UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

PROYECTO DE FIN DE CARRERA

FPGA prototype for adaptive optics algorithm acceleration

Ingeniería de telecomunicación

Raúl Martín Lesma

Mayo, 2014

FPGA prototype for adaptive optics algorithm 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

Mayo 2014

Resumen

La óptica adaptativa es una técnica usada principalmente para mejorar el rendimiento de los sistemas ópticos (como por ejemplo los telescopios astronómicos) reduciendo el efecto de las perturbaciones en el frente de ondas introducidas por la atmósfera. Los sensores de onda más comunes (Shack-Hartmann) producen una miríada de imágenes cuyo centroide hay que determinar para estimar las aberraciones. Como el tiempo de coherencia de la atmósfera está en torno al milisegundo, cada centroide debe ser estimado en el entorno de los microsegundos para que el sistema pueda dar una respuesta a tiempo. Debido a estas restricciones los algoritmos de centrado que se usan habitualmente son de baja precisión.

Los algoritmos de centrado de máxima verosimilitud desarrollados para la misión *Gaia* de la ESA alcanzan una precisión muy cercana a la máxima posible matemáticamente hablando, la frontera de Crámer-Rao. A pesar del alto rendimiento, el algoritmo usa intensivamente cálculos en coma flotante, así que normalmente no alcanza las restricciones en el tiempo que aplican en óptica adaptativa.

Para solucionar este problema una versión básica del algoritmo ha sido implementada en un sistema embebido basado en FPGA. Un módulo específico ha sido desarrollado con este propósito usando la herramienta de síntesis de alto nivel Vivado HLS. Para cumplir los requisitos temporales el algoritmo ha sido estructurado en un pipeline y paralelizado hasta un límite razonable.

Aunque implementar la versión completa del algoritmo requerirá más desarrollo, los resultados alcanzados por esta primera aproximación son muy prometedores y refuerzan la relevancia de los sistemas basados en FPGA para aplicaciones astronómicas.

Abstract

Adaptive optics is a technique used mainly to improve the performance of optical systems (as astronomical telescopes) by reducing the effect of wavefront distortions introduced by the atmosphere. The most common wavefront sensors (Shack-Hartmann) produce a myriad of images which centroid has to be determined to estimate the aberrations. As the coherence tome of the atmosphere is around the millisecond, each centroid has to be estimated in a time around the microseconds, so the system is able to give an output in time. Due to these requirements the centroiding algorithms most commonly used achieve low precision.

The maximum likelihood centroiding algorithm developed for the ESA Gaia mission provides a precision very close to the maximum mathematically achievable, which is the Crámer-Rao lower bound. Despite of its high performance, the algorithm is intensive in floating point calculation, so it is typically far from the strict time constraints.

To solve this problem a basic version of the algorithm has been implemented in a FPGA based embedded system. A specific core has been developed for this purpose using the high level synthesis tool Vivado HLS. To meet the time constraints the algorithm has been parallelized and pipelined up to a reasonable degree.

Although implementing the complete version of the algorithm will require further development, the results shown by this first approach are very promising and strengthen the relevance of FPGA based systems for astronomical applications.

Table of contents

Table of figures

Glossary

AMBA	Advanced Microcontroller Bus Architecture
AO	Adaptive Optics
ASIC	Application-specific Integrated Circuit
BPI	Byte Peripheral Interface
BRAM	Block RAM
CCD	Charged-Coupled Device
CPU	Central Processing Unit
DQPSK	Differentially encoded Quadrature Phase-shift Keying
DSP	Digital Signal Processor
ESA	European Space Agency
ESAC	European Space Astronomy Centre
FADD	Floating Point Addition
FF	Flip-flop
FIFO	First In, First Out
FIG	Figure
FMC	FPGA Mezzanine Card
FMUL	Floating Point Multiplication
FPGA	Field Programmable Gate Array
GPU	Graphics Processor Unit
HDL	Hardware Description Language
HLS	High Level Synthesis
HLST	High Level Synthesis Tools
HPC	High Performance Connection
HW	Hardware
JTAG	Joint Test Action Group
LPC	Low Performance Connection
LUT	Look-up Table
PSF	Point Spread Function
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
RMS	Root Mean Square
RTL	Register Transfer Level (Language)

SW	Software
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
VHLS	Vivado High Level Synthesis
WF	Wavefront
WFS	Wavefront sensor
XPS	Xilinx Platform Studio

1. Introduction

This Project has been carried out within the framework of the European Space Astronomy Centre (ESAC) trainee program, which gives the opportunity to contribute to current missions to young student scientists and engineers. This project covers the design and implementation of a FPGA prototype, which has the purpose of accelerating a centroiding algorithm. It will be evaluated if FPGAs are adequate devices to implement adaptive optics algorithms, given the typical time constraints.

1.1. Motivation and objectives

Several astronomical applications require obtaining and analysing the centroid of images with a very high duty cycle. Two examples are adaptive optics and attitude and orbit control systems. The speed requirements can be very stringent (µs for typical adaptive optics applications), and inefficient but fast algorithms are typically used, such as the image centre of gravity.

The maximum precision that any algorithm can achieve is given by the Crámer-Rao lower bound [BAS04]. The maximum likelihood algorithms developed for the ESA Gaia mission [LIN08] provide a precision very close to the Crámer-Rao limit, and can be considered optimal in terms of performance. They are based on forward modelling: the weighted Gauss-Newton optimization of a function resembling the observed data. However, they are time consuming, and typically far from the speed requirements needed for time critical applications.

FPGA based devices have been intensively used in the past in satellites and other ESA missions to process data on-board, and therefore to reduce the amount of data sent to Earth. They have been proven to be useful in several fields in the space industry, and accelerating heavy centroiding algorithms could be one of its uses.

Although not within the scope of this thesis, an advanced version of the prototype that will result of this project would be connected in some way to a

Shack-Hartmann wavefront sensor, receiving the data that it outputs. Then it would reconstruct the wavefront for a potential restoration using deformable mirrors.

The objectives of this project are listed next:

- Design and implementation of a specific core that performs the mentioned centroiding calculation algorithm.
- Optimization of the core that implements the centroiding calculation algorithm, adapting the hardware to the needs of the algorithm.
- Testing of core functionality.
- Design and implementation of a system that provides the core with data and general functionality.
- Optimization of the system, to approach the 1 μs time scale.
- Evaluation of the possibilities of a similar system to process in parallel several tens of lenslets, most likely accommodating an equal number of accelerating cores.
- Evaluation of the suitability of FPGA system for this type of algorithm and time constraints.

1.2. Document description

After the introduction, section ‘2. State of the Art of Adaptive Optics’, explains the purpose of adaptive optics, the problem that this project issues, several clarification about the technology involved, and finally the implemented algorithm. Section ‘3. The use of Reconfigurable HW for adaptative optics’ explains why this technology could be relevant for adaptive optics, details the features of the evaluation board used in the project and also includes some considerations to understand the design decisions. In Section ‘4. First functional prototype’ it is explained how was carried out the prototype itself, including both the computing core and the general system. Sections 5 and 6 present the results, the future work and the conclusions of the project.

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 works measuring the distortion and compensating for it with some device, usually a deformable mirror. AO systems are used in several fields, including ocular surgery and astronomical observation:

- **Ocular medicine and surgery**: AO enable doctors to see the eye internal structure with precision due to its capacity to eliminate the eye aberration. It is also useful to measure the exact amount of correction needed when surgery is performed or even to model the results of it.

- **Astronomical observation**: Before light reaches the focal plane of a ground based telescope it has been distorted by the atmosphere. The way the atmosphere is distorting the light is also changing continuously. Due to this effect the only way to achieve ground based diffraction limited imaging is to include an AO system integrated in the telescope.

Adaptive optics in astronomy has been proved very useful in achieving more accurate images. As can be seen in Fig. 1 the pictures obtained with an AO system are much sharper and clear than the ones in which no AO system has been implemented.

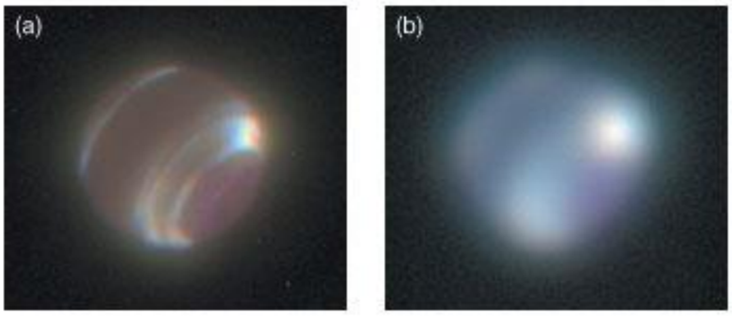


Fig. 1: Left, Neptune picture taken with AO; right, same picture without AO.

2.2. Problem in adaptive optics that is being solved

One of the main problems of AO is that the system is required to work in real-time, which in this case means that the whole process must be finish in a time scale of 1 ms. This time scale is called coherence time and depends on several parameters, as the Fried parameter and the average wind speed [DAV12]. So every step in a WF calculating algorithm should be completed in less than 1 ms. In particular it means that each of the around 1000 centroiding operation must be carried out in the microsecond time range, in order to provide timely feedback.

Why is this requirement a problem? Because the algorithm that has been chosen is particularly intensive in calculations. It involves several stages, each one of these including hundreds of floating point multiplications, hundreds of floating point sums and subtractions, a big amount of accesses to memory, etc.

As you may have noticed, there are lots of operations with floating point data, which is a known field in which general purpose processors struggle. Due to its characteristics (they include just a few very complex cores) they excel at performing complex tasks over a small set of fixed precision data, whereas they have significantly less performance working with large sets of floating point data which usually require a great number of operations [UND04].

Given the real-time constraint it is obvious that a general purpose processor (CPU) is not the best suited hardware for this task. Furthermore, as can be read in the next subsection, the used algorithm can be easily parallelized, so it is clear that a CPU is not suitable for this job.

2.3. Wavefront sensor

A wavefront sensor is an optical instrument that measures the deformation of a WF. In other words it calculates the phase divergence between the different parts of the WF. In order to do this it is composed of a lenslet array in which each one focuses the light that hits the corresponding fraction of the WFS in one point on a light gathering element. As the lenslet array will be set in a square regular pattern, the same shape is observed in the images, collected in the CCD detector located afterwards.

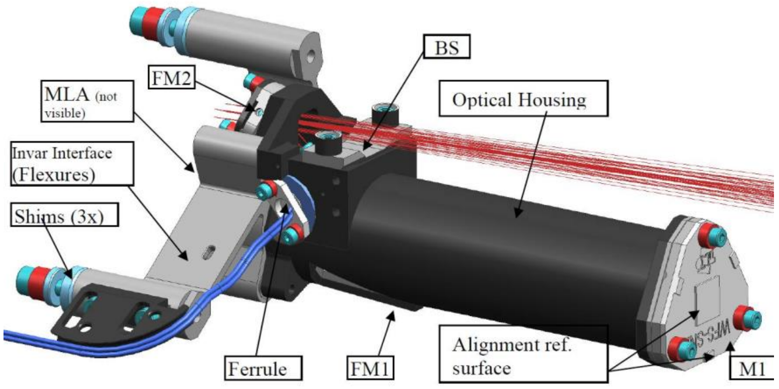


Fig. 2: Wavefront sensor included in Gaia (exterior), from [MOR12b]

As can be seen in Fig. 3, when a flat wavefront hits the lenslet array it produces a certain pattern of light dots, usually in a squared pattern (in this case it is set in a linear pattern since it is observed from one of its sides). If the wavefront is not in phase this pattern will be distorted, the distances between the light dots will not be the same. In this way the phase differences can be calculated from this ‘error’ in the position of the light point. The bigger the distance between the correct position and the real position, the less in phase is the wavefront.

The WFS provides an image composed of different subimages (one per microlens). The centroid of each subimage is derived fitting the electron distribution recorder in the CCD image.

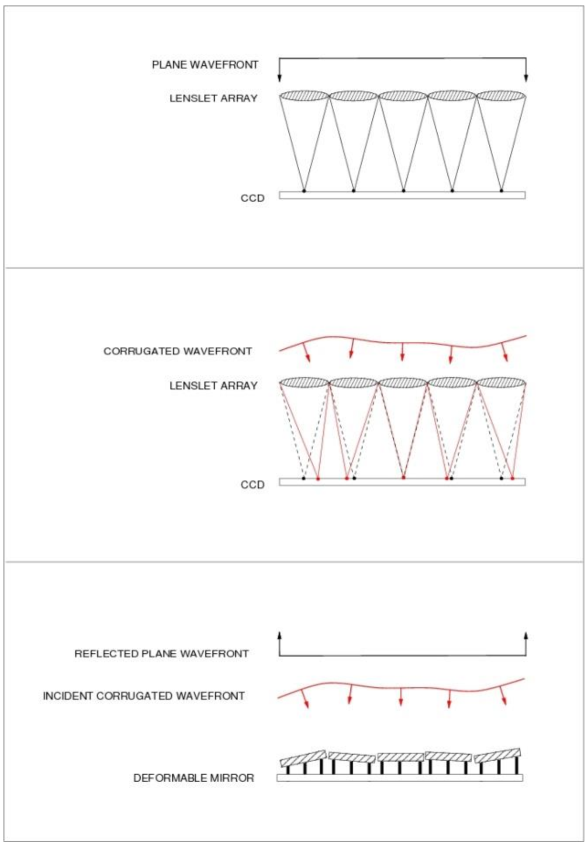


Fig. 3: WF schematic. Up, it shows a flat WF and the regular pattern that it produces; middle, an aberrated WF results in an irregular pattern; bottom, a deformable mirror is able to reconstruct a flat WF.

2.4. Purpose of Adaptive Optics algorithm: Centroids, Crámer-Rao lower bound and maximum likelihood algorithm

Although Shack-Hartmann wavefront sensors provide all the data which is necessary to determine the WF slopes, there has to be an algorithm that transforms all this information into a real wavefront. This process is divided in two differentiated parts:

1. **Determining the position of each microlens image:** Given the output of the WFS, the centroid of the point spread function (PSF) can be determined. The more precision the system achieves, the better it will work.

2. **Wavefront reconstruction from centroid positions:** Once all centroids have been determined the wavefront can be reconstructed.

Determining the centroids is not easy. Due to the nature of the light, and because of its diffractive properties, each “light point” mentioned in the previous subsection is not exactly a point. Instead of this it would be more similar to an Airy disk [GIA00] (fig. 4), if the optical system would be perfect.

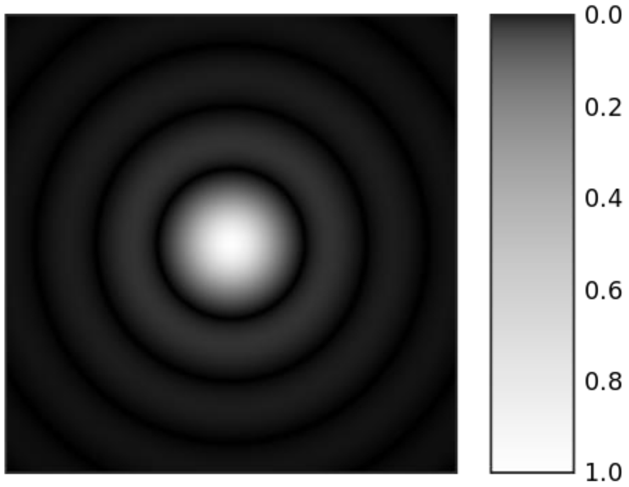
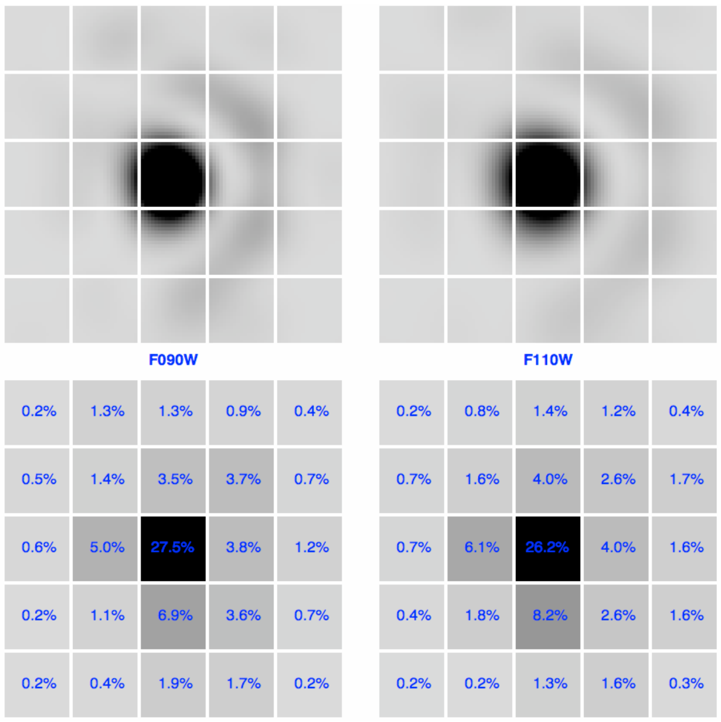


Fig. 4: Airy disk produced by refraction of the light [SAK07]

In reality there are small aberrations in the lenses, so this Airy disk will be deformed. This distorted Airy disk can be expressed by the number of electrons that hit the sensor multiplied by a matrix of the probabilities of an electron to hit a specific pixel. This matrix of probabilities is called point spread function (PSF, fig. 5), and its centroid is the same that the original deformed Airy disk had as its values are proportional to the initial ones. Because of this the coordinates of the subimage that are needed are also the coordinates of the centroid of the PSF.



F090W

0.2%	1.3%	1.3%	0.9%	0.4%
0.5%	1.4%	3.5%	3.7%	0.7%
0.6%	5.0%	27.5%	3.8%	1.2%
0.2%	1.1%	6.9%	3.6%	0.7%
0.2%	0.4%	1.9%	1.7%	0.2%

F110W

0.2%	0.8%	1.4%	1.2%	0.4%
0.7%	1.6%	4.0%	2.6%	1.7%
0.7%	6.1%	26.2%	4.0%	1.6%
0.4%	1.8%	8.2%	2.6%	1.6%
0.2%	0.2%	1.3%	1.6%	0.3%

Fig. 5: A pair of deformed Airy disks produced by a WFS lenslet, and its correspondent PSF, in a pixel matrix of 5x5. [JAM].

So a Shack-Hartmann Wavefront Sensor (WFS) provides information on the on the wavefront slopes measuring the centroid displacement of each lenslet image with respect to a given zero error reference. This process completely depends on the centroiding precision achievable. There is a maximum precision achievable for a centroiding algorithm: the Crámer-Rao lower bound.

Reaching this precision is not trivial, but it can be done fitting the coordinates of the centroid, and some other nuisance variables, with a mathematical model that depends on several variables. In this case this secondary data will be the number of electrons that hit the sensor and a shape factor of the PSF. This shape factor encapsulates information relative to effects such as PSF width, microlens diameter, etc.

In this way the initial centroiding problem has been transformed to a weighted least square minimization problem. There are many methods available to solve this type of problem, such as Gauss-Newton [BJO96] or Levenberg-Marquardt ones [MOR78].

2.5. Problems in SW solution, options to solve it

The PSF centroiding algorithm could be programmed in standard pc with a normal CPU, and run correctly, but it will not satisfy the time requirements. As it was stated in previous sections, for this algorithm to be effective it is necessary that all the computation is done in a time-scale of a few hundreds of µs, which is difficult to achieve with the most usual techniques.

The algorithm exposed is highly parallelizable, mainly in two ways:

There are lots of floating point matrix multiplications. This means that hundreds of floating point multiplications could run in parallel in order to be summed up after they are all finished.

In a standard Shack-Hartmann wavefront sensor there are between dozens and hundreds of lenslets. Each one of these PSF centroids need to be calculated with an algorithm like the one proposed before. So a solution to meet time requirements would be to process every lenslet at the same time.

In order to implement these two basic ideas it seems clear that a standard CPU program configuration will not do the job. Whereas the CPU solution is not an option, two other ones arise:

- GPU (graphic processor unit): Whereas a standard processor only has a few cores, a GPU usually includes from several hundreds to thousands. This is useful because each one of those can deliver an operation at the same time. Furthermore, they are specialized in floating point operations.

- FPGA (field programmable gate array): Due to its ability to link logical gates and create both logic and memory, this is one of the most flexible options to perform whatever type of algorithm. It is possible to create any hardware inside of an FPGA, so floating point operational hardware could be replicated as many times as necessary, while there still are resources available.

2.6. Adaptive Optics algorithm

The algorithm used in this project is based in a maximum likelihood algorithm developed for the ESA Gaia mission, and described in deep in [MOR10]. 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.

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, and then providing the best match between N_i and O_i.

$$RMS = \sum_{i=0}^{n_x n_y} w_i (N_i - O_i)^2$$

In this equation w_i are the weights that provide the maximum likelihood [MOR12a].

The RMS optimization algorithm that is used is the Gauss-Newton algorithm. It is an iterative method that uses an initial input for the vector $\boldsymbol{x}$ and approaches the optimal vector by summing the result values of the procedure ($\Delta\boldsymbol{x}$) to the previous vector. In each iteration $\Delta\boldsymbol{x}$ is calculated according to:

$$\Delta\boldsymbol{x} = (J^T W J)^{-1} J^T W (O - N) = M(O - N)$$

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 (or image), W is a diagonal matrix with the weights and J is the forward model Jacobian matrix.

The speed of the Gauss-Newton method depends on the speed with which the matrix M and the vector N can be computed for each iteration. This is an important point because the centroid has to be calculated within a time scale of 1 µs. 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 center 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.

The LUT has to be indexed by the four parameters of the vector $\boldsymbol{x}$, and it needs enough nodes to achieve the precision noted above (Crámer-Rao lower bound). The following number of nodes has been proposed in [MOR12a]: 21 elements for each variable (x_c, y_c, s) and 20 for N_T. It results in $21 \times 21 \times 21 \times 20 = 185220$ nodes. Every node requires one matrix M and one matrix N, this is 500 elements in total. Each element will be stored in floating point single precision format (4 bytes). This makes a total of:

$$21 \times 21 \times 21 \times 20 \times 500 \times 4 = \ 370.44\ MB$$

The knots of every variable are distributed in the segment where is most likely to have values in a real scenario. The points for the different parameters are:

- x_c and y_c: 21 points distributed from the centre of the image (in this case 4.5 in both axes) in both directions, from the start of pixel 4 to the start of pixel 6.
- s: 21 knots covering a reasonable interval of values (usually from 350 to 450 μm).
- N: 20 points from 500 to 10000 electrons, in steps of 500.

Although this is the size of the LUT that is needed to achieve an adequate precision, it was proposed to start creating a prototype that used a smaller LUT due to the problems of allocating and accessing large data arrays in an external memory from an FPGA. This smaller LUT was made with just 81 nodes (3 for each parameter), which makes a file size of 162 KB. All the work from now on will focus on this approach.

3. The use of reconfigurable HW for adaptive optics

3.1. Benefits of reconfigurable hardware and FPGAs

Reconfigurable hardware is a valuable option when very restricting time constraints exist in the problem to solve. This is due to the capacity of modelling specific devices which are perfectly adapted to the algorithm to perform, and that are not useful in any other one. Both FPGA and ASIC are devices that provides these capabilities.

Since many years reconfigurable hardware (FPGA) is a growing alternative to the classic ASIC (application-specific integrated circuit) approach to custom application hardware chips. Nowadays there are powerful several million logic cells FPGA, that are a cheap alternative to ASIC.

These are some of the most important benefit of FPGA technology:

- **Performance**: FPGA surpass the capabilities of digital signal processors by taking the advantage of hardware parallelism instead of keeping with sequential execution. They allow you to control what is happening to the lowest level, which in the end provides faster response times. FPGA are capable of reproduce complex systems as full System on a chip (SOC), with an integrated processor, RAM memory, etc.

- **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**: Whether ASICs are only affordable when making thousands of units per year, FPGA are economically viable from dozens to hundreds of devices.

- **Reliability**: FPGA provide a true hardware implementation of a program, instead of running it on a full system. 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 brands, main capabilities of some families

There are two main manufacturers of high performance FPGA: Xilinx and Altera. These are their more 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 [XIL14].

- Xilinx Virtex 6: Although built in 40 nm, the previous top series of Xilinx is still a reference in FPGA technology. It is able to manage a high bandwidth interface with DDR3 external RAM and high performance logic. There are different models for various necessities. Each sub-family contains a different ratio of features to most efficiently address the needs of a wide variety of logic designs. For example, it should be noted that among others there are models with up to 2,016 DSP, which makes this family of FPGA very appropriate for implementing heavy calculation algorithms [XIL12a].

- Altera Stratix 5: Stratix are one of the highest performance series in Altera. Built in 28 nm as the Virtex 7 they are also capable of variable precision signal processing and low power functioning. Stratix V devices are available in four variants, each one of them targeted for a different set of applications. The GS series supports up to 3,926 DSP, which makes it very appropriate for calculation intensive applications [ALT14].

It is clear that the power and computation resources have risen to a point where these devices are much more versatile than they were years ago. Describing the amount of DSP by thousands and having the capability to hold DDR3 RAM external memories allow a much more easy adaptation of algorithms developed for other platforms, or a high performance approach to traditional optimized ones.

3.3. FPGA traditional design flow

Design flow in hardware design, in general, and in FPGA in particular is quite different than in software design. This is due mainly to the level of abstraction that software developing implies. This subsection describes the usual workflow when developing hardware in FPGA environment.

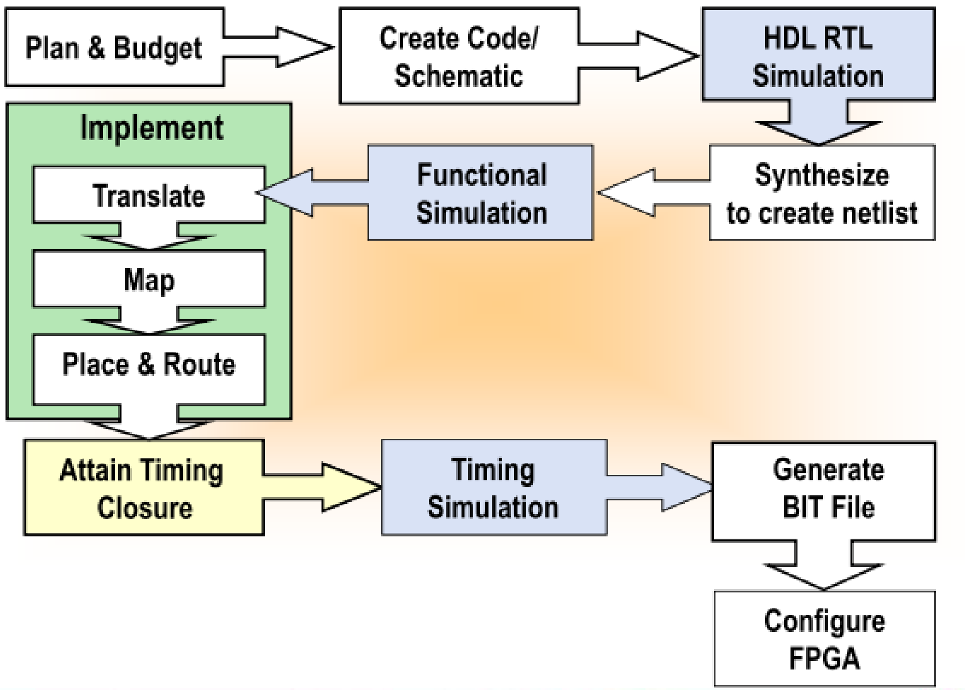


Fig. 6: Traditional step by step design flow (from [XIL11a])

The process can be divided in seven important parts:

1. **Write code or design schematic:** It is needed to write code for every independent module, specifying its inputs and outputs, and the function itself. A top module needs to be written in order to link all the other modules, also specifying the final inputs and outputs of the design. It is usually written in VHDL or Verilog, languages that describe the operation of the circuit, but not how it is translated into logic gates.

2. **Hardware Description Language (HDL)/Register Transfer Level (RTL) simulation:** Once the modules have been created the first step is to make sure that the system operation is correct. This is done creating a Testbench, which is another code –independent from the functional one– that is connected to the first one, and analyses if it is working properly in every situation. This is often very time-consuming, even more than writing the hardware itself, due to the theoretical need of trying all the input combinations, so the hardware responds to every possible stimulus as it should. Usually doing a complete test is not possible, and only the most relevant possibilities are tested.

3. **Synthesize:** After it is known that the hardware works correctly, it is needed that a logic synthesis tool reads the VHDL, and outputs a definition of the physical implementation of the circuit. In other words, synthesis will take the RTL and generate a gate level description that can then be placed and routed. This procedure outputs several netlist files.

4. **Functional simulation:** This is a gate level simulation that checks that every behavioural characteristic of the RTL description is kept when synthesized.

5. **Implement:** This process consists of three stages: merging different design files into the final netlist of the circuit; grouping logical symbols (gates) into physical components in the FPGA; and place these components in the FPGA chip, creating in addition a timing report.

6. **Timing closure and simulation:** Once a timing report is created, it is more likely that some connections –usually between modules- are not time consistent. This can be solved introducing some time constraints that change the place and routing of the circuit. Sometimes the design still does not meet every time requirement, so some strategies can be applied: multi-cycle constraints, false path constraints, map-timing options, changing manually the floorplan and even changing the code so as to avoid large critical paths. If the circumstances lead to this latter option it will mean start the process all over again. After every change a time simulation will be run.

7. **Bit file creation:** After this whole process the created structure needs to be passed to the FPGA in a format that it understands. This format is a '.bit' file, which is a sequence of bits that programs the FPGA changing the links between the different elements consecutively.

It can be guessed that this process is not an effortless one, since especially steps 1, 2 and 6 requires immense amounts of time. Among the difficulties of this design flux one has to be particularly noted: if a carried out design needs to be changed for some reason (e.g. a calculation system that needs to be more parallel), it will need a major restructuration in a high probability. This means that it is difficult to change an existing system, so it is better to carefully plan the system rather than doing later alterations.

3.4. High Level Synthesis tools: a new workflow

In the previous subsection it is described the whole traditional design flux, and it is stated that one of its main advantages is that it allows the developer to control every step, and to plan meticulously the design of the circuit at a very low level. It also has some important cons. For example, the process is time consuming; furthermore, it implies a big amount of work for some of the steps when compared to software development.

In the last paragraph it was highlighted a relevant problem of this working method: once the design is finished, if it requires further changes they will be very difficult and time consuming to carry out. This is a very common situation for different reasons, for example because of timing specifications that are not completely met in the final implementation and the program cannot solve through constraints or just because it has been found a better way to accomplish some action. It is even worse for some type of problems that require fast prototyping, and cannot admit restructuring the project from top to bottom frequently. For these kinds of projects a new workflow is needed.

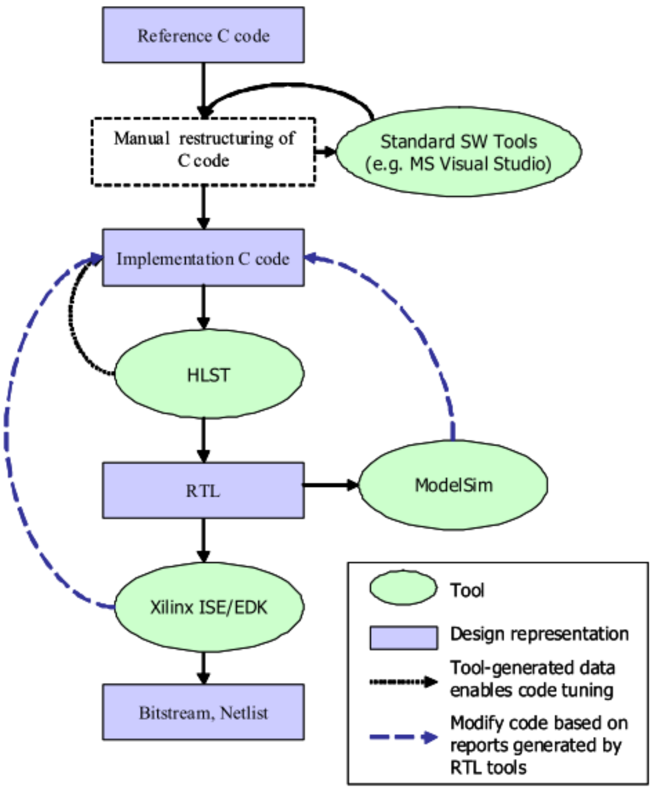

Reference C code
Manual restructuring of C code
Standard SW Tools (e.g. MS Visual Studio)
Implementation C code
HLST
RTL
ModelSim
Xilinx ISE/EDK
Bitstream, Netlist
Tool
Design representation
Tool-generated data enables code tuning
Modify code based on reports generated by RTL tools

Fig. 7: Typical high level synthesis tool workflow (combined with Xilinx tools), from [BDT10]

For applications as the ones described before and also for complex algorithms or applications that would take a lot of time to implement, there are other options. One that is creating lots of expectations as it is becoming more common to use is High Level Synthesis tools (as Vivado HLS). It allows developers to create custom modules from C, C++ or System C code. It then creates VHDL or Verilog files as output, so these modules can be included in a classic electronic design or in an embedded system with other components. In this way the program allows the developer to create hardware without having to manually create RTL files. This has several advantages compared to the traditional design flux:

- The same algorithm in a higher level language is easier to create, and conceptually closer to the traditional way of programming.

- It is faster to rewrite a piece of high level language than a module. Usually it is more intuitive.

- The testbench can also be created in C, C++ or System C, which leads to really fast check of the specifications.

- C and C++ are broadly used languages, which means that a big base of programs can be implemented in hardware with relative little effort.

- Easy directive driven optimization of algorithms and processes. It allows more control on the synthesis that Vivado HLS does, and then more control over the result module.

- Vivado HLS uses automatically on-chip memories (block RAM and flip-flops) and also arranges DSP elements – using floating-point libraries if they are required, for example–.

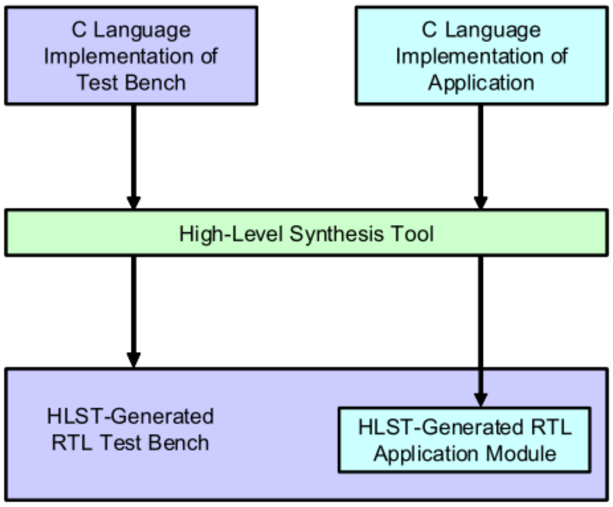

C Language Implementation of Test Bench
C Language Implementation of Application
High-Level Synthesis Tool
HLST-Generated RTL Test Bench
HLST-Generated RTL Application Module

Fig. 8: Typical HLSTs automatically generate RTL test benches in addition to the RTL module implementing the design, from [BDT10]

In opposition of the positive features these tools also have some negative ones:

- High-level synthesis tools cost considerably more than DSP processor software development tools, and more than the average tool used for the same aims.
- HLS tools take away part of the freedom RTL languages contribute to.

As the last point suggest, although Vivado HLS and other high level synthesis programs can be very convenient, this new work flow simplifies the process so much that nearly everything in the final hardware configuration is left to be decided by the synthesizer. In this context it can imply two different results: whether the synthesizer is really competent and outputs a very decent result; or it is not so good and then programming hardware in such a way is not a good idea.

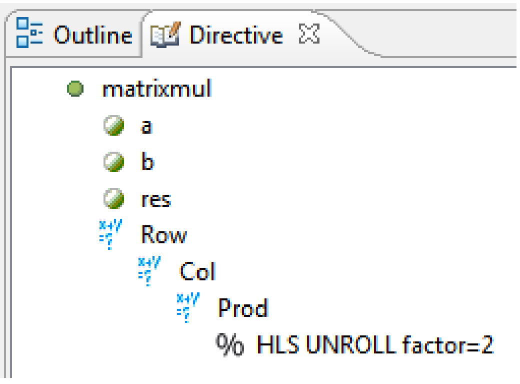
The answer is something in between. This question has been addressed in a number of studies. One of the most interesting ones was requested by Xilinx before buying the program that would be Vivado HLS in the future [BDT10]. In this publication two algorithms, which are very often implemented in FPGAs, were programed both in RTL and C in Vivado HLS (AutoESL), or with DSP processor implementation and VHLS. These two methods were a video processing algorithm (Optical Flow Workload) and a wireless communications receiver baseband application (DQPSK Receiver Workload). Then its results were studied to have a general view over what performance high level language synthesizers could achieve.

In the video processing algorithm the FPGA implementations created using high-level synthesis tools achieved roughly 40X the performance of the DSP processor implementation. BDTI also evaluated the efficiency of the HLST-based FPGA implementations of the DQPSK workload versus the same problem implemented using hand-coded RTL. Here, too, the HLSTs performed very well. The code produced by the HLS tool was comparable in resource efficiency to the hand-written RTL code.

Clearly, FPGAs used with high-level synthesis tools can provide a compelling performance advantages for some types of applications.

Although these tools have proved to be very useful, they do not get the work done by themselves. It is needed that the engineer ‘helps’ them with some kind of guidelines where the synthesizer can hold on: directives. Directives are code instructions for the synthesizer to read that will usually restrict the freedom of the toll, so it does what the developer chooses. These directives are usually aimed at certain parts of the code that can be optimized in a specific way, –such as loops–, or to variables that can be stored, outputted, read, etc. more conveniently.

Directives can be added with a dialog window, which allows the user to choose the type and set a few parameters of it. For example, among the options of the directive unroll there is one that allows the user to select how many sets of hardware will be created, the option ‘factor’. This gives directives a great flexibility, allowing countless possibilities. The directives added in this way are specified by the program in a special file called “directives.tcl”, which is unique for every solution. It simplifies the process of using new directives, and comparing the results given by each one.



```
matrixmul
    a
    b
    res
    Row
        Col
            Prod
                % HLS UNROLL factor=2
```

Fig. 9: Directives are shown up in the place where they take effect.

Although adding directives in this way may seem comfortable, it is very common to have a set of directives that are well known, i.e. interface ones. Because of this Vivado HLS allows the user to write them directly in the code file, so this directives will be permanent between solutions, and maybe more important, it makes it very easy to copy the code to another project without having to set each one of them.

It is clear that high level synthesis is a very valuable option, especially when a project is limited in time or budget, or when it is needed to prototype several times before having a final system. In the future using Vivado HLS and other high level language synthesizers will be very common in all kinds of projects as synthesizers are improved and more capable of dealing with a broader variety of problems.

3.5. Platform description (Board features and components)

The platform chosen to implement the system of this project is the Virtex-6 FPGA ML605 evaluation board [XIL11b]. It includes all the basic components of hardware, design tools, IP, and a reference design for system designs that demand high-performance, high speed connectivity and advanced memory interfaces.

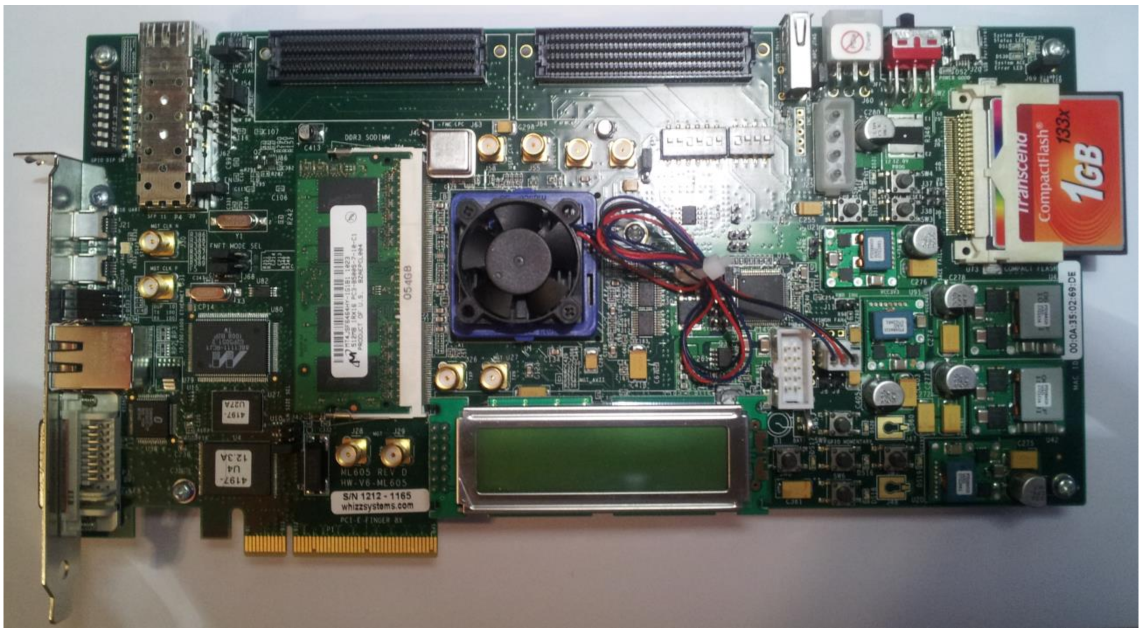


Fig. 10: Picture of the actual ML605 board that has been used

These components are the main ones:

- **Virtex-6 XC6VLX240T-1FFG1156 FPGA**: It belongs to the LXT family (inside the Virtex 6 one) which is the one specialized in high-performance logic and advanced serial connectivity. It includes more than 31,000 slices (each one with four LUTs, flip-flops, multiplexers and arithmetic logic) and 768 DSP. Nearly 15 Mb of block RAM are also embedded. It supports configuration both from JTAG (USB and CF) and from the Linear BPI Flash device.

- **512 MB DDR3 SO-DIMM RAM**: This external memory has been tested up to 800 MT/s, and its socket offers support up to 2GB of DDR3 memory.

- **32 MB Linear BPI Flash:** This non-volatile storage is often used to configure the FPGA when a connection with the JTAG USB is not provided, or it is being used without a PC.

- **System ACE CF and CompactFlash Connector:** This is a key feature of the board for this project. It includes a 2 GB Compact Flash card where files can be stored and read. It also enables the configuration of the Virtex-6 FPGA from the CF. System ACE CF controller supports up to eight configuration files. It also enables an embedded processor to access the files stored in the CF. easily.

- **10/100/1000 Tri-Speed Ethernet:** As this board is specialized in fast serial connections it includes a physical Ethernet connection with Gigabit speed, which enables the board to be fed in a very efficient way from a PC or a server.

- **USB UART:** The ML605 contains a USB-to-UART bridge device, which allows connection to a host computer with a USB cable. Drivers are provided so the connection appears as a COM in the host PC.

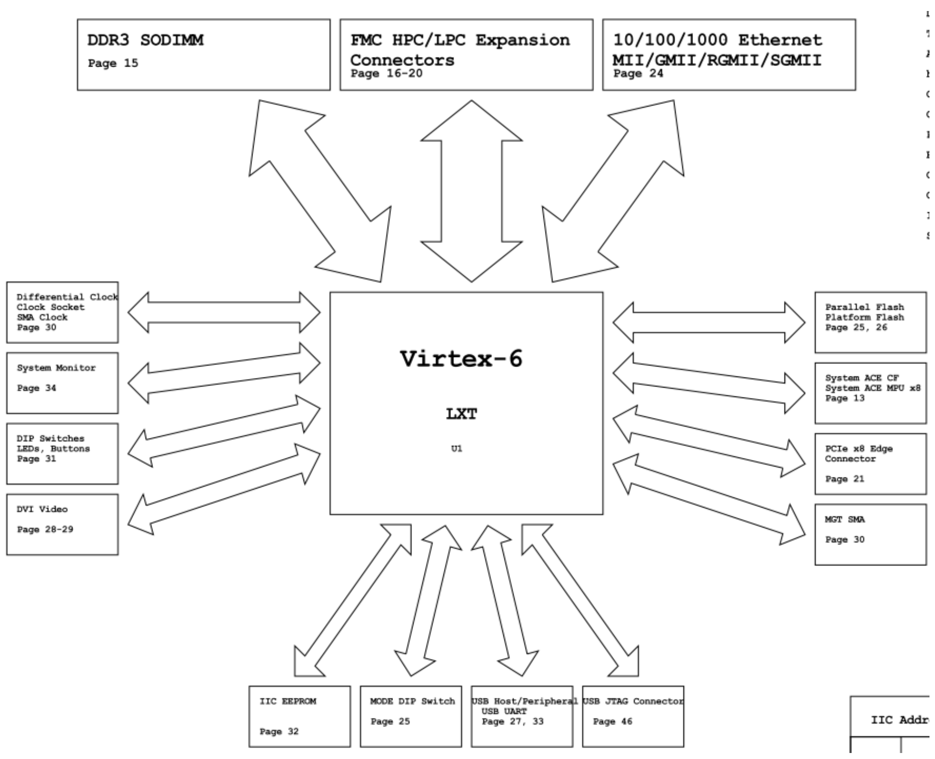


Fig. 11: Schematic of the ML605 board from [XIL09]

4. First functional prototype (reduced Look-up table set)

4.1. Matrix multiplication algorithm in Vivado HLS

The embedded system that has been developed consists on different interconnected modules. Most of them are commonly used ones (i.e. RAM controller, general purpose microprocessor…) that have been already developed by Xilinx. Even though, the most important part of the system is the particular purpose hardware module that performs the algorithm itself.

As it is stated in the 2.6 section, the algorithm that has to be implemented in hardware, optimized and therefore parallelized, consists on several parts. The most important and more time consuming one is the matrix multiplication. The main multiplication is:

$$M * (O - N)$$

Where M is a Matrix of 4x100 dimension, and O and N are both matrix of 100x1 size. This gives as result a matrix of 4x1, being each one of the individual values a necessary parameter:

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

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.

To find out the best way to optimize a matrix multiplication with so many elements (in this case 400 in one matrix and 100 in the other one) is difficult, so it is more reasonable to start with a general case that multiplies two 4 by 4 matrix. This size is big enough to give a good perspective about the process, yet not as large as to be unmanageable. In this case it is needed to do 4 multiplications and from 3 to 4 sums (depending on the way the algorithm operates) to produce each element of the result matrix.

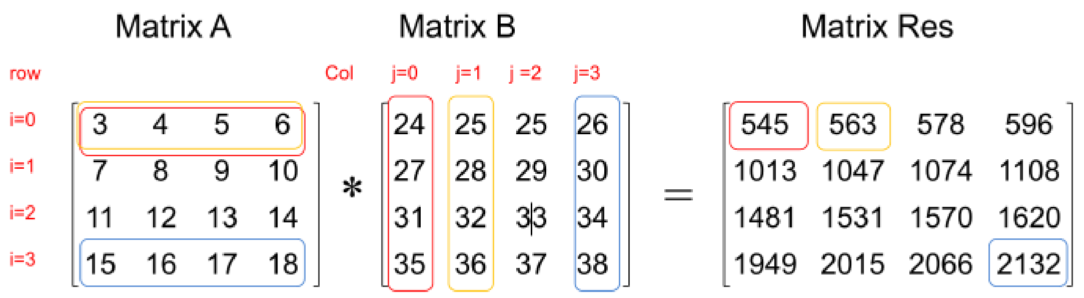


Fig. 12: Example of 4×4 matrix multiplication

One of the simplest C code which is capable of performing the full multiplication, remember that this hardware module is being programmed in Vivado HLS, is this one:

```
// Iterate over the rows of the A matrix
for(int i = 0; i < MAT_A_ROWS; i++) {
        // Iterate over the columns of the B matrix
        for(int j = 0; j < MAT_B_COLS; j++) {
                res[i][j] = 0;
                // Do the inner product of a row of A and col of B
                for(int k = 0; k < MAT_B_ROWS; k++) {
                        res[i][j] += A[i][k] * B[k][j];
                }
        }
}
```

Code piece 1: Basic code for matrix multiplication

The operation that this code is doing for each value of the result matrix is the following one (in this case for the element of the row 3 and column 3 of the result matrix):

$$R_{33} = A_{30}\cdot B_{03} + A_{31}\cdot B_{13} + A_{32}\cdot B_{23} + A_{33}\cdot B_{33}$$

This code is only using one accumulator and one multiplier, so it is just doing one action at a time, using a tiny fraction of the resources, but performing a lot of cycles. The result is that it takes for this algorithm 617 clock cycles to finish one complete operation and be ready to start again (latency).

From this initial state several optimizations can be made. For every optimization there has to be at least one directive in the code, which will usually affect to part of the code.

4.1.1. Loop unroll directive

The first directive that will be used is "loop unroll". As its name suggest this directive has effect in a loop. By default loops are rolled in Vivado HLS, so it will instance hardware only for one iteration, and therefore each one will be done successively. Loop unroll will force Vivado HLS to set more hardware when it "translates" a loop. By default it will create independent hardware for each iteration, so if the loop has to be repeated four times it will create the four sets of hardware. This will only occur if each iteration is independent from the results of each other, which depends on the case [12].

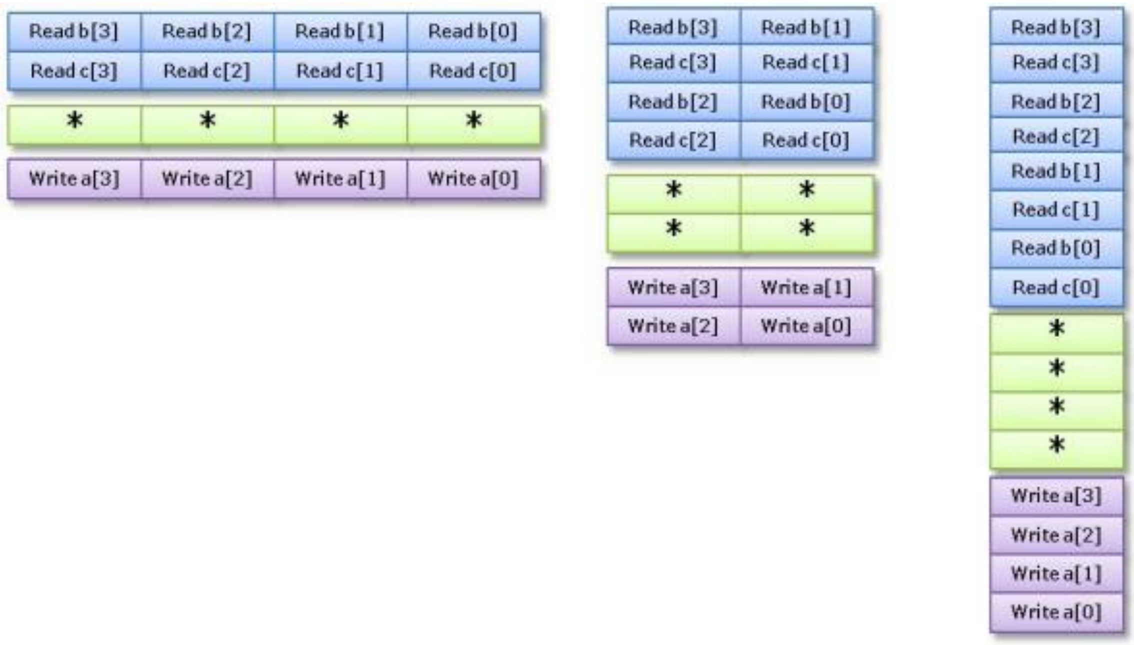
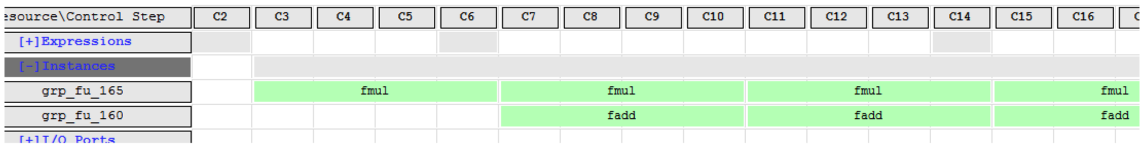


Fig. 13: Schematic of loop unrolling directive (From [XIL13]).

In Fig. 13 it is shown a schematic of how a loop can be rolled (right), partially unrolled (middle) or completely unrolled (left). Clock cycles are represented in vertical in the figure, while the number of hardware sets is the number of columns.

Applying this directive over the most inner loop with a factor of 2 creates 2 sets of hardware, and therefore the process is accelerated. In this case the latency goes down to 457 clock cycles. It does not duplicate the performance due to dependencies between the iterations. It is clear that to do each iteration of the most inner loop it needs the result of the previous one, so it cannot do two iterations at the same time. Despite of this, it is faster because it does not need to wait until the whole loop is finished to start multiplying the following numbers.



esource\Control Step	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C
[+]Expressions																
[-]Instances																
grp_fu_165			fmul				fmul				fmul				fmul	
grp_fu_160							fadd				fadd				fadd	
[+]I/O Ports																

Fig. 14: Analysis view of the unrolled basic code in VHLS.

Dependencies are so strong that even with a full unroll, which creates 4 sets of hardware, the latency is still 361 clock cycles. A quick look to the analysis view in Vivado HLS reveals that, even though there are two float adders and two float multipliers, the program cannot use them efficiently due to dependencies in the data.

4.1.2. Pipeline directive

Another interesting directive is “pipeline”. This directive will create a pipeline in the loop where it is taking effect. In this way it will try to maximize the time that every stage of the loop is used by executing several of them concurrently. So, for example, if the loop implies reading a data and then adding it to other one, when it starts adding the first stage starts reading the next data. This usually leads to a better overall performance [12].

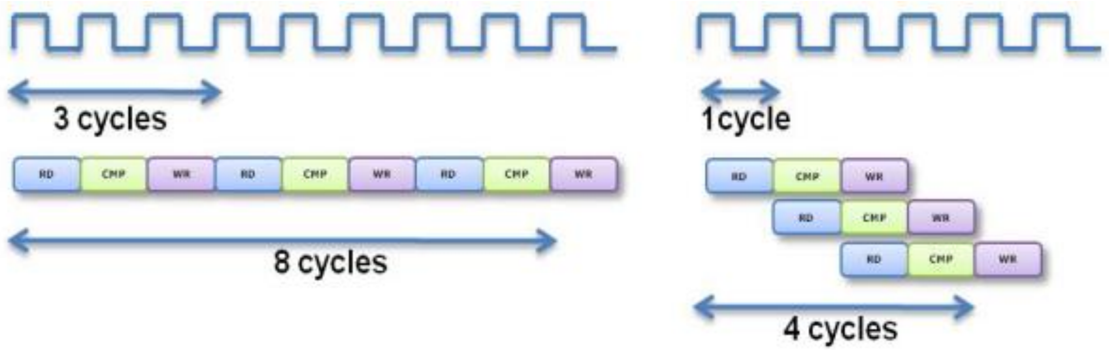


Fig. 15: Left: Loop without pipelining. Right: A totally pipelined loop uses all the resources all the time, from [XIL13].

A key concept in a pipeline is what Xilinx calls the initiation interval (II) [12]. It is the amount of cycles that are needed from the start of one loop iteration until the start of the next one. It is important because it determines how efficiently a pipeline is set, and then how efficiently the hardware resources are used, which is the aim of the pipeline itself. In the pipeline in the figure X (right picture) the initiation interval is one, because each iteration starts only one cycle after the previous one has done so. This is the most efficient way to set a pipeline. A II number greater than one indicates that there are dependencies between the iterations, and then the pipeline is not the best it can be. In figure 16 it can be seen that a pipeline composed by these two stages (*fmul* and *fadd*) would have an Initiation Interval of 4, because it would need 4 clock cycles between one iteration and another one, and thus it would not be an optimal pipeline.

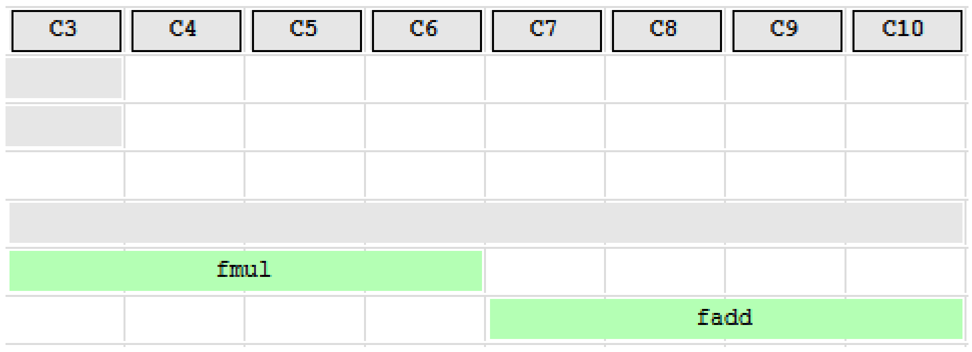


Fig. 16: Non-perfect pipeline with II=4.

By default the pipeline directive unrolls the loops that are inside the pipelined loop. In this way it uses more resources, but under normal conditions it is much more efficient. Pipelining the most inner loop (without the unrolling directive which was applied before) reduces de latency to 385 clock cycles. It can be seen in the figure 15 that the improvement should be much bigger for a number of cycles this big. This is, again, because of dependencies in the code. It is shown in fig. 17 how the *fadd* stage cannot start until the *fmul* one is not finished. This happens because of the operation order that the algorithm used imposes: it first needs the result of the multiplication of the elements to start summing it with the previous result.

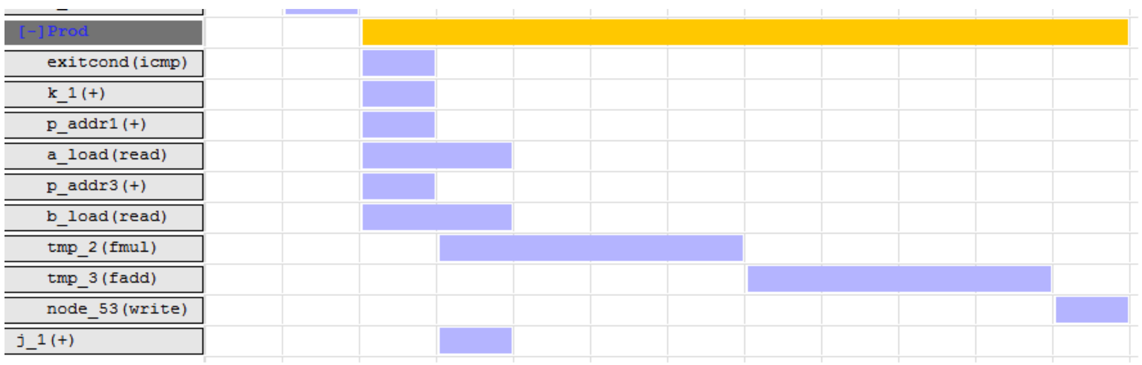


Fig. 17: Analysis view in which it is shown that *fadd* cannot start before *fmul* finishes.

It is also an option to put the pipeline directive in the intermediate loop. In this manner (because of the default settings of the directive in Vivado HLS) the synthesizer will unroll the most inner loop, helping to create more parallelisms in hardware, and possibly improving the results.

The results improve dramatically with this approach. The algorithm is performed with a latency of only 54 cycles. It can be seen now in the figure 18 (analysis view in VHLS) that four multipliers and four adders are created, and as the dependencies are only in between one addition and the next one, the whole process of delivering one result element can be performed in nearly the time needed for the sums. Then the upper loop is pipelined and the calculation for one matrix element is independent from the calculation of another one. In this case there are 4 sets of hardware, so the iterations of this loop can be correctly pipelined, with an initiation interval of only 2 (due to the reading of the data).

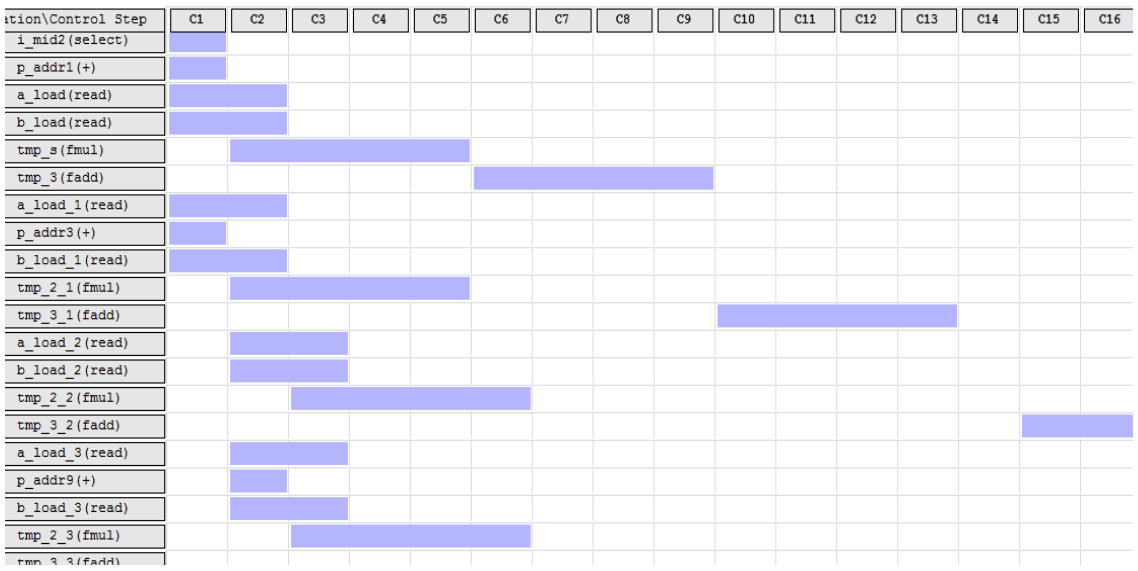


Fig. 18: The multipliers can be used at the same time in this approach, even if *fadd* modules cannot.

It can also be observed in the figure U that only one data of each matrix can be loaded at the same time (instructions *a_load* and *b_load* at the top left corner and below), and this is delaying one cycle each iteration. This detail will be addressed in depth in the following section.

Another interesting detail that a perceptive observer would see in fig. U is that the second data loads (*a_load_2* and *b_load_2*) starts in the second cycle (C2), while at the same time the first loads have not finished yet. Although this design can only load one data of each matrix at a time (because *a* and *b* matrix are each one placed in one blockram), as the operation of loading a data of this size takes less than 15 ns, and each clock cycle lasts 10 ns, it is possible that two sequential reads take place in two consecutive cycles. So both loads are not happening at the same time in real-time. This situation is usual and happens often.

4.2. Code optimization: thinking in hardware

In the last subsections has been described how to optimize the way in which Vivado HLS synthesizes the C code into hardware. But in both cases dependencies in the code have made impossible for the directives to be efficient. It turns out that the simplest code to perform matrix multiplications has direct dependencies in its inner loop, because it needs the result of the previous iteration to sum it up with the next one.

It is known that the calculation of each element of the result matrix is not dependent from the calculation of another one. Reordering the loops and adding an accumulator for each row of the matrix will solve the previous dependencies while keeping the algorithm at the same level of complexity. This design will use the same resources than the previous one, with the exception that it will need more registers to store the accumulators, in fact as many more as the number of rows.

From now on the pieces of code that are going to be shown work with the final configuration of the operation. In other words, the matrix multiplication will be actual one: sizes 4×100 and 100×1. It can be noted that the result matrix will be now a 4×1 matrix. The purpose of doing this is that continuing to make changes on the algorithm over an operation which is not the real one could potentially lead to misguided progress.

Within the necessary changes to adapt the code to a different size matrix multiplication, it is notorious that the number of loops has been reduced from 3 to 2. This is due to the special size of the second matrix, which is 100×1. The algorithm is the following one:

```
// 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++) {

                  temp[i] = a[i][k] * b[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];
    }
  }
```

Code piece 2: New algorithm applied to a multiplication of matrix of sizes 4×100 and 100×1

As it can be seen now the middle loop in the initial code (code piece 1) has been erased, because it was going through the second matrix columns, and in this case the size of it is 100×1, so no loop is needed.

After this consideration the inner and outer loops have been reversed. So the now named 'Prod' loop was before the inner loop, and 'Row' loop was the outer one, although they still go through the same indexes, which are the rows of the second matrix (or the columns of the first one) and the rows of the first matrix. From now on the matrix will be called A and B, for the 4×100 and the 100×1 respectively.

This inversion of the loop order makes the algorithm work differently, achieving the same results. In this way the first loop repeats itself 100 times, one for every column, and for every column of the matrix A the second loop performs 4 multiplications. It is easier to explain with a table the behaviour of the matrix multiplication:

i	**Res. Elem.**	**Prod Loop iteration k=0**	**Prod loop iteration k=1**	**Prod loop iteration k=2**	**Prod loop iteration k**	**Prod loop iteration k=N-1**
0	C_0	$A_{10}\cdot B_0$	$A_{11}\cdot B_1$	$A_{12}\cdot B_2$	$A_{1k}\cdot B_k$	$A_{1(N-1)}\cdot B_{(N-1)}$
1	C_1	$A_{20}\cdot B_0$	$A_{21}\cdot B_1$	$A_{22}\cdot B_2$	$A_{2k}\cdot B_k$	$A_{2(N-1)}\cdot B_{(N-1)}$
2	C_2	$A_{30}\cdot B_0$	$A_{31}\cdot B_1$	$A_{32}\cdot B_2$	$A_{3k}\cdot B_k$	$A_{3(N-1)}\cdot B_{(N-1)}$
3	C_3	$A_{40}\cdot B_0$	$A_{41}\cdot B_1$	$A_{42}\cdot B_2$	$A_{4k}\cdot B_k$	$A_{4(N-1)}\cdot B_{(N-1)}$

In each iteration of the 'Prod' loop the whole corresponding column is performed. In each column, a row is the equivalent to one iteration of the 'Row' loop. And every result element is the sum of the N elements in a row of the table.

The result of this loop reordering is that there are no dependencies between the iterations of the inner loop. It is so because there are no dependencies between the calculation of each result element, and this loop is calculating one part (of the N existent ones) of each of the four result numbers (C_0, C_1, C_2, C_3).

Another important change is that four accumulators have been added ('acc'). These accumulators prevent the constant writing in the final value, allowing the synthesizer to organize it in a more efficient way. It has also been differentiated the first multiplication, which only needs to be stored, from the rest of them, which in contrast need to be summed up to the previous result, because of the same reason.

When this approach is synthesized and its results analyzed it is clear that a big amount of reads are delaying the pipeline, making each iteration last more. This is due to the storage placing of variables A and B. They are supposed to come from the outside of this core (there is more information about this in the next section 'Design decisions'), so it is slow to read them over and over in each iteration. A solution to this situation is that if it is the first time that they are read, they must be stored in an internal variable, which is much faster to read. This alternative is expressed in code piece 3 (below).

```
// 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];
    }
  }
```

Code piece 3: Modified multiplication code. It includes some optimizations and changes.

In code piece 3 there is also another important change relative to the nature of the algorithm. In section 2.6 it is explained that the vector that is called B now comes from the subtraction of the model vector from the observational vector. The code has also included this detraction as vectors ‘c’ and ‘b’, and its results as ‘diff’.

In addition to the changes in the algorithm itself, there are two directives that are fundamental for the performance. One of those is pipeline, just as it is commented above. In this case it takes effect in the ‘Prod’ loop, which is the outer one. This leads to an unroll of the ‘Row’ loop that replicates the hardware for the operations four times, parallelizing in this way the algorithm.

Despite the fact that the pipeline creates ‘four paths’, they need to be fed properly if the whole hardware is wanted to work without bottlenecks. This is not the case now, because in every stage it needs four elements from the matrix A, and A is stored in only one block RAM, and comes to this block RAM from an external storage element by element. The way to optimize this is setting the ‘Array partition directive’.

This directive divides an array variable, in this case the matrix A, in different sets of variables, stored and managed separately in hardware. This effectively increases the amount of read and writes ports for the storage, improving the throughput of the design, and reducing the Initiation interval of the pipeline. In this particular code the directive is followed by the option ‘complete dim=1’ because it is convenient that the matrix is partitioned in the four rows, because then each of them will feed the data to the four identical data paths.

To this late code it is added this last piece:

```
if (k == (MAT_B_ROWS-1)) {
        res[i] = acc[i];
        index_res[i] = (int)(round((acc[i]-start[i])/step[i]));
        res_index[i] = index_res[i];
}
```

Code Piece 4: New line to calculate result index

As it shows, two lines are added in the 'if' structure that is executed when a final result is finished. They close the algorithm cycle by returning to the processor through 'res_index' the new values of the $\{N_T, x_c, y_c, s\}$ vector, so the next iteration has a starting point. To be able to do this the start point and the step length has to be received from the processor in order to know the range of the parameters.

The results in performance and resource usage efficiency of the final code are shown in the section 5. Results.

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 AO algorithm core.

4.3.1. Embedded system

In the beginning of the project the nature of the whole prototype was not decided, with the only exception of the FPGA base. It was clear that a new specific hardware needed to be created, and it needed to be provided with data, control, etc. Creating specific hardware for these functionalities would be very time consuming, and as the project is limited both in time and resources the best option is to reuse components that are general for most systems. This type of systems is called embedded ones. They are specific purpose systems, but they use lots of common components.

4.3.2. System Control

All the hardware that is going to be included in the system has to be coordinated, so every module knows when to start its functioning, and every action is performed in time. In a simple system usually specific control logic is developed for it. It saves area on the FPGA, although is generally limited in functions.

In more complex systems, or if a quick approach to the problem wants to be developed, general purpose microprocessors can be used. Because of the limitation in resources in a FPGA platform these processors tend to be simplified ARM architecture ones, or reduced instruction set cores (RISC). This lead to a relatively low performance, but it is typically enough for the requisites of the application, because (as it is the case) the complicated and specific parts of the algorithm are left to other modules.

This latter alternative was chosen in this project because of its ease of implementation. The model of microprocessor used was the Xilinx microblaze [XIL12b]. It was chosen because it is integrated in the embedded system tool that is used in the project (Xilinx Platform Studio, XPS [XIL12c]), so it allows an ease of use with some other standard Xilinx modules, which are necessary for the communication of the processor with peripherals.

Some useful features of the microblaze processor are:

- It is a soft core processor, which means that it can be wholly implemented using logic synthesis. In other words, it can be embedded inside of an FPGA, and optimized for this use.
- It is highly configurable, allows the user to choose what features are needed, therefore saving resources.
- It is compatible with every driver module that Xilinx provides, for example RAM or Ethernet ones.

4.3.3. External memory

It was explained in 2.6 section that the planned system requires a multidimensional look-up table (LUT) that stores pre-computed data to feed the implemented algorithm. It was also discussed that the storage space needed was 370.4 MB. It is clear that the internal memory in the FPGA block RAM is not enough (it is roughly 3.65 MB), so another resource must be used. In this case the ML605 board includes 512 MB of DDR3 RAM memory, which is sufficient for the purposes of the project.

This memory comes in a SO-DIMM format, and these are its main features (from [XIL11b] and [MIC07]):

- 64-bit wide interface, tested up to 800 MT/s.
- It is directly wired to the FPGA, so the memory controller must be embedded in it.
- Bandwidth up to 8.5 GB/s.

4.3.4. Compact Flash

This LUT file has to be written to the DDR3 RAM in the initialization of the system, because RAM memory is volatile. This means that the file must be stored in some permanent memory if the system is wanted to work independently. The easiest option is to store it in the Compact Flash (CF) included in the ML605 Kit (its size is 2 GB).

A Compact flash driver ('axi_sysace' module [XIL11c]) will be included in the system. It will be connected with the Compact Flash reader in one side, and in the other one with the Microblaze, so it can handle all the communication between the CF and the DDR3. This option was chosen because this process does not have any time constraint, since it will be done just during the initialization of the system, and will not affect any other process.

4.3.5. Algorithm core communications with the system

The specific core that has been designed has to have connections with the rest of the modules. These buses belong to the Axi family [XIL11d], which is a part of ARM AMBA, a family of micro controller buses. There are three different types of Axi buses, and all of them are used in this project:

- **Axi4**: Is a high-performance bus. It is used when memory-mapped data needs to be accessed.
- **Axi4 Lite**: It is a simpler version of the original Axi 4. It is used when data is accessed by address, but there is no need of a high throughput.

- **Axi4 Stream**: It is the simplest bus of the three. It allows high speeds, but the data cannot be addressed, so they have to be accessed in the same order they are stored.

Depending on the requirements of each port it is implemented with one type or another one. The implementation is represented as one or several directives in Vivado HLS, and the different buses created in Xilinx Platform Studio (the Xilinx tool for embedded system design). The piece of code 6 shows the directives used to connect the core with the rest of the system:

```
/******** AXI4Lite **********/
#pragma HLS RESOURCE variable=return core=AXI4LiteS
#pragma HLS RESOURCE variable=base_addr_ext core=AXI4LiteS
/******** DDR3 RAM **********/
#pragma HLS INTERFACE ap_bus port=ddr3
#pragma HLS RESOURCE variable=ddr3 core=AXI4M
/******** FIFO + AXI4Stream *********/
#pragma HLS INTERFACE ap_fifo port=c
#pragma HLS INTERFACE ap_fifo port=start
#pragma HLS INTERFACE ap_fifo port=step
#pragma HLS INTERFACE ap_fifo port=res
#pragma HLS INTERFACE ap_fifo port=res_index
#pragma HLS INTERFACE ap_fifo port=param
#pragma HLS RESOURCE variable=c core=AXI4Stream
#pragma HLS RESOURCE variable=start core=AXI4Stream
#pragma HLS RESOURCE variable=step core=AXI4Stream
#pragma HLS RESOURCE variable=res core=AXI4Stream
#pragma HLS RESOURCE variable=res_index core=AXI4Stream
#pragma HLS RESOURCE variable=param core=AXI4stream
```

Code piece 5: Input and output directives in Vivado HLS

The directives are divided in three blocks, one corresponding to each type of bus.

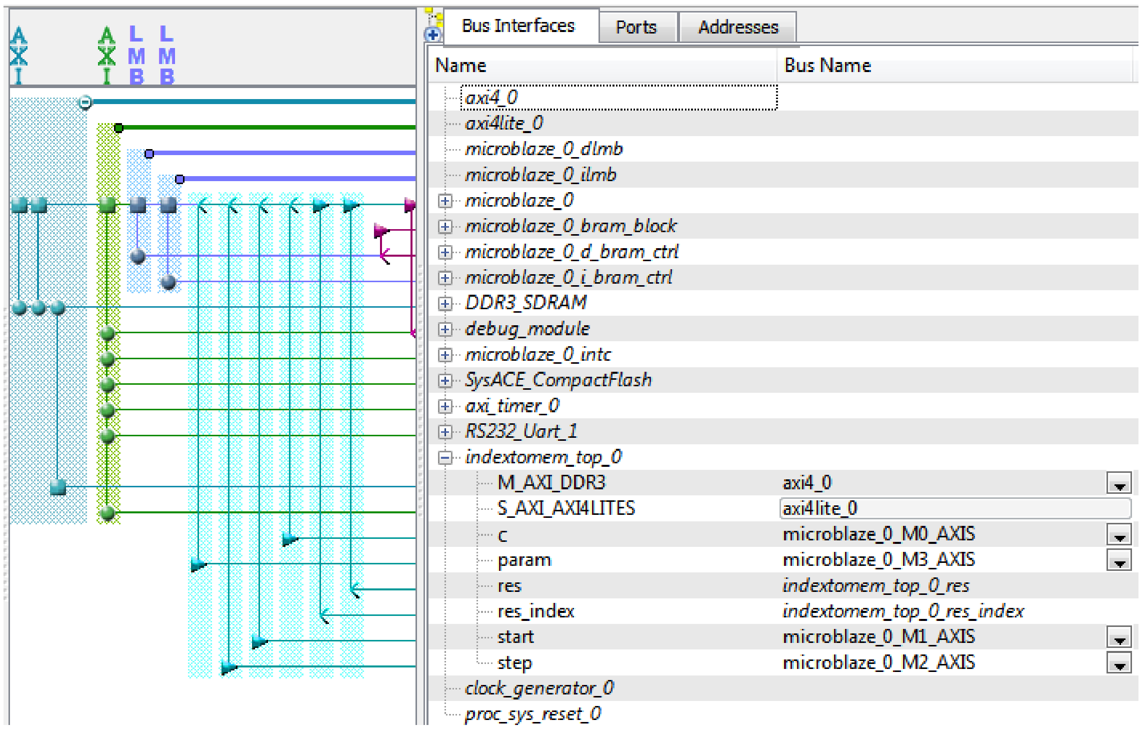
Axi4 bus is only used for the external RAM connection because, although it has a big impact on the resources, it is critical for the performance of the algorithm that the matrix stored in the LUT are accessed in the fastest way possible. It is also needed that this bus is memory-mapped, since the RAM needs an address to be accessed.

Axi4 Lite is used for two purposes: the default return parameters that Vivado HLS sets by default and the base address of the LUT stored in the external RAM. The default parameters are several data that the processor (control logic) must know to manage the specialized core. Among these parameters there are signals as start, stop, idle, etc. In the base address case this type of bus is used because there is no need of high performance here (it is only a 32 bit address which will be received).

All the other signals -*param*, *res*, *res_index*, *c*, *step* and *start*- are set as both FIFO and axi4 stream ports. This is done because they share some characteristics: they are crucial for the timing in the algorithm and they only need to be accessed in order (they do not need an address). Also the stream bus is the one that consumes less resources.

4.3.6. Complete System Schematic

This subsections aims to give a global view of the embedded system that has been developed, avoiding unnecessary complexity and showing clear bonds between the different modules that have been used.



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	

Fig. 19: XPS view of the embedded modules.

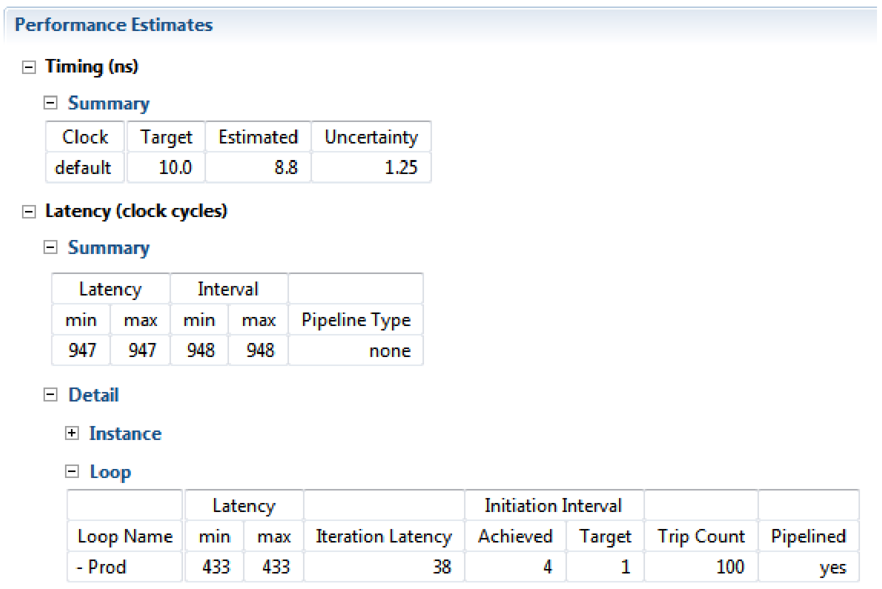
In the figure 19 Xilinx Platform Studio shows how the different modules are interconnected. The dark blue colored bus to the left is the AXI4 one, which links DDR3 RAM, Microblaze and indextomem_top_0 (the specific core that performs the AO algorithm). The dark green bus to the right of the AXI4 is the AXI4 Lite, which bonds mainly the Microblaze, Compact Flash module (SysACE_CompactFlash in the figure), the indextomem core and the RS232 UART. Lastly each of the light blue lines is a AXI4 Stream bus. Actually there are only 4 of this buses, but they are all duplex (can be used in both directions at the same time) and only two of the back channels are being used. Blue LMB buses are not relevant for this project.

5. Results

A prototype of adaptive optics algorithm accelerator has been built and it covers a basic functionality:

- Storing reduced LUT in external memory.
- Performs the algorithm with initial data from processor.
- Returns results prepared for next iteration.

In terms of timing performance these are the estimates results given by Vivado HLS (in clock cycles, at 10ns/cycle):


Performance Estimates

⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
default	10.0	8.8	1.25

⊟ Latency (clock cycles)

⊟ Summary

Latency		Interval		
min	max	min	max	Pipeline Type
947	947	948	948	none

⊟ Detail

⊞ Instance

⊟ Loop

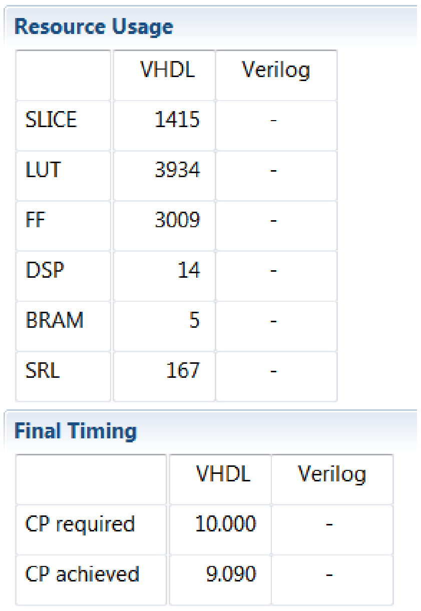
	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	Achieved	Target	Trip Count	Pipelined
- Prod	433	433	38	4	1	100	yes

Fig. 20: Performance estimates for the final VHLS code.

The real time has been also measured during the actual execution in the FPGA prototype, with a result from the activation of the algorithm core until the results are returned and held by the processor of 2979 clock cycles, which makes 29.83 µs. This result only applies for one iteration of the algorithm, even though several are needed to finish it (most likely between 3 and 10). It is an acceptable result, because even though it is said in [MOR12a] that time-scales of 1 µs in the algorithm resolution are needed, several tens of this hardware modules can be embedded in a Virtex-6 FPGA. This latter conclusion can be derived from

Even if the result is only acceptable more optimization can be still carried out, and therefore Vivado HLS proves capable of synthesizing hardware modules for intensive floating-point calculation modules.

In terms of resource usage it is concluded (for what can be seen in figures 21 and 22) that the algorithm core uses around a 2% of the available LUTs in the Virtex-6 FPGA, around a 1% of the flip-flops, nearly a 2% of the DSPs and less than a 1% of the block RAM. Comparing the results from figures 21 and 22 it can be seen that the estimations of Vivado HLS before getting the actual results have a good accuracy.



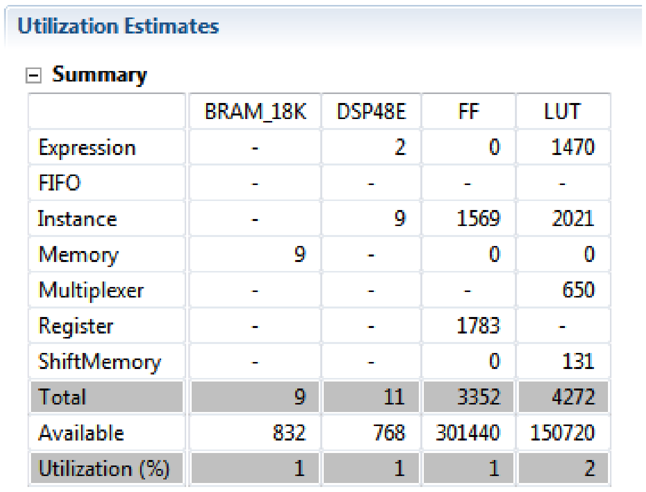
Resource Usage

	VHDL	Verilog
SLICE	1415	-
LUT	3934	-
FF	3009	-
DSP	14	-
BRAM	5	-
SRL	167	-

Final Timing

	VHDL	Verilog
CP required	10.000	-
CP achieved	9.090	-

Fig. 21: Actual resource usage of RTL code generated by VHLS



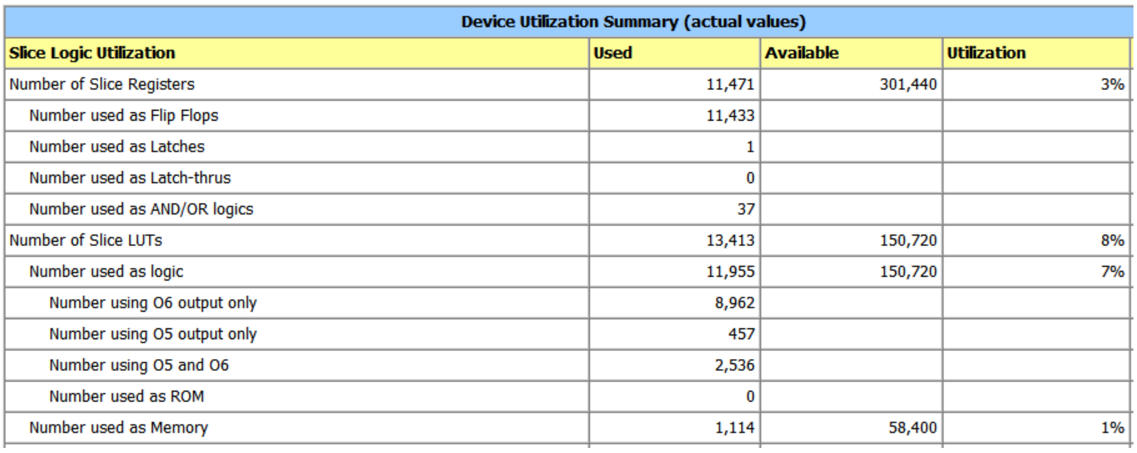
Utilization Estimates

Summary

	BRAM_18K	DSP48E	FF	LUT
Expression	-	2	0	1470
FIFO	-	-	-	-
Instance	-	9	1569	2021
Memory	9	-	0	0
Multiplexer	-	-	-	650
Register	-	-	1783	-
ShiftMemory	-	-	0	131
Total	9	11	3352	4272
Available	832	768	301440	150720
Utilization (%)	1	1	1	2

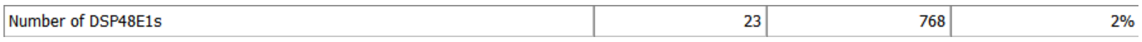
Fig. 22: Resource utilization estimate that VHLS does over the resources of the Virtex-6 XC6VLX240T

Taking a look at the whole system, figures 23 and 24 show the total use of resources. The difference in the resource usage is mainly because of the Microblaze processor, which according with the numbers uses around a 1% of the FF, a 6% of the LUTs, and less than 1% of the DSPs.



Device Utilization Summary (actual values)			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	11,471	301,440	3%
Number used as Flip Flops	11,433		
Number used as Latches	1		
Number used as Latch-thrus	0		
Number used as AND/OR logics	37		
Number of Slice LUTs	13,413	150,720	8%
Number used as logic	11,955	150,720	7%
Number using O6 output only	8,962		
Number using O5 output only	457		
Number using O5 and O6	2,536		
Number used as ROM	0		
Number used as Memory	1,114	58,400	1%

Fig. 23: Resource usage resume in XPS



Number of DSP48E1s	23	768	2%

Fig. 24: Resource usage resume in XPS (DSP)

6. Conclusion and Future work

6.1. Conclusion

These are main of the conclusions that can be extracted from this project:

- High-level Synthesis tools are capable of implementing several types of algorithms (especially the ones that are intensive in floating-point calculations) with a good overall result and taking less time than with the equivalent traditional work flow.

- FPGA are devices capable of implementing adaptive optics centroiding algorithms with a good performance. It seems clear that, with further development, algorithms which precision is very close to the limit of the information given can be implemented with the necessary performance for the whole system to be useful.

- Although HLS tools simplify the process of creating hardware, they let the developer choose over a broad variety of parameters, particularly where optimizing has become a commonplace (as loops and memory resources).

- Embedded systems have proven to be very useful when many elements of it can be taken and implemented without having to be made 'by hand'.

6.2. Future work and optimizations

This preliminary work can be improved and extended in the following directions:

Upgrading to actual LUT: As it is discussed in 2.6. section, all the previously explained work is done with the little LUT of only 81 nodes. This made possible to have results in time, but also means that the work has to continue. Next step is to upgrade the prototype from the 162 KB LUT to the 370 MB one. Some difficulties can maybe be encountered accessing to the file in the Compact Flash.

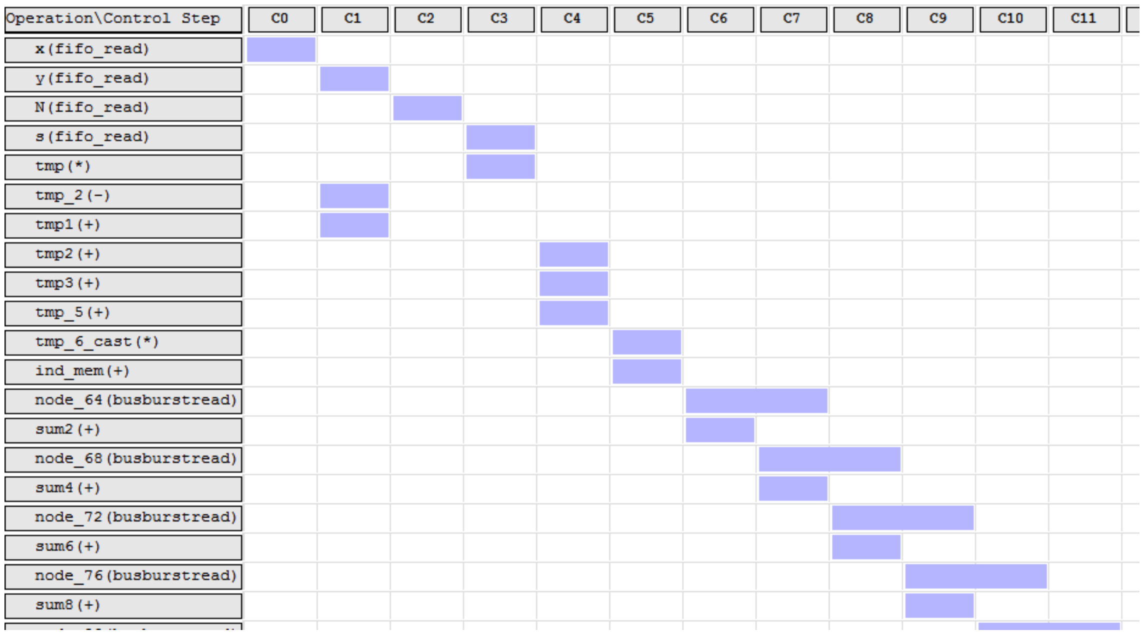
Optimal number of iterations: The hardware module is currently set to perform just an iteration of the Gauss-Newton algorithm. However, this algorithm consists of several ones. The number of them depends on the required precision for the application or on the limit provided by the number of nodes of the LUT.

At this moment it is prepared to find out this number using the code run in the processor, because it will be easier to set a few conditions. When the number of iterations is set, it will be required to integrate it in the hardware as 'for' loop that will include every other code. This loop will not be able to be pipelined, because the first data that needs to be introduced in it will be the last one to get out.

Reliability and accuracy test: It will be necessary to develop a specific test that assures the robustness of the system to invalid data, or to unexpected formats. In addition to this, and because this system is expected to be used with scientific purposes, an accuracy test must be developed. An option of design for this test would consist on running the embedded system, and then a Java program that executes the same algorithm with the same initial images. Then a comparison between the final results would be done.

Optimizing: The system has strong time requirements to be useful, so further optimization will be most likely needed. This optimization can go from more parallelization of the original hardware module to critical changes in the structure of the whole system. Incremental changes should be implemented, until a certain objective is achieved or until the improvements are negligible

Although several optimizations have been made (especially in the AO algorithm core) there are a few points that need further optimization. For example nearly none optimization has been done in the processor code due to the lack of time.



Operation\Control Step	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
x(fifo_read)	■											
y(fifo_read)		■										
N(fifo_read)			■									
s(fifo_read)				■								
tmp(*)				■								
tmp_2(-)		■										
tmp1(+)		■										
tmp2(+)					■							
tmp3(+)					■							
tmp_5(+)					■							
tmp_6_cast(*)						■						
ind_mem(+)						■						
node_64(busburstread)							■	■				
sum2(+)							■					
node_68(busburstread)								■	■			
sum4(+)								■				
node_72(busburstread)									■	■		
sum6(+)									■			
node_76(busburstread)										■	■	
sum8(+)										■		

Fig. 25: This screen capture shows the inefficiency of the read stage of the algorithm. It is one of the improvement points.

This figure shows the first clock cycles of an iteration of the matrices multiplication loop. It can be seen that there are time and resources wasted when it is reading the initial parameters.

Another point of optimization would be to evaluate if the inefficiencies in the algorithm core are really affecting the timing results in comparison with the memory access. If it happens to be so the multiplication algorithm could be more parallelized dividing the matrix multiplication in two sets of columns (even and odd for example) and then summing the results at the end. This mechanism can be repeated several times, if necessary.

7. References

[ALT14] Altera Corporation. 'SV51001 Stratix V Device Overview' (2014), available at: http://www.altera.com/literature/

[BAS04] Bastian, U. 'The maximum reachable astronomic precision – The Cramer-Rao Limit' (2004), available at: http://www.cosmos.esa.int/web/gaia/public-dpac-documents

[BDT10] Berkeley Design Technology Inc. "An independent evaluation of: High-Level Synthesis Tools for Xilinx FPGAs" (2010), http://www.xilinx.com/technology/dsp/BDTI_techpaper.pdf

[DAV12] Richard Davies, Markus Kasper. 'Adaptive Optics for Astronomy' (2012), available at: http://www.arxiv.org/

[GIA00] Giancoli, D. C., 'Physics for Scientists and Engineers' (3rd edition), p. 896, Prentice-Hall (2000).

[JAM] Space Telescope Science Institute, 'James Webb Space Telescope Near Infrared Camera PSFs', http://www.stsci.edu/jwst/instruments/nircam/PSFs/

[Kel99] Kelley, C. T. 'Iterative methods for optimization' (Vol. 18) Siam, (1999).

[LIN08] Lindegren, L., 'A general Maximum-Likelihood algorithm for model fitting to CCD sample data' (2008), available at: http://www.cosmos.esa.int/web/gaia/public-dpac-documents

[MIC07] Micron Technology Inc. 'DDR3 SDRAM SODIMM MT4JSF6464H – 512MB' (2007). Available at: http://www.datasheetlib.com

[MOR10] Alcione Mora, 'WFS sub-pixel centroiding' (2010).

[MOR12a] Alcione Mora. 'FPGA image centroiding', internal report ESAC, (2012)

[MOR12b] Alcione Mora, Amir Vosteen. 'Gaia in-orbit realignment. Overview and data analysis' (2012), available at: http://www.arxiv.org/

[MOR78] Jorge J. Moré, 'The Levenberg-Marquardt algorithm: implementation and theory' (1978), http://www.osti.gov/scitech/servlets/purl/7256021

[SAK07] User Sakurambo. Available at http://en.wikipedia.org/wiki/File:Airy-pattern.svg (2007)

[UND04] Keith Underwood 'FPGAs vs. CPUs: Trends in Peak Floating-Point Performance' (2004), available at: http://www.uoguelph.ca/

[BJO96] Björck, A. (1996). Numerical methods for least squares problems. SIAM, Philadelphia. ISBN 0-89871-360-9

[XIL09] Xilinx Inc. 'ML605 Block Diagram' (2009). Available at: http://www.xilinx.com/support/

[XIL11a] Xilinx Inc. 'Xilinx Tool Flow' (2011), from Electra Training Org. Course.

[XIL11b] Xilinx Inc. 'UG534 ML605 Hardware User Guide' (2011). Available at: http://www.xilinx.com/support/documentation

[XIL11c] Xilinx Inc. 'DS789 LogiCORE IP AXI System ACE Interface Controller (axi_sysace) (v1.01.a)' (2011). Available at: http://www.xilinx.com/support/documentation

[XIL11d] Xilinx Inc. 'UG761 AXI Reference Guide' (2011). Available at: http://www.xilinx.com/support/documentation

[XIL12a] Xilinx Inc. "DS150 Virtex-6 Family Overview" (2012), available at: http://www.xilinx.com/support/documentation

[XIL12b] Xilinx Inc. 'UG081 Microblaze Processor Reference Guide' (2012)

[XIL12c] Xilinx Inc. 'Embedded System Tools Reference Manual' (2012), p. 13. Available at: http://www.xilinx.com/support/documentation/sw_manuals

[XIL13] Xilinx Inc. 'UG902 Vivado Design Suite User Guide High Level Synthesis' (2013), available at: http://www.xilinx.com/support/documentation

[XIL14] Xilinx   Inc. "DS180 7 Series FPGAs Overview" (2014), available at: http://www.xilinx.com/support/documentation

Apéndice A: Introducción

El proyecto ha sido llevado a cabo en el marco del programa de becarios del centro de astronomía de la agencia espacial europea (ESAC), que da oportunidad de contribuir a las misiones en curso a jóvenes estudiantes de ciencia y de ingeniería. Este proyecto abarca el diseño y la implementación de un prototipo en FPGA, cuyo propósito es acelerar un algoritmo de centrado. También será evaluado si las FPGA son dispositivos adecuados para implementar algoritmos de óptica adaptativa, dadas las típicas restricciones en el tiempo.

Motivación y objetivos

Diversas aplicaciones astronómicas requieren la obtención y el análisis del centroide de imágenes con un rápido ciclo de trabajo. Dos ejemplos son los sistemas de óptica adaptiva y los de control orbital de actitud. Los requisitos de velocidad pueden ser muy restrictivos (μs para la aplicaciones típicas de óptica adaptativa), y los algoritmos que típicamente se usan son ineficientes pero rápidos, como el centro de gravedad de imágenes.

La máxima precisión que cualquier algoritmo puede alcanzar está determinada por la frontera de Crámer-Rao [BAS04]. Los algoritmos de máxima verosimilitud desarrollados para la misión Gaia de la ESA [LIN08] tienen una precisión muy cercana al límite de Crámer-Rao, y pueden ser considerados óptimos en términos de rendimiento. Están basados en el modelado previo: la optimización ponderada de Gauss-Newton de una función similar a los datos observados. A pesar de esto son lentos, y típicamente lejos de los requisitos de velocidad necesitados para aplicaciones críticas en el tiempo.

Los dispositivos basados en FPGA han sido usados en una gran variedad de satélites y otras misiones de la ESA para procesar datos a bordo, y en consecuencia reducir la cantidad de datos mandados a la Tierra. Su utilidad ha sido probada en varios campos de la industria espacial, y la aceleración de algoritmos de centrado pesados podría ser uno de sus usos.

Aunque no está entre los propósitos de esta tésis, una versión avanzada del prototipo que resultará de este proyecto pordía ser conectado de alguna forma a un sensor de frente de onda Shack-Hartmann, recibiendo los datos que este produzca. Entonces el frente de ondas sería reconstruido para una potencial restauración del mismo usando espejos deformables.

Los objetivos de este proyecto se listan a continuación:

- Diseñar e implementar un módulo específico que ejecute el mencionado algoritmo de cálculo de centroides.
- Optimizar el módulo que implementa el algoritmo de cálculo de centroides, adaptando el hardware a las necesidades del algoritmo.
- Probar la funcionalidad del módulo.
- Diseño e implementación de un sistema que provea al módulo con datos y soporte.
- Optimización del sistema, para aproximarse a la escala de 1 µs.
- Evaluación de las posibilidades de un sistema similar que procese en paralelo varias decenas de microlentes, probablemente con un número igual de módulos aceleradores.
- Evaluación de la conveniencia de un sistema FPGA para este tipo de algoritmo y sus restricciones en el tiempo.

Descripción del documento

Después de la introducción, la sección '2. State of the Art of Adaptive Optics', explica el propósito de la óptica adaptative, el problema que este proyecto aborda, varias clarificaciones sobre la tecnología implicada, y finalmente el algoritmo implementado. La sección '3. The use of Reconfigurable HW for adaptative optics' explica por qué esta tecnología podría ser relevante para la óptica adaptativa, detalla las características de la placa de evaluación usada en el proyecto y también incluye algunas consideraciones necesarias para entender las decisiones de diseño. En la sección '4. First functional prototype' se explica como se ha llevado a cabo el prototipo en sí mismo, incluyendo tanto el módulo de computación como el sistema en general. Las secciones 5 y 6 presentan los resultados, el trabajo futuro y las conclusiones del proyecto.

Apéndice B: Conclusión y trabajo futuro

Conclusión

Estas son las principales conclusiones que pueden ser extraídas de este proyecto:

- Las herramientas de síntesis de alto nivel son capaces de implementar distintos tipos de algoritmos (especialmente los intensivos en cálculo en punto flotante) con un buen resultado general y costando menos tiempo de trabajo que con el flujo de trabajo equivalente.

- Las FPGA son dispositivos capaces de implementar algoritmos de centrado de óptica adaptativa de una manera eficiente. Parece claro que, con más tiempo de desarrollo, los algoritmos cuya precisión está muy cercana a límite de la información dada pueden ser implementados con el necesario rendimiento para que el sistema completo sea útil.

- Aunque las herramientas HLS simplifican el proceso de creación de hardware, permiten al desarrollador la elección de una amplia variedad de parámetros, particularmente donde la optimización es más común (como bucles y recursos de memoria).

- Los sistemas embebidos han probado ser muy útiles cuando muchos elementos de ellos pueden ser tomados e implementados sin hacerlos ‘a mano’.

Trabajo futuro y optimizaciones

Este trabajo preliminar puede ser mejorado y extendido en las siguientes direcciones:
Usando las LUTs reales: Como fue argumentado en la sección 2.6., todo el trabajo explicado ha sido hecho con la LUT pequeña de 81 nodos. Esto hizo posible la obtención de resultados en plazo, pero también significa que el trabajo debe continuar. El siguiente paso es mejorar el prototipo usando la nueva LUT de 370 MB en vez de la de 162 KB. Esto puede causar algunas dificultades accediendo al archivo en la Compact Flash.

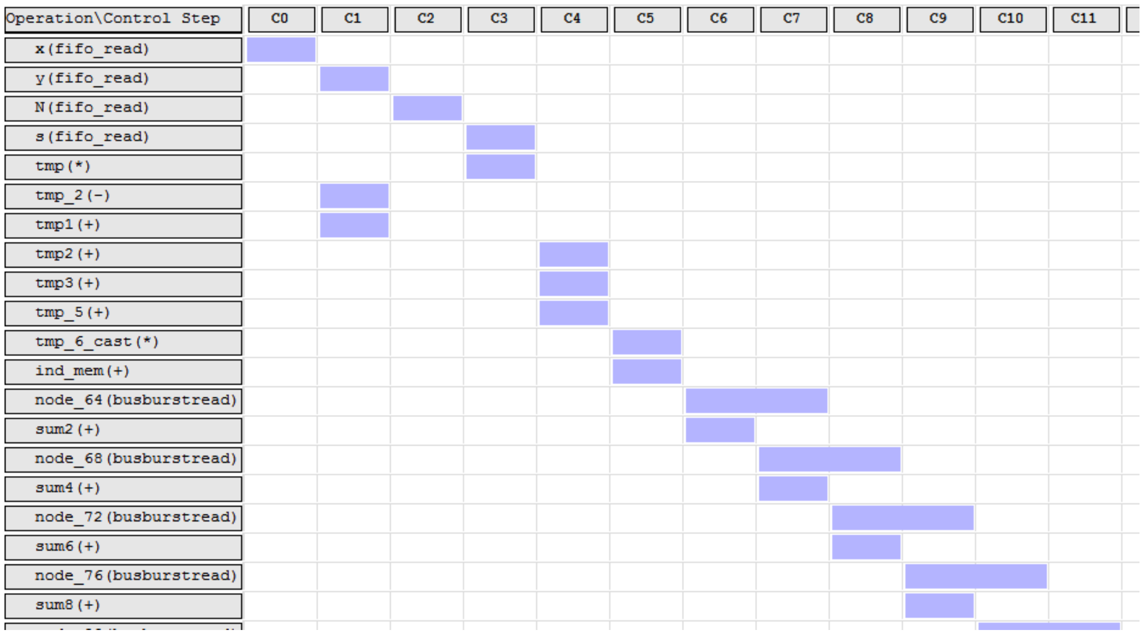
Número óptimo de iteraciones: El módulo hardware está actualmente realizando una sóla iteración del algoritmo de Gauss-Newton. Aunque este algoritmo se compone de varias. El número exacto depende de la precisión requerida para la aplicación o del límite que el número de nodos en la LUT impone.

En este momento el sistema está preparado para encontrar este número usando el código que corre en el procesador, porque es más sencillo poner algunas condiciones aquí. Cuando el número de iteraciones esté fijo, habrá que integrarlo en el hardware como un bucle "for" que incluya todo el resto del código. Este bucle no podrá ser puesto hecho pipeline porque cada iteración necesita al comienzo los datos que la anterior da.

Test de fiabilidad y precisión: Será necesario desarrollar un test específico que asegure la robustez del sistema a datos no válidos, o formatos no esperados. Además de esto, y debido a que el sistema será usado con fines científicos, un test de precisión deberá ser desarrollado. Una posibilidad de diseño para este test consistiría en comparar el sistema con un programa Java que ejecute exactamente el mismo algoritmo con las mismas imágenes iniciales. Entonces se haría una comparación entre los resultados finales.

Optimización: El sistema tiene fuertes restricciones en el tiempo para que sea útil, así que probablemente necesite más optimización. Esta puede ir desde más paralelización del módulo hardware a cambios dramáticos en la estructura del sistema completo. Se deberían implementar cambios incrementales, hasta que un cierto objetivo sea alcanzado o hasta que las mejoras sean despreciables.

Aunque se han hecho diferentes optimizaciones (especialmente en el módulo del algoritmo de AO) hay una seria de puntos que necesitan ir más allá. Por ejemplo casi ninguna optimización ha sido llevada a cabo en el código del procesador debido a la falta de tiempo.



Esta figura muestra los primeros ciclos de reloj de una iteración del bucle de la multiplicación de matrices. Puede verse que tanto recursos como tiempo son desperdiciados cuando está leyendo los parámetros iniciales.

Otro punto para optimizar sería evaluar si las ineficiencias del algoritmo de centrado están realmente afectando a los tiempos resultantes en comparación con los accesos de memoria. Si es así, el algoritmo de multiplicación podría ser más paralelizado dividiendo la multiplicación de matrices en dos conjuntos de columnas (pares e impares por ejemplo) y entonces sumando los resultados al final. Este mecanismo puede ser repetido varias veces si es necesario.

A. Presupuesto

1) Ejecución Material	**3.650 €**
Compra de ordenador personal (Software incluido)	1500 €
Compra Virtex-6 ML605 Evaluation Kit y licencia	2000 €
Alquiler de impresora láser durante 6 meses	50 €
Material de oficina	150 €
2) Gastos generales	**545 €**
15% sobre la "Ejecución Material"	545 €
3) Beneficio industrial	**365 €**
10% sobre la "Ejecución Material"	365 €
4) Honorarios Proyecto	**19.200 €**
960 horas a 20 €/hora	19.200 €
5) Material fungible	**400 €**
Gastos de impresión	100 €
Encuadernación	300 €
Subtotal Presupuesto (1+2+3+4+5)	24160 €
IVA 21% s/subtotal	5.073 €
Total Presupuesto	*29.223 €*

B. Pliego de condiciones

Este documento contiene las condiciones legales que guiarán la realización de este proyecto. En lo que sigue, se supondrá que el proyecto ha sido encargado por una empresa cliente a una empresa consultora con la finalidad de realizar dicho sistema.

Dicha empresa ha debido desarrollar una línea de investigación con objeto de elaborar el proyecto. Esta línea de investigación, junto con el posterior desarrollo de los programas está amparada por las condiciones particulares del siguiente pliego.

Supuesto que la utilización industrial de los métodos recogidos en el presente proyecto ha sido decidida por parte de la empresa cliente o de otras, la obra a realizar se regulará por las siguientes:

Condiciones generales

1. La modalidad de contratación será el concurso. La adjudicación se hará, por tanto, a la proposición más favorable sin atender exclusivamente al valor económico, dependiendo de las mayores garantías ofrecidas. La empresa que somete el proyecto a concurso se reserva el derecho a declararlo desierto.

2. El montaje y mecanización completa de los equipos que intervengan será realizado totalmente por la empresa licitadora.

3. En la oferta, se hará constar el precio total por el que se compromete a realizar la obra y el tanto por ciento de baja que supone este precio en relación con un importe límite si este se hubiera fijado.

4. La obra se realizará bajo la dirección técnica de un Ingeniero Superior de Telecomunicación, auxiliado por el número de Ingenieros Técnicos y Programadores que se estime preciso para el desarrollo de la misma.

5. Aparte del Ingeniero Director, el contratista tendrá derecho a contratar al resto del personal, pudiendo ceder esta prerrogativa a favor del Ingeniero Director, quien no estará obligado a aceptarla.

6. El contratista tiene derecho a sacar copias a su costa de los planos, pliego de condiciones y presupuestos. El Ingeniero autor del proyecto autorizará con su firma las copias solicitadas por el contratista después de confrontarlas.

7. Se abonará al contratista la obra que realmente ejecute con sujeción al proyecto que sirvió de base para la contratación, a las modificaciones autorizadas por la superioridad o a las órdenes que con arreglo a sus facultades le hayan comunicado por escrito al Ingeniero Director de obras siempre que dicha obra se haya ajustado a los preceptos de los pliegos de condiciones, con arreglo a los cuales, se harán las modificaciones y la valoración de las diversas unidades sin que el importe total pueda exceder de los presupuestos aprobados. Por consiguiente, el número de unidades que se consignan en el proyecto o en el presupuesto, no podrá servirle de fundamento para entablar reclamaciones de ninguna clase, salvo en los casos de rescisión.

8. Tanto en las certificaciones de obras como en la liquidación final, se abonarán los trabajos realizados por el contratista a los precios de ejecución material que figuran en el presupuesto para cada unidad de la obra.

9. Si excepcionalmente se hubiera ejecutado algún trabajo que no se ajustase a las condiciones de la contrata pero que sin embargo es admisible a juicio del Ingeniero Director de obras, se dará conocimiento a la Dirección, proponiendo a la vez la rebaja de precios que el Ingeniero estime justa y si la Dirección resolviera aceptar la obra, quedará el contratista obligado a conformarse con la rebaja acordada.

10. Cuando se juzgue necesario emplear materiales o ejecutar obras que no figuren en el presupuesto de la contrata, se evaluará su importe a los precios

asignados a otras obras o materiales análogos si los hubiere y cuando no, se discutirán entre el Ingeniero Director y el contratista, sometiéndolos a la aprobación de la Dirección. Los nuevos precios convenidos por uno u otro procedimiento, se sujetarán siempre al establecido en el punto anterior.

11. Cuando el contratista, con autorización del Ingeniero Director de obras, emplee materiales de calidad más elevada o de mayores dimensiones de lo estipulado en el proyecto, o sustituya una clase de fabricación por otra que tenga asignado mayor precio o ejecute con mayores dimensiones cualquier otra parte de las obras, o en general, introduzca en ellas cualquier modificación que sea beneficiosa a juicio del Ingeniero Director de obras, no tendrá derecho sin embargo, sino a lo que le correspondería si hubiera realizado la obra con estricta sujeción a lo proyectado y contratado.

12. Las cantidades calculadas para obras accesorias, aunque figuren por partida alzada en el presupuesto final (general), no serán abonadas sino a los precios de la contrata, según las condiciones de la misma y los proyectos particulares que para ellas se formen, o en su defecto, por lo que resulte de su medición final.

13. El contratista queda obligado a abonar al Ingeniero autor del proyecto y director de obras así como a los Ingenieros Técnicos, el importe de sus respectivos honorarios facultativos por formación del proyecto, dirección técnica y administración en su caso, con arreglo a las tarifas y honorarios vigentes.

14. Concluida la ejecución de la obra, será reconocida por el Ingeniero Director que a tal efecto designe la empresa.

15. La garantía definitiva será del 4% del presupuesto y la provisional del 2%.

16. La forma de pago será por certificaciones mensuales de la obra ejecutada, de acuerdo con los precios del presupuesto, deducida la baja si la hubiera.

17. La fecha de comienzo de las obras será a partir de los 15 días naturales del replanteo oficial de las mismas y la definitiva, al año de haber ejecutado la

provisional, procediéndose si no existe reclamación alguna, a la reclamación de la fianza.

18. Si el contratista al efectuar el replanteo, observase algún error en el proyecto, deberá comunicarlo en el plazo de quince días al Ingeniero Director de obras, pues transcurrido ese plazo será responsable de la exactitud del proyecto.

19. El contratista está obligado a designar una persona responsable que se entenderá con el Ingeniero Director de obras, o con el delegado que éste designe, para todo relacionado con ella. Al ser el Ingeniero Director de obras el que interpreta el proyecto, el contratista deberá consultarle cualquier duda que surja en su realización.

20. Durante la realización de la obra, se girarán visitas de inspección por personal facultativo de la empresa cliente, para hacer las comprobaciones que se crean oportunas. Es obligación del contratista, la conservación de la obra ya ejecutada hasta la recepción de la misma, por lo que el deterioro parcial o total de ella, aunque sea por agentes atmosféricos u otras causas, deberá ser reparado o reconstruido por su cuenta.

21. El contratista, deberá realizar la obra en el plazo mencionado a partir de la fecha del contrato, incurriendo en multa, por retraso de la ejecución siempre que éste no sea debido a causas de fuerza mayor. A la terminación de la obra, se hará una recepción provisional previo reconocimiento y examen por la dirección técnica, el depositario de efectos, el interventor y el jefe de servicio o un representante, estampando su conformidad el contratista.

22. Hecha la recepción provisional, se certificará al contratista el resto de la obra, reservándose la administración el importe de los gastos de conservación de la misma hasta su recepción definitiva y la fianza durante el tiempo señalado como plazo de garantía. La recepción definitiva se hará en las mismas condiciones que la provisional, extendiéndose el acta correspondiente. El Director Técnico propondrá a la Junta Económica la devolución de la fianza al contratista de acuerdo con las condiciones económicas legales establecidas.

23. Las tarifas para la determinación de honorarios, reguladas por orden de la Presidencia del Gobierno el 19 de Octubre de 1961, se aplicarán sobre el denominado en la actualidad “Presupuesto de Ejecución de Contrata” y anteriormente llamado ”Presupuesto de Ejecución Material” que hoy designa otro concepto.

Condiciones particulares

La empresa consultora, que ha desarrollado el presente proyecto, lo entregará a la empresa cliente bajo las condiciones generales ya formuladas, debiendo añadirse las siguientes condiciones particulares:

1. La propiedad intelectual de los procesos descritos y analizados en el presente trabajo, pertenece por entero a la empresa consultora representada por el Ingeniero Director del Proyecto.

2. La empresa consultora se reserva el derecho a la utilización total o parcial de los resultados de la investigación realizada para desarrollar el siguiente proyecto, bien para su publicación o bien para su uso en trabajos o proyectos posteriores, para la misma empresa cliente o para otra.

3. Cualquier tipo de reproducción aparte de las reseñadas en las condiciones generales, bien sea para uso particular de la empresa cliente, o para cualquier otra aplicación, contará con autorización expresa y por escrito del Ingeniero Director del Proyecto, que actuará en representación de la empresa consultora.

4. En la autorización se ha de hacer constar la aplicación a que se destinan sus reproducciones así como su cantidad.

5. En todas las reproducciones se indicará su procedencia, explicitando el nombre del proyecto, nombre del Ingeniero Director y de la empresa consultora.

6. Si el proyecto pasa la etapa de desarrollo, cualquier modificación que se realice sobre él, deberá ser notificada al Ingeniero Director del Proyecto y a criterio de éste, la empresa consultora decidirá aceptar o no la modificación propuesta.

7. Si la modificación se acepta, la empresa consultora se hará responsable al mismo nivel que el proyecto inicial del que resulta el añadirla.

8. Si la modificación no es aceptada, por el contrario, la empresa consultora declinará toda responsabilidad que se derive de la aplicación o influencia de la misma.

9. Si la empresa cliente decide desarrollar industrialmente uno o varios productos en los que resulte parcial o totalmente aplicable el estudio de este proyecto, deberá comunicarlo a la empresa consultora.

10. La empresa consultora no se responsabiliza de los efectos laterales que se puedan producir en el momento en que se utilice la herramienta objeto del presente proyecto para la realización de otras aplicaciones.

11. La empresa consultora tendrá prioridad respecto a otras en la elaboración de los proyectos auxiliares que fuese necesario desarrollar para dicha aplicación industrial, siempre que no haga explícita renuncia a este hecho. En este caso, deberá autorizar expresamente los proyectos presentados por otros.

12. El Ingeniero Director del presente proyecto, será el responsable de la dirección de la aplicación industrial siempre que la empresa consultora lo estime oportuno. En caso contrario, la persona designada deberá contar con la autorización del mismo, quien delegará en él las responsabilidades que ostente.