitation the algorithms used to calculate each centroid are fast, but usually achieve low precision.

A maximum likelihood algorithm to calculate centroids was developed for ESA Gaia mission, providing a precision very close to the theoretical maximum, the Crámer-Rao lower bound. As this algorithm is computationally complex, it is usually too slow for adaptive optics. A simplified version using look-up tables was developed to study if it could comply with the time requirements.

In a previous work, a first version of a FPGA-based system that implements this algorithm has been created. It consists of an embedded system that uses a Microblaze soft processor to control a system with a coprocessor. This coprocessor was created using high level synthesis tools, which proved to be adequate to implement data intensive algorithms.

This prototype covered a basic functionality, and had several limitations. The size of the used look-up table was too small, and it only performed one iteration of the centroid algorithm. In this project a more complete version of this prototype is provided, as well as a study of the precision achieved by different look-up table sizes and a study of the convergence of the algorithm.

Also the precision of the implemented algorithm is compared with the one achieved by the same algorithm in a software platform. The acceleration of the algorithm has also been measured, and a multicore study has been done.

Table of contents

Table of figures

Glossary

AO	Adaptive Optics
ASIC	Application-specific Integrated Circuit
BRAM	Block RAM
CCD	Charged-Coupled Device
CPU	Central Processing Unit
DDR	Double Data Rate
DSP	Digital Signal Processor
ESA	European Space Agency
ESAC	European Space Astronomy Centre
FMC	FPGA Mezzanine Card
FPGA	Field Programmable Gate Array
GPU	Graphics Processor Unit
HDL	Hardware Description Language
HLS	High Level Synthesis
HLST	High Level Synthesis Tools
JTAG	Joint Test Action Group
LUT	Look-up Table
PSF	Point Spread Function
RAM	Random Access Memory
RMS	Root Mean Square

RTL	Register Transfer Level (Language)
VHDL	VHSIC Hardware Description Language
VHLS	Vivado High Level Synthesis
WF	Wavefront
WFS	Wavefront sensor
XPS	Xilinx Platform Studio

1. Introduction

This project is the continuation of the project “Aceleración de algoritmos en óptica adaptaviva usando FPGA” [1], and therefore building over the results of the previous project. As [1], the project has been carried out in relation with European Space Astronomy Centre (ESAC). This project aims to study the application of adaptive optics algorithms using a reconfigurable hardware platform (FPGA), testing and improving the system already developed.

1.1. Motivation and objectives

Adaptive optics goal is to modify an arriving wavefront in order to recover its original shape, then boosting the quality of the image recovered in a optic system. This technique is used in telescopes and other devices. The analysis and calculus centroids of images is an extremely important part of the process, being its precision key to the results of the whole process.

As adaptive optics works by definition in real time, the time requirements for each part of the whole flow are very strict, typically being in the order of 1 ms [2]. Because of the described time constraint fast algorithms are very commonly used, as the centre of gravity or correlation ones, not achieving the best precision possible.

In fact, the best precision achievable comes limited by the Crámer-Rao lower bound [3] [4]. The maximum likelihood algorithms created during the development of the ESA Gaia mission [5] achieve a precision very close to this limit. Based on forward modelling they are computationally heavy, and then not suitable for adaptive optics under typical requirements.

FPGA platform have proved to be an interesting way to speed many algorithms, also image processing ones [6]. A FPGA system has already been created in [1], resulting in a first step to study the possibilities of the implementation of the maximum likelihood algorithms. The time to create the system was limited and therefore it lacked completion.

The objectives of this project are the following:

- Upgrade the size of the used look-up table (LUT): Since in the first prototype a very small LUT was used in order to easily test the functionality, in this second version a larger one should be used, for the algorithm to operate in conditions closer to reality.
- Test bench creation: The algorithm needs to be tested with realistic images, in order to prove its convergence and also produce comparable results.
- Determine the number of iterations needed in order to achieve necessary precision.
- Further optimization of the accelerator, if possible.
- Time and acceleration analysis.
- Evaluate further improvement of the system, exploiting FPGA parallelism.

2. State of the art of Adaptive Optics

2.1. Introduction to adaptive optics

Adaptive optics (AO) is the technology that is used to correct distortions in a wavefront (WF) in real-time. It does so measuring the phase alteration in the WF and restoring it, usually with a deformable mirror [7].

Adaptive optics in astronomy is used intensively to obtain more accurate images. There is an example in Figure 1, where the difference in the quality is very noticeable, especially in sharpness.

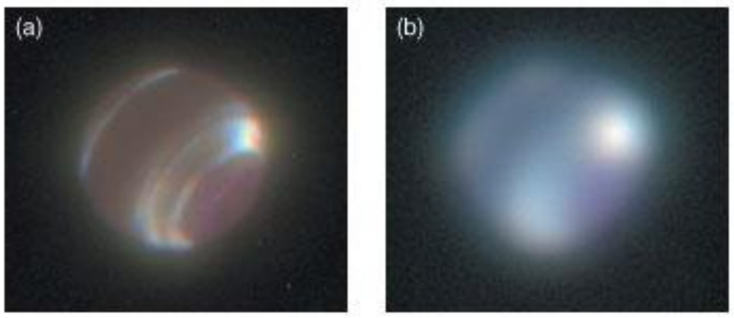


Figure 1: Left, Neptune picture taken with AO; right, same picture without AO.

In this section a very brief resume of the state of the art in adaptive optics is presented. For more information about it go to these sources [1] and [2].

2.2. The adaptive optics challenge

The main challenge of AO is the time scale in which it has to operate to be effective. The entire process has to be finished in around 1 ms. This is the typical coherence time of the atmosphere [8]. In this time hundreds of centroids of point have to be calculated. These calculus allows a Shack-Hartmann wavefront sensor (WFS) to determine how the wavefront (WF) has been deformed, and therefore to retrieve the necessary parameters to correct it [9]. This time constraint greatly limits the complexity of the centroid algorithms, and therefore their precision, which is critical for the result of the WF correction.

2.3. Centroids and their precision

There are two steps when it comes to calculate the WF distortion in a Shack-Hartmann WFS: First calculate the centroid of each PSF (diffracted light points) and then the reconstruction itself of the WF, using the calculated slopes between centroids in comparison with a known position. This work is centred on the calculation of the centroids with the maximum precision available, because it is critical for the posterior reconstruction.

There are several well-known algorithms to calculate the centroid of a point spread function (PSF), like the simple centroid, centre of gravity and correlation algorithms [10]. These algorithms are quite basic ones, which is explained by the time constrain that is characteristic of the AO. However, a much more complex algorithm has been developed during ESA Gaia mission. Its precision approaches the theoretical limit for the precision of the algorithm given a certain amount of information: The Crámer-Rao lower bound [3] [4].

Reaching this precision is a difficult task, but it can be done fitting the coordinates of the centroid, and some other nuisance variables, with a mathematical model of the WFS. In this way the initial centroiding problem has been transformed to a weighted least square minimization problem. There are several ways to solve this kind of problem, such as Gauss-Newton [11] or Levenberg-Marquardt ones [12].

2.4. Maximum likelihood algorithm for centroids

In the section 2.6 “Adaptive optics algorithm” of [1] the bases for the algorithm implemented in the project are described. Here it is a reproduction of part of it, as it is essential to understand this project:

The algorithm used in this project is based in a maximum likelihood algorithm developed for the ESA Gaia mission, and described in deep in [13]. It provides a precision very close to the Crámer-Rao limit. It is based in a forward-modelling algorithm: the weighted Gauss-Newton optimization of a function similar to the observed data [11].

A noiseless image of n_x by n_y pixels (10 by 10 in this case) can be described as a matrix, in which each number will be the electrons collected by each pixel. This matrix can be described also as the total number of electrons produced by the sensor multiplied by the PSF (described in 2.4 subsection):

$$N_i = N_T \cdot PSF\ (x_i - x_c, y_i - y_c, s)$$

Note that *(x_c, y_c)* are the PSF centroid, that is unknown, and s the shape factor of the PSF. The PSF forward modelling function includes a priori knowledge of the optical system, so N_i can be compared to the real number of electrons collected O_i (the real image). There is then a set of parameters that characterize N_i:

$$\boldsymbol{x} = \{N_T, x_c, y_c, s\}$$

These are the parameters that have to be fitted minimizing the weighted RMS sum.

$$\Delta\boldsymbol{x} = M(O - N)$$

In this equation M is a 4×100 values matrix (100 because of the 10 by 10 size of the image matrix), O and N are respectively the observational and model vectors composed by all the rows of its corresponding matrix (image). The steps to calculate these matrices include several non-trivial integrations. A strategy has been developed to bypass this situation, pre-computing in lookup tables (LUT) these matrices. A general description of the algorithm would be:

1. Apply centre of mass algorithm to obtain an initial guess for the final centroid.
2. Construct observation vector (O) by linking together the rows of the 10×10 image received.
3. Retrieve from LUT the M and N matrices that match the values of the initial guess.
4. Compute the operation: $\Delta\boldsymbol{x} = M(O - N)$
5. Update initial parameters: $\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \Delta\boldsymbol{x}$
6. Return to 3. and repeat until the difference between one iteration and the next one is below a certain threshold.

3. The use of reconfigurable HW for adaptive optics

3.1. Benefits of reconfigurable hardware and FPGAs

For a long time, reconfigurable hardware (FPGA) is a growing alternative to the classic ASIC (application-specific integrated circuit) approach to custom application hardware chips.

These are some of the most important benefit of FPGA technology in comparison with other options:

- **Performance**: FPGA are more flexible than digital signal processors, allowing the user to implement hardware parallelism. They designer is in control of everything, including low level operations, which has an enormous potential for fine-grain optimization.
- **Flexibility**: FPGA can be used with a full custom approach, while they are also capable of reproduce complex systems as full System on a Chip (SOC), with an integrated processor, RAM memory, etc. In the last years the main FPGA manufacturers are releasing combinations of hard processors along with FPGA area in the same chip, making SoC systems even more powerful.
- **Time to prototype**: FPGA offers quick prototype capabilities in comparison with other technologies, allowing the designer to test a concept directly on hardware and then even implement incremental changes.
- **Cost**: FPGA are very interesting in terms of cost while the project does not have the scale economy to make ASICs profitable.
- **Reliability and other advantages**: FPGA provide a true hardware implementation, allowing the user to make a deterministic latency system, for example. Its lack of operative system and few abstraction layers allow designers to perform time-critical tasks without the risk of another one interrupting due to true parallelism.

3.2. Commercial FPGA families

There are two main manufacturers of high performance FPGA: Xilinx and Altera. These are their most powerful series of products:

- Xilinx Virtex 7: With nearly 2 million of logic cells this family of FPGAs is one of the most powerful FPGA in the market. Built in 28 nm they are capable of lower power consumption than older generations, even with a greater performance. This new generation comes with 85 Mb in BRAM, which is the largest capacity among common families of FPGA. It also has up to 3600 DSP, which are the main blocks used for float operations, for example. It supports DDR3 external RAM memory at up to 1,866 Mbps [14].
- Xilinx Virtex Ultrascale: A new product family with the most powerful devices from Xilinx. It is built in 16 or 20 nm, with FinFET technology. There are devices with up to 5 million logic cells [15]. It supports DDR4 external RAM at a maximum throughput of 2666 Mbps.
- Xilinx Zynq an Zynq Ultrascale: Zynq is the SoC family form Xilinx. It always includes a hard ARM processor (ranging from a dual core Cortex A9 to a quad-core Cortex-A53) along with a variable FPGA. In these devices the reconfigurable logic is not as powerful as in the other families, being the maximum logic cells around one million [16] [17].
- Altera Stratix 10: Stratix is the most powerful product family in Altera. Built in 14 nm they have a hard processor system built in the same FPGA chip, making them a SoC by themselves. They have a similar number of logical elements in comparison with the Virtex Ultrascale family, around 5 million on the largest devices [18].

3.3. High Level Synthesis tools benefits

In this project the development of the system that implements the AO algorithm is based on a SoC architecture, being the auxiliary hardware built using a High Level Synthesis tool: Vivado-HLS [19]. These tools have been quite recently adopted by the most important FPGA manufacturers. Altera uses Altera SDK for OpenCL as high level synthesis approach, while Xilinx integrates it in Vivado-HLS.

In [1] there is a comparison between the traditional FPGA design flow, including HDL coding and simulation, synthesis, implementation, timing closure, etc. and the design flow when the main development is done with High level synthesis (HLS) tools. These tools allow generally a faster development of the modules at some cost in flexibility and freedom. Here the main differences are showed:

- The traditional FPGA design flow is slow, while an HLS flow should accelerate the whole process, at least for certain types of modules. This allows the user to prototype fast, and then progressively refine the design.
- The traditional flow makes it difficult to make posterior changes to the architecture, while it is a shorter process with HLS tools. For example, changing types in a high level language is quite easy, while in VHDL it can be very challenging, especially for certain types which traditionally have not been used in FPGAs, like floating point representation.
- The HLS flow lets the compiler decide by default how the described algorithm is going to be implemented. While it is true that most of the options can be modified to implement the design the way the user wants, the user has to have a deep knowledge of the tool defaults and it options. In the traditional flow the user has to specify most of the parameters, leaving less for the compiler to guess.
- HLS tools are particularly interesting to implement data intensive algorithms, as they produce better results than in control modules

4. Previous work

4.1. Introduction

As stated in the introduction, this project is a continuation of the work done in [1]. Therefore, its main achievements are explained in this section, putting special focus on the system developed.

One of the main tasks of the PFC was to create coprocessor which efficiently would implement the maximum likelihood algorithm, and more specifically one iteration of it, with the aim of integrating it in an embedded system. In this way the coprocessor or accelerator would process the most computationally heavy part of the algorithm, while mainly the initialization, control and storing tasks will be assumed by the general purpose processor.

All the system is based in a ML605 boards, which includes a Virtex 6 FPGA and several devices as DDR3 RAM, Compact Flash memory reader or Ethernet port [20].

4.2. Coprocessor

Created with a high-level synthesis tool (Vivado-HLS), this accelerator performs one iteration of the maximum likelihood algorithm. It mainly consists of a large matrix multiplication and a few other operations done over floating point represented data. The main operation is this one:

$$M * (O - N)$$

These data, the matrices, are provided from the exterior at the beginning of each iteration. The size of the matrices is 4x100 for M, and 100*1 for O and N, thus giving as result a vector of dimension 4x1. They matrices are retrieved from one node of the LUTs stored in the system, defined by four parameters:

$$\{x, y, N, s\}$$

From left to right these variables represent the position of the centroid in the horizontal axis, in the vertical axis, the number of electrons that hit the sensor, and finally the shape factor.

4.2.1. Directives

The multiplication has been optimized with several directives, which are the tools Vivado-HLS compiler use to determine how to transform the given algorithm into hardware.

Two main directives are applied to the loop that takes care of the multiplication, in order to parallelize and pipeline it:

- Loop unroll directive: By default, only one set of hardware is going to be created, and the operation will mainly be sequential. Applying this directive makes the compiler create many sets of hardware that will process the data in parallel.
- Pipeline directive: It pipelines the different stages in a loop, using as efficiently as possible all the resources created.

The effectiveness of both directives depends greatly on the dependencies between iterations, which leads us to the next point.

4.2.2. Code optimization: thinking in hardware

In order to reduce the dependencies, the code was reorganized. The main loop is showed in the Figure 2: Algorithm applied to a multiplication of matrix of sizes 4×100 and 100×1Figure 2:

```
// Iterate over the cols of the A matrix or the rows of the B matrix
Prod: for(int k = 0; k < MAT_B_ROWS; k++) {
  // Iterate over the rows of the A matrix
      Row: for(int i = 0; i < MAT_A_ROWS; i++) {

                  //If first read of c => save c, which is cache of c
                  if (i==0) c_copy[k] = c[k];
                  //If first read of b => save b, which is cache of b
                  if (i==0) b_copy[k] = b[k];
                  if (i==0) diff[k] = c[k] - b[k];

                  //Read a from a internal variable, and not from the FIFO
                  a_i_k = a[i][k];
                  //Actual multiplication
                  temp[i] = a_i_k * diff[k];

                  if (k == 0) acc[i] = temp[i];
                  //Accumulate on acc
                  else acc[i] += temp[i];

                  if (k == (MAT_B_ROWS-1)) res[i] = acc[i];
    }
  }
```

Figure 2: Algorithm applied to a multiplication of matrix of sizes 4×100 and 100×1

In this way the iterations go over the columns of the matrix A, storing the intermediate values in a temporary variable. The reorganization results in a better utilization of the resources, as the dependencies between rows intermediate results are eliminated, creating 4 data chains.

In addition to this, initialization variables have been created, ensuring that the data is read just once from the external bus, and then stored for its future use.

4.3. Design of complete System: Design decisions

This section describes some of the decisions that have been made during the development of the system that envelops the coprocessor system.

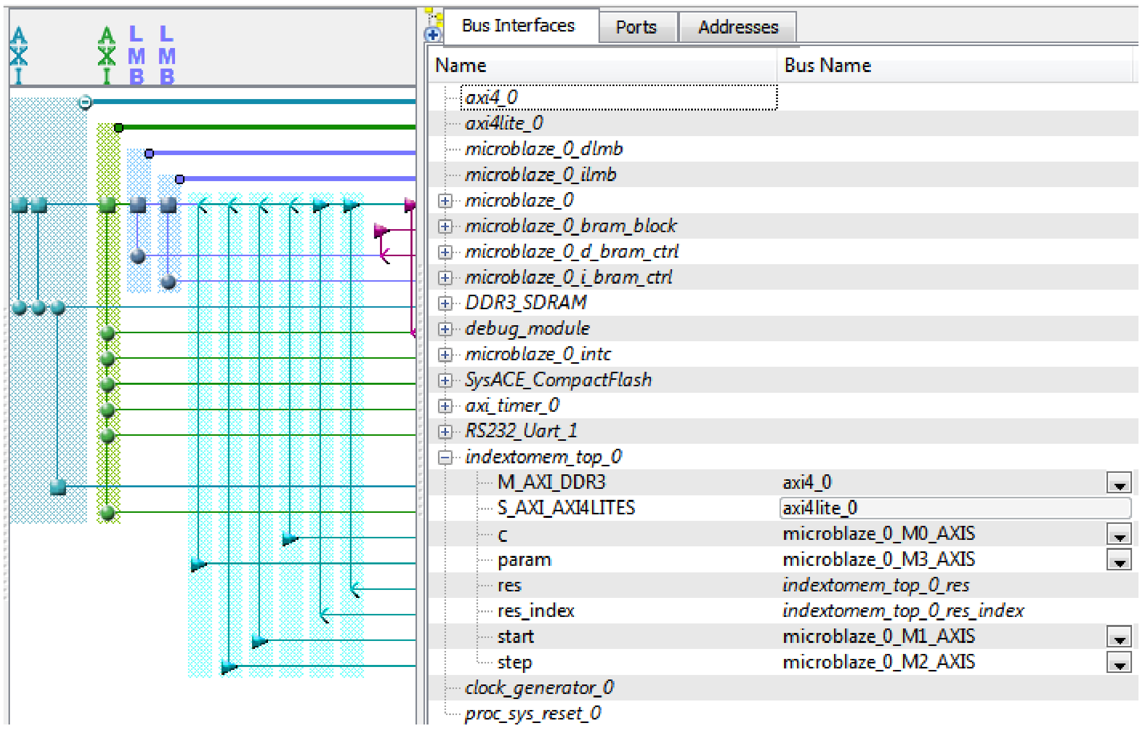
4.3.1. Embedded system

It was decided to create an embedded system because of the characteristics of the algorithm acceleration scheme (initialization, heavy algorithm and storage) and because it was the best option to access the peripherals needed, as the external storage card and the DDR3 RAM.

The system control is also assumed by the microprocessor, in this case a Microblaze [21], which is a soft-processor. It is connected to its peripherals with several AXI buses [22]. The following system peripherals are used:

- External memory (DDR3 RAM): It stores the LUTs once the system is initialized.
- Permanent storage: The LUTs are retrieved from a Compact flash memory during the initialization

Here it is presented a schematic view of the system as it is in Xilinx Platform Studio:



Name	Bus Name
axi4_0	
axi4lite_0	
microblaze_0_dlmb	
microblaze_0_ilmb	
microblaze_0	
microblaze_0_bram_block	
microblaze_0_d_bram_ctrl	
microblaze_0_i_bram_ctrl	
DDR3_SDRAM	
debug_module	
microblaze_0_intc	
SysACE_CompactFlash	
axi_timer_0	
RS232_Uart_1	
indextomem_top_0	
M_AXI_DDR3	axi4_0
S_AXI_AXI4LITES	axi4lite_0
c	microblaze_0_M0_AXIS
param	microblaze_0_M3_AXIS
res	indextomem_top_0_res
res_index	indextomem_top_0_res_index
start	microblaze_0_M1_AXIS
step	microblaze_0_M2_AXIS
clock_generator_0	
proc_sys_reset_0	

Figure 3: Previous work system schematic

4.4. Results

The prototype of the algorithm accelerator is working and the following steps are functional:

- Storing a reduced LUT (62 KB) in external memory.
- Gets the initial data from the Microblaze processor, and processes it according to one iteration of the algorithm.
- Returns results, in a prepared format for next iteration.

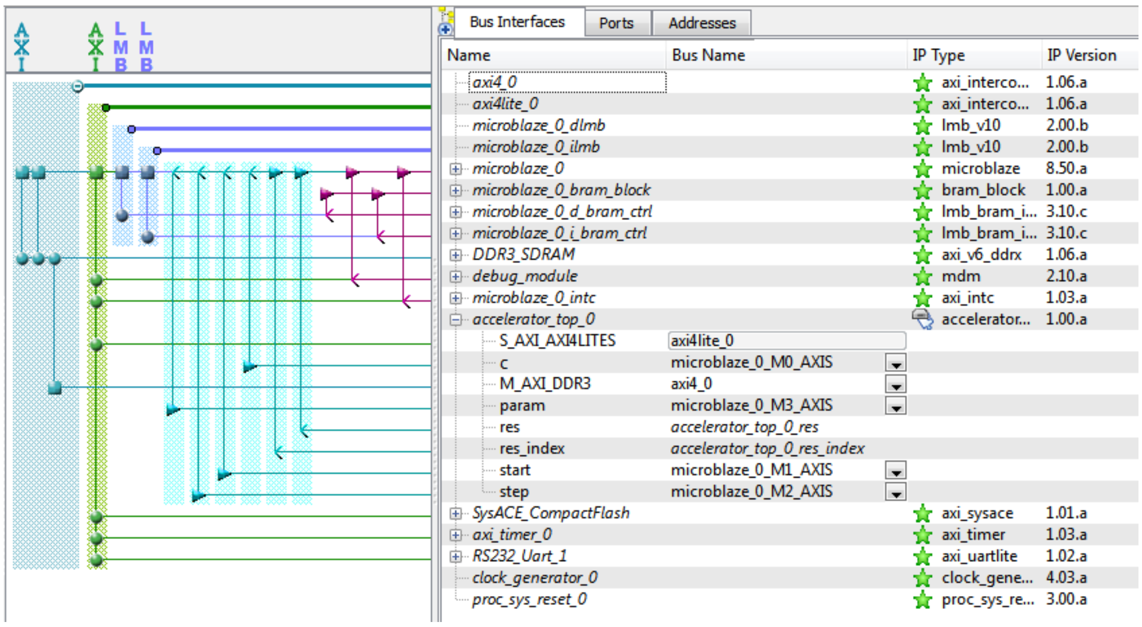
The latency achieved by the accelerator is 948 cycles of 10 ns period, which makes 9.48 µs. The total time of execution from the processor has also been measured, giving a result of 29.83 µs. The difference is mainly the time that is necessary to transfer the matrices to the accelerator.

The results are acceptable, but more optimization is needed if hundreds of images have to be processed, each of them requiring several iterations.

5. Hardware implementation: Zynq embedded system

5.1. Introduction

The hardware design in the previous work consists of a Microblaze processor, an accelerator core, a Compact Flash and DDR3 RAM, all set on a ML605 Virtex 6 platform (showed in Figure 4).



Name	Bus Name	IP Type	IP Version
axi4_0		axi_interco...	1.06.a
axi4lite_0		axi_interco...	1.06.a
microblaze_0_dlmb		lmb_v10	2.00.b
microblaze_0_ilmb		lmb_v10	2.00.b
microblaze_0		microblaze	8.50.a
microblaze_0_bram_block		bram_block	1.00.a
microblaze_0_d_bram_ctrl		lmb_bram_i...	3.10.c
microblaze_0_i_bram_ctrl		lmb_bram_i...	3.10.c
DDR3_SDRAM		axi_v6_ddrx	1.06.a
debug_module		mdm	2.10.a
microblaze_0_intc		axi_intc	1.03.a
accelerator_top_0		accelerator...	1.00.a
S_AXI_AXI4LITES	axi4lite_0		
c	microblaze_0_M0_AXIS		
M_AXI_DDR3	axi4_0		
param	microblaze_0_M3_AXIS		
res	accelerator_top_0_res		
res_index	accelerator_top_0_res_index		
start	microblaze_0_M1_AXIS		
step	microblaze_0_M2_AXIS		
SysACE_CompactFlash		axi_sysace	1.01.a
axi_timer_0		axi_timer	1.03.a
RS232_Uart_1		axi_uartlite	1.02.a
clock_generator_0		clock_gene...	4.03.a
proc_sys_reset_0		proc_sys_re...	3.00.a

Figure 4: Architecture in ML605 board

It was stated that there were technical difficulties with the access to the whole 512MB of DDR3 RAM. Also it was not possible to return data from the accelerator core to the processor. These problems have not been solved, and therefore another solution has been chosen.

5.2. Platform change

Since these problems, especially the one with the accelerator, were suspected to be associated to the platform and the software used (Virtex 6 and Vivado-HLS, respectively) it was proposed to implement the same solutions on a different hardware, fully supported by Vivado-HLS.

For it to be compatible the new FPGA must be a Xilinx Series 7. The chosen hardware is the Zedboard [23], due to the size of the DDR3 memory, availability and the new system on a chip (SoC) paradigm, which is in line with the embedded system used in this project. In this way the power of a true hard-processor would be exploited. This hardware was provided by the High Performance Computing and Networking investigation group (HPCN).

The Zynq family effectively overcomes the challenge of creating custom extensions while giving the engineer the ability to create their own application specific, using what is arguably the world's most widely adopted embedded processor [REF] as the basis for an accelerated, custom and application-specific solution that would, by definition, otherwise only be available as an ASIC.

The main features of Zedboard [23] are:

- Zynq®-7000 All Programmable SoC XC7Z020-CLG484-1 [16]: Includes an ARM Cortex-A9 Dual core processor and an Artix-7 FPGA [24].
- Memory:
    - 512 MB DDR3 RAM
    - 256 Mb Quad-SPI Flash
    - 4 GB SD card
- On board USB-JTAG Programming
- 10/100/1000 Ethernet
- USB OTG 2.0 and USB-UART

The Figure 5 shows the Zedboard and some of its features and ports.

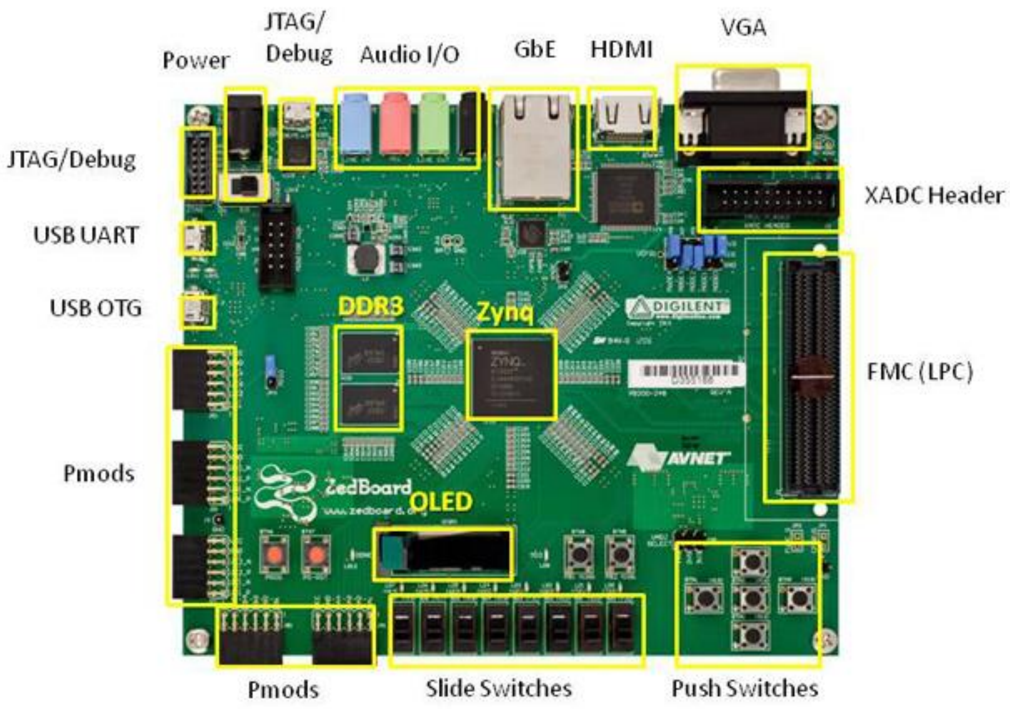


Figure 5: Zedboard and its main features

5.3. Architecture changes

The Zedboard packs a Zynq®-7000 All Programmable SoC XC7Z020, a system on a chip (SoC) which includes an ARM Cortex-A9 dual core processor and an Artix 7 FPGA within the same housing. The architecture of this solution is showed in Figure 6:

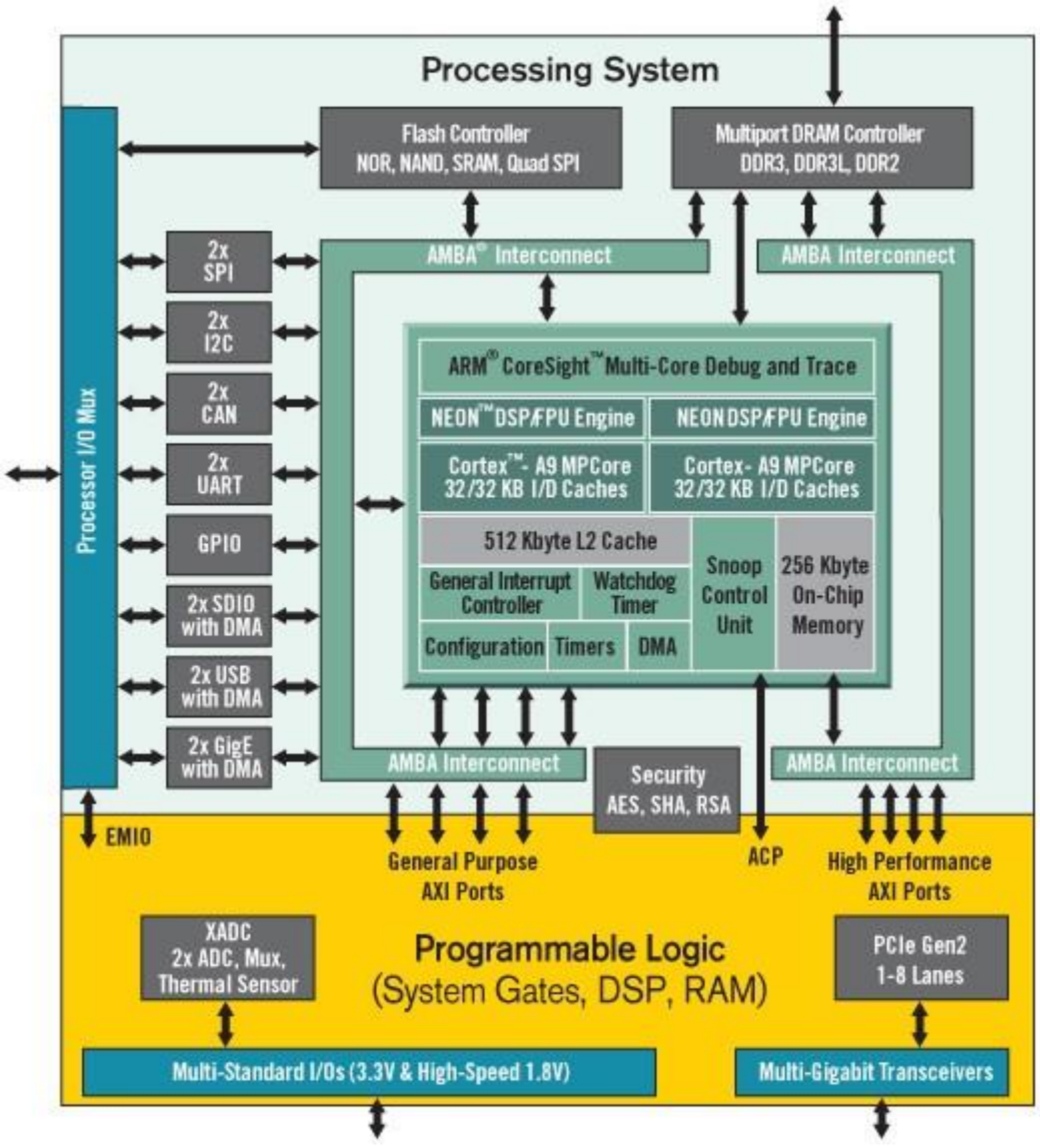

Processing System
Flash Controller NOR, NAND, SRAM, Quad SPI
Multiport DRAM Controller DDR3, DDR3L, DDR2
AMBA® Interconnect
AMBA Interconnect
Processor I/O Mux
2x SPI
2x I2C
2x CAN
2x UART
GPIO
2x SDIO with DMA
2x USB with DMA
2x GigE with DMA
ARM® CoreSight™ Multi-Core Debug and Trace
NEON™ DSP/FPU Engine
NEON DSP/FPU Engine
Cortex™- A9 MPCore 32/32 KB I/D Caches
Cortex- A9 MPCore 32/32 KB I/D Caches
512 Kbyte L2 Cache
General Interrupt Controller
Watchdog Timer
Configuration
Timers
DMA
Snoop Control Unit
256 Kbyte On-Chip Memory
AMBA Interconnect
Security AES, SHA, RSA
AMBA Interconnect
EMIO
General Purpose AXI Ports
ACP
High Performance AXI Ports
XADC 2x ADC, Mux, Thermal Sensor
Programmable Logic (System Gates, DSP, RAM)
PCIe Gen2 1-8 Lanes
Multi-Standard I/Os (3.3V & High-Speed 1.8V)
Multi-Gigabit Transceivers

Figure 6: Zynq architecture

In the Zynq architecture the processor is a hard core processor, which is different than the Microblaze processor (a soft core processor). A hard processor is more efficient than a soft processor, which is instantiated in the resources of an FPGA [25].

The Figure 6 also shows an interesting fact: in Zynq architecture most of the ports are linked to the ARM processor and not the programmable logic (the FPGA). In this case the FPGA is thought to help to the main processor processing tasks that are more appropriate for an FPGA because of their intrinsic characteristics.

5.4. Additional coprocessor optimizations

In the previous work [1] has been described how the coprocessor was optimized, taking advantage of the way in which Vivado-HLS synthesizes the C code into hardware. Besides the optimizations done before some more have been added that will prove critical for the performance of the coprocessor.

A usual technique used in Vivado-HLS consists in caching the input data, especially if it is going to be used more than once. This allows the synthesizer to store them in registers, or Block RAM (BRAM) if they are big enough. This is done through variable declaration and initialization with the input. An optimization done over these variables that store the input is to set an array partition directive. This directive makes the synthesizer to store the data in different memories (again registers or BRAM depending on the size). The ideal is to make a "partition" for every "chain" of operations that have to be fed. If so, every "chain" will be fed at the same time, since different memories can be asked for different data at the same time. This is a very important step to improve the performance of the module, since having four chains of multipliers and summers does not accelerate the operation if they cannot be provided with data every cycle.

An additional step in this direction is the difference between what it will be later be labelled as the "non-optimized" approach of the coprocessor, and the "optimized" one. The latter implementation makes use of the dual port that the BRAM provides. By doing so two 32-bits data can be provided by each BRAM in each cycle, so it does the same effect than partitioning the cached version input variables again (after the previous partitioning by rows of the matrix A) by a factor of two. In order to make use of this new data it is necessary to double the number of "production lines". It improves the performance of the module (in the operation part) by a factor of two without using more memory resources. In addition, it does use more FPGA area to set the additional multipliers and accumulators.

This last optimization has a potential to have an effect in the precision of the operations in floating point, because it alters significantly the order of the operations. Despite of implementing all the previous optimizations, if the algorithm implementation is analysed thoroughly, the operation order remains basically the same, which helps the “non-optimized” implementation to get the same results than the Java one. Despite, the order result is not the same in the “optimized” design. Now each half of the basic vector operated (the A rows, C, and B) is accessed by a different pipeline, and the two results, which correspond to each halve, are summed up in a final step. As relatively small numbers are operated, the results of this module are not the same as the ones obtained in Java or with the previous module.

5.5. Single core architecture

While the complete architecture is shown in Figure 7, the main parts of it are explained here:

ARM processor: It executes most of the code, which basically copies the required LUT from the SD card to de DDR3 RAM at the beginning of the execution and then provides the coprocessor Accelerator with the data needed in order to iterate over the algorithm. It serves as control during the operation and writes the results of a 1000-image test in “csv” files for its later analysis. It is the only component that is not instantiated inside the FPGA logic.

Coprocessor “Accelerator”: executes in an efficient way the matrix multiplication required by one iteration of the algorithm. Is coded in Vivado-HLS, and makes use of the BRAM (Block RAM) dual port. Receives the matrices from the DMA.

DMA (Direct Memory Access): A core that allows the Accelerator to efficiently retrieve the matrices from the DDR3 RAM without the active usage of the ARM processor. The hard processor serves only as a trigger for the operation. In the schematic showed in the figure 4 is linked to the Zynq processing system because the DDR3 RAM is only attached to the ARM processor (fig. 3). Despite of this the data goes through the AMBA bus to the coprocessor without involving the main processor.

DDR3 RAM: The matrices needed for every iteration are stored here, so it is accessed every iteration.

AXI Timer: It allows the measurement of times in a precise way. It runs with a clock of 100MHz, as the rest of the FPGA.

5.6. Resources used

The following table describes the resources used by the whole system in xc7z020 SoC FPGA (notice that the ARM processor is a hard processor and has no effect of the resources utilization):

Table 1: Utilization report summary

Resource	Utilization	Available	Utilization %
FF	6056	106400	5,69
LUT	5313	53200	9,99
Memory LUT	208	17400	1,20
BRAM	9	140	6,43
DSP48	10	220	4,55
BUFG	1	32	3,13

If this resource usage would scale maybe another 8 coprocessors could be added. This is not a definitive conclusion: due to the increasing route space needed when the design complexity grows the maximum amount of instances may be lower.

Table 2: Utilization per module

Name	**Slice LUTs**	**Slice Reg**	**F7 Muxes**	**Slice**	**LUT as Logic**	**LUT as Memory**	**LUT Flip Flop Pairs**	**Block RAM Tile**	**DSPs**	**Bonded IOPADs**	**BUFGCTRL**
rst_processing_system	17	31	0	13	16	1	22	0	0	0	0
processing_system7_0_axi_periph	584	715	0	282	521	63	766	0	0	0	0
processing_system7_0	0	0	0	0	0	0	0	0	0	0	1
axi_timer_0	277	213	1	111	277	0	290	0	0	0	0
axi_mem_intercon	1103	1340	0	483	1034	69	1473	0	0	0	0
axi_dma_0	1354	1877	1	660	1317	37	1919	3	0	0	0
accel_core_0	1979	1880	0	886	1941	38	2560	6	10	0	0

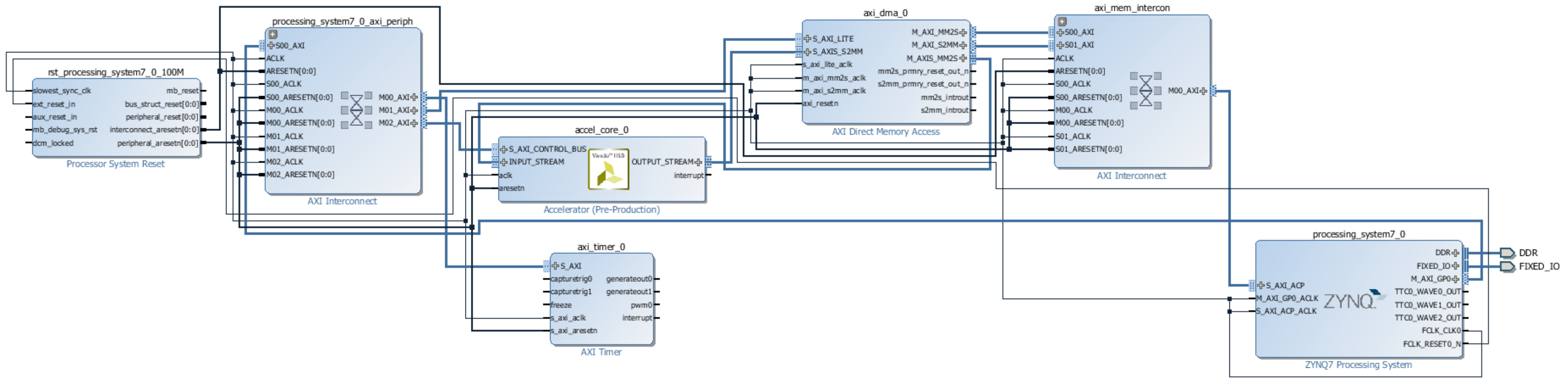


Figure 7: Complete schematic of monocore design in Vivado

6. Look-up table generation

6.1. Gaia WFS LUT creation software

A program was created within the Gaia team [26], in order to create the Look-up table (LUT) that will be stored in the DDR3 RAM in the embedded system.

The program starts with several configurations that are read from a configuration file. Among the most important, the characteristics defining the wavefront sensor itself: lenslet spacing and diameter, central pixel of each sub-pupil, number of pixels per PSF in AL and AC dimensions…

The configuration parameters for the LUT are set in the beginning of the program, as constants. They are the start, end and number of knots for every dimension (x, y, electrons and diameter).

Then, inside four loops (one for every dimension) the Jacobian matrix resulting from the function that defines the wavefront sensor is retrieved and the operations referred in subsection 2.6 of [1] are calculated (pseudoinverse, transposed, etc.). This gives as result the transfer matrix, which is stored in a binary file.

6.2. Endianness issue

After the LUT is created it has to be read by the embedded system, with an intermediate step of copying the binary file from the PC to a SD card. It had to be proved that the data written in the LUT file was going to be read in the same way from the SD card in the embedded system.

The solution consists on writing a certain and known stream of data resembling the LUT. In this case single precision floating point data written to binary file, and then reading it in the system just as the real LUT would be read. As the embedded system is little-endian, the data written with Java in a binary file had to be read reversing the byte order in each word.

With the LUT creator program it was created a LUT of 64 MB, which had 12, 12, 11 and 21 knots in x, y, s and N parameters, respectively. The resulting size is exactly 33264 arrays of 500 floats, which makes a total of 66528000 bytes, so 63.446 MB.

6.3. LUT Geometry: trade-off between size and effectiveness

The algorithm is written in Java language, in order to comply with previous versions of it. The most critical resource that affects the algorithm is the effective size of the DDR3 RAM memory, which strongly constrains the maximum grid density of the stored LUT.

The algorithm has been tested before with LUT interpolation, which requires bringing 2^n tables of 2000 bytes, where "n" is the number of dimensions of the LUT, only considering linear interpolation. Although in a non-realistic software system the time loading the tables may not be important, in a time constrained hardware system it is crucial. This is why in an embedded system interpolation is not the first option to consider, as it may affect the performance of the algorithm.

With this in mind, if there is no interpolation the algorithm has to be modified, because there will be a point in which the parameters of the matrix that have to be retrieved are not going to match with a stored one.

For example, if the algorithm needs to retrieve the matrix corresponding to X=4.5623 and the stored LUT only contains the ones corresponding X=4.56 and 4.57 the algorithm will assume that the matrix of the nearest LUT knot (4.56 in this case) is the same matrix that is needed (the one corresponding to 4.5623). This statement is not true, and then the effects of this assumption have to be analysed.

Because of this assumption, if the LUT is denser, there will be a lower difference between the ideal transfer matrix and image and the retrieved ones. Therefore, the error of the algorithm will be lower and its behaviour more predictable.

6.4. Memory limitations and its effect on LUT density

The embedded system is built on a Zedboard [23], which has 512MB of DDR3 RAM as a peripheral device of the processor-FPGA set. Therefore, the system was expected to store any LUT of less than the maximum RAM size. Despite of this, and because of the technical reasons explained in the third section of this document, it only has been possible to store without failure only 64 MB of information. The project is therefore limited to this LUT size and the expected precision is not going to be as good as it would be with a less-restricted LUT. This point will be discussed on the section 7.

As it is derived in [26], the total size of the LUT is the result of the following operation:

$$Size(bytes) = (Num\,param + 1) * pixel_x * pixel_y * knots_x * knots_y * knots_d * knots_N * sizeof(float)$$

In this case $Num\,param$ equals 4, because there are 4 parameters, $pixel_x$ and $pixel_y$ are the number of pixels in both axes of the PSF, and as the values are stored in single precision floating point each value needs 4 bytes to be stored.

These four parameters refer to the four variables that define an image (PSF) in this system:

- X or AC: Is the coordinate in the X or across scan axis which indicates where the centre of the PSF is.
- Y or AL: Is the coordinate in the Y or along scan axis which indicates where the centre of the PSF is.
- Electrons (N): Is the total number of electrons that constitute the image.
- Diameter (d): Measures microlens diameter in meters (usually around 380µm).

Once these variables are fixed, all that can be changed is the number of knots for each variable, that will define a grid. At each point of this grid (every combination of the 4 parameters values) there will be a matrix stored, composed by the corresponding image and the transfer matrix. The total space required for both matrices is 2000 bytes (500 simple precision floating point values). So in this case the above formula can be simplified to:

$$Size(bytes) = 2000 * knots_x * knots_y * knots_d * knots_N$$

It is clear then that limitations on the maximum amount of data that the system will store constrains the maximum LUT density.

6.5. Several LUT constructions

In [26] it is stated that a reasonable LUT, taking in account the limitations of the maximum amount of memory, would be one with 21 knots in each dimension, except for the electrons that would have only 20. This LUT results according to the above formula in a total size of 370440000 bytes, which is 353.27 MB. It would fit in the 512 MB of RAM that the Zedboard has, and it is reasonable because it covers a reasonable portion of the parameter space: from 4.0 to 6.0 pixels for both X and Y axis, microlens diameter within 350×10^{-6} and 450×10^{-6} meters, and from 0 to 10000 electrons.

With the new 64 MB limit this LUT had to be modified in order to fit in this size. The first decision taken was to reduce the range for X and Y, since it was expected that the slopes measured with the WFS would not be so harsh, and therefore the PSF would be around the centre. It would be desirable to keep the whole range for the electrons axis, in prevision of very different types of stars, and as it was thought that the diameter microlens would be similar from one to another (near diffraction limited performance), the range was also reduced. As it can be deduced from the ranges and the amount of knots in each variable, the density was not reduced in any of the parameters, in order to avoid disturbances in the performance of the algorithm. In the end, the final range was:

	Range start	**Range end**	**Knots**
X	4.5	5.5	12
Y	4.5	5.5	12
Diameter	380e-6 m	430e-6 m	11
Electrons	0	10000	21

Table 3: 64 MB LUT

7. Convergence and precision tests

7.1. Introduction

As it was stated in Subsection 6.3, there is a trade-off between the size of the LUT, which consists basically on the number of knots for each parameter, and the performance that the algorithm will achieve.

In this section it will be found out what would be the best LUT under these conditions.

7.2. Description of the algorithm in a Java program

Two classes were developed, based on [26], which create images (PSFs) and perform the fitting algorithm, respectively (TestImageCreation and TestLutFitLoop). In order to be able to test the performance of the algorithm they have been modified. The aim is that these modules can work together producing a realistic test over a large set of noiseless or noisy images, returning a file with the real and fitted parameters of each image, along with the error in each fit, both relative and absolute.

At the beginning of the program a set of variables are manually set, for the program to know which configuration and LUT file to use. Also here the boundaries for the image creation parameters are set, as well as the noise that will be added to the image and the amount of values that each parameter will take in the LUT coordinates. This late parameter defines the number of images that will be created and tested, because there will be as many as all the combinations of the parameter values. So, for example, if this parameter is set to 10 and there are 3 values included in the LUT, the amount of images created will be a thousand (10^3).

After setting these parameters two arrays are created in order to store the LUT file, which contains the transfer matrix and the model image for each node. This process is done reading the file with four loops that change the array indices where the values are stored.

The next step is to generate an image and fit its values. The purpose of the program is to compare the performance of the algorithm using different loops but the same set of images, so again four loops are created that set the creation parameters for the image to be analysed. The result is a cloud of known points for every parameter. Each combination among the parameter knots results in an image (either noiseless or not), which is fitted with the algorithm.

With each combination of values an image creation method from class "TestImageCreation" is called. The method "getNoiseLessImage" parses these values and then creates a lenslet object which simulates the properties of a real lenslet. Then the function that defines this object is retrieved and the image defined by the "getInputVariablesCache" is returned.

The algorithm consists of an iterative fit that get the closest transfer matrix and model images in the LUT (already stored in an internal variable) and follows the steps that are described in [26]. It basically consists of these two operations:

- Compute update vector $\Delta \boldsymbol{x} = \boldsymbol{M}(\boldsymbol{O} - \boldsymbol{N})$
- Update parameters vector $\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \Delta \boldsymbol{x}$

In the Java program $\boldsymbol{M}$ is the transfer matrix, $\boldsymbol{N}$ the model image and $\boldsymbol{O}$ the created image. The parameters are subsequently updated and a new iteration is carried out.

After each image is processed the results of the fit are written into a text file, including the fit achieved and its error compared with the original coordinates with which the PSF was created. With this file it is easy to compute statistics that are relevant to select the best LUT.

It can be seen that the computational core of this algorithm is to perform multiplications and subtractions over large matrices, 400 and 100 floating point values for the transfer and model image matrices, respectively. In a hardware platform, as a FPGA, these operations can be parallelized and pipelined, so the time to calculate each iteration would potentially be reduced compared to the software implementation.

7.3. Achieving convergence on an initial image

Once the algorithm is ready, the first step is to make sure that it can fit the parameters with an easy, noiseless image. It is required that the parameters resulting from the fit are reasonably close to the real parameters set in the creation of the analysed PSF.

In order to ensure that the algorithm works as expected it is better to start with a very small LUT centred around the final point which the algorithm has to recover. Due to its little size the density of the grid is going to be high resulting in a reduced chance of algorithm failure. A good initial guess will also help to rapidly find some LUT parameters which are good for the algorithm.

The initial image (Figure 8) is going to be created with realistic parameters, as well as the LUT.

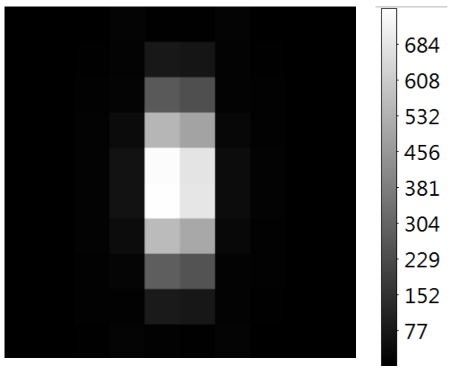


Figure 8: Initial image to fit

Table 4 presents the parameters used to create the initial image. They have been chosen applying a slight variation over the most typical parameters.

Table 4: PSF parameters

	Image parameters
X_AC	4.457
Y_AC	4.473
ELECTRONS	7931
DIAMETER	375.4x10-6

To begin an arbitrary LUT was built with the set of parameters written in Table 5. The parameter range in this LUT is very narrow because it is easier for the algorithm to converge if the LUT density is higher.

Table 5: Initial arbitrary LUT parameters

	Start	End	Knots
X	4.4	4.5	11
Y	4.4	4.5	11
N	7900	8000	11
D	360	379.999	11

The algorithm did not converge with this LUT. The program that performs the fit algorithm also gives information about the failure, which is usually that the algorithm tries to retrieve a matrix that is not within the bounds of the LUT in some parameter.

Table 3 presents the LUT parameters (start, end, number of knots and step) at which the algorithm converges. It also includes the initial guess, the fit that the algorithm achieves and the absolute error according to this formula:

$$Absolute\ error = Final\ fit - Image\ parameter$$

It can be seen in Table 6 that the most conflictive parameter was the diameter (D), because it was necessary to have 41 knots for the algorithm to converge. At some point the algorithm also went out of the electrons (N) parameter LUT limits, so 21 knots were set.

Table 6: Parameters of first fit

	Start	End	Knots	Step	Image	Initial Guess	Final Fit	Abs. Error
X	4.4	4.5	11	0.009091	4.457	4.44	4.45746	0.00046
Y	4.4	4.5	11	0.009091	4.473	4.46	4.47781	0.00481
N	7900	8000	21	4.761905	7931	7940	7940	9
D	360	389.999	41	0.731683	375.4	376	377.98	2.58

7.4. Studying convergence over an image with added noise

The goal is exactly the same as in the previous subsection, but for an image with a typical read-out noise of 8.44 electrons per pixel. This amount of noise is considered to be usual in a Gaia-like wavefront sensor [27]. The resulting image has the same parameters as the preceding one, and is presented in figure 2.

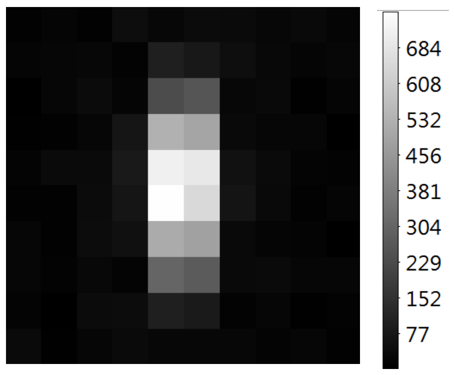


Figure 10: Initial image with added noise

In this case the image depends strongly and arbitrarily of the LUT density and bounds of the diameter parameter. It was not an easy task to find some parameters at which the algorithm could fit the PSF, because little variations which were supposed to be beneficial for the algorithm; like increments of the LUT knots in the diameter dimension, decreasing LUT bounds limits so the density increased or widening the LUT limits and the number of knots; made the algorithm diverge. In most cases the algorithm failed because the diameter was diverging without limit, but it sometimes affected also the electrons variable making it diverge.

The most problematic variable being the diameter, it was decided to remove this variable, fixing it to the typical value: 378 µm [28]. In consequence the LUT would have one less dimension, although it was preferred to keep the program with the four dimensions, reducing the diameter to only one knot at the typical value. This means the quoted LUT sizes are upper limits, and further size optimisation is still required.

This is an important step, because it indicates that the algorithm will not take into account different PSF forms. It could also suggest that the function variation over the diameter parameter is not linear, so an algorithm with a non-linear interpolation may be required for this parameter to be taken in account.

Once the diameter dimension was taken away, the algorithm was able to fit the image in nearly every case, either for noise or noiseless images. It only diverges if the image centre is set very close to the borders of the LUT range.

7.5. Enlarging the parameter space covered by the LUT

Once it was ensured that the algorithm would be able to fit at least a certain image with certain a set of parameters, the aim was to create a new LUT which covered a wider range. Ideally it should cover the range of parameters presented in table 3 in order to fit the parameters of images within that range, which are the ones expected to be provided by a WFS pre-processor.

Table 7: Ideal parameters of LUT

	Start	**End**
X	4.5	5.5
Y	4.5	5.5
N	1	10001

Now the diameter parameter is discarded because is being fixed at $378x10^{-6}$ meters, both in the LUT and in the generated images, so it does not affect the algorithm.

These parameters provided good results with an arbitrary number of knots for each of them. The algorithm always converged to reasonable values when the start values are close-enough and images that were not too close to the borders. This later condition is important because if the initial image is too close to the LUT limits the algorithm may produce updates outside of the LUT domain.

7.6. Effects of the grid density on the precision

The next step is to test the effect of the LUT density on the precision of the algorithm. The premise is that, as the LUT becomes denser, the algorithm error gets lower. because the algorithm assumes that the retrieved matrix is the correct one. As it is explained in 1.3 subsection it is assumed that the closer retrievable matrix is the correct one, which is not true most of the time.

To test the algorithm LUTs of several sizes are created, starting with 21 knots per variable (just for X, Y, and the electrons, the diameter was fixed to one value) and continuing with 41, 81 and 101. The boundaries of the LUT remain untouched at the ideal values of the previous section.

A thousand images were created from a pseudorandom generator. This allows the images to be random at grid effects while the same set of images can be produced in order to test the different LUTs with the same conditions. By setting a random pattern for the image parameters a systematic error is avoided. This systematic error would be introduced if the grid of images would be too similar to one LUT grid, but very different from another one. The result would be a lower error in the first case (because LUT knots would be artificially close to the image ideal fit), and a higher one in the second.

The parameter boundaries for the image creation have been set avoiding the closest areas to the LUT borders. This precaution has been taken because if the border is very close to the goal value for one parameter the algorithm would most likely search for a value out of the LUT range, and the algorithm would fail. If such a case would happen it would be necessary to enlarge the LUT, expanding its range in order to ensure the effectiveness of the algorithm.

The next figures show the behaviour of the absolute error (average and average deviation) for different LUT constructions (number of knots in each dimension) in noiseless images:

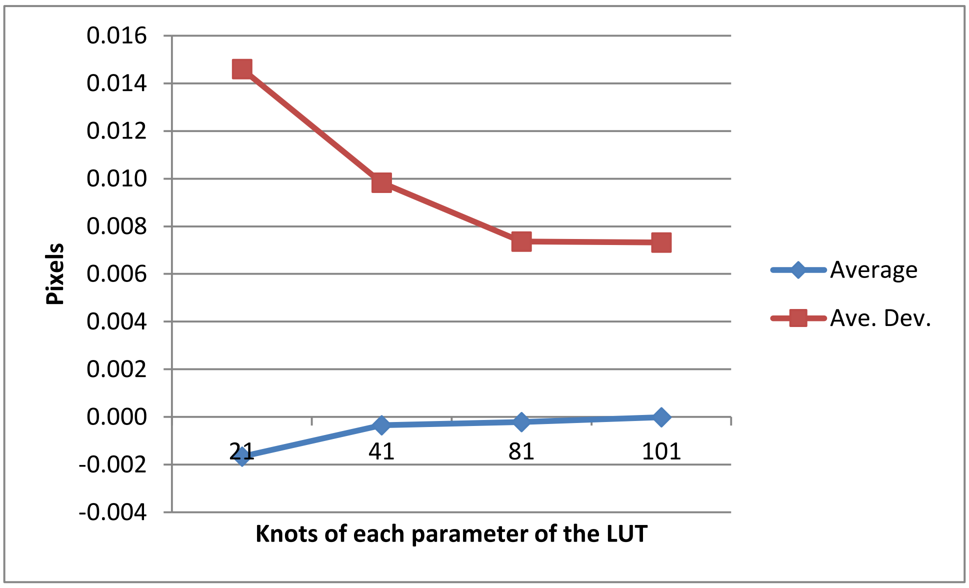


Figure 11: Absolute error in X parameter for noiseless images in pixels

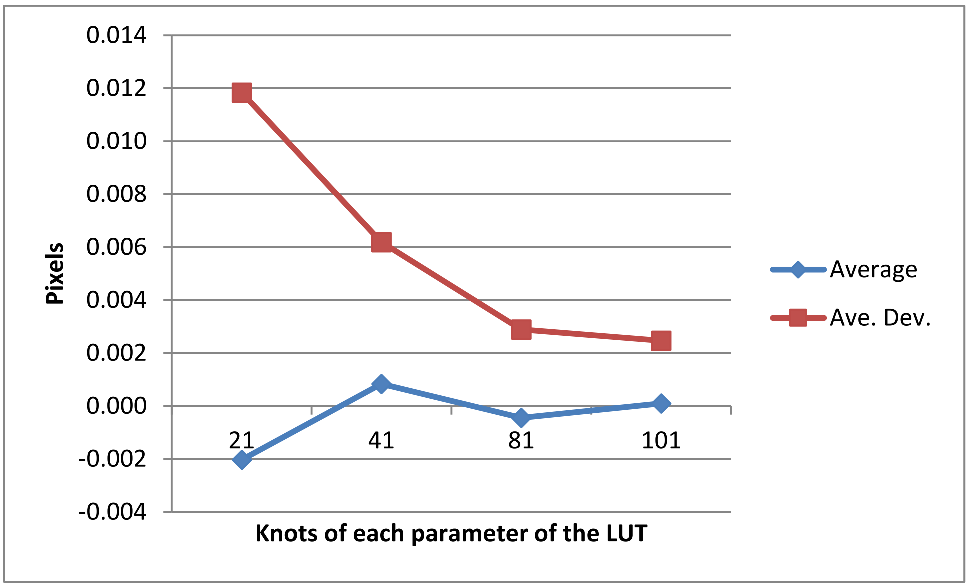


Figure 12: Absolute error in Y parameter for noiseless images in pixels

As expected, the average deviation on the absolute error (Ave. Dev. In the graphics above) diminishes as the LUT size increases, because the effect of quantisation gets smaller as there are more matrices. That is, the matrices are closer to the exact point that the algorithm is aiming to.

It is important to note that, as the LUT gets bigger, the average deviation on the absolute error does not get that much smaller. This is because there is a limit to the precision that can be achieved, which is set ultimately at the Crámer-Rao lower bound, even if there was no quantisation.

However, these images are noiseless, and therefore not realistic. While in noiseless images the average deviation of the error should always decrease as the LUT has more knots, because it would be approaching a perfect continuous function, it should not always get better if there is noise in them. If the image has noise, or the numerical precision impacts the algorithm precision, the limit where the algorithm performance improvement is negligible as the density of the LUT increases should be reached much sooner.

The next figures show the behaviour of the error (average and average deviation) for different LUT constructions (number of knots in each dimension) in noisy images:

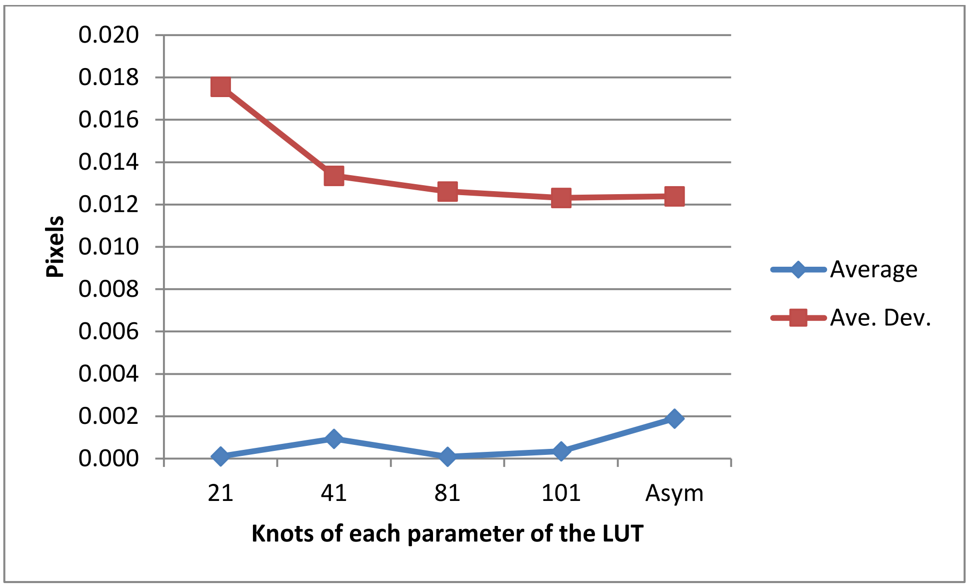


Figure 13: Absolute error in X parameter for noisy images in pixels

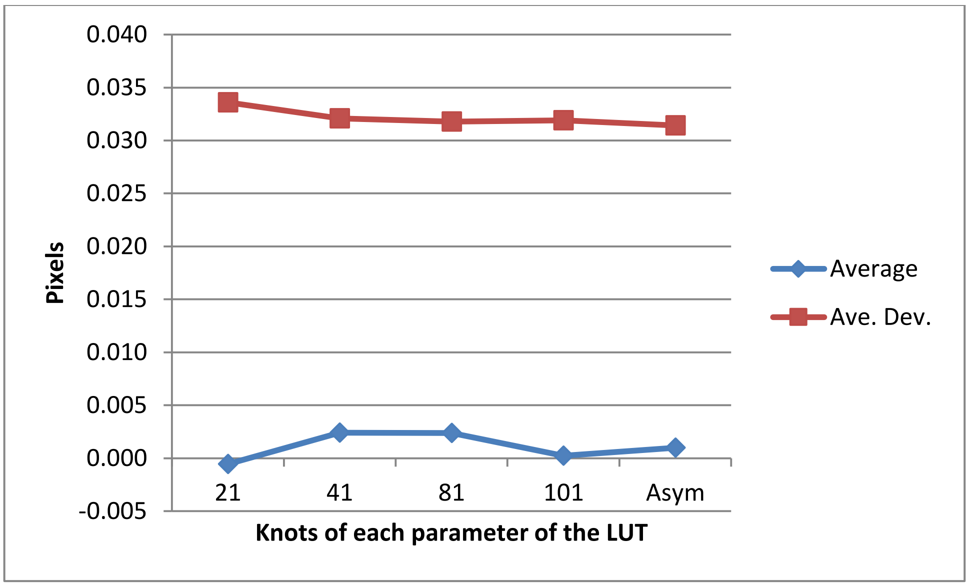


Figure 14: Absolute error in Y parameter for noisy images in pixels

It can be seen in figures 5 and 6 that the slopes are not as inclined as the ones in figures 3 and 4. For example in the Y axis the difference between the error with the 81-knot LUT and the one with 101 is negligible. Even the difference between 41 and 81 knot LUTs is not that big.

For the X variable the effect is more or less the same, although the average deviation of the error continues to go down even with the 101-knot LUT. This suggests that the error is not getting lower in X and Y dimensions at the same pace.

This is why there is a fifth point in figures 5 and 6, that is noted as Asym (short for asymmetrical). Until now all the LUTs were completely symmetrical in the remaining three dimensions, so the LUT of 21 nodes has 21 values in X, 21 values in Y and 21 values in the electrons axis. But as these last figures suggest that the error is not symmetrical, it is thought that a non-symmetrical LUT would either reduce the error for the same number of nodes or maintain the error level reducing then the size of the LUT.

Specifically, this LUT is constructed with 121 knots in the X dimension, 41 in the Y dimension and 81 in the N dimension, taking the points in which the difference in the error with the next LUT was not really big. In this case the LUT has a total of 401,841 nodes, while the 101 uniform LUT has 1,030,301, so the asymmetrical one is just a 39% of the 101 size. Even comparing its size with the 81 nodes symmetrical one the asymmetrical still takes 25% less space.

The size of the asymmetrical LUT is 766 MB, which would not fit in the DDR3 RAM in the Zedboard, which has a capacity of 512 MB. Despite of the storage convenience of the asymmetrical LUT there is a concern about the increase of the average error. It can be seen that the average error is reasonably low with the symmetrical LUT, especially in the fifth figure, but there is a peak of it with the asymmetrical one. There is still no knowledge about the reason of this anomaly, although it is thought that some outsiders could be affecting the measure. Despite of this the behaviour of the average deviation is at least as good as the 101knots LUT.

7.7. Relation between error in pixels and length units

The whole algorithm is based on the way the Gaia wavefront sensor works. The main difference of the Gaia WFS with a usual one is that the pixels are not squared, but rectangular. The length of one pixel is 10 µm in the Y direction (along scan) and 30 µm in X (across scan). This difference between the physical lengths of each side of a pixel has consequences on how good the sampling is in each direction. In this case it is clear that the sampling is better in the Y axis than in the X axis.

Although the size of the pixel in each axis would indicate that the error should be greater in the X axis than in the Y axis this is not true in pixels. While the star PSF is originally circular in the physical space, when this space is translated into pixels, the shape of the PSF is not circular anymore, but elliptical, with the greater diameter situated along the Y axis. This means that the PSF has more pixels in the Y axis than in the X axis (as can be seen in the figure below). As the absolute error measuring a large amount is bigger than the absolute error measuring something smaller, the error in pixels in the Y axis is bigger than the error in pixels in the X axis (the Y axis is longer in pixels than the X axis).

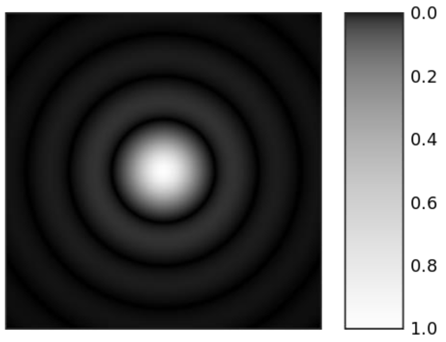


Figure 16: PSF before being sampled

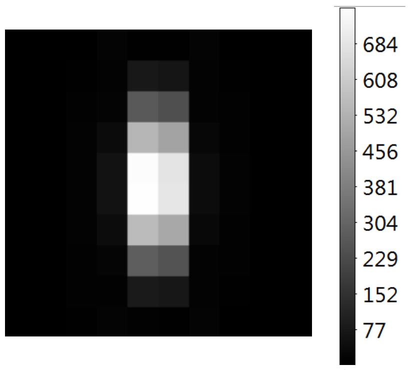


Figure 17: Projection of an elliptical PSF over 10x10 pixels

8. Time analysis

The algorithm consists of an iterative fit that get the closest transfer matrix and model images in the LUT (already stored in an internal variable) and follows the steps that are described in [26]. It basically consists of these two operations:

- Compute update vector $\Delta\boldsymbol{x} = \boldsymbol{A}(\boldsymbol{C} - \boldsymbol{B})$
- Update parameters vector $\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \Delta\boldsymbol{x}$

In the program $\boldsymbol{A}$ is the transfer matrix, $\boldsymbol{B}$ the model image and $\boldsymbol{C}$ the created image. The parameters are subsequently updated and a new iteration is carried out.

8.1. Time tables

The architecture presented in Figure 7 have been implemented with a first version coprocessor which is not taking advantage of the BRAM's dual port (non-opt accel in the table), and a more optimized Accelerator that does (opt accel). The column that corresponds with ARM are times taken when the algorithm is done by the ARM processor. In this case the times corresponding to the matrix transference is not that relevant, because burst capacity has not been used. However, in both accel case a DMA has been utilised.

Times have been taken for each part of one iteration of the algorithm, and are presented in cycles of 10ns in :

Table 8: 98% confidence values in 10ns cycles

	ARM	Non-opt accel	Opt accel
Send C Matrix	501	307	307
Send B Matrix	680	328	327
Send A Matrix	3727	624	626
Compute Time	2392	537	308
Total Time with middle times	7311	1883	1651
Difference	11	87	83
Total Time w/o middle times	7124	1693	1478
Send C Matrix (strict)		139	139
Send B Matrix (strict)		160	160
Send A Matrix (strict)		459	459

The times showed in the are times with a 98% of confidence for 2000 iterations of the algorithm.

The rows in match the following descriptions (and difference of times in the pseudo-code that follows them):

Send C Matrix: Cycles required to send the real image matrix (time 8 – time 7).

Send B Matrix: Cycles required to send the model image matrix (time 2 – time 1).

Send A Matrix: Cycles requires to send the model matrix (time 4 – time 3).

Compute time: Cycles used in the matrix multiplication (time 6 – time 5).

Total Time with middle times: Measuring times is a function and takes its time by itself. The total time is the number of cycles passed from the beginning of the iteration until the results are generated, when middle times are taken, in order to know how long it takes to send the matrices and computation step by step.

Difference: Difference in cycles between "Total time with middle times" and the sum of all the matrix transferences and the compute time.

Total Time w/o middle times: Total cycles for an iteration without measuring particular times.

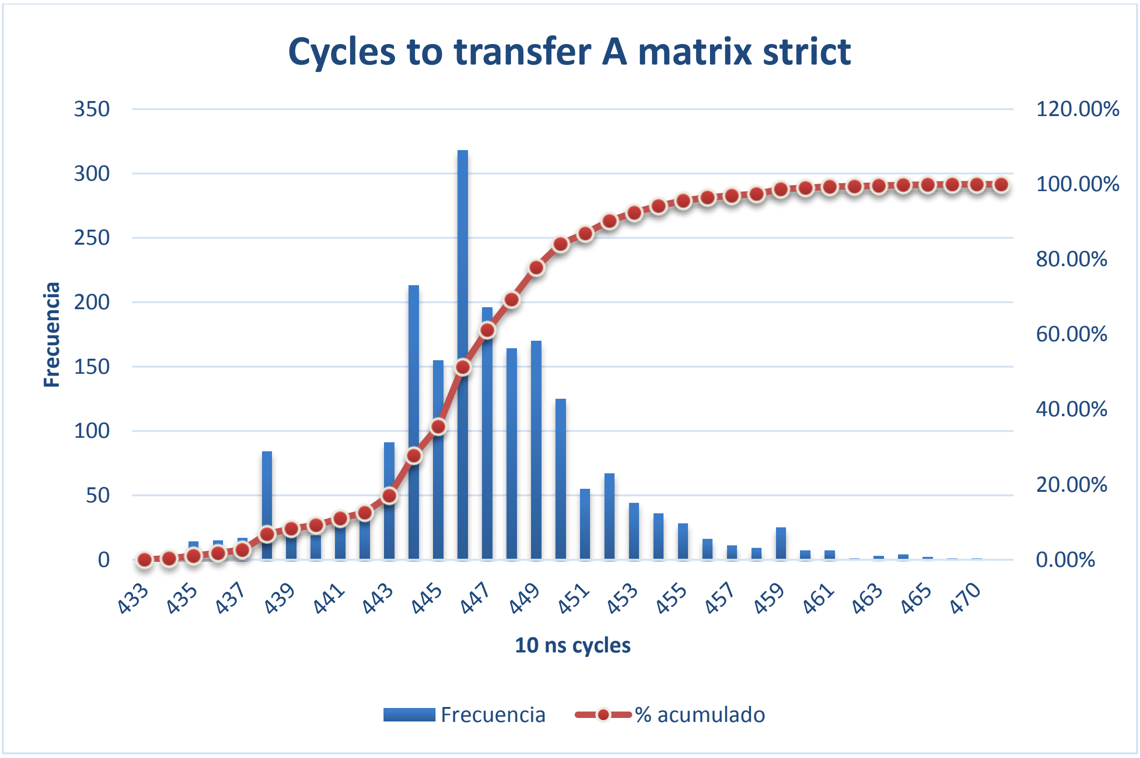
Send C/B/A Matrix (strict): It is the same as "Send C/B/A Matrix", but it does not include the call to the function that controls the DMA. So it only includes the transfer itself. C: (time 8 – time 16). B: (time 2 – time 17). A: (time 4 – time 18).

The following code is a pseudo-code that shows when each time is measured:

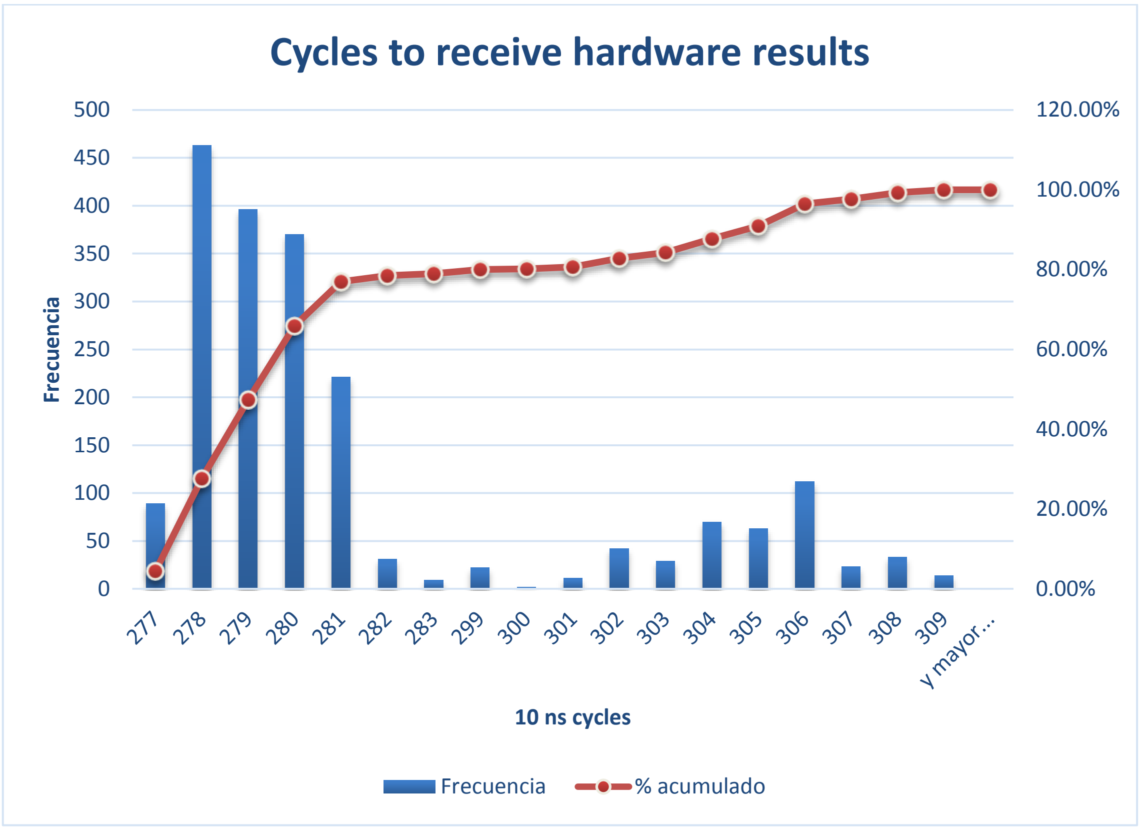
```
Begin_time = gettime();
Time_7 = gettime();
DMA_transfer(C, toCoprocessor);
Time_16 = gettime();
waitForDMAToFinish();
Time_8 = gettime();
Time_1 = gettime();
DMA_transfer(B, toCoprocessor);
Time_17 = gettime();
waitForDMAToFinish();
Time_2 = gettime();
Time_3 = gettime();
DMA_transfer(A, toCoprocessor);
Time_18 = gettime();
```

```
waitForDMAToFinish();
Time_4 = gettime();
Time_5 = gettime();
DMA_transfer(results, toCPU);
waitForDMAToFinish();
Time_6 = gettime();
End_time = gettime();
```

There have been extracted two histograms for “A” matrix transference (the strict one) and for the processing time in hardware:



Histogram 1



Histogram 2

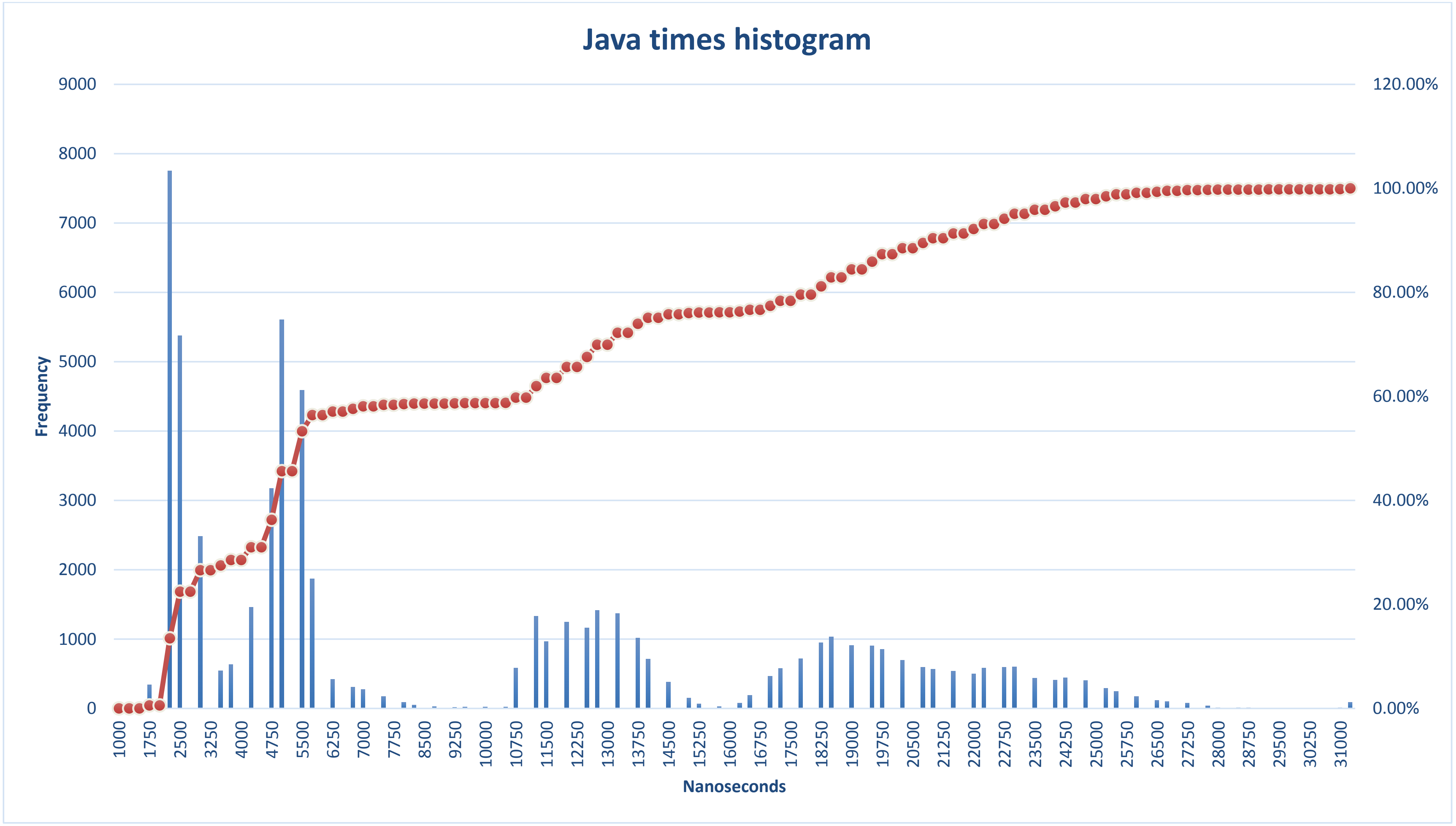
These histograms show that although FPGA circuits are supposed to be deterministic, these times are not. The variation may be due to communications with DMA, which communicates itself with the DDR3 RAM and the ARM CPU, which are not deterministic.

The Table 9 shows the processing times in µs, in order to compare with Java times, which are not measured with the same cycle:

Table 9: Times for an iteration in µs

	ARM	Non-opt accel	Opt accel	Java 95%	Java 98%
Time w/o mid times	71.93 µs	17.32 µs	14.85 µs	23.811 µs	25.864 µs

Times in Java are chosen to be maximum for which 95% and 98% of the iterations were done because the distribution of the times is very irregular and varies a lot. This point is showed in Histogram 3. This great variation responds to several causes. First the virtual machine that Java uses optimizes the loops as it is going through them, for example caching the data that is more likely to be used. Also the program is running over a machine with an operative system, so the scheduler can be delaying this process in order to run another one.



Histogram 3: Times for Java iteration (ns)

8.2. Time conclusion

1. As it can be seen in Table 9, the time of the most optimized accelerator is smaller than the one in Java, although the accelerator is implemented in a low-end FPGA that runs at 100 MHz and the Java application is running on an Intel Core i7 processor, which runs at 3.5 GHz.
2. Despite of this, the acceleration is just 1.74x, which indicates that the transaction of matrices from RAM memory to the processing unit is the bottleneck in this algorithm.
3. The maximum performance achievable in an iteration would need to hide the processing time (around 300 cycles) behind the transfer time (around 1250 cycles), which is an improvement of just 24%. That is the limit within the current architecture.

9. Precision evaluation

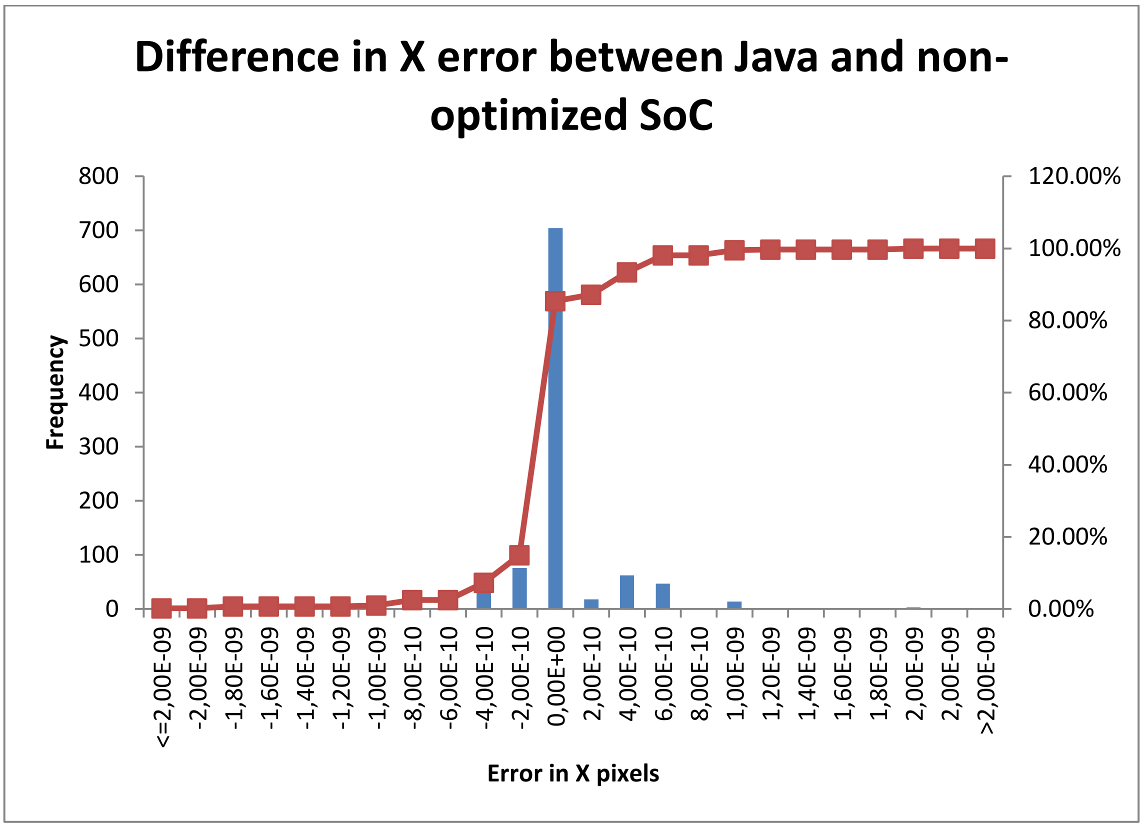
The fact that floating-point numbers cannot precisely represent all real numbers, and that floating-point operations cannot precisely represent true arithmetic operations, leads to many surprising situations, especially when dealing with relatively small numbers [29]. This is related to the finite precision with which computers represent numbers.

Also while floating-point addition and multiplication are both commutative, they are not necessarily associative. That is, $(a + b) + c$ is not necessarily equal to $a + (b + c)$ [29]. As a consequence, some implementations of one same algorithm may not lead to the same results than others. This is of especial application when optimizing an algorithm in FPGA, due to the parallelism that can be implemented.

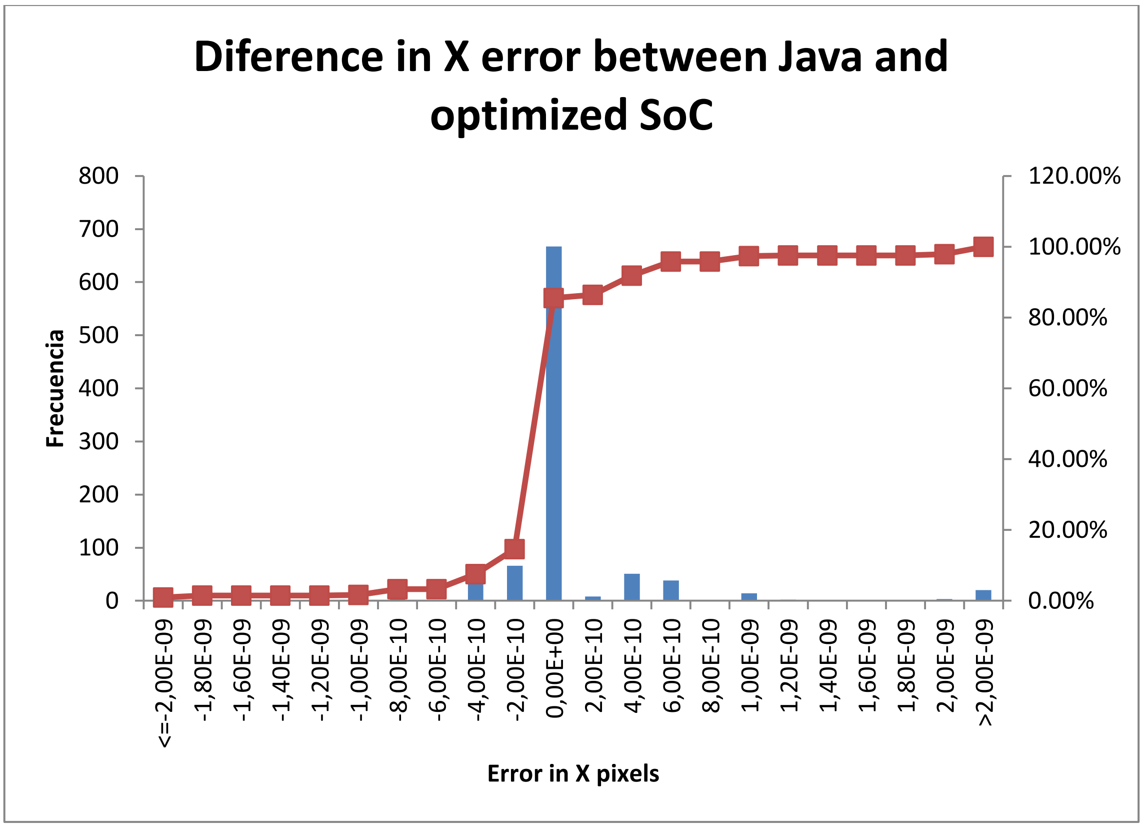
9.1. Precision results between implementations

There are three implementations whose results may be compared: Java implementation, simple coprocessor implementation and optimized coprocessor implementation.

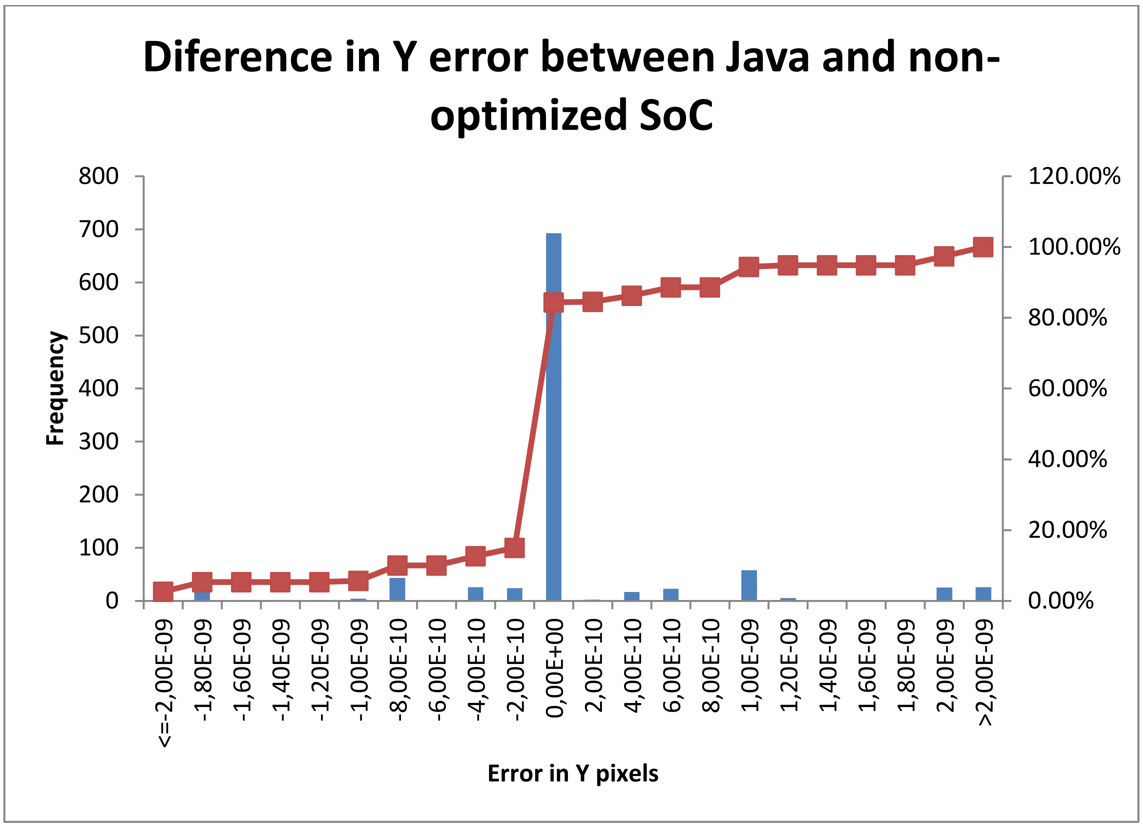
The following histograms show the absolute error in X pixels and Y pixels:



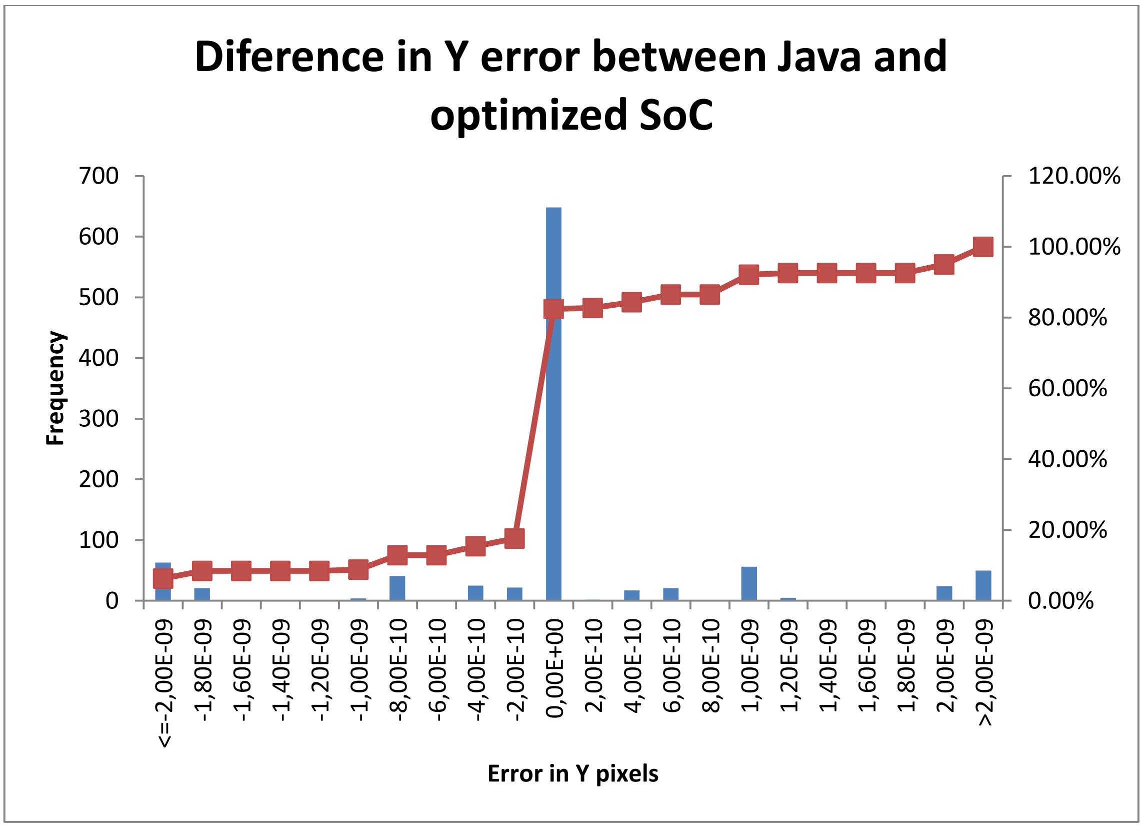
Histogram 4



Histogram 5



Histogram 6



Histogram 7

Not all the errors shown in the previous histograms are real errors. Due to the floating point representation for the output in different systems (Java and C) numbers that come from Java algorithm have 10 digits of precision, while results coming from C one have just 9 digits of precision. As the represented number is the subtraction of them both, results with an error smaller than 1e-9, or greater than -1e-9, are not real errors.

There are differences between the histograms due to different reasons:

Between Histogram 4 and Histogram 5: It is visible at the right end of the second histogram that there are some real errors that are not in the first one. The reason is that the optimized coprocessor is more parallel, then changing the operation order more compared to the Java implementation. This, as stated in [30] and [29], is a source of errors if the operated numbers have certain characteristics like, for example, being relatively small.

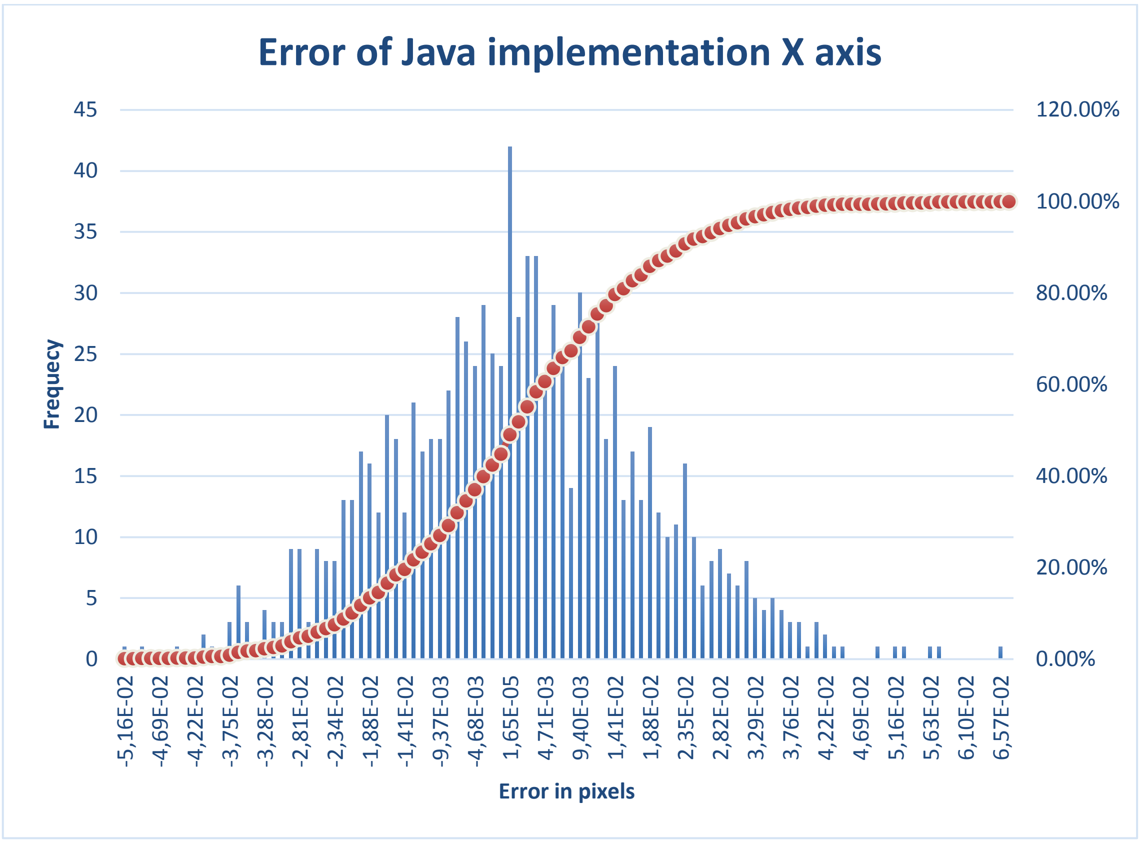
Between Histograms in X pixels, and histograms in Y pixels (Histogram 4 and Histogram 5; and Histogram 6 and Histogram 7): As it was settled in the 7.7 subsection, the algorithm is based on Gaia sensor pixel dimensions. The pixel size along Y axis is 3 times smaller than in the X axis. This leads to a better sampling in the Y axis. Then a circular PSF is represented with more pixels in the Y axis than in the X axis. As the number of pixels in Y axis is higher, the error tends to be higher too.

Between Histogram 6 and Histogram 7: The effect is just the same as between Histogram 4 and Histogram 5, but the difference is much more noticeable as a result of the effect described in the previous paragraph.

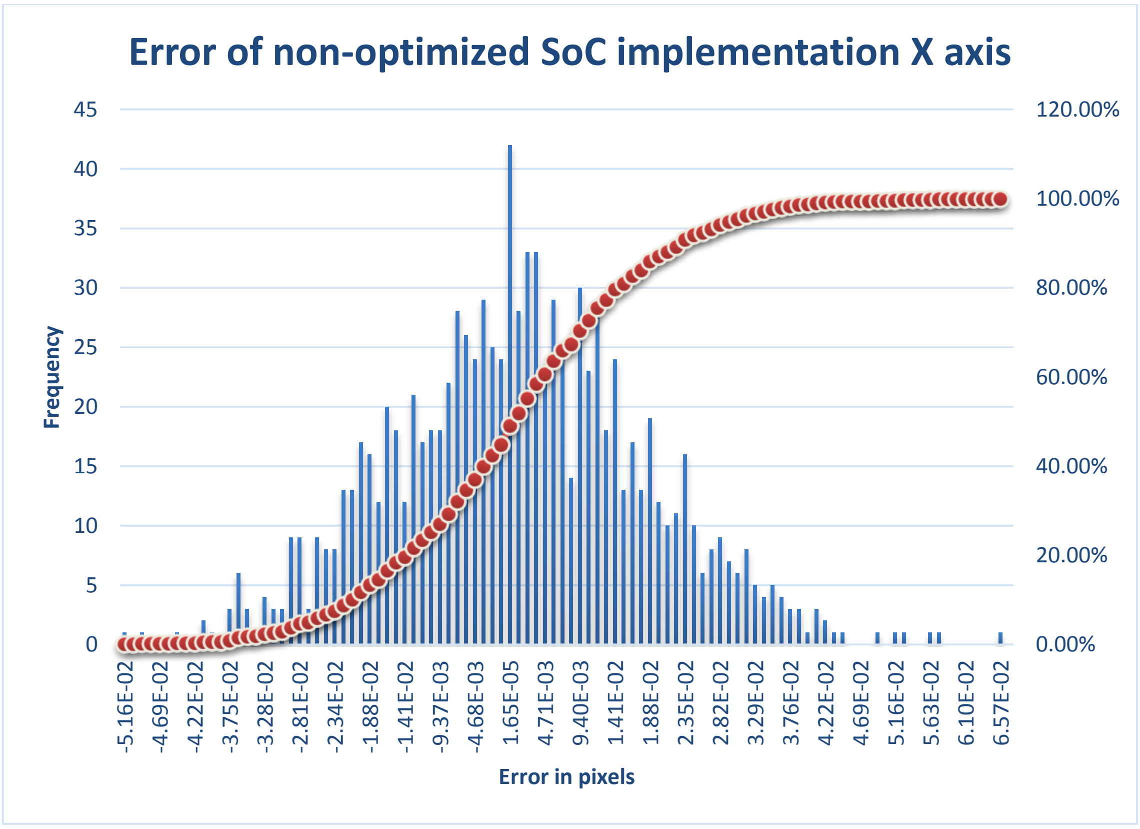
In the case of these two histograms it is especially important to know the magnitude of the errors that fall in the first and last bins of the histogram. The biggest difference between the Java implementation error (respect to the correct value known beforehand) and the optimized SoC error is 1.431e-6.

9.2. Precision of the algorithm

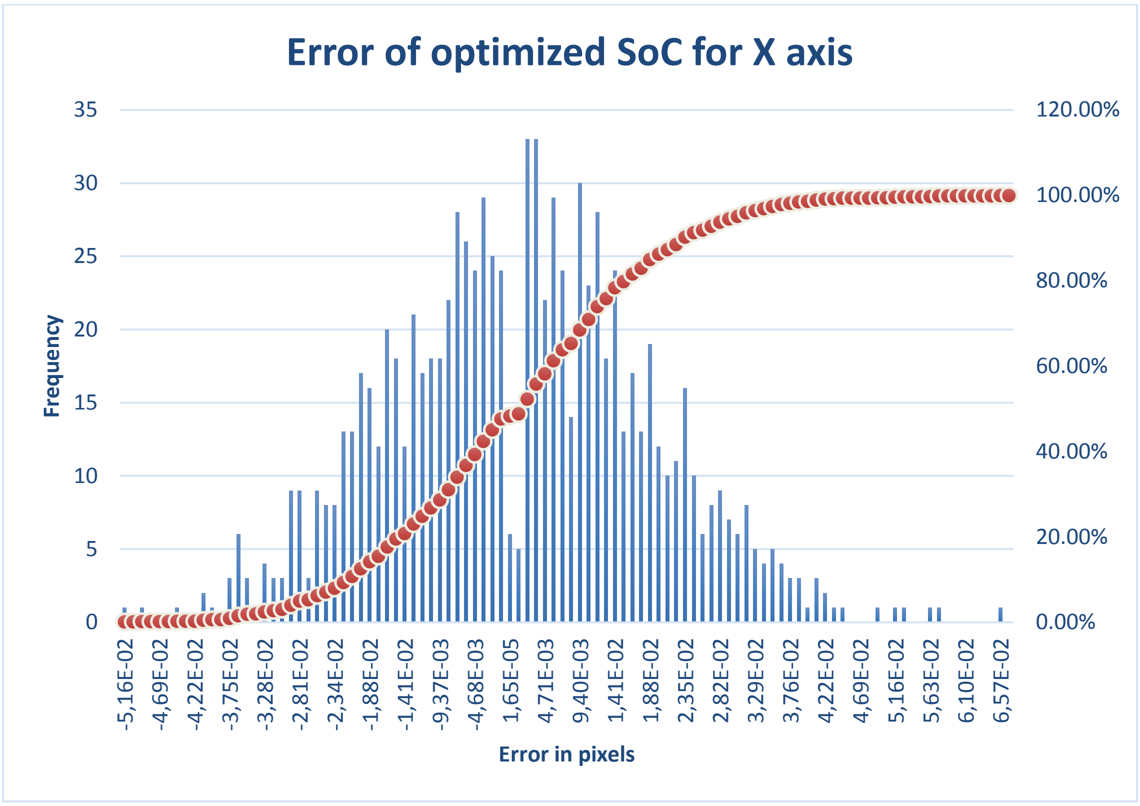
The following 6 histograms show the difference between the fit achieved by the different implementations of the program and the real parameters of each image fit. It represents the error of the algorithm, and its distribution over X and Y axis.



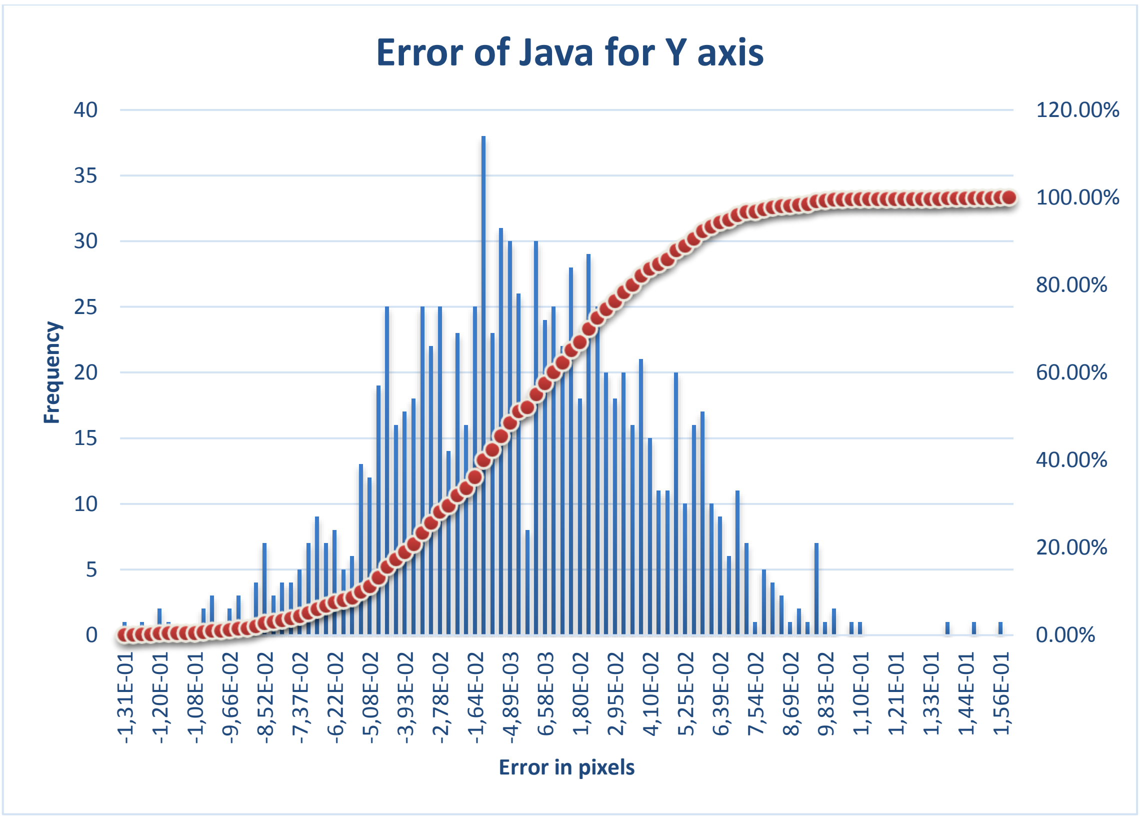
Histogram 8



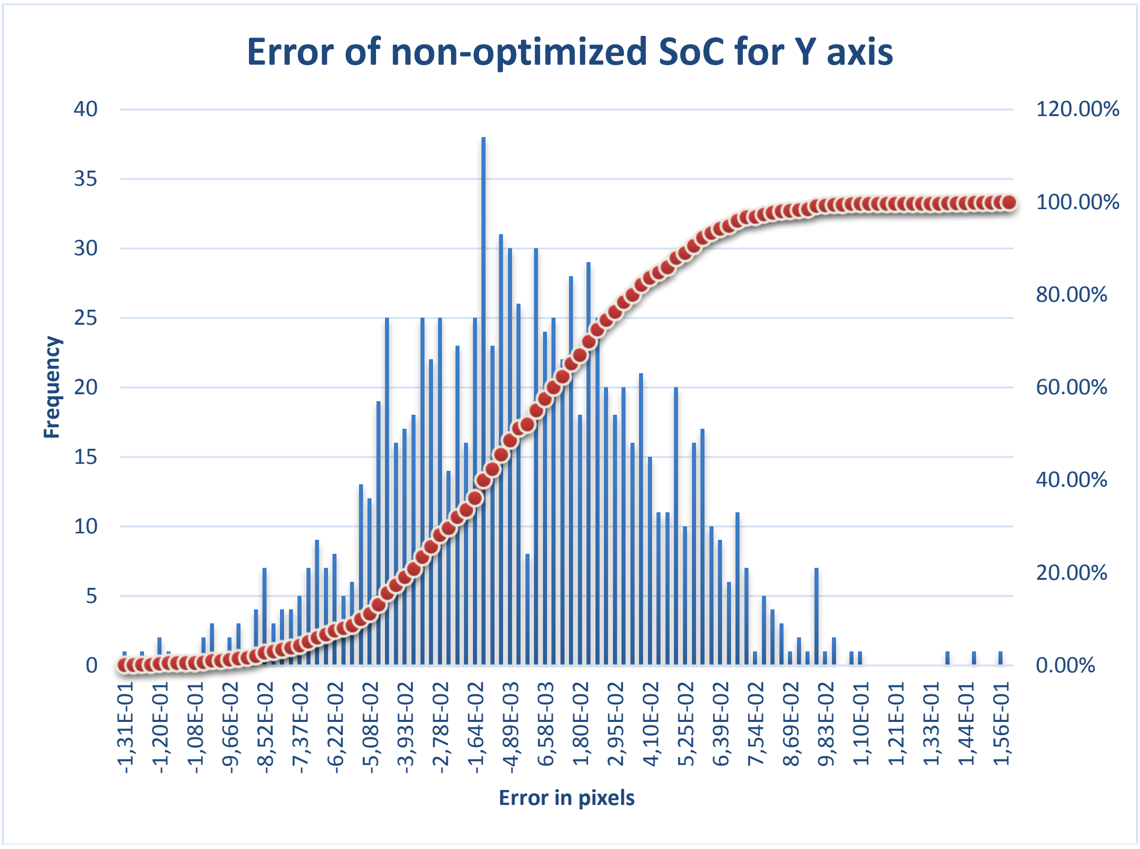
Histogram 9



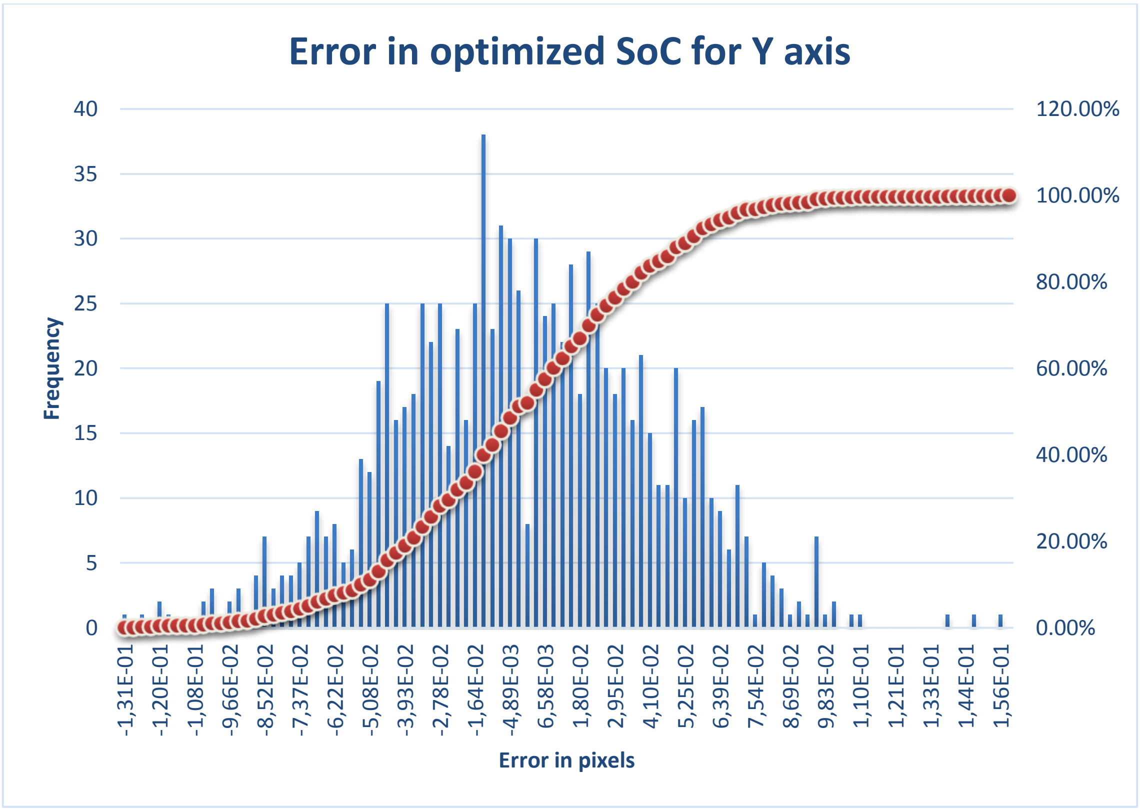
Histogram 10



Histogram 11



Histogram 12



Histogram 13

As all the error histograms resemble Gaussians, the mean and standard deviation do characterize the error. The Table 10 show their values:

Table 10: Mean and std. dev.

LUT41 with noisy images

	Java	Non-opt accel	Optimized accel
Mean of error in X pixels	0.00069	0.00069	0.00075
Std. Avg. of error in X pixels	0.01711	0.01711	0.01764
Mean of error in Y pixels	-0.00159	-0.00162	-0.00162
Std. Avg. of error in Y pixels	0.04219	0.04219	0.04258

9.3. Precision conclusions

The histograms above show that the error is 0 (or its equivalent due to representation precision) in nearly every case when the non-optimized core is used. This is because the order of the floating point operations is the same than in the Java implementation, so the possible errors when operating with small numbers are the same in both approximations. It also helps that after every iteration of the algorithm the result is quantized by retrieving the nearest matrix from memory. So, for example, if the result in "x" parameter at the end of one iterations of the algorithm is 5.26785 and there are only matrices whose "x" are 5.265 and 5.270, the matrix retrieved will be 5.270. Even if the result has a small error, and while the error is not big enough to change the matrix that will be retrieved, the selected matrix will be the same, so the initial values for the next iteration will remain unchanged.

In the case when the optimized accelerator is used the number of significant errors is not negligible. This happens because in order to optimize the multiplication and make use of the resources the FPGA has, a different operation order has been implemented. This is mainly because the use of the Block RAM dual port, which allows the transfer of two values at the same time, so each vector multiplication is divided into two vector multiplications with a consolidation step at the end of each half, in order to have the final result. This optimization is explained with more detail in the section 5.4 Additional coprocessor optimizations.

There is also a big difference between the errors in the X and Y direction. This difference is because of the physical size of pixels in Gaia sensor, which makes the PSF better sampled in the Y axis. As a result, the Y direction has more pixels than the X one, so the error is bigger comparatively.

10. Multicore study

In the subsection *8.2 Time conclusion* it is reasoned out that each iteration can only be optimized up to a point where all computational cost is hidden underneath data transfer cost, which leads to a poor 1.173x acceleration as a maximum.

Then the only way to achieve more performance would be to add more coprocessors until the matrices requests done over the DDR3 RAM saturate it. Thus in the following subsection it is calculated the maximum performance of optimal and suboptimal approaches to the multi coprocessor architecture.

10.1. Analysis from a suboptimal point of view: current architecture

The approximation that is going to be described in this subsection has the same current architecture, but with a multicore system goal.

The times achieved by the current architecture are:

Table 11: 98% confidence times for optimized accelerator

	Opt accel
Send C Matrix	307
Send B Matrix	327
Send A Matrix	626
Compute Time	308
Total Time with middle times	1651
Difference	83
Total Time w/o middle times	1478
Send C Matrix (strict)	139
Send B Matrix (strict)	160
Send A Matrix (strict)	459

The labels of each of the table times are explained below , in subsection Time tables.

In the current architecture it would only be needed to put two coprocessors. This would improve performance by only taking in account the memory transfer time, and not the computation time. A time schematic of this architecture is shown below in Figure 19.

The acceleration achieved over the monocore architecture would be 1.173x, dividing “total time without middle times” time and the sum of the matrix transferences.

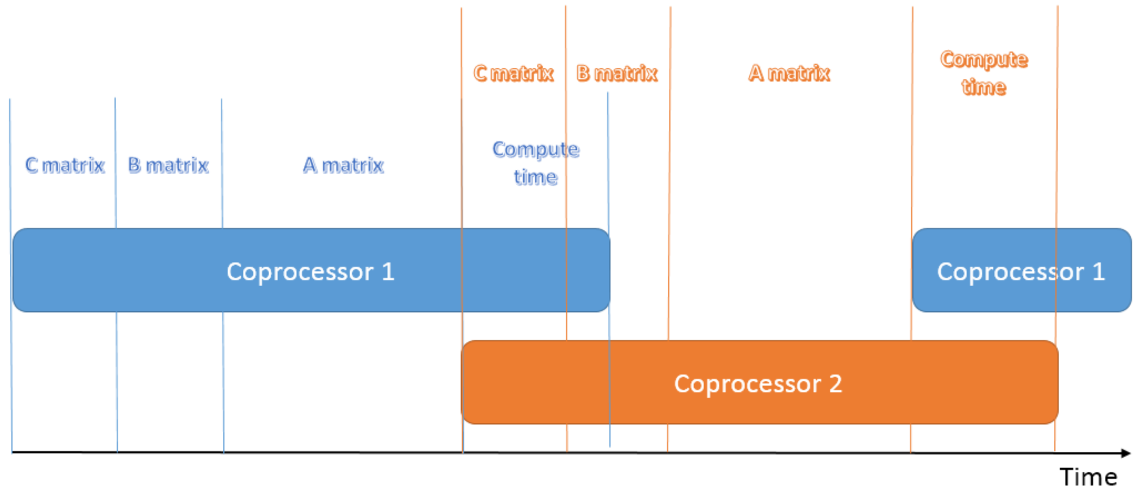


Figure 19: 2 coprocessors time schematic

10.2. Analysis from a more optimal point of view: changes in coprocessor and control

Two changes in the architecture would be able to improve the performance:

1. C matrix does not need to be sent in each iteration, can be at BRAM memory (inside FPGA) 5 out of 6 iterations, because it is the real image that the algorithm fits.
2. The computation time can be hidden behind the transfer time if a FIFO architecture is done. It would not be completely hidden, but the compute time which is not possible to hid would be a fraction of what it is currently.

Assuming that these two changes are done and in their best cases (C does not have to be sent and computation time is completely hidden) the necessary time to complete each iteration would be “Send B Matrix” plus “Send A Matrix”, which is 953 cycles of 10 ns. This time implies an acceleration of 1.55x from the current architecture.

10.3. Analysis from an optimal approach

In this subsection a fully optimal approach will be taken. The times used are completely theoretical some times.

In the section *Analysis from a more optimal point of view: changes in coprocessor and control* it is said that transfers take 953 cycles of 10 ns, which is 9.53 µs. During this time 500 simple precision floating point values are transferred, in other words, 2000 bytes. Then the binary rate at which these data are transferred is then:

$$\frac{2000\ bytes}{9.41\ \mu s} \times \frac{1\ MByte}{1024^2\ bytes} = 202.69\ MBytes/s$$

A DMA (Direct Memory Access) device like the one used have an optimal functioning limit of 1 4-byte data by cycle. Its cycle is the same as the one that drives the FPGA part of the SoC, as it is implemented inside the FPGA electronics (100 MHz), so it has a 10 ns period. So the theoretical binary rate limit for a DMA implemented in an FPGA with a clock of 100 MHz is:

$$\frac{4\ bytes}{10\ ns} \times \frac{1\ MByte}{1024^2\ bytes} = 381.47\ MBytes/s$$

Despite of this, the Look-up table (LUT) is stored in the DDR3 RAM module installed in the Zedboard, which are two Micron MT41K128M16HA-15E:D DDR3 memory components. In the 6th page of the Zedboard Hardware user guide [23] it specifies that these modules are configured with a 32-bit interface and a speed up to 533 MHz (although the manufacturer of the component specifies that these modules can reach 666.67 MHz in [31]). This means that 1066 MT/s can be reached on the Zedboard, each one of 32 bits. The resultant speed is:

$$1066 \frac{MTransferences}{s} \times 32\ bits \times \frac{1\ byte}{4\ bits} = 8528\ MBytes/s$$

This means that if the theoretical top binary rate of the DDR3 RAM would be reached, it could serve data for up to 22 coprocessors with optimal DMAs, or 42 coprocessors with the DMAs that are currently implemented.

However, this point would further increase the complexity of the system, because all these coprocessors and DMAs would have to be commanded from the main CPU, and an arbiter would have to be added in order to manage the access to the RAM limiting the possible conflicts over its use. It is also impossible to fit 20 to 40 coprocessors on this platform from a resources point of view (this point was analysed in section Resources used).

11. Conclusion and Future work

In what follow we summarize main contributions and achievements of this thesis, likewise we add some possible and ongoing future works.

11.1. Conclusion

Two of the LUTs have been produced are considered excellent in terms of the algorithm performance. The first one is the 101 nodes LUT. It provides the best performance and the smallest the error, both in average and average deviation. The second one is the asymmetrical LUT, which it reaches the performance of the 101-node LUT while being 60% smaller, a very valuable asset in the embedded system. It is not considered the best in general performance because the average error is bigger compared to any of the other LUTs. This is an undesirable effect, and the origin is currently unknown, which is a reason to continue using symmetrical LUTs.

Another conclusion is that the diameter variable prevents the algorithm to reach convergence when no interpolation between the LUTs is done. It could be due to a greater variability than expected. However, once the diameter dimension is not taken into account the algorithm precision improves with larger LUTs, although from a certain size the improvement is negligible. The improvement becomes negligible with even smaller LUTs if there is noise in the images, so the amount of noise limits the algorithm precision.

The main bottleneck for the performance of this algorithm is the access to the LUT in RAM memory, for both the SoC with the FPGA and the Java software implementation, for one fit up to 6 matrices have to be read.

A system which would process several lenslets with the current architecture would also be very inefficient, because with the real times provided by the single lenslet system, even hiding the operation time with the transference time, the increase in performance would be rather small.

A change in the architecture to solve this problem is out of the scope of the project due to its complexity and the time required to design and set up the system. Despite of this, even with an optimal multi-lenslet architecture only 8 or less coprocessors could be placed in the current platform, due to the resources needed to allocate them.

The precision of the non-optimized FPGA implementation has proved to be nearly as good as the Java one running on a high performance desktop computer. With the optimized coprocessor the numerical precision achieved is worse than in the other implementations, but still delivers perfectly acceptable results for the purposes of this algorithm.

11.2. Future work

As it has been proposed in the section *10 Multicore study* the next step would be to take advantage of the full area of the FPGA platform using as many coprocessors as it is possible to run in parallel, saturating either the RAM throughput or the area of the FPGA. For the Zedboard platform it has been calculated in the section 5.6 *Resources used* it would take 8 coprocessors to use all the resources of the FPGA.

However, this system would require a change in the architecture, in order to be able to use the RAM memory more efficiently. This architecture would still use the same coprocessor as it has proved to be precise enough for the requirements of the algorithm, but the control system would have to change.

12. References

[1] R. Martín, “Aceleración de algoritmos en óptica adaptativa usando FPGA,” *Proyecto Fin de Carrera, EPS-UAM*, 2014. Available: https://repositorio.uam.es.

[2] J. Beckers, “Adaptive Optics for Astronomy: Principles, Performance and Applications,” *Annual Review of Astronomy and Astrophysics,* pp. 13-62, 1993.

[3] U. Bastian, “The maximum reachable astrometric precision - The Cramer-Rao Limit,” Gaia DPAC public document, 2004.

[4] K. A. Winick, “Cramer–Rao lower bounds on the performance of charge-coupled-device optical position estimators,” *Journal of the Optical Society of America,* pp. 1809-18015, 1986.

[5] L. Lindegren, “A general Maximum-Likelihood algorithm for model fitting to CCD sample data,” Gaia DPAC public document, 2008.

[6] B. A. Draper, J. R. Beveridge, A. W. Bohm, C. Ross and M. Chawathe, “Accelerated image processing on FPGAs,” *IEEE transactions on image processing,* vol. 12, no. 12, pp. 1543-1551, 2003.

[7] E. Fernandez and P. Artal, “Membrane deformable mirror for adaptive optics: performance limits in visual optics,” *Optics express,* vol. 11(9), pp. 1056-1069, 2003.

[8] E. Masciadri and T. Garfias, “Wavefront coherence time seasonal variability and forecasting at the San Pedro Mártir site,” *Astronomy & Astrophysics,* pp. 708-716, 2001.

[9] B. C. Platt and R. Shack, “History and principles of Shack-Hartmann wavefront sensing. Journal of Refractive Surgery,” *Journal of Refractive Surgery,* vol. 17(5), pp. 573-577, 2001.

[10] S. Thomas, T. Fusco, A. Tokovinin, M. Nicolle, V. Michau and G. Rousset, “Comparison of centroid computation algorithms in a Shack-Hartmann sensor,” *Monthly Notices of the Royal Astronomical Society,* pp. 323-336, 2006.

[11] A. Björk, Numerical methods for least squares problems, Philadelphia: SIAM, 1996.

[12] J. J. Moré, “The Levenberg-Marquardt algorithm: implementation and theory,” in *Numerial Analysis Biennial Conference*, 1978.

[13] A. Mora, “WFS sub-pixel centroiding,” Internal report, 2010.

[14] Xilinx Inc. , “UG470 7 Series FPGAs Overview,” 2015. [Online]. Available: https://www.xilinx.com/support/documentation. [Accessed 2016].

[15] Xilinx Inc. , “UltraScale Architecture and Product Overview,” 2016. [Online]. Available: http://www.xilinx.com/support/documentation/. [Accessed 2016].

[16] Xilinx Inc. , “Zynq-7000 All Programmable SoC Overview,” 2016. [Online]. Available: http://www.xilinx.com/support/documentation/. [Accessed 2016].

[17] Xilinx Inc. , “UltraScale Architecture and Product Overview,” 2016. [Online]. Available: http://www.xilinx.com/support/documentation/. [Accessed 2016].

[18] Altera, “Stratix 10 Device Overview,” 2015. [Online]. Available: https://www.altera.com/en_US/pdfs/literature. [Accessed 2016].

[19] Xilinx Inc., “UG902 Vivado Design Suite User Guide,” 2014. [Online]. Available: http://www.xilinx.com/support/documentation. [Accessed 2016].

[20] Xilinx, Inc, “UG534 ML605 Hardware User Guide,” 2011. [Online]. Available: http://www.xilinx.com/support/documentation. [Accessed 2016].

[21] Xilinx Inc., “UG081 Microblaze Processor Reference Guide,” 2012. [Online]. Available: http://www.xilinx.com/support/documentation. [Accessed 2016].

[22] Xilinx Inc. , “UG1037 Vivado Design Suite: AXI Reference Guide,” 2015. [Online]. Available: http://www.xilix.com/support/documentation. [Accessed 2016].

[23] AVNET, “Zedboard Hardware User's Guide,” 2014. [Online]. Available: zedboard.com/support/documentation.

[24] Xilinx Inc., “DS180 7 Series FPGAs Overview,” 2014. [Online]. Available: http://www.xilinx.com/support/documentation/. [Accessed 2016].

[25] J. Rose, “Hard vs. soft: The central question of pre-fabricated silicon. In Multiple-Valued Logic,” in *Proceedings. 34th International Symposium on (pp. 2-5). IEEE.*, 2004.

[26] A. Mora, “FPGA image centroiding” Internal report, 2012.

[27] J. H. J. De Bruijne, “Science performance of Gaia, ESA’s space-astrometry mission,” *Astrophysics and Space Science,* vol. 341(1), pp. 31-41, 2012.

[28] P. Nussbaum, R. Voelkel, H. P. Herzig, M. Eisner and S. Haselbeck, “Design, fabrication and testing of microlens arrays for sensors and microsystems,” *Pure and applied optics: Journal of the European optical society,* vol. 6(6), p. 617, 1997.

[29] D. Goldberg, “Waht every computer scientist should know about floating-point arithmetic,” *ACM computing surveys,* vol. 23, no. 1, p. 195, 1995.

[30] N. Shirazi, A. Walters and P. Athanas, “Quantitative analysis of floating point arithmetic on FPGA based custom computing machines,” in *FPGAs for Custom Computing Machines, Proceedings IEEE symposium*, 1995.

[31] Micron, “2Gb: x4, x8, x16 DDR3L SDRAM Description. DDR3L SDRAM MT41K512M4 – 64 Meg x 4 x 8 banks MT41K256M8 – 32 Meg x 8 x 8 banks MT41K128M16 – 16 Meg x 16 x 8 banks,” 2014. [Online]. Available: https://www.micron.com. [Accessed 2016].

[32] Berkeley Design Technology Inc., “An independent evaluation of: High Level Synthesis Tools for Xilinx FPGAs,” 2010. [Online]. Available: xilinx.com/technology/dsp/BDTI_techpaper.pdf.

[33] R. Davies and M. Kasper, “Adaptive Optics for Astronomy,” *Annual Review of Astronomy and Astrophysics,* pp. 305-351, 2012.

[34] D. C. Giancoli, “Physics for Scientists and Engineers,” Prentice-Hall, 2000, p. 896.

[35] Space Telescope Science Institute, “James Webb Space Telescope Near infrared Camera PSFs,” [Online]. Available: http://www.stsci.edu/jwst/instruments/nircam/PSFs/. [Accessed 2016].

[36] C. T. Kelley, Iterative methods for optimization, Siam, 1999.

[37] A. Mora and A. Vosteen, “Gaia in-orbit realignement. Overview and data analysis,” in *SPIE Astronomical Telescopes*, 2012.

[38] A. Björck, Numerical methods for least squares problems, Siam, 1996.

[39] S. Thomas, T. Fusco, A. Tokovinin, M. Nicolle, V. Michau and G. Rousset, “Comparison of centroid computation algorithms in a Shack–Hartmann sensor,” *Monthly Notices of the Royal Astronomical Society,* pp. 323-336, 2006.